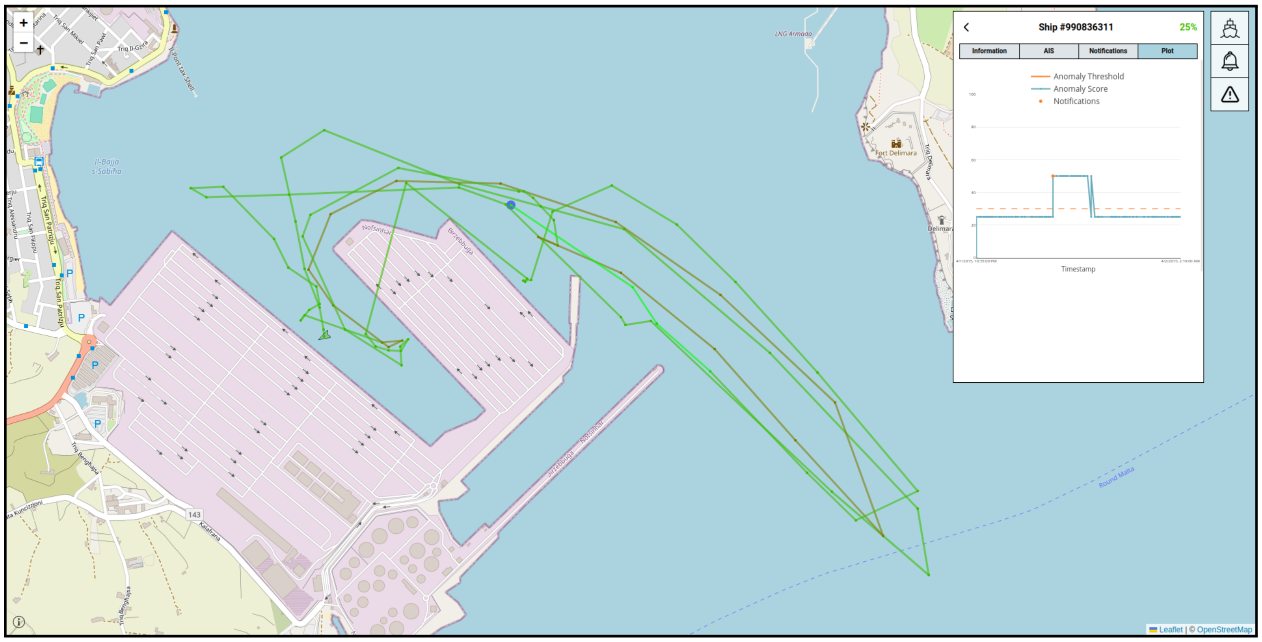
Developer Manual

Track Anomaly Detection



Developer Manual submitted as the final delivarable for course:

Software Project (CSE2000)

Group 18A

Augustinas Jučas, *5742439*

Victor Purice, *5777216*

Justinas Jučas, *5739500*

Marius Frija, *5804132*

Aldas Lenkšas, *5714192*

Client: **Dutch Ministry of Defence**

Client Representative: **Sebastiaan Alvarez Rodriguez**

Teaching Assistant: **Nathalie van de Werken**, Coach: **Kubilay Atasu**

Date of submission: **21 June 2024**

Contents

Chapter 1

Overview

This manual aims to provide all the necessary information for software developers to set up, edit, and maintain the code for the Track Anomaly Detection application.

As a note, all paths to the files or folders mentioned in this manual are relative to the root folder of the application's entire code base.

1.1 Terminology

- **Track** - the ship's path from the departure port to the destination.
- **AIS** - Automatic Identification System
- **AIS signal** - the signal that ships must frequently send. Such a signal contains information about the vessel: its position, rotation, destination, etc.
- **Anomaly score** - the score between 0% and 100% given by the application to a certain track. It tries to represent how suspiciously the ship is behaving in its track. However, note that it should not be taken as some "ground truth", and further evaluation should be done.

1.2 The structure of the code base

The repository consists of the following folders and files:

- `.gitlab` - Template files for creating Issues and Merge Requests in GitLab.
- `.idea` - IntelliJ IDEA project files.
- `backend` - Code for the backend part of the application.
- `config` - CheckStyle and PMD rulesets.
- `frontend` - Code for the frontend part of the application.
- `simulator` - Code for the simulator.
- `stress-tester` - Code for the stress tester.
- `.gitignore` - Files to be ignored in git repository.
- `.gitlab-ci.yml` - Gitlab CI pipeline setup.
- `README.md` - Main README file. Instructions on how to set up the project can be found there.

- `README.md` - Main README file. Instructions on how to set up the project can be found there.

1.3 Setting up and running the application

Instructions are in the README files of each part of the application (backend, frontend, simulator, and even stress tester). The main README file is at the top level of the repository: `README.md`.

Chapter 2

Technology Stack and Architecture

2.1 Technology Stack

- **Apache Kafka** for distributed event handling with its message queue. The events coming to the queue are ships' AIS signals.
- **Apache Flink** for distributed stream processing. It is used to calculate the anomaly scores for the tracks.
- **Apache Druid** as the database for storing a large amount of data coming in streams.
- **Java Spring Boot** for the web server.
- **React.js** for the frontend.

2.2 System Design

This section describes the architecture of the system. The more detailed explanation of each part of the system can be found in the following sections of this Developer Manual.

The mentioned technologies were connected to a coherent functional system. The design of it is presented in Figure 2.1. It can be split into the following parts:

- Simulator (Java)
- Event Messaging Queue (Kafka)
- Stream Processing and Anomaly Score Calculation (Flink)
- Web Server (Spring Boot)
- Database (Druid)
- Frontend (React.js)

The Simulator is the first part of the system. It is an application that simulates events that mimic historic AIS signals. Those signals are sent by ships in periodic intervals to the Kafka Messaging Queue. As the deployed system will use real-time world AIS data, the Simulator will not be run in the deployed system. Thus, it is a separate process not part of the web server.

