Távlink Remote Monitoring Platform

Final Year Project Semester 8

Zalán Tóth - 20102768
Computer Forensics and Security Year 4

SETU Waterford
School of Science and Computing
Department of Computing and Mathematics

Waterford City, Ireland
29/04/2026

Abstract

Távlink is a prototype remote monitoring platform designed for secure and flexible telemetry collection across IT and OT environments. The prototype uses a push-based model, where the monitored objects send telemetry data to the ingestion endpoints. Data is validated against user-defined configurations, stored in MongoDB time-series collections, cached and distributed using Dragonfly, and displayed through a WebSocket-based control-room web interface. The system is experimentally deployed on a self-hosted Kubernetes cluster using Forgejo CI/CD. Although the developed foundation is functional, it is still in an early prototype stage.

Table of Contents

1. Introduction

1.1. Background

The original concept behind Távlink was to address critical infrastructure needs by creating a remote monitoring system primarily targeted at operational technology (OT) environments. The main intention was to provide a secure and flexible method for collecting telemetry data from devices such as programmable logic controllers (PLCs) and sensors, allowing these systems to be observed remotely.

As the project progressed, the scope evolved into a broader platform-oriented approach. This change did not alter the original objective, but it highlighted the need for a stronger architectural and system design foundation. In practice, a monitoring solution must do more than simply collect data. It must also be scalable, fault-tolerant, highly available, and cost-effective, while remaining flexible enough to adapt to different customer and environmental needs.

For this reason, Távlink moved towards becoming a broader and more versatile monitoring platform. The platform is designed to support telemetry collection, storage, archival, and near real-time visualisation through a control-room-style interface where appropriate. This platform-oriented approach also enables future expansion into additional monitoring use cases beyond telemetry collection alone, such as health monitoring and other relevant services.

A further motivation for this self-built platform approach is operational control and cost efficiency. Instead of relying entirely on third-party cloud platforms, the project explores a private-cloud-oriented deployment model using self-managed infrastructure. While this increases architectural and operational complexity, it also offers stronger control over the environment, greater flexibility in system design, and potentially more favourable infrastructure costs at scale. Such a model is also capable of supporting multiple organisations within a software-as-a-service (SaaS) solution.

1.2. Problem Definition

Both OT and IT environments, as well as their convergence, introduce different challenges. The OT-related considerations were discussed in the previous Távlink report. On the IT side, however, providing a reliable monitoring solution requires the ability to ingest and process telemetry, preserve historical data, and provide near real-time observability where needed. The system must also remain flexible enough to support different customer requirements, future improvements, and broader use cases.

At the same time, flexibility introduces its own design challenges. A platform that is highly configurable must still remain usable, understandable, and maintainable. Balancing flexibility with usability is therefore a major challenge in the design of Távlink.

Another important challenge is the deployment strategy. A monitoring platform and its supporting infrastructure must remain available even when individual components fail, and it must scale in a controlled and cost-effective way. Traditional fully managed cloud services can solve many of these issues, but they may reduce control and introduce higher costs with platform dependencies. In contrast, self-managed infrastructure offers greater control and cost flexibility, but it requires more careful design to ensure resilience, fault tolerance, and operational maintainability.

1.3. Objectives

The aim of this project is to design a secure, scalable and flexible remote monitoring platform for telemetry data across different environments, particularly in critical infrastructures use cases.

To achieve this aim, the project focuses on the following objectives:

1) To design a platform capable of collecting, storing and presenting telemetry data.
2) To support a push-based telemetry model in which devices and services send data to Távlink's ingestion endpoints.
3) To provide near real-time monitoring through a remote, control-room style interface.
4) To enable high flexibility through reusable configurations and profiles so that the solution can adapt to different customer and environment needs.
5) To explore a deployment approach that is scalable, fault-tolerant, highly available and cost-effective.
6) To create the platform in a way that it can later be extended with additional capabilities such as alerting, notifications, pull-based monitoring, billing and other SaaS functionalities.
7) To create a strong architectural and design foundation for all possible future extensions and use cases such as bridging mechanisms.

Although the platform is intended to support many future monitoring scenarios, the core motivation remains the same as earlier: to provide safe and efficient monitoring of OT devices remotely from the IT domain.

1.4. Scope

The scope of this report is limited to the adjusted design and prototype implementation of the Távlink monitoring platform, with the main focus placed on flexible architecture, telemetry ingestion, data handling, and deployment foundations.

At the current stage, the platform focuses on an experimental push-based monitoring approach, in which devices and services report telemetry to Távlink's ingestion endpoints. This approach aligns well with the original OT-oriented concept, where monitored devices transmit data outward towards the platform. Although a pull-based approach is also relevant for services such as health checks and endpoint availability monitoring, it remains outside the scope of the current implementation and is treated as future work.

Similarly, while the long-term plans include support for bridging software to connect isolated or sensitive OT environments safely, such bridging functionality is not implemented at the time of writing. The current project scope ends at the platform itself. This limitation was necessary in order to prioritise the development of a stable and reusable platform base within the available time available for the project.

As a result, this report presents a prototype foundation for future development rather than a production-ready monitoring solution.

2. Technology Stack

2.1. Backend and Data Technologies

Backend is the core of this project, so the primary objective was to select technologies which could be future-proof and allow for scaling when needed.

