CodeFeedr

Connecting streaming jobs

J. Kuijpers
J. Quist
W. Zorgdrager

Technische Universiteit Delft

TU Delft

CodeFeedr

Connecting streaming jobs

by

J. Kuijpers
J. Quist
W. Zorgdrager

to obtain the degree of Bachelor of Science in Computer Science
at the Delft University of Technology,
to be defended publicly on Monday July 2, 2018 at 11:00 AM.

Project duration:	April 23, 2018 – July 2, 2018	
Thesis committee:	Dr. ir. T. Abeel,	TU Delft, supervisor
	Dr. ir. G. Gousios,	TU Delft, client
	Dr. ir. H. Wang,	TU Delft, coordinator

TU Delft

Preface

This report has been written to fulfill the requirements of the Bachelor Project TI3806. This project marks the end of the undergraduate Computer Science program at Delft University of Technology. In this project, we created a framework on top of the open-source stream processor Apache Flink. This framework allows for pipelining of Flink stream processing jobs.

We would like to thank a few people which helped us throughout the project. First of all, Georgios Gousios for his dedicated support and mentoring. Secondly, we would like to thank Thomas Abeel for his advisory role especially with regard to the report. Lastly, we would like to thank the people that we did not mention above but who did contribute to the project.

Jos Kuijpers, Joris Quist, Wouter Zorgdrager
Delft, June 2018

Summary

CodeFeedr is a research project at the software engineering division of the Delft University of Technology in collaboration with the Software Improvement Group. The research focuses on a software infrastructure which serves software practitioners in utilizing data-driven decision making [18].

Currently, frameworks like Apache Flink are capable of high-performance data streaming. However, these frameworks have a lot of overhead in setting up, and adding new streaming queries takes a lot of time. They also have several limitations in combining real-time data with historical data and doing aggregations on streams from multiple sources.

The developed product is a plug-in framework on top of Apache Flink, that provides a pipelining system for streaming queries. This product includes abstractions for well-known sources like GitHub, TravisCI and Twitter as well as support for historical data in mongoDB. With this framework the users can spend their efforts on actually writing streaming queries instead of setting up environments, input sources and output destinations. The product also includes orchestration tools for running streaming jobs on a distributed system.

In the future the framework can be extended to include more streaming sources and extra tools can be developed to improve usability. There are also some opportunities to do fine-tuning and fault tolerance testing.

Contents

List of Figures

1 Problem Statement

1.1. Problem Context

In 2016 the CodeFeedr research was founded by our client, Dr. Georgios Gousios. In the original vision [18], this research project is described as "a novel software analytics infrastructure supporting for a combination of three requirements to serve software practitioners in utilizing data-driven decision making". Those three requirements, as defined in [18], are (1) giving real-time insight, (2) offering a query model, and (3) providing data summarization. On the website of CodeFeedr [20] this vision is made more explicit. It describes an architecture in which (real-time) data streams are first processed by a stream processor like Apache Flink [6] to store in a data silo like mongoDB. These streams are then processed by a query engine being able to combine both historical and real-time data. The result of the query is then used to serve a real-time summarization framework. This architecture will be used to provide streaming analytics in the field of software engineering [18].

In our project we will mostly focus on providing the streaming processor functionality for the CodeFeedr research. Moreover, we will develop an architecture in which a user can easily create a stream processing job in the context of CodeFeedr.

1.2. Problem Definition

Streaming systems are designed to perform continuous queries on data streams within a small period of time from receiving the data. Consider a temperature sensor, which sends its temperature every millisecond. A streaming system could query the data and send an alert when the temperature gets below a certain point. This process is called a streaming job.

Powerful streaming frameworks like Apache Flink [6], Apache Spark [8] and Apache Storm [9] already exist. However, while these frameworks are powerful in processing streams they lack the ability to pipeline multiple streaming jobs. Pipelining means that the output of one streaming job is used as input for other streaming jobs. These frameworks do not allow multicasting, only one output sink can be defined per streaming job. In the temperature sensor example the data gets discarded after the alert is sent, but it might be useful to process the output data with a different query. To pipeline streaming jobs efficiently, a message broker like Apache Kafka [7] is necessary. A message broker works as data queue in between multiple jobs and also enables multicasting. One streaming job can write to the queue and multiple other jobs can read from it. However, the use of a message broker will require the user to know how to work with different platforms and integrate all of them. It also results in a lot of boilerplate code and will cost a lot of time which can be better spent on writing actual streaming jobs.

Most streaming frameworks lack functionality to connect with data sources or sinks which are useful in the CodeFeedr context. For instance, Flink offers no functionality to connect with data stores like mongoDB. Using these stores is necessary in a situation where it is needed to (re-)process historical data. In addition,

the CodeFeedr research is meant to develop streaming analytics tools in the field of software engineering. For that purpose, having connectors to API's like the GitHub API and TravisCI API is useful.

Whereas most streaming frameworks offer many ways to run the streaming job in a clustered setup actual configuration files or scripts are not available. Additionally, their documentation is often focused on running one streaming job instead of a pipeline of jobs. This means that developers, once again, lose a lot of time writing boilerplate configuration files for their orchestration.

The problems defined above can be split in roughly 3 sub-problems: pipelining streaming jobs, additional connectors and orchestration tools. Solving these will result in a powerful development kit in which developers can:

1. Easily write and pipeline streaming jobs
2. Use additional (data) connectors
3. Run jobs in a clustered fashion in only a few steps

1.3. Requirements

In this section we will elaborate on the requirements of our product. Using the MoSCoW method we specified and prioritized a set of requirements. In the MoSCoW method there are four levels of priority: must haves, should haves, could haves and won't haves. The must haves contain the requirements that make up the absolute bare minimum that is needed for a product to be functional. Then there are the should haves, which consists of the requirements that are still important, but which without there would still be a working product. Could haves are non-essential requirements which should only be implemented in case extra time is available after implementing the should haves and must haves. There are no explicit won't haves for this project.

- Must Haves
  - The product must support pipelining multiple Flink jobs: use the output of one Flink job as input for multiple other Flink jobs.
  - The product must be easily extensible with plugins, which provide streaming jobs related to a context.
  - The product must include a plugin which allows (re-)processing of historical data.
  - The product must include orchestration tools to deploy pipelines on a cluster.
  - The product must be implemented in Scala.
  - The product must be easily maintained and extended by other developers.
- Should Haves
  - The product should have a knowledge base or guide.
  - The product should have an example project that contains the right build files and a streaming job that is ready to be deployed on a cluster.
  - The product should have a GitHub plugin which streams data using the GitHub API.
  - The product should have a Travis plugin that makes it possible to get builds from the TravisCI API.
  - The product should have API key management to allow users to use API keys in a distributed environment.
  - The product should be easy to use by the end user.
- Could Haves
  - The product could have a Twitter plugin which streams Twitter statuses.
  - The product could have plugins to support simple data sources.

– The product could have schema exposure to expose the data type(s) of a streaming job to an external service.

There are only a few requirements given by the client related to the use of existing technologies. First of all, we need to use the Apache Flink [6] streaming platform. Mainly because this is also used by other contributors of the CodeFeedr project. Inherent to the use of Flink, we will be working with Scala. Flink is both written in Java and Scala but we have chosen Scala. This is because Java code is quite verbose and we wanted to focus on less boilerplate code.

2

Design

In this chapter we will elaborate on the design of the product. Based on the requirements, the design is split into two parts. First, a core framework which is a single Scala library that implements the pipelining system. It also includes utilities like API key management. Secondly, we worked on a set of plugins. A plugin is a simple library that relies on the core framework. Each plugin provides streaming jobs and data connectors related to a certain context.

2.1. Core framework

2.1.1. Pipeline

We define a pipeline as a directed acyclic graph in which a node is a Flink job and edges are queues in a message broker. The nodes in these pipelines are called **stages** and the edges are named **buffers**. The purpose of this pipeline is to flow data from one stage to another using the buffers.

We distinct three type of stages, each one having different interactions with a buffer. All the stages execute a Flink streaming job using the DataStream API[1].

1. Input stage: does not read from a buffer, but writes to one.
2. Transform stage: reads from one or more buffers and also writes to one.
3. Output stage: reads from one or more buffers but does not write to any.

A simple pipeline is shown in Figure 2.1.

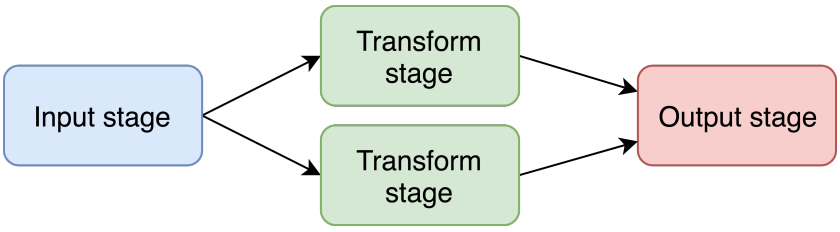

Figure 2.1: A visualization of a simple pipeline consisting of an Input Stage, two Transform Stages and an Output Stage.

Data that is written or read from a buffer needs a type. This means that if a stage is created, the input and output type of that stage should be defined. Only input stages do not have an input type and output stages do not have an output type because they do not interact with a buffer. Besides, for each stage a Flink streaming job is specified. This is a transformation of the input type to the output type using operations of the Flink

[1] https://ci.apache.org/projects/flink/flink-docs-release-1.5/dev/datastream_api.html

DataStream API. See 2.2 for a code example of an input stage. To view code examples of all the different stages see Appendix E.1.

```
case class SimpleData(str: String)

class SimpleInputStage() extends InputStage[SimpleData]() {

  override def main(): DataStream[String] = {
    environment
      .fromCollection(Seq(SimpleData("Simple"), SimpleData("data"), SimpleData("set")))
  }
}
```

Figure 2.2: Simple input stage.