Then, the Kafka Messaging Queue takes incoming produced data and stores it in topics (queues). Even though the only producer is the Simulator, Kafka Queue could handle more if needed. For instance, multiple military radars sending information from the ships could be added as separate producers to the Kafka Queue. Since Kafka needs to be started as a separate process, it is detached from the web server.

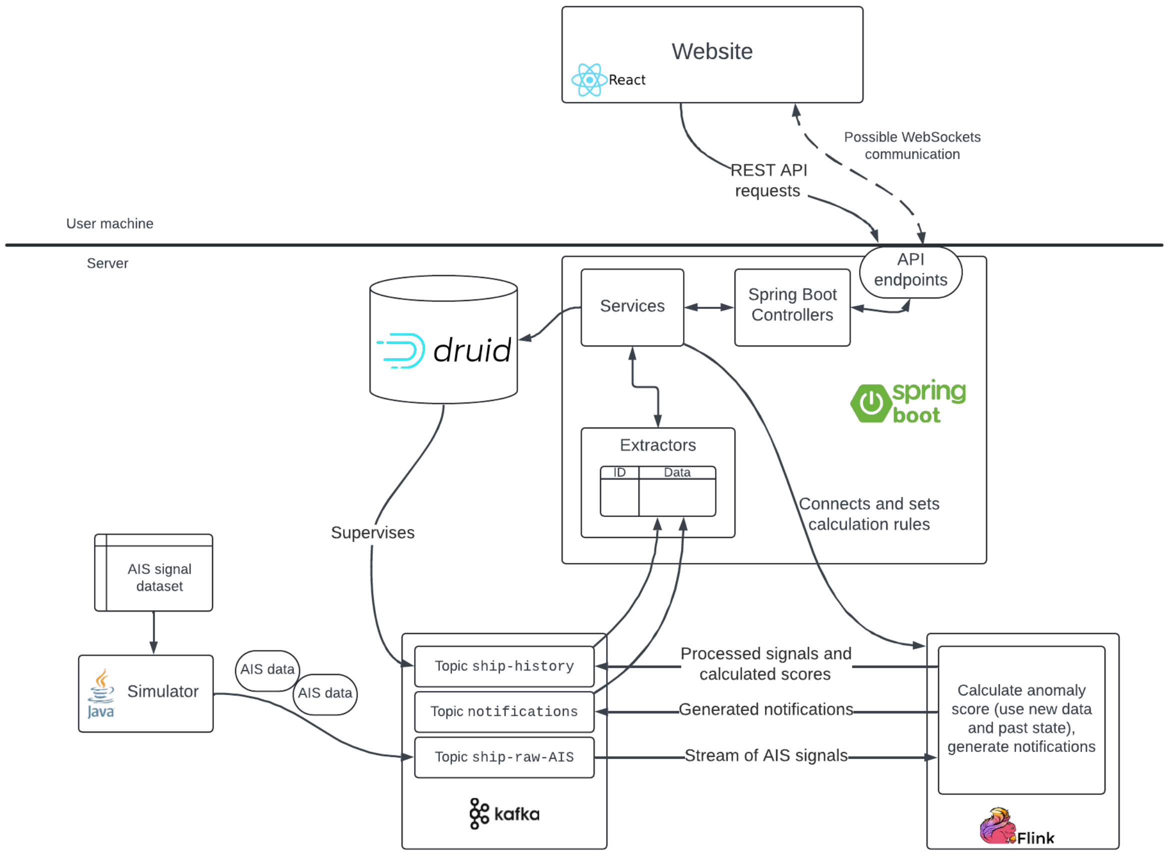


Figure 2.1: *System Design Layout for the whole Track Anomaly Detection application. Boxes represent separate steps and components in the system, while the arrows represent the data flow.*

The next step for the data flow is the Flink stream processing. This part of the code is a consumer of the Kafka Queue, meaning it takes a stream of events from it. The taken stream is split according to the vessel ID, and then each ship's anomaly score is calculated. A past state is used to calculate the anomaly score for a new data entry, which is provided by the Flink stateful map functions. In addition, during the processing of AIS signals, notification data is calculated, and the current ship details objects are aggregated. The new details are written to the Kafka topic, and also saved to the Druid database.

The communication between the client and the server is done through a REST API and WebSockets requests. To handle these, a web server was implemented using the Spring Boot framework and structured according to the layered architecture: controllers, services, and core domain code. The controllers handle the API endpoints. When a request is received, the controller calls service classes, which call the internal classes to handle the logic of the request.

The Apache Druid database is used for storing ships signals and score histories. The database is set to supervise the Kafka topic to which the calculated anomaly scores along with the corresponding AIS signals are sent. This way, web servers are able to query the database when the ship historical data is needed. For example, such a functionality is useful when services need to retrieve the ship's anomaly scores throughout the track, so that the colourful trajectory can be displayed in the frontend.

The frontend is the final part of the design and is the only visible part to the system user. Its implementation relies on the React.js framework. As a simplification, in Figure 2.1, the backend part of the frontend is skipped. However, when deploying, the React server should be started on the backend, so that the client computer can retrieve the resources, such as HTML pages, CSS, and JavaScript code.

Chapter 3

Backend

This section describes the backend in detail. It consists of the following parts: event handling with Kafka (3.1), stream processing with Flink (3.2), and a Spring Boot web server (3.3).

3.1 Kafka Topics

There are three Kafka topics (queues) used:

- `ships-raw-AIS`
- `notifications`
- `ships-history`

The events pushed to these topics are Strings, following the JSON format. These JSONs represent the required objects from the folders `backend/src/main/java/sp/model` and `backend/src/main/java/sp/dtos/ExternalAISSignal.java`. The examples of the Strings are below, and the general serialization logic is implemented in `backend/src/main/java/sp/pipeline/utils/json/SoftJsonSerializationSchema.java`.

3.1.1 Topic `ship-raw-AIS`

Ship AIS signals should be sent to this topic. Anomaly score calculation (done using Flink stream processing) consumes the ship data from this topic.

