UNIVERSITY OF COIMBRA

MASTER'S THESIS

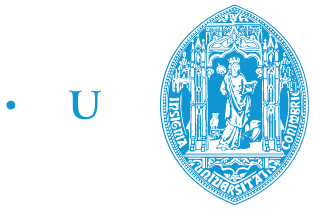
Objective Cyber Intelligence

Author:
Pedro STAMM

Supervisors:
Henrique MADEIRA (DEI)
André PINHEIRO (Dognædis)

A thesis submitted in fulfillment of the requirements
for the Master's Degree in Software Engineering

at the



Faculty of Sciences and Technology
Department of Informatics Engineering

2016/2017

University of Coimbra

Abstract

Faculty of Sciences and Technology
Department of Informatics Engineering

Master's Degree

Objective Cyber Intelligence

by Pedro STAMM

With the ubiquity of access to the global network, threats to computer systems have increased exponentially. The sheer amount and width of attack vectors make Cyber Security a focus of research and development, trying to minimize risk to individuals and corporations' technological assets.

A means to get a sense of how systems are behaving is the generation of events based on their current status and behavior, whether by the systems themselves or by outside sources.

However, the amount of events generated has increased exponentially as more and more resources and technologies are made available and used worldwide. Security Operations Center (SOC) teams were initially responsible for gathering, parsing and acting upon events as they arrived, but have recently become a bottleneck, as their manpower pales in comparison to the thousands of incoming data entries, possibly from multiple, highly diverse sources.

To solve that problem, and aid incident response teams, development of monitoring and analysis platforms has been a focus of Cyber Security experts. Not only can they help by making data presentable, they can also process, analyse and enrich it, creating new information from a multitude of originally "loose" events.

This project aims to add value to one such platform. Portolan, a Dognædis product, is a security monitoring platform outfitted for gathering and enriching data from multiple independent sources, formatting it into an uniform data model. Portolan gathers events from external sources in order add more context to internal security events. This document details the planning, implementation and evaluation of a module capable of Complex Event Processing, creating complex events from the data already gathered by Portolan.

Universidade de Coimbra

Resumo

Faculdade de Ciências e Tecnologia
Departamento de Engenharia Informática

Mestrado em Engenharia de Software

Objective Cyber Intelligence

por Pedro STAMM

A ubiquidade de acessos á rede global levou a um aumento exponencial do número de ameaças a sistemas digitais. A vasta quantidade e diversidade de vectores de ataque tornaram a área de Ciber Segurança um foco de pesquisa e desenvolvimento na tentativa de minimizar riscos para recursos digitais empresariais e de consumidores.

A geração de eventos para catalogar acções e interações de sistemas tornou-se um dos principais meios de perceber e monitorizar tanto o estado actual como o comportamento a longo prazo.

A utilização e disponibilização de cada vez mais recursos e tecnologias causou o aumento do volume de eventos gerados. As equipas de Segurança e Resposta a Incidentes que eram responsáveis for reunir, analisar e responder á chegada de Eventos não conseguem responder aos milhares de pontos de dados de diversos sistemas e tipos que recebem apenas com recursos humanos.

Para resolver esse problema e ajudar Equipas de Resposta a Incidentes, desenvolver plataformas de monitorização e análise de dados tem sido um foco de profissionais na área de Ciber Segurança. Não só ajudam a apresentar os dados recolhidos, como os conseguem processar, analisar e enriquecer, gerando nova informação com vista global a partir de Eventos originalmente isolados.

Este Projecto serve para acrescentar valor a uma dessas plataformas. O Portolan, produto da Dognædis, é uma plataforma de monitorização de segurança com o objectivo de reunir e enriquecer dados de múltiplas fontes externas e formatá-los para um modelo de dados uniforme, com o objectivo de adicionar mais informação ao contexto de segurança interna da empresa cliente. Este Documento detalha o processo que levou ao planeamento, implementação e avaliação de um novo módulo para o Portolan capaz de Processamento de Eventos Complexos.

Contents

List of Figures

List of Tables

Acronyms

CEP Complex Event Processing. iii, vii, viii, xv, xvii, 1, 3–6, 10, 13, 18–20, 23, 28, 34–38, 43–45, 67, 70–72

ESP Event Stream Processing. viii, xv, xvii, 3, 18, 23, 28, 35, 37, 44

PaaS Platform as a Service. 4

SOC Security Operations Center. iii, 5, 19, 30

Glossary

attribute A property of an event. 29

complex event An event that summarizes, represents, or denotes a set of other events[1]. iii, xi, 3, 24, 26–28, 30, 43, 50, 52, 53, 58–60, 67

Complex Event Processing Computing that performs operations on Complex Events, including reading, creating, transforming, abstracting or discarding them; may use Simple and Complex Events as input, but ultimately creates Complex Events[1]. iii, viii, xv, 1, 3–6, 10, 13, 18–21, 23, 28, 34–38, 43–45, 67, 70–72

Cyber Security Field concerned with the protection of information based on the concepts of Confidentiality (ensuring that it is not accessed without permission), Integrity (ensuring that it is not modified or destroyed in an unauthorized manner) and Availability (ensuring that it is available when necessary).[2][3][4]. iii, 1, 2, 7, 28

CyberSecurity Intelligence A field of Information Security focused on the production, gathering and processing of data from open and organizational sources to support strategic and operational decisions meant to ensure the security of a technological entity[5][6]. 1

dataflow An approach used in Event Processing in which events (the data) are processed by multiple nodes in sequence (the flow), each of which performs an operation based on the data received.. xi, xviii, 6, 10, 13, 27–30, 33, 37, 41, 43, 48, 49, 51–55, 58, 59, 64, 65, 67–69, 71, 72, 83, 86, 87

event Anything that happens, or is contemplated as happening; an object that records an event[1]. iii, 1, 3–5, 13, 26, 27, 29, 30, 32–37

event consumer An event processing agent that receives events[1]. 35

event correlation Event Correlation is a type of event processing in which complex events are generated from the events fed into the system; it is considered, therefore, a subset of Complex Event Processing.[7][8]. 13

event pattern A template containing event templates, relational operators and variables. An event pattern can match sets of related events by replacing variables with values[1]. 3, 4, 28

Event Processing Computing that performs operations on events, including reading, creating, transforming or discarding[1]. 7, 13, 23, 35

event producer An event processing agent that sends events[1]. 1, 3, 4, 35

event stream A linearly ordered sequence of events[1]. 3, 36, 37

Event Stream Processing Computing on inputs that are Event Streams[1]. xv, 3, 18, 23, 28, 35, 37, 44

message broker An intermediary piece of software used to route messages from a point to another. 35, 36

publish-and-subscribe A messaging pattern where producers publish messages without direct interaction with or knowledge of the subscribers, who consume the messages. 35

rule A prescribed method for processing events[1]. In this context, it is overloaded with the concept of Dataflow, an approach used in Event Processing in which events (the data) are processed by multiple nodes in sequence (the flow), each of which performs an operation based on the data received.. 3, 4, 27–30, 37

simple event An event that is not viewed as summarizing, representing, or denoting a set of other events[1]. 1, 22, 23, 26, 28, 30, 50, 60, 67

streaming window A bounded segment of an Event Stream[1]; can be bound by factors such as time or record count. 3, 28, 37

Chapter 1

Introduction

1.1 Overview

Currently, great amounts of information are produced by users, systems and organizations all around the world. That data, both from open and private sources, can be harnessed for CyberSecurity Intelligence in the form of events to aid decision making and operational activity.

One such example is Portolan, a Cyber Security platform developed at Dognædis, which uses event producers to gather data from specified sources into the system according to an uniform data model. Once in the system, events are then made available for examination. Portolan makes use of a modular architecture, with components that can be swapped in and out according to need.

Portolan distinguishes itself from similar products by focusing on data collected from external, publicly available sources, such as blacklists, pastebins and vulnerability reports. It uses that information to help provide the Users with a broader security context of their infrastructure and systems.

However, the sheer amount of data gathered makes it impossible to depend solely on manpower to filter, relate and act. Moreover, events are gathered as simple events, which means that relations between them have to be inferred manually. As such, there is a need for means to quickly and effectively process, correlate and present data gathered, reducing workload and allowing better decision-making and faster responses by the users.

This Project aimed to develop a new module for Portolan capable of High Volume Data Processing. The new module is capable of ingesting the Event Stream and process it record by record, delivering results in near real-time. That involved choosing components capable of high throughput ingesting, distributing and processing.

Since the data to be produced is a type of Operational Intelligence, the Data Processing paradigm most adequate is Complex Event Processing. Care was taken during the research phase to ensure that the CEP Module is capable of several kinds of high performance stateful computation, such as pattern matching and using Time Windows, natively. The features afforded by this paradigm can be used to develop new ways of correlating events, making use of complex structures and processing topologies. It also adds the capability of processing events from multiple sources (which can be both external and internal to the client's system), which is of vital importance in incident prevention and response.

Data is presented to the user through a new Django App added to the existing Portolan Django Web UI. In accordance with the nature of this project, it is completely modular and can be added and removed with no functional side-effects to the rest of the Interface.

A DataStream was developed for validation capable of filtering and aggregating Events through their IP and Domain name. This Datastream was used to study and apply DataStream optimizations, the functionalities offered by the Processing Engine's API, produce results to aid in developing the User Interface and test the overall system's performance.

Finally, the system's compliance with the Requirements was validated, leading to acceptance of the resulting system.

1.2 Context

This section details the context in which this document and internship are inserted, with the objective of introducing the preexisting environment. The theoretical concepts introduced here are further detailed in Chapter 4.

1.2.1 Dognædis

The internship is hosted at Dognædis[9], a company headquartered in Coimbra focused on information security. The company is a spin-off formed by a team of researchers from CERT-IPN who, after five years of activity, formed it as a private entity.

FIGURE 1.1: Dognædis logo

Dognædis continues the work started at CERT-IPN, maintaining a focused effort to advance and innovate information security in private and public entities, all the while maintaining a strict commitment of excellence in services and products provided for clients.

That mission statement is in accord with the etymology of Dognædis:

- *Dognitas*: Quality
- *Aedis*: Place, Temple

While the business model is focused on offering a number of security services such as audits, software assurancea and network and design management, Dognædis also develops products internally.

1.2.2 Portolan

Portolan is an Enterprise Security Intelligence platform developed at Dognædis meant to be integrated with existing infrastructure. It is leveraged by real-time and cognitive data analysis engines and directed towards providing decision support for Cyber Security.

It is based on three main concepts:

- **Integration**: Portolan is modular by design, allowing the platform to be adjusted to the user's necessities and evolve according to advances in technology without compromising existing features.

FIGURE 1.2: Portolan logo

- **Independence**: The platform is source independent. That means that it is capable of acting on data gathered from heterogeneous sources (ex: Social Media, IRC Networks, Monitoring systems) by event producers that adapt the data collected to the Data Model used by the system.
- **Pro-Activity**: Portolan's goal is to prevent security incidents by serving as a tool for both operational decision support and automatic response.

It is a sibling of IntelMQ, an open-source project that aims to process multiple data feeds (pastebins, social media posts) through a message queue.

The feature of interest to this dissertation that Portolan provides is the monitoring of data from multiple different sources (pastebins, blacklists, security events produced and made available by public and corporate systems) based on the similar functionality offered by IntelMQ.

Data is gathered, processed and fit into a defined Data Model by a pipeline composed of multiple nodes that outputs data gathered as events. The events generated can then be consulted in Portolan's dashboard.

This internship focuses on the planning and development of a Complex Event Processing module for Portolan capable of correlating the events generated by the previously mentioned feature through the identification of event patterns and generation of complex events.

1.2.3 Complex Event Processing

Complex Event Processing is considered a subcategory of Event Stream Processing, since both are used to extract extra information from events and operate over an unbounded event stream.

Instead of being focused on high throughput and simple operations, Complex Event Processing engines offer more complex functionalities to allow them to enforce rules meant to identify event patterns and analyse incoming data while maintaining state, such as streaming windows and aggregations.

Due to the flexibility required for this project and the usage of a custom data model, Complex Event Processing was considered the ideal approach, as is further explained in Chapter 4.

1.3 Problem Statement

This product serves to fulfill a need identified by Developers and Users of Portolan. That need is further detailed in this chapter, along with the objective of the Project.

1.3.1 The Problem

Portolan is currently capable of gathering, standardizing, storing and querying security events obtained from multiple and diverse kinds of data sources, both public and private (ex: pastebins, blacklists, enterprise monitoring systems). However, it lacks the

capability to further process and correlate events as a whole, leaving that work to the operators themselves.

With the amount of data received reaching the thousands at a time, using sheer manpower to keep track of every incoming event in an attempt to filter and identify patterns or notable situations is becoming a bottleneck. The density of events makes it difficult for users to take full advantage of the data gathered by the system.

That means that a technological solution capable of operating and correlating the incoming events as they arrive according to specified patterns is required. The implementation of such a system would result in reductions in overhead and an expanded understanding of the data, allowing the user to take full advantage of the system as a monitoring platform for operational decision support. Furthermore, it would reduce the time spent by operational team members in incident triage phases, speeding up response times.

1.3.2 Objective

The core objective of this project is to research, plan, implement and test a new module for the platform capable of processing the events gathered, enriching and correlating them by identifying event patterns and giving the platform the capacity to actually analyze the data it gathers.

Portolan innovates by focusing on processing events gathered from outside sources, incentivizing a new paradigm of sharing and gathering security information from the outside organizations. Processing and correlating them can add new information and context to a preexisting security infrastructure, some of which could be outright unobtainable by smaller teams.

From an implementation standpoint, while visualization is an important part of the end result, the most important aspect is the actual capacity to analyze and correlate events as they arrive. Since the platform already works based on events, taking advantage of the Complex Event Processing (CEP) paradigm is a natural fit. That means that a CEP-capable engine will be required, both due to the need to process events and ensure high throughput.

From a business standpoint, this project marks the initial exploratory steps of Portolan becoming a Platform as a Service (PaaS). Given that incident response teams are usually understaffed when it comes to dealing with medium or large incidents by themselves, offering Portolan as a service (even just partially) is an interesting business opportunity. Since it works based on a Feed model, this module can provide invaluable information for entities with weak security processes or small security teams.

1.3.3 Scope and Breakdown

The majority of this project is focused on the architectural design and adequacy of the technologies chosen, which means much effort was spent researching and learning existing solutions and technologies, deploying the chosen components in the available environment, configurating them to comply with the requirements, listing best practices and validating their inclusion through compliance with the Requirements. The theoretical study of usefulness of the DataStreams used by the system to process and correlate events is not a core component of the Internship.

The CEP Module must be capable of handling and processing the data gathered by the event producers already in use by Portolan. It must enable the extraction of additional context from the event Stream and any data contained in them through rules or sequences of processing steps.

Components used by the CEP Module that are already used by Portolan are considered production-ready and, unless strictly necessary, their configuration is unchanged.

The resulting system must be capable of presenting data to the user in an understandable fashion. This need can be translated into two objectives: the relations between events are established and clearly shown to the User, and the User must be capable of querying for events generated by the system. This entails the development or extension of the existing User Interface to search through the resulting data.

Developing a solution based on Machine Learning is not part of the Problem Statement. While it would be useful for Pattern Identification and Matching, the CEP Module also serves to process events in a stateful manner to extract more information from the event Stream as a whole according to the needs of the SOC.

Lastly, while the development and first production environments are expected to be single-machine, the system components should be prepared to scale horizontally to match the increasing volume of incoming events.

Goals

In order to achieve the stated Objective, the intern accomplished the following high-level goals across the First and Second semesters:

- **First Semester**
  - Familiarization with the domain
  - Research of State of the Art, Technologies and Architectures
  - Specification of Requirements
  - Definition of Architecture and chosen Technologies
  - Creation of a Proof of Concept
- **Second Semester**
  - Development of auxiliary tools
  - Definition of the Data Model
  - Implementation of Data Ingestion
  - Implementation of Data Storage
  - Implementation of Data Processing Engine
  - Development of DataStream components
  - Development of the User Interface
  - Verification and Validation

Constraints

The module developed by the intern should mainly make use of free open source tools, applications and frameworks, and is expected to run on Linux and Debian-based operating systems. The system will run in Virtual Machines running in company hardware.

The technologies chosen should support standalone and cluster deployments due to scalability concerns.

In order to be compatible and integrate with the platform in its current state, the user interface should preferably be made in Python/Django.

1.4 Document Outline

The present document is meant to support the project developed in the scope of the course "Dissertation/Internship in Software Engineering", required for completion of the Master's Degree in Software Engineering at the University of Coimbra.

The report aims to provide enough information to make known to the reader the required knowledge to understand the problem statement, requirements, state of the art, chosen process and decisions taken during the course of this first semester. As such, this report serves as a combination of a Problem Statement and Software Requirements Specification.

The chapters are organized as such:

1. **Introduction**: This First Chapter serves to present a short description of what was achieved during the course of the internship and a short introduction to the practical and theoretical context of the Project. It then presents the Problem Statement, detailing the motivation behind this Project and the objective to be achieved.

2. **Planning**: The Second Chapter details the Methodology chosen for the development process and management tasks associated with it, such as Task Estimations and Risk Management.

3. **State of the Art**: The Third Chapter presents the theoretical and practical concepts in detail. A good amount of effort was spent on research, as the context of the Project was very new to the Intern, and they serve as justification for the decisions made later in the Development process.

4. **Requirements**: The Fourth Chapter details the Elicitation Process and the Requirements agreed upon by the Intern and the Product Owner.

5. **Supporting Technologies**: The Fifth Chapter lists the technologies researched for every component along with their comparisons and the final decision for each.

6. **Architecture**: The Sixth Chapter is used to detail the chosen Architectural paradigm and present views useful to explaining the Architecture of the resulting system.

7. **Implementation**: The Seventh Chapter presents the implementation process, which was split into two main phases: the study, deployment and configuration of the Technologies chosen, along with the development of the CEP Dataflow; and the development of the User Interface.

8. **Verification and Validation**: The Eighth Chapter presents an overview of the process used to Verify that the components function as expected and Validates the final product's compliance with the Requirements.

9. **Conclusion**: The Ninth and final Chapter serves to close the Document with final remarks on the resulting system, looking back at what was achieved and detailing future work that can be built on this platform.

Chapter 2

Planning

Given that this is a rather large project, good planning and choice of methodology is a great asset to help manage time as a consumable resource, especially considering that it is being developed by a single person.

The project will be developed in two different stages, one for each semester of the course.

2.1 Time Budget

The following table presents the time budget available for each semester:

Time Period	First Semester	Second Semester
Start Date	12/09/2016	30/01/2017
End Date	23/01/2017	16/06/2017
Report Date	23/01/2017	03/07/2017
Weekly Effort	16 hours	40 hours
Weeks	19	19
Total Effort	304 hours	750 hours
Effort Percentage	28.8%	71.2%

TABLE 2.1: Time Budget per semester

As there is a clear discrepancy in estimated time available for each semester, each has a clear goal that fits with the budget available.

The first semester was thus used to achieve the goals detailed in Chapter 2: to research and understand the required concepts and context, research the State of the Art and formalize the requirements and architecture for the module to be developed.

Since it had a larger time budget, the second semester featured the implementation and validation phases of the project.

In the end, extra time was required due to a delay in obtaining a Virtual Machine for development and testing of the Data Processing Engine. This issue is detailed in the Chronogram section of this Chapter.

2.2 Methodology

Event Processing is a field that is currently in rapid evolution, with new tools, frameworks and architectures emerging frequently. Coupled with the intern's lack of experience in the fields of Cyber Security and Event Processing, the project lends itself to an agile methodology.

An incremental approach not only is more forgiving towards small mistakes but also permits freedom to learn and explore the necessary concepts and technologies while accommodating the possible need for quick change. As knowledge is gathered and the intern experiments different approaches the project can be gradually assembled and evaluated step-by-step by the company supervisor.

In order to achieve those goals, the intern chose an agile methodology inspired by Scrum and Kanban.

2.2.1 Scrum

Scrum is an iterative and incremental agile software development framework. It aims to cut through the complexity and overhead to deliver high-value products in incremental fashion with short development cycles.[10][11]

While the definition of the framework is team-oriented, the intern chose to adopt the key-concepts used by Scrum as part of the methodology for this project, with changes where required.

Product Owner

The Product Owner, in this the company supervisor associated with this internship, is responsible for maximizing the value of the product, prioritizing items in the Product Backlog, ensuring the items are understood, ensuring that the product's vision is maintained and providing feedback during each Sprint Review.

Product Backlog

The Product Backlog is an ordered list of everything that might be needed in the product and is the single source of requirements for any changes to be made to the product. It is the responsibility of the Product Owner, in this case represented by the company supervisor, and discussed with the intern to fit in the scope of the internship.

The Product Backlog lists all features, functions, requirements, enhancements, and fixes that constitute the changes to be made to the product in future releases.

Sprint

The Sprint represents the basic work unit of the method, a time-box of one month or less during which a complete, potentially releasable product is created. Sprints have consistent duration throughout the development effort, with each one having a specific Sprint Goal. A new Sprint starts immediately after the conclusion of the previous.

Sprints are divided in the following phases:

- **Sprint Planning**: The work to be performed during the Sprint is planned in this phase. In the context of the internship, the Sprint Goal will be decided by the intern according to the current Product Backlog specified with the company supervisor. After setting the Sprint Goal, the method to accomplish it must be chosen. The Product Backlog items selected for this sprint plus the plan are called the Spring Backlog.

- **Sprint Review**: At the end of each Sprint a Review is to be held to ensure that the Sprint Goal was accomplished, reflect on the current state and possible changes to the Product Backlog, and elicit feedback. The Sprint Review will involve the company supervisor and the intern.

Increment

The Increment is the sum of all the Product Backlog items completed during a Spring and the value of the increments of all previous Sprints. It must also be complete and ready to be deployed, regardless of whether the Product Owner decides to actually release it.

2.2.2 Kanban

Kanban is a technique for managing a software development process based on Toyota's "just-in-time" production system. It abstracts the process into a pipeline with feature requests entering from one end and improved software emerging from the other end[12].

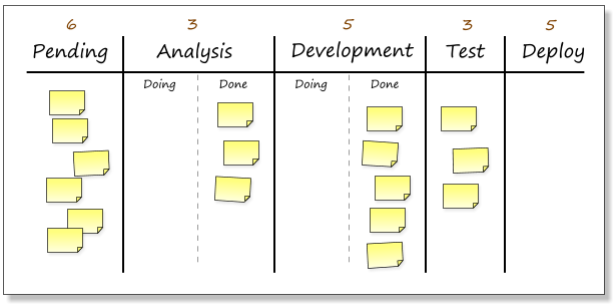


FIGURE 2.1: A Kanban board

In practical terms, it consists in the usage of a board divided into sections that represent a stage of the process where tasks are posted and moved as they change state. It allows visualization of the current work pipeline, with tasks being user stories that can be moved around and annotated as required.

