Bachelor Project

Monitoring-aware IDEs

Nick Yu
Maarten Lips
Ynze ter Horst
Thijs Molendijk
David Moolenaar

Hyperion

TU Delft

Bachelor Project

Monitoring-aware IDEs

To obtain the degree of Bachelor of Science
at the Delft University of Technology,
to be defended publicly on Wednesday July 1, 2020 at 10:00 AM.

Authors:	D. Moolenaar	
	T. Molendijk	
	M. Lips	
	N. Yu	
	Y. ter Horst	
Project duration:	April 23, 2020 – July 1, 2020	
Guiding committee:	M. F. Aniche,	TU Delft, supervisor
	Ir. J. Winter,	Adyen
	Ir. O. W. Visser,	Bachelor Project Coordinator

An electronic version of this thesis is available at http://repository.tudelft.nl/.

TU Delft

Preface

This project was conducted as part of the Computer Science and Engineering bachelor at the Delft University of Technology. Five students each worked over 400 hours on the project over the course of ten weeks. The resulting project, Hyperion, aims to integrate log monitoring solutions with the development environment for a more streamlined developer workflow. We would like to thank our coach M. F. Aniche for his continuous enthusiasm and guidance throughout the project. Special thanks also goes out to J. Winter, our company contact at Adyen. Jos, having implemented a similar system in the past, helped us out whenever we needed an opinion on a design decision or when we were unsure about how to tackle a certain problem. We've learned a lot during the development process, especially on how to build distributed scalable systems as a team. Additionally, we've enriched our development skills as students by working full-time on a single project and preparing it for suitable open-source distribution.

Summary

It has become a common practice for software developers to analyse monitoring data as a means to understand issues, trends and performance of large-scale software systems. The combination of this practice with the software development workflow can however be cumbersome at times. One recent effort into evaluating the effects of integrating monitoring directly into the software development workflow has been done at the large-scale payment company Adyen in cooperation with the TU Delft. This project is a new attempt at making a flexible, open-source and useful monitoring tool based on the results of that research effort. This report will detail how the system is designed, implemented and the development methodology that was used for its creation.

After researching popular monitoring stacks and solutions, an architecture for the Hyperion pipeline was designed that would maximize compatibility with logging setups. The monitoring data is first retrieved from a log source like Logstash or Elasticsearch. It is preprocessed by a pipeline consisting of configurable and flexible plugins. A large range of plugins is supported: functionalities include debugging, transformation, load balancing, and adding new data from third-party systems. There is also extensive support for third-party developers to create their own pipeline plugin to manipulate data in any way required, which allows Hyperion to account for unconventional logging setups. The pipeline typically ends at some type of aggregator, which exposes an interface for developer tools to retrieve metrics. These metrics can be queried with the Hyperion IDE plugin, which visualizes them in the development environment itself.

To verify that the designed Hyperion pipeline adheres to the requirements defined in the research report, validation tests have been performed on every module of the system. All code is extensively tested with unit, integration, and container-based system tests. An evaluation of the performance of the pipeline additionally verified the scalability of the system even when provided with a large number of incoming logs. Finally, in order to ensure easy extension and adoption of the Hyperion pipeline a large amount of documentation was written that explains the architecture of the project and how a third-party developer can adjust and extend it.

List of Abbreviations

API Application Programming Interface. 6, 10, 17, 19, 20, 37, 41

CI Continuous Integration. 31

ELK Elasticsearch, Logstash, Kibana. 14

IDE Integrated Development Environment. iii, 1–3, 19

JSON Javascript Object Notation. 7–10, 14

PR Pull Request. 31

TTL Time to live. 17

UI User Interface. 19, 22

VCS Version Control System. 20

ZMQ ZeroMQ. 6, 7, 9, 10, 15, 16, 30, 31

Contents

1 Introduction **1**

2 Problem Description and Analysis **2**
2.1 Problem Description 2
2.2 Problem Analysis 2
2.3 Requirements 4

3 Design and Implementation **6**
3.1 Architecture 6
3.2 Communication 7
3.2.1 Plugin Manager Protocol 7
3.2.2 Pipeline Communication 9
3.2.3 IDE Communication 10
3.3 Pipeline Plugins 11
3.3.1 Pipeline Commons 11
3.3.2 Included Plugins 13
3.4 Aggregator 16
3.4.1 Input Format 16
3.4.2 Aggregation 16
3.4.3 Querying 17
3.4.4 Scaling 18
3.5 IDE Plugin 19
3.5.1 Inline Metrics 19
3.5.2 Histogram Tool Window 22
3.6 Open-Sourcing Hyperion 23

4 Performance **24**
4.1 Complete Pipeline Performance 24
4.2 Individual Plugin Performance 26
4.3 Load Balancing 28
4.4 Comparing Buffer Sizes 29

5 Ensuring Software Quality **30**
5.1 Software Testing 30
5.1.1 System Tests 30
5.2 CI/CD 31
5.3 Software Improvement Group 31
5.3.1 First Inspection 32
5.3.2 Second Inspection 33

6 Process **34**
6.1 Group Dynamics 34
6.2 Communication with the Coach and Client 34
6.3 Development Methodology 35
6.4 Software Quality 35

7 Ethics **36**

8 Discussion **37**
8.1 Verification 37
8.2 Validation 39

9 Future Work **40**
9.1 Extend IDE Support 40
9.2 Extend Default Pipeline Plugins 40
9.3 Invidual Plugin Improvements 40
9.4 Emperical Testing. 40

10 Conclusion **41**

A Info Sheet **42**

B Project Description **43**

C Benchmark Results **44**

D Research report **46**

Bibliography **60**

1 Introduction

Many companies working with large-scale software require software developers to both write code and examine monitoring data. This monitoring responsibility is referred to as “DevOps” and is a fairly new methodology. While metrics can be very useful for instant feedback on code performance, issues and trends, they are often only accessible in a dedicated monitoring tool. As conjectured by Winter et al. [11], the need to switch between the development environment and monitoring tools decreases the value of metrics as the developer needs to consciously make an effort to consult the metrics.

A simple solution for this problem arises: integrate the metrics with the IDE that the programmer is presumably already using. By showing log metrics inline, Winter et al. propose that programmers can make more informed choices. This proposal was evaluated using a proprietary implementation created by payment processor Adyen, and has been shown to have positive effects. Due to privacy concerns and a specific integration with Adyen systems, this implementation was not open-sourced.

This report discusses our modular open-source implementation of the system described by Winter et al., called Hyperion. It will discuss the problem Hyperion attempts to solve, the development process, the architecture of the software, the tests performed to validate the use cases, as well as future steps and improvements.

We will first discuss and analyze the project in more detail in chapter 2. This is also where we define the functional requirements for the product and the criteria needed for a successful project. Chapter 3 extensively discusses the implemented product, both from a technical and an architectural point of view. Following the implementation, chapter 4 quantitatively analyzes the performance of the created Hyperion pipeline to assert that it is powerful enough to handle large logging rates. Chapter 5 discusses how the quality of the software was ensured, as well as the feedback given by the Software Improvement Group. We briefly touch upon the development process during the project in chapter 6, while chapter 7 discusses the ethical implications of the created product. Finally, chapters 8 and 9 discuss the created product and whether it fulfills all requirements, as well as potential future work that can be done to improve the product. A short conclusion and summary of the product is given in chapter 10.

2
Problem Description and Analysis

This chapter discusses a detailed analysis of the problem and possible approaches to solving it. The information presented in this chapter is a combination of the initial exploration done as part of the research report, as well as insights and observations made during the development of Hyperion. Should the reader be interested in more details and an evaluation of existing solutions, they are recommended to read the relevant sections of the research report, available in appendix D.

Section 2.1 formally describes the problem that Hyperion attempts to solve. Section 2.2 goes deeper and attempts to quantify the concrete challenges posed during development of the tool. Finally, the initial set of formal requirements from the research report are restated in section 2.3.

2.1. Problem Description

As briefly discussed in the introduction, the core problem this project aims to tackle is the cost of context switching between development tools and separate monitoring tools. As engineers are the ones who create software, they are the most capable of understanding the runtime behaviour of said software. Therefore, it makes sense for engineers to also be the ones that examine monitoring data to discover issues, performance bottlenecks, optimization opportunities or emerging trends.

However, analysing monitoring data is primarily done in monitoring tools such as Kibana[1] or Grafana[2]. Development on the other hand happens in separate tools, often text editors, terminals, or IDEs. Switching between these tools not only requires a conscious effort on the side of the developer, but research by Abad et al. [3] has shown that for a large majority of programmers the context switch needed to interrupt programming for consulting such metrics negatively impacts the efficiency and performance of the engineer.

Previous research done by Winter et al. [11] further confirms this. By integrating monitoring data inside the development environment, the cost of consulting these metrics decreases. As such, the Hyperion project aims to perform this integration in a dynamic and open-source manner. Due to a variety of logging and monitoring setups across companies, the Hyperion project needs to be flexible and stable while still being a useful tool to developers.

Concretely, the Hyperion project is an effort to bring metrics directly to the IDE using a plugin that integrates directly with the monitoring tools that a company already uses. The rest of this document explores how to do this in an efficient, scalable and correct manner.

2.2. Problem Analysis

Before starting development on Hyperion, we set out to answer a number of important main questions. By answering these questions, we aim to get a better idea of the full scope of requirements and challenges in the project.

[1] https://www.elastic.co/kibana
[2] https://grafana.com/

1. What logging information is helpful to the developer?
Logging information can generally be divided into two separate types: event logging and tracing. Event logging is purely the tracking of certain occurrences during execution. An example of such data and its use is comparing the ratio of successful and failed responses of a web service. Tracing on the other hand aims to keep track of the code flow of a single logical execution "unit", such as tracking which code paths are triggered by a request made to a web server. This information allows a developer to debug execution issues by tracking how the request got handled by the software.

Since tracing is more isolated for single transactions (and therefore only useful if a developer is searching for a particular issue), we chose to only implement tools for monitoring and visualizing event log metrics in Hyperion. For this purpose, the actual information needed to display useful information can be reduced to the time the log occurred, where it has occurred in the code and the corresponding severity level of the log. Additionally, we will want to track the both the service and the version of the code base that the log occurred in, as to differentiate between messages from different services and different versions.

2. How can this information be delivered from the source to the IDE?
The environment on which companies run their production system can vary in terms of machines, networking and location. One party could for example have a monolithic server running on a single machine while another could have a network of smaller services running on multiple machines spread over multiple server farms. Another aspect to consider is the technology stack used: having an intermediate logging pipeline (such as Logstash[3]) would require less steps in getting the metrics to the development tool compared to a setup that only produces raw log files.

Additionally, developers should be able to integrate the information in their development tool of choice. This implies that log aggregation needs to be done by some intermediate step, such that the actual integration with the IDE only needs to query this information instead of aggregating it directly. The external interface of this intermediate step would have to be flexible enough that any developer tool could be extended to interact with it.

3. How can this information be effectively presented to the developer?
How the information is shown in the IDE should be no less effective than if it were in a separate monitoring tool. The aim should be to assist the developer but not distract or even hamper development. One advantage that an integrated tool has over an external tool however is that metrics can be shown directly inline next to the relevant code. During development, this advantage should be considered when developing the IDE visualization. An example of how such visualization could look can be seen in Figure D.1.

From the results of the posed questions, we identified the following key issues and challenges that need to be solved in order to create a useful implementation of the tool.

- **Missing information**
  Some log sources do not encode the minimal set of information in the log message that is necessary for the tool to be effective.
- **Different production environment**
  The environment on which developers run their production system can vary in terms of machines, networking, location and technologies used.
- **Different log formats**
  Applications usually vary in both the output format of logs and the location where these logs are stored. There are standardization efforts such as Syslog[4], but they are not used widely enough.
- **Large log volume**
  As described by Winter et al. [11], applications used by large companies produce billions of log entries per month. Being able to process such a volume requires a scalable architecture.

[3]https://www.elastic.co/logstash
[4]https://tools.ietf.org/html/rfc5424

- **Tracing logs to their origin in the application**
  Due to the cost of resolving locations at runtime, not all companies include code locations when logging messages. Parsing code to heuristically trace these messages back to their origin has been done before (see Schipper et al. [10]), but this is a complex operation that requires specific implementations per language.

- **Fast indexing and retrieval of stored logs**
  Big applications produce a constant stream of logs which need to be indexed for retrieval. Efficiently processing and querying this data can be a challenge, especially when tens or hundreds of programmers access the same system concurrently.

- **Deciding what information to show to the developer**
  As this is relatively uncharted territory, there are no extensive studies on which information is most useful to show to developers.

2.3. Requirements

Based on section 2.1, section 2.2 and the client requirements, the list of functional requirements for the tool can be defined. These requirements have been adapted from the original requirements stated in the research report (see also section D.2.3), and are repeated here for later reference. They have been divided into categories based on the MoSCoW method described by Clegg et al. [7]. The requirements can be seen in Table 2.1.

For the project, we identified the main success criteria for a finished product to deliver a system that is easily adaptable to most possible logging systems that companies may have in use, while supporting enough throughput to handle the minimum requirement of 50 billion log lines a month. Additionally, since the product will be open-sourced we also expect to create a fully documented final product that is easy to setup. We consider the list of functional requirements defined in table 2.1 to be a suitable proxy for these criteria, and as such we expect a product that adheres to all defined requirements to be a successful completion of the project.

ID	Requirement	Justification
RQ1	**Must** support the Java language	Client requirement, commonly used enterprise language
RQ2	**Must** support the Elastic stack	Client requirement, most popular log management framework (as per section D.1.1)
RQ3	**Must** have a service for retrieving log statistics with an API	Allows for the creation of alternative front-ends, promotes modularity
RQ4	**Must** have a plugin for the Intellij IDEA platform (≥ v2020.1) that connects with the service	Client requirement, popular Java IDE (62% market share as of 2019 [1])
RQ5	Log line locations **must** be provided by the API	Allows for modular front-ends, less complexity tracing log lines on the client
RQ6	**Must** be able to show number of times log has been triggered in plugin	Client requirement, clear way too convey frequency of logging
RQ7	**Should** be able store logs statistics for a period of at least 30 days	The time to keep a log should be maximised but should not be excessive, older logs can be viewed in the company monitoring tool
RQ8	**Should** be scalable up to 50 billion log lines per month	Client requirement, upper estimate of log volume for Adyen
RQ9	Server **should** not re-aggregate when analytics data is already available	If the necessary data for the plugin is already stored in the user's database, it would be inefficient to aggregate it again
RQ10	Server **could** allow support for modularly adding missing metadata through plugins	When the log database does not contain all metadata needed for the plugin, this method can be used to compute this metadata during log ingestion
RQ11	**Could** support other IDEs (Eclipse, Netbeans, etc)	Supporting more users is a plus, but the API modularity allows anyone to implement a front-end
RQ12	**Could** visualize logs using graphs to view trend over time	With a graph the developer can more easily understand how certain code changes affect logging
RQ13	**Could** have heuristic log tracing	Tracing the origin of log lines can be inaccurate and inefficient, but is a requirement for certain tech stacks
RQ14	**Could** display the number of times a log line was involved in an exception	Provides a data-driven way for developers to see the impact of exceptions

Table 2.1: Functional requirements

3
Design and Implementation

To tackle the challenges detailed in section D.2, the Hyperion system had to be designed with certain requirements in mind. The design had to be modular, scalable and have low latency. This chapter discusses how each of these core requirements was achieved, by discussing both the overall architecture of the tool as well as a more detailed discussion of the individual implementation of Hyperion components. The following sections detail the general architecture of the tool (section 3.1), the protocol used for communication between the different parts (section 3.2), the implementation of the log processing pipeline and the plugins created for release (section 3.3), the implementation of the metrics aggregator and API (section 3.4) and finally the implementation of the Intellij IDEA plugin (section 3.5).

3.1. Architecture

The final architecture of Hyperion closely resembles the proposed architecture described in chapter D.3 of the research report. While the overall design of a pipeline system that originates at a data source and terminates in the aggregator is still appropriate, the pipeline was extended to support any number of plugins. Additionally, the protocol was adjusted from HTTP to using ZeroMQ (ZMQ) for performance reasons.

Figure 3.1 shows a full reference of the Hyperion architecture. Data is either pulled or pushed from a type of **data source** (Hyperion currently supports both Elasticsearch and Logstash), then is routed through the **pipeline**. The pipeline consists of a set of **plugins** that process the data in series, while the central **plugin manager** coordinates how data flows between the plugins. Finally, the data arrives at the **aggregator**, which indexes the results and provides an API for the **IDE plugin** to query against.

This main architecture was chosen for reasons also previously described in the research report. The main benefit of the pipeline architecture is the flexibility that it brings. By chaining configurable plugins together, it allows for almost any type of input data to be transformed into the format required by the aggregator. Similarly, since every plugin is its own process they can be run on separate machines to facilitate both horizontal and vertical scaling. This scalability requirement is the main reason why plugins are not embedded inside a single main process (such as the architecture that Logstash uses).

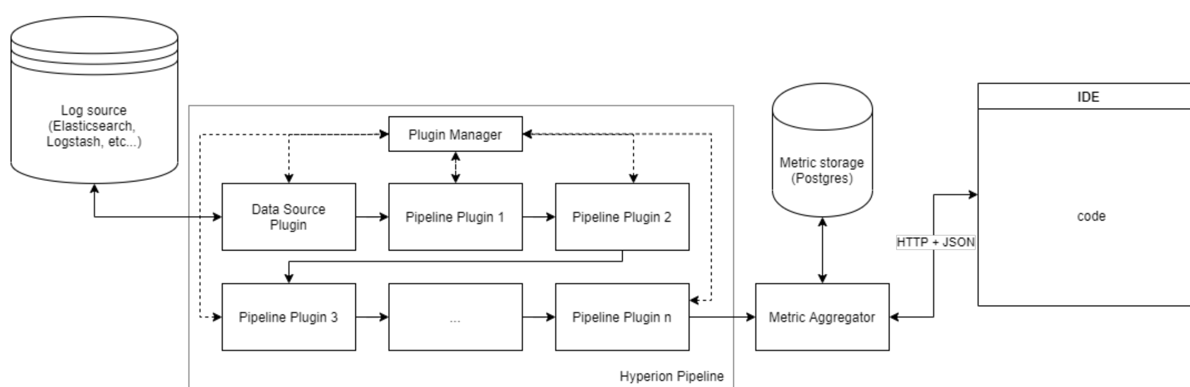


Figure 3.1: The overall architecture of Hyperion. Data is processed by a modular pipeline, before arriving at the aggregator. The aggregator indexes all data, then provides an API for the IDE to query against.

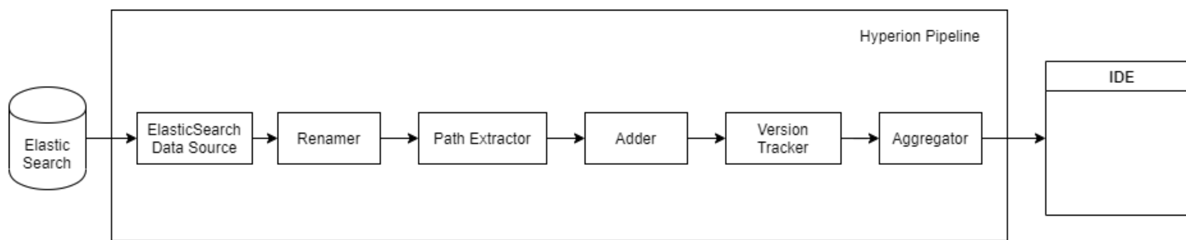


Figure 3.2: A concrete example of a Hyperion pipeline using the Elasticsearch data source and four processing plugins. The plugin manager and database are omitted for brevity.

Figure 3.2 shows a concrete example of a populated Hyperion pipeline with several plugins. Each plugin individually operates on the incoming data, transforms or extends it, then sends it to the next step. The communication through ZMQ ensures that data flows predictably and that networking issues are automatically resolved. Section 3.2 describes this communication protocol in more detail.

3.2. Communication

Within the Hyperion pipeline, there are a multitude of components that need to be able to communicate with each other. Pipeline plugins need to be able to send and receive messages, which not only involves a common data format but also knowing where previous and next steps are located inside the pipeline.

Functionally, we can divide all communication done by Hyperion components into three separate categories: communication between the plugins and the plugin manager, communication directly between plugins and communication between the aggregator and the IDE. It is important to note here that the start- and end-points of the pipeline (usually a data source and the aggregator) use the same protocol as intermediate plugins.

We will discuss each of these communication methods in more detail. For a full overview of the Hyperion protocol, including details about the exact message payloads and the settings used by the plugins, the reader is invited to read the `protocol.md`[1] documentation that is part of the Hyperion repository.

3.2.1. Plugin Manager Protocol

As briefly discussed in the previous section, the Hyperion pipeline uses ZeroMQ (ZMQ) to communicate between pipeline steps. Since this is a peer-to-peer protocol without a centralized server, this means that each plugin needs to know the location of at least the preceding plugin, as well as the succeeding plugin (where appropriate).