When talking about pipelines, there are two distinct views. First, the view of the data flow where an elements comes in from a source (for example an RSS feed, a file or a REST endpoint) and exits the pipeline at a sink (for example Elastic Search or a console). Secondly, the view of the underlying setup of the software and the communication between the services. Even though each stage has a buffer to another stage, all those buffers exist on a message broker. Each stage runs in Flink and communicates with the message broker. See Figure 2.3.

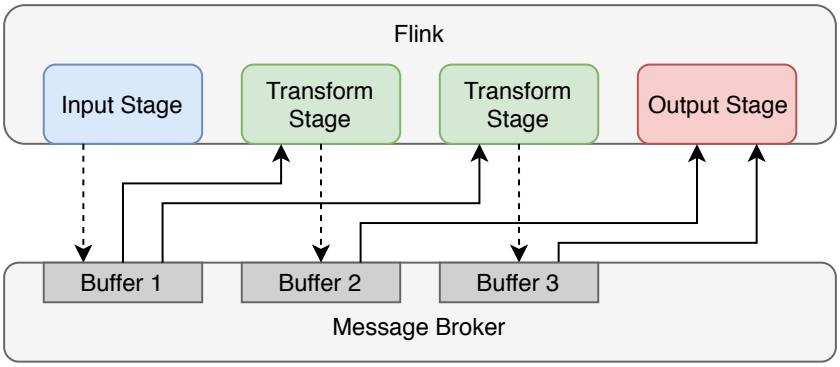

Figure 2.3: A visualization of the same pipeline as in figure 2.1, but now as a system. Dashed arrows represent data being written to the buffer and the solid arrows represent data being read from the buffer.

2.1.2. Buffers

A buffer is a short-term storage that can be written to and read from by stages. A buffer implementation has to meet some requirements to work. To enable multicasting the buffer should allow multiple subscribers to read the contents of the buffer independently. To maintain the guarantees that Flink gives, the buffer system should also be low latency, fault tolerant and scalable. If there is no Flink source and sink available it needs to be implemented.

Internally, the Flink source and sink of a buffer are wrapped around the Flink streaming job defined in a stage. Currently there are two buffer implementations: Apache Kafka [7] and RabbitMQ [3]. A pipeline can only be configured to use one buffer implementation. Figure 2.4 shows how each stage interacts with a (Kafka) buffer implementation. For example the TransformStage reads from a Kafka source, executes a Flink job and then writes to a Kafka sink.

Data inside a buffer is stored in binary format. This means that the data needs to be (de)serialized after reading or before writing. Depending on the application different types of serialization might be preferred. If the buffer will be read by external software, a JSON format might be favored over a highly-optimized binary representation. The core framework supports a set of configurable serializers. These include JSON, Kryo and BSON. JSON is a human-readable format but serialization is slow. Kryo has the best compression and

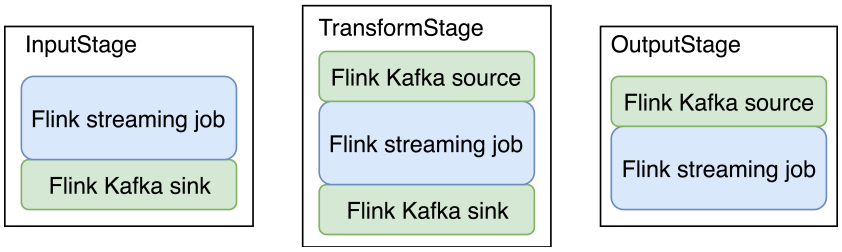

Figure 2.4: Interactions of stages with (Kafka) buffer.

performance but provides poor readability. BSON is in between those two, it is faster than JSON but less readable. Only one type of serialization can be configured per pipeline.

Buffers are essentially stand-alone message brokers which make the pipelines powerful. Buffers scale and in combination with Flinks scaling features[2] the whole pipeline is scalable. Additionally, if a stage crashes no data is lost. Stages that generate input for a crashed stage still continue to do so by just writing to the buffer. Data that was already processed by the crashed stage is either written to a buffer or a different output. Data still in memory at the moment of the crash can be restored by Flinks checkpointing mechanism [3]. If a crashed stage is restarted it will proceed processing where it left off and no data is missed.

2.1.3. Pipeline builder

A pipeline is created using the pipeline builder. This builder ensures pipelines to be immutable and assures there are no synchronization issues between Flink jobs and stages. Besides creating the pipeline, the builder is also used to configure properties of the pipeline. These properties are for example the serializer or buffer type. The easiest pipeline is a sequential one, with no joins or splits in the graph. A code example of building such a sequential pipeline can be seen in figure 2.5. To create a more complex pipeline, edges can be added from one stage to another. Any graph can be created this way as long as it is acyclic. The builder disallows adding edges that cause because this would allow the stream size to increase infinitely. A code example of building such a pipeline can be seen in figure 2.6.

```
new PipelineBuilder()
    .append(new SimpleInputStage)
    .append(new SimpleTransformStage)
    .append(new SimpleOutputStage)
    .build()
```

Figure 2.5: Simple pipeline. To view the complete code example see Appendix E.2.1

```
val inputStage = new SimpleInputStage
val outputStage = new SimpleOutputStage
val transformStage = new SimpleTransformStage
val otherTransformStage = new SimpleOtherTransformStage

new PipelineBuilder()
   .edge(inputStage, transformStage)
   .edge(inputStage, otherTransformStage)
   .edge(transformStage, outputStage)
   .edge(otherTransformStage, outputStage)
   .build()
```

Figure 2.6: Complex pipeline. To view the complete code example see Appendix E.2.2

[2] https://ci.apache.org/projects/flink/flink-docs-release-1.5/dev/parallel.html
[3] https://ci.apache.org/projects/flink/flink-docs-release-1.5/dev/stream/state/checkpointing.html

A pipeline can be started in three modes: mock, local and clustered. The mode is specified as argument when running an application with a built pipeline. In mock mode each stage in the pipeline is linked without buffers in between. This is mainly for testing purposes and only supports sequential pipelines. In local mode all stages are started on the same thread in one Flink execution environment. In clustered mode, only one stage will be started. Although stages rely on each other in terms of data flow, they can all be run as stand-alone Flink jobs because of the (stand-alone) buffers in between. Therefore, this mode allows to add, remove or modify stages already running in a pipeline. When running in clustered mode, the stage id needs to be specified.

2.1.4. Utilities

The core framework also provides utilities which can be used by stages in a pipeline. These utilities include key management and schema exposure.

Key Manager When retrieving data via an API a rate limit is almost always used by the provider to prevent abuse. This means the client can only do a certain amount of calls in a certain time. Usually they require a key to be sent with each request so that they can identify who is making the request and check if they are allowed to make the call. In a streaming setup it might be necessary to do more calls than the rate limit of one key provides. In that situation, multiple keys needs to be shared and managed in a distributed environment. This requires a centralized store accessible by all the stages. We provide key managers which can either store in Redis or in mongoDB. Only one type of key manager can be defined per pipeline and each stage has access to this manager.

Schema Exposure By default no centralized information is available for the datatypes in the buffers. This means that, apart from stages that read and write to that particular buffer, no one knows what type of data is stored there. Therefore, a schema exposer can be configured for a pipeline. This exposer sends the data type in the form of an Avro schema to a centralized store. This allows external parties to easily identify the types of data per buffer. The schema exposer can either be configured to expose to Redis or Zookeeper.

2.2. Plugins

A plugin is a library which relies on the core framework and contains stages related to a certain context. Plugins have individual dependencies and are kept out of core to lower the amount of bloat for simple applications.

An example of a plugin could be a news plugin which gets a stream of all news articles from a website. A stage in the news plugin could retrieve those articles to turn it into a Flink stream. A second stage reads from those articles and summarizes the content. A third stage also reads from those articles and filters on the popular ones. These stages could be used separately or connected to create a pipeline. Custom stages can be put in between as well, giving high flexibility.

In Chapter 3 a more detailed description of the plugins is given.

3

Final product

According to our design the product consists of two separate parts: the core, and the plugins. This chapter will elaborate on the final design of those two parts as well as the way we documented and tested them. It will also explain the orchestration tools created for running pipelines on a remote computer or cluster of computers.

3.1. Core

The core framework is mostly in line with the design that was created beforehand. Some small things were improved or changed throughout the project, such as configurability and the serialization options. In Appendix F we describe some of the issues we have faced when implementing our design and ideas.

It appeared to be easy to add a new buffer into the core framework, as we did with RabbitMQ. It was just as easy to add new serializers, as both BSON and Kryo were added in the later stages of the project.

3.2. Plugins

We have implemented a large group of plugins for the core framework, differing in complexity and context. These plugins can directly be used by an application or be used as demonstration. In the subsection below each plugin will be discussed briefly.

3.2.1. RSS and weblogs

The first two plugins that were created are proof of concepts. RSS polls from a feed and filters duplicates and then turns that into a stream. The weblogs plugin consists of a basic Flink job that turns a log into a finite stream. They were made to demonstrate how plugins fit into the framework.

3.2.2. MongoDB

MongoDB is a persistent document store. The plugin provides an input and output stage to write stream elements to a collection and to create a stream filled with elements from a collection. It automatically propagates stream event time by writing it to a special document key. It also supports streaming with a filter or starting from a certain time. The plugin also adds support for using mongoDB as key manager. It uses a single document per key and automatically updates keys of a target when a new key is requested.

3.2.3. ElasticSearch

The ElasticSearch plugin provides a simple output stage that wraps the ElasticSearchSink provided by the Flink project. It adds a default server at localhost for the simplest cases. It simply serializes an input stream element to JSON and sends it to an Elastic Search index.

3.2.4. GitHub

The GitHub plugin provides streaming jobs around the GitHub Events API [1]. The 'main' stage in this plugin creates a real-time stream of GitHub events. The other stages revolve around this event stream by doing some basic filtering. For example, one stage is able to transform an event stream into a stream of only push events.