The events that come to this topic should be Strings representing JSON serialized version of the object `ExternalAISSignal` (described in `backend/src/main/java/sp/dtos/ExternalAISSignal.java`). Here is an example of such a String:

```json
{
    "producerID": "simulator",
    "shipHash": "0x6d6794f3186f584637721a1e1789fd2e71c28195",
    "speed": 0.2,
    "longitude": -5.354052,
    "latitude": 35.92466,
    "course": 314.0,
    "heading": 52.0,
    "timestamp": "2015-04-01T20:29:00Z",
    "departurePort": "CEUTA"
}
```

3.1.2 Topic `notifications`

Notifications are meant for the user (the Navy operator) to get attention to the ships that became anomalous. After calculating the ship's anomaly score, some notifications are created in the pipeline. After being created, notifications are saved to this Kafka topic, from which the extractor frequently polls the notifications so that the frontend can get them (through the API endpoint via Spring Boot controller and service).

Refer to the representative sections to get more information about the pipeline flow (Section 3.2), extractors (Section 3.2.5), and web server structure (Section 3.3).

The events that come to this topic should be Strings representing JSON serialized version of the object `Notification` (described in `backend/src/main/java/sp/model/Notification.java`). Here is an example of such a String:

```json
{
    "id": 210698893,
    "shipID": 990836311,
    "currentShipDetails": {
        "currentAnomalyInformation": {
            "score": 50.0,
            "explanation": "Heading difference between two consecutive signals is too large: 63 degrees is more than threshold of 40 degrees.\nManeuvering is too frequent: 11 strong turns (turns of more than 40 degrees) during the last 60 minutes is more than threshold of 10 turns.\n",
            "correspondingTimestamp": "2015-04-01T21:59:00Z",
            "id": 990836311
        },
        "currentAISSignal": {
            "id": 990836311,
            "speed": 0.5,
            "longitude": 14.54607,
            "latitude": 35.82201,
            "course": 153.0,
            "heading": 192.0,
            "timestamp": "2015-04-01T21:59:00Z",
            "departurePort": "VALLETTA",
            "receivedTime": "2024-06-14T18:45:29.028162548+02:00"
        },
        "maxAnomalyScoreInfo": {
            "maxAnomalyScore": 50.0,
            "correspondingTimestamp": "2015-04-01T21:59:00Z"
        }
    },
    "read": false
}
```

3.1.3 Topic `ship-history`

This topic saves the ship details with their calculated anomaly scores. Similarly, as for `notifications`, the extractor frequently polls this topic so that the frontend can get the current ship details.

Note that usually firstly only the ship details without the anomaly score will be saved to this topic, and only after the score is calculated updated details are pushed to this topic.

The database (configured using Apache Druid) is linked to this topic and will permanently save the events from here. This also allows for implementing features, such as showing the history of anomaly scores for the ships.

Refer to the representative sections to get more information about the pipeline flow (Section 3.2), extractors (Section 3.2.5), and web server structure (Section 3.3).

The events that come to this topic should be Strings representing JSON serialized version of the object

`Notification` (described in `backend/src/main/java/sp/model/Notification.java`). Here is an example of such a String:

```
{
    "currentAnomalyInformation": {
        "score": 0.0,
        "explanation": "some explanation",
        "correspondingTimestamp": "2015-04-01T20:27:00Z",
        "id": 990836311
    },
    "currentAISSignal": {
        "id": 990836311,
        "speed": 5.9,
        "longitude": 14.54465,
        "latitude": 35.82241,
        "course": 109.0,
        "heading": 113.0,
        "timestamp": "2015-04-01T20:27:00Z",
        "departurePort": "VALLETTA",
        "receivedTime": "2024-06-14T18:41:43.028537624+02:00"
    },
    "maxAnomalyScoreInfo": {
        "maxAnomalyScore": 0.0,
        "correspondingTimestamp": "2015-04-01T20:27:00Z"
    }
}
```

3.2 AIS Signal Processing With Flink

The AIS signals that come to the Kafka topic `ship-raw-AIS` (all Kafka topics are described in Section 3.1) are processed, the anomaly scores are calculated, and, whenever needed, the notifications are created. The updated ship details are passed back to the Kafka topic `ships-history`, and the notifications are passed back to the Kafka topic `notifications`. All these steps are done using an "Anomaly detection pipeline" in the backend. Its implementation can be found inside `backend/src/main/java/sp/pipeline/AnomalyDetectionPipeline.java`.

The steps for the pipeline's flow are described below. These steps are constructed when the method `buildPipeline()` (in the class `backend/src/main/java/sp/pipeline/AnomalyDetectionPipeline.java`) is called.

For the API endpoints to work, there are "extractors" implemented that frequently poll the Kafka topics `ship-raw-AIS` and `ships-history`. More information about them can be found in Section 3.2.5.

3.2.1 Pipeline Part: ID Assignment

The first part of the pipeline is calculating and assigning internal IDs to the ships. This part of the pipeline is built in `backend/src/main/java/sp/pipeline/parts/identification/IdAssignmentBuilder.java`. To be more precise, the ID Assignment step first takes the stream of data from the Kafka topic `ship-raw-AIS`. This stream contains AIS signal data. An internal ID is calculated for each of the stream signals. The ID calculation details can be found inside `buildIdAssignmentPart` method in the mentioned class.

3.2.2 Pipeline Part: Anomaly Score Calculation

The stream of the AIS signals with IDs calculated goes through the anomaly score calculation part. For this, an interface is created: `backend/src/main/java/sp/pipeline/parts/scoring/scorecalculators/ScoreCalculationStrategy.java`. The interface allows for easy changes in the score calculation strategy so that developers can create or edit new ones easily.

The score calculator strategy that is currently used is implemented in `backend/src/main/java/sp/pipeline/parts/scoring/scorecalculators/SimpleScoreCalculator.java`. Its idea is to split the stream of all ships into a keyed stream (keyed by the ship's ID) and then for each ship to calculate an anomaly score by combining four heuristic checks.