The plugin manager component exists for this purpose. It needs to be configured by the user to contain a list of plugins, including their order and the hostname/port pair of the machine that hosts them. Listing 3.1 shows an example of such a configuration.

With this architecture, a plugin will only need to have a unique ID and need to know where the plugin manager is located. It can then query the plugin manager to receive information about its adjacent plugins.

To query this information, the `REQ`/`REP` ZMQ socket pair is used. The plugin will initiate a `REQ` socket, connecting to the plugin manager. It will then query for information using a JSON-based protocol. The plugin manager will use a `REP` socket, bound to a configured port, to respond to these queries. This handshake implies that the plugin manager needs to be running before any plugins can start. In practice however, ZMQ will automatically block until the plugin manager is available. This ensures that even if the plugin manager is launched later (such as when using an orchestrated deployment system), plugins are still able to request connection information.

[1] https://github.com/serg-delft/hyperion/blob/master/docs/protocol.md

Listing 3.1 An example of the configuration used by the plugin manager to specify the order of plugins within the pipeline.

```
plugins:
  - id: "Datasource"
    host: "tcp://datasource:30002"
  - id: "Adder"
    host: "tcp://adder:30003"
  - id: "PathExtractor"
    host: "tcp://pathextractor:30004"
  - id: "Renamer"
    host: "tcp://renamer:30005"
  - id: "Aggregator"
    host: "tcp://aggregator:30006"
```

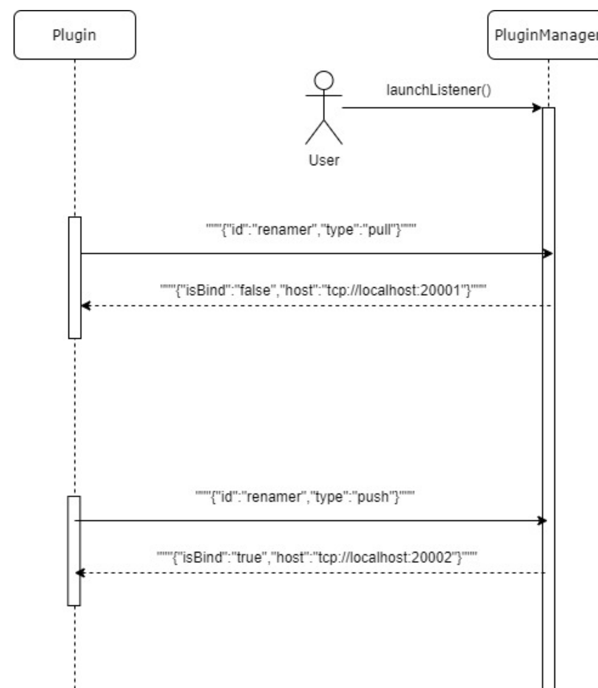


Figure 3.3: Sequence diagram of the plugin manager protocol

Figure 3.3 shows a sequence diagram depicting this plugin manager handshake, including the specific JSON format used. For brevity's sake, we will not document the handshake protocol in detail in this document. Readers interested in exact details can consult the `protocol.md` document available in the Hyperion repository [5].

One interesting aspect of the plugin manager protocol is that it allows a plugin to determine whether it can only send messages, only receive messages, or both. This was intentionally done such that debugging plugins can function as both pass-through plugins, as well as final plugins. Most other plugins will simply use this information to assert that the plugin is positioned properly within the pipeline. Algorithm 3.1 describes the process of a plugin connecting with the plugin manager.

Algorithm 3.1 Pseudo-code describing the steps taken by a pipeline plugin to retrieve connection information from the plugin manager.

```
1:  function Retrieve Connection Information(plugin manager host, id)
2:      Socket ← CreateZMQSocket(REQ)
3:      Socket.Connect(plugin manager host)
4:
5:      PreviousPlugin ← Socket.Request(id, PULL)
6:      NextPlugin ← Socket.Request(id, PUSH)
7:
8:      if PreviousPlugin is null then
9:          error if we expect to read data
10:     end if
11:     if NextPlugin is null then
12:         error if we expect to write data
13:     end if
14:
15:     IncomingSocket ← CreateZMQSocket(PreviousPlugin, PULL)
16:     OutgoingSocket ← CreateZMQSocket(NextPlugin, PUSH)
17: end function
```

3.2.2. Pipeline Communication

Once the plugin has requested connection information from the plugin manager, it can connect to adjacent plugins. As previously mentioned, ZMQ is also used for this purpose. By using ZeroMQ, the framework takes care of message sending, connecting and reconnecting, internally buffering messages and providing excellent performance.

Given that ZeroMQ sockets are modeled after classic networking sockets, they follow a single-server, multiple-clients model. This means that the plugins also need to know whether they need to act as a server (`bind` to the port) or as a client (`connect` to the port). This information is provided as part of the connection information returned by the plugin manager. Almost always, this results in the plugin `connect`ing to the previous plugin, and `bind`ing a socket for the next plugin to connect to.

For messaging, the Hyperion pipeline uses the `PULL`/`PUSH` ZeroMQ socket pair. This socket type is a classical socket, with the added benefit that messages are evenly distributed among receivers if there are multiple `PULL` sockets bound to a single `PUSH` socket. This allows for transparent load balancing of plugins (see also section 3.3). Figure 3.4 shows a schematic overview of all ZMQ sockets and how the plugins connect.

As a way to prevent slower plugins from causing infinitely growing buffers when the message rate is faster than what they can process, all messages are sent with the ZMQ `NOBLOCK` flag. This causes ZMQ to instead drop messages if they cannot be received by the next stage. This tradeoff was made to ensure that the pipeline does not collapse entirely. Section 3.3 discusses this in more detail.

Several Hyperion plugins (most notably the printer and rate plugins) also include support for acting as both a transforming plugin as well as a final step in the pipeline. These plugins simply check whether they have the ability to send messages to the next plugin, and drop messages if that is not the case. This allows for easier debugging, since a plugin like the rate plugin can be useful both as an intermediate (for detecting potential plugin bottlenecks in the pipeline) as well as a final sink (for benchmarking the overall performance of the pipeline).

One important detail to note is that ZeroMQ simply transports transparent byte streams. There is therefore no requirement to use a specific encoding for passing messages. All official plugins written during Hyperion development however use a JSON-based codec. JSON was used because of its ubiquity in logging systems such as Elasticsearch and Logstash. By using the same encoding, no transformation step is needed. The use of JSON is not mandatory for third-party plugins. If a more efficient format is

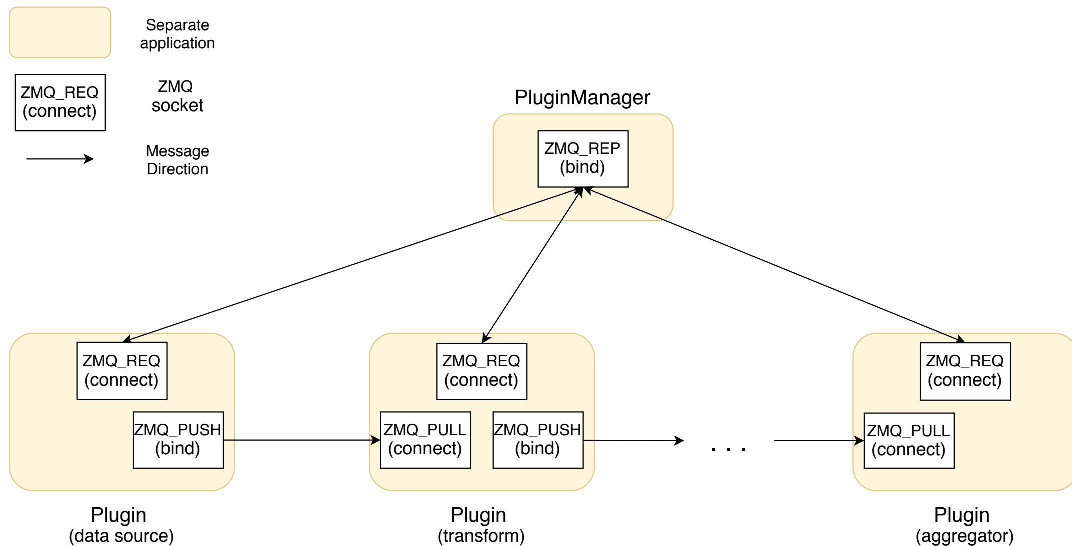


Figure 3.4: Overview of communication in the pipeline. The top text of each ZMQ socket specifies the socket type and the bottom text indicates whether it is a bind or connect socket. Note that even though the first plugin is a bind socket, the message direction goes towards the second plugin in the pipeline.

wanted or needed by a third-party, they are able to do so. This does however imply that these custom plugins are no longer able to interface with the official plugins included in Hyperion.

Currently, all data is sent in plain uncompressed text. Possible future work could include investigating the performance impact of compressing data before transmitting it to the next step. If a company has regular blobs of large text, the gains caused by faster throughput may outweigh the costs of (de)compressing the data.

3.2.3. IDE Communication

There is an additional communication protocol between the IDE and the aggregator. Since this communication channel needs neither realtime nor push capabilities, a simple JSON-based HTTP API is used in the aggregator. This allows the reference IntelliJ plugin, alongside any potential third-party IDE integrations, to communicate with the aggregator by issuing HTTP requests. The aggregator currently offers two separate API endpoints: inlay line metrics for the current file, and histogram metrics for powering the integrated chart views.

A key decision was to make the aggregator host and project name configurable inside the IDE. An alternative considered was to add support for the aggregator to provide a list of ”known projects”, and instead only having the user select the appropriate project from a dropdown list. We ended up not doing this because it would cost considerable extra effort in the aggregator source for questionable benefit. As a result of this decision, the IDE is required to supply the name of the relevant project in every request.

One other interesting consideration is the lack of authentication on the aggregator API. This was done partially because of the extra effort needed on both the frontend and the backend to implement some form of authentication, and partially because a system such as Hyperion is intended to be hosted in an internal network. Even when the aggregator API is open to the public, an attacker needs to guess project names, and will only be able to see the names of files inside the repository. Nevertheless, adding support for some form of authentication could be a future improvement.

Inline Line Metrics

The inline line metrics endpoint is located at `/api/v1/metrics`. When given a project name, file name, and list of intervals, it will return the relevant log lines in that file and how often they triggered over the specified interval. The IDE is then responsible for resolving the current line numbers of those

Listing 3.2 An example response of the inline metrics API call, showing log messages occurring on lines 10 and 20.


```
[{
    "interval": 60,
    "versions": {
        "bd5ed4a7680305b91d35ea56d7b103c1340dace3": [{
            "line": 10,
            "count": 20,
            "severity": "INFO"
        }],
        "8050421326b4a92a3fbe2f4d566ed72a940145db": [{
            "line": 20,
            "count": 1,
            "severity": "DEBUG"
        }]
    }
}]
```


files and their relevant lines in the current working copy of the developer. This endpoint is used to power the inline line metrics in the reference IDE implementation. An example of the data returned by this endpoint can be seen in listing 3.2.

Histogram Metrics

The histogram metrics endpoint is located at `/api/v1/metrics/period`. It works very similar to the inline metrics, but instead of simply returning the number of log entries that occurred in a specific interval it allows for the IDE to request a specific interval and the number of "buckets". This request powers the histogram graphs that are in the reference IntelliJ plugin implementation.

For brevity, the exact requests and their response formats are not described in this report. Should the reader be interested in the underlying implementation or in creating a new IDE implementation for Hyperion, we recommend they read the `writing-custom-ide-implementation.md`[2] documentation that was created to aid programmers.

3.3. Pipeline Plugins

In order to facilitate the easy creation of pipeline steps, a pipeline commons library was constructed for use in both official and third-party pipeline plugins. By abstracting away all the ZMQ communication and plugin manager handshaking (as previously discussed in section 3.2), plugin implementations can be made without the need to worry about implementation details. When needed, the internals are still exposed to implementing plugins.

Additionally, a set of default plugins was constructed to perform some of the most common transformations needed. Debugging plugins are also included to test new plugins, as well as plugins to measure and distribute load among multiple instances of the same plugin. The created plugins are briefly discussed in section 3.3.2, but they can also be found in the list of packages on the main repository README[3].

3.3.1. Pipeline Commons

The main structure of the pipeline commons library, as well as the adder plugin using it to process messages, can be seen in figure 3.5. At its core, the main `AbstractPipelinePlugin` class does most of the heavy lifting. It is responsible for handling the handshake with the plugin manager, as well as constructing and managing the incoming and outgoing sockets (where appropriate).

[2] https://github.com/serg-delft/hyperion/blob/master/docs/writing-custom-ide-integration.md
[3] https://github.com/serg-delft/hyperion/

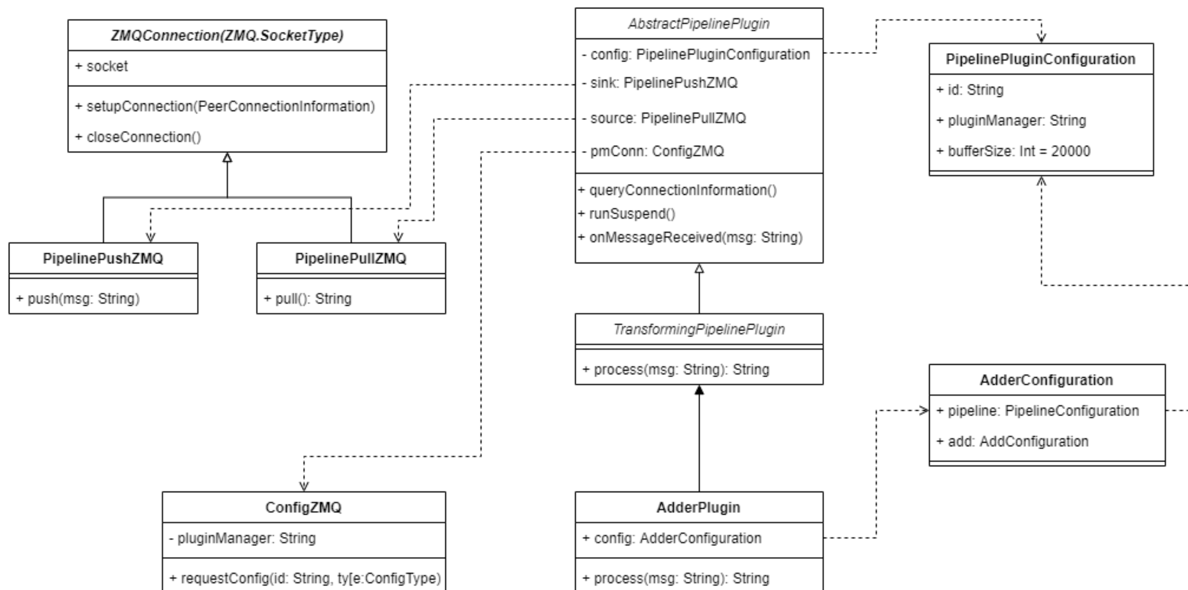


Figure 3.5: The class structure of the pipeline commons library, and the adder plugin extending it to implement plugin functionality.

Listing 3.3 An example transforming plugin, showing the ease of creating a new plugin using the pipeline commons library.

```
class AppendWorldPlugin(config: PipelinePluginConfiguration)
    : TransformingPipelinePlugin(config) {
    override suspend fun process(input: String) {
        return input + ", world"
    }
}

fun main(vararg args: String) {
    runPipelinePlugin(args[0], ::AppendWorldPlugin)
}
```

Classes extending from the `AbstractPipelinePlugin` are only required to implement `onMessageReceived`. This method is invoked whenever a message is received from the previous step in the pipeline. Despite the presence of this method however, the abstract pipeline plugin class also supports plugins that only send messages. For such plugins, the implementation of the receive method can simply be left empty. The `send` function can be used by the plugin to send messages to the next stage in the pipeline.

To simplify this process slightly for plugins that only transform incoming data, the abstract `TransformingPipelinePlugin` class is also provided. It combines `onMessageReceived` and `send` into a single `process` function that receives the incoming message and returns the transformed message to be sent to the next stage. Often, a transforming plugin can be implemented entirely in this single function. This reduces the code needed to write a new pipeline plugin to a handful of lines for simple transformations. To illustrate the simplicity of implementing a new function, listing 3.3 shows all code required to implement a simple plugin that appends `", world"` to all incoming messages.

Outside of simplifying the development process, `AbstractPipelinePlugin` also ensures proper stability and resource usage. By using the coroutine[4] functionality available in Kotlin, the abstract pipeline plugin distributes all processing work through a thread pool. This ensures that if a plugin

[4] https://kotlinlang.org/docs/reference/coroutines-overview.html

needs to do computationally heavy work, this is evenly distributed among all cores available on the machine.

Additionally, for transforming plugins `AbstractPipelinePlugin` keeps a counter in memory that tracks the number of messages currently being ”handled” by the system. This ensures that if a plugin is unable to keep up with the rate of incoming messages, these messages can be dropped entirely instead of infinitely consuming memory and causing the system to crash. The maximum number of messages that are currently being handled by the plugin can be configured by the user through the `buffer-size` configuration property. Before this functionality was implemented, plugins regularly consumed more than 10 GiB of RAM during stress-tests.

Outside of all transforming pipeline plugins, both the aggregator and the Elasticsearch data source also use the pipeline commons library. This demonstrates that the library is flexible enough to function even if messages are only being sent or only being received. Additionally, this ensures that only a single (properly tested) implementation of the networking logic exists.

3.3.2. Included Plugins

Given the large variety in common logging systems and the type of information they collect, a default set of various plugins was delivered alongside the Hyperion pipeline. These plugins each perform elementary operations on the data, and can be chained together to transform data as needed. For conventional logging setups, the set of default plugins delivered with Hyperion should be enough for almost all use-cases.

All Hyperion plugins have been benchmarked to ensure that they are able to handle the minimum throughput of 50 billion log messages a month as defined by RQ8. Depending on the type of hardware, each plugin is able to handle from 20,000 messages a second to up to 500,000 messages a second. Chapter 4 further discusses the performance of the pipeline.

Roughly speaking, the delivered plugins can be divided into four different categories: transforming plugins, data source plugins, debugging plugins, and the load balancer. Within this document, we will briefly discuss most of these plugins, opting to briefly describe their functionality and avoid going into details unless necessary. Should the reader be interested in more detail about a specific plugin, they are recommended to review the relevant README file located in the Hyperion repository [5].

Debugging Plugins

The Hyperion project ships with four different debugging plugins, as listed below. They are intended for use during benchmarking and plugin development, allowing developers to easily generate messages, print their contents and monitor the rate of messages going through the system. For regular use, these plugins will likely not be used.

- **Stresser**: A stress-testing plugin that sends a configurable amount of copies of the same message as fast as possible. Used for stress-testing a single plugin or the pipeline as a whole.
- **Rate**: A plugin that monitors the rate of messages passing through the pipeline, periodically printing a report on the rate of messages and the total received messages. Used for stress-testing a single plugin or the pipeline as a whole.
- **Reader**: A plugin that reads input from stdin and forwards it to the next plugin. Used during plugin development to test plugin behavior.
- **Printer**: A plugin that prints all incoming messages to stdout, optionally forwarding them to the next plugin if not the last step in the pipeline. Used during plugin development to test plugin behavior.

Transforming Plugins
The Hyperion project ships with five different transforming plugins that each perform elementary operations on incoming data. They all assume that the content is formatted in JSON, as this is the implicit transport codec used by all official plugins (see also section 3.2). For some plugins, we will briefly discuss some implementation details.

- **Adder**: A plugin that adds a static value to all incoming JSON objects. Used in situations where a field needs to be set to a static value that does not depend on other factors, such as when adding a `project` field.
- **Extractor**: A plugin that performs regex pattern matching on an incoming JSON field and extracts the results into separate fields. Used in situations where critical information (such as the message severity) is stored in a larger string and needs to be extracted.
- **Path Extractor**: A plugin that transforms a Java package name into the appropriate file path. Since many Java logging libraries only support printing the package name of a file, this plugin allows for easy conversion to the format that Hyperion expects.
- **Renamer**: This plugin renames a set of fields in incoming JSON objects. Used in situations where the required data is already in separate fields, but with a different name than the format expected by the aggregator (such as renaming Elasticsearch's `@timestamp` to `timestamp`).
- **Version Tracker**: This plugin automatically attaches the git hash of the most recent commit on a specified branch to all incoming JSON objects. This allows companies that use a git branch for deployment to automatically tag incoming messages with the commit hash of the version currently running in production.

Unlike most other plugins, the adder plugin is able to monitor the configuration file and automatically apply changes to it. This feature was introduced to allow for situations where a value is statically known but occasionally changes. An example situation where this feature is useful is when a webhook is triggered during deployment. A third-party can then implement a simple script that only updates the adder configuration file, instead of writing a custom plugin. This prevents the third-party from having to implement a custom plugin with a functionality very close to the one provided by the adder plugin.

The path extractor plugin is included for simple JVM logging setups, but it is unlikely to work in situations where an unconventional repository setup is used. This is because the plugin only does text rewriting and does not actually resolve the relevant file across sub-projects. If a company uses a single repository that hosts multiple projects, the path extractor plugin will not be able to automatically detect the relevant class.