2.1.1. Python & FastAPI

The core of the project is written using Python, which is a feature and library rich, high-level programming language. FastAPI, as its name suggests, is a fast API framework for web communications. It was chosen as it proved to be a solid foundation in previous projects for its robustness and because it is easy to learn and use. FastAPI is Python-based; it benefits from the vast amounts of Python libraries available, and unlike general Python, FastAPI is high-performance, matching the speed and throughput of even NodeJS. [1] There were several libraries used within Python, and one of them is important and influential enough to be mentioned: Pydantic, the core data validator, parser and serializer within the FastAPI codebases.

2.1.2. Uvicorn (Ultra Fast Python Web Server)

Uvicorn is an Asynchronous Server Gateway Interface (ASGI) web server implementation for Python. It works by bridging applications, such as FastAPI, to work with HTTP protocols. It can handle asynchronous frameworks with ease. [2]

2.1.3. MongoDB

MongoDB is a NoSQL database (not only structured query language), meaning it's non-relational and has a flexible schema. While flexibility allows for development on the fly, its scalability is the major reason why MongoDB was selected. Its community version is free to download, run and use. There is also a cloud native version through their platform, Atlas. Both self-hosted and cloud-native options are used in the project.

2.1.4. Dragonfly

The project needed a fast and short-term storage option to be used primarily for caching data. One of the most famous options includes Redis, an in-memory database. It's well proven and well supported across programming languages. However, it has limitations for future scaled use; more possibilities were explored, and Dragonfly was chosen as the final solution. Dragonfly is a high-performance, multithreaded in-memory database which is almost completely Redis compatible, meaning it benefits from compatibility across platforms just by using Redis libraries. Compared to Redis, it offers much higher performance and better data compression. [3] Later, its publish and subscribe feature was also explored and implemented.

2.1.5. Websockets

Websockets are two-way communication channels that remain persistent over a single TCP connection. [4] It was used to display near real-time information to the web interface control room. This technology was combined with Dragonfly's pub/sub capabilities to make the control panel work in a fast and low-latency manner. SSE was also explored here as a simpler option, as the current communication need is uni-directional (server to client); however, Websocket was chosen for its bidirectional communication capability. This opens up future capabilities for near-real time control features (client to server) alongside monitoring if needed.

2.1.6. Authbridge (Rifstar Identity and Authentication Provider)

Authbridge, also referred to as Rifstar ID, is a proprietary Identity and Authentication Provider. It is currently at the prototype stage, similarly to Távlink. As the author of this paper develops Authbridge, Távlink, and the latter described Databridge, many ideas and structures of the project resemble these other projects due to previous experience, and some features may be directly borrowed. These systems are planned to be highly compatible with each other, so users in the future can benefit from this much larger ecosystem, whether it's centralised billing, organisation or identity management. The authentication of Távlink is done very simply in a stateless way to accommodate future scaling capabilities, and central session management is handled by Authbridge itself. The first paper of Távlink explains how this works under Section 5.

2.2. Frontend and Interface Technologies

Frontend development was not the primary target this time, but rather a necessity in order to get a human interface to interact with the backend systems for testing. For this purpose, familiar technologies were chosen to work with. Experience in these were crucial, especially as this part was mostly developed using artificial intelligence; heavy reliance on such tools only works if the human who makes the requests and approves the code segments understands these technologies.

2.2.1. Typescript

TypeScript is a superset of JavaScript extending capabilities like type checking at compile time rather than runtime. This reduces bugs and improves maintainability. It was partially chosen because of its additional features, however primarily because it is the standard when working with frameworks like React and Next.js. TypeScript compiles into JavaScript.

2.2.2. Node.js

Node.js is a well-known runtime for JavaScript. It is a dependency of Next.js and also serves as the package manager (npm). [5]

2.2.3. Next.js

Next.js is a full-stack React framework. It has many features like filesystem-based routing, server-side and client-side rendering and built-in API route handling. [6] Due to its capabilities, it reduces many manual configurations and work, making it easier and faster to deploy.

2.2.4. React

React is a component-based JavaScript library for building user interfaces. It has extensive documentation, making it an excellent choice for AI-assisted development. [7]

2.2.5. Databridge (Rifstar Content Management System)

Similar to Authbridge, Databridge is also a proprietary service, functioning as a content management system. Its purpose is to serve as a central source of truth for variable and reusable content, such as translations or company info. In this context, the interface uses it to fetch and cache translations into the control panel interface as an experimental feature.

2.3. DevOps and Deployment Technologies

Important to emphasise that these technologies are also used in other proprietary projects such as Authbridge and Databridge, not just Távlink. Some methods may be different, but the technologies, core structure and underlying infrastructure are the same. That also means that when Távlink is deployed into a prototype, it communicates with the prototype deployment of Authbridge and Databridge.

2.3.1. Git & Forgejo

Git was used for source control alongside self-hosted Forgejo for remote repository management. Forgejo is an open source git platform and is a fork of Gitea. It's similar to GitHub, but self-hosted for full control. What makes it stand out is its CI/CD (Continuous Integration / Continuous Deployment) pipeline capabilities with Forgejo actions. While that feature is similar to GitHub actions, it's worth noting that there are key differences, so they are not directly interoperable with each other. For environment variables and secrets, Forgejo Secrets is used to safely provide sensitive variables in deployment.

