# GeoPath **Planner**

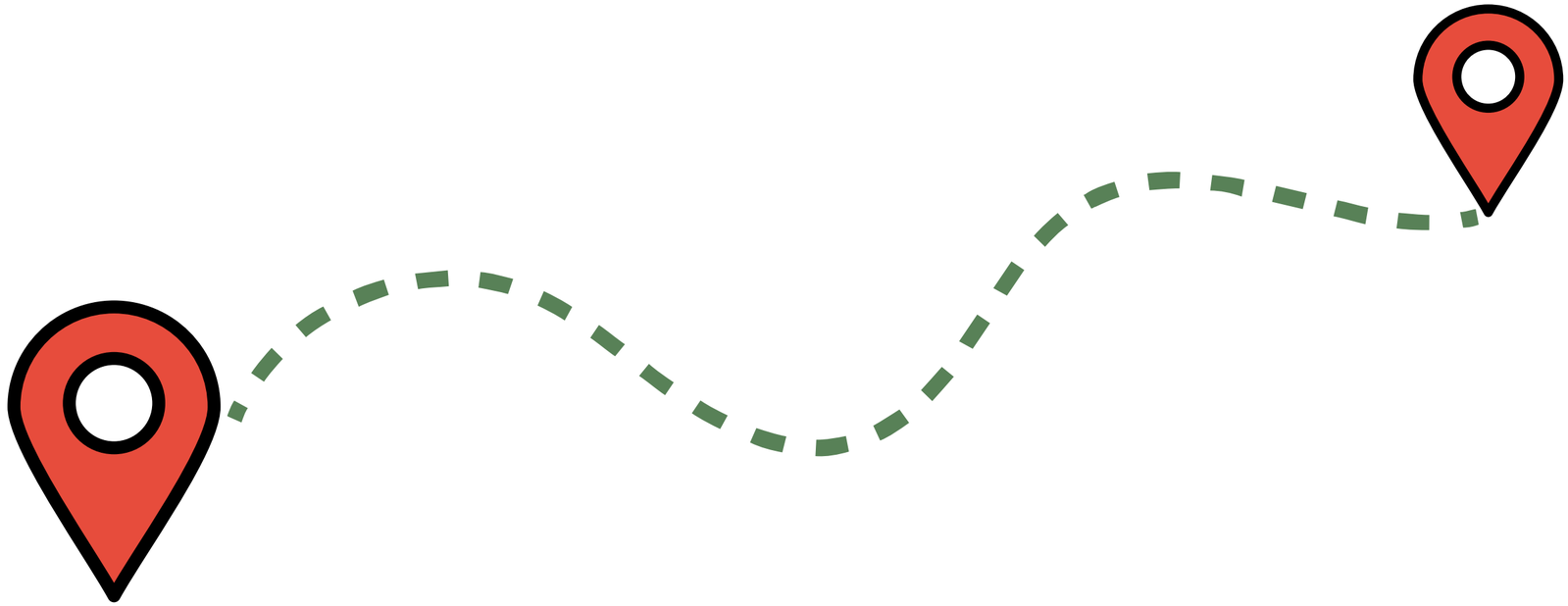
**PRESENTED BY** - Antonio Franzoso, Paolo Liberti, Umberto Di Tommaso, Alessandro Colantuoni