Finally, the version tracker plugin does not pull the latest version on every incoming message. Instead, the latest commit is periodically queried from a remote git repository and then attached to all incoming messages. This means that in some rate cases, log messages from a new deployment may still get associated with the commit hash of the old deployment. Due to Hyperion's focus on speed over accuracy, this trade-off was deemed acceptable over the alternatives.

Data Source Plugins
To satisfy RQ2, Hyperion ships with two different dedicated data sources (the stresser and reader also technically act as data sources, but they do not conform to the definition as used here). For data that is already located in Elasticsearch, a pulling pipeline plugin is included that periodically queries the server for new log messages and forwards them to the pipeline. For companies that use the ELK stack, an additional output plugin for Logstash cab be used that allows for messages to be pushed directly into the pipeline.

The Elasticsearch input plugin uses the pipeline commons library and requests log messages from Elasticsearch over HTTP. Due to this, its performance is lacking for companies that have a moderate to high throughput of logging messages (over 1000 messages a second). Unfortunately, Elasticsearch only exposes a pull-based API and has no way for new data to be pushed to listeners such as the

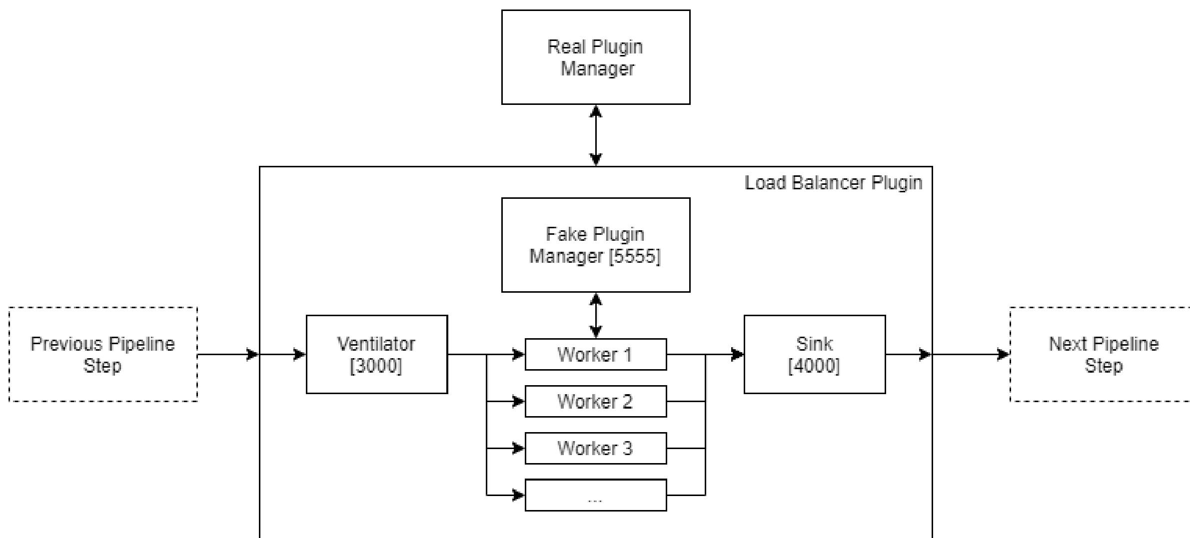


Figure 3.6: The architecture of the load-balancer plugin. A fake plugin manager talks to all workers, tricking them into connecting to the ventilator and sink ZMQ ports.

Hyperion pipeline.

To resolve this problem, an additional output plugin for Logstash was created. This allows companies that use Logstash to automatically push information into Hyperion, greatly improving performance. Unlike other pipeline plugins, this plugin was implemented in Ruby (Logstash runs on the JRuby[5] platform). This also means that the Logstash plugin has a separate implementation of the ZeroMQ networking logic. Naturally, a set of tests ensures that this implementation conforms to the standards.

Load Balancer Plugin

The load balancer plugin is different enough to warrant a separate category. Unlike all other plugins, it does not perform any changes on the input. Instead, its sole purpose is to evenly distribute incoming messages among multiple instances of the same ”worker plugin”. Using the load balancer, a company can improve the throughput of their Hyperion pipeline if it turns out that a single plugin step is the main bottleneck of the system. Using the load balancer can scale throughput almost linearly with the amount of workers. For more information on the load balancer and performance, see chapter 4.

The load balancer was designed in a way where it is effectively transparent to worker plugins. This ensures that a plugin does not need to be adjusted to be compatible with a load balancer: any plugin that works in a normal pipeline will also work in a load-balanced fashion. This was achieved by having the load balancer act as a fake plugin manager to its worker plugins. Since the worker plugins request connection information from the plugin manager, the load balancer can simply tell all worker plugins to connect to the same input port and push data to the same output port. Since communication goes through ZMQ `PUSH`/`PULL` sockets, the data will automatically be round-robin distributed among all available workers. A schematic overview of this in action can be seen in figure 3.6.

Despite effectively allowing a plugin to scale horizontally, there are a few caveats when using the load balancer plugin. First, a plugin can only really be used as a worker if it does not contain a shared state. For example, the rate plugin will not work in a balanced fashion as it will only report the number of messages passing through that single worker. Depending on the type of the plugin, such issues should be trivially resolvable. The documentation in the Hyperion repository regarding plugin creation also explicitly mentions that one should try keeping shared state inside a plugin to its minimum, for exactly this reason.

[5]https://www.jruby.org/

Additionally, due to the ZMQ socket configuration used by the Hyperion pipeline, the load balancer needs to host both a new ventilator socket as well as a new sink socket in order to be fully transparent to the worker plugins. This means that data effectively passes through the machine hosting the load balancer plugin *twice*. During load balancer stress-testing, we frequently ran into saturation issues with the networking card on the load balancer machine due to this issue (since a message rate of 1 Gbps effectively requires a networking card capable of 2 Gbps). Given that the alternative is to split the load balancer into two separate plugins that may or may not run on two separate machines, drastically increasing the complexity of configuring a load-balanced pipeline, this trade-off was considered acceptable. Future work can nevertheless consider splitting the load balancer into two separate components.

3.4. Aggregator

Once a log message has passed through the pipeline, it arrives at the aggregator for aggregation. The aggregator is responsible for indexing the tagged log messages, storing them for a specified amount of time, and providing them in an easily accessible format to the IDE implementations. The final design of the aggregator has seen relatively few changes compared to the original architecture proposed in section D.3 of the research report.

3.4.1. Input Format

The aggregator requires messages to conform to a certain JSON format. More specifically, each incoming message is required to have a `project`, `version`, `location`, `severity` and `timestamp` tag. The project field allows the aggregator to distinguish between log messages from different systems, while the version field allows the frontend to resolve the new position of the log line in the local file (as such, it usually is a git commit hash or tag). All other fields pertain to the log message, where it is located, and when it was issued. An example of a message accepted by the aggregator can be seen in listing 3.4.

Crucially, the actual logged message is not needed by the aggregator. This was done because the sole purpose of the IDE visualization is to render trends and rates of messages. Such visualization does not require the contents of the log, and therefore there is no real need to store the contents. This also allows for a more intelligent aggregation strategy (as described in the next subsection) and prevents storing the same information twice. Should the actual messages be needed, the programmer can still use the dedicated metrics system (such as Kibana or Grafana).

All messaging is handled by extending the aggregator from the same pipeline commons library described in section 3.3. This allows the aggregator to benefit from the fully tested communication logic also used in the pipeline plugins, without the need to reinvent the wheel.

3.4.2. Aggregation

In order to avoid storing all log messages individually, the aggregator keeps track of a "running total" over some specified granularity window. Messages are buffered inside memory for this granularity

Listing 3.4 An example of a logging entry when it is ingested by the aggregator.

```
{
    "project": "mock-logging",
    "version": "ac362e0a33062a0eec28dcb9a51d439976a53b4a",
    "timestamp": "2020-06-02T10:03:44.000Z",
    "location": {
        "file": "src/main/java/com/sap/enterprises/server/Main.java",
        "line": 11
    },
    "severity": "INFO"
}
```

window, after which they are committed as an intermediate aggregate to the database. This solution prevents storing duplicate data (both inside the aggregator and inside the original metric storage used by the company) and drastically reduces the amount of data that needs to be stored and queried when totals need to be computed.

One caveat of this approach is that delayed log messages may end up being aggregated in the wrong time slot (as the current buffered total pertains to the last granularity window, and does not go further back). Instead of extending efforts into adding support for potentially updating past intermediate aggregates, we instead opted to verify the `timestamp` field attached to the incoming log message. By default, the aggregator will reject messages that have a timestamp that does not fit in the current granularity window. A configuration toggle allows this verification to be turned off, although both the documentation and a runtime log message indicate to the user that this may result in messages being aggregated in the wrong time slot.

Algorithm 3.2 shows a pseudo-code overview of the aggregation. The actual implementation uses carefully picked data structures to allow for fast aggregation and includes support for multithreaded aggregation. Overall, this allows the aggregator to easily process more than 40,000 messages a second (for more information, see chapter 4).

Algorithm 3.2 The aggregation strategy used by the Hyperion aggregator.

```
1:  Intermediates ← {}
2:
3:  function handle message(message)
4:      if message.timestamp is too long ago then
5:          ignore message return
6:      end if
7:      Intermediates ← Intermediates ∪ {message}
8:  end function
9:
10: function commit intermediates
11:     write Intermediates to database
12:     Intermediates ← {}
13: end function
14:
15: loop
16:     Sleep(granularity)
17:     Commit Intermediates()
18: end loop
```

One additional feature of the aggregator is a check that periodically runs and removes old entries from the database. Since the aggregator is only meant to visualize recent trends, the aggregator has a built-in TTL (time-to-live) setting that automatically removes expired entries from the database. This ensures that the database does not grow infinitely in size. Additionally, requests made to the API exposed by the aggregator are automatically reduced to be within the range of the TTL.

3.4.3. Querying

Querying the intermediate aggregates stored in the database largely leverages the aggregation capabilities of the PostgreSQL database. As discussed in section D.3.3 of the research report, the PostgreSQL database was explicitly chosen because of its performance when querying metrics. Even during heavy load testing, we've seen consistent sub-50 millisecond query times.

The HTTP API methods exposed by the aggregator (documented in more detail in section 3.2.3) are fairly slim wrappers around a single database query. This design allows for all heavy lifting to be done by the highly performant PostgreSQL database, instead of writing and optimizing our own aggregation strategy. The aggregator only performs validation on the input, executes the query, then groups and

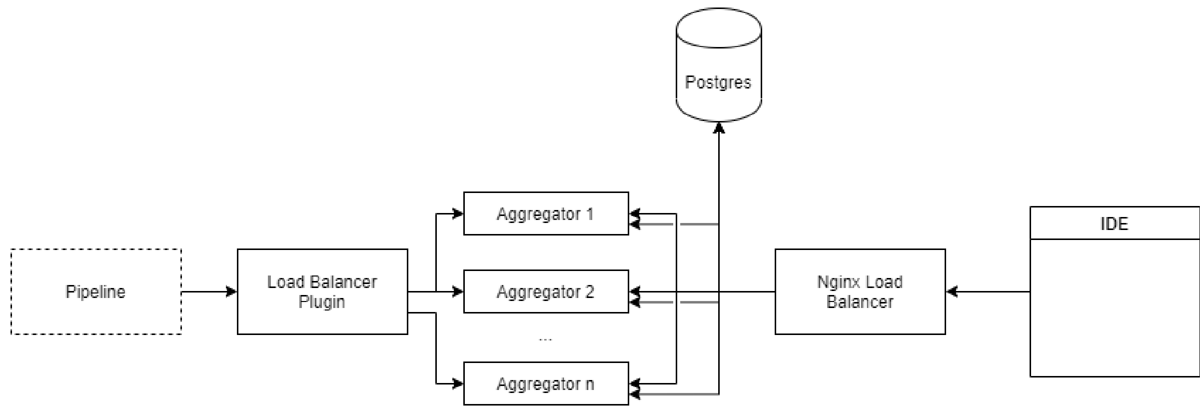


Figure 3.7: The aggregator used in a load-balanced setup, distributing messages and queries over all worker aggregator instances.

formats the output according to the input parameters.

One interesting feature missing from this querying design is live updates. The reason why live updates were not implemented is that the aggregator was designed with support for a multitude of different potential front-end plugin implementations in mind. Given that there are no current popular push-based communication strategies that work on a large number of platforms, alongside the extra effort needed to implement such a pushing system, we've opted to not implement pushing in the current version of the aggregator. If needed, a client can periodically pull metrics from the aggregator (doing so with an interval of `granularity` seconds will effectively act as live updates). The performance of querying allows such a design, although the overall performance will depend on the number of clients polling concurrently at a time.

While not deemed necessary during testing, future improvements to the aggregator may include using a memory cache such as Redis[6] to cache API requests. This could decrease the query time for companies with a large amount of developers, although this may be at the cost of real-time updates.

3.4.4. Scaling

Although not explicitly designed for this purpose, the aggregator supports being load-balanced using the pipeline load balancer plugin. Due to the way that intermediate aggregates are written to the database, there is no requirement for only a single intermediate aggregate to exist for a specific interval. This means that multiple instances of the aggregator can write to the same database in parallel, and each of them are able to query the same database to compute an aggregate for all metrics. Since each of these aggregator instances also exposes a web API for querying, this allows a web load balancer such as nginx[7] to distribute incoming queries among the different aggregator instances. Figure 3.7 shows a schematic overview of the aggregator running in a load-balanced fashion.

With this design, the aggregator is able to scale horizontally until either the load-balancer or the database become the bottleneck. Chapter 4 considers the performance of the load-balancer plugin, and from this we conclude that the aggregator is far more likely to be a bottleneck than the load balancer. At the traffic level where this becomes an issue, Hyperion may likely not be the correct tool to employ.

[6] https://redis.io
[7] https://nginx.org/en/

3.5. IDE Plugin

The IDE plugin is the final step in the Hyperion architecture. Communicating with the aggregator using the HTTP API described in section 3.2.3, it will retrieve current metrics for the current file/project and display them to the user. Given that this component is the main interface for the large majority of Hyperion users, extra focus was put on ease of use and ease of configuration.

As per RQ4 of the requirements (section 2.3), a plugin was created for the IntelliJ IDEA platform. The plugin therefore makes major use of the official IntelliJ Platform SDK[8], as well as Swing[9] for all UI components. The final plugin is distributed using the IntelliJ marketplace, allowing for a single-click install on all IntelliJ-based IDEs.

After installation, the user only needs to configure the hostname of the aggregator instance and the name of the project. These settings are stored on a per-project basis, allowing different projects to use different project names (or even different aggregator instances). Additionally, the user is able to configure the intervals shown as inlays in the main code. The settings panel can be seen in figure 3.8.

Functionally, the features of the IDE plugin can be divided into two separate categories. During file editing, the plugin will show inline metrics for relevant logging statements that have current recorded invocations. These are shown directly above the line and are intended to show the behavior of the code at a glance. Through the icon in the gutter of the editor, a more detailed visualization pane can be opened that shows a histogram of log messages over a configurable time interval. Examples of these features can be seen in figures 3.9 and 3.10 respectively.

3.5.1. Inline Metrics

The inline metrics are powered by IntelliJs flexible ”inlay” system, which allows a non-code block element to be positioned directly inside the editor. However, such inline blocks are not inherently attached to a specific position in the editor and therefore do not move with the line, should it be moved or indented. Given that the metric inlays are directly associated with the log line below them, extra effort was put into ensuring that the inlay always stays with the log line.

The first main challenge with this is to find the correct line for the inlay to render at. Given that the code currently running on production may be several commits behind the code the programmer is currently working on, it is naive to assume that the location of the log statement has not changed since the log was created. Therefore, intelligent log tracing is needed to resolve the historic position of the log line to the current file version as opened by the programmer. Given that the current version of a file may differ per programmer, we opted to do this resolving at the side of the plugin instead of centrally in the aggregator.

To resolve lines, the plugin uses the `git blame`[10] feature that is part of a normal git installation. It first issues a blame call on the version currently running in production, in order to resolve the commit

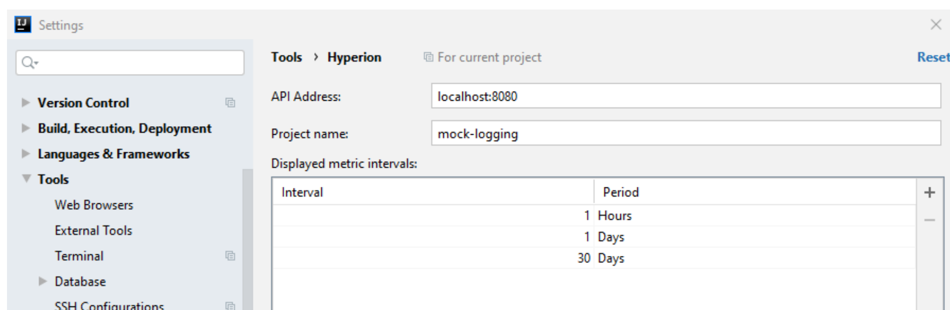


Figure 3.8: The settings panel shipped with the plugin. Both aggregator configuration and local UI settings can be edited here, and are stored on a per-project basis.

[8]https://www.jetbrains.org/intellij/sdk/docs
[9]https://docs.oracle.com/javase/8/docs/technotes/guides/swing/index.html
[10]https://git-scm.com/docs/git-blame

```
10     public TransportationService() {
          [23 last 1 h] [2780 last 1 d] [2780 last 1 w]
11        logger.info( s: "transportation serv
12   Hyperion: Visualize Line Metrics      iceActive =
13   Hyperion: Visualize File Metrics
14   Hyperion: Visualize Project Metrics
15     /**
16      * Use the teleportation service
17      */
```

```
18   public void teleport() {
19       if (!this.transportationServiceActive) {
             [188 last 1 h] [188 last 1 d] [188 last 1 w]
20           logger.warn( s: "Attempted to teleport
21
22           if (Math.random() < 0.5) {
                 [94 last 1 h] [94 last 1 d] [94 last 1 w]
23               logger.error( s: "Teleportation dev
24           }
25       }
```

Figure 3.9: [Left] The gutter icon expands with options to visualize the specified line, file or the entire project. [Right] The inline metrics rendered directly in the editor.

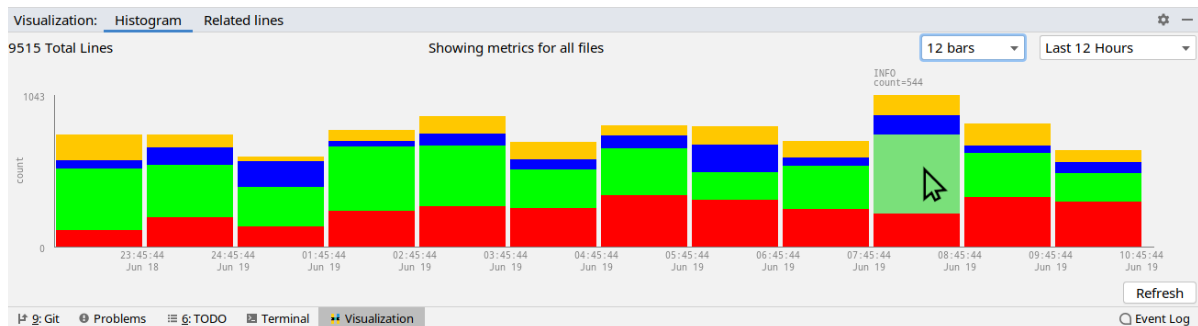


Figure 3.10: The detailed tool window with the an interactive histogram of log message trends. The cursor is shown hovering over the green box with some meta information appearing over the bar.

hash of the commit that introduced the log line in question. It then runs a blame on the current version of the file, and scans the results for log lines that are from the same commit. If found, the relevant line is extracted. A pseudo-code version of this line tracking algorithm can be found in algorithm 3.3. This simple but effective way to resolve lines is able to detect lines that have been moved or indented. Any more complex changes, such as breaking up the call into multiple lines or changing the log message will result in the log line being untraceable. When this happens, no inlay is shown in the IDE.

This was considered an acceptable trade-off, as changing the log message will effectively result in a new message anyway. One other downside of this system is that it requires the programmer to use the Git VCS to version their project. Given the complexity of otherwise tracing log statements, as well as the ubiquity of Git within the current software development world, this too was deemed an acceptable trade-off. To ensure cross-platform support, the APIs exposed by the git4idea[11] IntelliJ plugin are used to invoke Git. These are created and updated by Jetbrains, so they are a stable way to call git methods. Future work on the plugin could introduce support for different versioning systems, such as Mercurial or Subversion.

Once the relevant line is retrieved, a `RangeHighlighter` is attached to the first functional character of the line. This allows us to offload the tracking of the current line position to IntelliJ. As long as the programmer does not delete the first character on the line (e.g. the `l` in `logger.info`), we can ensure that the inlay is always attached to the line by ensuring that its offset always matches the offset of the `RangeHighlighter`. Even if the programmer moves the line, the inlay will move accordingly. The `RangeHighlighter` additionally provides for an easy anchor point for the gutter icon.