2.2.3 Resulting Process

The resulting process is a mixture of a simplified version of Scrum directed at a development team composed of a single member with the usage of a Kanban board. During the development process, Sprints will last around 1 work week (40 hours), with relevant tasks added to the Kanban board during Sprint Planning and later if necessary, as long as the Spring Goal remains unchanged.
If the environment changes enough to make a Sprint Goal unattainable, the Sprint will be canceled, reviewed, and a new Sprint will be planned to replace it with the necessary changes.
The Kanban board will contemplate the following stages for the tasks:

- **To Do**
- **Research**
- **Prototyping**
- **Implementation**
- **Validation**
- **Complete**

Each resulting Increment of the system will be presented to the company supervisor for evaluation to ensure that the project is going according to plan, gather feedback and to accommodate any necessary changes as soon as they are brought up.

2.3 Chronogram

All estimations were the result of a 3-Point estimate, as the intern lacked enough knowledge at the start to use a more precise technique.

This technique involves the estimation of 3 durations for each task, Optimistic, Pessimistic, and Most Likely, after which the following formula is used to get the final estimate:

$$(O + 4M + P)/6$$

Due to space constraints, the Gantt charts for the 1st and 2nd Semesters can be found in Appendix A. The 2nd Semester has the Gantt Charts for the initial plan and final result.

2.3.1 1st Semester

The reality matched up with the chart with fairly low deviation. Above all, planning the completion of tasks that required the most work, Research and Requirements Specification, until the end of November paid off greatly, since assignments for other courses became a considerable time drain starting in December.

2.3.2 2nd Semester

The 2nd Semester suffered deviations from the original plan due to the manifestation of Risk R10. Tasks were reorganized, and all tasks that did not require the Development Environment were addressed first. There was a short time during which the Intern had completed the reorganized tasks but the Virtual Machine was not available yet. That time was used for further study of the technologies and to start developing the Dataflow.

The Development virtual machine was made available in April 4th. There was some overhead since there was a need to set up the tools and technologies that had been implemented thus far in the new environment. Implementing Apache Flink and the initial development of the Dataflow were accomplished faster than initially estimated, thanks to the research done previously.

The User Interface phase started later than was initially planned due to the delay. Additional iteration of the Dataflow was necessary to keep up with the additions to the Data Model, necessary for properly presenting data to the User. This phase lasted until early July.

In late June, the Intern estimated that there wouldn't be enough time for properly verifying and validating the system, and finish documenting the development process in the Final Report. After deliberating with both Supervisors, it was decided that it would be best to delay the final delivery to September.

The remaining time was split over adding new information and details of the Development Process to the Final Report, verifying the Performance and Availability afforded by the CEP Module and validating the Requirements.

2.4 Change Management

In accordance with the experimental nature of the project, change is expected as the development process moves along.

2.4.1 Sources

Changes are expected to originate from two different sources:

- Failure to fully implement a component according to the requirements specified in Chapter 3, causing the technology to be replaced;
- Changes requested by the Product Owner to maintain the vision of the product.

2.4.2 Process

Given that time and manpower resources are limited, changes will be subjected to an acceptance process, in which their impact on time budget will be evaluated.

If a change would cause enough impact to severely alter the requirements specified in Chapter 3, causing the scope of the project to increase or change, it will be rejected. If the change is deemed to be interesting for the product design, it will be added to "Won't" priority requirements and listed under Future Work in the final report.

If the change is not deemed to add significant overhead to the development process and fits the available resource budget, it will be prioritized by the Project Owner and added to the Product Backlog, along with a new requirement to ensure the feature is traceable and measurable.

If changes do occur, the Change Management process and results will be documented in a Change Log in the Final Report.

2.5 Risk Management

This section details the Risk Management process for the project.

2.5.1 Process

Risk Management is the process of identifying, prioritizing and mitigating potential issues that may occur during the development process. This process is inspired in and serves as a very light implementation of the Risk Management concepts detailed in ISO 31000[13].

It is inserted into the development process at the end of each Sprint, during the Sprint Review phase, in which emerging risks should be identified and, if deemed necessary, existing risks may be re-prioritized.

Risks will be prioritized based on their Probability and Impact, rated Low, Medium or High, in accordance to a Risk Exposure Matrix.

The intern is the sole entity responsible for managing the risks associated with the project.

2.5.2 Format

Risks are tracked with the following fields:

- **ID**
- **Description**
- **Probability**
- **Impact**
- **Consequence**
- **Mitigation**

2.5.3 Identified Risks

These tables describe the risks identified and tracked during the Internship.

Probability \ Impact	Low	Medium	High
High	R01		
Medium	R04	R07	R02, R10
Low		R05, R06	R03, R08, R09

FIGURE 2.2: Risk Exposure Matrix

ID	R01
Description	Full complexity of the project could not be grasped by the intern.
Probability	Low
Impact	High
Consequence	If the concepts, context, scope and technologies required are not understood by the intern, the project risks total failure.
Mitigation	Study of the domain during the 1st semester.

TABLE 2.2: R01 - Full complexity of the project not grasped by intern

ID	R02
Description	Time budget reduced due to overhead with other 1st semester courses.
Probability	High
Impact	Medium
Consequence	The intern may not have enough time to study the domain.
Mitigation	Time management policies and possibly reducing the scope of the 1st semester activities.

TABLE 2.3: R02 - Time budget reduced in the 1st semester

ID	R03
Description	Non-existence of technologies adaptable to project requirements.
Probability	Low
Impact	High
Consequence	The project may be unfeasible or require considerably more time than initially expected.
Mitigation	Reduction of project scope and refocus of requirements.

TABLE 2.4: R03 - No existing technologies match requirements

ID	R04
Description	CEP Engine cannot directly use Data Model used by Portolan
Probability	Low
Impact	Medium
Consequence	Event correlation and Event Processing are impossible if the system cannot ingest events.
Mitigation	An additional component for pre-processing will be required to format data into an acceptable Data Model.

TABLE 2.5: R04 - CEP Engine and Portolan Data Model prove incompatible

ID	R05
Description	CEP Engine cannot handle event load specified in requirements
Probability	Low
Impact	Medium
Consequence	Event Processing takes longer than the time specified in Performance Requirements.
Mitigation	Given that these kinds of technologies are meant for high scalability, the addition of extra computing/memory resources or optimization of dataflows should solve the issue.

TABLE 2.6: R05 - CEP Engine cannot achieve target performance

ID	R06
Description	Event visualization cannot be integrated into Portolan's graphical dashboard
Probability	Low
Impact	Medium
Consequence	Results of event correlations cannot be visualized or easily consulted.
Mitigation	A new graphical interface can be developed, or a new function/component can be added between the Database and the Dashboard to process data into a presentable format.

TABLE 2.7: R06 - Complex Events cannot be integrated to existing GUI

ID	R07
Description	Selected technologies turn out to be inadequate.
Probability	Medium
Impact	Medium
Consequence	Problematic technologies will need to be replaced, either with already existing or custom developed ones.
Mitigation	Research was conducted into multiple technologies per component to ease the replacement process and forewarn which ones may become more problematic if inadequate.

TABLE 2.8: R07 - Inadequate technologies chosen

ID	R08
Description	Selected technologies are discontinued and open-source userbase disbands.
Probability	Low
Impact	High
Consequence	Product life expectancy will become considerably lower, requiring replacement or reenginering of a new solution.
Mitigation	Research was greatly focused on technologies in active development and with a solid open-source community.

TABLE 2.9: R08 - Technologies are discontinued mid-development

ID	R09
Description	Chosen Architecture turns out to be inadequate.
Probability	Low
Impact	High
Consequence	An inadequate architecture will need to be re-engineered, possibly making component choices null up to that point.
Mitigation	Research into different architectural paradigms for this kind of problem. Given that most solutions found follow the reference architecture, risk of architectural inadequacy was greatly lowered.

TABLE 2.10: R09 - Inadequate Architectural Design

ID	R10
Date of Identification	February 17th, 2017
Description	Delay in access to Development Environment
Probability	Medium
Impact	High
Consequence	Not having the necessary environment for the development process can cause the project to be delayed or fail.
Mitigation	Tasks can be reprioritized, with those that don't require the development environment being fulfilled first. This mitigation plan does not entirely solve the issue if the delay is serious enough, but can be used to avoid losing time, which can lead to the project's failure.

TABLE 2.11: R10 - Delay in access to Development Environment

2.5.4 Change Log

- **R10**: This risk was added due to the delay in obtaining the Virtual Machine to run the Development Environment.

2.5.5 Materialized Risks

This section details risks that materialized during the process and a short comment detailing how they were handled.

- **R01**: The intern did not fully understand the scope of the project at the start. However, the mitigation policy was very effective: research of the domain greatly helped the intern grasp the concepts, paradigms and technologies necessary to plan out the implementation phase and architecture for the project, as can be understood in this document.
- **R02**: The assignments required for completion of the other 1st semester courses did cause overhead, which ended up taking some time away from the dissertation. However, by using the extra free time at the start of the semester, the intern was able to accelerate the tasks planned out for this semester and complete the domain study.
- **R10**: The Development Environment was only made available in April 4th, approximatelly two months after the expected date. The mitigation plan was deployed, and the Intern reprioritized tasks, accomplishing those that did not require the Development Environment or depended on those that did. While that kept the Project manageable, it did not fully eliminate the impact of the delay, which carried over into the next phase of development and contributed to delaying the final delivery to September.

Chapter 3

State of the Art

This internship requires multiple concepts and technologies that are, at this time, not taught in-depth in the context of the Master's Degree and that the intern was unfamilliar with. Research was required for the theoretical and practical aspects of the project in order to remedy that issue, from the most adequate computing paradigms to possible architectural patterns to solve the problem at hand.

This chapter contains the results of the research conducted over the course of the Project. It details the concepts and technologies necessary to fully understand the problem, the proposed solution, and allow development of the resulting system.

3.1 Similar Products

Examining products similar to Portolan served as an aid to understanding the product's use and purpose. Two software solutions that have some similarities with Portolan, in different ways, are developed by AlienVault.

AlienVault develops a number of software solutions geared towards Cybersecurity. Two in particular are similar to Portolan in execution and objectives: Unified Security Management and Open Threat Exchange.

Unified Security Management[14] (USM) is a paid security platform. It integrates threat detection, incident response and compliance in a single platform. USM is made to manage security for Cloud, Hybrid and Local environments through a security dashboard that discovers any available assets, assesses vulnerabilities and provides views to aid in intrusion detection, behavior monitoring and correlating security events. It can be completely deployed on-premises, or make use of sensors that send data back to the AlienVault secure cloud to be centralized and analyzed.

Open Threat Exchange[15] (OTX) is a free solution for researchers to share data and collaborate. It is meant to provide a global, public view of current and past threats in order to monitor their behavior and aid in securing infrastructure. Users can boost their own Cybersecurity while helping others. Parts of OTX are automated, using big data processing for natural language processing and machine learning.

While both feature an impressive array of functionalities, they differ from Portolan. Both Portolan and USM provide a dashboard and utilities to explore security events. However, by being modular, Portolan can be adapted to the environment and needs of the user, making it easy to adapt to hardware constraints and extendable with new functionalities. It can also make use of data collected internally and from public sources.

OTX is similar due to its use of public data and engines that analyze data in-flow, identify trends and are updated often. Portolan differs from it by allowing Users with a deployment on-premises to use their own data without having to submit it to an external system. It allows a company that wishes to not share its security events to still take full advantage of the platform without requiring private data to leave their system. This does not mean that the benefits added by OTX are lost to the client; in fact, Portolan

is capable of using OTX as an Event Source, providing amplified capacities for cyber security analysis from the point of view of the client's private infrastructure.

3.2 Computing Paradigms

When it comes to Data Processing, both the manner in which data is processed and the time required are important factors. They heavily influence the Data Model, how records are transferred between components, the choice of components and architecture and the latency between data being ingested and results being available.

The choice of Computing Paradigm, then, is a key decision at the core of this project, so a good portion of the research effort was spent investigating the Paradigms most adequate for processing of large volumes of data, their tradeoffs and the choice use-cases for each.

Three Paradigms of interest were identified during this research: Event Stream Processing, Batch Processing and Complex Event Processing.

3.2.1 Event Stream Processing

Event Stream Processing is focused on constant processing on a boundless stream of data, producing a steady output. Instead of the regular Database model, where data is stored, indexed and only then made available for processing, it takes the data in-flight and iterates over it with a simple set of operations to ensure low latency and take as much advantage as possible of parallelism, with recent solutions leveraging entire clusters as pools of resources.

This paradigm is mostly directed towards use-cases that require processed data to be available as close to "instantly" as possible, whether because the value of results lowers significantly in a very short timespan or to improve peformance or user experience. Some systems that take advantage of Stream Processing are bank ATMs, radar systems and management systems when updating inventory upon a sale, which take Simple Events, process or enrich them and present the result.

It is the paradigm closest to Real-Time Processing, with latency expected to cap in the order of just miliseconds.

3.2.2 Batch Processing

Batch Processing consists in the non-continuous, non-real time processing of large volumes of data. Unlike ESP and CEP, data is initially collected over time and grouped into Datasets, which are processed whole instead of record-by-record.

Due to processing a large volume of events per iteration, Batch jobs can incur large amounts of latency, which generally makes them inadequate for activities that require results in a short timespan, such as decision support and incident response. Plus, batch Processing can be difficult to parallelize, depending on the nature of the data and on whether the operations require results previously obtained in the same Dataset.

An example of heavy-duty Batch Processing is in the banking industry, where transactions, calculating interest rates and other operations that require a broad view of the events that occured over a large period of time are processed. In cases such as this, when changes to live data could break the process, other transactions can be stalled until the system finishes processing and has been updated with the end results.

The data generated by periodic Batch Jobs is often referred to as Business Intelligence.

The latency for Batch Processing can vary greatly depending on the volume of data and the operations required. It is generally expected to be high; in the case of the banking

example provided each iteration can last hours. With low amounts of data latency can go as low as seconds, but in that case the Paradigm isn't being applied to its full potential, with expected overhead due to having constant context switches in a short timespan when processing a new small dataset.

3.2.3 Complex Event Processing

CEP can be considered a subtype of Event Stream Processing due to the conceptual similarities between them. CEP arose from the need to perform more complex operations over continuous streams of data, dinstinguishing itself from ESP by trading off some of its performance in exchange for more complex functionalities (such as pattern matching, combining data from multiple sources and keeping state).

Complex Event Processing can be implemented in systems designed for ESP if the components or architectural design is flexible enough, either through the addition of new components to perform more complex tasks or extension of already existing components.

It fills the necessity for data processing that required complex, stateful operations without incurring in the heavy latency imposed by Batch Processing. Complex Event Processing emerged as a means to generate Operational Intelligence from unbounded streams of data or events.

Unlike Business Intelligence, which consists of periodic reports and aggregations of data, Operational Intelligence is data meant to deliver additional insight in near-real time, used to aid in short-term decision making and operational response[16][17]. It can be used to detect patterns signaling that a service is down or detect cyber attacks such as intrusions and suspicious operations, among others.

Performance remains an important factor. Depending on the complexity of the data, the operations required and the timespan during which data is still valuable, permissible latency can vary in the range of seconds to minutes.

3.2.4 Decision

Given that the Computing Paradigm heavily influences the Project as a whole, it was necessary to consider the affordances and tradeoffs soon in the research process in order to choose the most adequate paradigm and to focus the remaining time budget on it.

Batch Processing is not a good fit due to high latency, which hampers its usefulness for incident response. ESP, on the other hand, boasts low latency and high throughput. However, considering the context of the product, the latency incurred by CEP is an accepted tradeoff in exchange for the extra data processing capabilities it affords over the simpler ESP.

Additionally, the kind of data required for SOC teams is Operational Intelligence, since they have to analyze and respond to situations developing in real-time. That makes Batch Processing even less fitting.

With the information presented, CEP was deemed as the most adequate paradigm for the use-case in this Project: it provides near real-time processing of unbounded streams of data, which is exactly what Portolan requires; it can afford complex operations such as time-windows and pattern matching, which are important functionalities for producing Cybersecurity Intelligence to be used for alarmistic, incident response and reporting purposes.

3.3 CEP Reference Architectures

Research was conducted into reference architectures and already deployed solutions in order to serve as guides to the architectural design process of this Project. This section goes over three architectural paradigms of interest to give a summary overview of the evolution of Data Processing and the current relevant paradigms.

In order to provide a sense of how high volume Data Processing and Architectural Design evolved, the paradigms are in chronological order of inception.

3.3.1 N+ Tier Architecture

The N+ Tier Architecture[18] was an initial approach at defining a generalized architectural paradigm used for Complex Event Processing. This definition dates from 2009, when Event Processing was starting to become a topic of discussion.

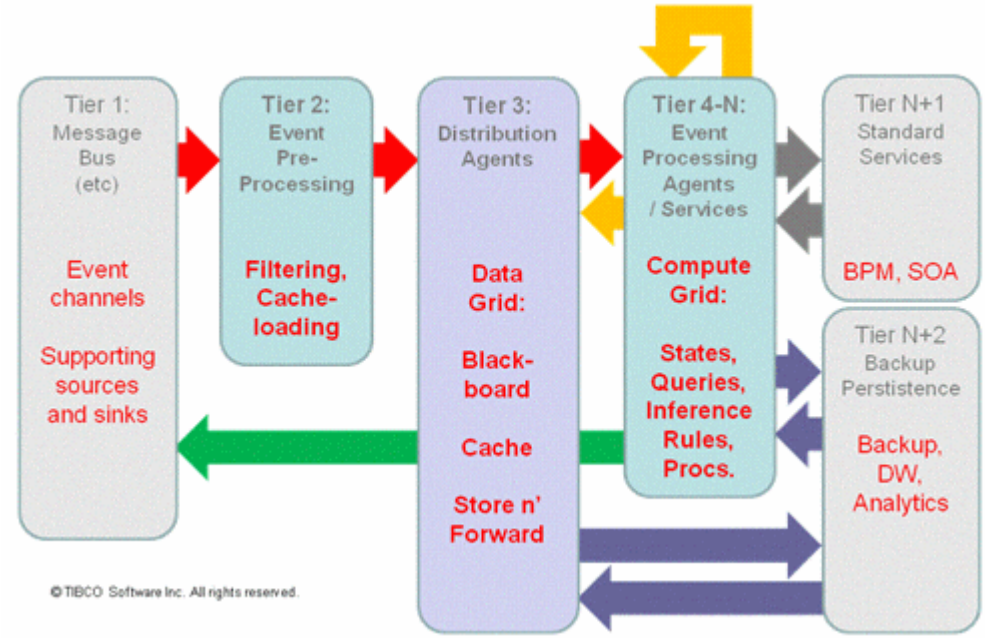


FIGURE 3.1: N+ Tier Architecture[18]

This pattern describes multiple layers of Data Processing organized in Tiers, with data flowing in-between them for ingestion, processing, storage and output. Each Tier corresponds to a set of functionalities similar enough to be grouped together or common to a particular component. The architecture is highly modular and allows the omission of some Tiers, or rolling functionalities from multiple tiers into one or more components.

Due to the specificity of each Tier and the early state of Stream Processing at the time, the actual usefulness of this paradigm as a building block for Data Processing architectures was limited, and it did not become a standard. However, the more recent patterns described below are essentially more generalistic versions of it, correctly adapted to the technologies available nowadays.

3.3.2 Lambda Architecture

The Lambda Architecture[19][20] is a high-level architectural pattern meant for high throughput, high volume Data Processing, leveraging both Batch and Stream Processing. It aims to balance latency, throughput and scaling by using Batch Processing to provide

accurate views of historical data, while using Stream Processing to fill in the gaps for data in-flight.

It was designed and formally named by Nathan Marz in 2013, based on his experience working with high volumes of data at Twitter. Nathan is also one of the people responsible for Apache Storm, one of the Event Processing Engines detailed in Chapter 5.

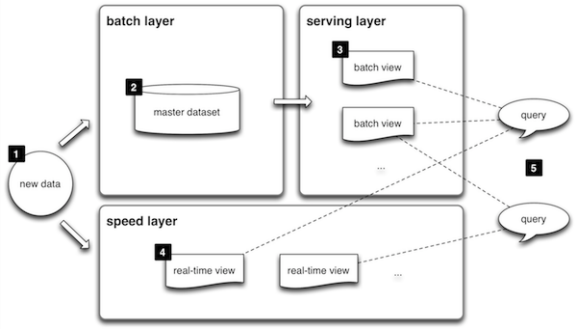


FIGURE 3.2: Lambda Architecture[19]

This pattern is divided into layers, each with a specific purpose.

- **Data Ingestion**: Similar to the N+ Tier's Layer 0, Data Ingestion is often not referred. However, the scalability of Data Ingestion is a factor in this Project. It can include a component to distribute the gathered data, or the distributors can be instead relocated to the Processing layers described next.
- **Batch Layer**: This layer requires a Data Storage system. New data should be appended instead of writing over old entries, making it a store for all historical data, which will be periodically processed by a Batch Processing Engine. The period depends on the amount of data and how often the views should be updated, and as explained in the previous section batch iterations can range anywhere from minutes to hours, days or even months.
- **Speed Layer**: The speed layer processes streams of data as records arrive, outputting incremental results with low latency. It requires the use of a Stream Processing engine. The expected latency is low, usually on the order of milliseconds to seconds.
- **Serving Layer**: Finally, the Serving Layer provides an interface to accept queries on the data produced by both the Speed and Batch layers. How it accomplishes that end is not specified, as it may vary depending on the type of data, processing and components used.

This paradigm is most useful for Business Intelligence due to its capability to strike a balance between trusted data with a complete view and readily available incremental data with some faults due to lack of context. It signifies a push for businesses to adopt Stream Processing along with already existing solutions of Batch Processing.

3.3.3 Kappa Architecture

The Kappa Architecture is an architectural design pattern for high throughput, low latency Data Processing. It is a simplification of the Lambda Architecture, with the Batch

Processing Engine removed and all data going solely through a soft real-time Stream Processing Engine.

It was first described by Jay Kreps from LinkedIn in a 2014 article[21] trying to address some of the criticisms raised at the Lambda Architecture, namely the need to maintain a separate codebase for each Data Processing Engine. In that article he proposes an alternative, which he tentatively names the Kappa Architecture.

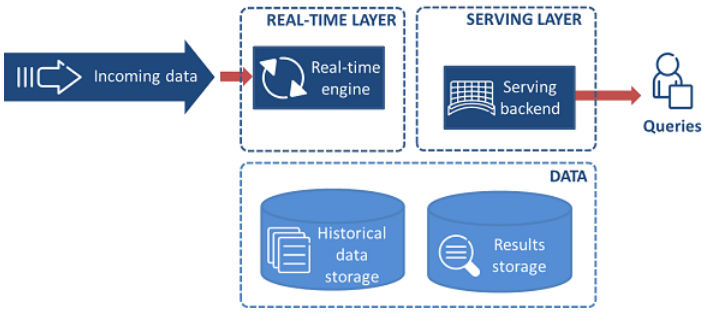


FIGURE 3.3: Kappa Architecture[22]

All Data Processing is handled by the Stream Processing Engine. If the system only requires the processing of data in-flight, the system functions similarly to the Lambda Architecture with just the Batch Layer removed.

