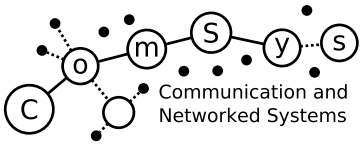
FACULTY OF
COMPUTER SCIENCE



Communication and Networked Systems

Bachelor Thesis

Data Visualization for IoT

Shashank Shorya

Matr. 235333

Supervisor: Prof. Dr. rer. nat. Mesut Güneş
Assisting Supervisor: Sebastian Bläsing

Institute for Intelligent Cooperating Systems, Otto-von-Guericke-University Magdeburg

15. September 2025

Abstract

The Internet of Things (IoT) is rapidly expanding, with forecasts predicting around 80 zettabytes of IoT data by 2025 [1]. Extracting value from this data requires not only reliable acquisition but also effective real-time visualization, particularly in measurement-driven domains such as FPGA-based data acquisition. Existing visualization platforms like Grafana and Node-RED offer powerful dashboards and integration capabilities [2, 3]. However, they typically depend on external databases or middleware and are not designed for direct, low-latency access to hardware buffers. This makes them unsuitable for embedded System-on-Chip (SoC) environments with constrained resources.

The goal of this thesis is to design and implement a lightweight, web-based visualization system for a Linux-based SoC with FPGA integration. The SoC provides measurement data via kernel modules that expose register access and a ring buffer, a common pattern for efficient data exchange between kernel and user space. The thesis contribution consists of two components: a backend, implemented in Rust using the Actix-web framework, which communicates with the kernel modules via IOCTL calls, parses data blocks, and exposes a RESTful API; and a frontend, implemented in HTML, CSS, and JavaScript, which cyclically retrieves the data and renders it as configurable XY curve plots.

The developed system demonstrates an end-to-end pipeline from FPGA hardware to browser-based visualization. It achieves real-time plotting with minimal overhead, requires no additional middleware, and is portable to other Linux-based SoCs. Compared to established frameworks, the presented solution is optimized for embedded use cases, offering direct hardware integration and lightweight deployment.

While the evaluation is limited to functional validation and a literature-based comparison, the results confirm that tailored implementations can outperform generic dashboards in resource-constrained environments. Future extensions may include advanced visualization methods such as downsampling algorithms (e.g., Largest-Triangle-Three-Buckets) for handling high-frequency signals [4], WebSocket-based streaming for lower latency, and integration with established platforms for broader applicability.

Contents

List of Figures

List of Tables

Listings

Acronyms

API application programming interface. 2, 9

IOCTL Input/Output Control. 1, 2, 9

IoT Internet of Things. 1, 3–6, 8, 9

SoC System on Chip. v, 1–3, 5, 7, 9, 10, 12–14

CHAPTER 1

Introduction

In the specific system context of this thesis, a Linux-based SoC with FPGA integration produces measurement data through kernel modules. These modules provide access to FPGA registers and a FIFO for sampled data. At present, visualizing such data requires custom development efforts or heavyweight frameworks that add unnecessary overhead. Existing tools do not directly support Input/Output Control (IOCTL)-based device access or efficient real-time plotting of raw multi-channel measurement data.

Thus, there is a clear need for a lightweight, platform-independent visualization solution that runs directly on the SoC, requires no additional software installation for end users, and integrates seamlessly with the provided kernel modules

1.1 Motivation

The proliferation of Internet of Things (IoT) devices has created unprecedented amounts of sensor and measurement data, with projections estimating around 80 zettabytes of IoT data by 2025 [1]. In domains such as industrial automation, smart infrastructure, and embedded systems, this data is often generated at high sampling rates and requires immediate processing and visualization to be useful. Visualization is not only essential for monitoring system behavior but also for debugging, evaluation, and decision-making.

Conventional IoT visualization platforms such as Grafana [2] and Node-RED [3] offer extensive features for time-series dashboards, data integration, and prototyping. However, these frameworks are optimized for enterprise or prototyping contexts and typically require external databases, brokers, or middleware. They are therefore poorly suited for resource-constrained embedded SoCs that need direct, low-latency access to hardware-level measurement streams.