The actual metrics shown in the editor are loaded once the editor instance is opened. They are loaded in the background, with careful rendering to ensure that the scrolling position of the editor does not change once the information is fetched and rendered. If needed, the metrics can be refreshed using an integrated refresh metrics action bound to a hotkey and a menu item inside the Tools menu, as seen in figure 3.11.

[11] https://github.com/JetBrains/intellij-community/tree/master/plugins/git4idea

Algorithm 3.3 The pseudo-code operation used to resolve historic log message locations to their relevant line in the current file.

```
1: function ResolveLine(file, old line number, old version hash)
2:     OriginBlameResult ← run
3:      git blame [old version hash] -L [old line] -M1 -C -C -w -n - [file]
4:     OriginCommit ← scan OriginBlameResult for old line number
5:
6:     if OriginCommit is not found then
7:         return −1
8:     end if
9:
10:    CurrentBlameResult ← run git blame -l -t -w - [file]
11:    CurrentLine ← scan CurrentBlameResult for OriginCommit and extract line
12:
13:    if CurrentLine is not found then
14:        return −1
15:    end if
16:
17:    return CurrentLine
18: end function
```

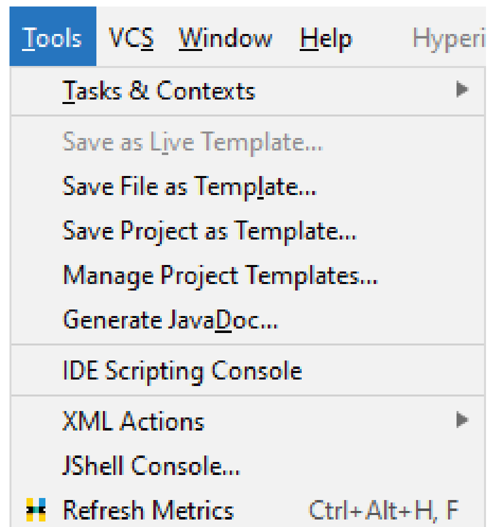


Figure 3.11: The 'Refresh Metrics' action, allowing the user to update all inline metrics in the currently opened editor.

3.5.2. Histogram Tool Window

For more details than just the number of log occurrences on a specific line, the plugin includes a tool window that is able to render histograms detailing the logging trends for a specific line, an entire file or the entire project over a configurable time interval. This functionality is similar to the histogram feature inside Kibana, shown in figure 3.12.

Since the IntelliJ platform uses Swing for UI, the choice of charting library was limited to Java libraries that supported the Swing platform. Options such as JFreeChart[12] and [13] were considered for this purpose, but due to the lack of interactivity they were not used. Instead, a custom charting library for histograms was built that allows for interactivity and scales automatically with the size of the tool window. This chart can be seen in figure 3.10.

Within the histogram, metrics are grouped by severity category. Each entry represents a single log message that was received and processed during the time interval rendered on the bottom of the chart. Each individual group, as well as each bar as a whole, is interactive and can be clicked to open a list of files and log lines that were triggered during that interval. An example of such a table can be seen in figure 3.13. Clicking on an entry in this table will automatically open the relevant file and navigate to the relevant line, allowing for easy navigation within the code base.

The visualization panel can either be opened directly by the user, or by using the gutter icon located next to an inline metric. If the user clicks the gutter icon, they are additionally offered options to visualize that specific line, the entire file, or the entire project. During visualization, the programmer can adjust the timeframe and the number of bars, as well as refresh the data.

The histogram view is mainly intended for developers to get a rough idea of how a certain log message trends over time. Given that a plugin implementation will almost always be inferior to dedicated tools such as Grafana or Kibana, the focus was instead put on integrating the visualized metrics tightly with the code, such that a developer can easily navigate the codebase and at a glance see the performance. Should a programmer need more detailed logging information, they can still reference the dedicated metrics interface their company uses.

Visualization: Histogram Related lines
Showing metrics for all files from 08:10:28 to 09:10:28

Path	File	Line Number	Severity	Trigger Count
src/main/java/com/sap/enterprise...	FailedResponse.java	11	ERROR	77
src/main/java/com/sap/enterprise...	TransportationService.java	11	INFO	185
src/main/java/com/sap/enterprise...	IntegerFactory.java	13	INFO	180
src/main/java/com/sap/enterprise...	ArithmeticAbstracter.java	24	INFO	179
src/main/java/com/sap/enterprise...	ArithmeticAbstracter.java	39	ERROR	152
src/main/java/com/sap/enterprise...	TransportationService.java	20	WARN	138
src/main/java/com/sap/enterprise...	HTTPResponseFactory.java	15	DEBUG	132

Figure 3.13: Tool window showing the relevant code lines for the clicked histogram bar. Clicking on an entry will navigate to the relevant code location automatically.

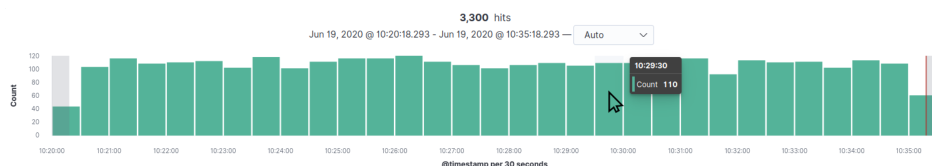


Figure 3.12: A histogram of logging metrics as used in Kibana. The user's cursor is shown to be hovering over a bar.

[12]http://www.jfree.org/jfreechart/
[13]http://jcckit.sourceforge.net/

3.6. Open-Sourcing Hyperion

Since the main intention behind the Hyperion project was to create an open-source adaptable version of the system described by Winter et al. [11], explicit focus was put on ensuring that all code as well as instructions were properly documented before a public release. This section will briefly discuss the efforts put into making sure that the product is of an acceptable quality and that users can begin using Hyperion for their own systems.

To ensure that developers can quickly get started with the Hyperion pipeline, compiled versions of all sub-projects are automatically released using the GitHub releases feature whenever a new tag is created. This allows a developer that simply wants to try out Hyperion to download the jar artifacts instead of having to first clone and compile the repository.

For users of docker-compose and tools like Kubernetes[14], Docker containers are also included for every sub-project. On a new tag, Docker Hub will automatically build these containers and tag them appropriately. For example, the container for version 0.1.0 of the adder plugin is located in the Docker Hub `sergdelft/hyperion:pipeline-plugins-adder-0.1.0` image. This ensures that companies can deploy pre-built images that are known to work remotely, without manually needing to create them.

For developers that wish to create a new plugin using the pipeline commons library, we have also published the library as a Maven artifact in the `com.github.serg-delft.hyperion:pipeline-common` namespace within Maven Central. This allows users of Maven and Gradle to add the library with a single line in their build configuration. Listing 3.5 shows an example of adding the library as a dependency to a new Gradle project.

Finally, a large effort was put into documenting not only the delivered code-base, but also including six different hands-on tutorials for developers to get started. Tutorials include information on how to extend the pipeline with custom plugins, detailed information on the protocol used by the plugins, as well as tutorials on how to get started with setting up a Hyperion pipeline. Each plugin is also individually documented with its full list of settings, an example usage of the plugin, and examples of the input and output formats of that specific plugin. In total, more than 10,000 words of accompanying documentation was written for the purpose of easier adoption by developers and companies.

Listing 3.5 The build configuration needed to add the pipeline commons library as a dependency to a new Gradle project.

```
dependencies {
    implementation "com.github.serg-delft.hyperion:pipeline-common:0.1.0"
}
```

[14] https://kubernetes.io/

4 Performance

One of the key requirements defined in the research report (see also section 2.3) is the ability for the Hyperion architecture to scale to a minimum of 50 billion log lines a month (RQ8). The composable architecture described in section 3.1 was also designed with this scalability in mind. In this section, we will evaluate the performance of the Hyperion pipeline and assert that it adheres to RQ8. The full results of the performed benchmarks can be found in appendix C.

Benchmarks were performed for a complete pipeline, as well as stress-tests for the maximum throughput of individual plugins. All benchmarks were performed on the DigitalOcean[1] cloud platform, using the `s-1vcpu-1gb` (small), `s-4vcpu-8gb` (large) and `s-8vcpu-32gb` (huge) machine types. These machines cost €5, €40, and €160 per month respectively. All machines were located in the same data center, to minimize networking delays.

The throughput value of 19,000 messages per second is used as the minimum requirement to satisfy RQ8. This roughly corresponds to a throughput of 50 billion log messages a month, assuming they are equally distributed.

4.1. Complete Pipeline Performance

In the first set of benchmarks, we evaluated the performance of an entire Hyperion pipeline located on a single logical machine. The reasoning behind this benchmark is to evaluate whether it is realistic to run the entire pipeline on a single machine without needing to resort to configuring deployment and distributed machine management for the pipeline. If the Hyperion pipeline is able to run at a decent performance rate on a single machine, this will drastically help adoption by making it easy to deploy a pipeline.

The benchmarked pipeline contains the following elements:

- **Stresser** - Generates fake log data.
- **Adder** - Adds the `project` tag to all incoming data.
- **Renamer** - Renames fields from the data source to their expected names.
- **Path Extractor** - Transforms Java package names into file paths.
- **Version Tracker** - Attaches version tags based on the latest Git commit.
- **Rate** - Receives final data and reports on the rate of arrival.

Effectively, the incoming data is transformed by four separate plugins. Since the size of the log data payload also determines the maximum throughput, benchmarks were performed for both 1000 bytes and 2000 bytes of incoming data per message. During development, we observed that the average log message from ElasticSearch was roughly 700 bytes in size. As such, we expect the performance of the pipeline with 2000 byte messages to be a good indicator of the worst-case scenario.

[1] https://digitalocean.com

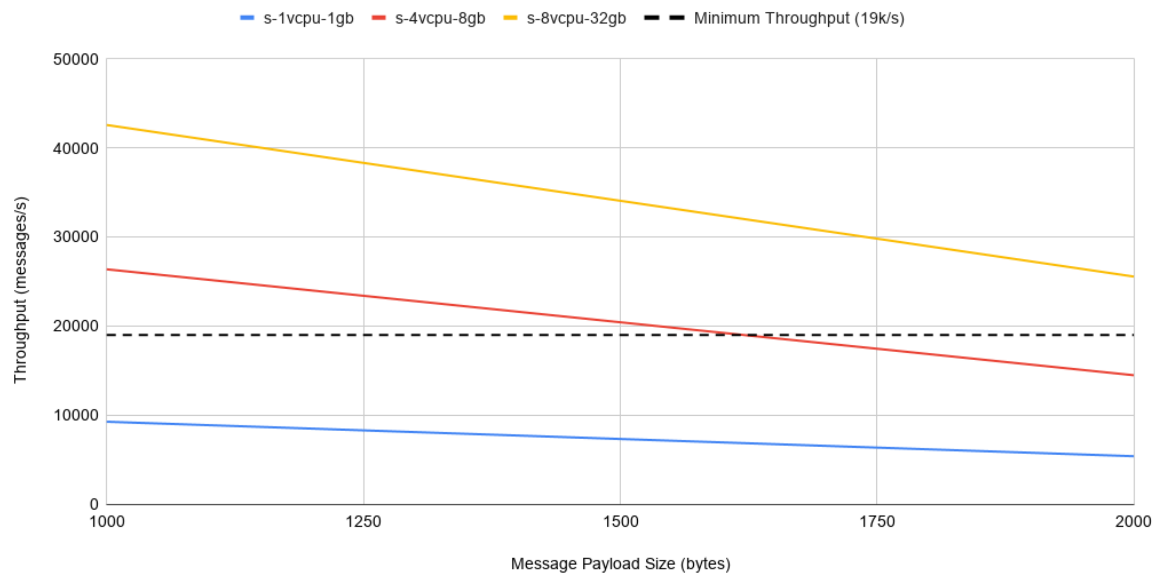


Figure 4.1: The performance of an entire pipeline contained on a single machine.

Message Size (bytes)	s-1vcpu-1gb (msgs/s)	s-4vcpu-8gb (msgs/s)	s-8vcpu-32gb (msgs/s)
1,000	9,261	26,366	42,562
2,000	5,392	14,481	25,552

Table 4.1: Raw message throughput of the Hyperion pipeline if all components are located on a single machine.

Figure 4.1 and table 4.1 show the results of the complete pipeline benchmarks. As can be seen, the `s-8vcpu-32gb` machine type satisfies the RQ8 minimum of 19,000 messages per second at all times, while the `s-4vcpu-8gb` machine is able to satisfy RQ8 for messages up to 1500 bytes in size. We therefore conclude that the Hyperion pipeline is able to conform to RQ8 even when all of the components are hosted on a single machine. The actual throughput depends on the machine hardware and the size of the log messages. Note that this result represents the worst-case scenario for a Hyperion pipeline: performance can massively increase if separate plugins are hosted on separate machines.

It should be mentioned that a setup in which every plugin is located on an individual `s-1vcpu-1gb` machine instance has a similar or better performance than hosting all plugins on a single `s-4vcpu-8gb` instance. Additionally, hosting each plugin on an individual machine would also be a cheaper solution (costing €30 a month, instead of the €40/month cost of a single `s-4vcpu-8gb` instance). The next section explores the performance of individual plugins on their own machine instance in more detail.

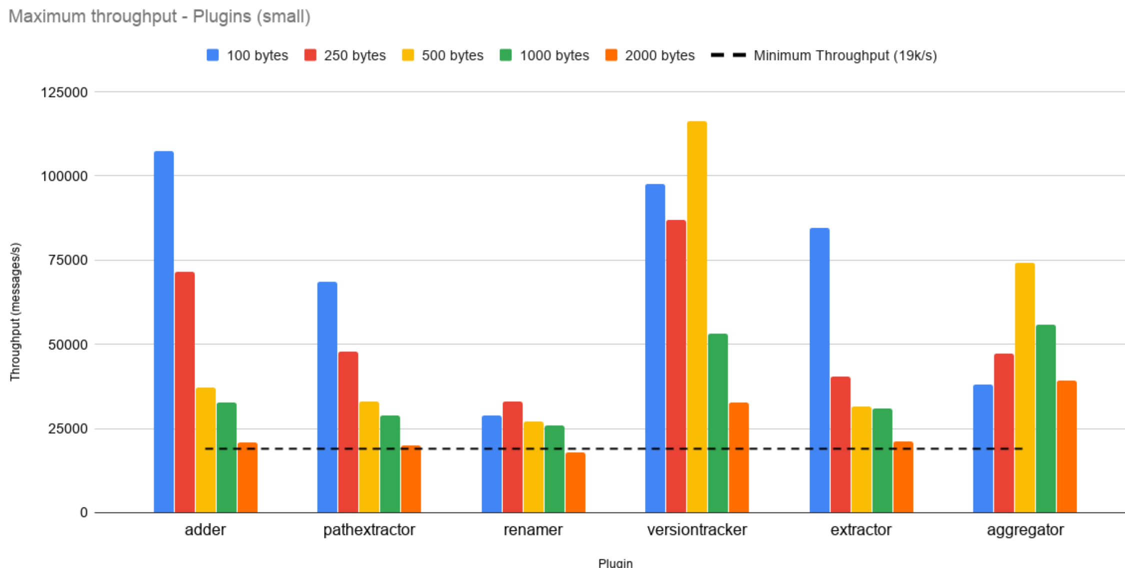


Figure 4.2: Maximum throughput of individual plugins in messages per second, as measured on a small (`s-1vcpu-1gb`) instance.

4.2. Individual Plugin Performance

Since the overall performance of the Hyperion pipeline is determined by the maximum throughput of the slowest plugin in the pipeline, we've also performed benchmarks of each individual plugin developed during the project. Unlike the full pipeline benchmark, these stress tests were performed with each component of the pipeline located on a different machine. These benchmarks should therefore represent the maximum performance achievable without resorting to load balancing. An additional benefit of running every plugin on its own machine is that a single plugin can be vertically scaled by giving it a more powerful machine, if needed.
The performance of a single plugin was evaluated using the following setup. Every component is located on its own machine.

- **Stresser** - Generates fake log data.
- **Plugin** - The plugin to be benchmarked.
- **Rate** - Receives final data and reports on the rate of arrival.

As with the full pipeline benchmark, payload sizes were varied to evaluate the performance of plugins with different payload sizes. For the individual tests, payload sizes of 100 bytes, 250 bytes, 500 bytes, 1000 bytes and 2000 bytes were used. Additionally, every plugin was benchmarked on both a small (`s-1vcpu-1gb`) instance and a large (`s-4vcpu-8gb`) instance. The adder, path extractor, renamer, version tracker, extractor and aggregator plugins were benchmarked. Values were averaged over two separate benchmarking sessions.

The result of the benchmarks can be seen in figures 4.2 and 4.3. The plugins individually confidently beat the target line of 19,000 messages/second on all but the 2,000 byte benchmark of the renamer plugin (which still scores an acceptable 17,998 messages/second). For most plugins, it is evident that a large machine is overkill and that if such a volume is needed it is smarter to instead load balance multiple plugin instances. It should additionally be noted that the renamer plugin was configured to rename three separate fields in the benchmark. Reducing the amount of renames or spreading them over multiple rename instances will also increase performance. For a full overview of the results, please see appendix C.

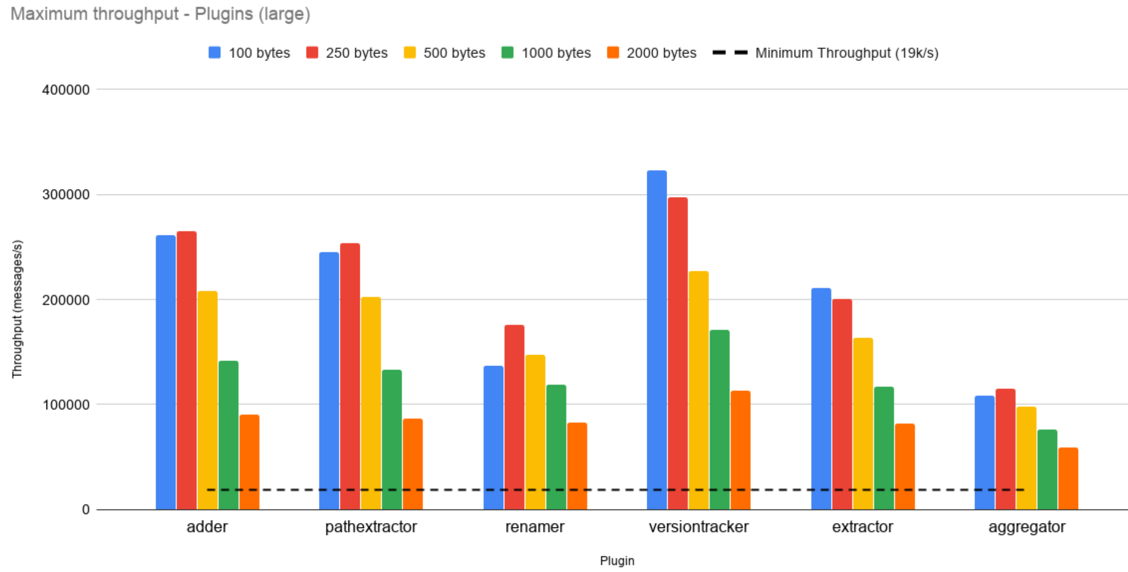


Figure 4.3: Maximum throughput of individual plugins in messages per second, as measured on a large (`s-4vcpu-8gb`) instance.

Based on these benchmark results, we conclude that the individual throughput of plugins is high enough where a pipeline can be constructed that easily beats the 19,000 messages/second threshold set by RQ8. Without load balancing, a pipeline that has a dedicated `s-4vcpu-8gb` instance for every plugin can easily reach over 100,000 messages per second. Should this not be enough, then the architecture of the Hyperion pipeline allows for infinite horizontal scaling by using the load balancer plugin.

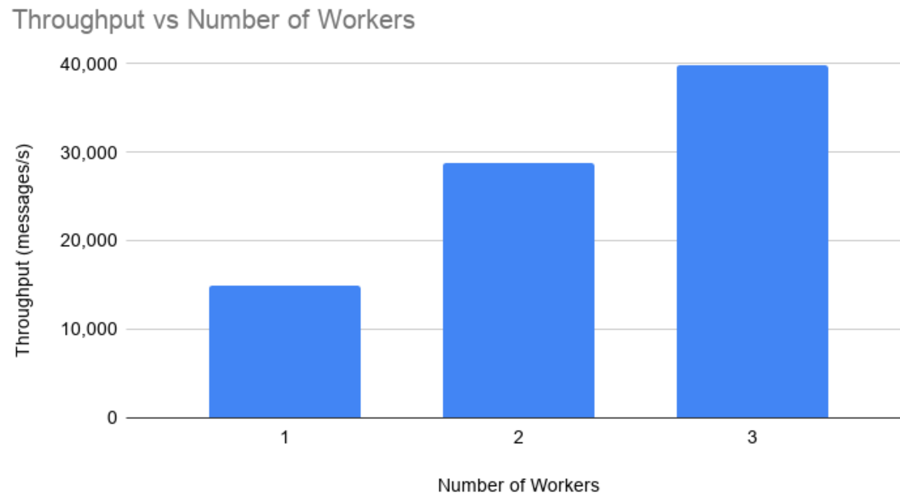


Figure 4.4: Changes in throughput when increasing the number of load-balanced workers.