GeoPath Planner

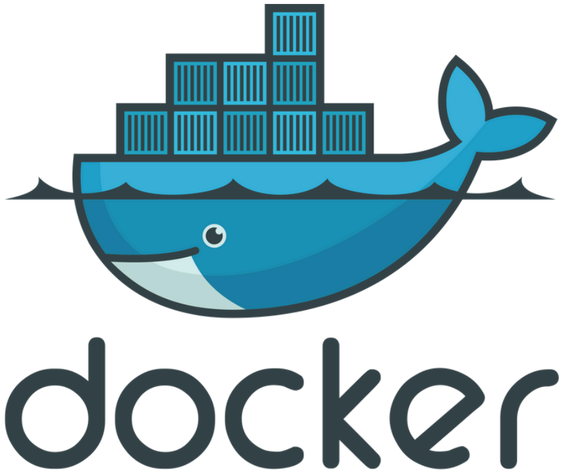
# Welcome To Our Application

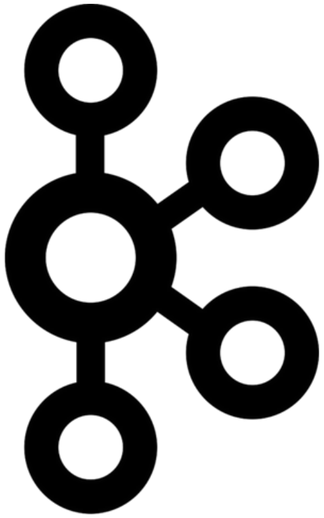
## About Our Application

GeoPathPlanner is a distributed web application used to find a feasible path between a pair of geographical locations for an aerial or sea vehicle avoiding 3D obstacles.
This will be achieved implementing path planning algorithms such that RRT, RRT* and AntPath.
The application is designed to be composed of a series of microservices which interact between them in such a way that all components function and provide an excellent user experience.



# Technologies




# Microservices Overview

1

## Frontend

- Interactive map-based web interface.
- Let the user draw waypoints, obstacles and set routing parameters.
- Visualizes computed routes and results.
- Provide access to personal area and routes history.
- Interacts with API and User Management.

2

## API

- Gateway between frontend and backend.
- Validates and forwards user requests.
- Orchestrates communication among services.
- Manages database of previously computed routes.
- Kafka Producer/Consumer interaction with routing service.

3

## Routing

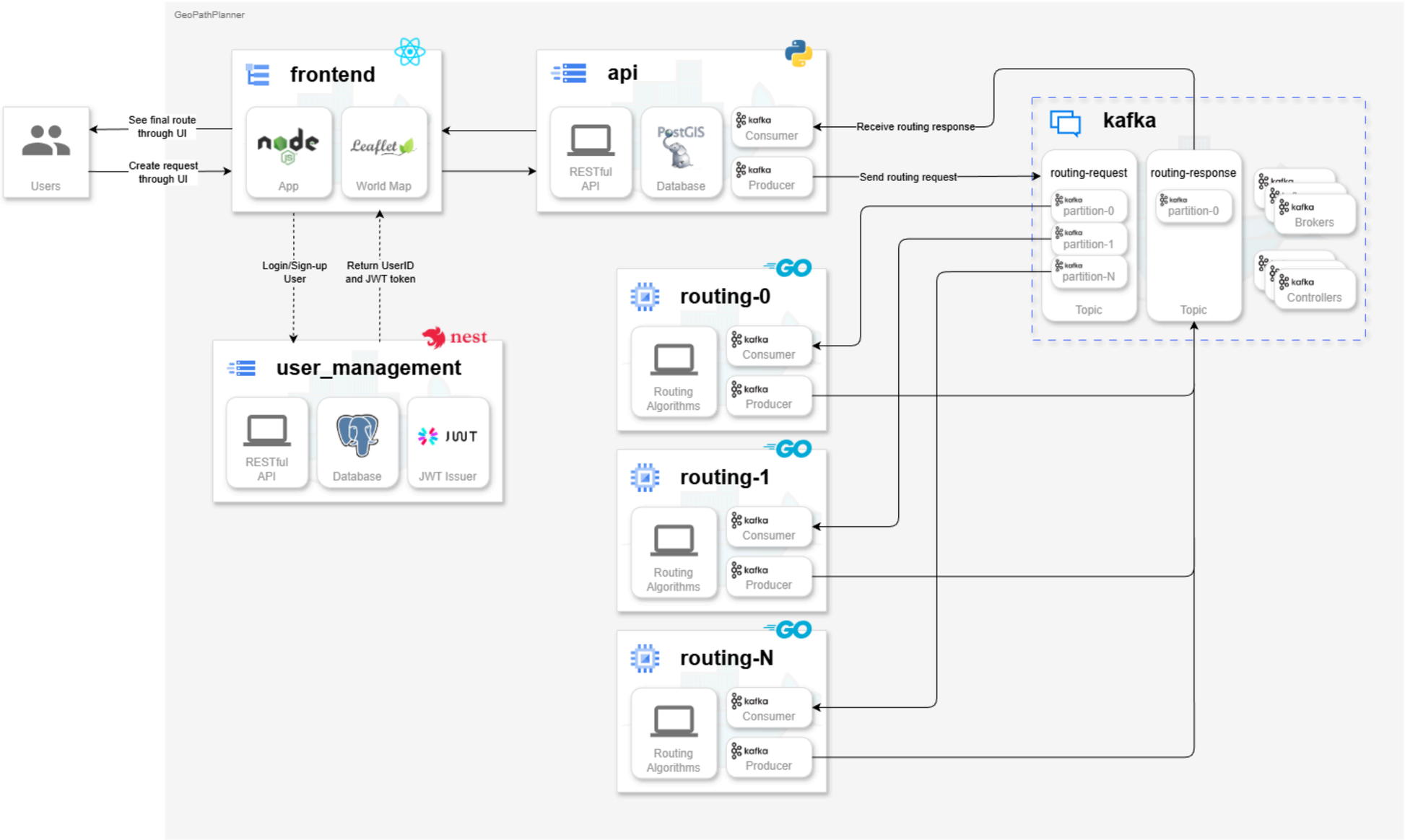
- Implement 3D path planning algorithms.
- Uses specific in-memory data structure (RTree) for efficiently performing geospatial queries.
- Uses goroutines for concurrently managing multiwaypoints requests.
- Kafka Producer/Consumer interaction with API service.