1.2 Thesis Structure

The next chapter introduces related work in the field of IoT data visualization. In Chapter 2, frameworks such as Grafana and Node-RED are analyzed with respect to their functionality,

system requirements, and applicability in embedded contexts. While these solutions are powerful in enterprise and prototyping environments, the discussion shows that they are not suited for direct integration with FPGA-based SoCs, which motivates the need for the lightweight approach pursued in this thesis.

Building on this analysis, Chapter 3 presents the core contribution of the thesis: the development of a backend and frontend tailored to the HH-AMV architecture. The backend, implemented in Rust, communicates with the provided kernel modules via IOCTL to configure registers and retrieve sampled measurement data from the ring buffer. It processes these data blocks and exposes them through a RESTful application programming interface (API). The frontend, implemented in HTML, CSS, and JavaScript, cyclically queries this API, visualizes the multi-channel measurement data as XY plots, and allows configuration of visualization parameters such as sampling rate and scaling.

Chapter 4 then evaluates the developed system. This includes a functional validation of the complete pipeline from FPGA data acquisition to browser-based visualization, a performance assessment considering latency, throughput, and resource usage on the SoC, and a comparative analysis against existing solutions based on a literature review. The evaluation thereby demonstrates both the strengths and the current limitations of the implementation.

Finally, Chapter 5 summarizes the findings of the thesis and reflects on the overall contribution. It also outlines directions for future work, such as the integration of advanced downsampling techniques to handle higher data rates, the use of WebSockets to further reduce latency, and potential interoperability with external frameworks to broaden the applicability of the developed system.

CHAPTER 2

Related Work

2.1 The landscape of IoT data visualization

The IoT is growing rapidly; industry projections indicate that over 75 billion devices will be connected by 2025 and will generate more than 79 zettabytes of data annually [1]. Such massive telemetry streams pose unique challenges for storage, processing and, critically, visualization. Visualization enables humans to interpret complex patterns, spot anomalies and make timely decisions. In IoT contexts, dashboards must operate in real time, support diverse data types and remain usable on resource-constrained devices.

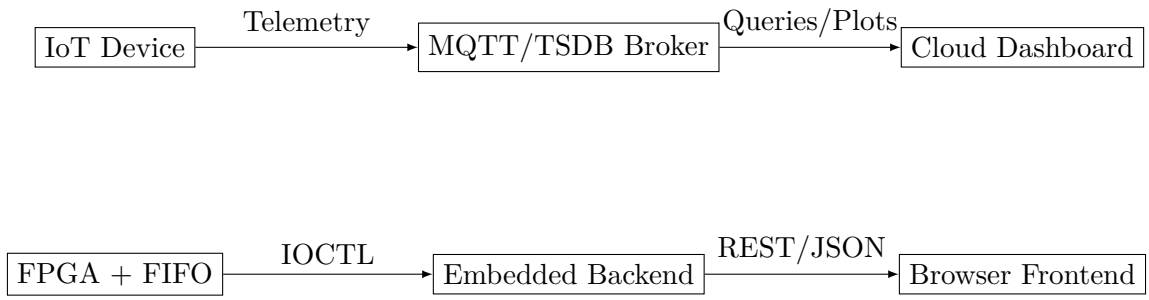
At a high level, IoT visualization solutions fall into two categories: *general-purpose dashboards* designed for enterprise systems and *specialized frameworks* built for specific hardware or deployment constraints. General-purpose tools, such as Grafana and Kibana, assume that telemetry flows into a time-series database (TSDB) or log store; the dashboard queries that backend to produce plots, alert panels and heatmaps. Specialized frameworks, like Node-RED or OpenHAB [5], focus on rapid prototyping or direct hardware integration, often trading advanced analytics for simplicity or low overhead. Tools such as FHEM highlight their strong use in home automation scenarios, where lightweight dashboards and device orchestration are prioritized [6]. At the diagnostic level, Wireshark [7] provides packet-level inspection and visualization of IoT protocols, though it is not a sensor-oriented dashboard in the strict sense.