4.3. Load Balancing

Finally, we've benchmarked the effect of using the load balancer plugin to distribute incoming messages over multiple instances of the same plugin. As target plugin, we've chosen to load balance the renamer plugin on a small `s-1vcpu-1gb`) instance, using a message size of 2,000 bytes. This plugin and machine combination was chosen as it was the slowest plugin in the individual plugin benchmarks.

The following configuration was used to benchmark the performance of the load balancer. Note that larger machines were used for the stresser and load balancer nodes, to ensure that any bottlenecks would be caused by the worker plugins and not by the load balancer. The results can be seen in figure 4.4.

- **Stresser** - Generates fake log data.
- **Load Balancer** - Evenly distributes data among workers.
  - **Renamer 1** - First renamer worker instance.
  - **Renamer 2** - Second renamer worker instance.
  - ...
- **Rate** - Receives final data and reports on the rate of arrival.

As can be seen from the figure, increasing the number of workers has an almost linear relation with the throughput of the entire pipeline step. This is to be expected, since the benchmark was specifically constructed in a way that the load balancer is not the bottleneck. We should mention that in such setups, the maximum bandwidth that can be provided by the network adapter of the load balancing node may become the bottleneck in the system. During benchmarking, we repeatedly ran into issues where the load was saturating a 2 Gbps network interface. This is also the reason why no fourth worker was benchmarked (as the network connection was saturated, barely causing an increase in throughput with additional workers).

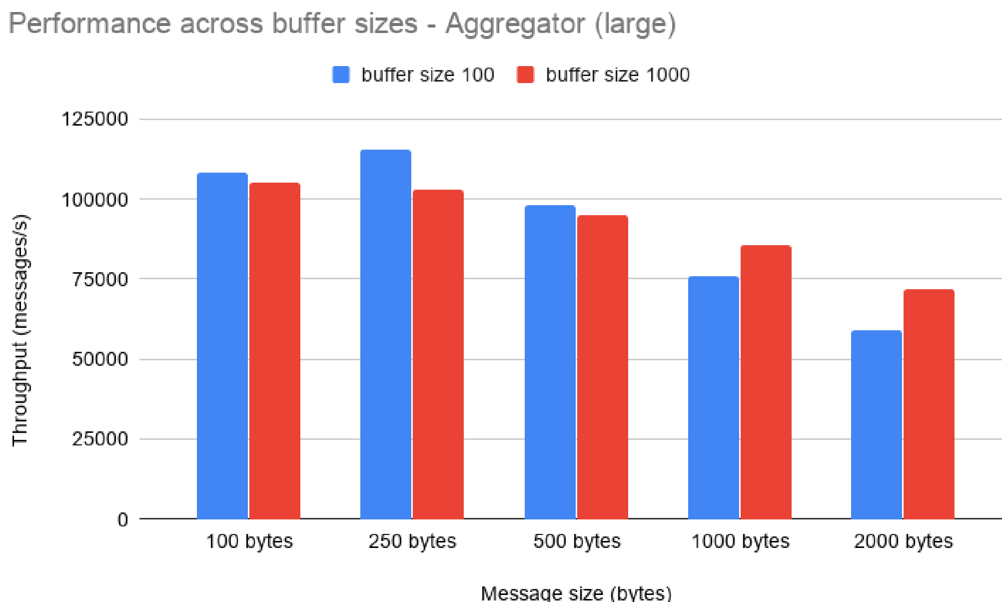


Figure 4.5: Differences in maximum throughput of the aggregator on a large `s-4vcpu-8gb` instance with varying buffer sizes.

4.4. Comparing Buffer Sizes

Each plugin that uses the pipeline commons library has an optional `buffer-size` property that defines the maximum amount of messages that a plugin may be processing at once. Currently, this value defaults to 20,000. During benchmarking, we've found that varying this property can result in significant changes in throughput. As an example, we've benchmarked the performance of the aggregator on a large `s-4vcpu-8gb` instance with buffer sizes of both `100` and `1000`. The results can be seen in figure 4.5.

As can be observed, a larger buffer size increases the performance of the aggregator for larger message sizes, at the cost of throughput for smaller messages. We've not observed a specific pattern for the performance when adjusting the buffer size, and we theorize this largely depends on the hardware the plugin runs on, as well as the type of work it performs and the type and size of the messages it processes. As such, we've not extended our benchmarking to varying buffer sizes for every plugin.

5 Ensuring Software Quality

An important part of any software project is the use of good development practices. As good as developers can be, the possibilities of errors still remains present. Because Hyperion is an open-source project, it is necessary that the quality of the software is up to standards. It is convention for open-source projects to employ methodologies such as automated tests and continuous integration to enforce certain standards. As such, Hyperion has also been developed using such techniques from the start.

This chapter will cover the techniques used to ensure software quality of Hyperion. The use of software testing and automated tools will be explained in section 5.1 and in section 5.2 subsequently. The source code was also inspected by the Software Improvement Group[1]. The results of this inspection and the changes made as a result are discussed in section 5.3.

5.1. Software Testing

Automated tests were heavily used during development, with most testing being unit and integration tests. Since the largest portion of the Hyperion pipeline is written in Kotlin, JUnit5 was chosen as a testing framework. For static analysis and linting, Detekt was used. These were the original choices for these libraries as discussed in the research report (see subsection D.4.3) and they have proven effective. The main challenge of testing both the back end pipeline and front end plugin was the use of external communication between components (e.g. via HTTP, ZMQ, etc). In order to test communication-based logic, the interfaces have to be simulated or mocked. For mocking purposes, the MockK[2] framework was used. Additionally, the Testcontainers[3] library was used to run Docker containers of other software which the Hyperion system interacts with (e.g. PostgreSQL and Elasticsearch). Using Testcontainers, we were able to create integration tests that automatically launch a database or other third-party system for testing purposes, emulating the real world as best as possible. The Logstash plugin was tested using RSPec[4], as it is written in Ruby and runs on the JRuby platform.

In total, the Hyperion code-base contains 333 tests, of which 7 are integration tests. A large majority of these tests are in the pipeline components, such as the pipeline commons library, the individual plugins, and the aggregator (collectively referred to as the back-end). In terms of coverage, a line coverage of 73% and branch coverage of 71% was achieved for the entire back-end (see Figure 5.1). Accounting for the IntelliJ plugin components as well results in an overall line coverage of 61% and branch coverage of 53%. This drop is due to most of the UI code being left as untested, as it was deemed not essential to test with respect to the time cost of the relatively short development period.

5.1.1. System Tests

Next to unit and integration testing, system tests were also used to validate the functionality of multiple components interfacing. The system tests use Docker and docker-compose[5] to easily start multiple components in a reproducible environment. Two of such system tests are published in the main repository: `elasticsearch-pipeline` and `raw-stresstest`. The prior is an end-to-end test which sets up an entire pipeline that transforms logs stored in an existing Elasticsearch instance to the suitable

[1] https://www.softwareimprovementgroup.com/
[2] https://mockk.io/
[3] https://github.com/testcontainers/testcontainers-java
[4] https://rspec.info/
[5] https://docs.docker.com/compose/

Hyperion

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
nl.tudelft.hyperion.pipeline		53%		48%	38	86	53	130	22	60	14	27
nl.tudelft.hyperion.pipeline.loadbalancer		67%		65%	17	56	27	110	8	36	6	18
nl.tudelft.hyperion.pipeline.plugins.reader		33%		50%	10	19	25	38	7	15	6	10
nl.tudelft.hyperion.pipeline.versiontracker		76%		72%	23	78	27	134	15	56	8	22
nl.tudelft.hyperion.aggregator		51%		75%	14	33	33	61	12	25	9	13
nl.tudelft.hyperion.datasource.plugins.elasticsearch		80%		96%	16	90	26	148	15	75	10	28
nl.tudelft.hyperion.pipeline.pathextractor		54%		100%	6	15	18	38	6	14	5	8
nl.tudelft.hyperion.pipeline.plugins.stresser		59%		50%	11	28	25	49	6	21	5	12
nl.tudelft.hyperion.pipeline.plugins.rate		57%		25%	10	21	15	38	8	19	7	10
nl.tudelft.hyperion.pipeline.extractor		76%		75%	14	38	19	72	7	22	5	11
nl.tudelft.hyperion.pipeline.plugins.adder		78%		83%	7	40	8	60	5	31	4	17
nl.tudelft.hyperion.pipeline.renamer		64%		75%	6	19	17	46	5	17	4	11
nl.tudelft.hyperion.pipeline.plugins.printer		23%		50%	5	8	16	21	4	7	4	5
nl.tudelft.hyperion.pluginmanager		90%		82%	13	59	11	99	6	39	6	21
nl.tudelft.hyperion.aggregator.workers		94%		84%	10	68	4	135	6	55	4	35
nl.tudelft.hyperion.aggregator.api		96%		100%	8	40	8	121	8	39	0	11
nl.tudelft.hyperion.pipeline.connection		86%		100%	6	19	0	37	6	18	6	12
nl.tudelft.hyperion.aggregator.intake		84%		75%	5	15	6	31	4	11	1	6
nl.tudelft.hyperion.aggregator.database		98%		n/a	2	17	0	24	2	17	1	6
nl.tudelft.hyperion.datasource.common		100%		n/a	0	2	0	3	0	2	0	1
Total	2,855 of 10,909	73%	97 of 339	71%	221	751	338	1,395	152	579	105	284

Figure 5.1: The JaCoCo coverage report of the pipeline. A 73% instruction coverage and a 71% branch coverage rate was achieved.

format and aggregates them. `raw-stresstest` starts up two plugins and a plugin manager: the `stresser` and `rate` plugin. This system test was only used to calculate raw throughput of messages, asserting the throughput of ZMQ communication when no transformation or parsing is happening.

5.2. CI/CD

All tests described in the previous section are configured to automatically run on every relevant commit using GitHub's Continuous Integration (CI) implementation, GitHub actions[6]. GitHub actions operate using workflows, which are automated processes configured using a special workflow file. Each Hyperion module has a separate workflow file that executes all checks for that module, allowing tests to be run independently for each module, as well as running them concurrently during a full test of the entire code-base. The Kotlin-based modules use a Gradle action to automatically run the test suites, while the Logstash plugin uses the official GitHub `setup-ruby` action.

To ensure that tests are only run when the relevant code is updated, the pipelines are configured to only run their actions when a Pull Request (PR) is made. Within pull requests, the pipelines are additionally configured to only run if the request makes changes to their project. This ensures that a pull request does not test all sub-components unless every component has been changed. In contrast, all commits on the default branch are always tested. This is to assert that the default branch always has passing tests.

Additionally, all test-based actions were set up to utilise caching to skip the installation of dependencies if no dependencies were changed. This drastically speeds up testing times, as well as prevent wasted network and time resources.

5.3. Software Improvement Group

In an effort to ensure good coding practices and to make sure that Hyperion as an open source project stays maintainable, the source code was inspected by the Software Improvement Group (SIG) on two separate occasions. The software maintainability model used by SIG rates a codebase from 1 to 5 stars, where the worst system relative to their benchmark achieves one star and the best achieves 5 stars. This section will discuss the two inspections and the changes made in response to the issues highlighted by SIG.

[6]https://github.com/features/actions

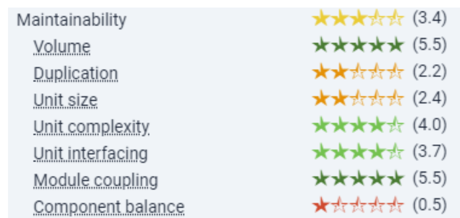


Figure 5.2: Hyperion's maintainability score according to the Software Improvement Group after six weeks of development.

5.3.1. First Inspection

The first inspection was performed after six weeks of development. In this inspection, Hyperion scored 3.4 stars on the SIG maintainability model scale. This main score was derived from seven separate sub-scores, which can be seen in figure 5.2. The volume of code, unit complexity, unit interfacing and module coupling statistics rate excellently while especially the unit size and complexity metrics are sub-optimal.

We believe the component balance score, on which we scored a meager 0.5 stars, to be largely invalid for our codebase. Component balance is the relative size of one component in Hyperion compared to others. Since Hyperion uses a monorepo[7], a variety in component sizes between components is not unusual. Since refactoring the modular pipeline infrastructure did not seem necessary and would have required a lot of resources, we did not spend effort improving the component balance score.

As a response to the first inspection, we largely focused our efforts into improving the code duplication, unit size and unit complexity metrics. This was done by identifying sources of complexity and code re-use, and refactoring or removing the offending code. In some cases, we adjusted our Detekt code linter configuration to flag methods that did not adhere to SIG guidelines. It is important to note that these refactors were only performed where the alternative was sensible and the deviation from recommended values was large enough to warrant a refactor. The next subsections will describe some of the larger changes done in response to the SIG feedback.

Improving Code Duplication

The largest source of code duplication within the repository was caused by the `build.gradle.kts` files used by the build system. Due to the large amount of individual packages in the repository, most of which used similar dependencies, a lot of these build configurations shared similar or identical lines. This issue was resolved by using the `buildSrc` feature in Gradle, which allows for a library to be used in the build scripts. This meant that we were able to extract common patterns (such as the configuration of the code coverage) into helper functions that could then be invoked in individual build files. As an added benefit, we used the refactoring opportunity to create a centralized configuration file for all dependency versions used in the project. This allows us to use a single unified version of a library across all projects, without the need to update every build file individually. Overall, this refactor removed roughly 400 lines of duplicated code, as can be seen in pull request #93[8].

Similarly, we discovered that some functionality was duplicated by a lot of pipeline plugins. Since every pipeline plugin already depends on the pipeline commons library, we decided to move this common functionality into the central commons library. This allowed us to maintain a single version of the utility code that we could ensure was tested probably. Examples of code extracted into the commons library were functions that deal with parsing and manipulating incoming JSON messages.

Finally, we adjusted the pipeline commons library to allow for plugins that could only send or receive (previously, only plugins that both sent and received messages were supported). This allowed us to use the library in the data source and aggregator, which removed more duplicated logic for receiving messages and communicating with the plugin manager. This change resulted in an increase in functionality while at the same time removing roughly 200 lines of duplicated functionality, as seen in pull

[7]A single git repository that contains multiple independent projects.
[8]https://github.com/serg-delft/hyperion/pull/93

request #91[9].

Improving Unit Size
The main entry-point in the Elasticsearch data source plugin was a monolithic function that did multiple steps in a sequence. To reduce the unit size, we refactored the function into separate functions. The final version of `Elasticsearch.run()` is now below the recommended length of 15 lines. A similar transformation was performed with the main function of the load balancer plugin.

One other major violation of the recommended unit size was in the logic for deserializing the Elasticsearch data source configuration. The original iteration of this code had a unit size of more than 30, largely because it was responsible for parsing two separate types of configuration. The complexity was easily reduced by extracting these separate configuration types into their own functions. The final result has a unit size of 13.

Improving Unit Interfacing
Most of the unit interfacing violations were caused by complex functions that had a large number of arguments. Where appropriate, we've reduced the number of parameters.

The `createSearchRequest` function in the Elasticsearch plugin had a large amount of parameters as it was meant to be used as a factory for creating API calls to Elasticsearch. The API calls require multiple parameters which could be logically grouped into a single object, so we extracted the parameters out of the function and into a new class which represented API call parameters. This refactor reduced the number of arguments to a single object.

Additionally, we configured Detekt to flag functions with a large number of arguments. This led to us refactoring parts of the plugin code, as it was still work-in-progress and hadn't been submitted in the first SIG review. This allowed us to preemptively avoid running into unit interfacing violations during the second inspection.

5.3.2. Second Inspection

The second code submission was done at the end of week 8. It included the changes described in the previous section, as well as new code that was written between the first and second submission. The results of the second evaluation can be found in figure 5.3. As expected, our overall maintainability score increased due to the refactors performed. Overall, we matched or improved our score in all categories except module coupling and unit interfacing. We think this can be explained by new complexity being introduced after our first submission.

Looking at the other dimensions, our significant improvements were on code duplication and unit size. This means that the refactors performed were successful at improving the long-term maintainability of the Hyperion source code. It should be noted that not all of the highlighted violations from the first inspection were adjusted: some code paths only slightly violated the recommended values and splitting them up would have decreased overall code clarity. This may explain our reduced score when it comes to unit interfacing.

Maintainability	★★★★☆	(3,8)	▲ 0,36
Volume	★★★★★	(5,5)	= 0,00
Duplication	★★★★★	(4,9)	▲ 2,72
Unit size	★★★★☆	(3,8)	▲ 1,39
Unit complexity	★★★★☆	(4,3)	▲ 0,27
Unit interfacing	★★★☆☆	(2,7)	▼ 1,05
Module coupling	★★★★★	(4,7)	▼ 0,72
Component balance	★☆☆☆☆	(0,5)	= 0,00

Figure 5.3: Hyperion's maintainability score according to the Software Improvement Group after eight weeks of development and two weeks of improvement on the first assessment.

[9] https://github.com/serg-delft/hyperion/pull/91

6 Process

In this chapter, we will briefly reflect on the process of developing Hyperion and discuss our perspective on the bachelor project as a whole. We also discuss the effect of the COVID-19 pandemic on the project.

6.1. Group Dynamics

Because of the COVID-19 pandemic, all physical on-site education was cancelled until at least the end of the academic year. Due to this, almost the entirety of the project was done completely remote. Tools such as Discord and Zoom were used to periodically meet with all team members and with the client and supervisor.

Initially, we were afraid that working remotely may have a compromising effect on motivation and that it would become harder to have a unified vision for the target product. To somewhat alleviate these issues, we constructed a meeting schedule that contained at least two meetings every day: one at 9.00 in the morning and one at 15.00 in the afternoon. These meetings ensured that issues could regularly be discussed and that people were motivated to work on the project throughout the day. Usually, the morning meeting was used to divide work for that day, the results of which were then discussed in the afternoon meeting. Even though it was unfortunate to lose part of the team experience, this schedule really helped us to keep pace.

All of the team members acted as both a developer and code reviewer of Hyperion during this project. Thijs was the main designer for the overall pipeline architecture, as well as the lead programmer on the aggregator component. Nick was the lead programmer on the data source plugins, as well as the main contributor for the histogram visualization features of the IDE plugin. Maarten was responsible for all other aspects of the IDE plugin, including inline visualization and line resolving. Ynze was the lead programmer on most pipeline plugins, as well as a significant contributor to the documentation and the final report. Finally, David was in charge of all developer tooling, test setup and deployment, as well as contributing to several smaller pipeline plugins.

It was strange to do such a large project for more than 2 months without ever seeing a teammate in person. We do realise however that we had the advantage of already having done a range of projects with the same team, so we did have to get acquainted over a digital voice channel. The group dynamic itself has been just as good as in the other projects, but where we normally might go off topic and joke around it felt more serious and down to business doing the project remotely. This likely positively affected our productivity, but it may have come at the cost of some enjoyment.

6.2. Communication with the Coach and Client

Next to meetings within the team, meetings were also held between the team and the coach and client. Multiple client meetings were held at the start of the project to work out the exact problem and requirements of the project. As the project progressed, the scheduled client meetings were changed to an on-need basis. This meant that when a new version of Hyperion was developed and feedback was necessary, or if a problem was encountered which required specific insight which the client had hands-on knowledge on, a meeting was scheduled on the next available opportunity.

Additionally, a meeting was held between the Hyperion group and a group from Jetbrains. The Jetbrains team was primarily interested in what problem the project was attempting to solve but also some im-

plementation details of the pipeline. Ultimately, the team of Jetbrains showed interest in the possibility of integrating Hyperion into an internal system in Jetbrains.

6.3. Development Methodology

As originally discussed in the research report, we used the Kanban methodology to keep track of our issues, using the GitHub projects feature as a Kanban board (see section D.4.2 of the research report for the justification behind this choice). Even though we did not have sprints that strictly opened at the start of the week and finished at the end, we attempted to roughly divide sprints up in weeks. The project in its entirety had a duration of 10 weeks, with each its own sprint respectively. Unlike classical sprints, we decided to review issues whenever they were finished instead of centrally at the end of the week. This allowed us to be more flexible when issues took longer or shorter than expected, and allowed us to quickly iterate when changes were requested without the need to wait for the next sprint. Despite this, we still attempted to integrate the current version of the project at the end of the week to create a coherent and stable version to be merged into the default branch. Usually, demos for the client and supervisor were also done on Fridays for this reason.

In total, the first two sprints were allocated to researching and writing the research report. The next six sprints were dedicated completely to writing the project. The final two sprints were mainly used for writing the final report, although various small code changes were also created, reviewed and applied.