2.3.2. Docker

Docker is a containerisation technology allowing to run applications inside isolated environments in a reproducible manner, making it easy to deploy almost anywhere. The Dockerfile present in each git repository acts as the blueprint which creates the Docker Image, a package which contains all the necessities for running the application, including the environment, code, libraries and settings of the project. The image can be stored in a Docker Registry, a central repository for docker images, to be reused later on. For this project, a self-hosted Docker Registry was used. Docker is also used as the backbone for Forgejo actions. Each task defined in the CI/CD pipeline is handled by Docker servers.

2.3.3. Containerd

Containerd is a Kubernetes-compatible containerisation technology similar to Docker. It is compatible with Dockerfiles and Docker Images, and in this project, it is used to deploy the created Docker Images within Kubernetes.

2.3.4. Kubernetes

Kubernetes, also called K8s, is a container orchestration platform which automates running, scaling and managing applications. The usage of this technology is a deviation from the original plans defined in the first Távlink paper under Section 7 Deployment Architecture. This new method of deployment was chosen as it works well with other existing infrastructure, such as Forgejo and Docker, and it's very robust and full of features. As the newer Kubernetes doesn't work with Docker, Containerd was used instead for the containerisation technology, but as previously mentioned, Containerd is compatible with Docker images. While Kubernetes is proven to have a difficult learning curve, it's certainly easier to use it than to write a completely new and unique system as previously planned. The prototype deployment of K8s includes 2 dedicated server machines, one acting as a control plane (responsible for the management of the cluster) and one acting as the worker node (where applications get provisioned), making it a small experimental cluster.

2.3.5. Netbird

Netbird is an open source peer-to-peer virtual private network (VPN), capable of many applications. In this context, Netbird was used for 2 major purposes: giving the backbone for safe internal communications between the K8S cluster nodes and providing a business VPN with exit nodes. When connected to this self-hosted VPN network, the client can be configured to route all internet traffic through this private network. When this is configured, in order to reach the internet, there is a need for a gateway where the data will exit to the internet. These are called exit nodes, and there are 4 of them configured. This makes your internet address look like you are located at that exit node, as your traffic reaches the internet through that node. For security, each project component has heavy whitelisting, and some servers include heavy firewalling rules only to allow connections from those 4 exit nodes (so access to the Távlink prototype deployment is severely restricted). This is needed to ensure the safety of these prototype systems, as they are not ready to be completely public on the internet.

2.3.6. Ubuntu

There are various dedicated server machines and virtual private servers (VPS) used in this project, but they all share the same Ubuntu LTS (22 or 24) operating system. These servers live in datacenters across Europe and are managed remotely via SSH. Forgejo, Kubernetes, Netbird, Docker and Docker Registry all live and run on these servers, often separated from each other.

3. Platform Design

3.1. Architecture Reference

The architectural design of Távlink was presented in the first Távlink paper.

For reference, Table 3.1 includes a list of services that make up the current system.

Service	**Repository**	**Port**	**Prototype Deployment**	**Transport**
Core API	tavlink-core	8267	prototype-core.tavlink.net	HTTPS (REST)
WebSocket Gateway	tavlink-core	8263	prototype-ws.tavlink.net	WSS
Ingestion Server	tavlink-ingestion-server	8268	prototype-ingest.tavlink.net	HTTPS (REST)
Control Panel Interface	tavlink-web	3779	prototype-app.tavlink.net	HTTPS

Table 3.1 Távlink services

For simplicity, there are 2 MongoDB and Dragonfly servers. Each has 1 for local testing and 1 for prototype deployment. For local testing, self-deployed versions are used, while the prototype uses cloud ones (MongoDB Atlas and Dragonfly Cloud).

3.2. Communication and Protocols

3.2.1. HTTPS REST - Request / Response

The external-facing protocol is HTTP/1.1 over TLS in a RESTful manner. The Core API endpoints use standard HTTP methods (GET, POST, PATCH, PUT and DELETE) with JSON request and response bodies. Pydantic enforces schema validation on inbound requests, and it rejects malformed payloads. The response follows the standard of HTTP status codes as well.

The Ingestion Server currently has a single POST endpoint to accept telemetry data points from devices. This separation was needed in order to allow for independent scaling as the load on these servers are completely different than the Core API. The client can follow the fire-and-forget approach, but there is also information if the data gets rejected, so that the device can retry the ingestion.

The web panel proxies all backend calls through its own API routing. This way, the browser never communicates with the Core API directly. All auth details in the browser get appended to requests, and the panel forwards them to the backend.

3.2.2. Websocket

For real time telemetry, the platform uses the Websocket protocol over TLS (WSS). The Websocket Gateway runs as a standalone service. The connection lifecycle is the following:

1. The client opens a WebSocket connection. The gateway validates the auth details from the appended JWT token, and it extracts the relevant details.
2. When connected, the client sends subscribe or unsubscribe actions as JSON frames. Each monitored object (device) is handled as a single subscription.
3. The gateway sends JSON frames to the client when there is new telemetry data or a notification.
4. The client keeps the connection alive via a ping-pong mechanism.

The Websocket Gateway simply subscribes to the appropriate Dragonfly Pub/Sub channels defined by the client and just simply forwards them using WSS.

3.2.3. Pub/Sub via Dragonfly

As Dragonfly is Redis compatible, it provides the same publish and subscribe commands. It is used to provide real-time delivery and cache invalidation. Table 3.2.3 shows the channels that are used (and working). Alarms and notifications are also using this method, but it is in a very early experimental way, therefore left out from this list.

