JOHANNES KEPLER
UNIVERSITY LINZ

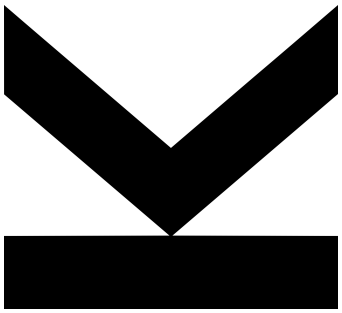
Author
Philipp Wandl
k01558060

Submission
Institute of Networks and Security

Thesis Supervisor
Assoz.-Prof. Mag. DI Dr.
Michael Sonntag

September 2022

Hidden Glider Finder



Master Thesis

to obtain the academic degree of

Diplom-Ingenieur

in the Master's Program

Computer Science

JOHANNES KEPLER UNIVERSITY LINZ
Altenbergerstraße 69
4040 Linz, Austria
www.jku.at
DVR 0093696

Abstract

Glider pilots might be flying alone without any assistance for hours, but in case of distress, timely assistance may be required, however the exact position of the pilot is potentially not known to anyone else. The implemented system aims to leverage existing systems and data sources like the OpenGliderNet and custom data relays to provide a near real-time collection and analysis of publically available data to find missing pilots more easily, while still maintaining a high degree of data privacy.
This thesis presents the design, implementation and deployment of the software system, which comprises multiple microservices to provide an automatic incident detection including a notification system and a visual presentation in a web application. Two different types of deployments are created and compared to highlight the differences between a modern cloud system and a more classic deployment. The testing and verification of the system is presented, leveraging custom tools and even Microsoft Flight Simulator to be less dependent on real world tests.

Kurzfassung

Segelflieger können stundenlang ohne jegliche Hilfe unterwegs sein, aber in einer Notlage kann zeitnahe Hilfe notwendig sein, auch wenn die exakte Position des Piloten möglicherweise niemandem bekannt ist. Das implementierte System nutzt bestehende Systeme und Datenquellen wie OpenGliderNet und ein selbstentwickeltes Datenrelay um in near real-time öffentlich verfügbare Daten zu sammeln und analysieren um vermisste Piloten leichter auffinden zu können, ohne die Datenschutzinteressen der Betroffenen zu verletzen.
In dieser Arbeit werden Design, Implementierung und Deplyment des Softwaresystems präsentiert. Dieses besteht aus mehreren Microservices welche unter anderem eine automatische Ereigniserkennung, ein Benachrichtigungssystem und eine visuelle Darstellung der Daten in Form eine Webanwendung bereitstellen. Zwei verschiedene Deployments wurden erstellt um die Unterschiede zwischen einem modernen Cloud-System und einem klassischeren Deployment zu beleuchten. Testen und Verifikation des Systems erfolgt mit Hilfe von speziell für das System entwickelten Tools, aber auch Microsoft Flight Simulator, um unabhängiger von realen Testflügen zu sein.

Contents

List of Figures

List of Tables

1 Introduction

While humanity has achieved powered flight, some pilots still prefer the challenge of gliding flight. Atmospheric conditions are utilized to keep flying in these unpowered aircraft. Due to technical advancements and improved understanding of the meteorological conditions, the sport has evolved from just staying airborne to covering significant distances. Despite all regulation and training efforts, gliding is not hazard-free, so safety systems are an important part of the sport. Especially cross-country flights where pilots possibly are far away from their home airfield pose a risk of aircraft disappearing on their way. The system presented in this thesis aims to tackle this challenge and provide improvements to detect and help glider pilots in distress as soon as possible.

1.1 Idea

This project aims to improve search and rescue operations for gliders by collecting available position data of opted-in gliders in near real time. In case of an incident, the system may be able to provide more complete data than current systems, and significantly faster. While there are various types of tracking and surveillance systems, most of them focus on a very specific part of a bigger problem. The target is to provide an easy-to-use system which supports glider pilots and their environment from incident avoidance to improved data for search and rescue operations. As shown in chapter 2, current systems focus on parts of the problem like surveillance (e.g. Glidernet) or emergency alerting (e.g. ELTs). The system presented in this thesis aims to provide a simple single point of entry for users and extended functionality by leveraging and combining the functionalities and data of existing systems when possible. It provides additional functionality based on the available data like incident detection and is designed for easy future extension, both for new data

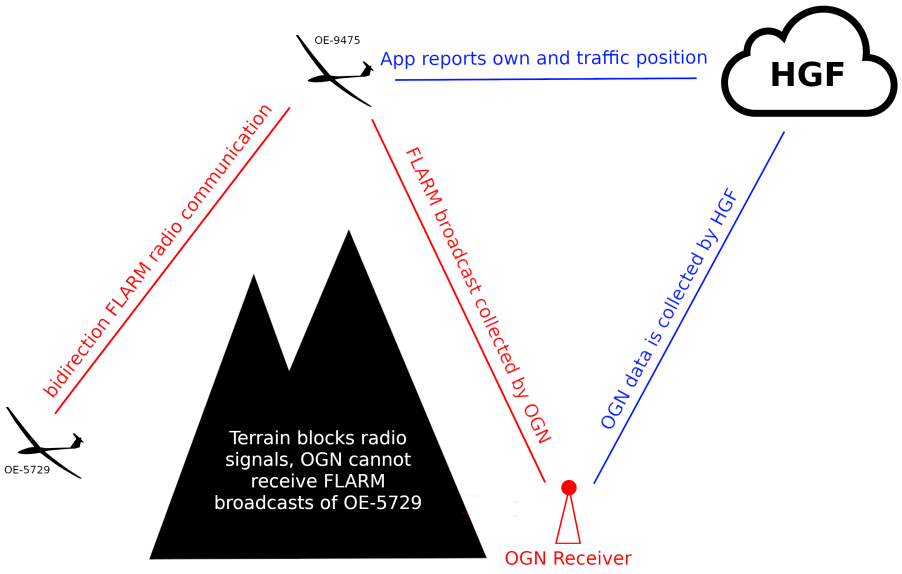


Figure 1.1: Radio ranges and communication paths

processing algorithms and new data sources.

An important goal of the project was to provide functionality similar to the existing FLARM Search and Rescue (SAR) protocol, but significantly reducing the required time until meaningful data can be provided.
FLARM is a traffic awareness and collision avoidance system which is very common in gliders, but also other aircraft like general aviation planes and helicopters. As a lot of gliders are equipped with FLARM and therefore continuously radio broadcast their position it is possible to track those gliders by receiving those radio messages. Currently, projects like the Open Glider Network (OGN) operate a network of ground receivers, collect those radio messages and provide the data to the public [1]. While this already provides lots of helpful information, it is hardly possible to cover all relevant areas with ground receivers, especially in mountainous regions. But gliders in particular tend to visit those areas in search of good soaring conditions, often flying below ridges and therefore out of radio contact with any ground stations. As gliders often accumulate at hotspots with good conditions, an aircraft out of range of the ground receivers could still be in

FLARM range of another aircraft, as illustrated in figure 1.1.
This is already considered in the FLARM Search and Rescue (SAR) protocol, where FLARM logs are used to trace flight paths of missing gliders. In case of a suspected accident with unknown accident location, pilots who have been flying in the relevant area are asked to download the logs from their FLARM devices and send them to the manufacturer. With these logs a possible flight path of the missing aircraft is reconstructed to direct search and rescue efforts[2]. By collecting the received position reports in a glider and forwarding those via mobile networks, a glider could fulfill similar duties as a ground receiver. Combined with the fact that gliders often fly similar routes, this could greatly improve the tracking coverage and be a valuable asset for search and rescues operations. This provides an additional data source for the system to use. Due to the fast and easy data access via a web application, this data is available almost immediately.

While the system needs to collect data to provide meaningful features, data protection is still important. Therefore, mechanisms need to be implemented to protect the collected data from unauthorized access without creating dangerous barriers in case of an emergency.

1.2 Scope

The system shall provide a helpful tool for glider pilots and their environment like flight instructors, club mates and airport operations manager. It is not intended to replace any existing systems or authorities. It does not issue alerts with absolute certainty and does not automatically dispatch search and rescue personal in case of a detected incident. However, it does notify pre-defined (per glider) persons of trust who can then use their best judgement of the data provided by the system to take further steps.

2 Current Systems and Limitations

Aircraft surveillance is an important part of flight safety, so there are systems in place to achieve these goals. However, these systems are often tailored to quite specific tasks.
Especially gliding flight provides further challenges. Compared to general aviation, gliders might seem more unpredictable as they are not able to stay at a specific cruise altitude and need to vary their course depending on the environmental conditions. Especially recreational pilots often have no specific predefined destination which makes their flights even harder to track. Depending on the terrain, gliders may fly close to or below mountain tops which may hide them from ground based tracking systems like radars. But it is also possible that gliders reach altitudes which put them out of reach of ground based systems like mobile phone networks.

Air traffic control usually relies on radars to localize air traffic. In Austria, this is the responsibility of the Austro Control which operates three secondary radars. In contrast to primary radars, which rely on the reflection of a pulse sent out by the system, secondary radars need transponders in the aircraft to provide data [3]. While those are required in most airspaces in Austria for powered aircraft, not all gliders are equipped with transponders or are not using them all the time [4]. And even when using them, mountainous topology often prevents accurate tracking at low altitudes.
Automatic Dependent Surveillance–Broadcast (ADS-B) might help the situation as it broadcasts the position of the aircraft determined via a system like GPS. This could enable better coverage by relying on less complex ground stations or even satellites to receive the data send by the aircraft [5]. However, in gliders this is usually implemented in the transponders, so the limitations of transponders usage still apply, and not all gliders which are equipped with transponders are equipped with ADS-B out capable transponders or have an appropriate data source like GPS attached.

Similarly to ADS-B, FLARM devices also broadcast the position of the glider. However, in contrast to ADS-B, FLARM devices were specifically designed for small aircraft like gliders or even para gliders as a traffic awareness and collision avoidance system. Its main goal is not to improve overall surveillance but to provide a traffic alerting system for pilots. FLARM devices receive the broadcasted locations of other aircraft in the vicinity and issue traffic alerts for collision avoidance if necessary. As it was designed for lower power usage, the range is limited and also depends on factors like the amount and installation of the antennas in the aircraft. But it is still possible to gather FLARM transmissions of airborne gliders with ground based receivers.
The Open Glider Network (OGN) operates a network of such ground based receivers and provides near real-time tracking of FLARM equipped aircraft by forwarding the data via APRS-IS which is used by multiple web and mobile clients. The limitations due to topology and range for ground based receivers still apply, but in some areas tracking of gliders in flight is quite reliable.
The FLARM developer, FLARM Technology, also provides a search and rescue (SAR) protocol in case a glider is missing and was not tracked by ground based systems. In this case, FLARM Technology requests pilots who have been flying in the area of the suspected flight path of the missing glider to download the logs from their devices and send them to FLARM Technology. Those logs contain the received messages from FLARM equipped traffic which was encountered during the flight. The logs are then analyzed with focus on the missing glider and a possible flight path is reconstructed to guide SAR operations. While this protocol has been proven to be effective, it is quite slow as it involves lots of manual time-consuming steps. The required time depends on how fast the data can be gathered, and can be as low as roughly an hour in optimal cases [2].

Another method of locating crashed aircraft is using Emergency Locator Transmitters (ELT). Those sense excessive g-forces during a crash and start transmitting a signal. The radio frequencies at which those signals are transmitted are monitored by the authorities. While ELT equipment is required in Austria, its usefulness is limited. ELTs may get damaged during a hard crash or not trigger at all if the g-forces are relatively low. The sending of the emergency signal requires an antenna, but the antenna might also get damaged during an impact or shielded by carbon fiber parts depending on the orientation of the wreckage [6].

Additional to the technical systems, human interaction is still a big part of flight safety and surveillance. Airfield operation managers monitor the flight operations close to the airfield and can spot incidents, but they also keep track of started and landed aircraft to see if a flight is missing. Club mates may notice that an aircraft is missing at the hangar in the evening and start the search. Pilots may check in on each other via radio and could therefore notice if a pilot who reported troubles does not react anymore. Other club mates might choose to watch other pilots flights from the ground using tracking apps to view OGN data. All these operations can be streamlined with a system which combines data sources and provides a helpful overview for the aircraft of interest.

3 Requirements

Based on the idea and overall goal, the following requirements have been chosen.

3.1 Functional Requirements

- The system shall provide an easy to understand visual representation of the collected data.
  - Tracked positions shall be visualized on a map.
  - Data points shall contain additional information like the data source and other data, like ground speed and heading, if available.
  - It shall be possible to only display data from selected data sources.
  - It shall be possible to filter data by time and aircraft.
  - Detected events shall be visible on the map.
  - Known airfields shall be displayed on the map.
- The system shall only provide data to personnel authorized by the system administrator.
- The system shall persist data only for whitelisted aircraft.
- The system shall provide a possibility to quickly escalate privileges of a user to access protected data in case of an emergency.
- The system shall provide a possibility to register contact data (e-mail addresses) for persons of trust per aircraft.

- Any data access shall be logged.
- Emergency privilege escalations shall be logged and administrators shall be notified of such accesses.
- Access and emergency access logs shall be easily viewable by administrators.
- The system shall periodically analyze the collected data to detect flight events of interest.
- Detected events shall cause a notification to be issued to administrators and the persons of trust.

3.2 Technical Requirements

- Build artifacts for web services shall be docker images.
- Backend services shall be horizontally scalable.
- Endpoints available via the internet shall use TLS.
- Continuous Integration: new changes shall be built and verified automatically.
- Continuous Delivery: new versions of web services on the master branch shall be deployed automatically.