6.4. Software Quality

Outside of using linting and tests to ensure software quality (see also chapter 5), we also required every change to go through GitHub pull requests. This allowed us to read changes and give feedback before they were merged into the default branch. During development, almost 400 of such comments were left on pull requests, discussing the implementation and proposing changes to the code. In total, 72 pull requests were reviewed and accepted during development.

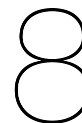
7 Ethics

As part of the development process, we've considered the ethical implementations of an integrated logging aggregation and visualization system like Hyperion. Even though it is a tool with a straight-forward set of features and seemingly without any ethical or moral dilemmas, there are a few points we briefly want to discuss as part as this report. We will open a number of ethical discussions, even if they are just for the sake of consideration.

First of all, Hyperion is an open source project. The ethical question that is often raised when it comes to open source software is whether it is safe and ethical to publish code along with its possible weaknesses. Since anyone can use Hyperion freely there is a possibility of it being used with confidential information. An example in our case could be log information that exposes system vulnerabilities. Simultaneously, everyone can see the source code and possibly find ways to exploit the system to obtain the confidential information. This discussion is obviously a universal discussion and not one specific to Hyperion, but it is worthwhile to think about the implications of creating open source software nonetheless.

Another discussion concerning flexibility versus safety is extensions of our product. Given the modular nature of the Hyperion pipeline, anyone is able to extend the system with their own processing plugins. It may seem like these extensions are outside of the scope of our responsibility as Hyperion, but this is not necessarily the case. Let's take the hypothetical scenario of a company creating their own custom pipeline plugin to account for a certain unconventional log setup; this plugin could somehow be faulty and drop log information that would otherwise have helped developers in identifying system bugs that could play a very important role. That scenario would raise a new discussion on indirect responsibility regarding poor documentation and the abstract functionality written by us.

The previous example is not the only one in which important log information could be lost in the use of our tool. If the pipeline or another component of the system is not set up properly because of poor documentation, it is arguable that there is an indirect responsibility for the authors of the documentation as well.



Discussion

In this chapter, we will discuss the results of the project and compare them to the requirements formally defined in section 2.3. We will attempt to quantify the results of the project against these requirements to conclude the effectiveness of the delivered product. We also briefly discuss potential improvements for the product, which are further expanded upon in chapter 9.

8.1. Verification

In this section, we will refer back to the initial functional requirements and evaluate whether they have been fulfilled in the delivered product. For requirements not fulfilled, we will justify why and provide recommendations for future work to complete them. We discuss each requirement as grouped with its MoSCoW severity, as initially introduced in the research report (see also section D.2.3).

'Must' Requirements

RQ1-6 were marked as the highest priority, indicating that a successful resulting product needs to adhere to all of these requirements. Within these requirements, RQ1, RQ4, and RQ6 are directly satisfied by the IntelliJ IDEA plugin created. The plugin supports all languages and is able to show inline metrics.

RQ2 and RQ3 are satisfied with the Elasticsearch data source plugin and the aggregator respectively. A direct HTTP API exposes a querying endpoint for front-end implementations such as the IntelliJ plugin, while the Elasticsearch data source plugin allows Hyperion to interact with the Elastic stack. Additionally, a Logstash plugin was created to allow interfacing with Elastic stacks that also employ Logstash.

RQ5 was not directly satisfied by the final product. The initial motivation behind this requirement was that performing log tracing server-side would allow for lightweight front-end implementations. However, we failed to account for local changes to files by programmers. This necessitates that log tracing is done client-side. In order to somewhat ease the efforts of implementing a front-end extension to Hyperion, extra documentation was written that explains the process of log tracing and how one can implement it in their own frontend.

'Should' Requirements

RQ7, RQ8, and RQ9 were marked as a medium priority, indicating that while they are not essential for a working product, their presence would highly improve the overall value of the project. RQ8 has been extensively discussed in chapter 4. Given the benchmarking results, we conclude that the delivered product easily handles the target metric of 50 billion log lines a month, often being able to handle more than 50,000 messages per second on commodity hardware.

RQ7 is easily satisfied by increasing the `time-to-live` setting in the aggregator. While not recommended (due to increasing storage sizes), the aggregator has no technical maximum statistic lifetime and can therefore store metrics indefinitely.

RQ9 was not directly satisfied by the final product. The initial motivation behind this requirement was that an efficient pipeline would not need to re-aggregate statistics already available in a database such as Elasticsearch. The aggregator could instead defer this computation to the original data source. During implementation however, it turned out that this would require a significant split in responsibilities

for the aggregator, as well as limit the number of logging setups supported. Deferring computation to Elasticsearch additionally was significantly slower than the aggregation strategy currently in use by the aggregator. Instead, extra efforts were put into the pipeline to ensure that companies that already have a metrics storage can easily start using Hyperion.

'Could' Requirements

Finally, we discuss requirements RQ10-14. These were optional requirements that we could consider if time allowed. With most of these, we briefly looked at their possibilities to see if they were doable within the given timeframe of 10 weeks.

RQ10 ended up being the core of the Hyperion pipeline design. Even early in the project it was clear that to enable the flexibility needed to support a large amount of logging stacks, a modular pipeline would need to be the main feature of Hyperion. Needless to say, this requirement has been easily satisfied through the more than 10 different pipeline plugins shipped with the final version of the project.

RQ11 was not shipped. The complexity of implementing a stable IDE plugin is too high to warrant the effort of creating plugins for other IDEs. Instead, extra effort was put into documenting the steps needed to implement an IDE extension for Hyperion. The intention is that if other developers are interested in adding Hyperion support to their favorite IDE, the documentation written by us allows them to do so.

RQ12 ended up being shipped as part of the reference IntelliJ IDEA plugin. Some of the project members had extra time left and ended up allocating this time towards improving the visualization features of the plugin. Section 3.5 discusses the charting feature and its implementation in more detail.

RQ13 was briefly considered, but eventually not implemented due to time constraints. However, the pipeline was designed with a potential implementation of heuristic log tracing in mind, and as such it has full support for all the features needed by such an implementation. We believe that with relatively little effort, the existing implementation by Schipper et al. [10] can be adapted to function as a part of the Hyperion pipeline.

RQ14 ended up severely clashing with the original design of the pipeline. Given that exception tracing requires transaction tracking within the relevant application, additional data would be needed to track the flow of these messages and recognize when they were involved in an exception. The problem analysis section (see section 2.2) discusses the key differences between these types of logs in more detail. We do believe that support can be added to Hyperion, but that this would require a non-trivial refactor in a number of projects. It was therefore considered out-of-scope for the delivered product.

8.2. Validation

While we have discussed how Hyperion fulfills the requirements set at the start of the project, this does not directly translate to actually having a useful product. In this section, we briefly discuss the actual quantifiable utility of our product and whether it meets validation criteria.

The main goal of the bachelor's project was to create an open-source implementation of the system described by Winter et al. [11]. Given that Winter et al. have previously quantified the actual use of such an integrated metrics system, we believe that those benefits also translate to any system that has comparable features. Therefore, if the original system was shown to be useful to developers, and Hyperion fulfills all the requirements of the original system, then we can reasonably assume that Hyperion has comparable use to developers.

As previously discussed, we believe that Hyperion does indeed qualify as a suitable open-source version of the discussed system. Therefore, we believe that the delivered implementation has quantifiable use to developers.

Additionally, we have personally used the system during testing over the course of the project. During this development process, we have made various changes to make the plugin or pipeline easier to use and more useful to developers. Unfortunately none of us have a project that produces enough logging data for a system such as Hyperion to provide real-world benefits, but we are all optimistic for the help that Hyperion can bring to software development.

9 Future Work

This chapter will discuss future work that can by done on the Hyperion pipeline and plugin implementations. We consider both individual improvements to components, as well as large-scale evaluations of the tool as a whole. We do not intend for this to be an exhaustive list, but rather a set of features that were considered but never completed due to time constraints.

9.1. Extend IDE Support

The plugin built for IntelliJ IDEA was explicitly designed to be language-agnostic. This allows for the plugin to support any language supported by the IDE itself. Given that most of the Jetbrains-family IDEs are based on the original IntellIJ implementation, this also means that Hyperion supports other IDEs such as WebStorm, PHPStorm, PyCharm, etc. While these IDEs cover a relatively large amount of languages, they all require the use of an IntelliJ-based IDE.

Future Hyperion work could include adding support for other popular text editor and IDEs. Text editors such as Visual Studio Code[1] are becoming increasingly popular and provide plugin APIs for third-party extensions. We believe that Hyperion could benefit from increasing support, as this drastically lowers the bar for adoption.

9.2. Extend Default Pipeline Plugins

While the delivered product comes with a large set of pipeline plugins, they only perform elementary transformations. We believe that Hyperion could benefit from adding more specialized plugins for common logging scenarios. The full scope of what plugins could be useful would likely require an empirical study of logging setups in order to detect common logging setups currently not supported by Hyperion.

One example of a pipeline plugin that could be created is the heuristic log tracing discussed by Schipper et al. [10] and documented in RQ13. We believe that such an implementation could be a useful extension to the Hyperion pipeline, as it allows companies to start using Hyperion without the relatively expensive cost of tracing log locations whenever a message is logged.

9.3. Invidual Plugin Improvements

As discussed in various sections, there are a large number of small improvements that can be made to the plugins currently shipped with the Hyperion pipeline. A non-exhaustive list includes adding authentication support for the aggregator, splitting the load balancer machine up into multiple steps, adding compression support to the pipeline transport, and implementing an in-memory cache for the aggregator. These improvements are discussed in more detail in the appropriate sections of chapter 3.

9.4. Emperical Testing

Finally, we believe that the Hyperion tool could benefit extremely from an empirical test in a real-world scenario. Integrating the system into an existing company or organization, similar to the tests done at Adyen by Winter et al. [11]. Such tests could highlight painpoints during regular Hyperion usage and show whether or not the system is stable and efficient in a real-world scenario.

[1] https://code.visualstudio.com/

10
Conclusion

The Hyperion project implements a flexible logging pipeline that brings log metrics directly to the IDE. It comes with a large range of configurable plugins that are not only for transformation but also for debugging and load balancing. Extensive support is included for custom pipeline steps, allowing any company to implement Hyperion in even the most unconventional logging setups.

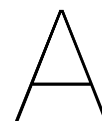
Hyperion scales horizontally far beyond the target rate of 50 billion requests a second, with plugins easily reaching more than 100,000 messages handled per second on commodity hardware. When even that is not sufficient, the built-in load balancer plugin allows any component of the pipeline to scale linearly with the number of worker instances.

To prevent the Hyperion pipeline from being cumbersome to setup and configure, there is extensive documentation on setup as well as automatically built Docker images for each individual component and automatic releases of the standalone executable jar files for other deployment methods. Each component additionally has a dedicated documentation and automatic tests, with a test-code ratio of 1.5 for the entire project.

The provided IntelliJ IDEA plugin implementation features automatic inline metrics that resolve even across code versions and edits by the developer. Additional charts showing the logging trends of lines, files, and even the entire project are just a single click away and can directly link back to the lines responsible for the log message. For other IDE support, extensive documentation is provided on how to get started with the aggregator API and how to implement the most common operations used during IDE visualization.

All development was done completely remote using a Kanban-style development methodology, with code changes being proposed through GitHub pull requests. Daily meetings were held by the team, with several demos and regular updates sent to both the client and the project supervisor. All proposed code changes were reviewed by the team, resulting in almost 400 comments left on pull requests to improve the overall quality of the project.

Overall, we are very happy with the quality of the delivered product. Hyperion either directly fulfills all set requirements, or provides a reasonable alternative for them. The resulting software is similar to the project described by Winter et al., but it is far more flexible and fully open-source. Since this was the main aim of the project, we are satisfied with the result.



Info Sheet

Monitoring-aware IDEs
Presentation date: 1 July 2020
GitHub page: https://github.com/SERG-Delft/hyperion
Contact: M.F. Aniche, M.FinavaroAniche@tudelft.nl
Description:
Recent research has shown that integrating production metrics directly inside the development environment of programmers can lead to increased understanding of the system and overall code improvement. We were challenged by our client to create an open-source implementation of a logging pipeline that is able to integrate with existing logging solutions to provide a system that can ingest, process, and visualize logging behavior directly in the IDE. Through a fully-remote Kanban workflow, a fully modular pipeline was created that fits in every logging setup. The hardest challenge, creating a system that scales to over 50 billion log lines a month, was easily achieved by using the research report to design a horizontally scaling architecture specifically optimized for throughput. Several third-party companies and developers have already shown interest in the resulting product, Hyperion.

Project members:
Thijs Molendijk
Interests: Programming Languages, Reverse Engineering
Main contributions: Aggregator, pipeline protocol

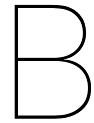
David Moolenaar
Interests: Distributed systems and their protocols
Main contributions: Build tools and testing, pipeline plugins

Nick Yu
Interests: Compilers and Interpreters, Data Science
Main contributions: Pipeline plugins, plugin metrics visualization

Maarten Lips
Interests: Security, Computer Graphics, Data Science
Main contributions: IntelliJ plugin

Ynze ter Horst
Interests: Artificial Intelligence, Big Data
Main contributions: Pipeline plugins

Client & Coach
Coach: M. F. Aniche, SERG Group, TU Delft
Client: Ir. J. Winter, Adyen



Project Description

The main goal of the project is to create a direct integration between monitoring tools and the IDE, in order to create an instant feedback cycle between the source code and its actual behavior running in production. The paper by Jos Winter et. al explores this concept in more detail.

More concretely, this will work by implementing a modular system that consists of components roughly divisible into four different tasks:

- Metrics intake (receive log data from an existing monitoring stack)
- Metric preprocessing (extract relevant information such as line number from log data)
- Metric aggregation (process this log data to create aggregated statistics)
- Metric visualization (display the aggregated data to the programmer)

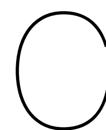
The client has set the following properties to be required for a minimal viable product:

- Metrics intake: support for an ELK stack (integration with ElasticSearch).
- Metric preprocessing: support for log lines with line numbers and file & class names.
- Metric aggregation: scalable to up to 50b lines/month.
- Metric visualization: support for the IntelliJ IDEA IDE.

Besides implementing this integration, we will also test the tool created. This means we need to create a testing environment with artificial log lines to test the scalability of our implementation. We will implement unit tests for the tool, the coverage percentage will be specified by the research report.

We aim to document the code in such a manner that future developers will be able to understand the code and improve on our product. The specific syntax and tools used for this purpose will be specified by the research project.

If time allows, the product will be extended to also allow inferring log locations by combining the source code and log messages, as used in the original Adyen implementation of this system. Other possible features include support for different IDEs and supporting log intake from different logging systems. The research report will research these opportunities in more detail.



Benchmark Results

This appendix documents the raw results of the various benchmarking runs performed. For more details on the benchmarking setup as well as an overview of the results, please see chapter 4. Within this appendix, machine types of small, large and huge refer to DigitalOcean instances of sizes `s-1vcpu-1gb`, `s-4vcpu-8gb`, and `s-8vcpu-32gb` respectively.

Entire pipeline on a single machine

Benchmarked with a pipeline consisting of a stresser, adder, renamer, path extractor, version tracker, and rate plugin, all located on the same machine. Message size was varied with 1,000 bytes and 2,000 bytes.

Message Size (bytes)	s-1vcpu-1gb (msgs/s)	s-4vcpu-8gb (msgs/s)	s-8vcpu-32gb (msgs/s)
1,000	9,261	26,366	42,562
2,000	5,392	14,481	25,552

Performance when varying buffer sizes

Benchmarked with an aggregator running on a large machine instance. Benchmarks were ran for a `buffer-size` of 100 and 1000. Results are in messages per second.

	100 bytes	250 bytes	500 bytes	1000 bytes	2000 bytes
buffer size 100	108,130	115,614	97,908	75,958	58761
buffer size 1000	105,131	102,824	94,917	85,677	71890

Throughput of load balancer with varying number of workers

Benchmarked with 1-3 renamer worker instances running on individual small machines. A message size of 2,000 bytes was used.

Number of Workers	Throughput (messages/s)
1	14,880
2	28,771
3	39,807

Individual plugin benchmarking

Every plugin was benchmarked twice on both a small and large instance. For the report, the average value of the two runs was used. Message size was varied between 100, 250, 500, 1000, and 2000 bytes.

	100 bytes	250 bytes	500 bytes	1000 bytes	2000 bytes
adder-small1	110,995	63,547	35,952	34,055	21,366
adder-small2	103,764	79,290	38,642	31,504	20,604
adder-small-avg	107,380	71,419	37,297	32,780	20,985
adder-large1	264,496	262,213	221,949	139,653	86,371
adder-large2	257,620	267,972	193,726	143,621	93,766
adder-large-avg	261,058	265,093	207,838	141,637	90,069
pathextractor-small1	69,299	47,866	31,959	28,746	20,360
pathextractor-small2	68,100	48,040	34,209	28,995	19,855
pathextractor-small-avg	68,700	47,953	33,084	28,871	20,108
pathextractor-large1	244,543	247,971	186,702	129,554	89,018
pathextractor-large2	246,172	259,837	217,464	137,375	84,973
pathextractor-large-avg	245,358	253,904	202,083	133,465	86,996
renamer-small1	30,593	33,105	26,998	25,465	16,214
renamer-small2	26,961	32,950	27,204	26,620	19,783
renamer-small-avg	28,777	33,028	27,101	26,043	17,999
renamer-large1	138,914	167,352	144,400	118,246	81,895
renamer-large2	135,132	185,192	150,072	119,143	83,109
renamer-large-avg	137,023	176,272	147,236	118,695	82,502
versiontracker-small1	102,737	90,753	118,594	74,927	35,555
versiontracker-small2	92,562	83,420	113,813	31,494	29,695
versiontracker-small-avg	97,650	87,087	116,204	53,211	32,625
versiontracker-large1	317,644	300,553	237,984	179,699	113,099
versiontracker-large2	327,170	294,764	215,448	162,218	113,741
versiontracker-large-avg	322,407	297,659	226,716	170,959	113,420
extractor-small1	88,176	40,117	30,522	31,262	20,498
extractor-small2	80,846	40,599	32,380	30,363	21,950
extractor-small-avg	84,511	40,358	31,451	30,813	21,224
extractor-large1	210,516	203,848	163,477	115,745	82,089
extractor-large2	211,403	197,092	164,067	118,167	82,150
extractor-large-avg	210,960	200,470	163,772	116,956	82,120
aggregator-small1	38,448	47,050	73,679	54,771	37,577
aggregator-small2	37,916	47,525	74,621	56,732	40,690
aggregator-small-avg	38,182	47,288	74,150	55,752	39,134
aggregator-large1	107,466	107,327	93,967	80,846	69,757
aggregator-large2	108,794	123,901	101,849	71,071	47,765
aggregator-large-avg	108,130	115,614	97,908	75,959	58,761

Research report

D.1. Existing Solutions

This chapter will discuss existing solutions, both in the area of monitoring and in the integration of monitoring data with IDEs. We will document the relative lack of current IDE integration and research the most common logging systems, which will later be used to justify the design choices of the product.

D.1.1. Log and monitoring services

Given that the product needs to be integrated with a log data source, we have evaluated the most common logging and monitoring solutions used by the industry. Since the goal of the project is to create a modular product that is able to integrate with various data sources, we have specifically looked at integration methods and the type of data usually stored inside these systems. The website StackShare.io[1] was used as a way to evaluate the popularity of common logging stacks.

The most common log management systems can be divided into hosted and managed solutions. Hosted solutions are set up by the company using the log system, while managed logging solutions are hosted by a third-party company. Managed logging solutions have the added benefit that the company is not responsible for handling scaling and compute resources, but managed solutions are often more expensive due to their subscription model. Commonly used managed logging solutions include Loggly[2], Sumo Logic[3], Scalyr[4] and Splunk[5]. These hosts manage everything, with the customer only needing to provide log ingestion.

Hosted solutions on the other hand are usually open-source. By far the most commonly used hosted solution is the ELK or Elastic stack[6]. Short for ElasticSearch, Logstash and Kibana, they are a set of tools working together to ingest, store and visualize log data. Our research showed that the Elastic stack is by far the most popular way to manage log data, including over four thousand 'stacks' making use of the technology as per StackShare.io [2]. Other common logging solutions only use parts of the Elastic stack, substituting components. For example, Logstash is also commonly replaced by FluentD[7] while Kibana can be replaced by Grafana[8].