Channel	Publisher	Subscriber	Payload
telemetry:{org}:{obj}	Ingestion Server	WS Gateway	Batch of live metric values
config_invalidate:{obj}	Core API	Ingestion Server	Config change > refetch
apikey:invalidate	Core API	Ingestion Server	API key hash (to invalidate)

Table 3.2.3 Pub/Sub channels

3.2.4. MongoDB Wire

The communication with MongoDB is done using the standard asynchronous Motor driver over MongoDB's binary wire protocol using a connection string. Both the Core API and the Ingestion server keep a persistent connection to MongoDB.

3.3. Transport Security

3.3.1. TLS (Transport Layer Security)

All external traffic enters the Kubernetes cluster through an NGINX Ingress Controller, which terminates TLS. Certificates are issued by a cert-manager within K8S using Let's Encrypt. After the TLS termination, traffic between the Ingress Controller and the pods travels within the cluster's internal network.

3.3.2. Mutual TLS

For future critical infrastructure deployments where defence-in-depth is needed, mTLS can be used instead of just simply TLS for extra security. This ensures that both parties can validate each other, and the trust is two-way.

3.3.3. Internal Cluster Communications

The Kubernetes cluster nodes communicate over Netbird, a WireGuard-based peer-to-peer VPN. All traffic between nodes is encrypted at the network layer by WireGuard. This way, neither service has to manage TLS certificates in this context. Simple, secure and easy to set up.

3.4. Data Flow

3.4.1. Telemetry Flow

Step-by-step process on how telemetry gets registered in the system:

1. The device sends telemetry by constructing an IngestBatch containing one or more DataPoints. It sends an HTTPS POST request to the Ingestion Server with the device's API key.
2. The request arrives at the NGINX Ingress Controller. The TLS gets terminated, and the plain HTTP request gets forwarded to the tavlink-ingest Kubernetes Service. Then it gets redirected to an available Ingestion Server pod.
3. The Ingestion Server extracts the key from the header and computes the hash of it, which it uses to evaluate against its available resources. If the key exists, it creates an ApiKeyContext containing scope, organisation and device info.
4. The Ingestion server validates whether the API key is allowed to submit data for the specified device.
5. The server loads the device's active configuration primarily from local cache. The configuration includes information about what can be stored and how.
6. Each datapoint gets inspected and evaluated according to the device configuration.
7. Document is constructed with a timestamp alongside metadata such as device id, organisation id, and metric slug and config version.
8. Dragonfly caches the value(s) and also publishes a new telemetry event into its channel. The Websocket Gateway, which is subscribed to this channel, picks it up and sends it to the client's user interface. Client gets near-real-time telemetry information in the control room interface.
9. Datapoints get inserted into the MongoDB Time-Series in bulk.
10. The Ingestion Server returns an answer whether the datapoints were accepted or rejected.

3.4.2. Configuration

When a user modifies a device's configuration (applies a collection profile), the change must propagate to the Ingestion server so the telemetry gets processed against the newer, more relevant configuration. When the config gets updated, the Core API publishes a message through Dragonfly to invalidate the config. The Ingestion Server, which is subscribed to that channel, receives the message and invalidates the local cache, forcing a refetch of configurations. API key change also works similarly.

3.4.3. Provisioning

Távlink has a zero-touch provisioning mechanism allowing devices to self-register themselves with the platform. The flow is the following:

1. The client creates a provisioning token and specifies templates.
2. The provisioning token is given to the device.
3. On new startups, where there are no credentials set yet, the agent makes a call to the Core API with the provisioning token containing device info like name and description.
4. The Core API validates the provisioning token, creates the device in the system and applies the specified templates if set. It also creates an API key scoped to the device and returns it in the response body.
5. The device stores the credentials locally and starts using the API key for ingestion requests.

Using this method, multiple devices can be registered with a single provisioning token with the same templates and settings, making onboarding much easier.

3.5. IP Whitelisting

All externally reachable components implement an IP whitelist middleware to restrict access to these services. It was borrowed from Authbridge and adjusted to the codebases, but the core idea and operations are the same:

1. It parses the allowed networks from an environment variable (comma-separated CIDR notation).
2. Extracts the client IP from the X-Forwarded-For header.
3. Rejects requests if not in the Whitelist (health endpoints are exempt from this).

This is the major wall guarding prototype deployments.

4. Implementation

4.1. Project Structure

While almost every component is separated into a different Git repository, they still share many similarities. Each codebase has its own Dockerfile, Kubernetes manifests and CI/CD pipeline. They all follow the same version management and similar deployment strategies.

In the Python projects, a sub-application mechanism was used to separate different API breaking versions. For the early, experimental API v0 was used, and v1 is reserved for stable, well-tested endpoints. The general versioning follows SemVer 2.0.0 (major.minor.patch-release+build.commit). This version is exposed in all projects under the /info endpoint, which also includes Kubernetes context information.

4.2. Data Models

The platform's data models are defined as Pydantic v2 models. With Pydantic, defining models is extremely easy and understandable. Be aware that there is an inconsistency about the schemas with their locations in the codebases. In the beginning, a single schemas.py file was used to contain model definitions, but at the end of the project, a newer convention was used within the models folder for better separation; the schemas.py was not yet migrated over.