4 Theoretical Concepts

4.1 Aircraft based data relay

As shown in figure 1.1, ground based collection of FLARM radio communication can be blocked by terrain and is therefore not reliable in all situations. This affects gliders especially, as they sometimes need to stay relatively close to mountains to make the best use of atmospheric conditions. A tighter net of ground receivers could help to improve the tracking coverage, but is not feasible everywhere.
As gliders usually tend to move to areas with favorable atmospheric conditions they accumulate at similar locations and sometimes fly similar routes. Additionally, the chance of terrain obstructing the radio communication path is significantly lower than with a ground based receivers. So the system tries to use gliders as data relays for FLARM messages of tracked gliders.
It is already common practice to connect smartphones to the avionics. Currently, this is used to feed data like barometric pressures or FLARM traffic into navigation apps [7]. But this connection can also be used to collect the FLARM traffic information for the tracking system presented here, so an implementation using a mobile app is possible without any hardware modifications in gliders which are already equipped with the necessary devices. Using today's powerful smartphones, it is also possible to buffer data in case no connection to the internet is available. Even when buffering data and transmitting it as soon as a connection is available again, it can still be significantly faster than following the FLARM SAR protocol and manually reading the logs from the devices after the flight.
To use aircraft as airborne receivers to relay data, only an Android smartphone with the HGF-app and TCP connectivity to a server providing the FLARM messages are required. The used gliders are equipped with devices which are connected to FLARM devices via a serial connection and relay the data via a TCP server. The forwarding devices also provide a WLAN access point via which the smartphones are connected. This feature might be

part of electronic instruments like variometers or navigation displays or purpose-built standalone components.
The FLARM protocol is based on the NMEA protocol, where data is transmitted in so-called sentences. FLARM uses some sentences which are specified by the NMEA protocol, but additionally also some custom sentences for data which was not considered in the NMEA specification [8]. The application only parses sentences which may contain relevant data, other sentences are ignored.
Those relevant sentences are:

GPRMC NMEA sentence providing GPS data including latitude, longitude, speed and direction. Used in combination with GPGGA data to determine the position of the user's glider.

GPGGA NMEA sentence also providing GPS data, but including latitude, longitude and altitude. Used in combination with GPRMC data.

PFLAA FLARM sentence providing data on other aircraft, possibly including the 3 dimensional relative position, track, turn rate, speed and more. Used to determine position of other aircraft, which can be sent to the collector.

PFLAU FLARM sentence including the current system status used as heartbeat. If some traffic is in the area, the sentence also contains traffic alarms, giving a rough direction of the most relevant traffic. Tested as additional data source to PFLAA messages, but disabled in favor of PFLAA messages.

PFLAC FLARM sentence providing configuration information. It was used experimentally to gather information about the connected FLARM device, but currently serves no important purpose.

Received data for whitelisted aircraft is buffered on the device until it is successfully uploaded to the collector endpoint of the system.
The Android App and its relevant endpoints have been developed in a previous project, but are an important part of the general idea and direction of the development of the system.

4.2 Incident detection

By continuously analyzing the collected data the system tries to detect anomalies in a flight which may indicate that an event of interest has occurred. Such events can be of varying nature, but the focus is on safety relevant events. These include landings at an airfield, off-field landings, and crashes. Due to the complexity of distinguishing an ordinary landing and a crash/crash landing, those events are currently grouped in the categories of "landing", where a safe landing is assumed, and "incident" where a crash is not proven, but more likely.

While the system is open for extension, currently only one detector algorithm is implemented. The "stop analyzer" tries to detect when an aircraft is stopped or almost stopped and depending on the distance to an airfield the stop is classified either as landing or as an incident.
In case of a stop close to an airfield, it is relatively safe to assume an ordinary landing instead of an incident, as the overwhelming majority of landings at airfields are not crash landings. This also helps to filter such common events to reduce the amount of notification when suppressing landing event notifications. However, if a crash landing at an airfield is falsely classified as ordinary landing, it is very likely that the event was observed by personnel at the airfield, like an airfield operations manager anyway, so it is quite likely that there are no negative consequences of suppressing the notification.
As an off-field landing will always require some kind of assistance, it is always classified as "incident" and no further distinction to a crash is made.

The algorithm is based on speed information of the aircraft. Depending on the data sources, this information might be available as part of the received information. Otherwise, the average speed is calculated based on the differences in time and position to the last reported position.
For the stop analysis, some predefined parameters are necessary. These have been chosen based on the expected behavior of the aircraft of interest, namely gliders and touring motor gliders which in general are aircraft with relatively low minimums speeds, but are subject to further tuning. The threshold speed below which an aircraft is assumed to not be flying is currently set to 10 m/s. This is significantly below the minimum speed of the

aircraft of interest and also provides some tolerance for inaccuracies, for example when the ground speed is calculated based on the position data, the actual airspeed (which is relevant for deciding whether the aircraft can continue flying) might be significantly higher due to headwinds. The algorithm assumes flight if the speed of the aircraft exceeds the predefined threshold speed. Consequentially, if the speed afterwards drops below the threshold, a "stop" is detected, except it the speed climbs above the threshold again in less than 30 seconds. Such cases where a stop is filtered due to the speed picking up again might be caused by flight maneuvers like touch-and-go landings, aerobatic flight or temporarily inaccurate sensor readings, e.g. during a side-slip.
To avoid multiple reports for a single event, all detected stops in a 60-second time frame are grouped to one single event.
The classification of the landing is considered to be at an airfield is based on airfield data provided by Austro Control [9]. If the position of the detected stop is in a 2 km radius around the airfield center and not more than 200 m above the airfield, it is considered to be located at the airfield. This simple model provides tolerances to allow for bigger airfields with multiple runways and inaccuracies in the data. Especially the reported altitude might be quite inaccurate, as different sources may use different sources and models - one system might report altitude based on barometric data which can be influenced by the atmospheric conditions, while another might use satellite positioning data.
An important limitation of the stop analysis is the fact that it can only analyze data which exists in the system. In case an aircraft loses tracking connectivity mid-flight, the algorithm will not detect an event. This might be uncritical if connectivity is restored or if the aircraft lands safely in an area without coverage, but might also miss a possible notification of a missing flight.

4.3 Data protection model

While most of the data the system processes is sourced from publically available sources, the detailed presentation and the inferred information from the data should not be publically available. So the system has to restrict the access of flight data to trusted personnel. But in case of an emergency, data protection is secondary to potentially rescuing a pilot. To achieve both goals, two authorization models have been combined: Role Based Access Control (RBAC) and the break-the-glass model (BTG). RBAC controls the standard

workflows, while BTG enables quick and uncomplicated access to data in case of an emergency.

4.3.1 Role Based Access Control

Role Based Access Control (RBAC) is the main model used in the application, which controls whether to grant or deny access to data or functionality for a given user.
With RBAC, users are assigned to roles which the system uses to grant or deny access. This allows the system to be unaware of concrete user identities when making access control decisions while only having to consider the roles of the user [10].
The available roles are predefined in the system, and it is not intended to create additional roles via the administration interface. Users can be assigned any combination of the following roles:

Admin Administrators are allowed to view any flight data, view access logs, view and manage BTG-logs, users, and fleet data

Viewer Viewers are allowed to view any flight data.

Reporter Reporters are allowed to upload data via the app.

Aircraft Viewer Allowed viewing data only of explicitly assigned aircraft.

An exception to the strict RBAC definition are the aircraft viewers, where users are directly assigned to allow access to a specific aircraft. This keeps the system relatively simple by just providing the absolutely necessary role and permission management features to administrators without the possibility to create a confusing amount of custom roles.
Additional to controlling access via those roles, any access to flight data is logged.

4.3.2 Break the Glass

The Break the Glass (BTG) model allows a user to quickly gain access to data in the system for which the user would normally not have permission, but requires the data for urgent matters. The name of the model aims to provide an analogy to breaking the glass of a fire alarm switch.
By using the BTG functionality users can decide when they need to override the strict

access rules of the system to resolve an urgent matter, like locating a potentially crashed aircraft. To detect abuse, BTG usage has to be audited by an administrator. The emergency access via BTG can be combined with other access models like the previously described RBAC by only granting access via BTG when the permissions via the main model are not sufficient to access the data [11].
The BTG model is only used for viewing flight data. In case the RBAC denies access to requested flight data, the user is informed that the request is denied but may be granted in case of an emergency by "breaking the glass", but any access will be reported to the administrator. If the user decides that it is necessary to access the data and confirms to break the glass, access to the requested data is granted immediately and a notification to the administrator is issued. This creates a BTG record in the system. Further access to the same data or a subset of it is granted without having to break the glass again as long as the BTG record is valid. A BTG record is valid for one hour after the initial breaking of the glass or until it is revoked by an administrator. In case of an invalid BTG record for the selected data or if additional data is requested, the glass has to be broken again.
Additional to the issued notification, administrators are able to see all BTG records in the system. Not only can an administrator revoke access if it seems unnecessary, it is also possible to approve such a BTG access. Approving only serves as information for other administrators who might review the access, so that it is visible that this action was already reviewed and deemed necessary by the approving administrator.

5 Implementation

The system consists of multiple components, which interface each other and also third-party systems. An overview of the system's components is shown in figure 5.1. The backend components follow the *Shared Database Server Pattern* where microservices share a common database server [12]. Some parts of the system have been developed in a pre-project and were extended as part of the thesis. The pre-project implemented the basic features for aircraft based data relay and OGN data collection, which also included the setup of the databases and authentication system. While the Android-App only received minor changes, components like the backend and frontend were significantly extended and improved as part of the thesis. The modular approach of the system design proved to be useful as it allowed to easily add and test new components without breaking existing functionality.

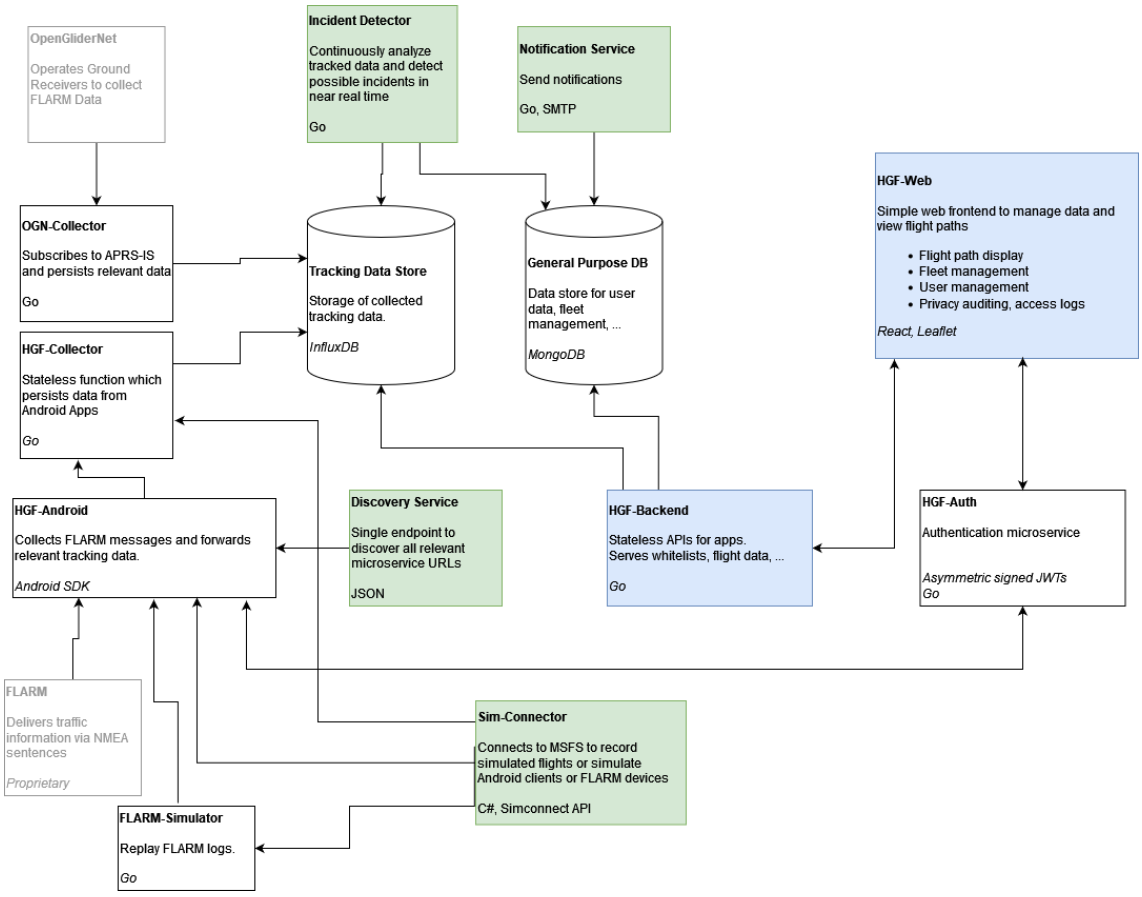


Figure 5.1: Overview of the System's Components with newly added components in green and significantly modified components in blue

5.1 Technologies

Whenever possible and reasonable, components were developed using Go as programming language [13]. Go was chosen as programming language due to its simplicity in development, building and deployment. Large, opinionated frameworks were mostly avoided as they would not fit the design philosophy of the small components. Third party dependencies were only used when they provided a significant benefit, otherwise it was avoided to introduce too many libraries and frameworks to keep things simple, easy to review and avoid introducing security issues via those dependencies. The Go components shown in 5.1 do not have compile-time dependencies on each other, so they can be built independently. Only the data analysis has been implemented in a Go module which is not deployed independently, but used as a compile-time dependency both in the HGF-Backend and the Incident Detector. This allows these components to reuse the analysis logic but also offers the possibility of different configurations - for example, one could add experimental analyzers to the backend and verify their functionality on existing production data without influencing and potentially breaking the Incident Detector.
The deployment units have been packaged into Docker containers to provide flexibility for deployment, as there are many solutions for deploying such containers. This provided enough flexibility to support two quite different deployments without having to significantly adapt the applications or build process, as described in chapter 6.