The four heuristic checks used are described in the folder `backend/src/main/java/sp/pipeline/parts/scoring/scorecalculators/components/heuristic`. They implement the abstract class `HeuristicStatefulMapFunction` (in the mentioned folder). This class is a child class of Flink's `RichMapFunction`. This allows to use Flink's states (such as, `ValueState` and `ListState`[1].). The setting up of the heuristic calculation is done in the method `open`, and the anomaly score calculation logic is implemented inside the `map` function (`@Override public AnomalyInformation map(AISSignal value)`).

To see the example of how one specific heuristic score calculation works, you can check the `SignalStatefulMapFunction` class. It extends the abstract class `HeuristicStatefulMapFunction`, and implements a method for checking whether a new AIS signal is considered an anomaly, compared to the past one. It considers the signal an anomaly if the ship did not send the signal for a long time and, during that time, travelled quite some distance. The exact thresholds can be changed inside that class.

How to change the threshold of any of the heuristic checks?
To change the threshold, you must go to the class that checks if a signal is an anomaly based on your desired heuristic.
For example, if you want to change the speed threshold, you have to go to the class `SpeedStatefulMapFunction` (`backend/src/main/java/sp/pipeline/parts/scoring/scorecalculators/components/heuristic/SpeedStatefulMapFunction.java`). At the top of this class, you can find `private static final` variables used for thresholds. One of them is `SPEED_THRESHOLD`.
Similar steps can be performed to change any other threshold.

[1]Check the Flink documentation at https://nightlies.apache.org/flink/flink-docs-release-1.19/docs/dev/datastream/fault-tolerance/state/ and https://nightlies.apache.org/flink/flink-docs-release-1.19/docs/concepts/stateful-stream-processing/.

How to use the past state in the score calculation?
Flink's Rich Map Functions provide availability to use states, such as Value State and List State. The example usage of them can be found in classes `HeuristicStatefulMapFunction` (has the variable `private transient ValueState<AISSignal> aisSignalValueState`) and `ManeuveringStatefulMapFunction` (has the variable `private transient ListState<AISSignal> previousSignalsListState`). You can check how they are used there. In essence, the steps to use the State are the following (these can also be found in the mentioned example files):

- You need to have the `private transient` state variable in the class.
- In the `open` method, initialize the state for that variable with the new state descriptor for the current Flink environment. You can reuse `getValueState` and `getListState` methods from `HeuristicStatefulMapFunction` class.
- In the `map`, you can get and update the values of the state. The methods are described in Flink's documentation: `https://nightlies.apache.org/flink/flink-docs-release-1.19/docs/dev/datastream/fault-tolerance/state/#using-keyed-state`.

Note that you should be careful when using List State as it can have a heavy load on the memory.

How to add a new heuristic?
As described at the start of this section, the currently used anomaly score calculator is described in `backend/src/main/java/sp/pipeline/parts/scoring/scorecalculators/SimpleScoreCalculator.java`. In its method `setupFlinkAnomalyScoreCalculationPart`, you can see that the four heuristics are used. You can also check the implementation of those heuristic classes to see how to implement one yourself. To add a new heuristic, you should add a class that extends the abstract class `HeuristicStatefulMapFunction` and override the abstract methods. Then, add this new heuristic class (that you have created) to `SimpleScoreCalculator`, similarly to how other heuristics are added. Note that you should change the `getAnomalyScore` method in the currently existing heuristics so that the sum of the anomaly scores among the heuristics adds up to 100. Another way would be to modify the score calculator to always normalize this score to 100.

How to add a new anomaly score calculation strategy?
The currently used anomaly score calculation strategy is `backend/src/main/java/sp/pipeline/parts/scoring/scorecalculators/SimpleScoreCalculator.java`. Similarly, you can create a new one. It must implement the interface `backend/src/main/java/sp/pipeline/parts/scoring/scorecalculators/ScoreCalculationStrategy.java`. You must override the method `setupFlinkAnomalyScoreCalculationPart` where you implement the logic of how a stream of AIS signals is turned into a stream with objects with calculated scores.
Once you have multiple score calculators, you should specify which one is used in the constructor of the `ScoreCalculationBuilder` in the `Qualifier` annotation (`backend/src/main/java/sp/pipeline/parts/scoring/ScoreCalculationBuilder.java`).

3.2.3 Pipeline Part: Score Aggregation

The stream of AIS signals with IDs assigned and the stream of the calculated scores (these two streams are computed in the pipeline parts described above) are then aggregated to the instances of class `backend/src/main/java/sp/model/CurrentShipDetails.java`. These objects are serialized and sent to the Kafka topic `ships-history`.

When setting up the Apache Druid database, the supervisor for the Kafka topic `ships-history` should be created[2]. This allows the database to save the events sent to this topic.

The critical detail is that two aggregated signals are generated for each AIS signal. At first, the event representing the AIS signal with calculated ID is sent to the mentioned Kafka topic. Later, once the anomaly score is calculated, it is combined with the corresponding AIS signal and the updated details are sent to the topic again. All of this is set up in `backend/src/main/java/sp/pipeline/parts/aggregation/ScoreAggregationBuilder.java`, and the aggregation logic is implemented in `backend/src/main/java/sp/pipeline/parts/aggregation/aggregators/CurrentStateAggregator.java`. Note that the order of these two events can be flipped in some rare cases (due to distributed and parallel computing): the aggregation will first get the calculated score and only later the AIS signal. However, aggregator logic is implemented so that the order of these events does not matter.

3.2.4 Pipeline Part: Notifications

The last part in the pipeline is the generation of the notifications. The notification detection takes a stream of aggregated ship details (described above), and the stream of notifications is generated using the notifications aggregator. These notifications are sent to the Kafka topic `notifications`. The notifications extractor frequently polls from this topic, so the API endpoints work correctly.

The notifications creation (stream aggregation) logic is implemented in `backend/src/main/java/sp/pipeline/parts/notifications/NotificationsAggregator.java`.

3.2.5 Extractors

The extractors are classes that poll from a specified Kafka topic with a specified polling frequency. An abstract class for an extractor is described in the file `backend/src/main/java/sp/pipeline/parts/aggregation/extractors/GenericKafkaExtractor.java`. The classes that inherit this class must override the method `processNewRecord`, which specifies how a consumer record from a Kafka topic should be processed.