However, in some cases reprocessing of historical data may be required, but the overhead of maintaining two separate codebases or the latency associated with the Batch engine can be prohibitive, even with the Stream Engine providing incremental output. In such cases, the Kappa Architecture specification solves that issue by maintaining the append-only Data Storage and moving the work originally done by the Batch Layer to the Stream Processing Engine. Instead of having an entire component for reprocessing of historical data, the Stream Engine is leveraged and has a data stream for that purpose.

Recent Event Stream Processing Engines were designed for that purpose and, asides from being highly parallelizable, can handle multiple data streams and both Batch and Stream Data Processing.

3.4 Components

Despite the different architectures available, the units that compose them are largely identical and fulfill similar needs. This section features overviews of what is expected from each component.

There are 5 key component types in Data Processing architectures: the Data Sources, the distribution agents responsible for Data Ingestion, the Event Processing Engine(s), Data Storage solutions and User Interface.

3.4.1 Data Sources

The Data Sources are responsible for feeding data into the system. They can produce the data themselves or collect it from other sources and redirect it into the system. The data is expected to be raw events fit into a Data Model used by the System, each one representing a simple event.

In this case, events are expected to be gathered from both external (pastebins, blacklists, security events and reports) and internal (network traffic, system and component logs) sources. There are multiple Data Sources collecting data at the same time, submitting it into the system at possibly arbitrary intervals.

It is not part of the CEP Module's responsibilities to collect events, but it is necessary to establish their source and format to properly plan the system architecture and estimate the reliability and scalability needed by components down the line.

3.4.2 Data Ingestion

These components are middleware between event producers (the Data Sources) and event consumers (the Processing Engines). simple events are submitted to Data Ingestion and Distribution agents, who are then responsible for distributing it to the Processing Engines as required.

Data can be pushed into the Data Ingestion agents by the Data Sources, or pulled from the Sources instead. Both approaches have tradeoffs: if there are enough Data Sources trying to push data at the same time, the Ingestion agents can be overloaded and struggle to cope with the incoming data. If data is pulled instead, the agents have to periodically request it from the Sources, possibly increasing the time between an event occuring and it being processed. For this kind of problem, Distribution Agents with focus on high-throughput are usually preferred to minimize latency and avoid being overloaded by the Data Sources.

They also serve as a buffer, queueing received data instead of overloading the Processing Engines. In situations of increased influx of data, depending on how they are implemented, it is possible for them to temporarily store the incoming data while the Processing Engines are scaled up to meet the increased demand, avoiding data loss.

3.4.3 Complex Event Processing Engines

As previously stated, CEP distinguishes itself from ESP by employing more intricate Event Processing operations. They can be included natively in the Processing Engine, added as extensions to an existing component, or by adding new components to a Data Processing Architecture.

Generally, CEP is obtained as a part of a larger product. Solutions such as IBM's ODM[23] and RedHat Drools Fusion[24] are meant to be integrated as part of their proprietary platforms, providing CEP under the hood for enterprise-level system, network and business monitoring. There are also standalone or modular CEP solutions, like Feedzai[25] and Espertech's Esper[26], meant to be added into existing architectures, integrated with data sources and provide CEP functionality. Those solutions, however, tend to be tailored for business-oriented use cases, such as fraud detection and generating real-time business intelligence. Finally, some companies write their own custom-made CEP Engine implementations for internal or a customer's use-case, limiting their adaptability or making them outright incompatible with different sets of requirements.

Alternatively, there are general purpose Stream Processing platforms developed to be highly scalable and extensible so that developers can heavily customize them. The first of these engines to undergo development, such as Apache Storm[27], do not have native CEP functionalities, but the logic required for it can be added. Their main focus was just prodiving a framework to manage Stream Processing, leaving the business logic to be implemented according to the use-case. More recent frameworks feature a more diverse set of native functionalities such as Apache Flink's[28] native stateful backend and CEP library.

3.4.4 Data Storage

The Data Storage layer is responsible for storing two major kinds of data: the raw, immutable, append-only events collected by the system; and the complex events generated by the Data Processing pipeline.

This layer can correspond to a single or multiple technologies, depending on the architectural specification and the system's needs.

3.4.5 User Interface

The User Interface is the layer with which the User interacts to view and query data. There are many methods and technologies to implement this layer, along with solutions that can function out-of-the-box if paired with other matching components.

Chapter 4

Requirements

In this section are detailed the gathering and analysis of Functional and Non-Functional requirements for the project.

According to the Methodology chosen by the intern, detailed in Chapter 5, the requirements serve as a Product Backlog for use during development. This list contains the necessary requirements to design and validate the architecture and technologies chosen.

4.1 Elicitation

Elicitation started with the study of the Problem Statement given by the company, research of the context and weekly meetings between the intern and the company supervisor. By being shown walkthroughs of PortoLan, the intern was able to understand the motivation behind the project and the scope of the task.

While researching the context and State of the Art the system's requirements were discussed and validated with the company supervisor, who effectively functions as the Product Owner.

4.2 Stakeholders

The system is expected to be used by and is therefore targeted towards the following actors:

Actor	User
Description	Operational Team Member
Interaction	Monitors complex events generated by the system. Monitors relationships between events. Makes queries in search of complex events.

TABLE 4.1: User Stakeholder

Actor	Developer
Description	Actor responsible for extending the system.
Interaction	Develops rules for pattern matching.

TABLE 4.2: Developer Stakeholder

Actor	Administrator
Description	Actor responsible for maintaining the system.
Interaction	Monitors current state of the system. Ensures system maintains event throughput and performance.

TABLE 4.3: Administrator Stakeholder

4.3 Functional Requirements

Functional Requirements were specified through the use of User Stories. which are a method to express functional requirements from the point of view of the stakeholder.

The MoSCoW Method[29], commonly used with agile methodologies, was used to establish priorities between requirements. This approach involves the use of 4 priority rankings.

4.3.1 Must Have

These are core requirements that reflect the minimum required functionalities of the product without which the development phase cannot be considered a success.

ID	FR01
Stakeholder	User
Title	Event Aggregation
User Story	As a User, I want to see which simple events relate to a particular complex event.
Rationale	As the core functionality of the system, being capable of uniting multiple simple events under one or more complex events is a means of representing the relation between events. Furthermore, it enables the inference of more abstract data from the lower level simple events (ex: a machine is very likely to be compromised if it is reported in multiple security alerts within a set time period).
Dependencies	None

TABLE 4.4: FR01 - Event Aggregation

ID	FR02
Stakeholder	User
Title	Correlated Event Navigation
User Story	As a User, I want to be capable of navigating between correlated events.
Rationale	Being capable of examining correlated events on demand provides another means to further investigate the situation.
Dependencies	FR06

TABLE 4.5: FR02 - Correlated Event Navigation

ID	FR03
Stakeholder	User
Title	Complex event Search
User Story	As a User, I want to be capable of searching for a particular complex events.
Rationale	Aside from presenting events as they are generated, the system must also allow operators to search for complex events based on their properties for consultation when necessary.
Dependencies	FR06

TABLE 4.6: FR03 - Complex Event Search

ID	FR04
Stakeholder	Developer
Title	Capability to add new rules for identification of patterns
User Story	As a Developer, I want the system to be extensible and allow the creation of new rules.
Rationale	As more event types are added into the system and the Operational team's needs expand, the system must be configurable to allow detection of new event patterns.
Dependencies	None

TABLE 4.7: FR04 - Modular addition of new rules

ID	FR05
Stakeholder	Developer
Title	Modular architecture
User Story	As a Developer, I want the components responsible for analysis to be detached from the rest of the Portolan architecture.
Rationale	Decoupling the analysis engine from the rest of the product greatly simplifies the development and deployment process. It also maintains Portolan's status as an adaptable solution that can be molded to the needs of the costumer.
Dependencies	None

TABLE 4.8: FR05 - Modular architecture

ID	FR06
Stakeholder	Developer
Title	Programmable Dataflows
User Story	As a Developer, I want dataflows to be programmable.
Rationale	The flexibility afforded by having dataflows programmable (whether in Java or another programming language) is very important, considering that events may come from a wide variety of sources and contain very distinct information. Not only does using a full-fledged programming language massively increase the variety of processing approaches, it uses a familiar paradigm to developers that lowers the learning curve of creating new rules.
Dependencies	FR01

TABLE 4.9: FR06 - Programmable Dataflows

ID	FR07
Stakeholder	Developer
Title	Data Manipulation
User Story	As a Developer, I want the CEP engine to be capable of operating (filtering and manipulating) over every data field of an event.
Rationale	Asides from identifying event patterns, the system should retain the basic Event Stream Processing capability of manipulating data received at will. It can help with data enriching and creation of more complex dataflows that require processing aside from pattern detection.
Dependencies	FR03

TABLE 4.10: FR07 - Data Manipulation

ID	FR08
Stakeholder	Developer
Title	Streaming windows
User Story	As a Developer, I want rules to be capable of using time windows.
Rationale	In Cyber Security, some complex events can only be inferred from simple events with enough certainty within a certain streaming window. Since this system will be used for operational support, it cannot create an alert in the case of an outlier. It would reflect on the usability and usefulness of the system, as operators would rely less on it, possibly ignoring it entirely after enough cases of false positives in a row. Supporting streaming windows is a means to increase the reliability of complex events generated.
Dependencies	None

TABLE 4.11: FR08 - Streaming Windows

ID	FR09
Stakeholder	Administrator
Title	Health Monitoring
User Story	As an Administrator, I want to monitor the current health status of the CEP engine.
Rationale	Simple metrics such as the current status of processing and master nodes are vital in understanding the health status of the system. Without that, it's difficult to identify issues in the module fast enough to minimize negative impact.
Dependencies	None

TABLE 4.12: FR09 - Health Monitoring

4.3.2 Should Have

These requirements are features not considered critical, but still capable of adding high value to the end product.

ID	FR10
Stakeholder	User
Title	Graphical Representation of Relations
User Story	As a User, I want to see a graphical representation of the relationships between events.
Rationale	Having a graphical representation of relationships between events allows operational team members to, at a sight, measure the impact and strength of the connection between events.
Dependencies	FR06

TABLE 4.13: FR10 - Graphical Representation of Relations

ID	FR11
Stakeholder	Developer
Title	Reusable Dataflow Components
User Story	As a Developer, I want components of dataflows to be reusable for other dataflows.
Rationale	Certain components may have functionalities common to multiple rules (ex: filter events by a particular attribute). Having to remake common components would result in overhead in dataflows development. As such, having the possibility of reusing dataflow components would greatly reduce overhead for that task.
Dependencies	FR01, FR03

TABLE 4.14: FR11 - Reusable Dataflow Components

ID	FR12
Stakeholder	Administrator
Title	Performance Monitoring
User Story	As an Administrator, I want to see statistics related to the performance of the system.
Rationale	Allowing system administrators to check performance metrics related to the system will allow them to identify and solve emerging issues (ex: bottlenecks, increase in resource consumption).
Dependencies	FR09

TABLE 4.15: FR12 - Performance Monitoring

4.3.3 Could Have

These are features that do not add enough value to be considered important, but can still be included if features with higher priority aren't affected. These will be the first to be removed from scope in case of necessity.

ID	FR13
Stakeholder	User
Title	Simultaneous Event Search
User Story	As a User, I want to be capable of peforming a search query over complex events and simple events simultaneously.
Rationale	Operating search queries on both complex and simple events is a feature that is expected to help investigating and inferring new relations between events, possibly helping in the creation of new rules.
Dependencies	FR06, FR08

TABLE 4.16: FR13 - Simultaneous Event Search

ID	FR14
Stakeholder	User
Title	Event Alarm
User Story	As a User, I want to be alerted whenever a type of complex event I flag is generated.
Rationale	Taking advantage of the alarm functionality in Portolan could help decrease the reaction time of SOC operators.
Dependencies	None

TABLE 4.17: FR14 - Event Alarm

ID	FR15
Stakeholder	Developer
Title	GUI for dataflow Deployment
User Story	As a Developer, I want to be able to deploy rules through a graphical interface.
Rationale	A quality of life requirement to boost the system's usability.
Dependencies	FR01

TABLE 4.18: FR15 - GUI for Dataflow Deployment

ID	FR16
Stakeholder	Developer
Title	Dataflow Wizard App
User Story	As a Developer, I want a simple Wizard app to quickly create simple rules.
Rationale	Having an app to partially automate the process of creating very simple rules from pre-built components would facilitate quick dataflow creation and deployment.
Dependencies	FR01, FR10

TABLE 4.19: FR16 - Dataflow Wizard App

ID	FR17
Stakeholder	Administrator
Title	System Performance Alarm
User Story	As an Administrator, I want to be alerted when the system's performance/throughput degrades.
Rationale	Expanding on the system status monitoring capabilities, metrics could be measured and an Administrator could be warned if performance falls below a determined threshold.
Dependencies	FR12

TABLE 4.20: FR17 - System Performance Alarm

4.3.4 Won't Have

These are requirements that have been requested but are excluded from scope during the planned duration. They may be included in future phases of development.

As every Functional Requirement was covered and deemed to not be decisively outside of the time-budget for this project, none were considered as "Won't Have".

4.4 Quality Attributes

Quality attributes are requirements that describe the system from a non-functional point of view. The quality attributes that drive the architectural design are, in order of importance, Performance, Reliability, Security and Scalability.

4.4.1 Performance

ID	QR01
Priority	Must
Description	The system must be capable of enduring a load of 20000 events per second.
Rationale	Portolan currently receives about 15 to 20 thousand events in burst. As such, it should be expected for the system to support at least that estimated load. This requirement will affect both required hardware specifications and technologies chosen. The choice in target value was assisted by research conducted by the Intern and Data Processing Benchmarks performed by Yahoo and Data Artisans.

TABLE 4.21: QR01 - Target event load

ID	QR02
Priority	Must
Description	The system can have latency of no more than 5 seconds between an event entering the system and it being processed, up to the load estimated in QR01.
Rationale	Since this is meant to improve Portolan as a decision support platform, processing latency should not exceed a specified amount to ensure that incident response is not delayed.

TABLE 4.22: QR02 - Target latency

4.4.2 Reliability

ID	QR03
Priority	Must
Description	The system will use stable, upgradable technologies.
Rationale	The use of stable technologies minimizes the risk of unstable features. Technologies being upgradable ensures that, in case of a discovered vulnerability or software faults, they can be fixed and the system can be updated.

TABLE 4.23: QR03 - Stable, upgradable technologies

ID	QR04
Priority	Should
Description	The system will support checkpointing.
Rationale	Checkpointing refers to the capability of, in case of an unexpected failure (ex: hardware failure), the system should be capable of resuming processing the event from a saved state to avoid restarting the entire dataflow.

TABLE 4.24: QR04 - Checkpointing

4.4.3 Security

ID	QR05
Priority	Should
Description	In the GUI, input fields will be treated to avoid injection.
Rationale	Code injection is a common software vulnerability and should be avoided given the security nature of the platform.

TABLE 4.25: QR05 - Avoid injection from user input

ID	QR06
Priority	Should
Description	In the GUI, access to the data processed by the CEP engine will be provided only to users whose session is validated.
Rationale	Only authorized users should have access to secure information.

TABLE 4.26: QR06 - Validated User access

4.4.4 Scalability

ID	QR07
Priority	Must
Description	The system must be capable of splitting workload across available processing nodes.
Rationale	Given that it is expected for event quantity fed into the system to increase, the Data Processing Engine should be capable of splitting workload and take advantage of processing data in parallel workflows if required.

TABLE 4.27: QR07 - Distributed workload over cluster

ID	QR08
Priority	Must
Description	The chosen technologies must be capable of horizontal scaling.
Rationale	Given that it is expected for event quantity fed into the system to increase, the new system components should be capable of scaling horizontally to make it easier to cope with the platform's growth down the line.

TABLE 4.28: QR08 - Horizontal scaling of technologies

4.5 Change Log

This section documents changes made to the requirements during the course of the project with the accompanying reasoning.

- **Added QR08 (4.28)**: This requirement was added to specify that scalability concerns apply to new technologies added as part of the project's scope. The rationale behind QR07 (4.27) was rephrased to distinguish it from QR08.

Chapter 5

Supporting Technologies

While the architectural patterns have some differences, there are sets of common functionalities that form the core components of all three.

This section servers as a follow up to the previous one by specifying the most important components of Event Processing solutions and listing feasible technologies for each one.

5.1 Data Ingestion

In CEP, message brokers are used as a middleware mechanism between event producers and event consumers. As such, the technology responsible for this must be capable of receiving data from multiple sources and serving it to the CEP Engine with high throughput and low latency.

5.1.1 Apache Kafka

FIGURE 5.1: Apache Kafka logo

Apache Kafka[30] is a distributed streaming platform, providing publish-and-subscribe capabilities while assuring very high throughput, making it ideal for Event Stream Processing use-cases. Kafka also assures durability by using persistent storage and sharding across nodes, reducing the risk of loss of data in case of a node going under.

While Kafka can also be used as an Event Processing Engine, it is not meant for Complex Event Processing. It can, however, be used for pre-processing events if the need arises.

Kafka uses Apache Zookeeper[31] for quorum and as a point of discovery, synchronization and leader election for brokers in the cluster. Zookeeper is a centralized service for maintaining configuration information, naming and providing group services.

Benchmarks of scaled deployments also promise that horizontal scalability is a well developed feature, as can be seen in the benchmark conducted by LinkedIn[32].

5.1.2 Redis

FIGURE 5.2: Redis Logo

Redis[33] is an in-memory data structure store commonly used as a database, cache and message broker. Thanks to its low level API and in-memory storage it is capable of very high performance. It also supports first synchronization, replication and auto-reconnection out of the box, making it painless to deploy.

5.2 Event Processing Engine

The Event Processing Engine will be responsible for processing the events that enter the system. The technologies researched natively feature or can be extended with CEP functionalities. All of them feature fully programmable Processing Pipelines and can be horizontally scaled to take advantage of distributed pools of resources.

5.2.1 Apache Storm

FIGURE 5.3: Apache Storm logo

Apache Storm[27] is a Distributed Real-time computation system, making it easy to reliably process unbounded steams of data like event streams. Storm topologies are composed of Spouts that serve as the source for the data streams and Bolts that process them.

Storm guarantees that every tuple that enters the system will be fully processed by the topology by tracking the tree of tuples triggered by every Spout and determining whether each tree has been successfully completed or not. Every topology has a "message timeout" which, when triggered by a tuple, causes the engine to fail the tuple and replay it later.

Topologies are executed across one or more Worker processes. However, preparing a Storm topology isn't a simple task, requiring care and understanding of the sample Spouts and Bolts provided with the tool.

Storm also provides native APIs to interface with common message brokers and data sources such as Kafka.

FIGURE 5.4: Apache Spark logo

5.2.2 Apache Spark

Apache Spark[34] is a generalized distributed platform for computing and data processing, providing libraries for a diverse set of purposes, from Machine Learning to Stream Processing.

Spark operates on RDDs (Resilient Distributed Datasets), a general collection of data partitioned across multiple machines, which makes it fit for batch-processing. Due to that same fact, Spark isn't a true event stream Engine, and while that isn't condemning, it decreases its usefulness and versatility for Complex Event Processing.

5.2.3 Apache Flink

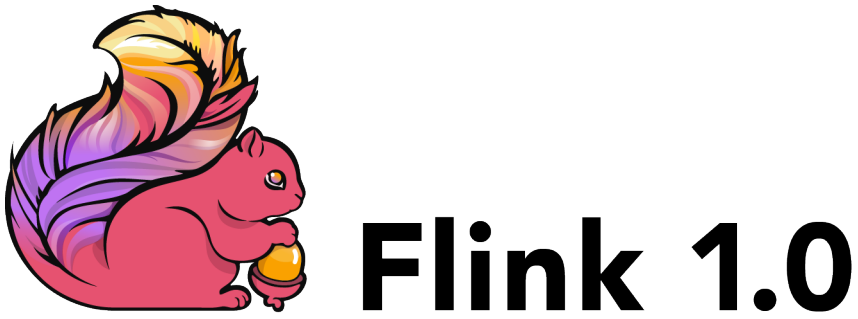


FIGURE 5.5: Apache Flink logo

Apache Flink[28] is a Streaming Distributed Dataflow Engine that provides data distribution, communication and fault tolerance for distributed computations over data streams. It is built from the ground-up as a true Event Stream Processing Engine, although it is also capable of processing events in batches.

Flink implements a lightweight checkpointing mechanism, ensuring exaclty-once semantics for the state in presence of failures while retaining high throughput rates. It also natively supports flexible streaming windows over time and count, which is an important factor for the definition of rules used in Complex Event Processing.

Implementation of a Dataflow in Flink is declarative, and can be achieved by writing a Java or Scala program. The engine takes care of compiling and optimizing the program for executing in a Dataflow executed in a cluster or cloud environment. The Dataflow is executed as Tasks on Worker processes across the available cluster, with all Workers being able to potentially fulfill all required tasks as necessary. Thanks to that, the engine itself is capable of distributing workload across the available resources.

The framework and APIs provided are also simple to use and fairly well documented, even including a CEP library and DataStreaming APIs to consume and produce data directly for multiple sources, including Apache Kafka.

5.3 Persistent Data Storage

A Data Storage solution for both Portolan and the CEP Module needs to be scalable, fast and flexible enough to permit adding new information with little effort. For those reasons only NoSQL storage solutions were considered.

Portolan already has an immutable data backend. However, research was still conducted to ensure that it was adequate for its purpose and to store the events generated by the CEP Module. One alternate Data Storage solution was found that offered enough advantages to consider implementing, with the Product Owner voicing some interest for replacing the current solution at some point.

5.3.1 ElasticSearch

FIGURE 5.6: ElasticSearch logo

ElasticSearch[35] is a distributed, RESTful search and analytics engine based on Apache Lucene. It provides a distributed full-text search engine with HTTP web interface and schema-free JSON documents, serving as both a DataBase and full-fledged, highly customizable SearchEngine.

ElasticSearch is highly scalable and features high throughput. Data is organized in Types, which are grouped into Indexes. When in a multitenant deployment, Indices can be divided into multiple Primary and Replica shards, which are automatically rebalanced across the cluster.

The data model can be heavily customized on a Type-by-Type basis to optimize the performance of the search engine. All fields added into ElasticSearch become full-text searchable by default. Field data and queries are subject to customizable transformations to make them easier to match. However, it also requires extra maintainability, as additions or changes to existing data may need to be thoroughly examined and demand extra configuration to make sure nothing is lost during the data field transformations or that queries behave as expected.

It is developed alongside Logstash[36], a data collection and log parsing engine, and Kibana[37], an analytics and visualisation platform. The three products can be used together as a complete solution or isolated as modular components in alternate architectures.

5.3.2 MongoDB

FIGURE 5.7: MongoDB logo

MongoDB[38] is a free, open-source NoSQL Document-oriented database. Mongo achieves high performance by making use of RAM and indexing fields with primary and secondary indices.