The technology stack for the HGF-Web component has been chosen to leverage modern frameworks and existing components. It is implemented as a single page application (SPA) using React, with the most important third-party dependencies being Leaflet to display flight data on a map using OpenStreetMap map data, and PicoCSS for styling [14, 15, 16, 17, 18]. While the web application can be served as static files by any web-server, it is also packaged as a Docker container to enable similar deployments to the other components.
The Sim-Connector was implemented using C# due to the availability of the SimConnect-SDK, which provides components to communicate with Microsoft Flight Simulator [19]. It is used as command line application on computers running Microsoft Flight Simulator for testing purposes and is not required for a production deployment.

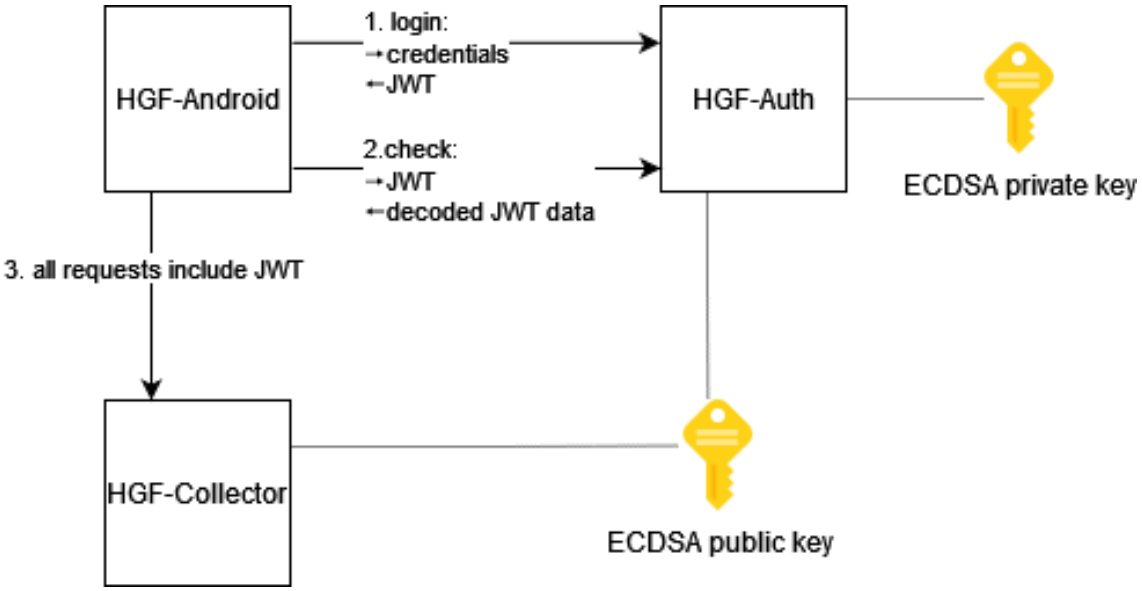


Figure 5.2: An example of the authentication flow demonstrated by HGF-Android and HGF-Collector

5.1.1 Authentication

JSON Web Token (JWT) was chosen to represent sessions for authenticated users and transmit session information. JWT is an open standard for securely transmitting data as JSON object, often used to transmit authorization claims [20]. This allows a simple and scalable way of transmitting the session data [21]. A JWT contains the username and assigned roles of the authenticated user. The JWTs are signed using ECDSA.
The authentication service, HGF-Auth, is only responsible for authentication, creating and verifying JWT tokens. Its functionality depends on the ECDSA private key, which is critical to be kept private. Reducing the functionality of the authentication service to a minimum also keeps the attack surface small. It can also be independently deployed from other components which makes it easier to meet performance requirements for this critical service.
An example of the authentication flow is shown in figure 5.2. As the JWT contains all the needed data, which is the unique username and the roles of the user, it is not necessary for the collector service to load data from a data store or another microservice for any request. This keeps the overhead to a minimum. Due to the chosen asymmetric ECDSA signing,

the collector service itself can verify the validity of the received JWT using the ECDSA public key. This key can however not be used to create new valid JWTs for future requests, making it less critical to manage this key. All services which need to validate JWTs require the public key.
Currently, no method of revoking sessions is implemented, except for changing the key pair. So once a user retrieves a correctly signed JWT from the authentication service it can be used until it expires. JWT specifies an optional "exp" field, which is used to set the expiration time after which the token has to be considered invalid. The expiration is currently set to 30 days after initial creation of the token. It has to be checked whenever validating a JWT.
To revoke a session, a new key pair has to be generated and pushed to the authentication service. All other services doing JWT verification need to change the public key to only validate the newly signed tokens. Not only is this quite an effort, it also invalidates all sessions, and it is not possible to only invalidate a single session. This is a drawback of this implementation, but there was no strong reason to require a simple session invalidation feature, so the system was kept as simple as possible. However, in units which are not trivially updatable as the Android app, some precautions were made.
In case of the Android app, the token is not validated directly on the device but validated against the authentication service, as depicted in the "check" call in figure 5.2. This would allow performing additional checks in the authentication service if a session might be revoked and force the user to re-login. Of course, other services which validate JWTs would have to implement a similar check. But as server-side services can be easily managed and updated when necessary, no precautions have been implemented on that side.

5.1.2 Data transfer

While some components share their state via the database (e.g. Incident Detector and Notification Service), others have to communicate directly, (e.g. HGF-Web and HGF-Android with HGF-Backend). Those communications are implemented via HTTPS calls, any data is serialized using JSON.

5.2 System Components

The following list gives an overview of the components shown in figure 5.1 and their functionality and relevant technical decisions.

General Purpose Database The general purpose database is used to persist data, except flight data. MongoDB was chosen over the commonly used relational databases for the added flexibility due to the schemaless design and simple usage for faster development [22]. Deployments locally or in cloud environments are easily done due to the amount of prebuilt packages and guides available. Performance was not really a consideration, as any widely used data store for such purposes will be able to handle the relatively small amount of data of this project.

Tracking Data Store The tracking data is stored in an InfluxDB. InfluxDB is a time series database designed to handle large amounts of time based data [23, 24]. Although the expected amount of data is currently rather small and should not be too big for most storage systems, it was decided to use an optimized time series database for multiple reasons. One reason being that the kind of collected data matches the use case for such databases quite well, another being the specialized tooling Influx provides which allows quick prototyping and testing of queries, including simple visualizations of the queried data.

OGN-Collector To gather data from the Open Glider Network an application has been developed using Go which connects to the APRS-IS and persists the data in the tracking data store. The Automatic Packet Reporting System (APRS) is an amateur radio based system to transmit data. Via the APRS internet service (APRS-IS), this system is also connected to the internet, which enables our application to easily connect to the system [25]. Only data of whitelisted aircraft is collected. To work correctly, the application needs to always maintain an open TCP connection to the APRS service to receive updates when they are broadcasted [26]. While running one instance of the OGN-Collector should be sufficient, running multiple instances does not cause any problems like duplicated data, as this is handled by Influx. In case of duplicated insertions of a data point, Influx merges it with the existing point if additional fields are added, or overwrites existing data which is unproblematic as the collector instances should not receive conflicting data anyway. It can be meaningful to run multiple instances of the collector, for example to improve availability so that

data is still collected even if one instance is shut down or fails. Even when handling such cases and restarting failed instances by using health-checks, a restart will lead to some downtime which can be covered by another already running instance. The OGN-Collector was implemented in the pre-project.

HGF-Auth The Auth service provides endpoints to login, which creates a new JWT and to verify such tokens. It was implemented in the pre-project.

HGF-Collector The HGF-Collector provides an endpoint for other tools to push flight data into the system. The endpoint requires authentication and users are only authorized to upload data if they have the "reporter" role. Received data is stored in the tracking data store. The HGF-Collector is mainly used by the Android-App, but also by testing tools like the Sim-Connector. The HGF-Collector was implemented in the pre-project.

HGF-Android The native android application which is used for aircraft based data relay as described in section 4.1. The app was mostly implemented in the pre-project and received minor improvements.

FLARM-Simulator This tool was already used in the pre-project to simulate the hardware available in a glider. By using the tool, the Android app and related components can be tested without the need to operate actual hardware. The simulator is a command line application written in Go and simply replays logged FLARM messages via a TCP endpoint which behaves similar to the access point available in gliders.

Incident Detector The incident collector regularly analyzes the flight data available in the tracking store by using the analysis library. In case of a detected incident, a notification is persisted in the General Purpose Database. It was developed from scratch for this project.

Notification Service The Notification Service monitors the General Purpose Database for newly created notifications and creates and sends the relevant emails via SMTP. It was developed from scratch for this project.

Discovery Service The Discovery Service provides the URL for relevant endpoints, like Auth, Backend and the Collector. It currently is a static file which is created manually for each deployed system. This requires the user to only enter one base URL in the

app instead of the three required service URLs, but does not limit the system to a predefined addressing schema. It was developed from scratch for this project.

HGF-Backend The backend service provides required endpoints both for HGF-Web and HGF-Android. The backend connects to the InfluxDB to load flight data, for any other data the MongoDB is used as data store. The base functionality was implemented in the pre-project, however it was vastly extended and improved to support the new features.

HGF-Web The web application provides the user interface for the system. It is developed using React and multiple third-party react components for the general functionality. Geodata is displayed on a map using Leaflet. Similar to the HGF-Backend, HGF-Web already existed in a basic version but needed significant changes, both improving existing features and adding new functionality, like the notification and auditing user interface and additional information on the flight data map and a general UI styling.

Sim-Connector The Sim-Connector can be used to simulate flights using Microsoft Flight Simulator and pushing the generated flight data into the system. It can be used in multiple ways:

1. Generate FLARM log files, which then can be used with the FLARM simulator
2. Push data directly to the HGF-Collector
3. Simulate a FLARM device (similar to the FLARM Simulator) to connect the Android app.

5.3 Notification System

The notification system is a vital part of the system. It is used to notify relevant users in urgent cases, for example detected incidents.
The notification system is designed for extension. Not only would it be possible to implement additional message channels, it is also possible to add new types of notifications and new sources of notifications quite easily. The currently implemented notification

channel is email, the only notification sources at the moment are HGF-Backend and Incident Detector.

5.3.1 Notification Creation

If a subsystem wants to generate a notification, it is sufficient to insert a respective notification document into the General Purpose Database. The document contains the notification data and some metadata, it is expected to match the data model shown in table 5.1.
Due to the flexibility of the used data model, notifications can include arbitrary data as additional information. In case the notification sender is aware of the specific type of notification it may use the additional data to send more specific notification messages to the user, otherwise it can just ignore the additional data and send a generic message.

5.3.2 Notification Processing

Notifications are processed by the Notification Service using the following algorithm:

Find and mark The application queries the database for documents matching on of the following criteria:

- *State*="created" **and** *Retries* less than 5
- *State*="processing" **and** *LastUpdatedAt* more than 10 minutes ago **and** *Retries* less than 5

This finds both new notifications and notifications stuck in processing. If a notification is in the "processing" state for more than 10 minutes, it is assumed that something failed during processing, and it was newer handled successfully. If a notification was tried to be processed for 5 times and never got handled successfully, it is assumed that the data is corrupt and will not be handled successfully in the future as well, so it is excluded from further processing to avoid unnecessary load on the system. The notification is still visible in the web application and is handled by the defective notification detection.

Name	Datatype	Description
ID	ObjectId	An unique identifier to identify the document
Notification	Object	The document containing the actual notification data contained in the *Data* property, described by the *What* property
Notification.What	string	A string describing the type of notification, e.g. "flightevent"
Notification.Data	Object	Additional data, content depends on the type of the notification
CreatedAt	Date	Time when the notification was initially created
LastUpdatedAt	Date	Time when the notification was last updated, initially equal to *CreatedAt*
State	string	A string describing the state of the notification **created** initial state, waiting for processing **processing** item is currently processed by the notification service **defective** item could not be processed correctly by the notification service **sent** notification was processed and e-mails were sent
AcknowledgedBy	string	The username of a user who has acknowledged the notification, initially unset

Table 5.1: Notification data model

In the same query (*findAndUpdate*), found documents are updated to reflect their new status:

- *State*="processing"
- *LastUpdatedAt*=Now

Now, other instances of the service will not try to handle the respective documents for the next 10 minutes, so duplicate processing is avoided.

Generate and send messages The system now determines the receivers of the notification message depending on the type. This might be system administrators for security related notifications or administrators and aircraft contact personnel for flight related events.
A message containing relevant information is generated, and an e-mail is sent via SMTP to the receivers.
Receivers may or may not be users in the system - e.g. a flight event could issue a notification to a friend of a pilot who is not part of the gliding club but can initiate contact to the club where competent personnel with user accounts can check the flight data.

Finish When the message was sent successfully, the notification's *State* is updated to "sent" or in case of an error, *State* is reset to "created" and *Retries* is incremented.

Defective notifications (state = "created", retries >= 5) are handled similarly, but instead of trying to parse the data a generic message is sent to admins and the state is set to "defective".

5.3.3 Notification Acknowledgment

Administrators are able to "acknowledge" notifications via the web application. This just sets the *AcknowledgedBy* property to their username which is reflected in the web user interface. Functionally, this has no further effect, but it may help other administrators to decide whether a specific notification still needs to be investigated or if somebody else already took care of it.

Name	Datatype	Description
Lat	float64	Latitude in degrees
Lon	float64	Longitude in degrees
Alt	int64	Altitude in meters
Heading	float64	Heading in degrees (optional)
Speed	float64	Speed in meters per second (optional)
Time	time.Time	GPS Time if available from the data source, otherwise time of first entry of the data point in the systems at seconds precision

Table 5.2: Position data structure

5.4 Data Analysis

Analysis of flight data is needed both for incident detection and for visualization of the data. Therefore, a module was implemented which implements the analysis algorithms and can be used in other applications.
The analyzer package provides two interfaces: the *Analyzer* and *Combiner*. Both features operate on the same data structures.