3.2.5. TravisCI

The Travis plugin makes it possible to get a stream of builds on TravisCI from a stream of push events from Github. The Travis plugin includes two transformation stages. One that filters push events to only keep the ones that are from repositories that are active on Travis. The other stage takes these push events and requests the build information from Travis.

3.2.6. Twitter

The Twitter plugin provides some simple input stages for the Twitter API[2]. This plugin makes use of an external library, twitter4s[3], which handles all the connections with the Twitter API. The first input stage streams twitter statuses based on a set of filters. These filters include locations, keywords, languages and specific users. Another input stage streams twitter statuses based on the current trending topics. It refreshes the trending topics after a configurable time.

3.3. Orchestration

The pipelines created by our framework should easily be deployed on remote servers or even clusters. To support this, we created a sample configuration for Docker. This configuration consists of all services necessary to run a whole pipeline. This includes Flink, a message broker and optional services like mongoDB. A single Docker command deploys all those services and makes it ready to use. The use of Docker allows for easy scaling of all the separate services.

To start a pipeline, every stage needs to run as a Flink job. Therefore, we created a simple Python script which manages and deploys pipelines on a cluster. To use this script, the user has to assemble its pipeline application to a .jar file (a sample application can be found in Appendix E.2.1). Using this jar the script is able to start, stop and scale pipelines or stages. Furthermore, it can also give an overview of already running pipelines.

To make it easy to use both our core framework and plugins we created a project template[4]. This template allows a user to create their own pipeline with their preferred plugins. The template also includes the orchestration tools to run the created pipeline on a cluster. Using the default Scala build tools this template project can be generated.

With the help of the template it is possible to create and deploy a pipeline using the following steps:

1. Generate project from template
2. Import plugins (optional)
3. Create custom stages (optional)

[1] `https://developer.github.com/v3/activity/events/`
[2] https://developer.twitter.com/en/docs.html
[3] https://github.com/DanielaSfregola/twitter4s
[4] `https://github.com/Jorisq/bep_codefeedr-project.g8`

4. Create pipeline
5. Assemble project into .jar
6. Setup cluster using provided Docker configuration
7. Deploy pipeline using provided Python script

3.4. Testing

Testing is vital for creating a reliable and maintainable software product. Naturally, we unit tested almost all our code. Furthermore, we also added integration tests for most parts of the software.

Pipelines were tested using integration tests, but due to the use of buffers there was a lot of overhead in running these tests. This additional overhead is not necessary if the pipeline in question is a sequential pipeline. Therefore we added a mock run mode that does not add buffers in between stages, but instead just connects all stages directly to each other. This improved the speed of the integration tests.

Input stages usually make use of external dependencies, for instance a GitHubEventsInput stage reads events from the GitHub API. When testing, external dependencies are undesired because the result of the test would not be deterministic. Besides, a test could fail because a request timed out. To solve this, we mocked the external dependencies in our test setup. This both improved the speed as well as the consistency of the test. However, the pitfall of this method is that the mocked responses should correctly correspond to all the potential responses of the external dependency.

Streaming jobs are normally not meant to run for a short amount of time. In essence, streaming jobs are unbounded and run forever. For this reason there is not really a clean way to gracefully stop Flink jobs. In a testing scenario this is disadvantageous. Therefore, we created some testing utils to force a streaming job shutdown and collecting the results.

3.4.1. Continuous Integration

To improve and standardize our work-flow we use the continuous integration system Travis CI. This automatically builds and runs the tests every time a new update is pushed to a code branch. We used pull-based development and these builds made it possible to add extra constraints to a pull request to be able to merge. The pull request had to be reviewed by at least on team member and the build had to pass in the TravisCI environment.

3.4.2. Coverage Tracking

Using the automated building and testing from Travis CI we also added automated test coverage for pull requests. This allowed us to check if the changes that were made in the branch were thoroughly tested. This allowed us to add the constraint that a pull request can only be merged when to overall test coverage is at most 1% lower than originally. Furthermore, per pull request it gave a coverage report of the changes in that branch. After creating coverage for the initial project, we never got below 94% line coverage, according to Coveralls.

3.5. Documentation

Documentation is an important step to make a framework accessible for new users. It is generally the first thing to read. We decided to use the documentation to both describe how the framework works and how it can be used. That also includes information on each plugin.

The documentation was first written on the GitHub wiki but a special documentation service was found that gave nicer documentation. It also offered an option to put the documentation within the repository, putting it in alignment with the changes being made in the actual code. It can now be found on `https://codefeedr.readthedocs.io`. In Figure 3.1 a screenshot of the documentation is shown.

On top of the documentation we also supplied Scaladoc comments to all public interfaces. This is ideal for looking up functions and works great in combination with an integrated development environment.

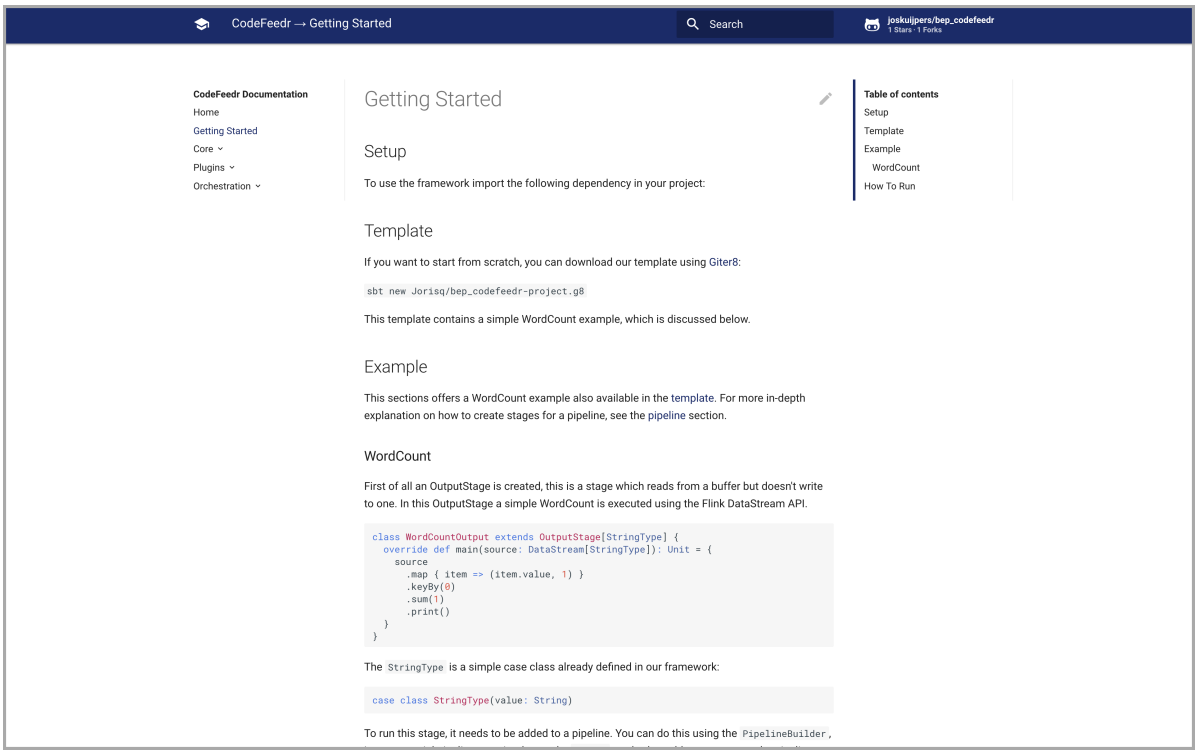

Figure 3.1: A screenshot of one of the pages of the Codefeedr documentation website.

4

Evaluation

This chapter will be used to evaluate the different aspects of the entire project. This includes the final product, the software and management process and the ethical implications. At the end of the chapter some recommendations are made about how the client can follow up on this project.

4.1. Product

With the examples in Appendix E.3 and the tests we have written we have shown that is is now very easy to write a more complex stream processor with just a few lines of code. It is now possible to attach multiple streaming jobs to each other and pipeline events with ease. The project has a set of plugins implemented with interesting data sources and has support for reading and writing to a long term storage, usable for historical data, testing, and creating data copies. Lastly, tools have been provided to run a pipeline on a cluster or powerful computer.

All of the functional requirements as stated in Chapter 1 have been implemented in the final product. The first prototype already consisted of most 'must-haves'. We tried to rapidly implement most of the requirements still keeping the quality in mind. Therefore, we had more time for optimization in the end.

It is hard to measure if we met the non-functional requirements like documentation and ease-of-use. However, we tried to be critical and set standards. Besides, we discussed these requirements also with our client who gave feedback on them throughout the project. This helped us to judge if, for instance, our code was indeed maintainable and sufficiently documented.

We have implemented a couple of plugins that were not set in the requirements, such as Apache httpd log parsing and ElasticSearch. These implementations were mainly as a proof-of-concept for the plugin system.

4.2. Process

The client gave us quite some freedom in decision making. This allowed us to have a lot of influence in the final design of the product. We did not have major development struggles nor did we have issues within the team. We enjoyed working together and were motivated to make this project a success.

We had regular contact with our client, which included a meeting every two weeks and contact on Slack or GitHub as necessary. In those meetings we showed our progress, discussed issues and exchanged feedback. With our supervisor we had two official meetings, one at the beginning of the project and one midterm meeting. In the first meeting we discussed the project and he explained some requirements. At the midterm meeting, we demoed our product and got some feedback. Additionally, the supervisor gave feedback on the report a few times.

4.2.1. SCRUM

During our project we made use of the SCRUM methodology to have an agile framework for managing our work. We always worked together at the university so there was little communication overhead. Therefore, it was easy to have daily SCRUM meetings and tackle issues instantly. We set our sprint length to one week and evaluated this every Monday. With the help of GitHub projects we organized the tasks and assigned them in a SCRUM board. We also used this to set our milestones according to our planning. We did not have specific roles in our team. Everyone took a fair share in writing code, documentation, tests and the report.