4.3. Core API Implementation

Repository: https://git.rifstar.net/tsc/tavlink-core

The Core API, as its name suggests, is the core component of Távlink. This service was a major focus during development. Most time was spent on improving this service and introducing new features here.

4.4. Ingestion Server Implementation

Repository: https://git.rifstar.net/tsc/tavlink-ingestion-server

The Ingestion Server is the other major component of this project, as it is an edge for data registration and ingestion. While the codebase looks normal sized due to many dependencies, its contents are quite small, with 1 single endpoint serving as the data ingestion edge.

4.5. Websocket Gateway Implementation

Repository (shared with the core API): https://git.rifstar.net/tsc/tavlink-core

The Websocket is a small independent component making real-time telemetry available to the user interface. This technology was explored with some AI assistance to kickstart it and better understand it. After getting to know websocket better, it turned out to be extremely useful, and it certainly gave inspiration to other projects alongside Dragonfly.

4.6. Web Panel

Repository: https://git.rifstar.net/tsc/tavlink-web

While the Web Panel would be considered one of the most important components, the least amount of time was spent on this Frontend, mainly on code approval. A user interface is needed to pilot the project and to do manual testing with ease. AI was used to fill in the gap and generate the project; most of the time on this project was spent on reviewing and approving the generated code. This component is important, and once the backend is solid enough, better care will arrive for this Frontend.

4.7. AI-assisted Implementation across projects

AI-assisted software development, and in this context, it was used as an editing support tool. Windsurf Editor was used alongside Cascade, which is a context-aware auto-completer coding tool within Windsurf.

AI was used for a variety of tasks, including code generation, explanation, editing, review and debugging. Different models were selected depending on the complexity of the task. For example, more heavy models such as Claude Opus 4.6 were used for tasks requiring better accuracy and stronger reasoning, while lighter models such as Windsurf's SWE 1.5 and 1.6 were used for simpler and more routine tasks. In more complex cases, prompts were carefully designed and provided with extensive project context to improve the accuracy of the output.

All code in the codebase that was pushed into the repositories was reviewed and approved by the developer. While AI was used to support development, speed up implementation and fill in development gaps, the ultimate responsibility remained with the developer.

The level of AI use was not the same across all repositories. The core API and the ingestion API were written more heavily by the developer, while other components, such as the web interface, relied more heavily on AI-generated code to accelerate and pilot the project more quickly.

For transparency, AI-powered prompts are referenced in the "ai_ref" folder in the relevant codebases where the request was initiated.

In this project, it should be seen as a practical aid. It improved speed, supported implementation, helped with problem solving and introduced new technologies. Of course, human oversight remained in place to ensure the correctness of the generated output.

5. Features

5.1. Device Management

A Device in this context refers to any monitorable object that reports in telemetry data. The settings of a device can be managed through the Core API's endpoints.

5.1.1. Device Registration

Devices can be registered using 2 major methods:

- by creating it manually within the user interface or through the Core API
- by zero-touch provisioning, where the device registers itself through the Core API

When manually registering, the user must also apply a collection profile, which will act as the device's configuration on telemetry. When using provisional tokens, the token can be set to use an exact collection profile, so when the device is doing the self-registration, the system automatically applies the configuration. Provisional tokens are reusable, making it easy for large-scale onboarding.

5.1.2. Device Organisation & Management

Devices can be organised into groups, where the user can apply collection profiles in bulk in order to save time if the devices act the same way and in need of a configuration update. Additional tags can also be appended to them to further distinguish them. Each device also has extra properties like location info, allowing for further organisation of monitored objects.

The device listing endpoint supports filters, pagination, and even sorting. It returns a list of devices as an array and total count information. Deleting a device also deletes all associated configurations and settings except for API keys.

5.2. Flexible Configuration System

The configuration system is one of the most important features of the platform. These configurations define what and how to measure for which device, providing a level of flexibility to accommodate various use-cases. While many definitions are symbolic, the system can be expanded later on, further improving the platform.

5.2.1. Metric Templates

A metric template is a reusable configuration of a single metric. It has:

- a slug which acts as a reference point (for example: cpu-usage)
- a type which defines how the data is stored (float, int, bool or string)
- a unit used to display the measurement unit (%, km, etc.)
- a scaling to transform the measurement by scaling it

Templates are versioned; an updated version of an existing metric template will get a new, incremented version number and will get the “is_latest” flag.

5.2.2. Collection Profiles

A collection profile is a top level configuration template and acts as a policy. It includes:

- a sampling interval which defines how often the data gets generated (currently not validated)
- a flush interval which defines how often the device sends in data (currently not validated)
- a list of metric templates alongside a storage policy (like raw or drop)

Profiles are versioned with the same mechanism as templates.

5.2.3. Device Configurations

A device configuration is the direct configuration of a device, which can be created by either applying a configuration profile or by manually defining the profile. Each configuration change increments the device’s config version. This is done to protect previous data for resolving, if needed.

For example, let’s say a device was configured to send in 4 metric types, but the device got an update to send even 5 metric types. If the device tries to send all 5 in, the newer unrecognised metric will be rejected by the Ingestion Server on validation, and the configuration needs an update. Once the new metric is added and applied, the system will accept all the 5 metrics, but that creates an inconsistency in the history of the data, as from a certain point in time, an extra metric appeared. In order to handle any change without issues, versioning helps to identify when and what configuration was active for the device, allowing for the reconstruction of historical data even when it’s inconsistent. When a config is changed, there is a fast invalidation process, making the propagation fast.

