## Abstract

The main objective of this project is to create a high-performance data system that allows data ingestion from different telemetry devices and allows frontends to request the ingested telemetry data from an API-endpoint. To create a more future-proof, reliable, and robust software product, it should be designed to provide NoSQL-like functionality and be built around containerization to enable horizontal scalability and maintainability. With the help of modern technologies and established methodologies, the data system will be able to manage substantial amounts of data efficiently. Moreover, it is designed to enforce a simplistic coding style and increase upgradability for changing conceptional requirements.

## Concept

The data system is aimed to provide containerized Docker services, which includes: A running database, an API, and an import script.

- The database was built on the generated database entities from our API and initialized at container startup once. The database management software of choice falls onto PostgreSQL because of its existing structure around containerization and existing official images. Furthermore, it supports Docker-health-checks and enables increased automation within this project.
- A Python-based import script reads the dataset on **first** container start and ingest the data accordingly into the running database. It also creates an initial authentication token with correctly set token-permissions.
- The API provides one gRPC-based service that provides two distinct endpoints for each functionality. One endpoint is meant to take data from a telemetry device and store it in a relational database with an additional NoSQL-field. The second endpoint offers the functionality to read data from the system and returns a pageable result that offers additional information about the amount of requestable data. Every functionality is present as REST- and gRPC-endpoint to increase flexibility and allow for easy access from browser interfaces while still providing efficient Remote Procedure Calls (RPC) for the telemetry devices and additional flexibility to increase the functionality with bi-directional streaming if needed.

## Implementation

Extreme Programming (XP) was the chosen methodology for this project and consisted of 4 phases: Planning, designing, coding and testing. Because of the planning and designing phases of extreme programming as well as my existing experience in computer science, I have not encountered any major issues regarding coding, implementation, or project-specific tooling. I made use of extensive features of various frameworks (.NET, EF Core, protobuf) and used a platform- and language-agnostic implementation of the Remote Procedure Call (gRPC) as a base for defining the transactional data inside a file that is reusable and portable to other projects without breaking compatibility to

newer API versions. The resulting software product corresponds to the main goal of this project because it meets the project's quality expectations:

- Robust, efficient transmissions from telemetry devices via RPC or REST
- Scalability, since fully containerized with Docker and PostgreSQL's sharding support
- Maintainable and future proof with backwards compatible Remote Procedure Calls (RPC)
- Flexible requests and pageable results

### Optimizations

Even though the project was already providing all the mentioned functionality, there were some optimizations that could or just should be made. One of the optimizations was figured out when finishing phase 2 of this project, and the code was about to be documented. In this case, it was gRPC-JSON-Transcoding. This feature enables classic REST-endpoints and creates a proxy between the gRPC-services and web requests. Every request to the proxy will internally be translated to gRPC-calls, which allows for automated OpenAPI documentation with Swagger. With this change, all .NET controllers were removed, and the web-endpoints as well as the gRPC-endpoints will return identical data structures based on the Protocol Buffers. One of the other issues that should be optimized was naming consistency since Docker on Linux manages the capitalization of folder names a bit differently than Docker Desktop on Windows. Docker would straight up refuse to build the Python image on Linux since the folder name was capitalized in the docker-compose file, while the actual folder name was not. Additionally, one whole endpoint was removed due to redundancy with an already existing endpoint, which had a lot more options and flexibility to request ingested data. These options were further improved by providing the ability to filter the database entries by a specific day.

### What have I learned?

A few of the most important lessons that I personally learned while developing this project were:

- **Know your tools**: Know what the tools of your choice are capable of. What are their strengths, and what are their known weaknesses? In what way are you going to profit from its strengths? And how are you going to avoid weaknesses?
- **Make mistakes**: What are the common mistakes with the tools of your choice? Are there FAQs or other resources to help with your issues? Are there best practices to avoid certain issues at all?
- **Always use a version control system**: Never start a project without setting up a proper version control system (e.g., Git, Mercurial, or Subversion). This will help to avoid crucial issues, data loss, and easier development with multiple branches and options to integrate Continuous Integration (CI) and Continuous Deployment (CD).
- **Change one thing at a time**: Minor changes increase the likelihood of finding common errors and increase the readability of commits and pull requests. This is especially useful if unit tests are already present and verify the implemented functionality.