We made extensive use of the issues system of GitHub. We created some useful labels to categorize our issues according to priority, feature and type. Each time we came across a bug, a new feature, an improvement or just a simple note we created a new issue and assigned it to one of us. We integrated this with our SCRUM board, so that we could work on it in the current or a next sprint. This was a perfect way to keep track of all our tasks, but also to keep the client informed about our current progress. Regularly, the client commented on one of the issues to give some feedback.

To ensure the quality of the code we used pull-request driven development. Each feature, improvement or bug fix was implemented on a separate branch. Build and coverage checks had to pass before this could be merged. Besides, we used a simple checklist to ensure the branch was properly documented. Using pull-requests is in line with the SCRUM paradigm, because at all times there is a working product on the master branch. Each pull-request improves this product.

4.2.2. Planning

Beforehand we made a rough planing in the form of a roadmap. This helped us plan our sprints and set our milestones. This roadmap can be found in Appendix C.

In the first two weeks we worked on the research and designed the core framework prototype. In the two weeks after we completely fleshed out the framework, spending a lot of hours on making it work as fast as possible so we had a working product to show at our midterm meeting with our supervisor in week 5. This goal was achieved and gave us some flexibility in the last weeks of the project.

This also meant that we ended up having more time than we needed to implement all the features we wanted. The core framework was mostly done in week five, together with some basic plugins. The orchestration was finished in a couple of days. The real time-saver was in the historic data: we planned two weeks for this as it was a very important requirement but it ended up being just an input and output stage. This only took three man-days, allowing us to focus more on fine-tuning details of the code and improving documentation. We ended up having enough time to focus on the report, fix small issues and work out some improvements.

4.3. Ethical Implications

The essence of the project is a development kit that provides tools that make it easier to do more complex stream processing. From this system itself, no ethical issues arise. However, stream processing systems can be used for processing immense amount of data. As with other fields within big data, the ethical implications depend on what kind of data is being used as well as what kind of information is being extracted from this data. When processing data from sensors of the machines in a factory to monitor performance the ethical considerations are not as important as when processing sensitive user data. An example of this is Cambridge Analytica, they collected the Facebook profiles of 50 million Americans without permission to create targeted political advertisements[1]. So the ethical implications depend mainly on what the end user will use the framework for. Research has been done into how ethical theories can be applied to big data processing[21]. If a user is in doubt about whether or not the data processing that they are doing is considered ethically wrong, they can use these theories to verify this.

It is however important to distinct that this project created a framework and does not provide any tools for mining non-public data. For example, GitHub data is already free to access, and the parts that are not (private repositories) can't be accessed with the GitHub API used. The only possible sensitive information that can be collected from this API are email addresses from the authors of commits. These addresses are already

publicly available on the website of Github, but this plugin would allow users to mine the email addresses of all developers that commit to public repositories. In theory we could remove this field from the data that is received from Github, but this would not help that much because the framework is open source. Anyone who really wants to collect the email addresses could just fork the project and change the code. The same holds for the other plugins.

So in short, the framework enables the user to process large amounts of data, but it has no control over what data is being processed or how it is processed. Therefore the user has the responsibility to think about the ethical consequences of the way they use the framework.

4.4. Recommendations

All requirements that were set at the beginning of the project are implemented. However, there are still recommendations we can make with regard to this product as well as the CodeFeedr research.

The project could use some fault tolerance testing and bench marking. We mainly focused on creating the framework and not explicitly on performance. On top of that, long term testing would be useful; does a pipeline survive for a month of normal production traffic?

We suggest two improvements for the core framework: making external buffers possible, and make the schema expose format configurable. Now the buffers are all have to be implemented inside the core, but ideally they could be in their own plugins. Schema exposure is currently always using Avro schemas. Making this configurable to allow the use of, for example, JSON schemas, would be a useful addition. Additionally, we recommend to extend the current plugins and create new ones as well. We implemented a few plugins, mainly to show the capabilities of the framework. These plugins could use some fine tuning and additions to make them more powerful.

The orchestration tools of CodeFeedr could use an interactive web interface, allowing management of a pipeline, with visual information such as the stages and their connections. This could be a great improvement on the barebones script tool. Adding Kafka configuration and other tools could be helpful as well. This would be perfect for a student project. The CodeFeedr research talks about a REPL system on top of the system we have created. We think there are a few difficulties into creating such a system, especially when attempting to generate, compile and assembly Scala code on the fly. Creating a useful, helpful interface is also never easy. This on itself could be another project.

A

Project Description

Currently, frameworks like Apache Flink and Kafka are capable of high-performance data streaming. However, these framework have a lot of overhead in setting up and adding new streaming queries takes a lot of time. These frameworks also have some limitations in combining real-time data with historical data and doing aggregations on streams from multiple sources.

In the context of CodeFeedr, the goal of the proposed project is to create a simple plug-in infrastructure on top of Apache Flink and Kafka, that removes this overhead and makes it possible to combine real-time and historical data as well as the aggregation of multiple data sources. This includes making some abstraction for well-known sources like GitHub, TravisCI and Twitter. This way the user can spend its efforts on actually writing streaming queries instead setting up environments, input sources and output.

Some use cases might be:

1. Plot the amount of issues opened per hour for your whole (GitHub) organization in real time.
2. Plot the files that are frequently updated the last week within a repository.
3. Output the user which opened the most pull request last month.
4. Plot the average time between the opening and closing of an issue.

B

Research report

In our initial research phase we tried to answer a set of questions to help us find what problem we are trying to solve, what is possible, what isn't, and what we should look out for when designing our product. These questions are based off the project description and first talk with the client.

1. What is the problem we are trying to solve?
2. Are there any existing solutions, and if so, how do they work?
3. What are best practices and guides on making a good framework?
4. How does the streaming ecosystem look; are there systems similair to Flink and Kafka?
5. What services exist that can be used for solutions with our project?
6. How can a set of software be managed on a (set of) remote server(s)?

B.1. Problem Statement

B.1.1. Problem Context

In 2016 the CodeFeedr research was founded by our client, Dr. Georgios Gousios. In the original vision [18], this research project is described as "a novel software analytics infrastructure supporting for a combination of three requirements to serve software practitioners in utilizing data-driven decision making". Those three requirements, as defined in [18], are (1) giving real-time insight, (2) offering a query model, and (3) providing data summarization. On the website of CodeFeedr [20] this vision is made more explicit. It describes an architecture in which (real-time) data streams are first processed by a stream processor like Apache Flink [6] to store in a data silo like mongoDB. These streams are then processed by a query engine being able to combine both historical and real-time data. The result of the query is then used to serve a real-time summarization framework. This architecture will be used to provide streaming analytics in the field of software engineering [18].

In our project we will mostly focus on providing the streaming processor functionality for the CodeFeedr research. Moreover, we will develop an architecture in which a user can easily create a stream processing job in the context of CodeFeedr.

B.1.2. Problem Definition

Streaming systems are designed to perform continuous queries on data streams within a small period of time from receiving the data. Consider a temperature sensor, which sends its temperature every millisecond. A streaming system could query the data and send an alert when the temperature gets below a certain point. This process is called a streaming job.

Powerful streaming frameworks like Apache Flink [6], Apache Spark [8] and Apache Storm [9] already exist. However, while these frameworks are powerful in processing streams they lack the ability to pipeline multiple streaming jobs. Pipelining means that the output of one streaming job is used as input for other streaming jobs. These frameworks do not allow multicasting, only one output sink can be defined per streaming job. In the temperature sensor example the data gets discarded after the alert is sent, but it might be useful to process the output data with a different query. To pipeline streaming jobs efficiently, a message broker like Apache Kafka [7] is necessary. A message broker works as data queue in between multiple jobs and also enables multicasting. One streaming job can write to the queue and multiple other jobs can read from it. However, the use of a message broker will require the user to know how to work with different platforms and integrate all of them. It also results in a lot of boilerplate code and will cost a lot of time which can be better spent on writing actual streaming jobs.

Most streaming frameworks lack functionality to connect with data sources or sinks which are useful in the CodeFeedr context. For instance, Flink offers no functionality to connect with data stores like mongoDB. Using these stores is necessary in a situation where it is needed to (re-)process historical data. In addition, the CodeFeedr research is meant to develop streaming analytics tools in the field of software engineering. For that purpose, having connectors to API's like the GitHub API and TravisCI API is useful.

Whereas most streaming frameworks offer many ways to run the streaming job in a clustered setup actual configuration files or scripts are not available. Additionally, their documentation is often focused on running one streaming job instead of a pipeline of jobs. This means that developers, once again, lose a lot of time writing boilerplate configuration files for their orchestration.

The problems defined above can be split in roughly 3 sub-problems: pipelining streaming jobs, additional connectors and orchestration tools. Solving these will result in a powerful development kit in which developers can:

1. Easily write and pipeline streaming jobs
2. Use additional (data) connectors
3. Run jobs in a clustered fashion in only a few steps

B.1.3. Requirements

In this section we will elaborate on the requirements of our product. Using the MoSCoW method we specified and prioritized a set of requirements. In the MoSCoW method there are four levels of priority: must haves, should haves, could haves and won't haves. The must haves contain the requirements that make up the absolute bare minimum that is needed for a product to be functional. Then there are the should haves, which consists of the requirements that are still important, but which without there would still be a working product. Could haves are non-essential requirements which should only be implemented in case extra time is available after implementing the should haves and must haves. There are no explicit won't haves for this project.

- Must Haves
  - The product must support pipelining multiple Flink jobs: use the output of one Flink job as input for multiple other Flink jobs.
  - The product must be easily extensible with plugins, which provide streaming jobs related to a context.
  - The product must include a plugin which allows (re-)processing of historical data.
  - The product must include orchestration tools to deploy pipelines on a cluster.
  - The product must be implemented in Scala.
  - The product must be easily maintained and extended by other developers.
- Should Haves
  - The product should have a knowledge base or guide.