5.4.1 Data Structures

The *Position* objects described in table 5.2 represent a recorded position of an aircraft. Depending on the data source, some properties might not be available and are therefore treated as optional. The *PositionReports* as described in table 5.3 is used to pool the reported position data of a given reporter (e.g. OGN or HGF-Android users).

5.4.2 Combiner

The *Combiner* is used to merge flight data from multiple sources into a single flight path. The interface receives an array of *PositionReports* and returns a single flight path,

Name	Datatype	Description
Reporter	string	An identifier for the reporter / data source
Positions	Position[]	An array of Position representing the recorded flight path

Table 5.3: Position reports data structure

represented as array of *Position*.
Currently, the combination algorithm is very simple but proofed to be sufficient for the available data - a simple windowed average.
Positions are grouped into 5 second windows. For each window, a new, combined position is created by using the arithmetic average of each property of the input positions. The resulting list of position represents the combined flight path.
The algorithm could be fine-tuned by weighting values based on how reliable a data source is and experimenting with the time window size. However, the current data shows that this very simple approach seems to be sufficient and a finer granularity than 5 seconds is not required at the moment as shown in chapter 7.

5.4.3 Analyzer

The *Analyzer* operates on a flight path, represented as an array of *Position* and returns an array of *FlightEvent*s. Additionally, each *Analyzer* implementation also has to provide a title string to describe itself.
At the moment two implementations are available: the *StopAnalyzer* and the *CombiningAnalyzer*. The *StopAnalyzer* algorithm is described in section 4.2
The *CombiningAnalyzer* is a utility for the scenario when additional analyzers are implemented and should be used in combination. The *CombiningAnalyzer* is constructed by providing an array of *Analyzer*s. All provided analyzers are then executed when analyzing flight data and all resulting flight events are returned, so the analyzers can complement each other and support different scenarios. The detected events of all analyzers are collected in a sorted list. Afterwards, the events are filtered for duplicates by filtering events with the same event type in a predefined time window of 60 seconds.

5.4.4 Incident detection

The Incident Detector is executed periodically and analyzes flight data of the last 24 hours. Therefore, it would be theoretically sufficient to run the detection once a day to just detect all events. However, this would lead to quite big delays between the actual occurrence of an event and the generation of a notification. Hence, the incident detection is executed in tighter intervals depending on the deployment, e.g. every five minutes. The detection could also run continuously to achieve quick response times with the drawback of putting steady load on the system.
Incident detection is executed on a per-aircraft basis, so data of other traffic does not influence the analysis of an aircraft's data.
The last 24 hours of data are loaded from the data store, mapped to the required data structures and passed to the analyzer described in the previous sections. The resulting list of detected events is compared to the list of already persisted events. Only for new events a new document is persisted in the general purpose database. Additionally, for all new events a notification is created and added to the general purpose database which can then be handled by the notification system as described in section 5.3.

5.5 Data Inspection

Manual data inspection can be conducted using the web application. While it works best on bigger screens, it is also usable on mobile devices, however the user experience might be degraded, especially in portrait mode. Any functionality in the web application is only available for authenticated users. The JWT is only kept in-memory, no persistence like a cookie is needed. A page refresh loses the JWT and therefore also acts as logout, but due to the implementation of the web application as single page application, normal workflows do not require any page reloads and the user stays logged in when navigating in the application.

5.5.1 Flight Data Inspection

Figure 5.3 shows the visualization of flight data in the web application.
All data is loaded asynchronously from the backend, where also the authorization of the

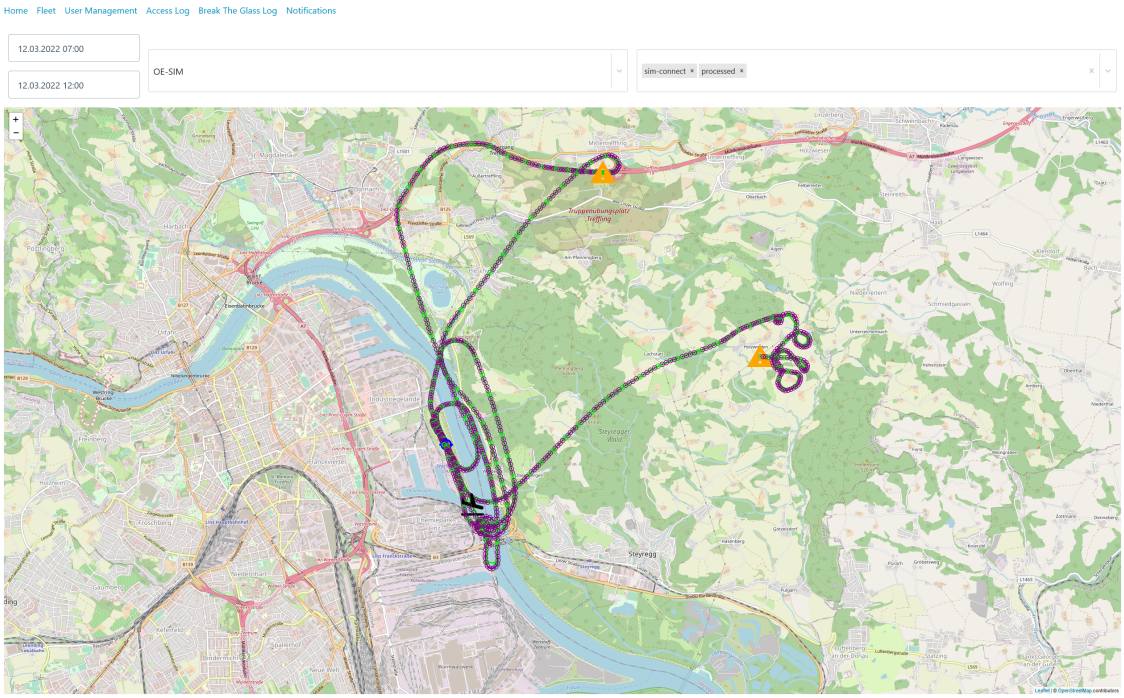


Figure 5.3: Visualization of a simulated flight in the web application

user is checked. The input elements above the map are used to select and filter the data to display. Initially, the date pickers are set to select the last 24 hours, however this can be freely adapted by the user to select any date range of interest. This also populates the next filter element - the aircraft selection. This dropdown menu is only populated with aircraft for which flight data is available in the selected timeframe. After selecting an aircraft, the actual flight data is loaded from the backend.
To select a specific flight, start and end times have to be entered, and the correct aircraft has to be selected. Otherwise, the system does not distinguish between separate flights, only time and aircraft is considered. The rightmost filter control can be used to display data only from selected data sources, like raw data from OGN or only processed data which may be the result of combining multiple data sources as described in section 5.4.2. By default, all sources are drawn on the map which helps to visualize possible differences between them.
The data is drawn as a path on the map, with the data points drawn as points. This gives both a nice visualization of the flight path, but also shows where the data is quite sparse (long lines without points) which can help to identify possible areas of bad tracking coverage. Line and point colors depend on the data source, so from different sources can be quickly identified. Additionally, the map contains icons for known airfields which positions are also loaded from the backend and icons for detected events like landings and incidents which are included when loading the flight data. Flight data is automatically reloaded when changing the filter settings.
Tooltips are used to provide additional information when clicking on a map item like a flight position point or an incident icon as shown in figure 5.4. The event tooltip contains the type of event, in the case of the second example image in figure 5.4 it is of type "incident" and the time it occurred. The flight data point tooltip contains additional information reported with the position. As some data, like heading and speed is optional and reported by all data sources, the tooltip might contain less information if some optional attributes are not set.
The example flight shown in 5.3 was conducted using the flight simulator to be able to safely generate some events. The visible events are a landing on the airfield Linz-Ost, an outlanding on the motorway A7 near Treffling and finally a crash at the eastern part of the flight path. Both the originally reported positions (green) and the processed positions (purple) are visible. The processed positions sometimes differ slightly due to the used averaging algorithm as described in section 5.4.2.

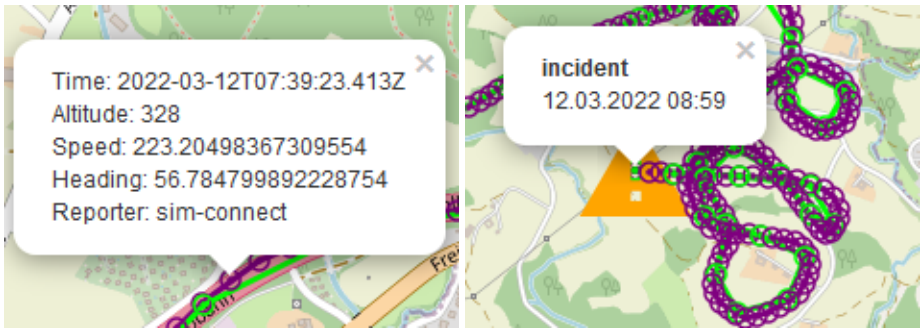


Figure 5.4: Tooltips shown when clicking an element on the map

Accessing Data with restricted Permissions

As described in section 4.3, users might have limited permission to view flight data but can still access data by using the break-the-glass (BTG) functionality. Those users are still able to use the filters and view which aircraft have available flight data in the selected time range. However, when selecting an aircraft in the filter, the BTG dialog informs the user of their options as shown in figure 5.5. Should the user decide to cancel the access at this point, no permanent record is created, and no data is loaded. If the user chooses to get emergency access, a new BTG token is created in the backend and a notification about the action is sent to administrators. The BTG token is used to transparently authorize further requests to the same data. For example, a user might initially choose a longer time period and then wants to focus only on a very specific timespan. This action is immediately allowed without further need to break the glass, as the user already gained access to the requested data anyway, so repeatedly having to break the glass for already seen data would not provide additional security but pose an annoyance to both the user and administrators. This is possible as long as the BTG token is valid, which is one hour or earlier, if an administrator revokes the token. After that, a new BTG token has to be requested, which again will trigger a notification. In case the user wants to view more data and extends the selected time range, a new BTG token needs to be requested as requests with longer time ranges may contain yet unseen data.

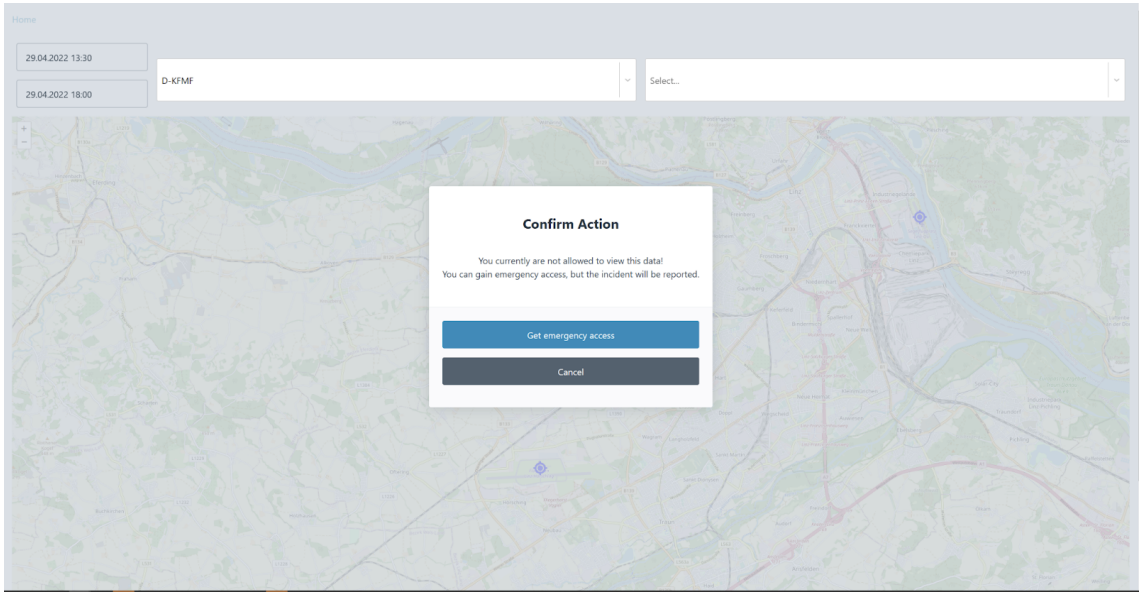


Figure 5.5: The BTG dialog shown when accessing restricted flight data

Backend Implementation

The backend operations required to correctly load the flight data are more complex operations. This is mostly due to the authorization logic required for the flight data access using BTG.
Generally, so-called "Middleware" handlers are used to intercept the requests in the Gin framework and handle authentication and authorization [27]. Gin is a framework for web applications in Go. It was chosen due to its simple API which provides the required functionality to streamline the development of HTTP APIs without being too opinionated. The *AuthHandler* is registered globally, so all requests have to pass its checks. It parses the *Authorization* HTTP header and checks for a valid JWT. If no valid token can be found, the request handling is aborted and an empty response with HTTP status 401 "Unauthorized" is returned. Otherwise, the successfully parsed data from the token is set as context for the request, so further handlers can easily access the session data contained in the token. Usually this is followed by a generic RBAC middleware. This handler checks if the session contains one of the roles required for the requested endpoint, otherwise the request is also aborted with HTTP status 401.