5.3. Telemetry Ingestion, Storage and Retrieval

5.3.1. Push-Based Ingestion

The platform uses a push-based ingestion model where the devices initiate a POST request to the Ingestion Servers. This model was chosen because it scales easily, and it also allows devices to buffer data locally and send it in batches. This buffer can also help on connection issues as the data can be backfilled into the database when the connection restores. At the end, what gets ingested and how depends on the configuration of the device. For example, if the metric is specified as drop, that means the data will be accepted, but won't be ingested into MongoDB; it will still show up in the control room as near-real-time data. In the future, even this real-time data will be configurable. The metric type is also taken seriously; if the provided datapoint doesn't match the type of the metric (string vs float), the datapoint will be rejected.

5.3.2. MongoDB Time-Series

Telemetry data is stored in a Time-Series collection called ts_metrics. Each data point is configured to include a timestamp, config version and a value. Datapoints are packaged into buckets, where a bucket can hold many datapoints. Each bucket has metadata which makes the bucket unique. For each device, the metadata includes the device ID, organisation ID, and metric slug. Granularity is set to minutes, which tells MongoDB that a bucket can only hold a maximum time-range value of 1 day (but a new bucket is used regardless when the maximum of 1000 datapoints is reached within a bucket). Each bucket uses delta compression to reduce storage. Instead of storing the values themselves directly, delta compression uses the previous value as a reference point and only saves the difference if there is any. That means when the change is little or non-existent, storage is minimal. Only the first value in the bucket is stored as full data; the rest is just changes compared to the previous one. In order for delta compression to work, the type of data must remain consistent, and that's why the ingestion servers validate metric type.

5.4. Near Real-Time Monitoring

5.4.1. Receive Data

The platform streams live telemetry to connected clients using the Websocket Gateway. When the user opens the control panel and selects the desired devices, the frontend establishes a WSS connection. During that connection, telemetry is almost directly delivered to the user as fast as possible.

5.4.2. Cache Burst

The Websocket and Dragonfly Pub/Sub are capable of providing new information, therefore it will not be able to load in previous information. To solve this, the recent values are also stored in Dragonfly as a sorted set with a 1-minute TTL. This cache burst allows the display of the most recent data to kickstart real-time monitoring with previous data context. In the future, both real-time data and cache burst will be configurable within the device config for customisability.

5.5. API Key and Provisioning System

5.5.1. API Keys

In order for devices to communicate with the ingestion servers, API keys are used as authentication. These keys can be either device-, bridge- or organisation-scoped. Keys can have permissions as well, such as “ingest” for ingestion capabilities. When empty, it defaults to all-access. Keys can be created, deactivated, reactivated, rotated and deleted. Rotation has a built-in grace period, which can be specified by the user. Keys are generated as random strings and are outputted only once upon creation. Only its hash (SHA-256) value is saved in the database for lookups. Invalidation is done as fast as the keys are stored in the Dragonfly cache, and invalidation is triggered through the Pub/Sub mechanism.

5.5.2. Provisioning Tokens

The system enables automated self-registration using provisional tokens. This token can be set to have max uses, expiry time and also to automatically apply certain configurations and settings to the device, such as a collection profile or even append it to a group. When the device supports this mechanism, using this token, the device can self-register itself (it can also name and describe itself, which will show up in the user interface, or the user can specify that as well). During registration, the device gets an API key and saves it in its credentials. The system autoapplies all preapplied configurations and settings. Using the API key, the device can start ingesting data.

5.6. Multi-Tenancy and Access Control

The platform is designed as a multi-tenant system where each organisation’s data is logically isolated but shared on the same infrastructure. All organisation-related data carries the organisation ID to provide logical separation. When a user makes requests in Távlink, they can only do so with an organisation selected. Due to how identity and authentication are implemented, it’s fairly easy to add access controls. Identity and organisations are managed by Authbridge. When the user logs in, it receives a session and an access token.

The session token is long-lived and can be used to refresh the access token. The general access token (JWT) is stateless, includes roles, scopes, permissions and groups, is signed by Authbridge and is short lived. Távlink can validate if the access token is valid by using the public key of Authbridge. Once the token is marked as valid, its content can be extracted, which includes general identity and access control details, but that still doesn’t include organisational information.

When the user is part of an organisation within Authbridge, it can request for an organisation-scoped access token which includes the roles and permissions within the organisation. When a user has access in more than 1 organisation, they can switch in-between (in each switch, a new organisation-scoped access token is requested, which allows access to resources within the organisation). Currently, access control is limited within Távlink, as all feature access is tied to the owner role. It is future work to implement a more fine-grained access control with user management, but the framework is ready for it.

6. Deployment, Infrastructure and DevOps

6.1. Infrastructure Overview

Távlink prototype is deployed on a self-managed Kubernetes cluster hosted on two dedicated bare-metal servers. The cluster consists of one control plane node and one worker node. This is only an experimental cluster with simple configurations and settings. Self-hosting was chosen to have full control over the underlying infrastructure. Renting full servers can be expensive, but when it is used to deploy many services on them, it becomes financially viable. It's also very rewarding to set up and manage. It provides a completely different skill set. Of course, on the other hand, it is operational overhead; maintenance can be tedious (upgrades, backups, etc.). Setting up and learning Kubernetes is also quite difficult and time-consuming. Just an experimental cluster setup can take around 1 week for a single person who has just started learning it as many things can go wrong with it.