  - The product should have an example project that contains the right build files and a streaming job that is ready to be deployed on a cluster.
  - The product should have a GitHub plugin which streams data using the GitHub API.
  - The product should have a Travis plugin that makes it possible to get builds from the TravisCI API.
  - The product should have API key management to allow users to use API keys in a distributed environment.
  - The product should be easy to use by the end user.
- Could Haves
  - The product could have a Twitter plugin which streams Twitter statuses.
  - The product could have plugins to support simple data sources.
  - The product could have schema exposure to expose the data type(s) of a streaming job to an external service.

There are only a few requirements given by the client related to the use of existing technologies. First of all, we need to use the Apache Flink [6] streaming platform. Mainly because this is also used by other contributors of the CodeFeedr project. Inherent to the use of Flink, we will be working with Scala. Flink is both written in Java and Scala but we have chosen Scala. This is because Java code is quite verbose and we wanted to focus on less boilerplate code.

B.1.4. Scala

The Scala programming language allows for many constructs, including any known for the Java ecosystem, but also from functional programming. Scala also allows the creation of new operators and has a lot of syntactic sugar to give way to new notations of intent [19].

The trait system of Scala allows for maximum code reuse. Interfaces can be defined as traits to make them re-implementable by the user of the framework. When subclassing is allowed, custom behaviour is possible. After all, not everything can be thought of when building such framework. Scala also implements type aliasing. This is very useful for giving new names to types of external packages. The user of the framework would not need to import those external packages anymore, or even know what package the type originated from [26].

On top of all of this, Scala allows various ways to give information to the type checker, eliminating the need to ever use a non-specific type. These techniques include its generics, traits, case classes and implicits [27].

B.1.5. Documentation

The framework will be used by other developers, therefore it needs proper documentation. Both to inform the users about its usage, and to help new and existing developers to understand the internals of the system. For the former, a quick start guide and code examples are very helpful. For both an API documentation is very useful as well. For the latter, design documents come in handy, especially if it documents the choices made to come to the current design [14].

On top of documentation that can be written separately from the code, Scala had API documentation support using Scaladoc. The Scala developers describe an exhausting style guide at [28]. Scaladoc also supports writing extra formatted pieces of text with packages, making it possible to put nearly all documentation inside the code. This has as big advantage that there is a single source of truth, and only a single documentation generator (in this case Scaladoc).

B.1.6. Testing

The programming language for this project, Scala, has an excellent testing framework named ScalaTest. It has eight different test styles to choose from, fitting any project and team [11]. It even allows testing of pri-

vate methods [12]. ScalaTest is an extendable system. New testing systems can be added on top of the core elements of ScalaTest to fulfill special testing requirements. This might be useful when working with way streams work. ScalaTest also has integration with many development tools and Scala related systems. There does not seem to be any other usable testing framework for Scala.

Automated builds and testing using TravisCI is ideal: it detects when builds start to break and can also automatically create artifacts such as test reports, coverage reports and product files. It can also give feedback on the tests during all stages of the product lifetime and code life-cycle.

Code coverage analysis tools detect the code that is run when running all tests, computing what code is tested and what code is not. It is not a complete analysis but it is close. For Scala a single coverage library exists, named *scoverage*. It is a plugin for the Scala compiler. It has integration into TravisCI and coverage visualizers as well.

B.2. Stream Processing

The project will mostly revolve around a stream processor. As described in [29], stream processing is getting more and more popular these days, because processing big volumes of data on itself is not enough. Data has to be processed fast, so that decisions can be made in near real-time. A stream processor analyzes real-time data making use of continuous queries. Stream processing frameworks are often designed to handle big volumes of data, be scalable, fault tolerant and highly available [29]. As required by our client, we will be using Apache Flink [6] as stream processing framework. In Section B.2.1 the features of Apache Flink will be discussed briefly. In Section B.2.2 some alternative stream processing frameworks are mentioned and we discuss the reasons why these are less optimal for this project.

B.2.1. Apache Flink

Apache Flink[13] is an open-source large-scale streaming analytics platform. It offers functionality of processing both streams and batches, which is necessary if we want to support historical (batch) data. Flink is also scalable, because it runs on clusters. If more processing power is needed, it is always possible to add new computers to the cluster. It also has an easy to use programming interface in both Scala and Java using the functional programming paradigm. Lastly, Flink is part of the open-source data processing ecosystem. This allows easy integration with frameworks like Apache Kafka [7]. More information and detailed descriptions of Flink features can be found on their project website [1].

B.2.2. Alternatives

There are also other popular streaming frameworks like Spark Streaming [8] and Apache Storm [9], but these are not as powerful as Flink or have other disadvantages. For instance, Spark Streaming provides a high-throughput using micro-batches, which comes at cost of introducing high latency. However, Storm provides low latency but does not offer high-throughput, and does not support restoring state after failures [17]. In Figure B.1 a visualization of this comparison is made showing the capabilities of the three frameworks.

Whereas the other stream processors only give a exactly-once guarantee, Flink is able to offer both exactly-once and at-least-once guarantees. These guarantees can be given because it makes use of a checkpoint and replay mechanism [16].

B.3. Message Brokers

In order to combine multiple streaming jobs the framework should support that output of stream jobs can be used by multiple other stream processing jobs as input. To implement this in a scalable manner, a distributed message broker is necessary. A message broker can be used to decouple two stream processing jobs by using a publish-subscribe pattern. The first stream job can just publish its output without needing to know who or how many are going to read it and subsequent jobs in the pipeline can subscribe to the output and the message broker will forward messages to all subscribers. It is not necessary for use to stick to one message broker,

[1] http://flink.apache.org/features.html

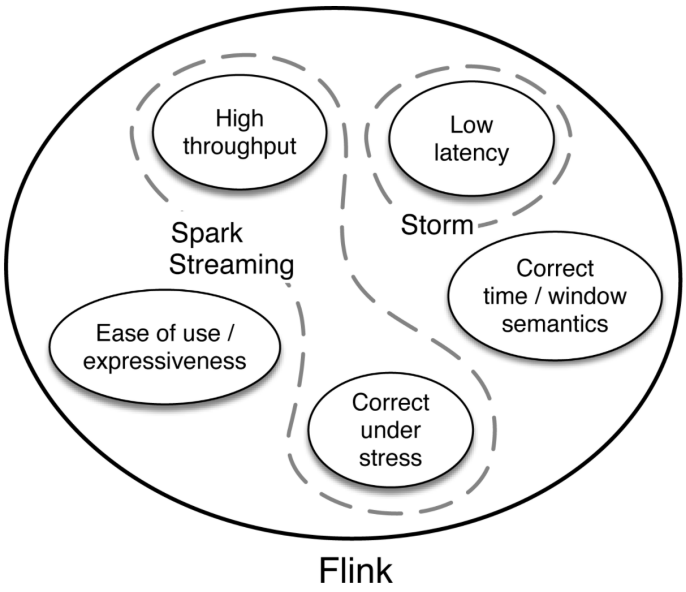

Figure B.1: A visualization of Flink compared to Spark and Storm as given in [17].

since we can design our framework in such a way that the broker between streaming jobs is configurable. However, the rest of this section will focus on two popular message brokers; Kafka and RabbitMQ [30].

B.3.1. Kafka

There are several reasons why Kafka is a good choice for our project. Flink has built-in support for Kafka, so that makes it less work to implement them together. Its messaging system has very little overhead which results in a very high througput compared to its competitors [25]. Kafka also stores the records in a fault-tolerant way, so even in the event of a crash no messages are lost. Lastly it both supports exactly once guarantees and transactional messaging. These last three points make it possible for us to offer the exactly once guarantee throughout the whole pipeline.

Kafka has two APIs that are useful for our framework. The Producer API and the Consumer API. In figure B.2 you can see these APIs visualized.

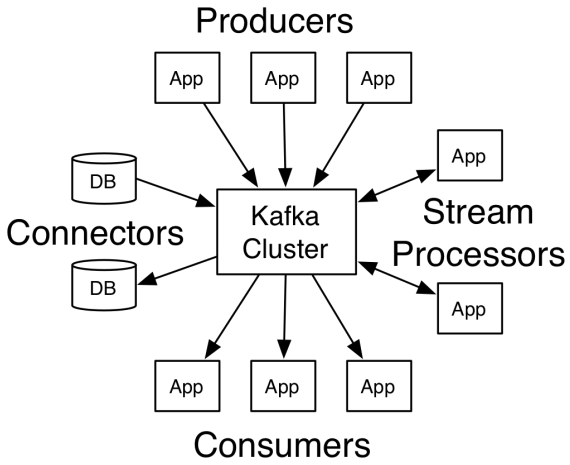

Figure B.2: A visualization of the Kafka APIs[7].

We want to use Kafka as a buffer in between streaming jobs. The first streaming job in the pipeline will get a

stream from a certain source (e.g. Github, Travis, etc) and use the Producer API to push the stream to a Kafka topic. Subsequent jobs will use both the Producer API and the Consumer API to pull from a certain topic and then push to a new topic and the last object will only consume a stream and output to a log or some other external output.

Doing this enables the user to reuse the output of a streaming job for multiple other streaming jobs. This is even possible for streaming jobs that were already running or jobs that haven't even started yet.

Another reason to choose for Kafka is that it is scalable. We want our framework to be scalable, so if the messaging system in between two streaming jobs would not be scalable than that would become a bottleneck for the rest of the pipeline. This would mean that the entire framework would not be scalable anymore.

B.3.2. RabbitMQ

RabbitMQ is a more matured message broker compared to Kafka. It is a generic message broker with 'queues', as opposed to the 'topics' in Kafka [3]. It has connectors for many languages so integration with other software is quite easy.

RabbitMQ has built in support for high availability and is easier to set up than Apache Kafka together with Zookeeper. It has built in mechanisms for clustered running and federation. It's througput is however lower than that of Kafka [22].