While this simple middleware is easy to use and works fine for features not supporting BTG, it is not sufficient for the flight data endpoints. In this case, no previous RBAC checks are used. Instead, the endpoint handler has to parse the exact request, including aircraft ID and time range and decide based on this data if the requesting user is authorized to view the data. Authorization might be granted based on the user's roles or on an existing BTG token. If the user neither possesses an appropriate role, BTG-token, or has opted to actively break the glass, the request is aborted with status 401. If the user does not possess an appropriate role or BTG-token but has chosen to actively break the glass (i.e. clicked the according dialog button in the frontend as shown in figure 5.5) a new BTG-token is created, and a notification is issued by adding a new notification document to the general purpose database.
After these authorization steps, the actual request processing is delegated to the correct handler and proceeds to load the requested data. The data is loaded from the Tracking Data Store using a Flux-Query. The query string is dynamically computed for the requested aircraft and time range. As this query will contain user supplied data in the query string, it is very important to thoroughly check this data to avoid injection vulnerabilities. The time parameters are verified by converting them to a Go *time.Time* object and then formatting the time to a string using the RFC3339 format [28]. The aircraft ID is checked to only contain alphanumerical characters which is sufficient for this data and will fail any input strings containing key characters of Flux. The returned data is then collected into Go structs as described in section 5.4.1. An audit log entry is created to record that the authenticated user has accessed this data. The data is passed to the analyzer to get events contained in the loaded flight data. Both the events and the position data are contained in the final result which is automatically serialized to JSON when writing the HTTP response.

5.5.2 Auditing Data Inspection

The web application offers multiple pages for administrators to audit data access and system activity as shown in figure 5.6.
The *Access Log* page shows all flight data accesses. If a user requests flight data for an aircraft in a given time range, the request will show up on this page, regardless of the user's roles and permissions. Also, all subsequent requests to the same data is recorded as separate entry. So the page gives an overview over the user's activity and not only which

data was accessed. This can help to identify particularly nosy users who might be abusing their viewer privileges out of personal interest. The access log auditing is designed to be extensible for other access types as well. Currently, only flight data access is audited because it is the only viewer feature non-administrators are allowed to use. However, if an additional role for e.g. fleet management would be introduced, it could be a requirement to audit fleet data access or changes too. In that case, only a new type of access would be added in the backend without the need for adaptions in the database and in the web application, as the current data model is designed and well-prepared for such changes.
The *Break The Glass Log* shows the BTG-tokens which are used and created as described in the previous section. As described in section 4.3.2 administrators can approve and revoke these tokens. Once an administrator has executed an approval or revoke action on a token, the change is final and cannot be overruled by other administrators.
The *Notifications* page gives an overview over the notifications which were created in the system. Administrators can acknowledge a notification, which just serves as information for other administrators that the notification was taken care of. This page is also designed to handle notifications with arbitrary data in yet unknown data structures, where unknown data is just rendered as JSON string.

Figure 5.6: The auditing web interface

5.6 Tools and Simulators

An important part of testing the system was to reduce the dependency on third party components for real world tests. To achieve this, some tools and simulators were developed to provide easily controllable parts for testing to replace the real systems.
An additional need for simulations arises when testing complex and dangerous scenarios like crashes. For obvious reasons, one would not to conduct a test flight which ends in a crash, but especially this important scenario has to be tested. But also for less dangerous scenarios simulations quickly pay off as real world tests are quite time-consuming and expensive. With the developed tools it is possible to conduct a reduced amount of real world testing and use the generated logs and data to repeat the tests whenever necessary.

5.6.1 FLARM Message Collection

The HGF-Android application can be used to collect the FLARM messages received by the device. It logs each received message to a file which can then be accessed via an *DocumentsProvider*. This allows a developer to extract the exact received messages and either analyze them manually or replay them using the FLARM Simulator.

5.6.2 FLARM Simulator

The FLARM Simulator is a simple tool to use in place of a real aircraft with a FLARM device and a TCP endpoint to access the data.
The Go-application opens such a TCP server which can then be connected to with the Android application. It supports two modes of operation: simple data, which just always sends the same hard-coded messages for very simple connectivity tests, or file-replay, which replays the FLARM messages provided via a text file. This text file can be created manually or a log file from the HGF-Android application or the Sim-Connector can be used.
The FLARM Simulator replays the messages with a fixed frequency (e.g. 1Hz), which might influence the timing of the messages. Therefore, the FLARM Simulator is used for testing scenarios where exact timing is not crucial, e.g. testing visualizations or persistence. At the current state of the system, this is only a problem when replaying data which

does not contain speed information, as the timestamps are needed for the average speed calculation and data which contains multiple events in short time spans, as the merge algorithms are time based as described in section 4.2.

5.6.3 Sim-Connector

The Sim-Connector can be used as a FLARM Simulator as well, but instead of replaying log-files it directly relays data received from an active Microsoft Flight Simulator flight. It also supports writing log-files for the FLARM Simulator or pushing data directly to the HGF-Collector as replacement for an Android client.
The C# application uses the *SimConnect SDK* to connect to Microsoft Flight Simulator - and therefore also requires Microsoft Flight Simulator running on the same machine. It uses the *Task Parallel Library* to provide a TCP-Server in parallel to receiving the data from the Flight Simulator but also uses the *Event-based Asynchronous Pattern* for easier integration with the SDK [29, 30]. The Connector task polls every second using the *ReceiveMessage* method. If data is available, the SDK invokes registered receive handler which casts the data to the internally defined structure and passes it to an event. Via this event, the other components of the program can receive the data. A NMEA sentence is generated from the data, written to a log-file, sent to the collector or to a TCP-Client, depending on the configuration.
The simulator currently only generates GPRMC and GPGGA sentences, so it only reports the position of the aircraft but not of any traffic. However, for most tests this is sufficient as only some source of position data is required, and it does not matter if it got tracked as traffic or directly reported its own position.

6 Deployment and Operations

The system was designed to be platform-agnostic to avoid dependencies on any specific infrastructure vendor, except for the mobile application, which is only available on Android. Therefore, build artifacts of backend services are Docker containers, which could be deployed on any host capable of running such containers, container orchestration systems like Kubernetes or managed platforms like Google Cloud Run. To prove this concept, two different environments have been deployed. The first environment, called "Classic Deployment" in this thesis, features a single, inexpensive virtual server running all services and is described in detail in section 6.4. The second environment, called "Cloud Deployment" is based on Platform-as-a-Service (PaaS) solutions provided by Google Cloud Platform (GCP) and Software-as-a-Service (SaaS) solutions provided by the database vendors and is presented in section 6.5. The project's source files were managed using Git on Gitlab.com and using *GitLab CI* for continuous integration (CI) and continuous delivery (CD) [31]. For continuous integration, a build pipeline is executed on every push to verify that the code actually builds, and the unit tests are executed. Additionally, for pushes to the *master* branch the deployment step is executed which pushes the newly built artifacts to the respective environments to achieve continuous delivery. The whole pipeline is shown in figure 6.1 and includes all build and deployment jobs which do not need any manual intervention to deliver the newly pushed code to the *Classic* and *Cloud* systems where the running services are immediately updated.

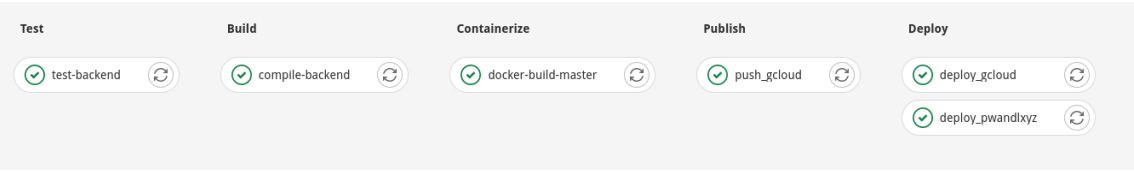


Figure 6.1: The build pipeline which is executed for each push to the *master* branch

6.1 Building

Building the backend services from the source code is quite straightforward thanks to the Go tooling. Only two commands are executed in a *golang* container to fetch dependencies and build the application for amd64 Linux systems:

Listing 6.1: Go Build

```
go get -v -d ./...      # load dependencies as specified in the
   go.mod file
CGO_ENABLED=0 GOOS=linux GOARCH=amd64 go build -ldflags="-w -s" -o
   $CI_PROJECT_DIR/hgf-backend # build a statically linked binary
   for amd64 linux without debug information
```

The frontend build uses *npm* for package management and the npm build tasks are provided by *create-react-app*. The build process does not produce an executable file, only static files which can be served by any web server.

Listing 6.2: Frontend Build

```
npm install
npm run build
```

6.2 Unit testing

Unit tests were implemented to verify functionality of components of backend services. Those tests are especially important for critical components like the analyzer. With the unit tests it can be verified that the current implementation produces the expected results, but they are also helpful to determine whether future changes to the code impact those results and might cause a regression.
The tests are implemented using Go's testing tools, a sample unit test which tests the speed calculation needed for the stop analysis is shown in the following listing. The amount of test cases has been reduced for brevity.

Listing 6.3: Speed calculation unit test

```
package common
```

```
import (
        "testing"
        "time"
)

func Test_calculateSpeed(t *testing.T) {
        type args struct {
                posA Position
                posB Position
        }
        tests := []struct {
                name string
                args args
                want float64
        }{
                {
                        name: "speed",
                        args: args{
                                posA: Position{
                                        Lat: 48.298427340179686,
                                        Lon: 14.335288657343233,
                                        Alt: 250,
                                        Time: time.Date(2021, 10,
                                            17, 15, 54, 14, 98000,
                                            time.Local)
                                },
                                posB: Position{
                                        Lat:  48.29819100688015,
                                        Lon:  14.3354589906351,
                                        Alt:  249,
                                        Time: time.Date(2021, 10,
                                            17, 15, 54, 15, 64000,
                                            time.Local),
                                },
                        },
                        want: 29.177073998403447,
                },
        /* further test cases removed for brevity */
        }
```

```
        for _, tt := range tests {
                t.Run(tt.name, func(t *testing.T) {
                        if got := CalculateSpeed(tt.args.posA,
                           tt.args.posB); got != tt.want {
                                t.Errorf("calculateSpeed() = %v,
                                   want %v", got, tt.want)
                        }
                })
        }
}
```

The unit test tests the *CalculateSpeed* function, which wants two *Position* objects as parameters. Based on this data, the average ground speed is calculated and returned. The return value is compared to the *want* value in the test case. If those values do not match, the test is failed and an error is recorded.
Unit tests are executed before every build by using the *go test* command.

6.2.1 Real world test data

For the analyzer a testing system was introduced which allows using real data from the system to create new test case. With these tests it can be ensured that interesting scenarios which might come up during operation of the system can be added to the test suite easily to verify that those scenarios will continue to work correctly in future versions.
But this can also be used to create tests for scenarios which were not correctly analyzed (e.g. false positive incident detected) and allow the implementation of improvements in the analysis algorithm in a test driven way.
The creation of such test scenarios is quite simple: the users loads and visualizes the relevant data via the web application. The backend response for the requested flight data is saved to a file and the test scenario is specified by adding the new file and the expected results to the *tests* array in the unit test.

6.3 Containerization

All backend services are containerized for deployment using very similar Dockerfiles to create the containers. The following listing shows the Dockerfile of the *Backend*:

Listing 6.4: Backend Dockerfile

```
FROM golang:alpine as build
# Redundant, current golang images already include ca-certificates
RUN apk --no-cache add ca-certificates

FROM scratch
COPY --from=build /etc/ssl/certs/ca-certificates.crt
   /etc/ssl/certs/
COPY hgf-backend hgf-backend
COPY waypoints.cup waypoints.cup
EXPOSE 8080
EXPOSE 2112
ENTRYPOINT ["/hgf-backend"]
```

The Docker build is only executed after the Go-build step has succeeded, so the binary *hgf-backend* is already available as artifact of the previous build and does not need to be built from the source code again. However, a multi-stage build is used anyway to build a small container only containing the necessary elements [32]. In the first stage, the *golang:alpine* image is used and the *ca-certificates* package is installed, if not already contained in the image [33]. This allows the second and final stage to populate an empty image (*scratch*) with the SSL certificates installed in the first stage [34]. Those certificates are needed if the application in the container wants to open a connection using TLS and verify the certificates. Additionally, the binary of the application (in this case *hgf-backend*) and the waypoints file are added to the image. The image exposes port 8080 where the application will provide its functionality and port 2112 where a Prometheus endpoint provides metrics. The entry point of the image is the application binary.
Other Go services are containerized similarly.
The web frontend's Dockerfile differs from the Go services, as the frontend build does not produce an executable application, only static files which can be served by a web server. Therefore, nginx is used via the official nginx base image to build a container image as shown in the following listing [35]:

Listing 6.5: Frontend Dockerfile

```
FROM nginx
COPY build /usr/share/nginx/html
COPY nginx/default.conf /etc/nginx/conf.d/default.conf
```

Only the build outputs and a configuration file are copied to the image. The configuration file is necessary to configure nginx to correctly handle the single page application - for any requested path, the index.html is served as the routing inside the application is handled via JavaScript on the client side.

6.4 Classic Deployment

The classic deployment uses one single host to operate all the necessary services for the system. This host is a "Cloud-Server" hosted by Hetzner running Debian stable [36].
The databases use the official Docker images of the vendors, i.e. MongoDB Community Edition and Influx 2.0 OSS [37, 38].
All containers provide services reachable from outside the host are running behind a nginx reverse proxy which takes care of TLS using Letsencrypt certificates.
Backend services are executed using the Docker images. Systemd is used to manage these services.
A deployment of a Go service consists of two steps. First, the correct container is assigned the "production" tag. Afterwards, an update script is executed via ssh on the target server which updates the service. This requires the CI service to be able to authenticate to the target server. To achieve that, an RSA private key is provided via an environment variable to the CI job. To limit the abuse potential via these keys, they are limited to a single command, and can therefore only trigger the update of a specific service, but it is not possible to execute arbitrary commands on the target server via the CI script or by extracting the key from the CI environment. The executable deployment which is triggered via ssh is not managed by the deployment process and cannot be remotely modified using a deployment key. Only the server administrator is supposed to update these critical scripts.
The aforementioned update scripts trigger a restart of the corresponding systemd service. These services take care that the current instance is shut down, the latest image with the "production" tag is pulled and started.

Pulling from the Docker registry requires a deployment key which is allowed to read, but not write from the registry and is saved on the server.