6.2. Containerisation

Each service has its own Dockerfile in order to build a Docker Image that can be used for deploying the service into Kubernetes using Containerd. All images are pushed to a private self-hosted Docker registry at registry.rifstar.net so Kubernetes can pull the images when needed.

6.3. Kubernetes Configuration

Each codebase has a k8s folder containing Kubernetes manifests. The base includes shared base configurations, while the overlays include environment (like prototype) specific information. Configurations were borrowed from Authbridge in this context, but it was radically changed to make it much more simpler.

6.4. Networking and Ingress

Currently, traffic is routed directly onto the worker node, but in the future, load balancers will be used. The worker node has an NGINX Ingress controller where the TLS termination happens. Each service has DNS records set up to lead to the correct service. TLS Certificates are deployed automatically within K8S by the cert-manager service. For prototype deployments, only staging (fake) certificates are used.

6.5. CI/CD Pipeline

The platform uses Forgejo for source code management and CI/CD. Each repository contains a .forgejo/workflows/ directory with YAML workflow tasks. These workflows can be used for many tasks like automated testing, but currently, in this project context, it is only used for deployment into the Kubernetes cluster.

The project includes the same version management across all codebases borrowed from Authbridge for consistency. With the project files and environment context, the script builds the version info, which can be inspected at /info endpoints.

6.6. Deployment Environments

The adjusted level of deployment environments contains a minimum of 3 stages:

1. Prototype: running on a small experimental cluster, primarily used for testing, with no load balancing at the moment.
2. Staging: a new pre-production level, attempts to replicate a full production environment, the final stage of testing
3. Production: business operations

While the original plans were similar, there was a need for a clean new level (staging) for separating certain test cases. However, the current project stage stops at Prototype, so this is for future reference.

7. Discussion, Limitations and Future Work

7.1. Discussion

Távlink was developed with the ambition to create a comprehensive IoT telemetry platform, and the scope certainly exceeded what could be fully completed within the allocated project timeline. The result is a system where the ingestion, data storage, retrieval and real time telemetry are functional, but there is still room for improvement. Of course, there are several features which are not yet implemented or are in a very early experimental stage. The development will continue regardless and will certainly accelerate afterwards due to the reduced workload of academic projects. Building such systems, while challenging, is very enjoyable with the amount of creativity and problem solving needed to address issues that are constantly popping up. These experiences are extremely valuable, and there is much to learn from the process and outcome of this project. Making a system as large as this takes time, knowledge, and practice. There must be some level of naivety to start a project like Távlink, as it doesn't feel like a 1 person project after getting in-depth with the implementation. It all looks easy from the outside as a system that just collects and displays data, but Távlink is much more than that, and the architecture shows it. Its current stage is a first step towards a secure, scalable and flexible production-grade telemetry platform that might be even commercialised as a fully capable Software-as-a-Service platform later on.

7.2. Limitations & Future Work

The platform has a significant number of limitations that must be addressed before even considering the deployment to the internet, even in a pre-production environment.

7.2.1. Testing and Quality Assurance

The platform lacking proper automated testing, is the most critical limitation of the project. Code changes are validated mostly through manual testing, and there is only a little unit testing done. The system consists of many components already, and it will just continue growing, meaning it is already difficult to spot issues and bugs across services, and it will continue to become worse and worse over time. Therefore, future work will start by addressing this issue by implementing various testing mechanisms. This limitation is one of the primary reasons why the code has not been deployed to the open internet, as exposing the platform would be irresponsible. Besides, the first stage of deployment into K8S will always remain restricted for security concerns.

Future quality assurance must cover all kinds of automated testing, including unit, feature, stress, security, performance and penetration testing on all components. And of course, the resilience of the system as a whole must be tested as well to make sure the system handles correctly when components malfunction, degrade or fail. All stages must have integration testing for deployments to make sure the code getting deployed meets certain requirements. Writing those tests and exhausting all possibilities is not a small task; that's why it was out of scope. This is the next step that has to be taken before making the project even larger, as it would become unmanageable. What can be tested should be tested. If that is followed, 100% code coverage is on the horizon!

Manual evaluation and approval are a requirement before deploying into production. New code has to be reviewed, especially if it is AI-generated. Systems must also always be ready to roll back in case something is not functioning as expected.

7.2.2. Alarm Engine and Notification Router

While there is experimental code for alarm and notification features, neither is functioning as expected, and due to the lack of testing, it's difficult to move forward this time. Once quality is assured, these systems can be properly implemented to provide a comprehensive automation for telemetry evaluation with possible follow up actions supporting notifications on various platforms.

7.2.3. Bridging Software and OT Site Support

The original problem Távlink tries to solve is collecting telemetry data from critical infrastructure environments (see the first Távlink paper). It was not implemented, as first, there is a need for a platform that collects the data, but once the platform is ready to serve, bridging from Operational Technology might become a reality. A bridging unit would collect data directly from devices such as PLCs using Modbus or other protocols. This is a completely separate problem from the platform and has to be robust, secure and fault tolerant. Using just TLS for secure communications between the bridge and the platform isn't enough; rather, mTLS, a defence-in-depth approach and heavy firewalling is needed as baselines.