B.3.3. Conclusion

Both Kafka and RabbitMQ have their strength and weaknesses, picking one depends on the context and use case of a user's application. Kafka is more appropriate for streaming events, stream processing and stream history. It is however much harder to set up correctly than RabbitMQ [22].

We plan to design our framework in such a way that the message broker is configurable. Next to that, adding a new message broker should be easy so that users are not limited to either Kafka or RabbitMQ.

B.4. Serialization

Our goal is to pipeline multiple streaming jobs using a message broker, which means we need a serialization framework in between the broker and the streaming job. In the stream processing jobs we are limited to the data types of Flink, which in the case of Scala revolves mostly around case-classes. There are a lot of serialization frameworks out there, like Jackson (JSON) [15], Avro [5], Thrift [4], Protobuf [2] and Kryo. However, we are bound to the implementations of these frameworks in Scala which means that we can not use some of those. Since we are working with big data we judge these frameworks based on speed and compression. Next to that, we should consider their ease-of-use since we also envision that for our framework.

In this article [24] many of those Scala serialization frameworks are compared based on the criteria mentioned above. In Figure B.3 and B.4 the results of some of those benchmarks are shown. These benchmarks show that, apart from Jackson, these frameworks achieve quite similar results. BooPickle appears to be the fastest, however it does not support backwards compatibility [24]. BooPickle is followed by ProtoBuf and Thrift, but those two frameworks require user's to define their data schemas in advance and then generate a (case) class from that in Scala. This is a huge drawback, if we want our framework to be easy to use. To conclude, same as for the message broker we will probably support multiple serialization frameworks. Next to that, our framework should be easily extended with a new serializer.

B.5. Key Manager

When creating an automated script to get data via an API a rate limit is almost always used by the provider. That means the client can only do n calls per time unit (*interval*). To be able to track which keys are available and how much calls are left on each of them, they need to be stored somewhere. For a standard non-distributed application this could simple be done with some sort of map. However since this framework is build for distributed applications a different solution is needed.

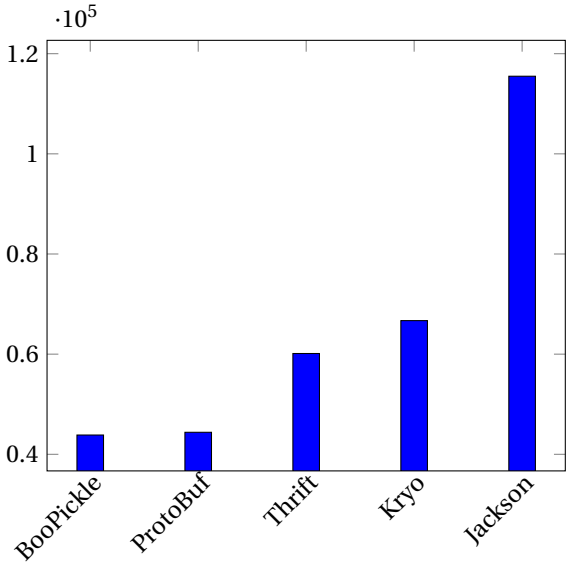

Figure B.3: Speed comparison of two-way serialization of a 8kb record (in nanos)

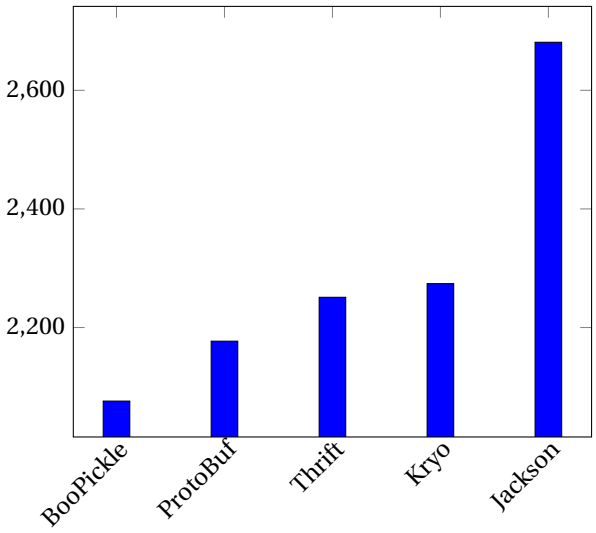

Figure B.4: Compression comparison of a 8kb record (in kb)

Putting the keys in a central places solves this problem. Basically, a simple list of keys on a key-value store could do. However, there are more requirements: Keys have rate limits and those limits reset after a while. This has to be handled as well, and preferably atomically: there should be no inconsistency between the data used by one client and the data used by another. Such conditions make it very hard to reason about the data and also very hard to find issues, especially when there are race conditions.

We look at two pieces of software that can provide storage for such key manager, with their advantages and disadvantages; ZooKeeper and Redis.

B.5.1. ZooKeeper

Zookeeper is a centralized service for keeping configuration data and other key-based values. A great advantage of Zookeeper is that it is already supported by other elements of the CodeFeedr ecosystem. However, for shared-state utilities like API key management it seems to be insufficient: the process to get a key and update the number of calls the key can do is very long with multiple calls to Zookeeper and no guaranteed atomicity unless full locking is applied which causes an even slower process and more issues.

Zookeeper is an excellent solution for storing configurations and storing where servers are located: it is very read-oriented. Reads are very fast while writes are very slow. Zookeeper is also an excellent solution for leader selection systems (such as Flink and Kafka). Performance of Zookeeper can be seen in Figure B.5.

B.5.2. Redis

Redis[2] is a lightweight in-memory data structure store, mostly key-value based, that can be used as cache, message broker and as database. It has some great features such hash sets, sorted sets and custom script support. It does not have automatic failover so for maximal uptime it is not the best choice. It is however extremely fast.

As Zookeeper is required for both Flink and Kafka, using a new service for key management with CodeFeedr is not ideal. However, it is very lightweight, easy to use and not required to make a Codefeedr project work.

[2] `https://www.redis.io`

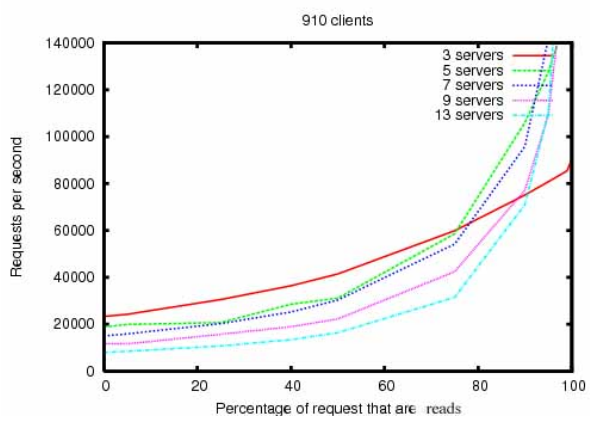

Figure B.5: A graph that plots the throughput of Zookeeper against the percentage of requests that are reads.[10]

B.5.3. Conclusion

As a key manager is a relative simple concept: only asking for a key when needed, it ought to be possible to make it a generic system and allow implementations for Redis, Zookeeper and even systems like mongoDB. It seems however that Zookeeper has quite some disadvantages over Redis, so a built-in Redis-only implementation might be desirable. However we will make our system configurable allowing for multiple implementations.

B.6. Containerization and orchestration

When spreading a piece of software across a cluster and scaling it, it is ideal to have the software in a format that is easily configurable and can run on any machine with any operating system. The software can then run on a developers machine or on any machine that can be found to add more power to the cluster. A simple 'start' command is much preferable over a long, specific, installation process.

These problems are solved with containers. Before the existence of containers this was solved with a much heavier format: virtual machines. The issue of scheduling containers on the cluster and placing them on servers is solved by container orchestration. In Subsection B.6.1 containerization will be discussed. Lastly, orchestration will be explained in Subsection B.6.2

B.6.1. Containerization

Docker

Docker is currently the market leader of containerization systems and has a large ecosystem of existing container images, management and monitoring tools, and user guides [3]. Docker containers are much better than virtual machines in terms of performance [23]. It is easy to get started but allows for complex setups as well.

The ability to build Docker images and run them on a Docker infrastructure has been requested by the client. It is a very good solution for setting up a group of services, as most of the more popular services have a ready-to-use image available on *Docker Hub*.

Other container systems

There are other container implementations such as *rkt* [4]. Rkt is a container system used on CoreOS based on the same standards as Docker is using (and building). Together with other projects it is part of open initiatives to standardizing containers and runtimes. However, it does not have the ecosystem Docker has and requires more technical knowledge to get working.

[3] `https://www.docker.com`
[4] `https://coreos.com/rkt/`

B.6.2. Orchestration

Running a set of services on a local computer is easy. It becomes more complex once these programs need to spread on a cluster and scale. Especially when resources need to be optimized. Orchestration issues are, for example: scale a certain service but keep them close together (within the data center). Or always keep an instance of a certain service on every node. Orchestration systems can also automatically scale applications.

Docker Compose/Swarm

Docker compose is a way to set up a whole network of services, connect them, and manage them from the command line. The compose tools comes built-in with Docker and has been used a lot by providers of Docker images as well. Docker Compose allows for running the whole orchestration locally on a computer with nothing more than the compose file.

Docker swarm mode, previously a separate system called Docker Swarm, is used to connect multiple machines and use them as if they were a single one. Using a network layer the machines act as if being one. It can be used to scale services across machines. It automatically handles service discovery and load balancing. Swarm mode is integrated in the Docker engine and readily available and supported. It has full integration in Docker Compose as well, allowing a single file to completely define a whole set of services across a bigger cluster.

Kubernetes

Kubernetes is an open source system for container orchestration. It is more capable than Docker but connects to the Docker ecosystem by using Docker containers. It is however much more complex to set up as well and is overly complex for using locally on a development machine. Kubernetes is mostly used by big SaaS providers.