6.4.1 Deployment Example for HGF-Auth

In this section the necessary configuration files and scripts for a deployment of the authentication service are shown. All other go services are deployed similarly. The following listing shows the deployment job which is configured in the *.gitlab-ci.yml* which deploys to the host *pwandl.xyz*.

Listing 6.6: .gitlab-ci.yml

```
deploy_pwandlxyz:
  image: docker:latest
  stage: deploy
  services:
    - docker:dind
  environment: production
  variables:
    CI_REPOSITORY_URL: git@gitlab.com:$CI_PROJECT_PATH.git
  only:
    - master
  dependencies:
    - docker-build
  before_script:
    - docker login -u "$CI_REGISTRY_USER" -p
       "$CI_REGISTRY_PASSWORD" $CI_REGISTRY
    - echo 'preparing environment'
    - apk add --update openssh-client
    - eval $(ssh-agent -s)
    - mkdir -p ~/.ssh/
    - echo "$PWANDLXYZ_DEPLOY_KEY" | tr -d '\r' | ssh-add - >
       /dev/null
    - touch ~/.ssh/known_hosts
    - echo "$PWANDLXYZ_SSH_KNOWN_HOSTS" > ~/.ssh/known_hosts
    - chmod 644 ~/.ssh/known_hosts
  script:
    - echo 'tagging image for release'
    - docker pull "$CI_REGISTRY_IMAGE:$CI_COMMIT_SHORT_SHA"
```

```
- docker tag "$CI_REGISTRY_IMAGE:$CI_COMMIT_SHORT_SHA"
   "$CI_REGISTRY_IMAGE:production"
- docker push "$CI_REGISTRY_IMAGE:production"
- echo 'prepared image, deploying now'
- ssh hgf@pwandl.xyz /srv/hgf/deploy-auth.sh
- echo 'deployment completed'
```

The deployment job is only executed for builds on the master branch and can only be run after the "docker-build" step has successfully finished.
The deployment job is executed in a "docker" Docker container which provides the tools to manage the docker containers. In the *before_script* block, the system is prepared by using GitLab-CI's environment variables to make sure both docker and ssh commands later can be used without issues. For Docker, it is necessary to login at the GitLab Docker registry. For SSH, the *openssh-client* package is installed, the *known_hosts* file is created with the known host key of the target server, and the private key is read from an environment variable and added to the ssh authentication agent.
The docker image built in a previous step is pulled by the script, tagged as "production", and finally the tag is pushed. Afterwards the deployment script on the server is started via ssh.
The deployment script is very simple, it only triggers a restart of the systemd service:

Listing 6.7: deploy-auth.sh

```
#!/bin/bash

sudo systemctl restart hgf-auth.service
```

The sudo capabilities of the user are very limited and only relevant commands are allowed as shown in the sudoers configuration:

Listing 6.8: /etc/sudoers.d/hgf

```
hgf    ALL=NOPASSWD: /usr/bin/systemctl restart hgf-auth.service
# additional entries redacted for brevity
```

The systemd service is configured as follows:

Listing 6.9: hgf-auth.service

```
[Unit]
Description=HGF auth service
```

```
Requires=docker.service,hgf-mongo.service
After=docker.service
User=hgf
Group=hgf

[Service]
Type=Simple
ExecStartPre=-/usr/bin/docker rm -f hgf-auth
ExecStartPre=/bin/sh -c '/usr/bin/cat /srv/hgf/gitlab.key |
   /usr/bin/docker login --username hgf_production
   --password-stdin registry.gitlab.com && /usr/bin/docker pull
   registry.gitlab.com/h-g-f/hgf-auth:production'
ExecStart=/usr/bin/docker run --name hgf-auth -p
   127.0.0.1:9002:8080 --network hgf -v
   /srv/hgf/auth.properties:/srv/hgf/auth.properties:ro -e
   CONFIG_PROPERTIES=/srv/hgf/auth.properties
   registry.gitlab.com/h-g-f/hgf-auth:production
ExecStop=/usr/bin/docker stop hgf-auth
Restart=always

[Install]
WantedBy=default.target
```

Before starting the service, the old container is removed and the docker daemon logs in to the GitLab registry and pulls the *hgf-auth* image tagged with production.
The start command creates a new container from the previously pulled image. Noteworthy here are some of the passed arguments. The application listens to port 8080 inside the container, which is published to the host port 9002. This is necessary for the reverse proxy to connect to the container, as the reverse proxy is currently running natively on the host due to historical reasons. Additionally, the container is connected to the *hgf* network, which is also used by the database container, so the application and the database can communicate via this docker network. The application configuration is read from property files and only the location of the configuration file is passed as environment variable. Therefore, the configuration file is mapped as read-only volume into the container.
The configuration file specifies some properties and application secrets which need to be adapted to the environment and should not be hard coded or managed with the code:

Listing 6.10: auth.properties

```
auth.key.public=<redacted>
auth.key.private=<redacted>
cors.origin=https://hgf.pwandl.xyz
mongo.user=hgf-admin
mongo.password=<redacted>
mongo.db=hgf
mongo.endpoint:hgf-mongo
```

It provides the keys for the JWTs, defines the origin, so the application can correctly use CORS and configures the connection to the MongoDB.

6.4.2 Monitoring

To provide some insights into the system's behavior, a simple monitoring system has been set up. The main components of the monitoring system are *Prometheus*, a database which stores the application metrics, and *Grafana*, a web application used to visualize the data [39, 40].
The Go applications can be easily monitored employing the Prometheus Golang client. The following listing shows how the Prometheus client package is used to expose metrics via HTTP on port 2112:

Listing 6.11: Prometheus Go Client

```
go func() {
    promMux := http.NewServeMux()
    promMux.Handle("/metrics", promhttp.Handler())
    err := http.ListenAndServe(":2112", promMux)
    if err != nil {
        log.Printf("error serving prom metrics: %v", err)
    }
}()
```

This endpoint is polled by the Prometheus server to collect the data. The data includes technical information like memory usage and garbage collection statistics, but can also contain custom metrics. An example of such a custom metric is timing a performance critical part of the application using a Prometheus histogram and a Timer object.

Figure 6.2: Custom Grafana dashboard

Grafana is able to connect to a variety of data sources, so not only Prometheus data can be visualized, but also data contained in Influx or MongoDB. Existing dashboards from third party sources can be imported to visualize standard data like the aforementioned technical information provided by the Golang client. Custom dashboards can be built to visualize system specific data and provide a nice overview as show in figure 6.2. The dashboard shows some information about the system usage, like the amount of users authenticated in the selected time range or the amount of detected events. It also contains more technical information, like the service health and memory usage of the services. The values show that the tracked traffic and the load on the system was very low in the selected time frame.
Logs are collected using Debian's default logging system and can be easily accessed using *journalctl*.

6.5 Cloud Deployment

The cloud deployment uses Google Cloud Platform (GCP) services and commercial services of the database vendors to run the system [41].
Mongo Atlas is a Software-as-a-Service (SaaS) product offered by MongoDB which is used for the general purpose data store [42]. The InfluxDB is provided similarly via the *Influx Cloud* service by InfluxData [43]. These services allow the usage of those technologies without the need of provisioning servers or any other infrastructure or having to maintain and update these services, as those tasks are fulfilled by the respective vendors.
The application services are deployed in *Cloud Run* services. Cloud Run executes the provided software in the Docker container and exposes an HTTPS endpoint to communicate with the deployed software [44]. To be able to use a custom domain name, *Load Balancing* is used. This allows listening on custom DNS names with managed TLS certificates, redirecting HTTP traffic to HTTPS and as the name suggests, load balancing if multiple instances of a service are running in *Cloud Run* [45].
The *Cloud Run* configurations have been created manually using GCP's web console. Similar to the classic deployment, the actual deployment of the software is automated in the CI pipeline and the current version is automatically deployed for any push to the *master* branch.
First, the Docker image is pushed to a Docker repository in the *Artifact Registry* [46]. Then the deployment of the new version is triggered via *Google Cloud SDK* command line tool [47].

6.5.1 Deployment Example for HGF-Auth

In this section the necessary configuration files and scripts for a deployment of the authentication service are shown. All other go services are deployed similarly.
The initial creation of the Cloud Run service via the web console is quite straightforward, some parameters like name of the service, network configuration and capacity settings (e.g. memory, vCPUs, min/max number of instances) have to be specified. The application specific configuration is provided via environment variables (in contrast to the classic deployment, where configuration files are used). Those variables can be also defined in the web console and can also reference secrets, like database authentication keys, managed in *Secret Manager* as shown in figure 6.3. The *Load Balancing* is configured via the web

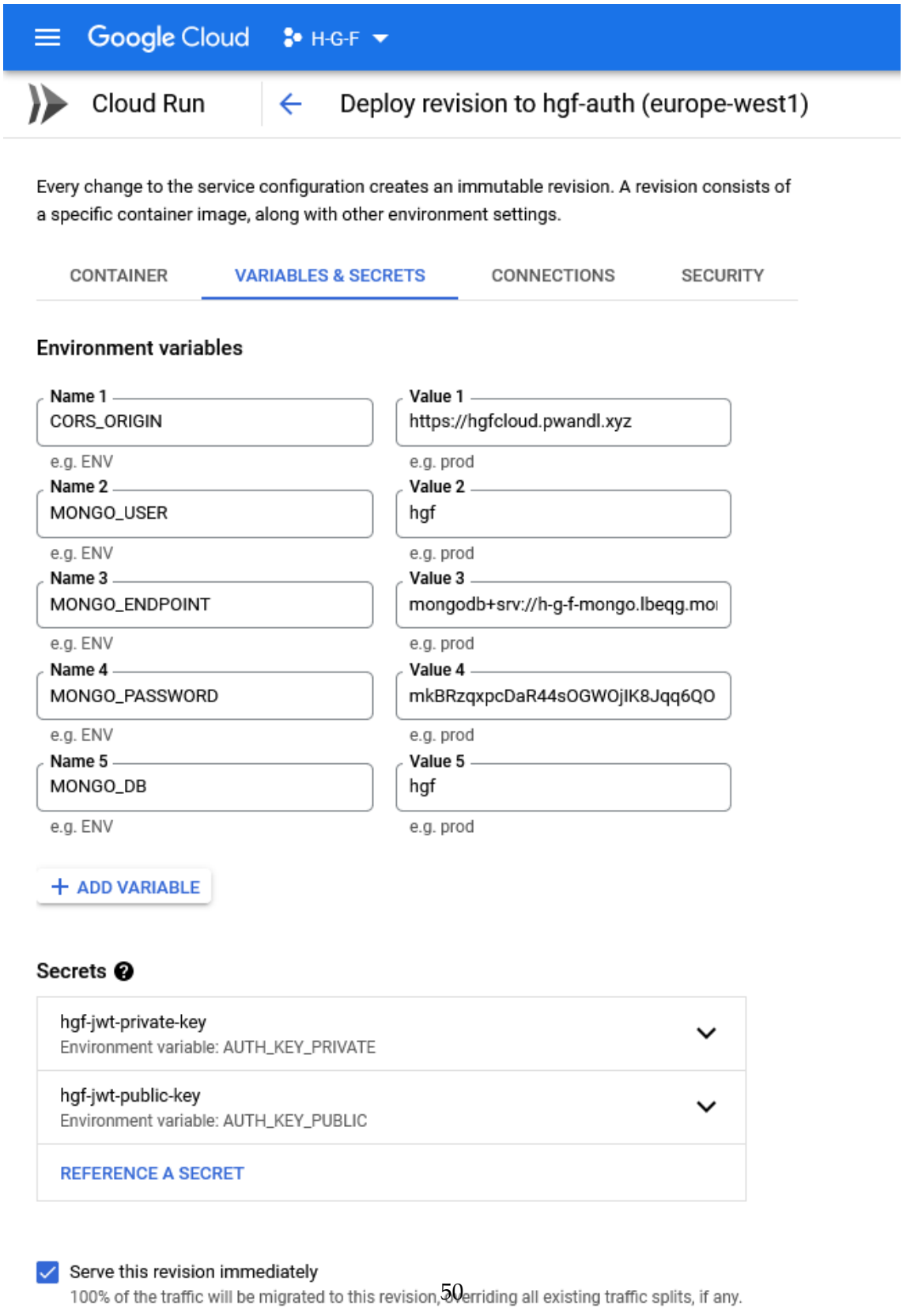


Figure 6.3: GCP Cloud Run Variables & Secrets configuration

console as well, specifying the HTTP to HTTPS redirect rule, the connection to the *Cloud Run* and the DNS name and platform managed TLS certificate. Future deployments might rely on *Cloud Run Domain mappings* instead, however those are only in preview yet, so the *Load Balancing* was needed to be able to use custom DNS names for the services.

The following listing shows the two jobs needed for the deployment to GCP as defined in *.gitlab-ci.yml*.