4

## User Management Service

- Handles registration, login, and authentication.
- Manages user data in a a database for persistence and retrieval.
- Issues JWT tokens for authenticating user’s requests.

# Application Architecture



# Frontend – Technologies and User Interface

- Main Technologies: Developed as a Single Page Application (SPA) using React with Vite for a modern development experience.
- UI and Mapping: The interface is built with Bootstrap 5 for a responsive layout. The interactive map is implemented with React-Leaflet, extending its functionality with Leaflet-Draw to allow users to draw and edit geometries (waypoints and obstacles).
- User Interaction:
  - Users can search for locations using a geocoding service (Nominatim) and center the map on the result.
  - Waypoints (markers) and obstacles (polygons) can be drawn, edited, and deleted directly on the map.
  - Supports the import of waypoints and obstacles from .geojson files.
  - Allows you to configure the altitude for each waypoint and the altitude limits (min/max) for obstacles.
- Containerization: The application is containerized with Docker and served by an Nginx web server, configured to correctly support client-side routing managed by React Router.

# Frontend

Route Calculation Flow:

- The “Compute” action collects the data entered by the user (waypoints, obstacles) and the algorithm parameters.
- It builds a payload that complies with the specifications (based on GeoJSON) and sends it to the */routes/compute* endpoint of the API Gateway.
- During processing, the interface displays a loading modal.
- Upon success, the calculated route is drawn on the map and a summary of the results (e.g., cost, time) is shown to the user.

Authentication and User Functionality:

- Integrates with the User Management microservice for registration and login.
- Session management is handled via JWT (JSON Web Token), saved in localStorage to authenticate subsequent requests.
- Authenticated users can access a personal area to view the history of calculated routes.
- From the history, it is possible to reload a route on the map to view it or delete it permanently.

# User Management Service

- This microservice is developed using NestJs, that is a Node.js framework. It manages the authentication lifecycle, from registration to login.
- The microservice is fully containerized using Docker. The configuration includes three main service: the NestJS appliction exposed on port 3000, the PostgreSQL database and pgAdmin for database management
- The system exposes a set of APIs for registration, login, and modification. API documentation is available through Swagger, accessible at the dedicated endpoint, which provides an interactive interface for testing endpoints and understanding the structure of requests and responses.
- User data are stored in a pgAdmin database.

# User Management Service

- This microservice issues a JSON Web Token to securely represent a user’s authentication. Each token consists of a header, a payload and a signature.

- This is a token signed with the HS256 symmetric algorithm; the signature is calculated using the JWT_secret.

- Since the algorithm is symmetric, all services that need to validate the token must know the secret.