These stacks provide a couple of observations. First, we notice that all used stacks (whether they are hosted or managed) provide methods for both getting access to the raw log data and to aggregates of the data. Depending on the metadata stored in the logging system, it may not be necessary to do additional aggregation in a new system but instead offload this aggregation to the systems that already have the data needed. In cases where data still needs to manually be aggregated (such as when initial preprocessing is needed), the raw log data can be used instead. A second observation is that all mentioned stacks store log data as a combination of the raw log line and additional metadata. By

[1] StackShare.io is a site where engineers share the 'stack' their company uses. Numerous large and small companies have shared their stack on the site.
[2] https://www.loggly.com/
[3] https://www.sumologic.com/
[4] https://www.scalyr.com/
[5] https://www.splunk.com/
[6] https://www.elastic.co/what-is/elk-stack
[7] https://www.fluentd.org/
[8] https://grafana.com/

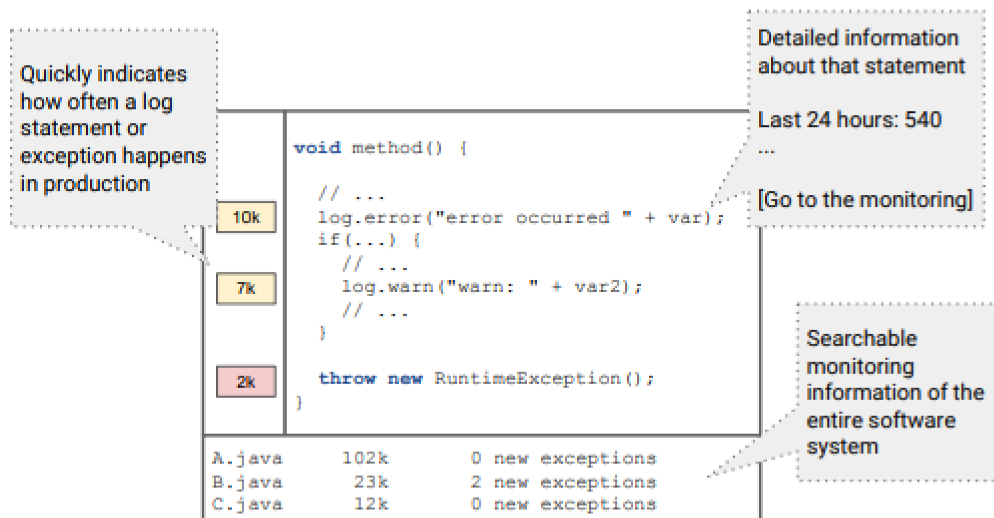


Figure D.1: Graphical User Interface proposal of Monitoring Aware IDEs by Winter et al. [11]

designing a system that is able to ingest these relatively dynamic documents, a system can be created that should support most of the mentioned stacks directly.

D.1.2. Monitoring IDE Integration

The idea of integrating monitoring systems into IDEs is not new. Research by Cito et al. (2015) [6] proposed integrating monitoring solutions in general. However, very few actual concrete implementations have been created. The first paper describing an actual implementation and evaluating its effectiveness is *Monitoring-aware IDEs* by Winter et al. [11]. This paper proposes a monitoring-aware IDE (see figure D.1) and discusses its implications in the workflow of a software development environment. A practical study was performed at Adyen, a large-scale payment company that produces a vast amount of logs. While the paper shows that a monitoring integration plugin is a promising tool for developers, there are various reasons why the plugin created by Winter et al. does not suffice as a generic monitoring-aware IDE implementation.

Firstly, the tool designed and evaluated by Winter et al. is tightly coupled with Adyen's internal systems. This facet, combined with privacy considerations, means that the specific implementation as discussed in the paper was not open sourced. The goal of this project therefore is to reconstruct this plugin as an open-source project, while at the same time making it generic enough that it generalizes to software stacks outside of the ones used at Adyen.

Secondly, the tool created by Adyen specifically uses heuristics to couple a log line with the source code that it originated from (see also "Tracing Back Log Data to its Log Statement: From Research to Practice" by Schipper et al. [9]). This was done because the runtime cost of computing the source location of a logging statement (along with the cost of storing this location data) is expensive enough to warrant avoiding it when possible. This heuristic approach is not perfect, especially in cases where log lines are partially or even fully dynamic. A generic re-implementation of the tool can use bundled location data, where possible. In cases where this is not feasible, it can fall back to the heuristics used by Adyen's implementations.

Winter et al. also mention that there is improvement possible with scalability and log ingestion times. The large amount of regular expressions used for matching log lines to their originating expression requires significant computation power, something that an implementation needs to support. Similarly, an integrated monitoring tool should ideally be processing logs as they arrive. The implementation created by Adyen uses a polling system for this and is therefore not fully real-time.

D.2. Problem Analysis

This chapter will discuss the aim of the project briefly discussed in the introduction in more detail. In particular, the design goals and requirements of the project are formalized and justified. First, the design goals section will discuss the challenges and goals needed to implement a minimum viable product. After this, formal requirements are listed using the MoSCoW method.

D.2.1. Challenges

As previously mentioned, the goal of the project is to directly integrate the metrics generated by log aggregation services such as ElasticSearch with the IDE. We have identified some key challenges that need to be solved to create a minimum viable product implementation:

- **Different log formats**
  Applications usually use their own format or some type of standardized format for storing their logs.

- **Large amount of logs**
  As described by Winter et al. [11], applications used by large companies produce logs in the counts of millions to billions per month. Being able aggregate this large number of logs is difficult.

- **Tracing logs to their origin in the application**
  Parsing code to match log statements to their origin is not easy if the origin is not embedded in the log itself.

- **Fast indexing and retrieval of stored logs**
  Big applications produce a constant stream of logs which need to be indexed for retrieval. Efficiently processing and querying this data can be a challenge.

- **Deciding what information to show to the developer**
  As this is relatively uncharted territory, there are no extensive studies on which information is most useful to show to developers.

D.2.2. Design Goals

Based on these outlined challenges, the requirements given by the client, and the previous work as described in [11], we have derived the following design goals. These design goals describe traits that an implementation should have in order to function as a sustainable minimum viable product. They are however implementation-agnostic (i.e. their existence does not necessarily mandate a certain implementation).

Modularity

The main design goal is to create a modular system where components can easily be swapped. By creating a modular system, a large amount of IDEs, log systems and architectures can be supported. Given that the result of the project will be open-sourced, a modular system allows third-parties to create their own extensions without needing full knowledge of the entire stack.

Scalability

To create a tool that is useful for larger companies, the tool needs to be able to scale to a large volume of requests. When possible, systems need to be designed that they can either directly handle large volumes of logs or are easily horizontally scalable.

Latency

This is a dual goal, referring to both the latency between a line being logged and it showing up inside the IDE, and the latency between the developer opening a document and receiving the metrics from the aggregation service. The latency for a developer should be minimized as much as possible through query optimization and caching, while the time from logging to being included in metrics should ideally be less than five minutes.

D.2.3. Requirements

Based on the listed design goals, the following requirements have been identified. They have been divided into **must** have, **should** have, **could** have and **won't** have, based on the MoSCoW method described by Clegg et. al [7].

ID	Requirement	Justification
RQ1	**Must** support the Java language	Client requirement, commonly used enterprise language
RQ2	**Must** support the Elastic stack	Client requirement, most popular log management framework (as per section D.1.1)
RQ3	**Must** have a service for retrieving log statistics with an API	Allows for the creation of alternative front-ends, promotes modularity
RQ4	**Must** have a plugin for the Intellij IDEA platform ($\geq$ v2020.1) that connects with the service	Client requirement, popular Java IDE (62% market share as of 2019 [1])
RQ5	Log line locations **must** be provided by the API	Allows for modular front-ends, less complexity tracing log lines on the client
RQ6	**Must** be able to show number of times log has been triggered in plugin	Client requirement, clear way too convey frequency of logging
RQ7	**Should** be able store logs statistics for a period of at least 30 days	The time to keep a log should be maximised but should not be excessive, older logs can be viewed in the company monitoring tool
RQ8	**Should** be scalable up to 50 billion log lines per month	Client requirement, upper estimate of log volume for Adyen
RQ9	Server **should** not re-aggregate when analytics data is already available	If the necessary data for the plugin is already stored in the user's database, it would be inefficient to aggregate it again
RQ10	Server **could** allow support for modularly adding missing metadata through plugins	When the log database does not contain all metadata needed for the plugin, this method can be used to compute this metadata during log ingestion
RQ11	**Could** support other IDEs (Eclipse, Netbeans, etc)	Supporting more users is a plus, but the API modularity allows anyone to implement a front-end
RQ12	**Could** visualize logs using graphs to view trend over time	With a graph the developer can more easily understand how certain code changes affect logging
RQ13	**Could** have heuristic log tracing	Tracing the origin of log lines can be inaccurate and inefficient, but is a requirement for certain tech stacks
RQ14	**Could** display the number of times a log line was involved in an exception	Provides a data-driven way for developers to see the impact of exceptions

Table D.1: Functional requirements

D.3. Design

This chapter will discuss the proposed architecture of the tool. Its design has been chosen based on the design goals and requirements outlined in chapter D.2, keeping in mind the existing monitoring solutions discussed in chapter D.1. The sections within this chapter will discuss how the proposed architecture was chosen and how it achieves the defined requirements.

D.3.1. Architecture

Figure D.2 shows the complete version of the proposed architecture.

Conceptually, the proposed architecture works similarly to the system described by Winter et al. [11].

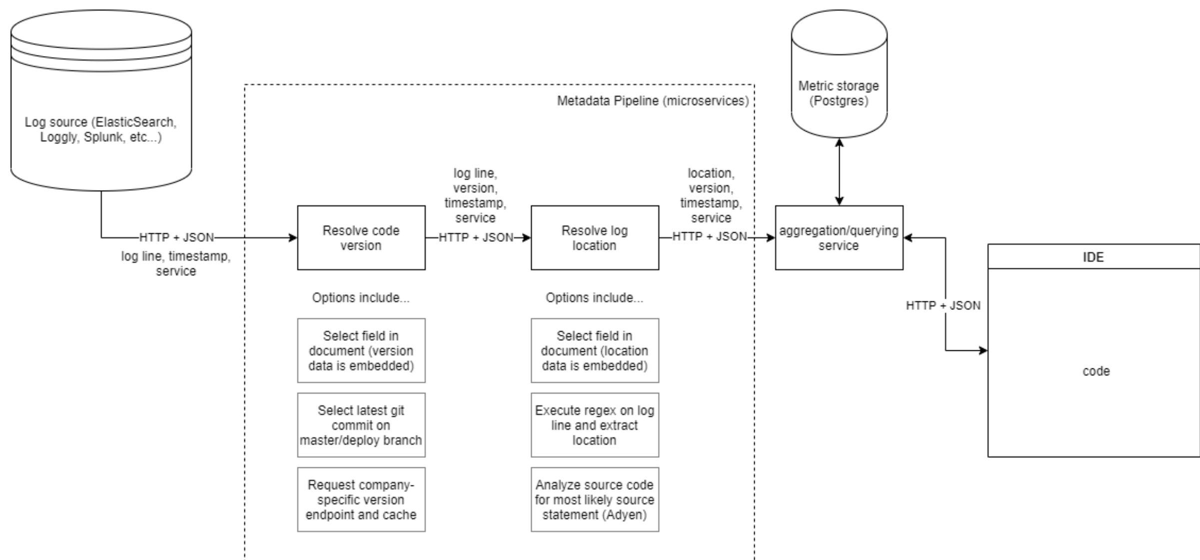


Figure D.2: The full proposed architecture of the tool.

A central aggregation service is responsible for exposing an API[9] to a plugin running inside the IDE. Using this API, the plugin is able to query for metrics that are currently relevant for the developer (such as metrics for the file the developer currently has open). The aggregation service has a dedicated database that it uses to store the intermediate aggregates and may optionally have a cache service attached.

Crucially, the aggregation service has two tasks: query known data that has previously been (partially) aggregated, and aggregating newly incoming metadata from the metadata pipeline. While in figure D.2 the aggregation service is shown as a single instance, it may be implemented as multiple microservices working together. By only limiting the aggregation service to these two tasks, any metadata source and any front-end can be easily attached to the system without requiring significant changes.

As discussed in the problem analysis, one of the challenges of the project is that logging data is not uniform and may require further processing before it can be ingested by the aggregation service. The metadata pipeline is intended to solve this problem. It functions as a set of composable plugins that allow a company to create a pipeline that attaches the necessary metadata to log data on the fly. For the initial design, we propose a pipeline that is able to resolve the version of the code that the system is running on (to group data and identify the source line) and the location where the log was triggered (either through extracting this information from the log line or by using a heuristic approach as described by Shipper et al. [10]).

The architecture specifically does not specify a single log source or communication format. The aim is to use a generic JSON-based object format similar to ElasticSearch objects that can get transformed by the plugins inside the pipeline. The aggregation service will only look at the required metadata and ignore any extra data. This allows any system that is able to return some format convertable to JSON to interface with the system. Additionally, this allows complete freedom for the language/tools used to implement metadata plugins.

For a similar reason, the metadata pipeline is implemented as individual processes instead of extensions embedded in the main process. This allows for easy scaling, introducing of extra dependencies, asynchronous metadata resolving and more, without the need for direct support from the aggregation server.

[9] Application Programming Interface

	Ease of scaling	Simplicity	Flexibility
Microservices with broker	Good	Fairly Poor	Fairly Good
Monolithic system	Poor	Good	Fairly Poor

Table D.2: Scalability of architectures on a scale of Poor/Fairly Poor/Fairly Good/Good

D.3.2. Pipeline Design

We will now discuss the metadata pipeline and how it communicates between sections in more detail. In the case where the client does not have the necessary metadata for the plugin to function, an additional pipeline is necessary for adding metadata. The problem with this, is how to deal with an increasing number of logs as with the case of Adyen.

One of the characteristics of this step of the pipeline is that it can be highly parallelized to handle more throughput. However, there are some caveats involved in parallelizing this step. In the case where there are parallel instances and there is some internal state that is shared over the instances (e.g. the current version of the source software), the state should not be changed by input logs. Ideally each instance would be stateless, but that would require constant retrieval of the missing metadata that needs to be added to the logs, which is inefficient. Another approach would be to handle this step asynchronously but with a single instance that has a single shared state that is not affected by input.

Adding additional fields to a document is not computationally intensive, but tracing logs back to line numbers in a codebase could be (depending on the size of the codebase and method of tracing). So the scalability should be considered in that case. As previously stated, a microservice architecture could be a solution. This would make horizontal scaling easy as you would simply have to start new instances and tell the message broker or load balancer where the new instance is. Microservices add much more complexity and overhead however, as the interfaces between the services should be well defined, otherwise the pipeline could easily fail. Managing which version of the service is running on each instance also adds additional complexity. Microservices are easier in terms of development however, as the choice of languages and tools is more flexible as long as the interfaces between the interfaces are well supported (e.g. syslog, http). The choices regarding architectures to process logs can be summarized in table D.2.

Based on these findings, we have opted to prefer flexibility over simplicity and have chosen to implement a microservice-based architecture for the metadata pipeline. For communication between the services, we have opted to choose a JSON format over the HTTP protocol. HTTP was chosen since it is by far the most common transport protocol for microservices (other alternatives include gRPC[10] and sockets). JSON was chosen since we as a team have the most experience with the protocol.

D.3.3. Intermediate Aggregation Storage

We have compared and benchmarked various potential databases for the aggregation service. We will now discuss why we have chosen PostgreSQL as the database implementation for the aggregation service.

The data that needs to be stored is the metadata per log, meaning that relationships are not necessary as a single table could suffice for storage. Therefore, both traditional relational databases and NoSQL databases are considered that support fast querying and indexing. Based on commonly used databases and databases specifically designed for analytics, we restricted our choices to Elasticsearch, PostgreSQL, ClickHouse and Apache Solr. These systems will be compared with respect to their scalability in this section. For the scalability of the storage and retrieval system, multiple metrics are used to compare solutions. A summary of the choices and metrics is given in D.3. The metrics used are as follows:

- **Can it be distributed?** If the storage system be split over multiple nodes, either for performance or storage. Can it be distributed out of the box?

[10] https://grpc.io/

- **Scaling for heavy querying**: How the system scales for more queries and how the speed of querying is affected as the amount of stored data increases. This is especially important when multiple developers are requesting heavy queries simultaneously.
- **Scaling for more storage**: How the system scales when the amount of data increases and how the performance is affected with more data.
- **The clustering elasticity of the system**: The speed and ease at which instances can be provisioned into a cluster, or removed from a cluster, respectively.

PostgreSQL
PostgreSQL is a relational database commonly used on a single machine. As the database may be used for heavy reading if multiple developers are using the tool simultaneously, it should be able to scale for heavy querying. A common method would be to have a master DB which is duplicated to multiple databases which are purely used for querying. An extension of solution would be to use a load balancer such as Pgpool-II[11]. The users' requests then get routed to one of the duplicate databases by the load balancer. The drawback with this is that duplicating the databases can be severely inefficient in terms of used storage and the data on the duplicates is not guaranteed to be real time.

Other than heavy querying, scaling for storage is also a consideration. The speed of single query decreases with the amount of stored data. In terms of storage scaling, PostgreSQL supports data sharding, making it easy to scale with more data nodes. A popular sharding solution is PostgreSQL-XL[12] for example. Citus [13] could also be used, which is an extension built on top of PostgreSQL which is fully distributed version of PostgreSQL that also offers parallel querying across multiple nodes. Citus also offers data sharding which is entirely handled by a rebalancer.

Elasticsearch
Elasticsearch is the de facto standard in terms of performance with search heavy queries. Elasticsearch is designed to be horizontal scalable by default. The data stored can automatically be sharded and each shard is assigned to a node in a cluster. Regarding query speed, Elasticsearch supports distributed search split over a query and fetch phase. One advantage of elasticsearch is that a search request can be handled by any node, where the node then acts as a coordinator to find the requested data. Whereas other storage and retrieval systems would usually have a master coordinator (e.g. in Citus).

Apache Solr
Similarly to Elasticsearch, Apache Solr[14] also supports index sharding and distributed search. Solr is also comparable to Elasticsearch in terms of performance as shown by AKCA et al. [4] and Luburic et al. [8], this is due to both being based on the Lucene search engine[15]. One area where Solr is not as comparable is its ecosystem. Due to the large adoption of the ELK stack, there is a much richer tooling ecosystem for Elasticsearch.

ClickHouse
Clickhouse is a big data store aimed at analytical queries. Clickhouse also supports parallel/ distributed queries and data sharding like Elasticsearch, Solr and Citus. Sharding requires manual configuration, whereas other solutions can do it automatically when you add more nodes. To remedy this problem, a tool such as Zookeeper[16] can be used to simplify configuration. Unlike PostgreSQL, Clickhouse does not use and support transactions, but speed of querying and storage expansion have more priority than ACID properties in our use case, thus this limitation is not of importance. Sharded data is split over a group of replicas, so even if transactions are not supported, it is highly fault tolerant. Clickhouse is designed with time series data in mind (can process up to 2TB per second), which makes it suitable as

[11] https://www.pgpool.net/
[12] https://www.postgres-xl.org/
[13] https://www.citusdata.com/
[14] https://lucene.apache.org/solr/
[15] https://lucene.apache.org/
[16] https://zookeeper.apache.org/

	Distributed	Scaling for querying	Scaling for storage	Elasticity
PostgreSQL	.	Good	Fairly Poor	.
PostgreSQL + Citus	□	Good	Fairly Good	Good
Elasticsearch	□	Fairly Good	Good	Fairly Good
Solr	□	Fairly Good	Good	Fairly Good
Clickhouse	□	Fairly Good	Good	Fairly Poor
Clickhouse + Zookeeper	□	Fairly Good	Good	Fairly Good

Table D.3: Scalability of storage on a scale of Poor/Fairly Poor/Fairly Good/Good

	Time to sum all (ms)	Time to aggregate per hour (ms)	Storage space used
PostgreSQL	5ms	6ms	1.1 GiB
ClickHouse	90ms	140ms	74 MiB

Table D.4: Benchmarking of PostgreSQL and ClickHouse on the same dataset.

storage and retrieval of logs and metadata where the timeframe is important.

Based on these results, additional benchmarking was performed between PostgreSQL and ClickHouse on a single machine. A mock database was filled with roughly 4.5 million rows of mock data with a granularity of 60 seconds. Where appropriate, indexes were created to help improve the query speed. Queries were performed on summing the entirety of the database, as well as grouping per hour. The results of the benchmarking can be seen in table D.4. Given the superior querying speed of PostgreSQL, we have decided to use it as main driver for the aggregation service. Should extra sharding be needed, CitusDB can be used to facilitate this need.

D.3.4. Front-end

In this section, we will discuss different options for the front-end plugin implementation. As a general guideline, we aim to implement most of the features present in the plugin implementation described by Winter et al. [11]. We have identified the following properties as essential for the plugin implementation:

- **Timely Integrated Feedback:** Near real-time textual/visual representation of how often a log statement or exception happens in production.
- **Traceability:** There should be a direct link between monitoring information and source code.
- **Search Capability:** Monitoring information should be available and searchable in the IDE.

For the initial implementation of the front-end we are limited by the capabilities of IntelliJ IDEA. The original prototype inserts a number in the side gutter displaying the number of times a log line has recently been triggered, with extra information available when the user hovers above the icon. An example of this implementation strategy can be seen in figure D.3. We agree this is an effective way of conveying the monitoring data to the user, but for completeness sake we also researched different ways of visualizing data. One particularly promising idea is to highlight an entire log line, as show in figure D.4. The benefit of this is that is is possibly more noticeable compared to the other method, as programming usually requires one to look more at the code than the sidebar. A downside of this method is that it conveys less information, only allowing color to convey severity. We think combining the two methods would result in getting the best of both worlds.