Data entries are saved as schema-free Documents in BSON, a binary representation of JSON with more high-level datatypes. A Collection is a group of Documents, and a Database groups Collections.

MongoDB supports replication and sharding.

Replication is primarily used to ensure high availability, although it can also be used for horizontal scaling of Read operations. Using additional MongoDB instances as Replica Sets involves the existence of a Primary and one or more Secondary Nodes, on which data is replicated. The Primary Node receives all write operations and propagates them asynchronously to the Secondary Nodes. If the Primary Node fails to communicate with the other members of the set for a period of time, the Secondary Nodes elect a new Primary from the remaining active Nodes. When performing a Read operation, the client can opt to send the operation to a Secondary Node, lowering the load on the Primary. The asynchronous nature of the replication can cause Secondaries to return data that does not reflect the state on the Primary, however.

Sharding was later added to distribute data across multiple machines for horizontal scaling. Since MongoDB makes very intensive use of RAM and struggles whenever it needs to swap memory from Disk, this solution makes it much more adaptable to high throughput operations or very large datasets. It uses a combination of Shard servers to store the divided collections and Routing servers to direct clients to the correct server. Each Shard server is responsible for managing writes related to its share of the data, making distributed reads and writes possible.

If a Shard server becomes unavailable so does the data it holds, although the remaining shards can continue to function, so the cluster can handle partial read/write operations. Sharding and Replication can be combined to make a distributed, high availability setup.

5.4 User Interface

The User Interface is the means by which the User views the results generated by the CEP Module. Portolan already features a Web GUI for Data Browsing and high level

management, developed in HTML and JavaScript, which interacts with the Data Storage through Django.

Instead of creating a whole new interface, the Product Owner wished for the new View to examine the CEP Module's results to be integrated into the existing User Interface.

Research was conducted to better understand the Django framework and Javascript, along with the libraries and apps used in the existing product.

5.4.1 Django

FIGURE 5.8: Django logo

Django[39] is a free, open-source web framework developed in Python. Its main selling point is accelerating development through taking care of much of the hassle usually associated with Web development.

The core Django web framework is an implementation of the Model-View-Control paradigm.

- **Model**: It fulfills the functions of an Object-Relational Mapper (ORM), mapping data models defined as Python classes into tables and entries in a Relational Database backend. While Django's ORM is meant to be used with Relational backends, Django Apps consist of arbitrary code, which makes it possible for Portolan to use MongoDB through the PyMongo driver.
- **View**: It processes HTTP requests and can generate views from page templates.
- **Control**: The Control layer is where Django affords most of its flexibility. Django provides a regular-expression based URL dispatcher, which allows the definition of static and dynamic endpoints. A valid endpoint ends in a Class or function call, and the request is then processed by arbitrary Python code. The request data is available during processing, making it possible to bring about different behaviors depending on the type of request, data in the request itself, URL arguments, etc.

Django can be extended with Django Apps, Python packages that add extra functionalities. Each Django app has its own internal URL mapping, and the root Django instance must have a base URL mapping for each app. Some Apps are available as part of the Django release, providing solid solutions for general use cases such as User Management and Authentication, Session Management, serving an Administration Dashboard and Security (mitigating typical web attacks such as SQL injection, cross-site scripting and cross-site request forgery).

To aid development efforts it also has an interface to Python's built-in unit test framework, validating a newly added or updated app before it is deployed or on demand.

Chapter 6

Architecture

This chapter details the Architecture Design of the CEP Module. It was designed taking into account all the conceptual and component-related data presented in chapters 3 and 5, in fulfillment of the requirements listed in Chapter 4.

First, the Architectural Styles identified previously are compared and one is chosen as the model paradigm for this project.

Then, a contextual view of the whole system is presented, showing where the newly made CEP Module fits and how it interacts with the Stakeholders and existing system.

Finally, a component is chosen for each available slot and the container view of the CEP Module is presented.

6.1 Architectural Style

As detailed in Chapter 3, there are currently 2 standard Architectural Patterns for Data Processing systems: the Lambda and Kappa Architectures.

The main tradeoff between them is the existence of a layer dedicated to Batch Processing in the Lambda Architecture, with the Stream Processing layer used to provide results in-between executions of Batch Processing. The Kappa Architecture, however, features only Stream Processing.

That makes the Lambda Architecture generally more indicated towards generating Business Intelligence. In cases where the volume of data is too large, or accurate computations require all historical data to be reprocessed, Stream Processing may simply not be an alternative to Batch Processing, as the entire context of the data may be required for producing the final views and it cannot generate the same kind of results. One example of such a case is the training of business oriented Machine Learning algorithms, which is often done through Batch jobs with complete Datasets.

The Kappa Architecture is more lean, since it doesn't have two kinds of Processing engine. It is a simpler codebase to maintain, and a lighter technical stack. It still retains the capability of reprocessing historical data if required, and the Processing Frameworks researched are capable of maintaining multiple dataflows at once. It is indicated for generating time-sensitive results, which makes it ideal for decision support and Operational Intelligence.

Both Architectures are based off of microservices, making it so components can be individually scaled according to necessity. As stated previously, the core components that are expected to take the heaviest load (Data Ingestion, Processing Engine and Data Storage) are scalable by design.

Taking the above tradeoffs into account, and the fact that this system is to be used for supporting operational teams during research and incidents, data has to be available readily (with top latency of a few seconds) the best fit for this system is the Kappa Architecture.

6.2 Context View

The CEP Module is a new addition to the pre-existing architecture. The Context View represents the neighboring systems and components, the contextual placement of the CEP Module and its additions and the interactions between them and the stakeholders.

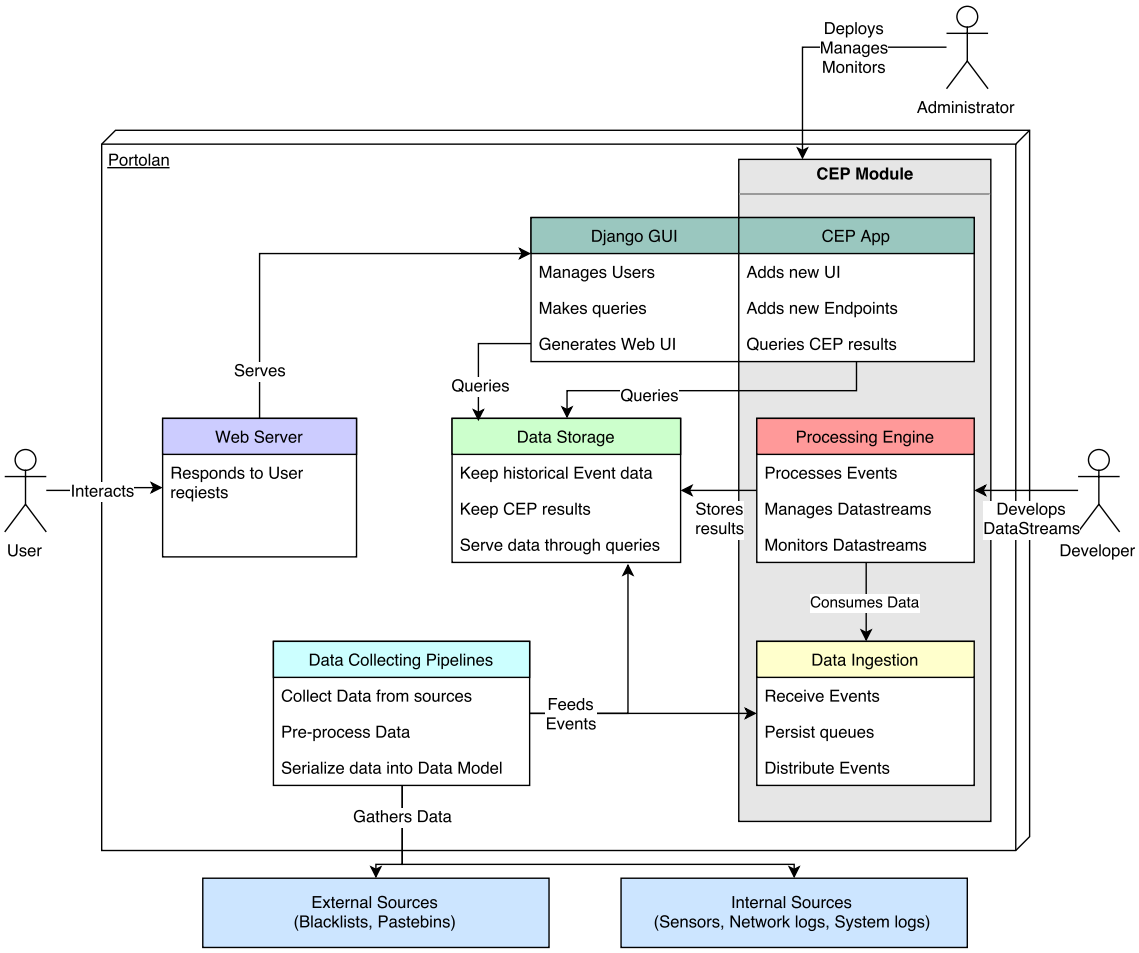


FIGURE 6.1: Context View

6.3 Components

6.3.1 Data Ingestion

As can be seen in table 6.1, the main differences between Kafka and Redis reside in their storage media and clustering capabilities.

For the CEP Module to be a success and comply with the requirements defined in Chapter 4, the system must be horizontally scalable to grow in response to collected data growth. As such, the component responsible for Data ingestion must be capable of high throughput. Both Redis and Kafka qualify, although Redis is faster due to being mainly memory-based.

However, RAM is not cost-effective and is much harder to scale up. Plus, it is volatile, meaning data is lost in case of a failure, and Redis' solution for data persistence is an eventual dump to disk, which may lead to lost data. On top of that, Redis does not enforce order guarantees, meaning that events may be delivered out of order. That is not

Kafka	Redis
Data is persisted on disk, so some speed is sacrificed in exchange for extra storage capacity.	Memory-based. Very fast, but highly limiting if records are large or in enough volume. Persistence can be enabled, but is not as reliable as Kafka and decreases performance.
Can hold records for a longer retention period thanks to cost-effective storage media in case events need to be reused.	Short retention period due to being limited to RAM.
Messages are stored as streams of bytes, which permits Kafka to have more control over the structures on disk and increase performance.	Can handle high-level data structures (Strings, Hashes, Lists, among others) to increase effectiveness in memory access and usage.
Partitions and replicates Topics over multiple servers, allowing distributed, parallel access and high availability setups.	Only capable of Master-Slave replication, with no parallel access to the data.
Kafka has strong order guarantee, meaning that messages will be delivered in the same order they are received.	Redis does not ensure order, making use of eventual consistency to increase performance.

TABLE 6.1: Main differences between Kafka and Redis

acceptable in this context, since the CEP Engine will require events to be delivered as orderly as possible and may require reprocessing recent events when a dataflow is newly developed or created. It is expectable for there to be multiple dataflows consuming the same type of events, which favours Kafka further due to its capability for parallel access.

Kafka has the additional overhead of requiring Apache ZooKeeper to store and synchronize configurations and state (even in standalone mode). In exchange, ZooKeeper eases the deployment of a Kafka cluster by taking care of synchronizing state and leader election for a simple implementation of a robust High-Availability setup, which is an acceptable tradeoff.

While Kafka puts the system under a heavier load than Redis, it boasts features more adequate to the use-case than Redis while maintaining performance and scalability[40].

For the reasons stated above, Kafka was considered the best choice for data ingestion and distribution.

6.3.2 Data Processing

The Complex Event Processing (CEP) Engine is the main component of the system, as it is responsible for the vast majority of Data Processing for enriching, correlating and generating complex events based on the input data.

The three technologies fit this purpose. All three are highly scalable, capable of high throughput, and programmable, allowing data to be processed in arbitrarily complex ways.

Part of the Problem Statement is that events are delivered in a Stream. While they could be collected into DataSets to work with Spark Streaming, comitting to a Batch-only Engine would cause a number of issues. Each new batch would cause the scheduling of a whole new job, which would be exacerbated by the small size and increased frequency of micro-batches; in cases where each Event generates an individual output, the size of the results would be tied to the size of each batch; keeping state between batches can be an issue, which makes pattern matching through CEP difficult; due to each batch being

	Flink	**Storm**	**Spark**
Processing Model	Event and Batch	Event	Micro-batch
API	Declarative	Compositional	Declarative
State Management	Distributed Snapshot	State Acknowledgment	Checkpoints
Strictest Guarantee	Exactly-once	At-least-once	Exactly-once
Out of order Processing	Yes	Yes	No
Latency	Milisecond to second	Milisecond to second	Seconds
Autoscaling	No	No	Yes
Well Documented API	Yes	No	Yes
Strong native framework	Yes	No	Yes

TABLE 6.2: Main differences between Flink, Storm and Spark

self-contained, taking advantage of time-windows would be complicated, as they would largely be restricted to the time interval of events available in each batch.

For the above reasons, Spark was no longer considered a viable solution.

The decisive differences between Flink and Storm are the quality of the documentation and the functionalities offered natively by the framework itself.

At the time this decision was made, Flink had considerably better documentation, which was expanded with the functionalities added in major releases since then. Storm, however, was poorly documented. It was enough to get started, but did not give a sufficient overview of the system, its functionalities and inner workings, making developing and optimization difficult.

Both were capable of keeping state while processing, but Storm had been designed primarily for ESP, and its acknoledgement system for stateful processing greatly decreased its performance. Flink, however, was designed from the beginning to be capable of keeping state. When it comes to stateful computation, Flink boasts better performance with its distributed snapshots than Storm with acknowledgements enabled, as can be verified in Data Artisans' extension of the Yahoo benchmark[41]. Losing stateful processing in exchange for better performance was not a viable option, so Storm's better performance with acknowledgements turned off was not an important factor.

Finally, Flink features optional APIs for CEP, graph analysis (Gelly) and machine learning (FlinkML), all of which make use of native capabilities. An API was added in version 1.3.0 to make Storm Topologies compatible with Flink Streaming and allow reusing code that was initially implemented for Storm. It is still in an early stage, but allows both the use of Storm Bolts/Spouts in a Flink Streaming program and the execution of a whole Storm Topology with minor changes.

Those reasons made Flink the most adequate choice for the CEP Engine.

6.3.3 Data Storage

While ElasticSearch offers a fresh array of functionalities characteristic of a Search Engine, the automatic optimizations performed by ElasticSearch to ease full-text search could result in unexpected errors or loss of data, meaning it would require additional

planning and meticulous configuring. It was thus deemed that replacing MongoDB would require too much effort and time budget, so it was considered out of scope. It could have been used exclusively for the CEP Module, but even then Elastic requires considerable effort to optimize and would lead to an addition to the technical stack of the product.

Furthermore, not all deployments of Portolan are large enough to justify the deployment of a completely separate DBMS for a module.

For those reasons, the Data Storage solution chosen for use by the CEP Module was MongoDB. However, care was taken to make sure the Data Storage can be switched in case an alternate solution is developed later on, making change easier in the future.

6.4 Container View

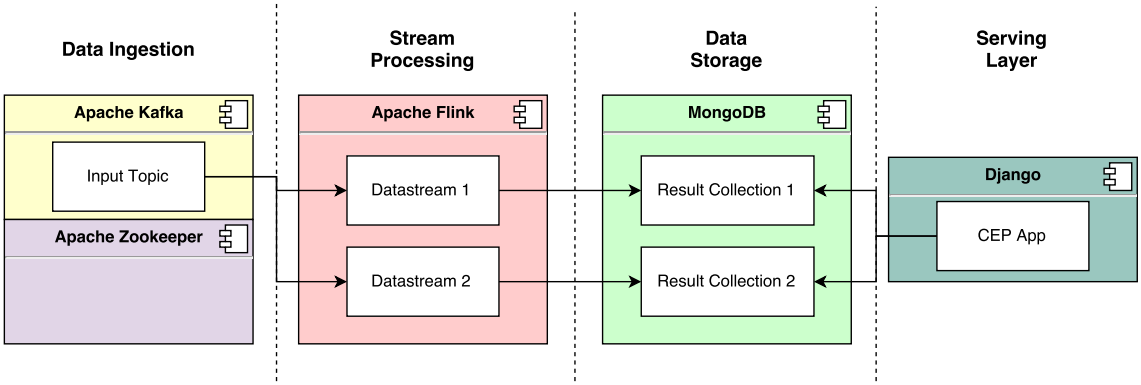


FIGURE 6.2: Container View

In accordance with the Requirements specified in Chapter 3, the CEP module will make use of open-source components. This view shows the component structure and the flow of data between components.

The Component distribution and interaction follow the Kappa Architecture design pattern.

Each component is individually capable of horizontal scaling, and the Microservices-style Kappa Architecture ensures that each component is detached enough from the others to ensure that changes to one do not directly affect the others.

There was need to research and experiment with configurations for each component, establishing interfaces between them and define the Data Models used for communication between components, storage and presentation.

Chapter 7

Implementation

This chapter addresses the implementation process and the steps involved in setting up, working with and managing the CEP Module, its components and User Inteface. The Implementation of the system was performed in the Second Semester of the Internship.

Through this Chapter the reader should understand how the system components were deployed, how they interface with each other and the issues that were encountered and solved.

Unlike most, this Internship focused heavily on the design and implementation of an architectural solution, integrating multiple different components into a microservices-based architecture instead of the development of a particular artifact. As such, most of the code developed for the Data Processing section serves to bridge and test the components' interfaces.

7.1 Overview

The Implementation process can be divided in two main phases, the implementation of the Data Processing Module and the development of the User Interface and its interactions with the Data. The definition of the Data Model was a process of its own, done in parallel with both phases.

In accordance with the initial Chronogram for the Second Semester depicted in Chapter 2, the Implementation phase had an initial buffer to set up the Development environment and Testing details. However, due to the delay in obtaining the Virtual Machine for development, those actions were delayed until it was made available.

As part of the mitigation plan, all actions relating to the Development environment were delayed in favour of those that could be achieved solely with the available workstation.

First, auxiliary tools for data exploration, generation and testing of the final system were developed. Using the tools to explore the data currently gathered by Portolan, a compatible Data Model was defined to store the results from the CEP Module. The Data Model suffered alterations during the User Interface phase to make presenting data in a useful manner possible without extra input from the User.

Implementing all the necessary components required an extra phase of study to achieve proper configurations for the expected setups.

When deploying the components, achieving a setup that balanced the available resources between components adequately was made a priority. Resource mismanagement in a live deployment could lead to severe bottlenecking. To avoid that, each component's effect in the machine's resources was studied to identify what could lead them to perform poorly.

There was a short time interval after which all components except Flink were ready for an initial system test, but the Virtual Machine wasn't available yet. During that time

the Intern studied Apache Flink and the front-end technologies necessary for the User Interface phase.

After the VM was made available Apache Flink was deployed and configured. Next came the development of a Dataflow for data aggregation requested previously by the Product Owner, along with modules to allow data to be ingested from Kafka and later commited to MongoDB.

The User Interface phase required the Intern to read and understand the pre-existing code for Portolan's Django apps in order to use matching coding patterns (to ensure maintainability by company developers) and avoid duplicating code. The Data Model also suffered adjustments to allow viewing records with arbitrary fields in a single interface, which made this phase take longer than expected.

7.2 Hardware

This section details the different environments involved in this Project and their specifications.

7.2.1 Development

The Development Environment consisted of a single Virtual Machine supplied by Dognædis in company infrastructure, according to available hardware. While its specifications are lower than the ones in a Deployment environment, they are significant in the validation of the Module. The Non-Functional Requirements specified in Chapter 4 apply to the superior Deployment Environment, and the research conducted by the Intern (including the component benchmarks referred in Chapter 5) suggested that they were achievable with the Hardware made available by the company (as can be seen in Chapter 8).

Next follow the specifications of the Virtual Machine.

- CPU: 4x vCPU
- RAM: 10GB
- Storage: 256GB

7.2.2 Deployment

The Deployment Environment consists of the specifications expected to be available to this Module in a full deployment, as stated by the Product Owner. The target environment consists of one or more dedicated Virtual Machines in an enterprise infrastructure, with access to and accessible by the machines used by the Database and Web Server components of Portolan.

- CPU: 16x vCPU
- RAM: 64GB
- Storage: At least 256GB

7.2.3 Cluster

In a clustered setup, machine specifications can be adjusted to the individual requirements of the component. For each component, a clustered setup ensures high availability and parallelism. These hardware configurations correspond to the minimum suggested specifications for a single instace of each component in an enterprise-level deployment.

- Apache Kafka
  - CPU: 4x vCPU
  - RAM: 8GB
  - Storage: 128GB, in dedicated storage devices
- Apache Zookeeper
  - CPU: 1x vCPU
  - RAM: 2GB
  - Storage: 64GB, in dedicated storage devices
  - High availability: At least 3 instances in separate machines.
- Apache Flink
  - CPU: 4-16x vCPU, according to task requirements
  - RAM: 16GB
  - Storage: 64GB
  - High-availability: State storage backend in a shared filesystem such as HDFS[42].

7.3 Auxiliary Tools

The auxiliary tools developed at the start of the Second semester served multiple purposes.

In order to fully understand the data to be processed by the system, the Intern needed an understandable view of its scope and variability. The first tool to be developed produced a list of all event sources, the data fields of all events and on a source-by-source basis, and the number of occurences of each field. This provided an overview of the data available, its nature and the fields that could be used for processing later on. The current Dataflow and suggestions for Future Work are based on this data.

A dummy event generator was developed afterwards. It generates a specified number of events, according to a provided initial schema, to aid in load tests.

Finally, three tools were developed to perform read and write operations on Apache Kafka and MongoDB individually and in a single iteration. They take an Event dataset as source and benchmark the read and write speed of each component, and were used to perform these components' benchmarks mentioned in Chapter 8.

7.4 Data Model

Data is exchanged between components in JSON format. During processing in Apache Flink, the Events are deserialized into custom-made Java Objects to ease data manipulation and automate serialization and deserialization.

Defining a Data Model for complex event storage was a significant challenge. Since complex events generated by the Module can have a variety of fields and data, creating a hard schema was not viable without greatly reducing the flexibility of the system. However, being completely unable to predict the data in a record made it impossible to present to the user without significant computing power spent on just-in-time parsing and make future approaches to dynamic correlation impossible.

To solve this issue, a number of field names were declared as reserved, seeing as how they are common or predictable enough to ensure the same data model.

- **_ id**: This field stores a unique ID to the object. It can be used to directly reference the record from an alternate source. It is already a reserved field in MongoDB, but considering that the Data Storage solution can be changed if required, having a field set as unique identifier avoids issues raised in the User Interface from a change in Data Storage.
- **events**: This field holds a list of the simple events from which each complex events was derived. This provides a common point of correlation between complex events.
- **ip**: This field holds one or more IPv4 addresses related to the complex event. It was singled out to take advantage of an IP masking feature that already exists in Portolan, discovered during the study of the core Portolan Django App.