Listing 6.12: .gitlab-ci.yml

```
push_gcloud:
    image: docker:latest
    stage: publish
    services:
      - docker:dind
    environment: gcloud
    variables:
      CI_REPOSITORY_URL: git@gitlab.com:$CI_PROJECT_PATH.git
    only:
      - master
    dependencies:
      - docker-build
    before_script:
      - docker login -u "$CI_REGISTRY_USER" -p
          "$CI_REGISTRY_PASSWORD" $CI_REGISTRY
      - echo $GCLOUD_KEY_BASE64 | docker login -u _json_key_base64
          --password-stdin https://europe-west1-docker.pkg.dev
    script:
      - echo 'tagging image for gcloud release'
      - docker pull "$CI_REGISTRY_IMAGE:$CI_COMMIT_SHORT_SHA"
      - docker tag "$CI_REGISTRY_IMAGE:$CI_COMMIT_SHORT_SHA"
          "europe-west1-docker.pkg.dev/h-g-f-344207/hgf/hgf-auth:production"
      - docker push
          "europe-west1-docker.pkg.dev/h-g-f-344207/hgf/hgf-auth:production"
      - echo 'pushed image to gcloud'

deploy_gcloud:
    stage: deploy
    only:
```

```
    - master
  image: google/cloud-sdk:latest
  environment: gcloud
  dependencies:
    - push_gcloud
  script:
    - echo $GCLOUD_KEY_BASE64 | base64 --decode | gcloud auth
        activate-service-account --key-file=-
    - gcloud run deploy "hgf-auth" --image
        "europe-west1-docker.pkg.dev/h-g-f-344207/hgf/hgf-auth:production"
         --project h-g-f-344207 --region europe-west1
```

The necessary key used for authentication to GCP is provided base64 encoded as variable *GCLOUD_KEY_BASE64* to the build job. It is used for the *docker login* command and for the *gcloud auth* command.
The *push_gcloud* job simply tags the image with the correct name and "production" tag to push it to the project's GCP repository where it will be available for services like *Cloud Run*. The *deploy_gcloud* job triggers a deployment for the pushed production tag. Only then the *Cloud Run* service pulls the updated container for the selected tag and deploys the new versions.

6.5.2 Monitoring

For the cloud deployment, no custom monitoring system was set up. Instead, GCP provides enough tools to fulfill the basic monitoring needs, which is shown in figure 6.4. Logs are available via the *Log Explorer* which also allows creating alerts, e.g. for all logs with *error* log level or for a specific log message [48].

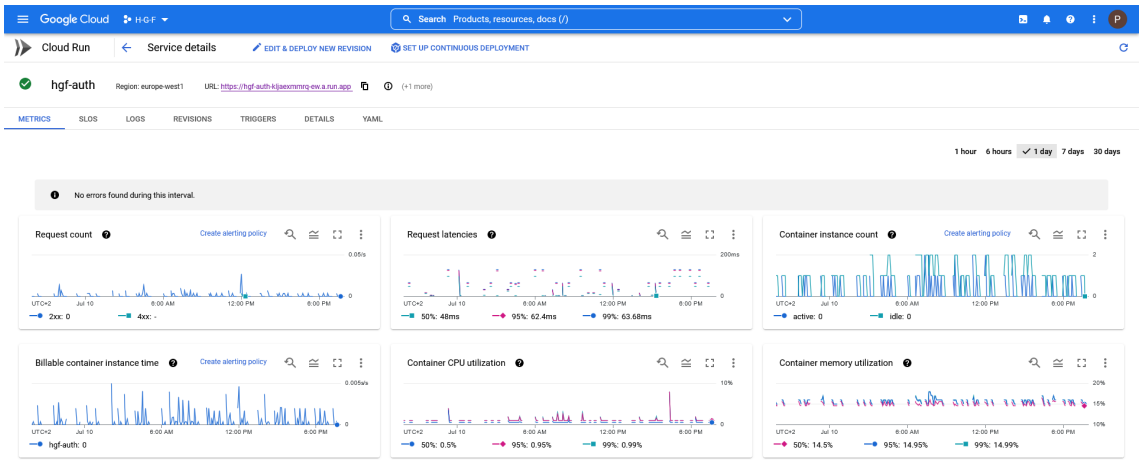


Figure 6.4: GCP Cloud Run Metrics

6.6 Deployment Comparisons

While both of the deployments fulfill the requirements to run the system, the differences are quite big in some aspects. Table 6.6 provides an overview for important aspects of the system. The added flexibility and comfort of the cloud system comes with a significantly bigger price tag, but also significantly more possibilities, e.g. redundant deployments over multiple geographic regions with little effort. The actual pricing of the cloud system depends on usage of the system, more load leads to higher cost, while the classic deployment offers more predictable pricing.

	Classic	Cloud
Managing Services	systemd unit	Google Cloud Run
Horizontal Scaling	no, exactly 1 instance	yes, 0+ instances
Load Balancing	No load balancing possible	Cloud Load Balancing
HTTPS	letsencrypt cert	Managed SSL certs
General purpose data store	MongoDB Community Server	MongoDB Atlas
Time series data store	InfluxDB OSS	InfluxDB Cloud
Monitoring	Prometheus + Grafana	Google Cloud tools
Pricing	ca. €8	ca. €100

Table 6.1: Comparison of Deployments

6.7 Data Privacy

The system aims to only collect as much data as necessary.
Flight data collection is strictly limited to whitelisted aircraft; all other data is ignored as soon as feasible. No data of aircraft which has not opted in to tracking will be stored in the system. Flights are not connected to pilots, however with access to additional information like start lists it might be possible to achieve that connection outside the system.
The flight data is currently only collected from publically available sources, i.e. the *OpenGliderNetwork* or the broadcasted FLARM messages.
Opt-in is trivial for privately owned gliders, where the pilot/owner can choose to accept the tracking for themselves. With club aircraft, multiple pilots may use the same aircraft, who would need to consent to the tracking. However, clubs can introduce additional usage limitations to their aircraft (like experience limits, additionally required training, ...) which may include accepting the tracking of the used aircraft.
The position data stored in Influx is automatically deleted by Influx after it reaches a configured age. Currently, the configured age is one year. For a productive use without the goal of developing system improvements based on the data, a much lower retention time could be achieved, like 14 days after which a missing aircraft should be at least identified to be missing, so any needed data can be extracted. Data used for development does not need to be connected to a real aircraft, so pseudonymization is possible. In contrast to e.g. tracking of cars or e-scooters, which might be used to travel directly to one's home or office, it is harder to identify pilots based on their flight path as the start and endpoints are usually limited to airfields used by lots of other pilots. The flight path itself might identify

a pilot, but only in very specific cases, like a pilot known for repeatedly flying a specific route with unique waypoints. Usually, it would not be possible to identify a specific pilot by the flown route, especially in soaring flight, as those routes are heavily influenced by external circumstances like meteorological conditions and traffic.
Users do not need to provide any personal information to use the system, except a unique username and a password (which is stored as a hash). E-Mail addresses can be optionally entered if the user wishes to receive notifications. User behavior is recorded by the auditing functionality to detect possible privacy infringements when a user abuses the system.
As described in 4.3, none of the data is made publically available, and it is taken care that only authorized users are allowed to access the data they need to access.

7 Tests

Testing has been done using both artificial test data and real world flight data. The simulation tools, described in section 5.6, played an important role during testing. It allowed testing changes to the system immediately without having to conduct a test flight based on real world data. Test flights would of course be the most realistic tests, but they are quite time-consuming and expensive. Additionally, some scenarios are not testable in a test flight, e.g. crashing an aircraft just to test the incident detection is not advisable. It is however possible to test the outlanding detection by providing manipulated airfield data, so that the system is not aware of an airfield being located at the position of the landing and therefore detecting an incident as shown in figure 7.1. Logged data from test flights at least allowed replaying scenarios with real world data without having to repeat the actual flight.
Unit testing was used to cover important algorithmic parts like the analysis components. Tests which involve more than one component were conducted manually as the effort of creating automated end-to-end tests was estimated too great to be worth it at the moment.

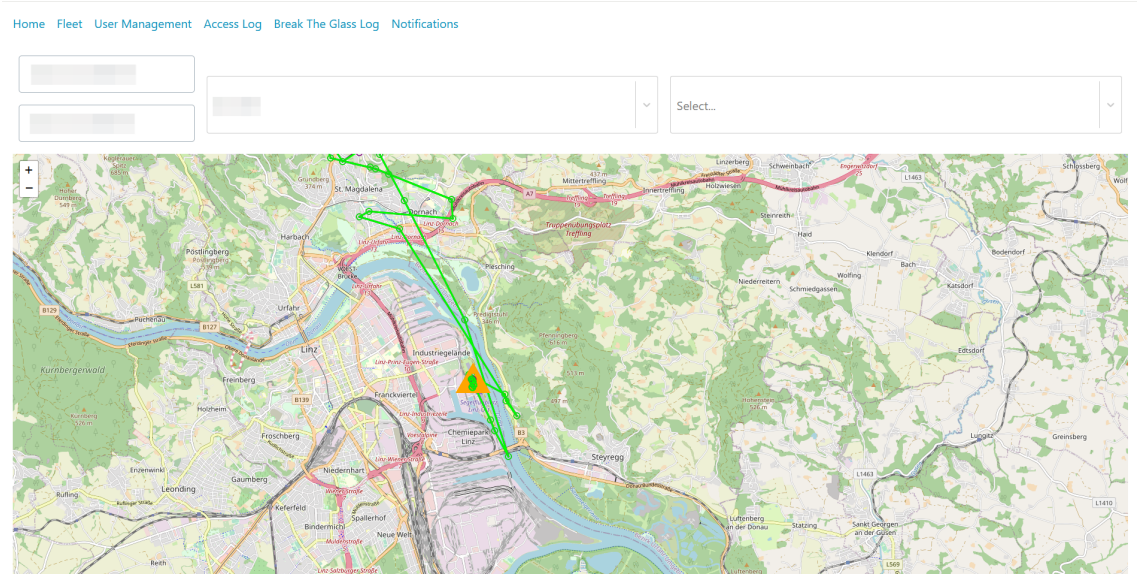


Figure 7.1: "True" positive incident with manipulated airfield data shown in the web application

7.1 Data Verification

Data from the different connected data sources has been checked to be of sufficient quality. While some limitations have been detected during testing as described in the following sections, all data sources provided quite reliable data and can benefit the system. Verification of data was mostly done manually with the usage of the developed tools and third party tools to provide a comparison.

7.1.1 OGN Data

To verify data ingested from the OGN network is of usable quality, third party OGN clients have been used to compare the collected data to the data shown in those third party applications. The used third-party clients were web applications (`https://live.glidernet.org`, `https://glidertracker.de`) and a mobile application (`https://play.google.com/store/apps/details?id=com.meisterschueler.ognviewer`). The data available in the HGF system, including the visualization in the web application was compared to the data available via those third party applications. Even when the data itself seems to match,

visualization differences exist due to different data processing and visualization, e.g. the *Glidertracker* shows no markers for the data points and seems to interpolate the data for a smoother and visually more appealing presentation while the HGF web application draws straight lines and markers for each data point.
Possible latencies introduced by the OGN network are not problematic, as the messages contain a timestamp (with milliseconds resolution) when the FLARM message was created.
One issue with the OGN data was found when comparing different clients: not all instances seem to receive all data points. So while the data points which are received on all instances contain equal data, it seems that not all data points are sent to all instances. This behavior can be observed even when comparing multiple instances of the same application, e.g. running the *OGN Viewer* app on multiple phones or comparing the different deployments of the HGF system.
The achieved sampling rate (up to 0.1 Hz) is sufficient for the stop analyzer to be able to detect events, but the limitation has to be kept in mind for future analysis algorithms, which should not rely on uninterrupted data streams and higher sampling rates, if only OGN data is expected to be available.
Comparing OGN data to other data sources, like the collected tracks of the navigation software (LK8000) or visible traffic around an airfield showed no significant errors in the data.

7.1.2 Relayed Data

Data relayed by the Android app was compared to other available data, like the aforementioned OGN data and verified similarly using the visualization provided by the web application. An aero tow was used as a real life test scenario to collect data. In this test, the tow plane used the Android application to ingest data, both the tow plane and the glider were equipped with FLARM. As the flown routes and speeds are very well known in this scenario, data could be verified to be plausible even without perfect OGN coverage in lower altitudes. OGN data and data from the Android application does not match exactly, but the differences of up to a few meters in location data are minor and not significant for the currently used analysis algorithms. This difference can be explained by the more complex calculation of the position of the tracked aircraft when using the Android application. Via OGN, the position data is received via the FLARM message and used without further

processing. Via the Android application, it has to be reconstructed from multiple NMEA messages due to the limitations of the FLARM data port specification. As only the relative position of the tracked aircraft is received, the absolute position needs to be calculated by using the last known position of the tracking aircraft. This introduces multiple sources for errors. The last known position of the tracking aircraft might be slightly outdated, the position data itself can contain some inaccuracy as it is based on GPS data and all calculations use floating point numbers which might also introduce minor imprecision.
The timestamp of the data is based on the smartphone's system time as the traffic sentences do not contain a timestamp. As modern smartphones usually have a quite accurate time setting, no problems were detected with the timing of the data.

7.2 Processing Verification

The initial testing of the implemented flight data processing was based on data from flight simulator flights. This provided great flexibility with possible scenarios and generated useful data quickly. Additionally, no environmental factors like the coverage of a tracking system were a concern.
The algorithms were also applied to real world data available in the system and false results were used to create unit tests and improve the algorithms even further. In addition to unit tests, manually testing new versions of processing algorithms is very straightforward using the web application, as the backend always applies the latest detection algorithms to the data when loading flight data for visualization, so the new results are immediately available also for old data.
Testing showed a perfect detection of crashes and landings for simulated flights. Every crash, outlanding and landing on an airfield in simulated flights was correctly detected, and neither false positives nor false negatives could be provoked.
For real world flights, the results depend mostly on the data quality. In areas with good tracking coverage, the landing detection works as perfect as in the simulations. Bad coverage and therefore incomplete data inevitably lead to false negatives as the current algorithms are not designed to handle such cases. Detection of a real incident was luckily not possible as no relevant incident occurred during the time of testing the system. A few false positives were noticed, some of which could be explained by aerobatic flight maneuvers. The other false positives seem to have occurred in scenarios where the

Figure 7.2: False positive incident shown in the web application

coverage and therefore the sampling rate was not optimal and additionally where the ground speed did not match the airspeed very well, possibly due to headwinds and/or flight maneuvers like pulling up into a thermal and starting circling.
An example of a false positive is shown in figure 7.2. In this case, the speed data was collected via OGN and was part of the message, so no calculation based on the location data was necessary. As the equipped FLARM in this glider does not have a pitot sensor, it can only report speed based on GPS data and therefore only ground speed, not air speed. This can explain why the aircraft was still flying at reported speeds (37 km/h, 40 km/h, 50 km/h) way below the stall speed (ca. 60 km/h) [49]. The reason for the low reported ground speed cannot be determined with certainty, but probable causes are winds and the glider changing directions quite quickly.
The results might be improved further with more fine-tuning of the preset parameters, like the speed threshold or the averaging window. Such improvements might be possible after operating the system for a while and gathering more data and might also depend on additional data sources. For example, a narrower averaging window will currently not bring any improvements, as the actually achieved sampling rate of the data is not significantly higher than after the windowed sampling. If an additional data source with higher sampling rate would be connected to the system, it could be necessary to reevaluate such parameters.