# API

Communicates with frontend and routing module

Containerized with Docker

Kafka as a communication layer with Routing microservice

Postgis database for requests' history

Gives list of endpoints

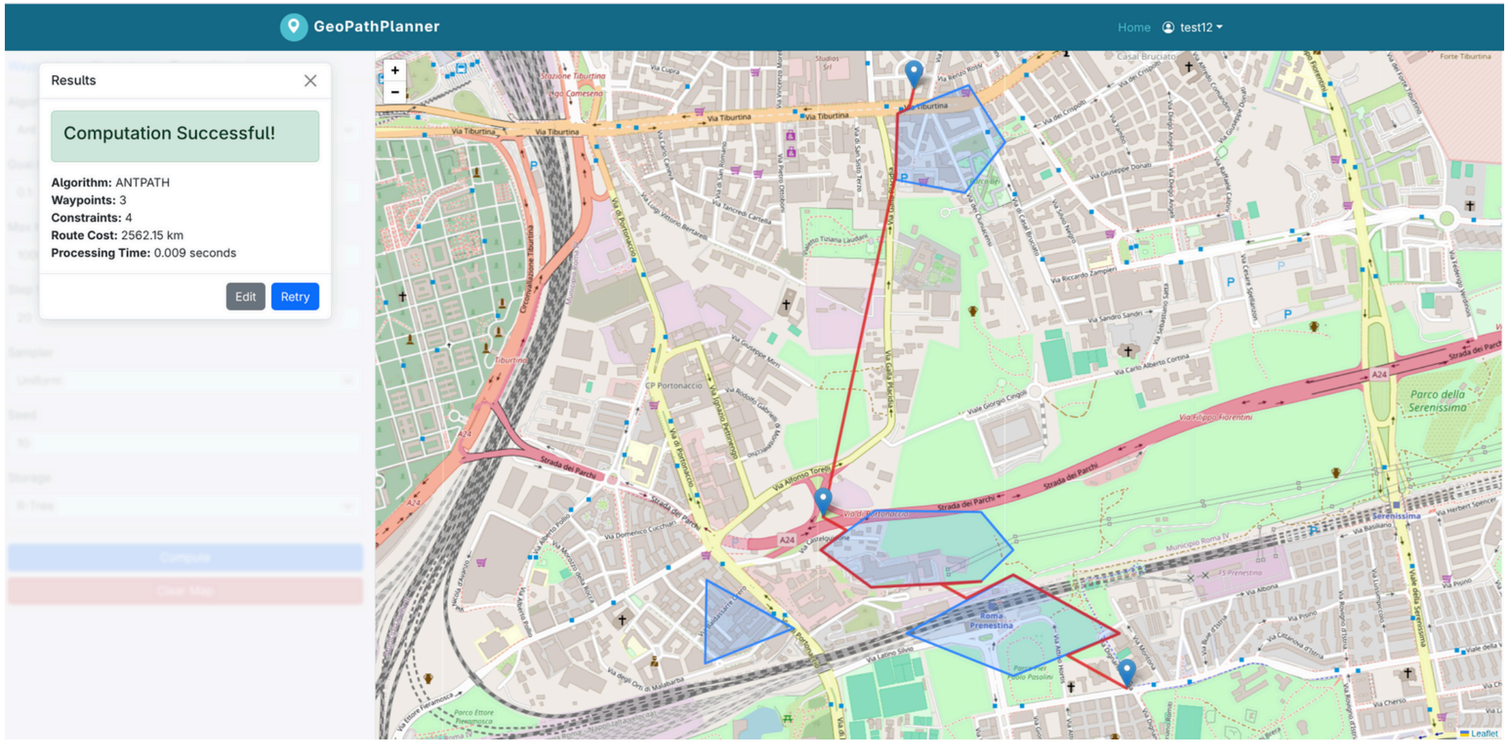
Authenticates requests using user_id (query) and JWT token (header)

# API: Process description

1. Frontend sends JSON with waypoints & parameters
2. API validates data, generates unique request_id
3. Publishes message to Kafka's topic "routing_request"
4. Waits for the response from the routing module on the topic "routing_response" (with a default 120s timeout)
5. Consumes routing response and give the results to the frontend
6. Stores the route in a Postgis database (if user is authenticated)