If a record had one of those fields, the interface would be capable of presenting and performing queries correctly. It mimics a light enforcement of schema, which is an acceptable tradeoff for keeping a vast majority of flexibility afforded by MongoDB. In order to differ between complex events, each type is saved in its own MongoDB collection, grouping similar events together. All complex events are stored in the same MongoDB Database.

This solution proved to be enough until the User Interface phase. The next challenge encountered was presenting useful data to the User. Since MongoDB is NoSQL, no schema is enforced by the DBMS. When browsing and querying a collection of complex events in the UI, the User would be presented with a paged list of resulting events. Under the previous Data Model, presenting fields in that list involved either forcing a static set of fields, which decreased usefulness, or dynamically parsing every returned object and capture every field's name, without a way of knowing how valuable it was to the user to begin with.

The solution devised was the creation of an additional MongoDB collection called CEPSchema to store an array containing the names and types of the most relevant fields per complex event collection. That made data easily presentable to the User without additional stress on the Django app or browser-based Javascript and ensured that the User Interface is agnostic to the kind of data being presented, as long as the relevant fields are specified.

Images depicting the User Interface developed for the CEP Module can be found in the User Interface section of this Chapter.

7.5 Data Ingestion

Implementing the Data Ingestion layer involved the configuration of two technologies, Apache Kafka and Apache Zookeeper.

Zookeeper was deployed with minimum changes in default configuration, limiting memory usage to 1GB as a conservative estimate. This value was verified with load tests on Kafka.

Kafka needed a point of integration with the Data Gathering Pipelines from Portolan, which act as the Event Sources. The pipelines themselves are modular, and can make use of connectors to interface with other components, which led to the development of a connector using the kafka-python[43] client. The Connector sends the JSON events created by the pipeline into a configurable Kafka topic.

A Kafka topic named "CEP_ Input" with 4 partitions was created in Kafka to serve as the general event input queue. That approach was selected over individual topics per input type to mirror the current way Portolan stores all events in a single MongoDB collection. Additionally, considering the potential diversity in Dataflows in the Data Processing Layer and the scalability afforded by Kafka, sending all current events through a single topic is a viable choice in throughput and complexity. Replication Factor was set at 1 due to using only a single machine. This setting can be changed later on when a clustered setup is needed.

Docker[44] containers were created to simplify the deployment of both Kafka and Zookeper. The containers make use of a common isolated software-defined network to communicate between each other if deployed in the same machine. In remote deployments Kafka and Zookeeper expose listen ports through a configurable network interface. A URL to a custom configuration file can be provided as an argument when starting the Zookeeper and Kafka containers.

7.5.1 Resource Management

RAM usage was estimated according to the size of the Dataset. Kafka keeps records for a period of two days, after which it starts to clear older records out.

In the Development environment, 1GB of RAM was given to Zookeeper, and 2 GB to Kafka. This was more than enough for Kakfa to handle a month's worth of event data (approximately 193MB) in a single iteration without performance losses.

7.5.2 Best Practices

- Parallel Reads: If Kafka Consumers have different IDs, they are capable of reading the same topic without issue. To have Consumers with the same ID reading from the same topic, the topic should have as many or more partitions than the number of consumers, as partitions are divided between consumers with the same ID.

- Parallel Writes: Similarly to reading, Parallel Writing can be achieved by partitioning a topic. Write operations can be applied to each partition independently. Since the system could not exhaust Kafka's throughput, the current topic with 4 partitions offers parallelism without adding overhead. In testing, overhead due to excessive partitioning was only severely noticeable (over 10 miliseconds of difference) past 8 partitions.

- Replication: In a single machine deployment, having a topic with replication factor greater than 1 serves no purpose, as the extra replicas will just be stored in the same Broker. However, in a clustered setup, replicas are spread over the cluster. The number of replicas should be at most the number of Brokers in the cluster to avoid duplicating data in the same Broker instance.

- Avoiding Contention: Kafka and Zookeeper should be deployed in separate machines, or at least write to separate storage devices. Since both make heavy use of persistent storage, their performance is expected to decrease due to contention between them.

- High-Availability: For high-availability setups, a Zookeeper cluster should be composed of an odd number of nodes, with a minimum of 3. In a distributed setup of N+1 nodes, Zookeeper can tolerate the failure of N/2 instances. A Kafka topic should have a replication factor greater than one. With a replication factor of N and a minimum of N Brokers available, a Kafka topic remains entirely functional if up to N-1 Brokers are lost.

7.6 Data Storage

The resulting complex events are stored in a MongoDB instance in accordance with the Data Model defined earlier.

The configurations of MongoDB were not explored in detail, as it already is a deployed part of Portolan. However, it was tested for performance purposes, as can be seen in Chapter 8.

7.6.1 Integration with Apache Flink

Integrating MongoDB and Apache Flink required the development of a custom Data Sink, due to the lack of an official one.

Apache Flink provides RichSinkFunction, a class to serve as base for user-developed Data Sinks. That class was extended to create 4 different sinks using the MongoDB Java Driver. Two of the sinks are general purpose and just write the received Document objects to a specified collection. The other two sinks were developed to be used with a Dataflow that correlates events by IP and Domain Name, which require Upsert operations instead of simple writes.

Experiments were made with 2 Synchronous and 2 Asynchronous sinks to assess performance and operational differences. MongoDB writing performance is below par, which adversely affected the Data Processing layer's performance. This issue was explored, and a solution had to be devised to ensure the complex events were not lost in case of failure while making sure data could be written to the database without seriously affecting the entire Dataflow.

The performance benchmarks that identified this issue are presented in Chapter 8, along with the benchmark of the solution.

Synchronous Writing

- **MongoDBSinkSync**: This Sink writes records synchronously, one at a time.
- **MongoDBUpsertIPDomainSinkSync**: This Sink synchronously upserts the complex events generated by the first Dataflow developed to aggregate multiple events' data in fewer, more informative records.

The synchronous sinks proved detrimental to performance. The writing speed of MongoDB is superior when multiple records are written in bulk instead of individually, as the current MongoDB setup does not permit parallel writing. However, in case the Dataflow fails (for example, the Apache Flink Taskmanager crashes), the Dataflow progress can be fully recovered through the Flink Checkpointing system, ensuring exactly-once execution per record.

Asynchronous Writing

- **MongoDBSink**: This Sink writes records asynchronously. The MongoDB Asynchronous Driver creates a thread that gathers records in bulks and writes them asynchronously, communicating success through a callback function.
- **MongoDBUpsertIPDomainSink**: Like its Synchronous counterpart, this sink was customized to upsert complex events for the IP-Domain correlating Dataflow.

The asynchronous driver showed a slight improvement in performance. However, it breaks the Checkpointing system, as the Sink task ends before the record is confirmed as being written.

Solution

This issue was solved by using Apache Kafka as a middleware solution for records written by Apache Flink. Instead of having the main Dataflow write directly into MongoDB, it instead writes records to Kafka at a highly increased rate. This solution caps the Data Processing Engine's performance, and Kafka is capable of delivering and ingesting events at a rate higher than they can be processed.

Another Dataflow was developed solely to read records from a newly created "CEP_Output" Kafka topic and write them to MongoDB. In case of failure, the finished events are persisted by Kafka and the bottleneck imposed by MongoDB on the Dataflow is avoided.

7.7 Stream Processing

This section describes the functioning of Apache Flink, the developed Dataflows and their purpose.

7.7.1 Management

Flink offers a Web Interface for management purposes.

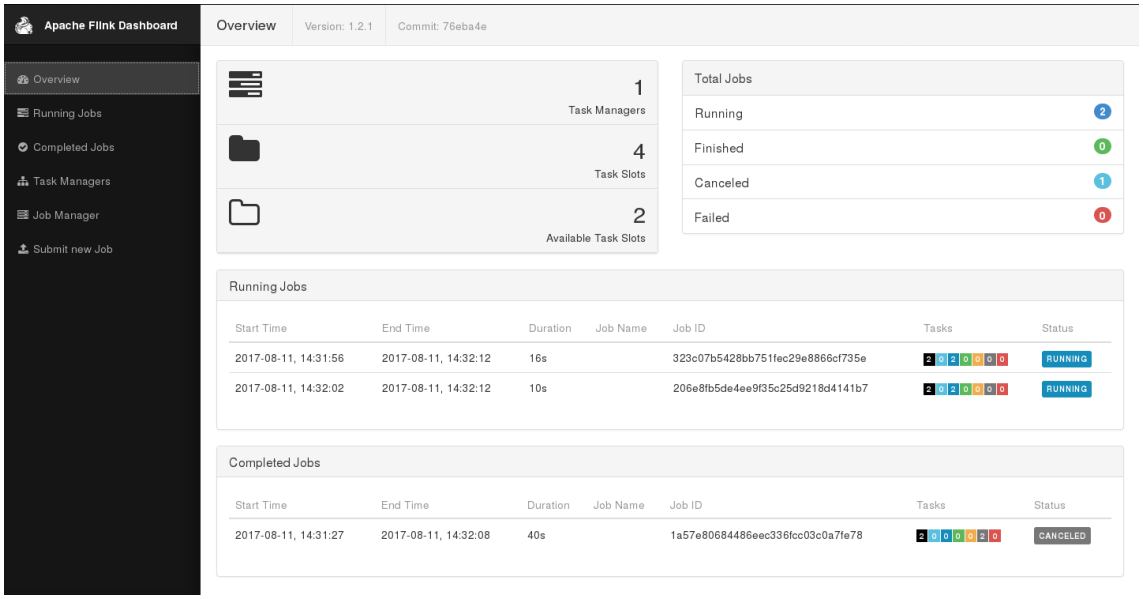


FIGURE 7.1: Flink Dashboard

The main dashboard shows an overview of the framework's state, listing the total and available number of Taks Slots, Running and Completed jobs and their state.

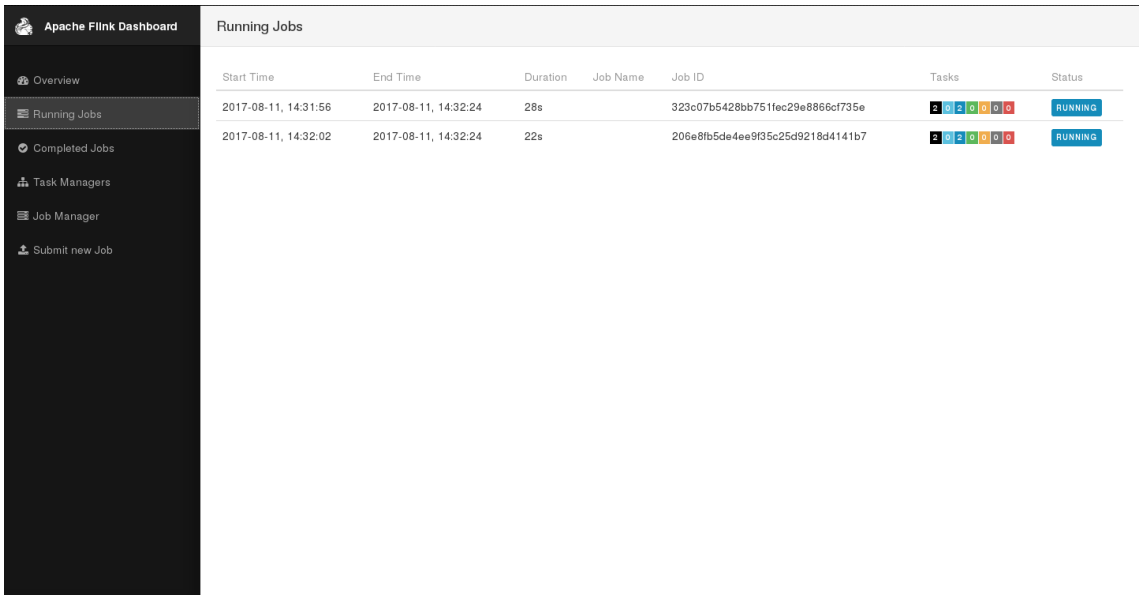


FIGURE 7.2: Flink Job List

A full list of running and past jobs can be obtained, and its details can be consulted as necessary, showing a graphical representation of the Dataflow and statistics such as elapsed time, received and sent data, among others.

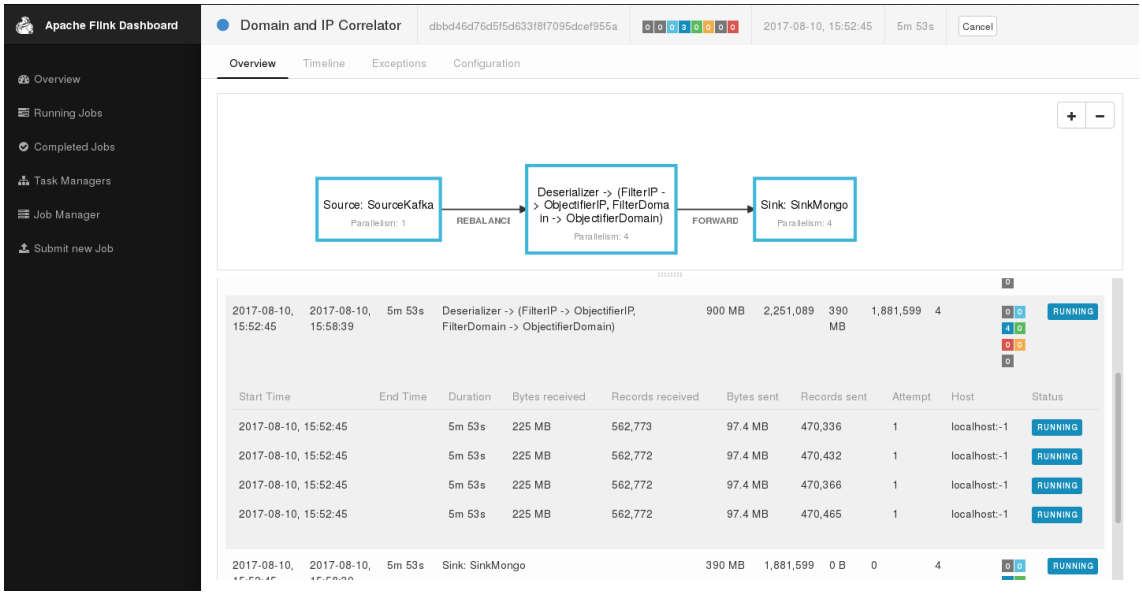


FIGURE 7.3: Running Flink Job details

Jobs can be submitted to Flink through the Web UI, a CLI utility or a REST API endpoint.

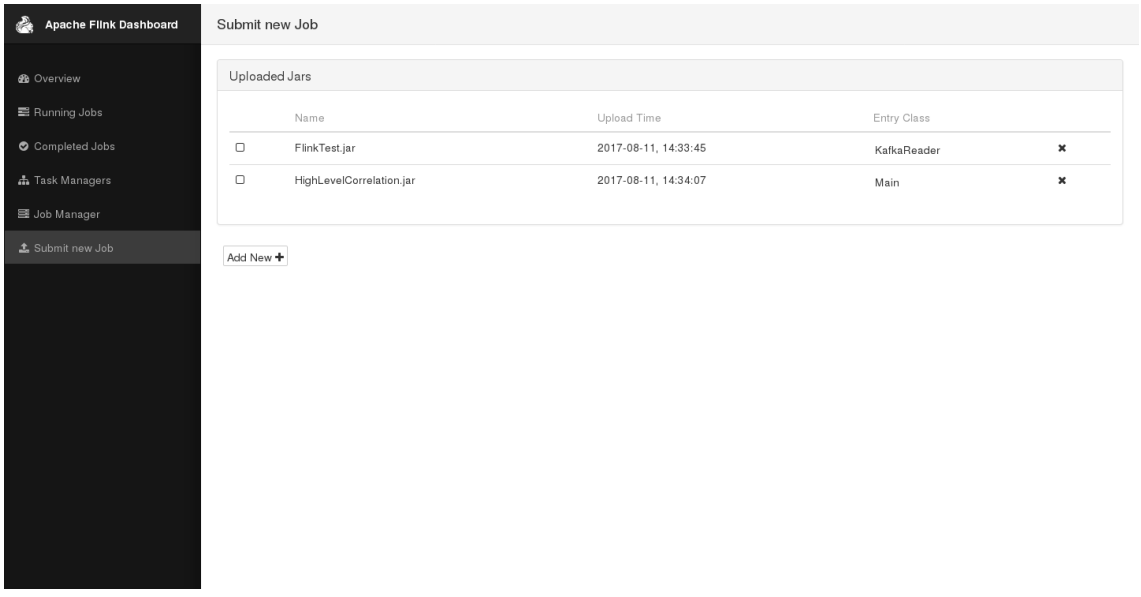


FIGURE 7.4: Uploading jobs to Flink

When submitted through the Web UI, the Job's Jar is stored by the Job Manager and can be executed from the UI. Arbitrary parameters can be specified at will, and the Apache Flink Java Library provides a Parameter Parser. Parallelism and Entry Class are parameters native to the framework. An alternate Savepoint path for the Checkpoints can be specified, if there is a need to store Checkpoints in any specific path or shared filesystem.

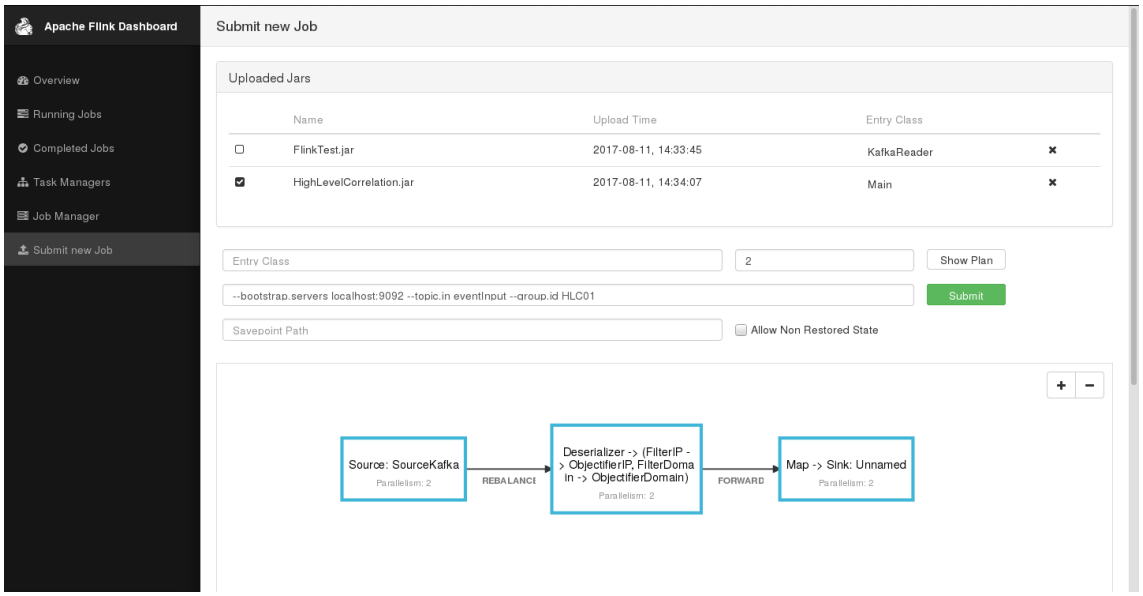


FIGURE 7.5: Submitting an uploaded Job for execution

7.7.2 Resource Management

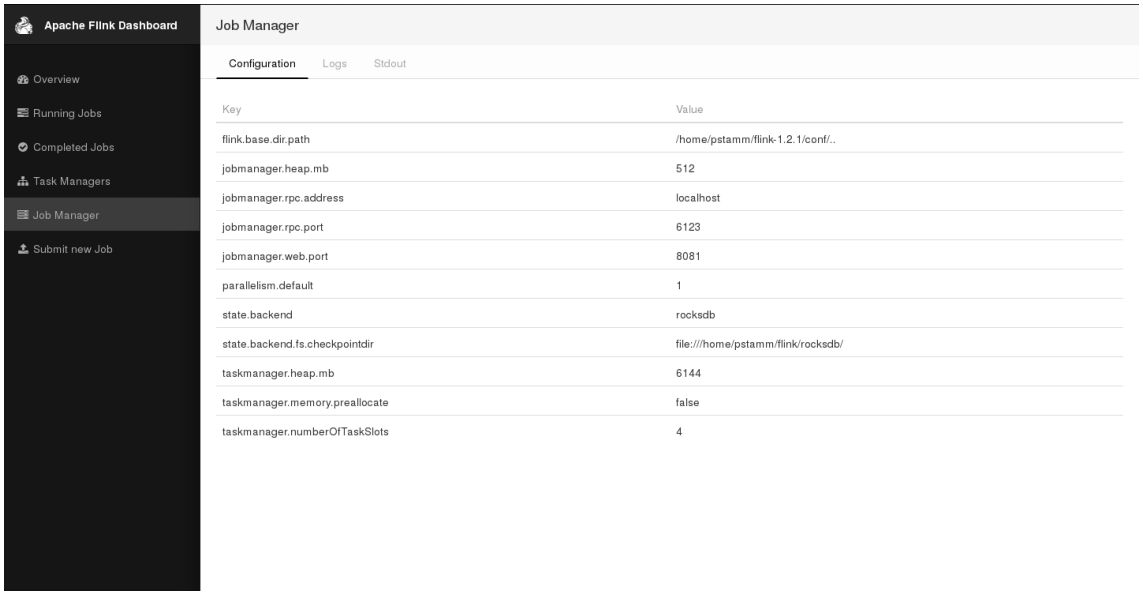


FIGURE 7.6: Job Manager Resources

Flink operates with two main management processes.

The active Job Manager handles Job submission and distributes tasks across Task Managers with free slots. Any extra Job Managers in the cluster take a backup role, with only a single one being active at any time. When the active Job Manager fails, the ones in backup duty hold a Leader Election and one becomes the new active leader.

Task Managers receive tasks to be executed in available task slots. The number of task slots in a Task Manager is configurable, but to the Job Manager Task Managers and Slots serve as a global resource pool.