These extractors keep the required objects ready to be asked by the service classes. For example, the `ShipsDataService` (described in `backend/src/main/java/sp/services/ShipsDataService.java`) calls the `ShipInformationExtractor` (`backend/src/main/java/sp/pipeline/parts/aggregation/extractors/ShipInformationExtractor.java`) to get the current ship information. Ship Information Extractor can provide the current ship information stored inside the hashmap, which is updated by processing new recors. `NotificationsExtractor` works analogously.

3.3 Spring Boot Web Server

The web server handles the logic for the API endpoints. The full description of the endpoints can be found in the OpenAPI specification (see Appendix A).

[2]The steps to do that are specified in `backend/README.md`.

The web server is implemented using a layered architecture. The controllers are described in `backend/src/main/java/sp/controllers`. They have the Spring Boot's annotation `RestController`, and the Spring Boot framework ensures that the required methods are called in the controller classes. Controllers call service classes to be able to handle the request. Service classes are described in `backend/src/main/java/sp/services`. If services need to get ship or notification data (which are coming to Kafka topics, see 3.1), then they call the extractor classes (see 3.2.5).

Chapter 4

Frontend

The general structure of the React components and other frontend code is described in Section 4.1. The way the map is rendered and ship clustering works is described in Section 4.2. Some frontend values can be changed in the configuration files, which is explained in Section 4.3.

4.1 Structure of the frontend code

4.1.1 React Components

The components implemented and their hierarchy are as follows:

```
App
|__ LMap
|__ Side
    |__ InformationContainer
    |    |__ AnomalyList
    |    |__ ObjectDetails
    |    |__ NotificationList
    |    |__ Settings
    |    |__ ErrorList
    |
    |__ Sidebar
```

These React components are implemented in the file `frontend/src/App.tsx` and the folder `frontend/src/components`.

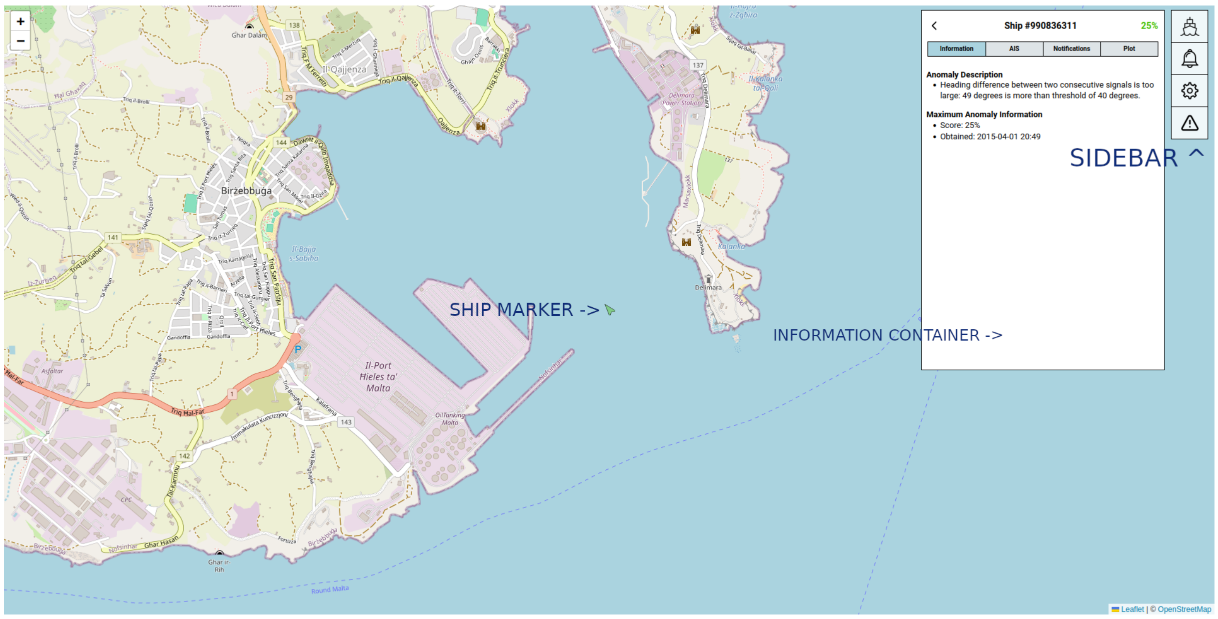
A view of the website showing some of the mentioned components can be seen in the Figure 4.1.

Some of the mentioned components (`Side` and `LMap`) were implemented as `forwardRef`. This allows for reference of some parts or methods from the upper components, which would otherwise not be possible. Such a design choice was made to reduce React rerendering as much as possible. When a variable that is a state[1] in some component is updated, the whole component (and its children) is rerendered. Because of that, variables like `pageChanger` were moved to the component `Side`. However, they are still needed for ship markers (created by the `LMap` component). Using `forwardRef` allows reaching them without rendering the map every time the state changes.

4.1.2 Other files

Apart from the React components, the other files are:

[1]Check `useState` hook at https://react.dev/reference/react/useState.



Anomaly Description
- Heading difference between two consecutive signals is too large: 49 degrees is more than threshold of 40 degrees.

Maximum Anomaly Information
- Score: 25%
- Obtained: 2015-04-01 20:49

Figure 4.1: *View of the website showing the mentioned components: ship marker, information container and sidebar.*

- `assets` - the folder where all the icons are put.
- `configs` - the folder with main configuration values.
- `model` - the folder with the model classes representing the objects retrieved by HTTP requests from the backend.
- `services` - the folder for the service classes.
- `styles` - the folder with CSS styling files.
- `templates` - the folder with some template interfaces used as intermediate objects for the values retrieved by HTTP requests.
- `utils` - the folder with helper methods.

4.2 Map

Map component and all of the related code can be found in the folder `frontend/src/components/Map`. The map is implemented using `Leaflet` library[2], and the clustering of the ship markers is done with the help of the `Leaflet.markercluster` library[3].