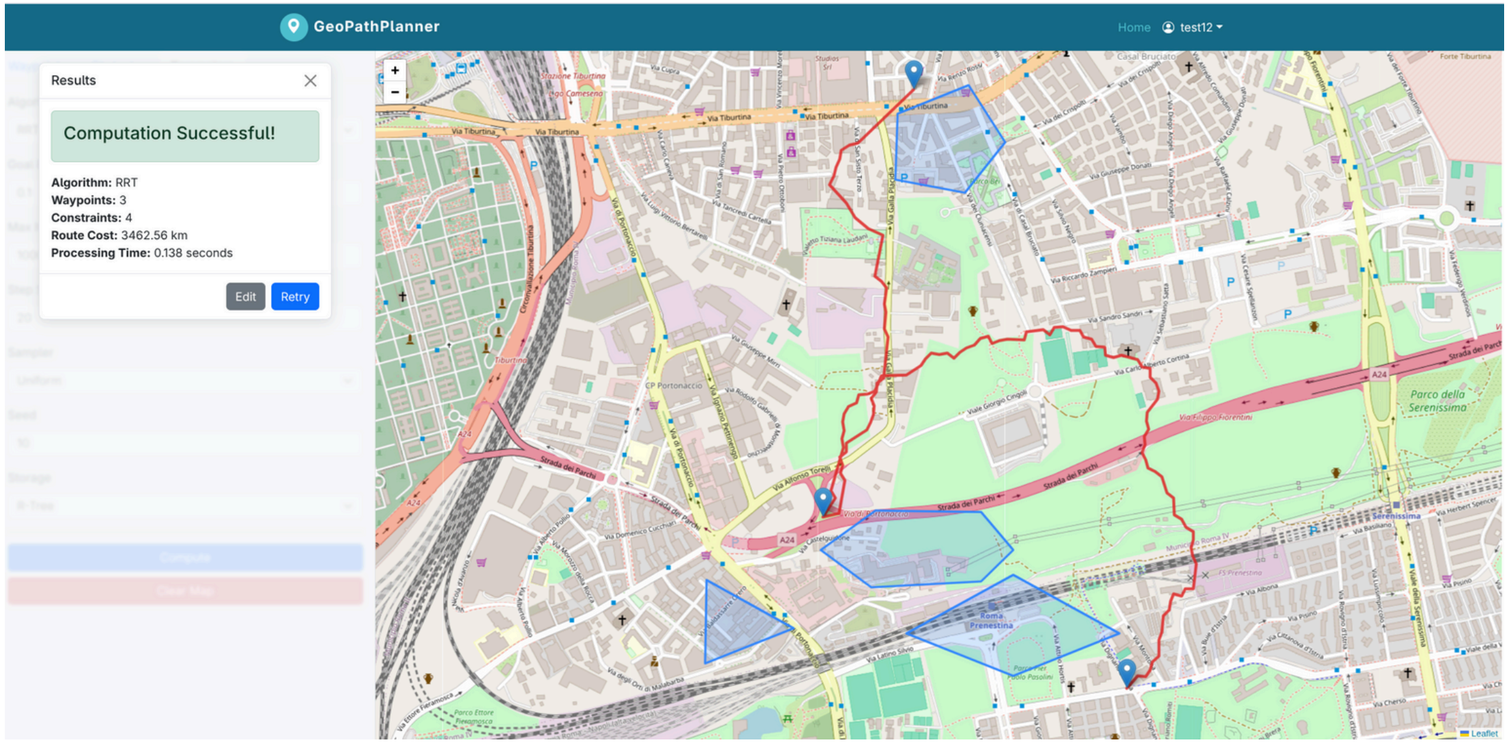
# Routing

- Developed in Go, consumes routing requests from a Kafka topic and produces results to another topic
- Implements three algorithms:
  - **AntPath** (fast, finds any route)
  - **RRT** (fast, finds feasible route)
  - **RRT*** (slower, aims for optimal route)
- Store spatial data in **R-Tree** for optimized geospatial queries
  - k-NN and computing intersections can run in O(log N) vs O(N)
- Uses goroutines to **parallelize multi-waypoint computations** (one per waypoint pair)
- **Scalable** with N replicas consuming from N Kafka partitions