In the Development Environment setup, Flink has a single Job Manager and Task Manager. Approximately 6.5GB of RAM are managed by Flink. 6GB are used by the Task Manager due to the fact that it performs the Data Processing jobs. The Job Manager does

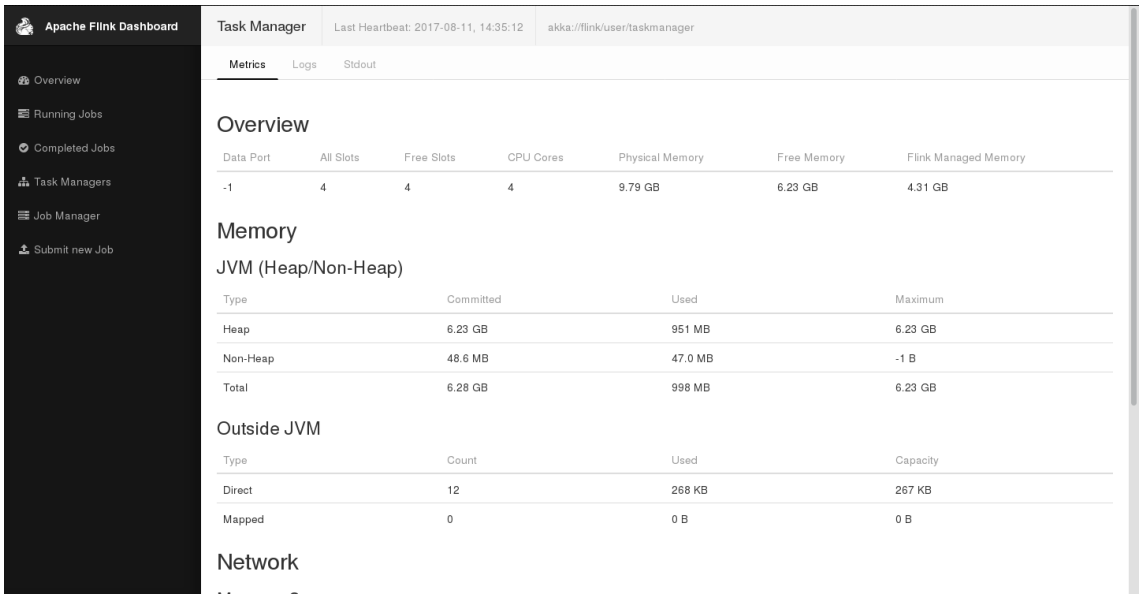


FIGURE 7.7: Task Manager Resources

not require much RAM and can handle this setup with 512MB of memory. This was done due to the fact that MongoDB requires RAM to function properly, so the entirety of the machine's remaining RAM could not be reserved for Data Processing in Development.

7.7.3 Monitoring

The status of the active Job Manager and available Task Managers can be monitored through the menus shown previously. All Task Managers are listed and can be consulted separately.

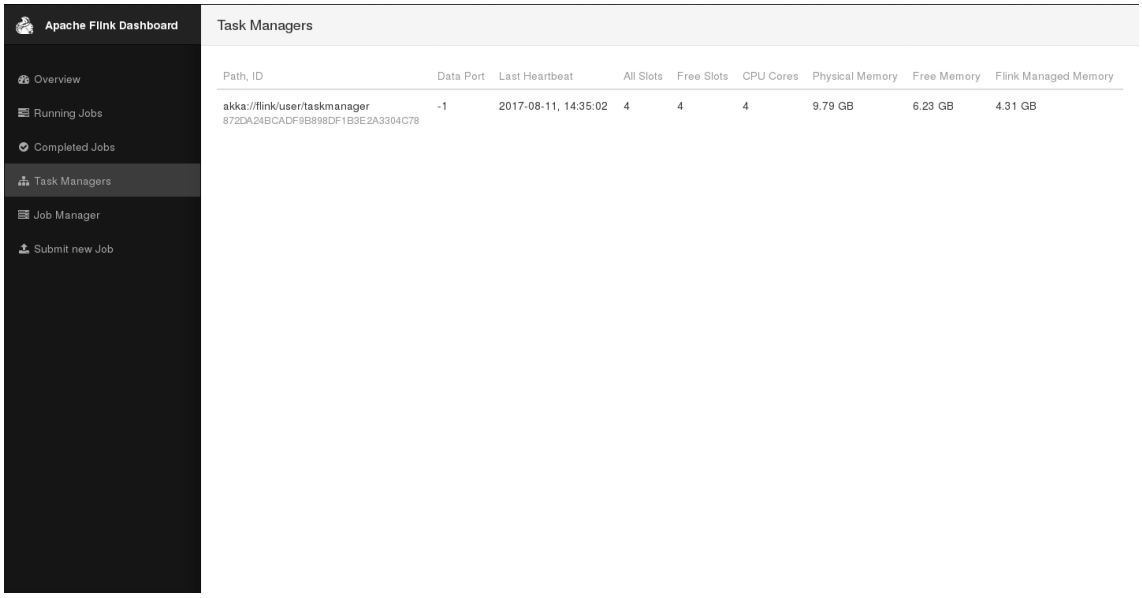


FIGURE 7.8: List of Task Managers in the cluster

For running jobs, the Web UI offers a number of statistics, with accompanying graphics.

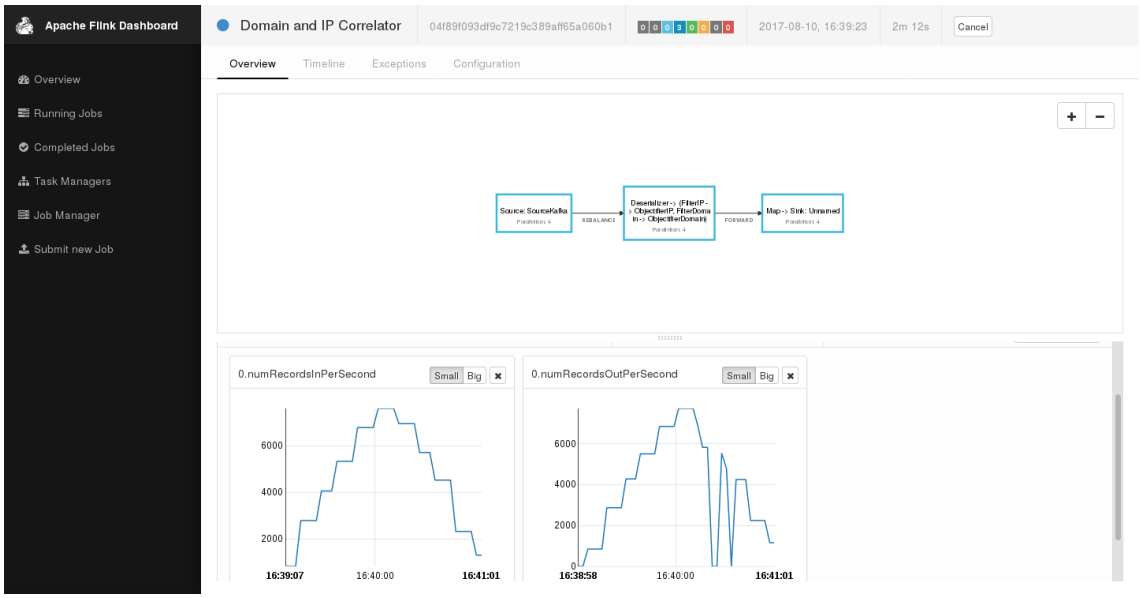


FIGURE 7.9: Graphic metrics on a running Job

Finally, Flink has a REST API from which the status of the entire cluster can be checked in detail. In accordance with standard practice, REST API calls return JSON objects.

7.8 IP-Domain Correlation Dataflow

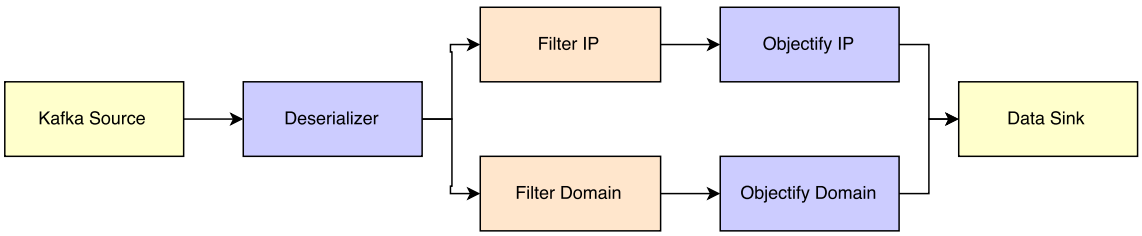


FIGURE 7.10: IP-Domain Correlation Dataflow

This Dataflow consumes JSON Events from a general input Kafka topic, deserializes them, and then splits the records over two separate branches, one for records with an IP field and the other for records with a Domain Name or URL. Each branch uses a filter to discard unwanted records. The validity of the IP address and Domain Name is checked, and then a complex event is created as a Java Object, based on the data of the initial record.

Finally, the event is then fed into an output Kafka topic in JSON format to be later inserted in the Database by another Dataflow.

This Dataflow went through multiple iterations and served as the base to test Flink's performance with various constructs. The most expensive operations in it are the de-serializations and serializations. However, they are unavoidable in order to maintain interoperability between systems and ease integration with other systems later on.

The setup with best performance had the Kafka Source with no parallelism and the remaining tasks with Parallelism 4, making maximum use of the 4 vCores available to process data in parallel. The Kafka topic used to store the resulting complex events was

created with 4 partitions, allowing the 4 instanced Kafka Sinks to write simultaneously. This configuration exhausted the Data Processing layer, causing it to apply backpressure on the Kafka Source to lighten the event influx. The resulting throughput exceeded the minimum event throughput specified in Chapter 4, as can be seen in the benchmarks referred in Chapter 8.

7.9 Database Storage Dataflow

This Dataflow was implemented to grab data from Kafka and write it into a specified MongoDB Database.Collection. It was created because MongoDB's writing speed did not meet the throughput requirements. This way the architecture becomes more generalistic, the events can be reused if need be (for example, for a Dataflow that monitors network connections in search of suspicious IPs or Domains), and the Data Processing Dataflows avoid being bottlenecked by a slow sink. The throughput and latency requirements for the Data Processing component are also fulfilled.

The current event throughput is still incapable of stressing the resources available to Apache Kafka, and this means that the system, in the inferior Development Environment, is fully capable of processing the Event volume corresponding to a full month in about 10 seconds.

7.10 User Interface

The User Interface for interacting with the results produced by the CEP Module was developed in Django and JavaScript (JQuery and Bootstrap) by extending the existing Portolan interface, both in views and functionality.

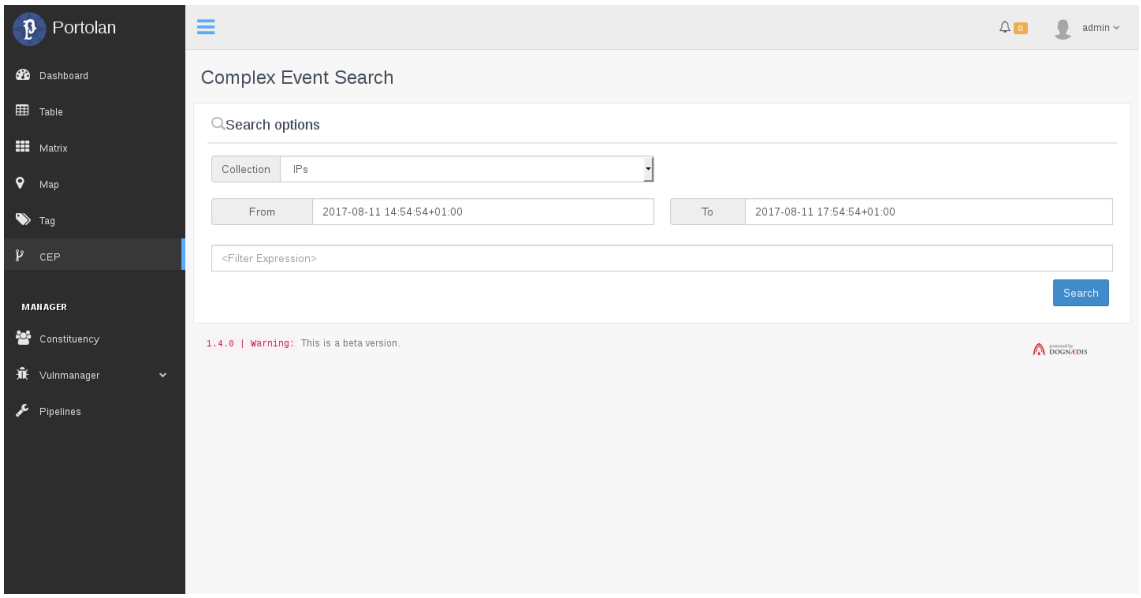


FIGURE 7.11: Complex Event Search menu

This interface is meant for the User to query events within a specified timespan. As specified in the Data Model section, complex events are split over multiple collections. All collections in the CEP Database are listed in the drop down menu, and the User can select which Collection he wishes to query. The Filter Expression textbox allows the user

to specify additional filters through binary operators. The validity of the expression is verified in JavaScript.

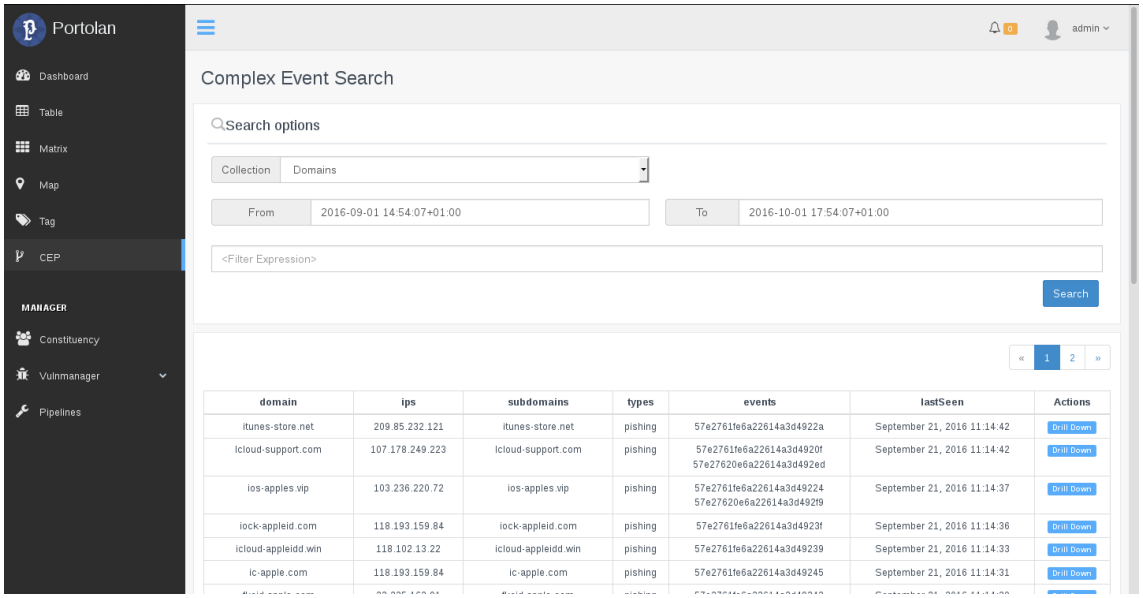


FIGURE 7.12: CEP data being presented to the User

When the User performs a search, a request is made to an endpoint in the Web Server with the Query Parameters. The query is then executed by the Django Control layer through the PyMongo driver.

Instead of querying for the full set of fields, the Mongo query uses the *$project* MongoDB operator to present only those specified in the **CEPSchema** collection.

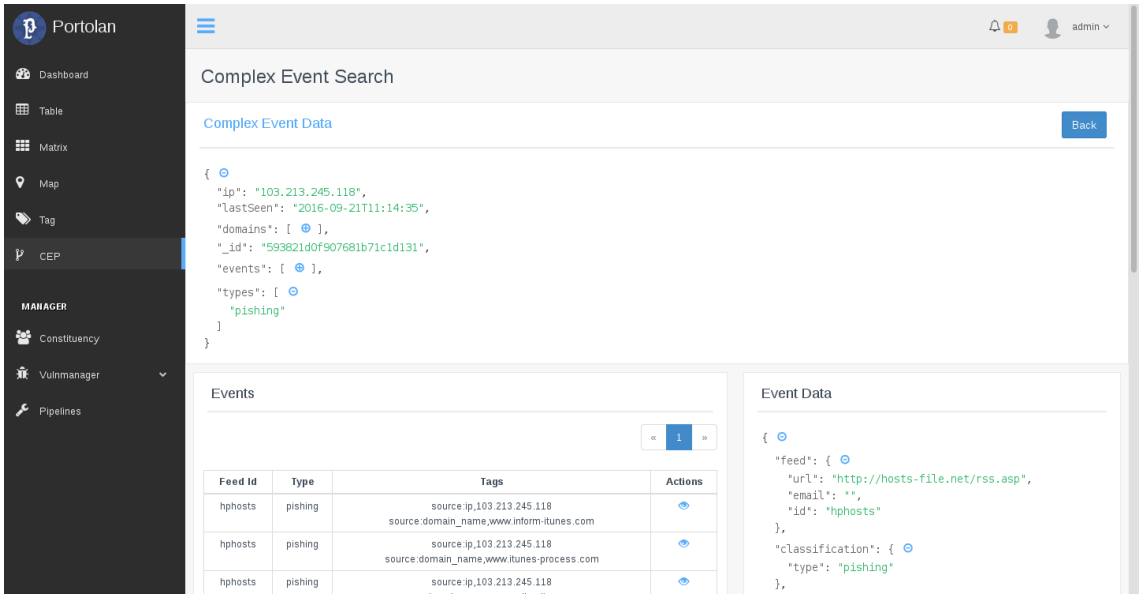


FIGURE 7.13: Details of a complex event in the Web UI

The **DrillDown** button allows the user to see an event in Detail, listing all of its fields in JSON format. Additionally, this menu crosses information with the raw event Data and allows the User to browse the simple events related to the active complex event.

The user interface is deployed as a Django App. It can be added or removed from Portolan with no consequences to pre-existing parts of the product.

The CEP App shares dependencies with the core Portolan App for parsing queries and page templates.

Chapter 8

Verification and Validation

This chapter will contain the verification and validation of the system.

Verification consists of making sure that the artifact produced was properly developed through testing, collection and analysis of metrics and management of technical debt.

Validation means confirming whether or not it serves the purpose it should, analysing which Functional Requirements were fulfilled and unfulfilled, if the Quality Attributes were satisfied and their overall impact on the end result.

It should be noted that all tests were conducted in the Development Environment described in Chapter 6, as an equivalent to the Deployment Environment was not a possibility. This was already an expected occurrence since the Requirements phase, so all Performance-related requirements were designed with that in mind.

In order to be representative both in volume and content, the Data used during testing was the data gathered by Portolan in September, totalling 311.792 Events of assorted variety (blacklist entries, vulnerability reports, etc.) in JSON format.

In cases where extra events were required, such as the more extreme load tests, the Dummy Event Generator detailed in Chapter 7 was used to create new events, which were then appended to the representative Dataset. The generated events included enough data to make sure they were processed by the IP-Domain Correlation Dataflow. This was done to show that the system is capable of handling high volumes of Data, even in hardware with lower specifications than the expected deployment environment.

8.1 Verification

Since the development process chosen was heavily iterative, Verification consisted of a number of tests during and after each development phase, according to necessity and components affected.

The results obtained are referenced and discussed in this section. More details can be consulted in Appendix B.

8.1.1 Data Sources

Since they are external to the system, the Data Sources were not subjected to testing. For the testing of other components, they were replaced by injecting the test Dataset directly into the system.

8.1.2 Apache Kafka

Apache Kafka was subjected to a Performance Benchmark, Broker Failure Simulation and Zookeeper Failure Simulation. More detailed information can be found in B.

Performance Benchmark

This Performance Benchmark serves the purposes of ensuring that this component performs well enough to make complying with the Requirements set by the Performance Quality Attribute possible (QR01 and QR02) and knowing the practical throughput limits achievable. The Kafka Consumer provided by Apache Flink can be configured the same way as the official Kafka Consumer written in Python, so the configurations tested can be used in an Apache Flink Dataflow.

All tested configurations exceeded the expectance of 20000 set in QR01 by a large margin, capping at an average of 46123 events per second. Similarly, response time proved staggeringly below the maximum threshold, averaging as low as 0.009 milliseconds when consuming a single event at a time. Latency increased with the time interval allowed for polling but did not surpass 3 milliseconds in testing.

From these results it can be concluded that, even in non-ideal conditions, the current Kafka setup does not bottleneck the system and complies with the Performance Quality Attributes.

Broker Failure Simulation

This test involved simulating the shutdown and failure of the Kafka Broker, both in standalone and cluster setup. The test was conducted for a single Apache Kafka Broker in a standalone setup and two clustered Brokers. This test aims to test the data persistence and availability concerns of Kafka.

In a single-Broker deployment, data that has been received and flushed to disk is persisted and is not lost upon Broker failure. It should be noted that the flush interval can be customized. A flush can be triggered at periodic intervals and by receiving a configurable number of messages before the time interval has passed.

The cluster setup shows that Kafka can handle a failure of a broker, as long as an alternate Broker with a replica of the target topic is available. Consumer and Producer instances become aware of the cluster layout after initializing a connection and are capable of switching Brokers according to the cluster's health, mitigating the effects of a Broker failure.

The message broker did not lose committed data, and in a clustered setup managed to continue serving it as long as a partition was available in another Broker in the cluster. While the test was executed on a small cluster, Kafka's mechanisms for data management ensured that data confirmed as delivered was not lost and can remain available even with the failure of the Broker instance with leadership status for a partition.

Zookeeper Failure

As the backbone of Kafka, a Zookeeper failure can be catastrophic. Without an available Zookeeper instance, Kafka is not capable of updating its own data, such as cluster layout, configuration changes, locating Topic partitions and Leader Election, all of which are vital.

The tests for Zookeeper failure involved a standalone and clustered setup for Zookeeper, a single and two Broker cluster setup for Kafka, and the Consumer/Producer Java Program. They were meant to assess the impact of Kafka losing connectivity with a functioning Zookeeper instance.

The results show that a Zookeeper ensemble can resist node failures as advertised, as long as over half of the total expected nodes remain available. Both Zookeeper and Kafka showed no problems in this situation.

In the case of a failure of the Zookeeper ensemble, the Kafka Cluster still retains the capability for receiving, serving and persisting data. This limited functionality mode affords extra time to address the issue. This method of failure also ensures that an ensemble cannot be split into two clusters functioning separately with conflicting information.

Zookeeper nodes can reconnect to an existing ensemble to restore it to an healthy state or recreate it if it was down. Asides from Zookeeper instances needing external intervention (manual or tool-assisted) to restart, a cluster is self-healing as soon as connections are established.

These results show that, according to the technology's specifications, Zookeeper becomes more reliable with the addition of independent nodes. In case of failure, Kafka is robust enough to retain some functionality, even if just to buy time while the Zookeeper ensemble is brought up.

8.1.3 Apache Flink

Apache Flink was submitted to Performance Benchmarking and Manager Failure Simulation.

The custom-developed modules of the Dataflows were only submitted to Unit Tests due to their individual simplicity.

Performance Benchmark

The Performance Benchmark serves to prove that the Processing Pipeline complies with the Quality Attributes QR01 and QR02, defined in Chapter 4.

The developed Dataflow read events from a Kafka topic with a single partition, preloaded with the monthly event Dataset made available by Dognædis. The processing pipeline was parallelized, for a total of 4 concurrent pipelines. The Data Source was configured with parallelism 1, so that all 4 pipelines would draw events from the same Kafka Source. A different setup was tested for each Data Sink available: the custom-developed Data Sinks for Mongo (synchronous and asynchronous) and the provided Kafka Sink.