B.6.3. Conclusion

Kubernetes is a big, complex system, mostly used for bigger cluster setups. As one of our goals is to make a simple framework, Docker Compose fits with this goal. The client has requested for a Docker set-up as well. Both arguments combined gives a good reason to use Docker with Docker Compose for the orchestration part of the project.

B.7. Discussion

We found various technologies which can be used to solve the problem statement. For most technologies there is simply not one way to go. Therefore we will first focus on creating a framework which is not specifically bound, apart from Flink, to a certain technology. This means that switching or adding elements like a message broker or serialization framework should be easy. The same holds for centralized state technologies necessary for key management. In terms of orchestration there is also a lot available, but functionality often goes at cost of complexity. Therefore, we will focus on using Docker in our product.

C

Roadmap

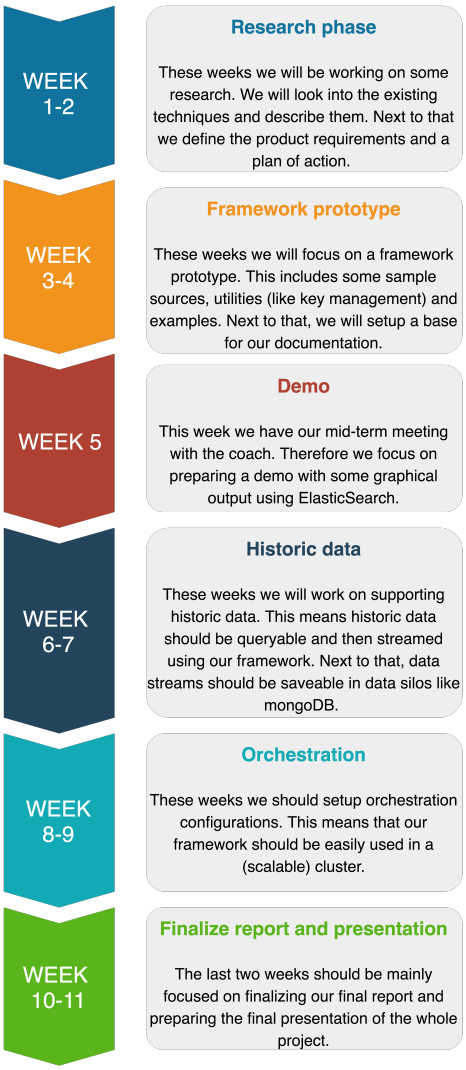

Figure C.1: Roadmap of our project.

D

Software Improvement Group

D.1. First Feedback (Dutch)

De code van het systeem scoort 3.9 sterren op ons onderhoudbaarheidsmodel, wat betekent dat de code marktgemiddeld onderhoudbaar is. We zien Unit Interfacing en Unit Size vanwege de lagere deelscores als mogelijke verbeterpunten.

Voor Unit Interfacing wordt er gekeken naar het percentage code in units met een bovengemiddeld aantal parameters. Doorgaans duidt een bovengemiddeld aantal parameters op een gebrek aan abstractie. Daarnaast leidt een groot aantal parameters nogal eens tot verwarring in het aanroepen van de methode en in de meeste gevallen ook tot langere en complexere methoden.

In jullie project hebben een aantal constructors, zoals bijvoorbeeld TravisBuild, wel erg veel parameters. Dat is in Scala bij een case class een stuk minder erg dan bij een "gewone" class, maar desondanks zou wat meer abstractie en wat minder primitives helpen om deze datastructuren begrijpelijk te houden.

Voor Unit Size wordt er gekeken naar het percentage code dat bovengemiddeld lang is. Het opsplitsen van dit soort methodes in kleinere stukken zorgt ervoor dat elk onderdeel makkelijker te begrijpen, te testen en daardoor eenvoudiger te onderhouden wordt. Binnen de langere methodes in dit systeem, zoals bijvoorbeeld, zijn aparte stukken functionaliteit te vinden welke ge-refactored kunnen worden naar aparte methodes.

Bij jullie zijn er niet zo heel veel lange methodes, maar er zijn nog een aantal voorbeelden die je nog iets verder zou kunnen aanscherpen. Zo zou je in getRSSAsString() in RSSSource.scala het retry-mechanisme van de daadwerkelijke logica kunnen scheiden. Die twee lopen nu door elkaar heen, wat de leesbaarheid zou kunnen verlagen op het moment dat de hoeveelheid functionaliteit gaat groeien.

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. Adjustments made based on the feedback

The feedback consisted of two points: Unit Interfacing and Unit Size. We have used the SIG-built tool called BetterCodeHub to identify the code that needed changes in more detail as it seemed to mostly match the feedback we got.

The Unit Interfacing has mostly stayed the same. Most of the classes with many constructors parameters are stages. Important here is that constructor parameters are the only way to pass information to stages, because of our immutability requirement (needed for distributed computing with more confidence). This means we can't more parameters to setters. To make it easier on the user we have given most of these parameters default values and put the required parameters in the front. Putting the parameters in some kind of attributes or configuration object would only have moved the problem.

Many other classes that were noted were actually case classes with direct mappings onto a REST API (for example the TravisBuild class). Splitting those is detrimental to readability and understanding. They are also never created by a programmer but only by (de)serializers.

The feedback on the Unit Size has mostly been addressed. All methods that had gotten too long were refactored to multiple smaller methods to make them easier to read and understand, unless it was detrimental to the understanding. Methods with a very clear process were kept as is (for example the mongoDB key manager key refreshing).

D.3. Second Feedback (Dutch)

In de tweede upload zien we dat het project een stuk groter is geworden. De score voor onderhoudbaarheid is in vergelijking met de eerste upload gestegen.

In de feedback op de eerste upload werden twee mogelijke verbeterpunten genoemd: Unit Size en Unit Interfacing. Bij allebei zien we een duidelijke stijging, die zowel door refactoring van bestaande code als door verbeterde standaarden in de nieuwe code komt. Bij Unit Size is de stijging zelfs zeer groot, complimenten daarvoor.

Zoals jullie per email hebben aangegeven hadden jullie in de eerste upload wel degelijk testcode, dus die opmerking uit de feedback op de eerste upload komt te vervallen. Naast de toename in de hoeveelheid productiecode is het goed om te zien dat jullie ook nieuwe testcode hebben toegevoegd. De hoeveelheid tests is wel wat lager uitgevallen dan bij de eerste upload, probeer hier in de toekomst op te letten.

Uit deze observaties kunnen we concluderen dat de aanbevelingen uit de feedback op de eerste upload zijn meegenomen tijdens het ontwikkeltraject.

E

Code examples

E.1. Simple stages

```
case class SimpleData(str: String)
case class SimpleDataReduce(str: String, amount: Int)

class SimpleInputStage() extends InputStage[SimpleData]() {

  override def main(): DataStream[String] = {
    environment
      .fromCollection(Seq(SimpleData("Simple"), SimpleData("data"), SimpleData("set")))
  }
}

class SimpleTransformStage() extends TransformStage[SimpleData, SimpleDataReduce]() {

  override def transform(input: DataStream[SimpleData]): DataStream[SimpleDataReduce] = {
    input
      .map(x => (x.str, 1))
      .keyBy(0)
      .sum(1)
      .map(x => SimpleDataReduce(x._1, x._2))
  }
}

class SimpleOutputStage() extends OutputStage[SimpleDataReduce]() {

  override def main(input: DataStream[SimpleDataReduce]): Unit = {
    input.print()
  }
}
```

E.2. Pipelines

E.2.1. Simple pipeline

```
object Main {
  def main(args: Array[String]) {
    new PipelineBuilder()
      .append(new SimpleInputStage)
      .append(new SimpleTransformStage)
```

```
      .append(new SimpleOutputStage)
      .build()
      .start(args)
  }
}
```

E.2.2. Complex pipeline

```
object Main {

  def main(args: Array[String]) {
    val inputStage = new SimpleInputStage
    val outputStage = new SimpleOutputStage

    val transformStage = new SimpleTransformStage
    val otherTransformStage = new SimpleOtherTransformStage

    new PipelineBuilder()
      .edge(inputStage, transformStage)
      .edge(inputStage, otherTransformStage)
      .edge(transformStage, outputStage)
      .edge(otherTransformStage, outputStage)
      .build()
      .start(args)
  }
}
```

E.3. Using plugins

The following code gets events from GitHub, splits them into push events, issue events, and issue comments. It sends all these events separately to ElasticSearch indices.

```
object Main {
  val eventSource = new GitHubEventsInput(-1, 5000, true)
  val pushEvents = new GitHubEventsToPushEvent
  val issuesEvent = new GitHubEventToIssuesEvent
  val issueCommentEvent = new GitHubEventToIssuesCommentEvent

  val pushEventSink = new ElasticSearchOutput[PushEvent]("push_events")
  val issuesEventSink = new ElasticSearchOutput[IssueEvent]("issues")
  val issueCommentEventSink = new ElasticSearchOutput[IssueCommentEvent]("issue_comments")

  def main(args: Array[String]) = {
    new PipelineBuilder()
      .setKeyManager(new RedisKeyManager())
      .edge(eventSource, issuesEvent)
      .edge(eventSource, issuesCommentEvent)
      .edge(eventSource, pushEvents)
      .edge(pushEvents, pushEventSink)
      .edge(issuesCommentEvent, issueCommentEventSink)
      .edge(issuesEvent, issuesEventSink)
      .build()
      .start(args)
  }
}
```

F

Implementation challenges

We describe some of the problems we run into that caused us to change design or drop features all together. It is a useful read for those that want to contribute to the core framework. If you are interested in why some features were dropped this can be interesting as well.

F.1. Type system limitations

We had many adventures with the Scala type system. One of the problems is that Scala runs on the JVM which has no generics, so Scala erases those types. To use any type information at runtime (needed for serialization), we had to indicate that we wanted Scala to keep that type attached to the class. This is done using context bounds (a sugar on implicits), but they only work locally and do not propagate to subclasses. This means we have to annotate every stage with both ClassTag and TypeTag.