4.2.1 Map Rendering

To initialize the map only once (and not initialize it again on every rerendering), the map is saved as a reference[4], and only initialized once using with `useEffect` hook[5]. The map is initialized by the `initializeMap` method inside `frontend/src/components/Map/LMap.tsx`.

[2]`https://leafletjs.com/`
[3]`https://github.com/Leaflet/Leaflet.markercluster`
[4]Check `useRef` hook: `https://react.dev/reference/react/useRef`
[5]`https://react.dev/reference/react/useEffect`

How to change map properties?
Possible options for the Leaflet Map (such as max bounds, zoom level, etc.) are described in its documentation: `https://leafletjs.com/reference.html#map`.
Any of these options can be used during the initialization of the map, which is done in the method `initializeMap` method inside `frontend/src/components/Map/LMap.tsx`.

How to change the map used?
Currently, the tile layer is using OpenStreetMaps. It is created in the method `getTileLayer` inside `frontend/src/components/Map/LMap.tsx`.
However, Leaflet provides more possibilities for the tile layers via plugins. They can be found on the website and used in the mentioned method. Check the official Leaflet website: `https://leafletjs.com/plugins.html#basemap-providers`.

4.2.2 Ship Markers Rendering

Every time the state `ships` is changed (for example, due to frequent polling in the backend), the ship markers on the map are rerendered. It is done by the method `updateMarkersForShips` inside `frontend/src/components/Map/ShipMarkerCluster.tsx`.

To update the markers on the map, the old ones are first cleared, and then the new ones are created and added as a batch to the Marker Cluster Group. After adding them to the marker cluster, the `Leaflet.markercluster` library figures out how to render them and groups some markers into clusters so that the screen view is not cluttered. Clusters show the maximum anomaly score among the ships grouped into it (implemented inside the method `getClusterIcon`).

To speed up the rendering, the `doFilteringBeforeDisplaying` is set to `true` in the configuration files (it is set to `true` by default). This forces to filter the ships which are not on the screen currently so that only the visible markers are added. This also shows only the most anomalous ships. Without such filtering, rendering the map with all the ships becomes too slow once the number of ships is too large (it also depends on the computer the website is running on).

4.2.3 Why filtering was implemented?

As mentioned above, if the value `doFilteringBeforeDisplaying` is set to `true`, the ships are filtered before displaying them on the screen. This is done to tackle the two major bottlenecks of the rendering. These bottlenecks arrise from the fact that the markers and clusters on the screen need to be frequently updated (old ones deleted, and new ones added). If the ship positions would be static for a long period of time, this would not be a problem since when the markers are added and clusters calculated, then the UI works smoothly.

The first mentioned bottleneck is the rendering of the markers. Leaflet becomes slow and freezes the website if it needs to load a lot of markers. For example, that means that it cannot handle adding 10 000 ship markers every second. Grouping the ships into the cluster makes this a bit better. However, the way the `Leaflet.markercluster` library is implemented still forces to add a lot of markers, and that also means there is a huge computation overhead. This is because calculating the clusters is the second bottleneck. Recalculating the clusters have to be done frequently, otherwise they do not represent the accurate information.

To solve these problems, the filtering was implemented. When filtered, only the 50 (or any other number selected in the configuration files) most anomalous ships that could be seen on the current screen are displayed (and clustered). This way, the user still sees many ships, and the UI remains responsive. If the users wants to see more ships, they can zoom in to the wanted region.

As a note, if there are not too many ships, and the UI can handle this (all of this vary on different computers), then the filtering can be turned off, so that all the ships are always rendered.

4.3 Configuration for the frontend

The configuration files are placed in the folder `frontend/src/configs`.

4.3.1 File `errorListConfig.json`

The default values are:

```json
{
  "savedNotificationsLimit": 1000,
  "shownNotificationsLimit": 100
}
```

Explanation of the values:

- `savedNotificationsLimit` - how many error notifications at most are saved in the `ErrorNotificationsService`.
- `shownNotificationsLimit` - how many errors at most are shown in the error notification list that can be accessed on the website through the Sidebar.

4.3.2 File `generalConfig.json`

The default values are:

```json
{
  "notificationsRefreshMs": 1000,
  "shipsRefreshMs": 2000,
  "anomalyListMaxEntries": 250,
  "notificationListMaxEntries": 250
}
```

Explanation of the values:

- `notificationsRefreshMs` - how frequently the notifications (about anomalous ships) are polled from the backend. This value is in milliseconds.
- `shipsRefreshMs` - how frequently the ship data is polled from the backend. This value is in milliseconds.
- `anomalyListMaxEntries` - how many entries at most are shown in the anomaly list (that can be accessed through the Sidebar). This is used so that the rendering does not slow down too much.
- `notificationListMaxEntries` - how many entries at most are shown in the notifications (about anomalous ships) list. This is used so that the rendering does not slow down too much.

4.3.3 File `mapConfig.json`

The default values are:

```json
{
    "centeringShipZoomLevel": 15,

    "doFilteringBeforeDisplaying": true,
    "maxShipsOnScreen": 50,

    "clusterChunkedLoading": true,
    "clusterChunkInterval": 200,
    "clusterChunkDelay": 100,
    "clusterMaxRadius": 80
}
```

Explanation of the values:

- `centeringShipZoomLevel` - the zoom level used when the ship is centred (triggered when clicking on the ship in the list of anomalies or notifications).
- `doFilteringBeforeDisplaying` - whether to do filtering of the ships before rendering them. Ship rendering is described in Section 4.2.2.
- `maxShipsOnScreen` - the number of ships shown on the screen. All ships are shown if it is set to `-1`. It only takes effect if the value `doFilteringBeforeDisplaying` is set to `true`.
- The last four values (`clusterChunkedLoading`, `clusterChunkInterval`, `clusterChunkDelay`, and `clusterMaxRadius`) are used to set up the marker cluster. To see their effect, refer to the official `Leaflet.markercluster` documentation[6].