Another feature implemented in the prototype was an extensive search feature allowing the developer to filter logs by class name, order by occurrence frequency or other search options. We plan on implementing this as well. The implementation of this will be a simple table containing all values of each log line with a search feature and sortable columns. Future work can be done to improve this feature.

Outside of implementing the features described by Winter et al. [11], our client also proposed several improvements that might improve usefulness of the tool. These features include but are not limited to:

- Visualize specific logs using a graph to get a more in-depth view of the trend.
- First and last occurrence of a specific log entry.

```
37    private void checkInstance(Connection instance) {
38        if (instance == null) {
39 1K         logger.error( msg: "Connection instance is null");
40        } else if (instance instanceof P2PConnection) {
41 8K         logger.log( msg: "Instance is a P2P connection");
42        }
43
```

Figure D.3: Simple recreation of the interaction design given by the original prototype. Numbers on the left correspond to how often a log statement or exception happens in production. More information can be requested by hovering over the line.

```
37    private void checkInstance(Connection instance) {
38        if (instance == null) {
39            logger.error( msg: "Connection instance is null");
40        } else if (instance instanceof P2PConnection) {
41            logger.log( msg: "Instance is a P2P connection");
42        }
43
```

Figure D.4: Another possible way of visualizing severity of log statements or exceptions occurring in production. Color corresponding to the severity and more information can be obtained by hovering over the highlighted text.

We think being able to visualize logs with a built-in graph feature could be especially helpful and potentially give more information on the given log statement. The first and last occurrence of a log entry can best be shown when hovering over it or when using the search feature. Due to their optional nature however, they will not be prioritized in the initial implementation of the product. Should time allow however, they will be explored and potentially implemented.

D.4. Development

This chapter will describe the development process. We will discuss the exact details of languages and architectures used, describe how the team will coordinate during the development process and how the resulting product will be tested for correctness. These decisions currently reflect what we consider to be the best course of action based on the research outlined in this paper, but they may change based on encountered problems during development. The final report will outline a complete report of the final product.

D.4.1. Technologies

We have selected the frameworks and technologies to use based on the initial architecture described in chapter D.3.

Front-End & Communication

The front-end plugin will interact with the Intellij IDEA platform and thus use the libraries and methods exposed by that platform. It will communicate with the aggregation service through HTTP, encoding information using JSON. JSON was chosen since it is a common transport platform and all project members are familiar with the format. If time allows, the plugin will be extended to allow the dynamic pushing of new information. Such pushing can be implemented using a (Web)Socket implementation, but in the initial design phase no extensive research has been done into this topic. Section TODO describes more information about the front-end plugin and it's design.

Metadata Pipeline

The metadata pipeline is by design modular and is therefore not limited to a single language or technology. We have explicitly decided to move the pipeline out to separate processes instead of embedding them in the main aggregation service, as we expect this to scale better and give more freedom when companies need to implement a custom pipeline for their own infrastructure. This microservice architecture is also discussed in section D.3.2. Similarly to the front-end, communication between the components will be done in the JSON format. When needed, the metadata pipeline components may introduce their own technologies such as an intermediate cache server or communication with an external platform.

Aggregation Service
The aggregation service will be using the PostgreSQL database engine to store aggregated metrics. The reasons for choosing PostgreSQL are outlined in section D.3.3. If deemed necessary, a Redis server will be used to provide an extra in-memory cache. Redis was chosen over alternatives because it is commonly used in the industry and one of our team members has previous experience with it. As discussed, it will communicate with other components using the JSON serialization format. Individual actions in the aggregation service will be stateless as much as possible to allow for scalability.

Retrieving Logs
The proposed modular design explicitly does not name a single log source. As such, the method for retrieving log data from the storage depends on the specific log engine used. As ElasticSearch, the implementation required for a minimum viable product, does not support notifying other services when new data arrives, this section will support a polling mechanism that will likely use standard HTTP APIs to interact with log storage. When possible however, methods in which the log storage pushes new data to the aggregation service will be preferred over methods that require polling.

Programming Languages
As per the requirements, we will create a front-end plugin for the JetBrains IntelliJ IDEA platform. This platform is written in Java and Kotlin and runs on the Java Virtual Machine. As such, a plugin for the platform will necessarily need to be written in one of these languages as well. We have opted to choose Kotlin for this purpose, as it is fully supported by JetBrains, recent development efforts in the IDEA IDE have all been written in Kotlin, and it has our own personal preference. Since Kotlin has full interoperability with existing Java libraries and compiles to Java bytecode, it is fully suitable to use for plugin development. The Gradle build tool will be used to build the project, as it is fully supported by Kotlin and IntelliJ.

The aggregation server does not have such an inherent requirement. Instead, we evaluated the technologies that the aggregation server needs to interface with to limit our options. To interface with the PostgreSQL database, official drivers exist that support Java, Go, C# and C++, among others. Similarly, Redis drivers exist for all these languages. We have decided to opt for using Kotlin in the server implementation as well, as this allows us to stay consistent with the plugin implementation and use Java libraries when needed. Similarly to the frontend, Gradle will be used to build the project.

For the metadata pipeline, we have deliberately designed it to not require a single language. For the metadata components that we will be writing, we will likely use Kotlin to stay consistent with the other projects. If a specific metadata component implementation requires a different language, this will be justified in the final report.

Development
We will be using the Git version control system (VCS), as it is the most commonly used VCS in current development. Development will take place on a GitHub repository provided by SERG Delft, located at SERG-Delft/monitoring-aware-ides. We will be using the tools provided by GitHub for development, including GitHub Actions for CI/CD and GitHub issues/boards for issue management. Sections D.4.2 and D.4.3 discuss the use of these in more detail.

Packaging & Distribution
Given that the goal is to create a modular system, we will be distributing the components individually as both a single compiled Java JAR file and a Docker container image. The Docker image is provided to allow easy setup for companies using Docker, Kubernetes or similar applications for their machine provisioning. Raw JAR files are distributed for all other situations, requiring only a JVM installation to run.

We will also distribute some example Docker Compose configuration files for example setups. Such setups will include the database, cache servers and pipeline setup. They are not intended for production systems (as it will run all components on the same machine by default), but rather for evaluation of the system and for developers to quickly set up a development infrastructure.

D.4.2. Methodology

Whenever a software product is being developed, a specific approach to the development is either implicitly or explicitly used. In 1-man projects, just starting to code until you have a product can be completely fine. When it comes to group projects with defined clients however, some level of formalism is preferred. The formalisation of processes which are used to develop software are regularly described as Software Development Methodologies or SDMs. Software Development Methodologies help to clarify what problem needs to be solved, what features the result should have to solve this problem, how and in what order these features are implemented, and how the features are verified and validated. Each SDM tries to structure and solve these topics in a different way.

For use within this project, we have evaluated some common methods within the agile methodology family. In particular, we discuss Scrum, Kanban and Scrumban. After this, we argue our choice of software development methodology for the project.

Scrum

Scrum is probably one of the most well known implementation of the Agile Manifesto. A key principle of Scrum is the observation that customers do not really know what they want or need (requirements volatility) and that there will be challenges which are hard to predict and therefore cannot be comprehended in a predictive approach. Scrum accepts that a problem cannot be fully understood from the beginning and focuses instead on how to maximize the team's ability to deliver quickly and adapt to emerging requirements, evolving technologies and changes in market conditions.

The Scrum methodology defines several roles the team has to fulfill, first of all the product owner. As defined in the previous section, every Agile team needs a representative for the stakeholders, in Scrum this person is called the product owner. This role tries to reach a consensus within the development team on *how* to implement a requirement, it does not instruct the developers on how to solve the requirement. The product owner is a crucial part of the team and requires a deep understanding of both the business related incentives and the engineers.

The Scrum master – which cannot be the product owner – is the facilitator of the Scrum process. The Scrum master is accountable for removing impediments such that the team can function in an optimal manner. This includes coaching the team on the Scrum principles, facilitating regular team events and promoting self-organization within the team. Finally there is the development team. Scrum tends to limit the number of developers in the team to 9 members which accommodates the efficient communication. Even though it is called the development team, they actually do everything that is needed to deliver the final product. The members can be researches, software architects, software engineers, software testers and many other roles. We will refer to members of the development team as team members.

Scrum calls each development cycle a sprint and which is usually one to four weeks long. Each sprint stars with the sprint planning, in this team-wide meeting the team agrees on what to requirements to fulfill within the sprint (sprint goal). These requirements are picked from the product backlog and moved into the sprint backlog. In the second half of this meeting the team specifies the task needed to solve each of these requirements.

During the sprint there are daily stand-up meetings, these 15-minute meetings will allow each developer to convey three things: What he achieved yesterday and what it contributed to the sprint goal; What he plans to complete today and if he sees any impediment that could prevent him or the team from achieving the sprint goal. No detailed discussions should happen during the daily scrum meetings, once the meeting ends, members can get together to discuss issues in detail.

At the end of the sprint two events are held, the sprint review and the sprint retrospective. In the sprint review the team reviews the work that was completed and the planned work that is was not completed. They also present the completed work to the clients in form of a demo. At the end of the demo the stakeholders can indicate what they think is important to work on next. In contrast to the sprint review, the sprint retrospective is purely for the team itself. The team members reflect on the past sprint and

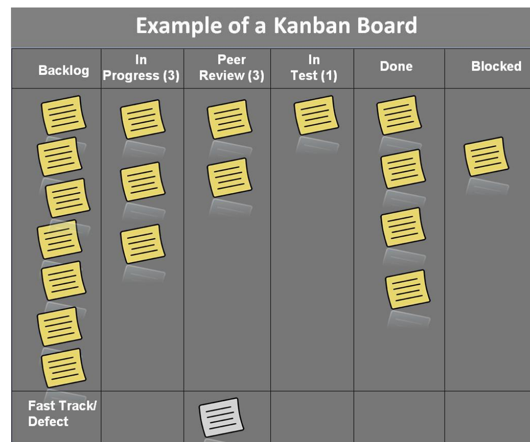


Figure D.5: Example of a Kanban Board

answer: What went well?; What did not go well? What could be improved in the next sprint?

Kanban
Kanban – an Agile Software Development Methodology – focuses on visualizing progress. This visualization is achieved by introducing the kanban board as shown in figure D.5. This board works in a continuous flow and pull based fashion. This means that there are no set iterations on which releases happen but releases can happen at any time features are created. The items in the backlog are ordered on prioritization and are not indexed on how much time they consume or to which developer it is assigned. Every other column has a limit on the amount of items it can contain(work-in-progress or WIP limits). This ensures that developers do not work on multiple different features at the same time but try to focus on just one or two. When there is no more room on the kanban board for a developer to move its card. He is encouraged to help other developers which may be blocked. This process of creating features is called 'flow' in kanban, the purpose of this method is to optimize flow. In Kanban the whole team owns the board, this means there is no product owner which decides on what is important to do next.

Kanban allows for rapidly changing requirements since items in the backlog are ordered on priority. The items in the backlog are also allowed to be removed from the board, it is not as set in stone as a scrum sprint. These benefits are due to the flexibile nature of Kanban, but this nature has its disadvantages as well. In Kanban it is very hard to plan things like efficient chronological releases. The Kanban methodology also does not specify any meetings with the development team. This means developers are not encouraged to review the process itself which may deliver key insights and can make the team more efficient.

Scrumban
As its name suggest, Scrumban is a combination of Scrum and Kanban. Specifically, Scrumban takes the Scrum methodology as its baseline and implements some Features from Kanban to further optimize the methodology. In Scrumban the Kanban board is introduced including its WIP limits, this allows for a continuous flow and visualization of progress just link in Kanban. Scrum tries to estimate the amount of work for each requirement, this is not required anymore, items should be prioritized just like in Kanban. Scrumban does not have strict iterations in terms of delivery but the sprints do allow for the planning and retrospective sessions. In the planning session new items are moved from the Scrum backlog to the Kanban to-do section. The retrospective meeting motivates developers to think about how effective the process is, what went wrong and what could be improved.

Choice
Based on the discussed software development methodologies, we have decided to choose the Scrumban SDM. We have ignored waterfall methods alltogether, as they require an exhaustive list of well-defined requirements from the start of the project. Since we are not experts on the topic of log ag-

gregating, it is very hard for us to predict what difficulties are involved in this problem. Since agile methodologies allow for quick adaptation of emerging requirements, we have limited our scope to those.

Our team has completed many projects using the Scrum methodology already, this makes Scrum a rational choice. We do however see the importance of visualizing work as Kanban projects. In addition, the continuous flow release cycle is a better fit for this project than (bi-)weekly releases. We conjecture that we will have few major releases with many small feature improvements, which would mean that having weekly releases would have little to no benefit. We do however value having retrospective meetings to reflect on the process. Because we are not currently allowed to meet as a team in-person (due to COVID-19) we have to use online voice communication platforms like Discord and Jitsi to have discussions and meetings. To enhance the part of a team feeling we will do multiple stand up meetings per day, this also allows us to motivate each other to stay productive while at home. Finally, we will use the GitHub board feature to create a Kanban board, have weekly planning meetings and weekly retrospectives.

D.4.3. Testing

Within this section we will discuss two types of software testing: static and dynamic testing. We will propose frameworks and guidelines to reach testing goals necessary to ensure a high quality product.

Static testing

Static software testing means testing the code without executing it. The technique involves both developers dry reading their own and other code and automatic tools that do this to look for bugs or deviations from conventions. This type of analysis can benefit mostly within the development process of the application. It enables developers to consistently name variables, have a consistent coding style, pass correct value types into functions and make code more readable in general. By making code more readable and consistent, it is easier for both current en future developers to work on the product.

Dynamic testing

Dynamic testing encompasses all testing techniques which require the source code to run. Dynamic software testing can be executed on many levels, we will briefly discuss these test levels.

The lowest level of testing coupled to detailed design is called component testing. Component testing searches for defects in and verifies the functioning of small software components like specific classes or modules. Component testing can be done through unit tests, test suites can either be manually executed or automatically by a CI/CD tool. Manual execution of test suites requires less setup and resources than CI/CD setups but becomes more cumbersome when a project grows in size.

Since exhaustive testing is often impossible, code metrics are used to limit the amount of tests. Popular code metrics include statement coverage, branch coverage and Modified Condition Decision Coverage (MC/DC). MC/DC tests all cases where one if the inputs single-handedly changes the output, this metric is often use in crucial systems designed for aircrafts and healthcare. Branch coverage requires for each branch in control structures like if statements to be at least called once. Together with statement coverage, which means calling each statement at least once, branch coverage is widely used as a coverage metric.

These test suits can also cover integration testing, which tests the interaction between several modules. Integration testing is extremely important for systems which allow for a modular design and are extendable. The setup and installation of the software is tested with system tests which test the system end-to-end (e2e). Stress tests do also account for this test level.

Choice

We will include both the validation and verificaton aspects of testing in our product. Validation of our product (acceptance testing) is given according to the paper by Winter et. al [11], and as such the product can be validated by communicating with the client.

Static analysis will be done by the ktlint and Detekt, which will integrate with Kotlin and Gradle to ensure that a consistent coding style is applied during development. Jacoco will be used to track coverage during testing, with a target of at least 80% branch coverage across the board. MC/DC testing is not applied, since we do not believe it to be necessary for the project. Coverage reports will be automated using the Github Actions CI/CD, automatically failing builds that do not meet this guideline.

Given that modularity and easy extensibility is one of the core goals of the project, larger system tests such as end-to-end testing, installation and usage tests will also be performed. To ensure that the product scales appropriately, JMeter and custom scripts will be used to stress-test various components.

D.5. Technology Summary

This appendix provides a brief overview of the architecture and technology used for the application, as described in the research report. Figure D.6 shows a full overview of the architecture. Table D.5 shows all the technologies used and where they are justified.

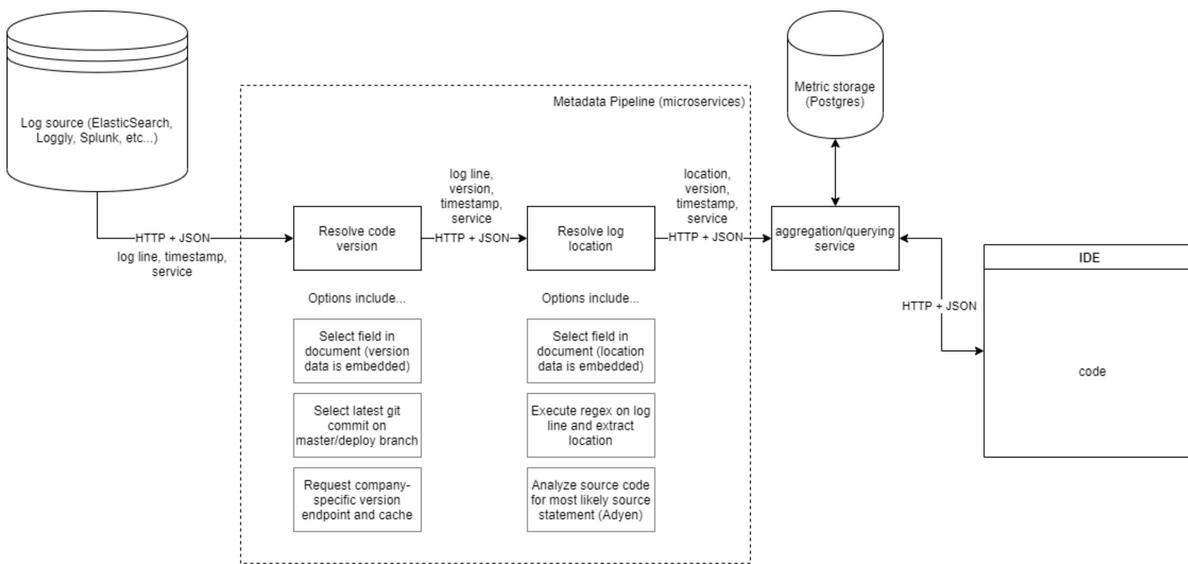


Figure D.6: Project architecture.

Purpose	Technology	Relevant Section
Plugin IDE	IntelliJ IDEA	Section D.2.3
Metric Database	PostgreSQL	Section D.3.3
Caching Backend	Redis	Section D.4.1
Programming Language	Kotlin	Section D.4.1
Build Tool	Gradle	Section D.4.1
Aggregation Service	ELK Stack	Section D.2.3
Build Packaging	Docker	Section D.4.1
Version Control	Git	Section D.4.1
Code Hosting	GitHub	Section D.4.1
Continuous Integration	GitHub Actions	Section D.4.3
Software Development Methodology	Scrumban	Section D.4.2
Kanban Board	GitHub Projects	Section D.4.2
Static Analysis	Detekt	Section D.4.3
Load Testing	Apache JMeter	Section D.4.3
Code Coverage	JaCoCo	Section D.4.3

Table D.5: Technologies used and where they are defined in the report.

Bibliography

[1] Java 2019 - the state of developer ecosystem in 2019 infographic. URL `https://www.jetbrains.com/lp/devecosystem-2019/java/`.

[2] Stackshare.io - logstash. URL `https://stackshare.io/logstash`.

[3] Zahra Shakeri Hossein Abad, Mohammad Noaeen, Didar Zowghi, Behrouz H Far, and Ken Barker. Two sides of the same coin: Software developers' perceptions of task switching and task interruption. In *Proceedings of the 22Nd International Conference on Evaluation and Assessment in Software Engineering 2018*, pages 175–180, 2018.

[4] Mustafa Ali AKCA, Tuncay Aydoğan, and Muhammer İlkuçar. An analysis on the comparison of the performance and configuration features of big data tools solr and elasticsearch. *International Journal of Intelligent Systems and Applications in Engineering*, pages 8–12, 2016.

[5] Hyperion Authors. The hyperion github repository. URL `https://github.com/SERG-Delft/hyperion`.

[6] Jürgen Cito, Philipp Leitner, Harald C. Gall, Aryan Dadashi, Anne Keller, and Andreas Roth. Runtime metric meets developer: Building better cloud applications using feedback. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, Onward! 2015, page 14–27, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450336888. doi: 10.1145/2814228.2814232. URL `https://doi.org/10.1145/2814228.2814232`.

[7] Dai Clegg and Richard Barker. *Case method fast-track: a RAD approach*. Addison-Wesley Longman Publishing Co., Inc., 1994.

[8] Nikola Luburić and Dragan Ivanović. Comparing apache solr and elasticsearch search servers. 2016.

[9] Daan Schipper, Maurício Aniche, and Arie van Deursen. Tracing back log data to its log statement: From research to practice. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 545–549, United States, 2019. IEEE. ISBN 978-1-7281-3370-6. doi: 10.1109/MSR.2019.00081.

[10] Daan Schipper, Maurício Aniche, and Arie van Deursen. Tracing back log data to its log statement: from research to practice. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 545–549. IEEE, 2019.

[11] Jos Winter, Maurício Aniche, Jürgen Cito, and Arie van Deursen. Monitoring-aware ides. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 420–431, 2019.