8 Future Work

While the system can already be practically used and fulfills the requirements, there are still several possible improvements.

8.1 Additional Data Sources

As the quality of the visualized data and incident detection depends highly on the collected data, integrating more data sources could bring significant benefits. The system itself can be easily extended with additional collectors, so no big adaptions would have to be implemented. Some data providers only provide paid licensing for API access and their data sources. A big benefit for the system would be a collection of ADS-B data to also track aircraft equipped broadcasting their location via ADS-B. Some gliders already feature *ADS-B out* capable transponders, so only a new collector which gathers data from a service providing ADS-B tracking would be need to track these aircraft.

8.2 Improved Incident Detection Algorithms

The data analysis algorithms described in section 4.2 only support very specific and limited scenarios. More advanced algorithms might be capable of detecting more incidents, such as crashes where the avionics are destroyed and no further data is broadcast by the aircraft. This might include implementing heuristics which can detect a possible incident in case of lost tracking connection or some anomaly detection based on modern machine learning techniques.
Every development in this directing potentially leads to significant testing efforts and might even require more sophisticated testing tools and simulators.

8.3 Improved Testing Tools

While the current incident detection algorithm can be tested easily with the implemented tools, future development might be limited by the test data sources.
As the *SimConnector* provides perfect data quality, it is not quite representative for real world scenarios. It could be extended to add some noise to the data to mimic GPS errors and to drop some data to simulate imperfect tracking coverage.
The *FLARM Simulator* might be extended to support multiple logs and merge and replay them with accurate timing to be able to simulate multiple aircraft at the same time. This could be especially important when implementing heuristic detections, e.g. if one aircraft is visible at the expected position of another tracked aircraft which lost connection, it might indicate an incident.
Generally, end-to-end testing for all system components would help to ensure correct functionality of all parts working together without having to manually test the system after implementing or changing features, but is only worth the effort if the system will be actively developed and operated for a longer time period.

8.4 Acceptance Testing

The usability of the user interfaces is not yet verified by testing with a diverse audience. Acceptance tests with users of the target group might deliver some insights into how a typical user would use the system and where the user interface might need improvements.

8.5 Connection to Start List

As lots of clubs use electronic start lists, it would be possible to connect this system to these start lists and therefore be able to link aircraft and pilots. However, this is currently not implemented due to both data privacy concerns and the low cost-benefit factor of such a feature.

8.6 Supporting Multiple Clubs

Currently, the system is designed with global roles like Administrators and Viewers and does not provide multi-tenant support. In the future, it might be needed to use the systems in multiple clubs without sharing data and permissions. This could be achieved by just deploying another instance of all components so that each club hosts their own instance of the system. While this would not require any code changes, it would need an improved deployment process for infrastructure components to save time and avoid errors. This could be achieved by automating the infrastructure deployment using Infrastructure-as-Code (IaC) technologies like Terraform [50].
Another possible solution would be to implement multi-tenant capabilities and offer the systems as Software-as-a-Service product. While this would come with significant development effort, it would possibly keep recurring infrastructure costs significantly lower.

Bibliography

[1] *About OGN*. Apr. 3, 2019. URL: http://wiki.glidernet.org/about (visited on 11/11/2021) (cit. on p. 2).

[2] FLARM Technology Ltd. *FLARM as an additional tool when searching a missing aircraft*. 2021. URL: https://flarm.com/wp-content/uploads/2015/02/SAR_Text.pdf (visited on 11/11/2021) (cit. on pp. 3, 5).

[3] Austro Control. *Surveillance*. 2019. URL: https://www.austrocontrol.at/en/atm/ats_engineering/radarsurveillance (visited on 06/05/2022) (cit. on p. 4).

[4] Austro Control. *AIP AUSTRIA GEN 1.5-1*. June 2021. URL: https://eaip.austrocontrol.at/ (visited on 06/05/2022) (cit. on p. 4).

[5] K. Werner, J. Bredemeyer, and T. Delovski. "ADS-B over satellite: Global air traffic surveillance from space". In: *2014 Tyrrhenian International Workshop on Digital Communications - Enhanced Surveillance of Aircraft and Vehicles (TIWDC/ESAV)*. 2014, pp. 47–52. DOI: 10.1109/TIWDC-ESAV.2014.6945446 (cit. on p. 4).

[6] Ajit Jesudoss. "An analysis of the effectiveness of emergency locator transmitters to reduce response time and locate wreckage in US general aviation accidents". In: (2011) (cit. on p. 5).

[7] Paolo Ventafridda and Ulli Heynen. *LK8000 Tactical Flight Computer User Manual update*. LK8000 Developers Team. Feb. 2013 (cit. on p. 9).

[8] FLARM Technology Ltd. *DATA PORT INTERFACE CONTROL DOCUMENT (ICD)*. May 10, 2021. URL: https://flarm.com/wp-content/uploads/man/FTD-012-Data-Port-Interface-Control-Document-ICD.pdf (visited on 11/18/2021) (cit. on p. 10).

[9] Austro Control. *Airspace Structure Austria*. 2022. URL: https://www.austrocontrol.at/en/pilots/pre-flight_preparation/aim_products/airspace_structure (visited on 08/03/2022) (cit. on p. 12).

[10] David Ferraiolo, Janet Cugini, D Richard Kuhn, et al. "Role-based access control (RBAC): Features and motivations". In: *Proceedings of 11th annual computer security application conference*. 1995, pp. 241–48 (cit. on p. 13).

[11] Runnan Zhang et al. "Improved Bell–LaPadula Model With Break the Glass Mechanism". In: *IEEE Transactions on Reliability* 70.3 (2021), pp. 1232–1241. DOI: 10.1109/TR.2020.3046768 (cit. on p. 14).

[12] Davide Taibi, Valentina Lenarduzzi, and Claus Pahl. "Architectural patterns for microservices: a systematic mapping study". In: *CLOSER 2018: Proceedings of the 8th International Conference on Cloud Computing and Services Science; Funchal, Madeira, Portugal, 19-21 March 2018*. SciTePress. 2018 (cit. on p. 15).

[13] *The Go Programming Language*. 2022. URL: https://go.dev (visited on 08/03/2022) (cit. on p. 17).

[14] Meta Platforms Inc. *React - A JavaScript library for building user interfaces*. 2022. URL: https://reactjs.org/ (visited on 08/03/2022) (cit. on p. 17).

[15] Paul Le Cam and contributors. *React Leaflet*. 2022. URL: https://react-leaflet.js.org/ (visited on 08/03/2022) (cit. on p. 17).

[16] Vladimir Agafonkin. *Leaflet - a JavaScript library for interactive maps*. 2022. URL: https://leafletjs.com/ (visited on 08/03/2022) (cit. on p. 17).

[17] OpenStreetMap Foundation. *OpenStreetMap*. 2022. URL: https://www.openstreetmap.org/about (visited on 08/03/2022) (cit. on p. 17).

[18] Lucas Larroche. *Pico.css • Minimal CSS Framework for semantic HTML*. 2022. URL: https://picocss.com/ (visited on 08/03/2022) (cit. on p. 17).

[19] Microsoft. *SimConnect SDK*. 2021. URL: https://docs.flightsimulator.com/html/Programming_Tools/SimConnect/SimConnect_SDK.htm (visited on 08/03/2022) (cit. on p. 17).

[20] auth0. *JSON Web Tokens - jwt.io*. 2022. URL: https://jwt.io/ (visited on 08/03/2022) (cit. on p. 18).

[21] Xiuyu He and Xudong Yang. "Authentication and authorization of end user in microservice architecture". In: *Journal of Physics: Conference Series*. Vol. 910. 1. IOP Publishing. 2017, p. 012060 (cit. on p. 18).

[22] MongoDB Inc. *Schemaless Database | MongoDB*. 2022. URL: https://www.mongodb.com/unstructured-data/schemaless (visited on 08/03/2022) (cit. on p. 20).

[23] InfluxData Inc. *InfluxDB: Open Source Time Series Database | InfluxData*. 2022. URL: https://www.influxdata.com/products/influxdb-overview/ (visited on 08/03/2022) (cit. on p. 20).

[24] Mohammad Nasar and Mohammad Abu Kausar. "Suitability of influxdb database for iot applications". In: *International Journal of Innovative Technology and Exploring Engineering* 8.10 (2019), pp. 1850–1857 (cit. on p. 20).

[25] APRS Software and Bob Bruninga. *APRS-IS*. 2022. URL: http://www.aprs-is.net/ (visited on 07/29/2022) (cit. on p. 20).

[26] *Subscribing to OGN Data*. Apr. 3, 2019. URL: http://wiki.glidernet.org/wiki:subscribe-to-ogn-data (visited on 08/03/2022) (cit. on p. 20).

[27] *Using middleware*. Apr. 29, 2022. URL: https://gin-gonic.com/docs/examples/using-middleware/ (visited on 07/12/2022) (cit. on p. 32).

[28] Chris Newman and Graham Klyne. *Date and Time on the Internet: Timestamps*. RFC 3339. July 2002. DOI: 10.17487/RFC3339. URL: https://www.rfc-editor.org/info/rfc3339 (cit. on p. 33).

[29] *Task Parallel Library (TPL)*. 2021. URL: https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/task-parallel-library-tpl (visited on 08/03/2022) (cit. on p. 37).

[30] *Event-based Asynchronous Pattern Overview*. 2021. URL: https://docs.microsoft.com/en-us/dotnet/standard/asynchronous-programming-patterns/event-based-asynchronous-pattern-overview (visited on 08/03/2022) (cit. on p. 37).

[31] *GitLab CI/CD*. 2022. URL: https://docs.gitlab.com/ee/ci/ (visited on 08/03/2022) (cit. on p. 38).

[32] Docker Inc. *Use multi-stage builds*. 2022. URL: https://docs.docker.com/develop/develop-images/multistage-build/ (visited on 08/03/2022) (cit. on p. 42).

[33] Docker Inc. *Golang - Official Image | Docker Hub*. 2022. URL: https://hub.docker.com/_/golang (visited on 08/03/2022) (cit. on p. 42).

[34] Docker Inc. *scratch - Official Image | Docker Hub*. 2022. URL: https://hub.docker.com/_/scratch (visited on 08/03/2022) (cit. on p. 42).

[35] Docker Inc. *Nginx - Official Image | Docker Hub*. 2022. URL: https://hub.docker.com/_/nginx (visited on 08/03/2022) (cit. on p. 42).

[36] Hetzner Online GmbH. *Günstiges Cloud Hosting - Hetzner Online GmbH*. 2022. URL: https://www.hetzner.com/de/cloud (visited on 08/03/2022) (cit. on p. 43).

[37] Docker Inc. *Mongo - Official Image | Docker Hub*. 2022. URL: https://hub.docker.com/_/mongo (visited on 08/03/2022) (cit. on p. 43).

[38] Docker Inc. *Influxdb - Official Image | Docker Hub*. 2022. URL: https://hub.docker.com/_/influxdb (visited on 08/03/2022) (cit. on p. 43).

[39] Prometheus Authors. *Prometheus - Monitoring system & time series database*. 2022. URL: https://prometheus.io/ (visited on 08/03/2022) (cit. on p. 47).

[40] Grafana Labs. *Grafana: The open observability platform | Grafana Labs*. 2022. URL: https://grafana.com/ (visited on 08/03/2022) (cit. on p. 47).

[41] *Cloud Computing Services | Google Cloud*. 2022. URL: https://cloud.google.com/ (visited on 08/03/2022) (cit. on p. 49).

[42] MongoDB Inc. *MongoDB Atlas Database | Multi-Cloud Database Service | MongoDB*. 2022. URL: https://www.mongodb.com/atlas/database (visited on 07/29/2022) (cit. on p. 49).

[43] InfluxData Inc. *InfluxDB Cloud*. 2022. URL: https://www.influxdata.com/products/influxdb-cloud/ (visited on 07/29/2022) (cit. on p. 49).

[44] *What is Cloud Run | Cloud Run Documentation | Google Cloud*. 2022. URL: https://cloud.google.com/run/docs/overview/what-is-cloud-run (visited on 08/03/2022) (cit. on p. 49).

[45] *External HTTP(S) Load Balancing Overview | Google Cloud*. 2022. URL: https://cloud.google.com/load-balancing/docs/https (visited on 08/03/2022) (cit. on p. 49).

[46] *Overview of Artifact Registry | Artifact Registry Documentation | Google Cloud*. 2022. URL: https://cloud.google.com/artifact-registry/docs/overview (visited on 08/03/2022) (cit. on p. 49).

[47] *Command Line Interface Gcloud Cli | Google Cloud*. 2022. URL: https://cloud.google.com/cli (visited on 08/03/2022) (cit. on p. 49).

[48] *Configure log-based alerts | Cloud Logging | Google Cloud*. 2022. URL: https://cloud.google.com/logging/docs/alerting/log-based-alerts (visited on 08/03/2022) (cit. on p. 52).

[49] B. Grob Flugzeugbau. *Astir CS Jeans Flug- und Betriebshandbuch*. 1977 (cit. on p. 60).

[50] *Infrastructure as Code*. 2022. URL: https://www.terraform.io/use-cases/infrastructure-as-code (visited on 08/03/2022) (cit. on p. 63).