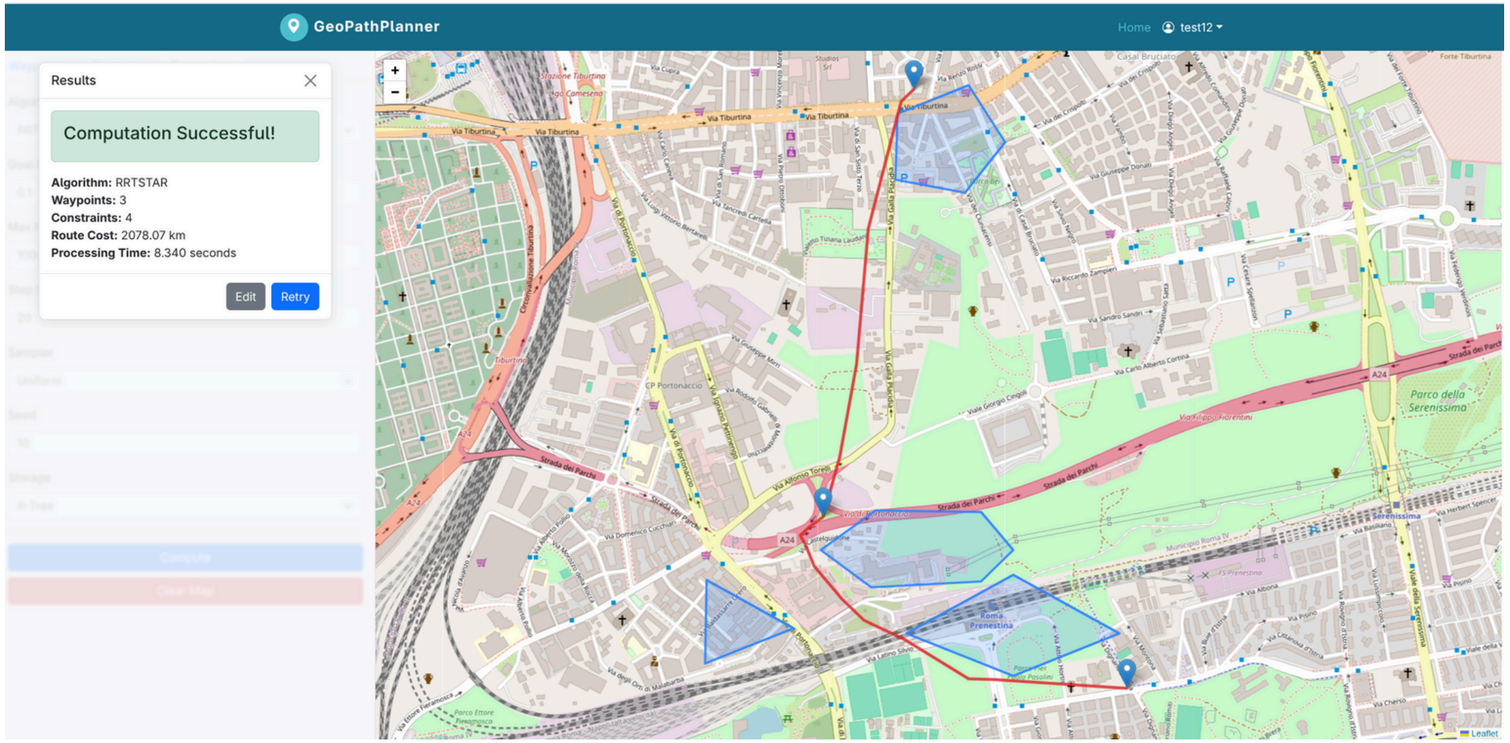
# Routing

- Developed in Go, consumes routing requests from a Kafka topic and produces results to another topic
- Implements three algorithms:
  - **AntPath** (fast, finds any route)
  - **RRT** (fast, finds feasible route)
  - **RRT*** (slower, aims for optimal route)
- Store spatial data in **R-Tree** for optimized geospatial queries
  - k-NN and computing intersections can run in O(log N) vs O(N)
- Uses goroutines to **parallelize multi-waypoint computations** (one per waypoint pair)
- **Scalable** with N replicas consuming from N Kafka partitions