For representing a pipeline, we created a small data structure for a directed acyclic graph. It has some custom functionality for testing for sequentiality and parent ordering, needed for binding the buffers to the input types of a stage. However, it was not possible to make the DAG class use generics, and uses AnyRef as content type instead. This is because PipelineObjects can't be cast without their input and output types being covariants. If they are covariants (which would be valid in our case because of the way we use the types) the type checker fails because the elements of Flink DataStreams are invariant in the Scala type definitions. We have asked some Scala experts for ideas or solution but none work in our situation. Thus we decided to stick to AnyRef and cast instead. This is only done internally.

F.2. Serializability

Apache Flink serializes objects in order to send them to job managers, which in turn execute that object. It is their technique for distributing and scaling the streaming process. This requires many parts of the code surrounding Flink to be serializable. This was quite an issue when using services for plugins or when passing special objects like LocalDateTime, which could not be serialized by Flink.

Whenever such a problem appeared we needed to circumvent it by either executing the code on each node by making it lazy, or not using those types at all. When using a lazy class property, the value will be created on the job runner instead of on the task manager. Its contents don't need serialization. This is not possible when you have state you need shared (a big reason to circumvent any changing state such as key management). Such state can also be the constructor arguments. We have also used strings to pass information where a type would have been better but the type was not serializable (see the date format in the RSS input stage).

Note: Not everything is made serializable in libraries because once a class is made serializable is becomes incompatible with any version that changes the contents.

F.3. Avro support

Initially, we planned to support Avro serialization before reading and writing to buffers. Especially, since Avro offers schemas for datatypes which could be exposed and used by external parties. Unfortunately, Avro does not have native Scala support. There are some frameworks out there like: avro4s [1] and shapeless-datatype [2] which try to support Avro using macro conversion of case classes. They both gave us some problems:

1. To use this serializer in a generic way, we had to give context bounds or pass implicits to every class which specified or forwarded the type of the data. This made compiling the code extremely slow (a few minutes for simple case classes), because all the generics were type checked and expanded on compile time. Complex case classes took even longer and recursive case classes compiled infinitely.
2. If types were not supported by the library like 'java.util.Date', we could not support it throughout the whole framework even if you would not use the Avro serializer. This was caused by the use of context bounds, which were still defined even if you did not use them.
3. It limited us in only allowing case classes (no tuples, no primitives etc.), because they were not supported by both libraries.

Due to these issues we decided to drop the Avro support and focus on serializers which did not require compile-time macro expansions and type checking nor limited type support.

F.4. New serializer

After implementing the mongoDB stages into a plugin, we discovered that its BSON implementation is quite easy to work with. We decided to add it as a new serialization option. It is a smaller and faster format than JSON but not directly readable by humans and less supported.

[1] https://github.com/sksamuel/avro4s
[2] https://github.com/nevillelyh/shapeless-datatype

Info Sheet

A plug-in infrastructure for the CodeFeedr project

Client organization: Delft University of Technology
Presentation date: July 2, 2018

Description

CodeFeedr is a research project at the software engineering division of the Delft University of Technology in collaboration with the Software Improvement Group. The research focuses on a software infrastructure which serves software practitioners in utilizing data-driven decision making [18].

Currently, frameworks like Apache Flink are capable of high-performance data streaming. However, these frameworks have a lot of overhead in setting up and adding new streaming queries takes a lot of time. These frameworks also have some limitations in combining real-time data with historical data and doing aggregations on streams from multiple sources.

The product that has been developed is a plug-in framework on top of Apache Flink, that removes the overhead on setting up stream processing jobs and makes it possible to combine real-time and historical data as well as the aggregation of multiple data sources. This product includes abstractions for well-known sources like GitHub, TravisCI and Twitter. This way the user can spend its efforts on actually writing streaming queries instead setting up environments, input sources and output. The product also includes orchestration tools for running streaming jobs on a distributed system.

Members of the project team

Jos Kuijpers
Interests: Computer languages, pragmatic programming and code design.
Role: Mostly worked on the initial framework design and implementation. Focused on Redis and Mongo services and orchestration.
Contact: jos@kuijpersvof.nl

Joris Quist
Interests: Big data processing, software engineering and software testing
Role: Mostly worked on plugin implementations and general issues with the core framework.
Contact: jorisquist@gmail.com

Wouter Zorgdrager
Interests: Software engineering, big data, data science and project management
Role: Mostly worked on framework internals like the buffer system and serialization as well as several plugins like GitHub and Twitter.
Contact: zorgdragerw@gmail.com

Client: Dr. ir. G. Gousios, Software Engineering, Delft University of Technology
Department: Dr. ir. T. Abeel, Delft Bioinformatics Lab, Delft University of Technology

The final report for this project can be found at: http://repository.tudelft.nl

Bibliography

[1] The cambridge analytica files: the story so far. https://www.theguardian.com/news/2018/mar/26/the-cambridge-analytica-files-the-story-so-far, 2018. Accessed: 21-06-2018.

[2] Protocol buffers | google developers, 2018. URL https://developers.google.com/protocol-buffers/. Accessed: 06-06-2018.

[3] Rabbitmq, 2018. URL https://www.rabbitmq.com/. Accessed: 12-06-2018.

[4] Apache thrift, 2018. URL https://thrift.apache.org/. Accessed: 06-06-2018.

[5] Apache Avro. Apache avro, 2018. URL https://avro.apache.org/. Accessed: 30-04-2018.

[6] Apache Flink. Scalable stream and batch data processing, 2018. URL https://flink.apache.org/. Accessed: 26-04-2018.

[7] Apache Kafka. Apache kafka, 2018. URL http://kafka.apache.org/. Accessed: 26-04-2018.

[8] Apache Spark. Apache spark™ - unified analytics engine for big data, 2018. URL https://spark.apache.org/. Accessed: 26-04-2018.

[9] Apache Storm. Apache storm, 2018. URL http://storm.apache.org/. Accessed: 26-04-2018.

[10] Apache ZooKeeper. Zookeeper 3.2 performance. https://wiki.apache.org/hadoop/ZooKeeper/Performance, 2018. Accessed: 02-05-2018.

[11] Artima. Selecting testing styles for your project. http://www.scalatest.org/user_guide/selecting_a_style, 2018. Accessed: 30-04-2018.

[12] Artima. Using privatemethodtester. http://www.scalatest.org/user_guide/using_PrivateMethodTester, 2018. Accessed: 30-04-2018.

[13] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[14] Adam DuVander. 8 great examples of developer documentation. https://zapier.com/engineering/great-documentation-examples/, 2017. Accessed: 30-04-2018.

[15] FasterXML. Fasterxml/jackson, 2018. URL https://github.com/FasterXML/jackson. Accessed: 06-06-2018.

[16] Apache Flink. Apache flink - checkpointing, 2018. URL https://ci.apache.org/projects/flink/flink-docs-release-1.4/dev/stream/state/checkpointing.html. Accessed: 01-05-2018.

[17] Ellen Friedman and Kostas Tzoumas. *Introduction to Apache Flink*. OReilly, 2016.

[18] Georgios Gousios, Dominik Safaric, and Joost Visser. Streaming software analytics. In *Proceedings of the 2nd International Workshop on BIG Data Software Engineering, BIGDSE@ICSE 2016, Austin, Texas, USA, May 16, 2016*, pages 8–11. ACM, 2016. doi: 10.1145/2896825.2896832. URL http://doi.acm.org/10.1145/2896825.2896832.

[19] Li Haoyi. Old design patterns in scala. http://www.lihaoyi.com/post/OldDesignPatternsinScala.html, 2016. Accessed: 30-04-2018.

[20] Joseph Hejderup. Codefeedr project website. https://codefeedr.github.io/, 2018. Accessed: 26-04-2018.

[21] Richard Herschel and Virginia M. Miori. Ethics & big data. *Technology in Society*, 49:31 – 36, 2017. ISSN 0160-791X. doi: https://doi.org/10.1016/j.techsoc.2017.03.003. URL http://www.sciencedirect.com/science/article/pii/S0160791X16301373.

[22] Pieter Humphrey. Understanding when to use rabbitmq or apache kafka, April 2017. URL https://content.pivotal.io/blog/understanding-when-to-use-rabbitmq-or-apache-kafka.

[23] Babu Kavitha and Perumal Varalakshmi. Performance analysis of virtual machines and docker containers. In Shriram R and Mak Sharma, editors, *Data Science Analytics and Applications*, pages 99–113, Singapore, 2018. Springer Singapore. ISBN 978-981-10-8603-8.

[24] Dmitry Komanov. Scala serialization, Jun 2016. URL https://medium.com/@dkomanov/scala-serialization-419d175c888a. Accessed: 06-06-2018.

[25] Jay Kreps, Neha Narkhede, Jun Rao, et al. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, pages 1–7, 2011.

[26] Age Mooij. From a to l: Designing scala libraries. https://speakerdeck.com/agemooij/from-a-to-l-designing-scala-libraries, 2013. Accessed: 30-04-2018.

[27] Martin Odersky and Lex Spoon. The architecture of scala collections. https://docs.scala-lang.org/overviews/core/architecture-of-scala-collections.html, 2017. Accessed: 30-04-2018.

[28] Scala Community. Scaladoc. https://docs.scala-lang.org/style/scaladoc.html, 2018. Accessed: 30-04-2018.

[29] Kai Wähner. Real-time stream processing as game changer in a big data world with hadoop and data warehouse, 2014. URL https://www.infoq.com/articles/stream-processing-hadoop. Accessed: 03-05-2018.

[30] Peter Zaitsev. Exploring message brokers: Rabbitmq, kafka, activemq, and kestrel - dzone integration, Jun 2014. URL https://dzone.com/articles/exploring-message-brokers. Accessed: 12-06-2018.