7.2.4. Web Control Panel Interface

As the current web control panel was mostly AI-generated, while it is being functional, it has to be evaluated carefully, and almost certainly there will be a need to improve or even rewrite if needed. There are still some features that are present in the Core API, but are not displayed on the interface. Of course, for humans, these kinds of interfaces are the most important. Making a user-friendly, functional, accessible and fast frontend should be a top priority once the backend is robust and secure enough. While it was not a major focus, it will be one of the most important components of the project once it gets to Production.

7.2.5. Additional Client Interfaces

The platform is currently only accessible from the Web Interface or directly via API calls. To make the service production ready, customer needs have to be assessed and evaluated. If there is a need for mobile applications, then that must be implemented. Whether it's a native desktop application or a Linux command line interface tool, the system should exhaust customer needs if realistic.

7.2.6. Data Services & Archival

MongoDB and Dragonfly are each deployed as single instances without any replication, failover or sharding. In order to scale this project in the future and improve availability and fault-tolerance, these current short-comings have to be addressed. Currently, the data has a TTL of 30 days, meaning it simply gets deleted. For the future, if the user requires archival, data reaching the end of TTL should be exported to some kind of archive using services like S3.

7.2.7. Rollup Aggregation

While the code resembles some level of support for rollup collections, it is far from a functional feature. This is an important data optimisation approach which allows for reduced data storage while keeping the core most important information, such as statistics like minimum, maximum, mean and average values in a specified time interval.

7.2.8. Rate Limiting & Misuse Cases

Limiting API usage to reduce overload and abuse of the underlying infrastructure is a very important enforcement that is needed for production deployment. This is a requirement before publicly deploying openly to the internet. Of course, other misuse cases have to be identified as well and addressed appropriately.

7.2.9. Audit and Logging capabilities

Sensitive and important actions must be logged, possibly in a remote system, to ensure authenticity and integrity of the gathered logs, especially security related ones. In case an audit is needed, the data must be available for evaluation, and the platform must accommodate this. As the system is multi-tenant, users and organisations should have their own logging with easy traceability for the origins of actions.

7.2.10. Pull-Based Monitoring

The current system is limited to push-based telemetry monitoring, but there are various other methods of monitoring for different use-cases. Customer needs have to be carefully evaluated, and support for different monitoring methods, such as pull-based monitoring, should be implemented.

7.2.11. Knowledge Base & Customer Support

As Távlink aims to be a Software-as-a-Service platform, supporting documentation and materials are crucial to provide relevant and up-to-date guidelines to customers. And in case it’s needed, an easy-to-access and reliable customer support is needed. Such a system could be implemented into the platform interface itself. Of course, the other side of the support agency has to be solid as well to manage customer cases efficiently and quickly.

7.2.12. Dependency on Other Projects

The platform currently depends on two proprietary services that are also in the Prototype stage of development, which means that Távlink cannot be deployed to the internet openly until those services reach production level. And of course, their capabilities shape Távlink’s capabilities as well.

7.2.13. SaaS Features

While implementing most of the other features solves the limitations of Távlink, it doesn’t make it immediately commercially ready. As a last step, general SaaS features must be implemented, such as usage metering, billing and transaction processing. Of course, legalities and compliance are also crucial to start such a platform.

7.2.14. Deployment

The platform can currently only run in a local and severely restricted prototype environment. When the project is ready, the service must deploy to a proper pre-production environment (staging) that replicates an exact production environment.

7.2.15. Multi-Region and Multi-Zone Expansion

The prototype deployment is currently deployed in a single location, but Távlink is planned to meet global needs. To ensure low latency, compliance and other needs, covering different regions is a long-term goal to achieve. Of course, deploying in different availability zones also improves availability and fault-tolerance, which are core values in a telemetry platform.

8. Conclusion

Távlink has successfully reached a very early-stage prototype level. The results so far show that this design, architecture and tech stack, while it might need some adjustment and improvement, is very much suitable and is a solid base for future work. While the foundations are good, the majority of tasks and goals are still ahead. The current stage of the project is only the beginning of the journey. The work just begins now. The potential is here to build upon this project and expand Távlink into a versatile, secure, scalable and flexible monitoring platform for both IT and OT environments.

9. References

Artificial Intelligence is referenced in the repositories in the ai_ref folder using markdown format.

[1] S. Ramírez, “FastAPI,” FastAPI Documentation. Available: https://fastapi.tiangolo.com/. Accessed: Apr. 23, 2026.

[2] I. Khan, “Uvicorn and Gunicorn for FastAPI,” Medium, 2024. Available: https://medium.com/@toimrank/uvicorn-for-fastapi-00a1ddb5ca4d. Accessed: Apr. 23, 2026.

[3] DragonflyDB, “Dragonfly: An In-Memory Data Store without Limits.” Available: https://www.dragonflydb.io/. Accessed: Apr. 25, 2026.

[4] Mozilla Developer Network, “WebSocket API (WebSockets),” MDN Web Docs. Available: https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API. Accessed: Apr. 25, 2026.

[5] OpenJS Foundation, “Node.js — Run JavaScript Everywhere.” Available: https://nodejs.org/en. Accessed: Apr. 27, 2026.

[6] Vercel, “Next.js by Vercel — The React Framework.” Available: https://nextjs.org/. Accessed: Apr. 27, 2026.

[7] Meta Open Source, “React.” Available: https://react.dev/. Accessed: Apr. 27, 2026.