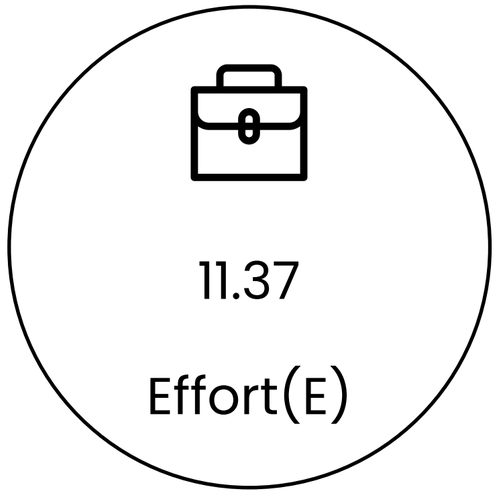
# Routing

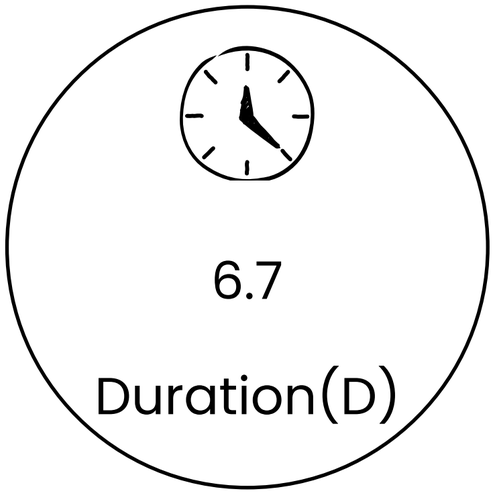
- Developed in Go, consumes routing requests from a Kafka topic and produces results to another topic
- Implements three algorithms:
  - **AntPath** (fast, finds any route)
  - **RRT** (fast, finds feasible route)
  - **RRT*** (slower, aims for optimal route)
- Store spatial data in **R-Tree** for optimized geospatial queries
  - k-NN and computing intersections can run in O(log N) vs O(N)
- Uses goroutines to **parallelize multi-waypoint computations** (one per waypoint pair)
- **Scalable** with N replicas consuming from N Kafka partitions

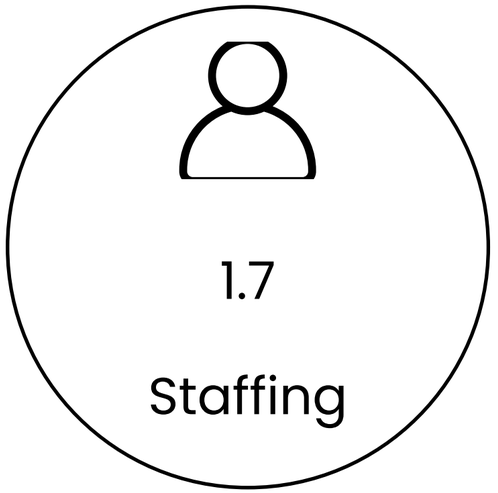


# Development Process

- Managed using JIRA
- 18 user stories: all completed except 1
  - Total: 82 story points, assigned using Fibonacci sequence (1, 2, 3, 5, 8, ..)
- Average lines of code per function point equal to **52**
- In order to estimate the effort we use the **Semi-Detached model**, suitable for a project with a mix of experienced and inexperienced team members.