Both MongoDB sinks proved slow and ended up bottlenecking the rest of the Dataflow. The synchronous version was capable of writing an average of 4344 records per second, while the asynchronous version raised the value to 6632. The number of events entering the processing pipeline was greater, averaging 5196 and 8036 respectively. The Data Sink received less events due to a previous task that discards unwanted events. However, both values fall short of the established Performance Requirements, and analysis of the data made available by Flink showed that the Data Sinks were responsible for backpressuring all previous tasks, forcing the pipeline to slow down its event consumption and processing rate to avoid overloading the MongoDB Sinks. The latency in event processing was minimal, respectively averaging at 0.23 and 0.15 milliseconds.

Using a Kafka Sink proved much more effective. The Sink was capable of writing an average of 26920 events per second to Kafka, while the Processing tasks achieved a throughput of 32356 events per second. Unlike the Mongo Sinks, this setup caused the Processing tasks to backpressure the Kafka Source. The system reached its maximum processing capabilities and had to stifle the volume of incoming events to avoid overloading the processing tasks. The latency in event processing averaged at 0.05 milliseconds.

Failure Tolerance

The tests for failure tolerance were performed with an active Dataflow and were meant to test the resilience of Task Managers. To simulate a failure, the Task Manager

process was forcibly shut down, since closing it properly would cause it to deregister from the list of available resources before shutting down.

Shutting down a Task Manager with occupied Task Slots caused Flink to keep the interrupted task on hold. If a suficient number of Task Slots became available, whether immediately or eventually, the Task was restarted from the latest available Checkpoint.

These tests show that Apache Flink is capable of handling the failure of worker nodes without affecting any other part of the system. Both the Task and its state were kept by the Job Manager, and could be restored if assigned Task Slots failed. In the final product, Tasks are possibly expected to run with no foreseeable endpoint. The system's capability to redistribute the Task across available slots is important, as it reduces the need for manual intervention and ensures that the processing is stateful, even through failures.

8.1.4 MongoDB

MongoDB was subjected to a Write Performance Benchmarking. Initially, this test was not planned. However, since MongoDB was revealed as a bottleneck in the Development Environment, a short study was conducted to find the engine's limits in this kind of setup and means to get better performance.

Write Performance Benchmark

MongoDB shows considerably better speed when writing in Bulk, averaging nearly 15.000 records written per second. The major problem when writing into MongoDB as part of the Dataflow is that records are written one by one, which has the considerably lower speed of about 5.000 records per second.

Still, both methods of writing are off of the desired Performance Requirement for the Data Processing Engine, which is why the alternative method of writing detailed in Chapter 7 was devised.

8.2 Validation

This section serves to list the Requirements fulfilled by the system.

8.2.1 Functional Requirements

The fulfilled requirements and validation can be consulted in tables 8.1, 8.2 and 8.3. Each table contains the ID, Title and reasoning for the Validation of the Requirement. Unfulfilled Requirements are not listed.

All Must Have requirements were fulfilled, meaning that the System has the minimum set of capabilities required by the Project Owner and that the Project is a success. Some Should and Could have requirements were also fulfilled, either afforded through the choice in component technologies, architecture or care in the development process.

ID	FR01
Title	Event Aggregation
Validation	As shown in Chapter 7, the DrillDown view in the CEP Web Interface shows the User which simple events make up a complex event.
ID	FR02
Title	Correlated Event Navigation
Validation	The DrillDown view in the CEP Web Interface allows the user to view the details of each simple event related to the active complex event.
ID	FR03
Title	Complex Event Search
Validation	The Web Interface provides the User with the capability of querying the CEP data with the aid of logical operators, which can be applied to data fields.
ID	FR04
Title	Capability to add new rules
Validation	New jobs can be developed and submitted as required, and will be executed as long as there are enough Task Slots available.
ID	FR05
Title	Modular Architecture
Validation	This requirement is validated by the architecture detailed in Chapter 6. None of the system's other functionalities were altered, making the CEP Module an entirely optional addition to Portolan.
ID	FR06
Title	Programmable Dataflows
Validation	Apache Flink's data processing Dataflows are fully programmable and very extensible. Creating a new Dataflow is functionally identical to developing a Java or Scala program, with the only caveat being having to make use of Apache Flink's Dataflow API.
ID	FR07
Title	Data Manipulation
Validation	With Dataflows being fully programmable, data can be manipulated at will as long as it can be deserialized from its original state. As a proof of concept, the IP-Domain Correlation Dataflow deserializes the JSON data and fits it into new Java Objects, with all the functionalities of Object Oriented Programming.
ID	FR08
Title	Streaming Windows
Validation	Streaming Windows are natively supported by the Flink API, which includes tumbling, sliding, session and global windows.
ID	FR09
Title	Health Monitoring
Validation	Apache Flink provides both a Web Dashboard and a REST API that can be consulted to check the current status of the system, its job history, job details and Manager status. The Dashboard is immediately accessible and currently viewed as part of the CEP Module.

TABLE 8.1: Validation of Must Have Requirements

ID	FR11
Title	Reusable Dataflow Components
Validation	Dataflows are modular by design. The components developed for the IP-Correlation Dataflow were created with reusability in mind, and as such represent general functionalities. The only case where that was not as possibility were the MongoDB Sinks for upserting data. To mitigate that issue, base Synchronous and Asynchronous sinks were developed first, and the Sinks capable of upserting are OOP extensions of those classes.
textbfID	FR12
Title	Peformance Monitoring
Validation	Job statistics can be consulted through Apache Flink's REST API or through Metric Graphs in the Web Dashboard.

TABLE 8.2: Validation of Should Have Requirements

ID	FR15
Title	GUI Dataflow Deployment
Validation	As shown in Chapter 7, jobs can be submitted and started through the Web Dashboard.
textbfID	FR12
Title	Performance Monitoring
Validation	Job statistics can be consulted through Apache Flink's REST API or through Metric Graphs in the Web Dashboard.

TABLE 8.3: Validation of Could Have Requirements

8.2.2 Quality Attributes

Quality Attributes are validated in the tables 8.4, 8.5, 8.6 and 8.7.
All Quality Attributes were fulfilled.

ID	QR01
Description	The system must be capable of enduring a load of 20000 events per second
Validation	Validated in the Verification section of this Chapter. The system achieved an average throughput of 32356 events per second while outputting data to a Kafka Topic.
textbfID	QR02
Title	The system can have latency of no more than 5 seconds
Validation	The highest latency accumulated by a record going through the system was slightly over 3 milliseconds if the Dataflow's Kafka Source consumed events in batches and used one of the MongoDB Sinks, the slowest setup encountered during testing.

TABLE 8.4: Validation of Performance Attributes

ID	QR03
Description	The system will use stable, upgradable technologies.
Validation	Both Kafka and Flink have been updated with major releases. However, both also had bug fix updates to previous releases. The new versions retain backwards compatibility and any deprecations and removals are well documented in the release notes to prevent mistakes. Both technologies have a documented upgrade process and are capable of rolling upgrades as an alternative to bringing down an entire cluster at once.
textbfID	QR04
Title	The system will support checkpointing
Validation	Kafka stores the Consumer Offset so that a consumer can receive unread messages from a topic. Flink supports Checkpointing natively in the form of Distributed Snapshots.

TABLE 8.5: Validation of Reliability Attributes

ID	QR05
Description	In the GUI, input fields will be treated to avoid injection.
Validation	The Expression text field of the CEP Web Interface supports a limited set of logical operations, which are later translated to MongoDB operations. Furthermore, since in MongoDB a database operation is identified by the "$" prefix, the symbol is removed from every token previously to submitting the query, avoiding injection. The only control given to the user over the query are the filter expression and which Collection to query. The Database to query is set in the Django configuration file, out of the User's reach.
textbfID	QR06
Title	In the GUI, access to the data processed by the CEP engine will be provided only to users whose session is validated.
Validation	Validated through the use of the Django Session Management app. Both the Interface and API Endpoints respond only to connections with an active, valid session.

TABLE 8.6: Validation of Security Attributes

ID	QR07
Description	The system must be capable of splitting workload across available processing nodes.
Validation	Validated by Apache Flink. The framework uses a resource pool of Task Slots which can be made available by multiple Task Managers over a cluster of different machines and are used for jobs by the Job Manager.
textbfID	QR08
Title	The chosen technologies must be capable of horizontal scaling.
Validation	Kafka and Flink were purposefully built with clustering capabilities in mind, taking advantage of multiple machines for horizontal scaling. MongoDB is capable of Sharding and Replication, which also make use of a cluster.

TABLE 8.7: Validation of Scalability Attributes

Chapter 9

Conclusion

9.1 The Project

The Project proved to be an interesting challenge to the Intern. It required the use of every aspect of Software Engineering learned over the course of the Master's Degree, from planning to validation. It also took a great deal of research, as the subject matter was foreign to the Intern, who studied the necessary concepts and technologies as part of the Internship. This kind of solution and components are a recent addition to the field, and making the correct choices was paramount.

The development team at Dognædis were instrumental to understanding Portolan as a product, from both a conceptual and technical point of view. Their involvement in the initial phase was vital in detailing how Portolan operates, its intended use and the additions requested as part of this Project, which pointed the Intern's initial research efforts in the correct direction.

The problem was not trivial and required a careful, planned approach. There was the possibility of a bad decision in an early phase, such as choosing technologies or setting requirements, only being identified as such in a much later phase. This Project required a methodical approach and proved to be an exercise in Methodology. The iterative approach proved to be a good choice, as it afforded enough flexibility to change the task layout due to the delay without adversely affecting the process.

9.2 Future Work

The core CEP Module was developed to address the immediate concerns of the product and Operational Teams. However, the system was purposefully planned to be heavily extensible, and the modular nature allows even whole components to be swapped in and out. New ideas and features were considered, but not implemented due to being out of scope.

New Dataflows can be developed and deployed with minimal effort, and are expected to become the main additions to the system over time. Thanks to being source-agnostic, the CEP Module can also receive internal events and process them along with the ones gathered externally by Portolan, adding a more personalized layer of security to the client.

One feature requested by the Project Owner but deemed possibly out of scope was the graphical representation of events and their relations.

Another possible development is the addition of components capable of facilitating search and analytics, such as the previously mentioned ElasticSearch.

9.3 Conclusion

The CEP Module adds a new layer of stateful Data Processing to Portolan. By achieving the objectives and requirements set in the first phase of the Internship, the platform is now capable of more than collecting and basic enriching individual events: it can now analyze them in context, storing and updating information obtained from a global view of the data as it is collected. The flexiblity of programmable dataflows can allow a set of circumstances or stimuli (such as identifying a known suspicious event pattern or an invulgar sequence of network connections to addresses in a concurrently updated list of reported malware servers) to raise an alarm or even elicit an automated response from one or more systems (separating the suspect machine from the network, block the connections or redirect them to a sinkhole for further investigation by the Incident Response team).

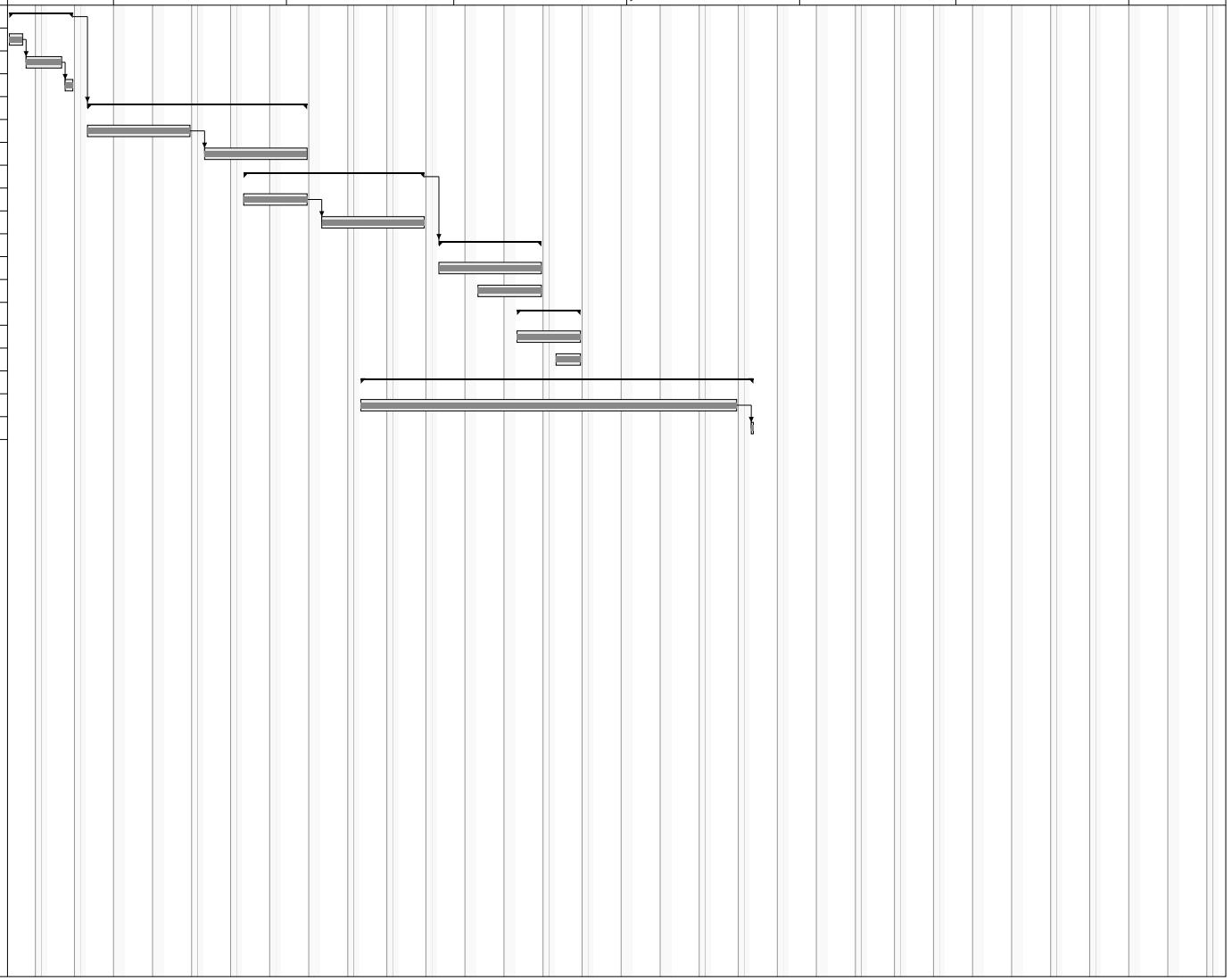
This new addition marks the evolution of Portolan to meet the increasing demand for systems capable of growing with the ever-expanding amount of vulnerabilities and attack vectors that currently characterize digital systems. In these last few months the number and variety of Cyber Attacks has increased and taken a greater hold of public awareness. As other systems fail, vulnerabilities and compromised machines and addresses are reported, Portolan's potential pool of information grows. Now, the platform has the means to analyze this information in a much larger scope and, with enough resources, in a much smaller timespan.

Appendix A

Gantt Charts

A.1 1st Semester

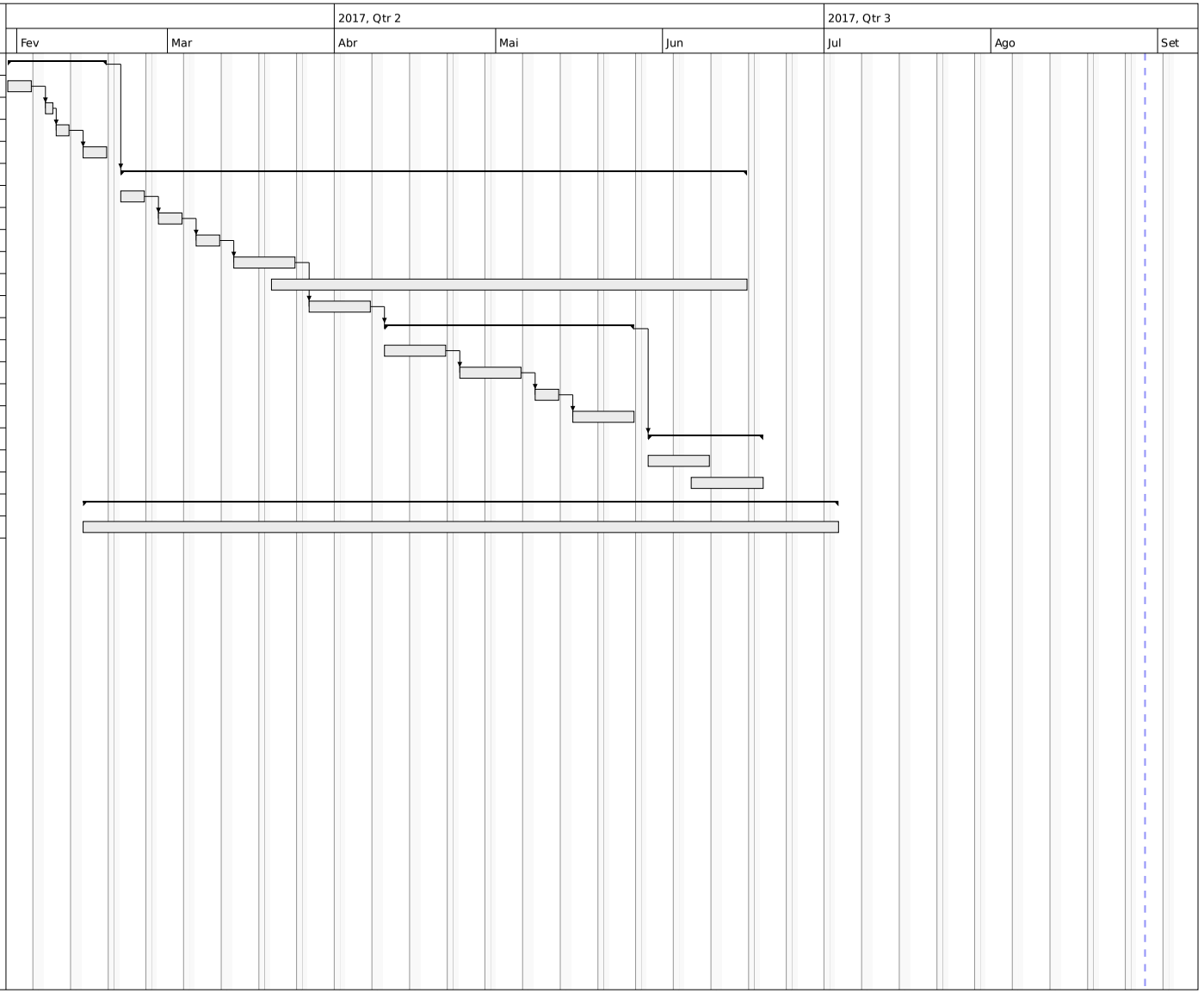
Name	Work
Introduction	**10d**
Introduction to Company	3d
Set up work environment	5d
Formalization of Internship Request	2d
Research	**30d**
Research of core concepts	15d
Research State of the Art	15d
Specification of Requirements	**25d**
Requirement elicitation	10d
Requirement formalization	15d
Architecture	**25d**
Initial design	15d
Initial choice of components	10d
Proof of Concept	**15d**
Prototyping	10d
Implementation of a simple dataflow	5d
Report	**51d**
Writing	50d
Delivery	1d



WBS	Name	Start	Finish	Work	Duration
1	**Introduction**	**Set 12**	**Set 23**	**10d**	**10d**
1.1	Introduction to Company	Set 12	Set 14	3d	3d
1.2	Set up work environment	Set 15	Set 21	5d	5d
1.3	Formalization of Internship Request	Set 22	Set 23	2d	2d
2	**Research**	**Set 26**	**Nov 4**	**30d**	**30d**
2.1	Research of core concepts	Set 26	Out 14	15d	15d
2.2	Research State of the Art	Out 17	Nov 4	15d	15d
3	**Specification of Requirements**	**Out 24**	**Nov 25**	**25d**	**25d**
3.1	Requirement elicitation	Out 24	Nov 4	10d	10d
3.2	Requirement formalization	Nov 7	Nov 25	15d	15d
4	**Architecture**	**Nov 28**	**Dez 16**	**25d**	**15d**
4.1	Initial design	Nov 28	Dez 16	15d	15d
4.2	Initial choice of components	Dez 5	Dez 16	10d	10d
5	**Proof of Concept**	**Dez 12**	**Dez 23**	**15d**	**10d**
5.1	Prototyping	Dez 12	Dez 23	10d	10d
5.2	Implementation of a simple dataflow	Dez 19	Dez 23	5d	5d
6	**Report**	**Nov 14**	**Jan 23**	**51d**	**51d**
6.1	Writing	Nov 14	Jan 20	50d	50d
6.2	Delivery	Jan 23	Jan 23	1d	1d

A.2 2nd Semester - Planned

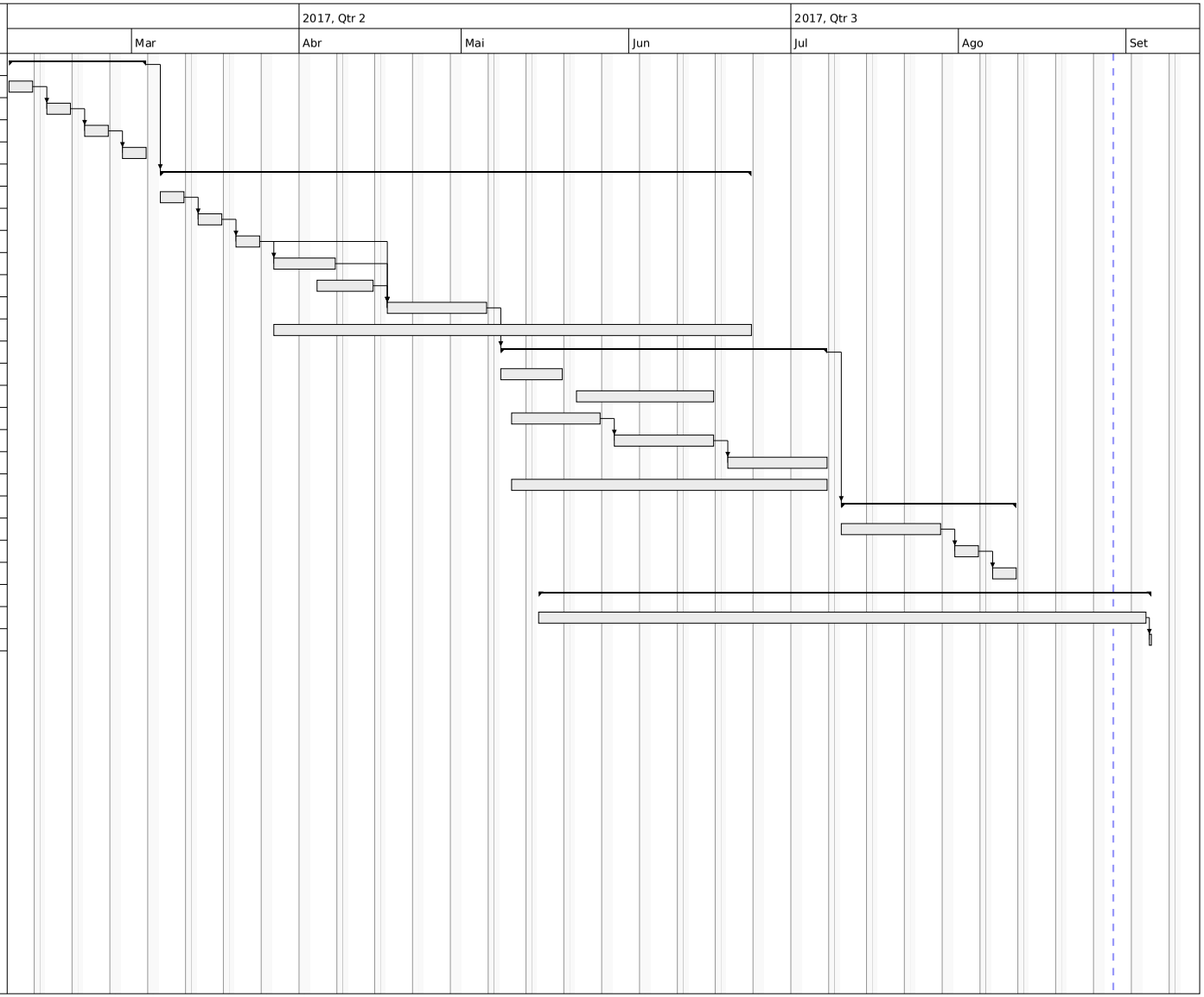
Name	Work
Preparation	**15d**
Set up development environment	5d
Set up testing environment	2d
Development of auxiliary tools	3d
Definition of Data Model	5d
Develop Back-End	**100d**
Deploy Apache Zookeeper	5d
Deploy Apache Kafka	5d
Deploy MongoDB	5d
Deploy Apache Flink	10d
Develop Dataflow	65d
Deploy additional datastores	10d
Develop Front-End	**35d**
Event queries	10d
Event visualizer	10d
Real time datafeed	5d
System monitoring	10d
Validation	**20d**
Testing	10d
Benchmarking	10d
Report	**101d**
Write report	101d