[6] https://github.com/Leaflet/Leaflet.markercluster?tab=readme-ov-file#chunked-addlayers-options

Chapter 5

Simulator

The simulator is an application that takes the given dataset of AIS signals and starts simulating them from a specified timestamp until some specified end time and with a specified speed. When started, the simulator reads the given dataset, sorts the signals based on their time, and sends them one by one to the Kafka topic `ship-raw-AIS` (see 3.1.1).

How to customize the simulator?
Everything is explained in the file `simulator/README.md`. In short, you can change the constants in the method `main` in `simulator/src /main/java/Main.java` before running the simulator.

Chapter 6

Stress Tester (Scalability Testing)

Stress tester is meant to test how well the application can handle higher data loads.

This tester works similarly to the simulator (see Section 5). The only difference is that it does not take a dataset but instead generates random AIS signals. The randomly generated AIS signals are sent to the Kafka topic `ship-raw-AIS` (see Section 3.1.1).

The instructions on how to run it can be found in `stress-tester/README.md`. The main value to customize is `signalsPerSecond` in `stress-tester/src/main/java/Main.java`.

Chapter 7

Distributing the Application

The backend of the application contains three large stream-related components that can all be distributed: Kafka, Flink and Druid. With the final version of the application, the main bottleneck concerning the throughput is located in the anomaly score calculation part, which is done entirely in Flink. Therefore, a considerable effort was made during the development process to ensure this bottleneck can be made larger by distributing Flink. Therefore, we describe how to distribute the application with a clear focus on distributing Flink.

By default, when the application starts, an embedded internal Flink cluster is spawned entirely inside of the web server. That is different from what happens with Kafka and Druid, which are fully external services that run independently of our application. The fact that Kafka and Druid are independent from our application allows scaling Kafka and Druid very easily - there are tens of tutorials online on how to distribute these systems, and that can be done separately from our application. The only things that might have to be changed when doing that are the connection, IP addresses, ports, and connection URLs (all in the 'kafka-connection.properties' file).

Fortunately, we have implemented our application, making sure that distributing Flink is also easy. In order to distribute Flink, 3 general steps have to be made:

1. Flink binaries should be downloaded from the Flink repository.
2. An external Flink cluster should be started according to some specifications (we provide a custom specification that we use later).
3. A few configuration properties should be changed in the main application.

We will discuss all of these steps in greater detail next.

7.0.1 Downloading Flink Binaries

Unlike what is happening in the application by default, where Flink binaries are downloaded and managed fully inside of the application using Gradle, for the distributed settings, a separate Flink download has to be performed. In particular, we have used Flink 1.19.0 binaries[1]. Just like Kafka and Druid, it is enough to have Java to be able to work with them. In case of Windows, you should also use WSL.

Additionally, note that you have to download these binaries to every computer/node that will be part of the cluster. Each node will have to run some Flink application, so they all need Flink binaries.

[1] `https://www.apache.org/dyn/closer.lua/flink/flink-1.19.0/flink-1.19.0-bin-scala_2.12.tgz`

7.0.2 Starting External Flink Cluster

A Flink cluster consists of two main types of components. In our fairly simplified case, that would be a single Job Manager and multiple Task Managers. In essence, a Job Manager is an application that controls Task Managers. A Flink job (in our case, a Flink job is our whole score calculation pipeline) is submitted directly to the Job Manager, which then decides how to distribute this job among Task Managers.

To put it simply, there will be one computer running a Job Manager and multiple interconnected computers that each run a Task Manager, all connected to the same network. The computer that runs the Job Manager can also run a Task Manager. So starting a Flink cluster is as simple as starting a Job manager at a single computer and starting multiple Task Managers in other connected computers.

Before starting the Job and the Task Managers, you first have to write a configuration file for every machine in the cluster. The key settings to change are:

- `jobmanager.bind-host` - we changed it to `0.0.0.0`. It makes sure the Job Manager accepts requests from all over the network.
- `taskmanager.bind-host` - we changed it to `0.0.0.0`.
- `rest.bind-address` - we changed it to `0.0.0.0` to ensure the Flink UI panel is accessible from all of the network.
- `jobmanager.rpc.address` - change it to the IP of the computer running the Job Manager. You can find this IP by running `ifconfig` on the terminal (or `ipconfig` if you are using Windows). In case you are using Windows this, (and all other similar) IP should be of the main Windows computer, not of the WSL container.
- `rest.address` - change it to the IP of the computer that is running the Job Manager

These were the key settings required for networking purposes. We have additionally experimented with other configurations a lot in order to achieve better results (and make them stable, since the default Flink settings resulted in a lot of errors being generated due to occasional lack of resources). Therefore, we suggest using the following configurations:

```
cluster.io-pool.size: 4
parallelism.default: 18
pekko.ask.timeout: 5 min
taskmanager.memory.framework.heap.size: 128 mb
taskmanager.memory.framework.off-heap.size: 128 mb
taskmanager.memory.jvm-metaspace.size: 256 mb
taskmanager.memory.jvm-overhead.max: 1 gb
taskmanager.memory.jvm-overhead.min: 1 gb
taskmanager.memory.managed.size: 128 mb
taskmanager.memory.network.max: 300 mb
taskmanager.memory.network.min: 300 mb
taskmanager.memory.task.heap.size: 1011627776 bytes
taskmanager.memory.task.off-heap.size: 1099511627 bytes
taskmanager.network.sort-shuffle.min-buffers: 16
taskmanager.numberOfTaskSlots: 18

blob.server.port: 6124
taskmanager.data.port: 6121

jobmanager:
  bind-host: 0.0.0.0
  rpc:
    address: <IP>
    port: 6123

  memory:
    process:
      size: 1600m

  execution:
    failover-strategy: region

taskmanager:
  bind-host: 0.0.0.0
  host: <IP>
  rpc.port: 6129
  collect-sink.port: 6130

rest:
  address: <IP>
  bind-address: 0.0.0.0
  port: 8084
```

Note that for readability purposes, we have skipped `env.java.opts.all` configuration property that is by default in the Flink configuration file.