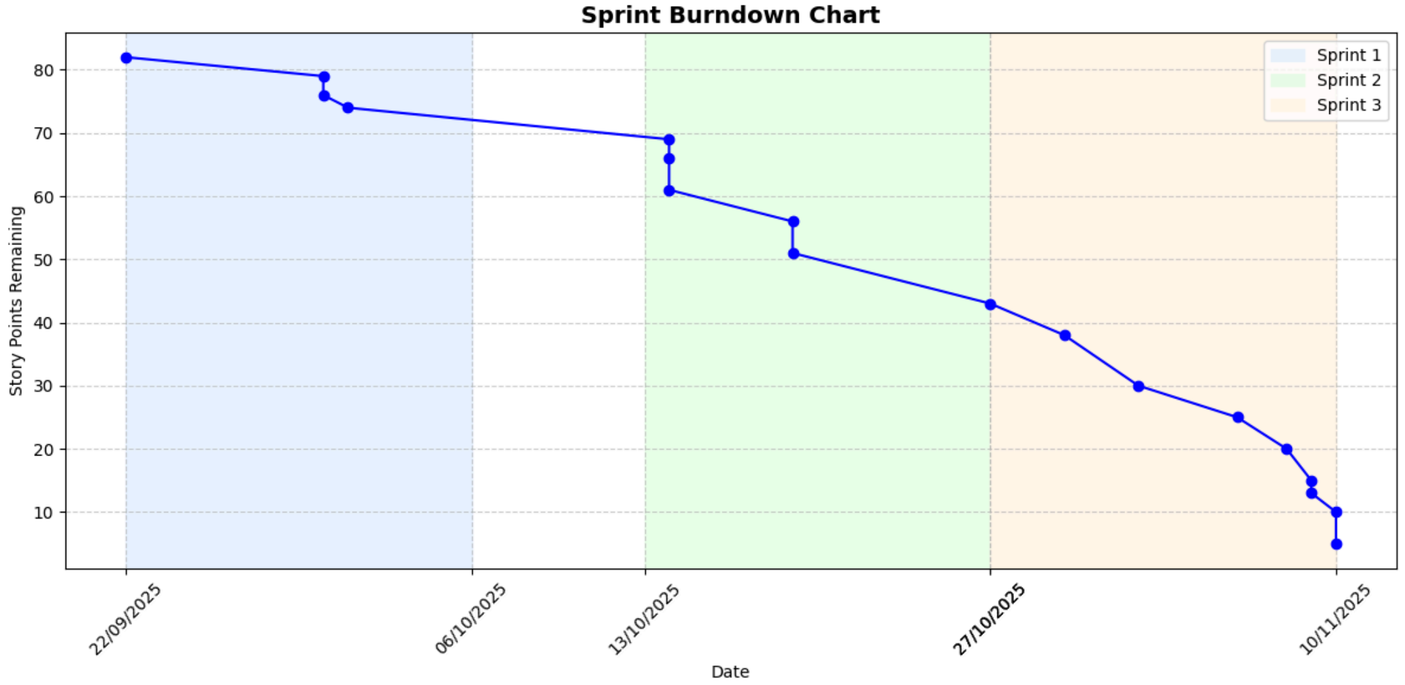



# Development Process

- **SPRINT 1** *(22/09/2025 - 06/10/2025)*
  - Build the overall architecture and define the project structure.
  - User stories: [8, 7, 9] = 8 SP

- **SPRINT 2** *(13/10/2025 - 27/10/2025)*
  - Implement the routing and advance other microservices for integration.
  - User stories: [1, 2, 3, 10, 14] = 23 SP

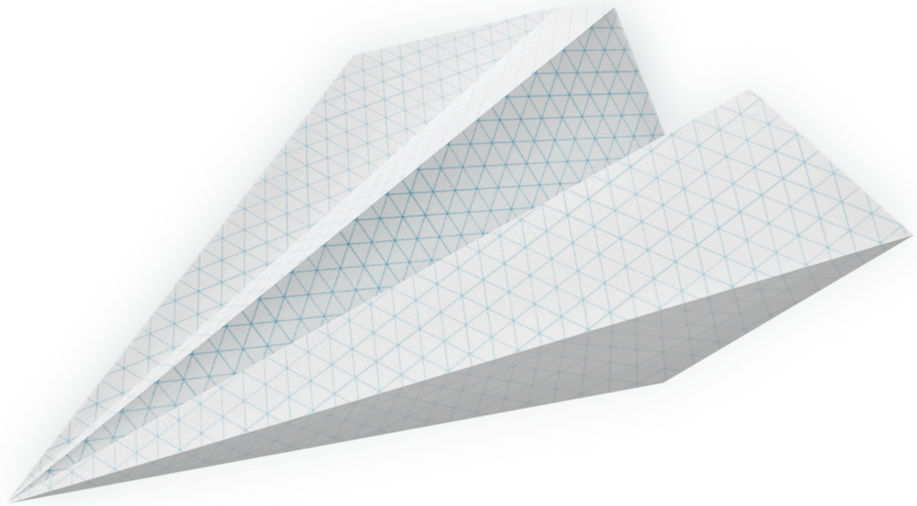
- **SPRINT 3** *(27/10/2025 - 10/11/2025)*
  - Complete the project and prepare for the demo.
  - User stories: [4, 5, 11, 12, 13, 15, 16, 17, 18] = 46 SP



# Conclusions & Future Work

- System successfully computes 3D drone routes with customizable inputs and parameters.
- Supports independent, parallel routing requests and scalable resource usage thanks to kafka consumers scaling.

- Ideas for Future Enhancements:
  - Leverage PostGIS for geographic filters on route history
  - Use Redis caching for faster access to historical routes
  - Support OAuth login via Google, Facebook, etc.
  - Implement more efficient routing algorithms



# Thanks for your attention!

# DEMO

http://localhost:8081/