WBS	Name	Start	Finish	Work	Duration
1	**Preparation**	**Jan 30**	**Fev 17**	**15d**	**15d**
1.1	Set up development environment	Jan 30	Fev 3	5d	5d
1.2	Set up testing environment	Fev 6	Fev 7	2d	2d
1.3	Development of auxiliary tools	Fev 8	Fev 10	3d	3d
1.4	Definition of Data Model	Fev 13	Fev 17	5d	5d
2	**Develop Back-End**	**Fev 20**	**Jun 16**	**100d**	**85d**
2.1	Deploy Apache Zookeeper	Fev 20	Fev 24	5d	5d
2.2	Deploy Apache Kafka	Fev 27	Mar 3	5d	5d
2.3	Deploy MongoDB	Mar 6	Mar 10	5d	5d
2.4	Deploy Apache Flink	Mar 13	Mar 24	10d	10d
2.5	Develop Dataflow	Mar 20	Jun 16	65d	65d
2.6	Deploy additional datastores	Mar 27	Abr 7	10d	10d
3	**Develop Front-End**	**Abr 10**	**Mai 26**	**35d**	**35d**
3.1	Event queries	Abr 10	Abr 21	10d	10d
3.2	Event visualizer	Abr 24	Mai 5	10d	10d
3.3	Real time datafeed	Mai 8	Mai 12	5d	5d
3.4	System monitoring	Mai 15	Mai 26	10d	10d
4	**Validation**	**Mai 29**	**Jun 19**	**20d**	**16d**
4.1	Testing	Mai 29	Jun 9	10d	10d
4.2	Benchmarking	Jun 6	Jun 19	10d	10d
5	**Report**	**Fev 13**	**Jul 3**	**101d**	**101d**
5.1	Write report	Fev 13	Jul 3	101d	101d

A.3 2nd Semester - Final

Name	Work
Preparation	**20d**
Set up workstation	5d
Development of auxiliary tools	5d
Study of Portolan Data Model	5d
Initial definition of CEP Data Model	5d
Develop Back-End	**114d**
Deploy Apache Zookeeper	5d
Deploy Apache Kafka	5d
Deploy MongoDB	5d
Definition of Data Model	10d
Set up development environment	9d
Deploy Apache Flink	15d
Develop Dataflow	65d
Develop Front-End	**116d**
Study of Django Framework	10d
Refactoring of CEP Data Model	20d
Interface for Event Search	13d
Interface for Search Results	15d
Interface for Event Details	15d
Endpoints for querying Events	43d
Validation	**25d**
Testing	15d
Benchmarking	5d
Final validation of Requirements	5d
Report	**82d**
Write report	81d
Delivery	1d



WBS	Name	Start	Finish	Work	Duration
1	**Preparation**	**Fev 6**	**Mar 3**	**20d**	**20d**
1.1	Set up workstation	Fev 6	Fev 10	5d	5d
1.2	Development of auxiliary tools	Fev 13	Fev 17	5d	5d
1.3	Study of Portolan Data Model	Fev 20	Fev 24	5d	5d
1.4	Initial definition of CEP Data Model	Fev 27	Mar 3	5d	5d
2	**Develop Back-End**	**Mar 6**	**Jun 23**	**114d**	**80d**
2.1	Deploy Apache Zookeeper	Mar 6	Mar 10	5d	5d
2.2	Deploy Apache Kafka	Mar 13	Mar 17	5d	5d
2.3	Deploy MongoDB	Mar 20	Mar 24	5d	5d
2.4	Definition of Data Model	Mar 27	Abr 7	10d	10d
2.5	Set up development environment	Abr 4	Abr 14	9d	9d
2.6	Deploy Apache Flink	Abr 17	Mai 5	15d	15d
2.7	Develop Dataflow	Mar 27	Jun 23	65d	65d
3	**Develop Front-End**	**Mai 8**	**Jul 7**	**116d**	**45d**
3.1	Study of Django Framework	Mai 8	Mai 19	10d	10d
3.2	Refactoring of CEP Data Model	Mai 22	Jun 16	20d	20d
3.3	Interface for Event Search	Mai 10	Mai 26	13d	13d
3.4	Interface for Search Results	Mai 29	Jun 16	15d	15d
3.5	Interface for Event Details	Jun 19	Jul 7	15d	15d
3.6	Endpoints for querying Events	Mai 10	Jul 7	43d	43d
4	**Validation**	**Jul 10**	**Ago 11**	**25d**	**25d**
4.1	Testing	Jul 10	Jul 28	15d	15d
4.2	Benchmarking	Jul 31	Ago 4	5d	5d
4.3	Final validation of Requirements	Ago 7	Ago 11	5d	5d
5	**Report**	**Mai 15**	**Set 5**	**82d**	**82d**
5.1	Write report	Mai 15	Set 4	81d	81d
5.2	Delivery	Set 5	Set 5	1d	1d

Appendix B

Verification Details

B.1 Apache Kafka

B.1.1 Performance Benchmark

This Performance Benchmark serves the purposes of ensuring that this component performs well enough to make complying with the Requirements set by the Performance Quality Attribute possible (QR01 and QR02) and knowing the practical throughput limits achievable. The Kafka Consumer provided by Apache Flink can be configured the same way as the official Kafka Consumer written in Python, so the configurations tested can be used in an Apache Flink Dataflow.

These tests were conducted to benchmark Apache Kafka's throughput with the chosen configurations and the effect of the number of polled records on throughput and latency.

Table B.1 shows the results obtained. Latency is measured in milliseconds and total time in seconds.

Polled Records	Throughput	Latency (ms)	Total Time (s)
1	36889.73	0.009	8.452
25	44490.867	0.132	7.008
100	45239.698	0.499	6.892
200	46123.077	0.895	6.76
450	46508.353	1.814	6.704

TABLE B.1: Apache Kafka Performance Benchmark

Apache Kafka shows better performance when polling multiple records at a time. An increase in records polled caused higher latency. However, both latency and throughput are adequate for the intended use and Quality Attributes defined as part of this Project.

B.1.2 Broker Failure

This test involved simulating the shutdown and failure of the Kafka Broker, both in standalone and cluster setup. The test involved one Apache Kafka Broker in the standalone setup and two when clustered.

The standalone failure test was conducted to verify that data and Queue offsets persisted even if the Broker was shut down orderly, and then forcibly. In order to verify the Queue offsets an auxiliary test program was written in Java to consume messages preloaded into a topic. As expected, the data remained available when the Broker was brought back up, and the consumer kept reading from where it had left off, so the current offset was correct.

The cluster test aimed to show that a topic with replication factor greater than 1 remains accessible even in case the current Broker leading that topic fails, and that consumer offset is not lost. The setup for this test involved the use of two Kafka Brokers, distinguished by their IDs, respectively 0 and 1. A topic named KafkaBench01 was created with replication factor 2 and a single partition to make sure that there would be only two replicas for the partition, each in a different Broker.

This test was split into two phases: one with the Topic preloaded with Messages and the Consumer Java Program, and another with one instance of the Java Program producing new messages and another instance Consuming them.

The Java Program used Broker 1 as the initial connecting point to the Kafka cluster, which also served to test if a consumer is capable of switching to a new Partition Leader while consuming messages despite, at launch, not having any information related to the address of an alternate Broker. It was then tested with the address of Broker 0, and finally with a comma-separated list containing the addresses of both Brokers.

A CLI tool was used to discover that Broker 1 was the Partition Leader, which was then forcibly shut down while a Java program read from the Topic.

Upon shutting down Broker 1, the Consumer Program suffered a short interruption, after which it continued receiving responses with no further issue. The Partition Leader changed to Broker 0. This test was executed for a clean and forced shutdown (using kill -9).

Attempts to create a new connection to the cluster proved unsuccessful, as the configured bootstrap server, Broker 1, was down. Switching it to Broker 0's address solved that issue, and the Consumer was capable of reading messages once again.

The test was redone with the only difference being Broker 0 serving as the entry point. The results were identical, with the exception of, upon trying to reconnect with Broker 1 down, the connection was successful. Finally, both Broker 0 and 1's addresses were listed as bootstrap-servers. The Consumer was able to connect to the cluster from a total restart without any intervention after Broker 1 had been brought down. To fully confirm the hypothesis, Broker 1 was brought up and, after synchronizing, Broker 0 was shut down. The Consumer was, again, able to connect and read messages.

To finalize, a new instance of the Java Program was used to Produce messages. This time, the Producer instance generated messages and sent them to the previously used Kafka Topic, while the Consumer instance polled them. The test battery was repeated with a message Producer and Consumer. The results were identical, with the Producer also capable of switching Brokers dynamically. It can also be started with a list of multiple bootstrap servers, like the Consumer.

These tests prove the reliability of Apache Kafka. In a single-Broker deployment, data that has been received and flushed to disk is persisted and is not lost upon Broker failure. It should be noted that the flush interval can be customized. A flush can be triggered at periodic intervals and by receiving a configurable number of messages before the time interval has passed.

The cluster setup shows that Kafka can handle a failure of a broker, as long as an alternate Broker with a replica of the target topic is available. Consumer and Producer instances become aware of the cluster layout after initializing a connection and are capable of switching Brokers according to the cluster's health, mitigating the effects of a Broker failure.

B.1.3 Zookeeper Failure

As the backbone of Kafka, a Zookeeper failure can be catastrophic. Without an available Zookeeper instance, Kafka is not capable of updating its own data, such as cluster

layout, configuration changes, locating Topic partitions and Leader Election, all of which are vital.

The tests for Zookeeper failure involved a standalone and clustered setup for Zookeeper, a single and two Broker cluster setup for Kafka, and the Consumer/Producer Java Program. They were meant to assess the impact of Kafka losing connectivity with a functioning Zookeeper instance. The Zookeeper instances all ran in the same machine, but for the purpose of these tests causing an instance to fail is a simulation of a machine failing. Server failures are caused by shutting down the processes forcibly.

Trying to launch a Kafka Broker without a connection to a functioning Zookeeper node causes the Broker to report error messages and shut down after a configurable number of tries.

All tests that follow use Kafka instances launched with an active Zookeeper connection. All disruption is caused after the Broker instance has finished the startup process.

On a single node deployment for both Zookeeper and Kafka, shutting down the Zookeeper node caused Kafka to keep reporting errors while trying to connect to Zookeeper. However, the Broker did not shut down, still managed to receive and send messages and keep track of Consumer offsets. From this point, there were two experiments: run the Zookeeper instance to see if Kafka would reconnect and bring everything up to date, and forcibly shut down Kafka to verify if there was data loss without a Zookeeper instance.

After restarting Zookeeper, the Kafka instance reconnected without issues and persisted the new offsets and topic metadata. After restarting both components, the state remained the same.

Trying to do a clean shutdown of Kafka before a Zookeeper instance is available causes the message broker to repeatedly try to connect to a Kafka instance, looping endlessly until then. If a Zookeeper instance recovers, Kafka completes the shutdown process. Otherwise, the process has to be forcibly closed. Upon being launched again, with connection to a functioning Zookeeper instance, it recovers the state it had before being killed and does not lose data.

On a clustered deployment of 2 Kafka Brokers, shutting down the Zookeeper node causes both Brokers to report errors while trying to reconnect to it. However, sent messages and consumer offsets are still synchronized between them, so the Kafka cluster retains some functionality. If the Zookeeper instance recovers, the Brokers reconnect and the situation is corrected.

If a Broker fails while Zookeeper is unavailable, it cannot rejoin the cluster, as it simply does not launch without a ZK instance. If the Broker was a Partition Leader, the cluster can no longer serve or receive data for that partition, as Leader Election is achieved through Zookeeper.

The system exhibits a dangerous behavior if the entire system is restarted and the Brokers are not synchronized (if one Broker failed while the other kept receiving or sending messages, for example). The first Broker to connect to Zookeeper becomes the elected Leader for all of its Partitions. As such, its state is imposed as the truth. If the Broker that kept operating then connects to Zookeeper, it matches its state to the Leader's, losing all extra operations that were not synchronized. This situation is extremely unlikely. Kafka and Zookeeper do not fail often, and it would require an extremely remote set of circumstances for this to happen. Plus, a setup with two Brokers and a single Zookeeper instance does not ensure availability, as all that is required for the system to break down is the single Zookeeper instance or machine failing. Still, unlikely and result of poorly planning as it is, this situation must be pointed out to avoid serious mistakes.

Finally, in a clustered 3 machine setup Zookeeper was capable of handling the failure of a single node. The Brokers suffered no issues. The failure of another node would cause

the remaining Zookeeper instance to stop functioning, leading the Brokers to raise errors as detailed on the standalone node failure test.

B.2 Apache Flink

B.2.1 Peformance Benchmark

The Performance Benchmark serves to prove that the Processing Pipeline complies with the Quality Attributes QR01 and QR02, defined in Chapter 4.

The developed Dataflow read events from a Kafka topic with a single partition, preloaded with the monthly event Dataset made available by Dognædis. The processing pipeline was parallelized, for a total of 4 concurrent pipelines. The Data Source was configured with parallelism 1, so that all 4 pipelines would draw events from the same Kafka Source. A different setup was tested for each Data Sink available: the custom-developed Data Sinks for Mongo (synchronous and asynchronous) and the provided Kafka Sink.

- Mongo Synchronous Sink:
  - 4344 records outgoing sink per second
  - 4344 records outgoing processing pipeline into sink per second
  - 5196 records going into processing pipeline per second
  - 6472 records outgoing source per second
  - 5196 records outgoing source per second after backpressure kicks in
  - Data Processing and Source were backpressured, meaning Sink couldn't write into Mongo fast enough
  - Latency: 0.23 ms
- Mongo Asynchronous Sink:
  - 6632 records outgoing sink per second
  - 6632 records outgoing processing pipeline into sink per second
  - 8036 records going into processing pipeline per second
  - 8036 records outgoing source per second
  - Data Processing and Source were backpressured, meaning Sink couldn't write into Mongo fast enough
  - Latency: 0.15 ms
- Kafka Sink, same instance as read:
  - 26920 records outgoing sink per second
  - 26920 records outgoing processing pipeline per second
  - 32356 records going into processing pipeline per second
  - 32356 records outgoing source per second
  - Source was backpressured, meaning source was capable of ingesting data faster than it could be processed
  - Latency: 0.05 ms

It should be noted that there is a constant ratio of 1.2 between events entering and leaving the Processing Engine in all tests, meaning that the only difference is the overall throughput. That is due to events being filtered out if they have no data fields used by the Dataflow.

B.2.2 Failure Tolerance

The tests for failure tolerance were performed with an active Dataflow and were meant to test the resilience of Task Managers. To simulate a failure, the Task Manager was forcibly shut down.

The first setup made use of a single Task Manager. After shutting down the process, the Dataflow stopped to execute, and the Job Manager stopped reporting any available Task Managers. Restarting the Task Manager caused the Job Manager to reassign the interrupted Job, and it resumed from the latest checkpoint available.

The second setup used two Task Managers. The second Task Manager was added to the available pool after the Dataflow started processing to distinguish both Managers and make sure that shutting one down would stop the Job. Causing the working Task Manager to fail caused the Job Manager to try and restart the Job. After the Task Manager timed out, the Job Manager reassigned the job to the remaining Task Slots in the other Task Manager, resuming the Job from the latest checkpoint.

B.3 Mongo DB

MongoDB's write speed was summarilly benchmarked to understand its poor performance. The test was conducted with a Java Program that submitted the event Dataset in configurable bulks to a MongoDB Collection.

Bulk Size	Throughput
20000	14501.547
10000	15120.120
5000	14948.317
1	4354.351

TABLE B.2: MongoDB writing throughput

As can be seen in table B.2, MongoDB performs considerably better when writing in bulk than when writing event-by-event. This issue caused the poor processing speed of Apache Flink when using MongoDB directly as a Data Sink.

Bibliography

[1] David Luckham and W. Roy Schulte. *EPTS Event Processing Glossary*. Event Processing Technical Society. URL: http://www.complexevents.com/wp-content/uploads/2011/08/EPTS_Event_Processing_Glossary_v2.pdf.

[2] *Definition of Cyber Security*. ITU. URL: http://www.itu.int/en/ITU-T/studygroups/com17/Pages/cybersecurity.aspx.

[3] *NCSC Glossary*. United Kingdom's National Cyber Security Centre. URL: https://www.ncsc.gov.uk/glossary.

[4] *NICCS Glossary*. National Institute for Cybersecurity Careers and Studies. URL: https://niccs.us-cert.gov/glossary.

[5] David Chismon and Martyn Ruks. "Threat Intelligence: Collecting, Analysing, Evaluating". In: (). URL: https://www.ncsc.gov.uk/guidance/threat-intelligence-collecting-analysing-evaluating.

[6] *Definition of Cyber Intelligence*. Wikipedia. URL: https://en.wikipedia.org/wiki/Cyber_threat_intelligence.

[7] *Esper Event Stream Processing and Correlation*. O'Reiily onJava.com. URL: archive.oreilly.com/pub/a/onjava/2007/03/07/espe\-event-stream-processing-and-correlation.html.

[8] *CEP = event correlation + decision management*. James Taylor on Everythign Decision Management. URL: http://jtonedm.com/2009/09/02/cep-event-correlation-decision-management/.

[9] Dognædis Main Website. URL: https://www.dognaedis.com/.

[10] *What is Scrum?* Scrum.org. URL: https://www.scrum.org/Resources/What-is-Scrum.

[11] Ken Schwaber and Jeff Sutherland. *The Scrum Guide*. URL: http://www.scrumguides.org/docs/scrumguide/v2016/2016-Scrum-Guide-US.pdf#zoom=100.

[12] *What is Kanban?* KanbanBlog.org. URL: http://kanbanblog.com/explained/.

[13] John Lark and Valentin Nikonov. *ISO 31000 - Risk Management*. International Organization for Standardization.

[14] *Alienvault USM*. URL: https://www.alienvault.com/products.

[15] *Alienvault Open Threat Exchange*. URL: https://www.alienvault.com/open-threat-exchange.

[16] Thomas Maclsaac. *What is the difference between Business Intelligence and Operational Intelligence?* URL: https://www.linkedin.com/pulse/20140723150020-78290-what-is-the-difference-between-business-intelligence-and-operational-intelligence.

[17] Christy Wilson. *Business Intelligence Versus Operational Intelligence: What's the difference?* URL: http://blog.syncsort.com/2015/10/big-data/business-intelligence-versus-operational-intelligence-whats-the-difference/.

[18] *CEP and the N+Tier Architecture*. TIBCO. URL: http://www.tibco.com/blog/2009/05/21/cep-and-the-ntier-architecture/.

[19] *Lambda Architecture.net*. URL: http://lambda-architecture.net/.

[20] *Lambda Architecture at MapR*. URL: https://mapr.com/developercentral/lambda-architecture/.

[21] Jay Kreps. *Questioning the Lambda Architecture*. URL: https://www.oreilly.com/ideas/questioning-the-lambda-architecture.

[22] *Data Processing Architectures - Lambda and Kappa*. URL: https://www.ericsson.com/research-blog/data-processing-architectures-lambda-and-kappa/.

[23] *IBM Operational Decision Manager*. URL: http://www-03.ibm.com/software/products/en/odm.

[24] *RedHat Drools*. URL: http://drools.org/.

[25] *Feedzai*. URL: https://feedzai.com/.

[26] *Espertech Esper*. URL: http://www.espertech.com/esper/.

[27] *Apache Storm*. URL: https://storm.apache.org/.

[28] *Apache Flink*. URL: https://flink.apache.org/.

[29] *MoSCoW: Requirements Prioritization Technique*. Business Analyst Learnings. URL: https://businessanalystlearnings.com/ba-techniques/2013/3/5/moscow-technique-requirements-prioritization.

[30] *Introduction to Apache Kafka*. Apache Kafka. URL: https://kafka.apache.org/intro.

[31] *Apache Zookeeper*. URL: https://zookeeper.apache.org/.

[32] *Benchmarking Apache Kafka*. LinkedIn. URL: https://engineering.linkedin.com/kafka/benchmarking-apache-kafka-2-million-writes-second-three-cheap-machines.

[33] *Redis Main Website*. URL: https://redis.io/.

[34] *Apache Spark*. URL: https://spark.apache.org/.

[35] *ElasticSearch*. URL: https://www.elastic.co/products/elasticsearch.

[36] *Elastic Logstash*. URL: https://www.elastic.co/products/logstash.

[37] *Elastic Kibana*. URL: https://www.elastic.co/products/kibana.

[38] *MongoDB*. URL: https://www.mongodb.com/.

[39] *Django*. URL: https://www.djangoproject.com/.

[40] *Kafka vs Redis*. URL: https://logz.io/blog/kafka-vs-redis/.

[41] *Extending the Yahoo Streaming Benchmark*. Data Artisans. URL: https://data-artisans.com/blog/extending-the-yahoo-streaming-benchmark.

[42] *Hadoop HDFS*. URL: https://hortonworks.com/apache/hdfs/.

[43] *Python client for Apache Kafka*. URL: https://github.com/dpkp/kafka-python.

[44] *Docker*. URL: https://www.docker.com/.