> In case you are using WSL, you may also need to ensure that all ports that are used by Flink inside of WSL are exposed as the ports of the host Windows machine. To do that, a Powershell terminal with Administrator permissions should be started, and the following command should be run for each port that is mentioned in the configuration file:
> `netsh interface portproxy add v4tov4 listenport=<port> listenaddress=0.0.0.0 connectport=<same port> connectaddress=<WSL IP>`
> Where `<WSL IP>` is the IP of the WSL container that can be found by running `ifconfig` from inside of WSL.

After the configuration files have been written, a single Job Manager needs to be started on one machine. To start that Job Manager, locate the Flink binaries in the terminal and start the Job Manager script:

```
cd <path/to/flink-1.19.0-bin-scala_2.12/flink-1.19.0>
bin/jobmanager.sh start-foreground
```

Afterwards, the following commands should be run on all machines in the cluster:

```
cd <path/to/flink-1.19.0-bin-scala_2.12/flink-1.19.0>
bin/taskmanager.sh start-foreground
```

If all goes well, you should be able to open the following URL in the browser: `<JOBMANAGER_IP>:8084` and see all Task Managers listed.

7.0.3 Changing Backend Configuration

First, a few ports and IP addresses have to be updated in the `kafka-connection.properties` file. In particular, the following ones should be adjusted:

```
# Kafka server URL (to be changed if external Flink cluster is used)
kafka.server.address=<kafka_IP>:<kafka_port>
bootstrap.servers=<kafka_IP>:<kafka_port>

flink.job.manager.ip = <job_manager_ip>
flink.job.manager.port = 8084
flink.parallelism = <parallelism>
```

In particular, the Job Manager's IP should be specified and the desired application parallelism should be specified. The parallelism should not exceed the number of slots in the cluster.

Additionally, in case Kafka was running locally (on `localhost`), you might have to update its IP address to an IP in the LAN. For that, you may have to restart Kafka and add the following two lines to Kafka's `config/server.properties`:

```
listeners=PLAINTEXT://0.0.0.0:9092
advertised.listeners=PLAINTEXT://<IP>:9092
```

Where `IP` is the IP address of the computer running Kafka. These two configuration changes (adding the two lines to Kafka configuration and adjusting the two Kafka-related lines in the `kafka-connection.properties` file) allow for all machines in the Flink's cluster to connect to the Kafka server.

The final change to make sure the external cluster is used in the application is to change which Java bean for the Flink environment is used in the application. In particular, in `AnomalyDetectionPipeline.java` class constructor, the qualifier has to be changed from `localFlinkEnv` to `distributedFlinkEnv`.

If all of these steps are done successfully, when started, the application will submit a job to the external Flink cluster, which will distribute the job to Task Managers.

Appendix A

OpenAPI Specification

Track Anomaly Detection - OpenAPI 3.0

Overview

This represents the OpenAPI Specification of the backend server of the application.

Tags

ships

Ship Information management component

notifications

Notification management component

Paths

GET `/ships/details/{id}`

Retrieve the most recent information about a ship.

Parameters

Type	Name	Description	Schema
path	**id** *required*	The ID of the ship	number

Responses

Code	Description	Links
200	Successful operation *Content* **application/json**	No Links
404	Ship not found	No Links
425	Backend pipeline still starting	No Links
500	Internal server error	No Links

GET `/ships/details`

Retrieve the current details of all the ships in the system.

Responses

Code	Description	Links
200	Successful operation *Content* **application/json**	No Links
425	Backend pipeline still starting	No Links
500	Internal server error	No Links

GET `/ships/history/{id}`

Retrieves all the CurrentShipDetails intances of the respective ship.

Parameters

Type	Name	Description	Schema
path	**id** *required*	The ID of the ship	number

Responses

Code	Description	Links
200	Successful operation *Content* **application/json**	No Links
500	Internal server error	No Links

GET `/notifications`

Gets all the notifications stored in the database

Responses

Code	Description	Links
200	Successful operation *Content* **application/json**	No Links

GET `/notifications/{id}`

Gets a certain notification from the database.

Parameters

Type	Name	Description	Schema
path	**id** *required*	The ID of the notification to be fetched	number

Responses

Code	Description	Links
200	Successful operation *Content* **application/json**	No Links
404	Notification not found	No Links

GET `/notifications/ship/{id}`

Gets all notifications of a particular ship.

Parameters

Type	Name	Description	Schema
path	**id** *required*	ID of the ship	number

Responses

Code	Description	Links
200	Successful operation *Content* **application/json**	No Links

Components

Schemas

CurrentShipDetails

Properties

Name	Description	Schema
currentAnomalyInformation *optional*		AnomalyInformation
currentAISSignal *optional*		AISSignal

Name	Description	Schema
maxAnomalyScoreInfo *optional*		MaxAnomalyScoreDetails

Notification

Properties

Name	Description	Schema
id *optional*	ID of the notification	number (long64)
shipId *optional*	ID of the corresponding ship	number (long64)
currentShipDetails *optional*		CurrentShipDetails

AnomalyInformation

Properties

Name	Description	Schema
score *optional*	Computed Anomaly Score	number (float64)
explanation *optional*	Corresponding explanation of the computed score	string
correspondingTimestamp *optional*	Corresponding timestamp	string (date-time)
id *optional*	ID of the corresponding ship	number (long64)

AISSignal

Properties

Name	Description	Schema
id *optional*	ID of the corresponding ship	string (long64)
speed *optional*	Speed of the vessel	number (float64)
longitude *optional*	Longitudinal coordinates of the vessel	number (float64)
latitude *optional*	Latitudinal coordinates of the vessel	number (float64)

Name	Description	Schema
course *optional*	Course of the ship	number (float64)
heading *optional*	Heading of the ship	number (float64)
timestamp *optional*	Timestamp of the signal	string (date-time)
departurePort *optional*	Departure port	string
receivedTime *optional*	Received time	string (date-time)

MaxAnomalyScoreDetails

Properties

Name	Description	Schema
maxAnomalyScore *optional*	Maximum anomaly score assigned to the respective ship	number (float64)
correspondingTimestamp *optional*	Timestamp corresponding to the maximum acheived anomaly score	string (date-time)