For ARM-based SoCs, compatibility is not just a preference but a strict requirement. Such devices typically operate with limited CPU resources and memory budgets, often well below 2 GB of RAM, and rely on Linux distributions compiled for the ARM instruction set. Therefore, visualization frameworks intended for this class of hardware must fulfill several requirements: they must be lightweight enough to run without dedicated database clusters, provide binaries or source code that can be cross-compiled for ARM Linux, and avoid dependencies on heavy middleware such as Elasticsearch or Cassandra. Instead, practical solutions rely on small-footprint backends and embedded HTTP servers, combined with browser-based frontends. In this context, complex enterprise dashboards with external TSDBs are rarely feasible, whereas lightweight frameworks that compile and execute natively on the SoC offer a realistic path forward.

Table 2.1: Comparison of IoT visualization frameworks.

Tool	Data rate capacity	Deployment model	ARM feasibility
Grafana	$> 10^6$ points/s (with TSDB)	Requires TSDB + plugins	Heavy on small ARM SoCs
Kibana	10^5–10^6 log events/s	ElasticSearch backend	Not practical on embedded ARM
ThingsBoard	$\sim 10^4$ points/s	DB + rule engine	Runs on ARM but resource-heavy
Node-RED	Hundreds–10^3/s	Event flows + add-ons	Well-suited to ARM SBCs
OpenHAB/FHEM	Hundreds/s	Home automation integration	Designed for Raspberry Pi
Wireshark	Network-level pcap, not sensors	Desktop analysis tool	Not suitable for ARM SoC dashboards

This chapter reviews both categories and highlights research on transport protocols, frontend libraries and downsampling algorithms relevant to high-rate sensor streams.



(**Top**) Cloud dashboard with TSDB; (**Bottom**) Embedded SoC dashboard.

Figure 2.1: Comparison of data paths for IoT visualization: cloud-based vs. embedded ARM SoC.

2.1.1 General-purpose dashboards

Grafana and the observability ecosystem. Grafana is one of the most widely adopted open-source dashboards for time-series data. It was originally developed for monitoring infrastructure but has since expanded to IoT and industrial contexts. Grafana does not store data itself; instead it reads from various backends such as Prometheus, InfluxDB, Elasticsearch and TimescaleDB [8]. Typical deployments therefore involve a collector (e.g., Telegraf) that ingests metrics from devices and writes them to a TSDB; Grafana queries the TSDB via its plug-in system and displays results on configurable panels. This architecture enables

scalability and flexibility but introduces dependencies on external storage and message brokers. The overhead of maintaining a TSDB and query engine can be acceptable for cloud or data-center applications but is impractical on a small SoC that streams high-frequency measurements directly from a ring buffer.

Kibana and Elasticsearch. Kibana is the visualization front end for the Elastic stack [9]. It provides dashboards for logs and metrics, with features such as Lens (a visual builder), TSVB (Time Series Visual Builder) and anomaly detection. Kibana presumes that data are indexed in Elasticsearch or Elastic's time-series data stream (TSDS) and that queries return aggregated buckets for display [10]. Elastic dashboards excel in multi-metric analyses and long-term observability but require a heavy backend and do not integrate with hardware registers or ring buffers.

ThingsBoard. ThingsBoard is an IoT platform combining device management, rule engine and visualization. Its architecture supports telemetry storage in SQL (TimescaleDB) or NoSQL (Cassandra) and exposes APIs for historical queries, downsampling and WebSocket subscriptions [11]. Clients can build dashboards from a library of widgets, subscribe to real-time updates and configure alarms. ThingsBoard demonstrates how dashboards can integrate stream and batch processing, but like Grafana it relies on an intermediate database and rule engine, which are unsuited to embedded, single-board deployments where no external TSDB exists.

OpenHAB and FHEM. OpenHAB [5]and FHEM [6] represent another class of general-purpose IoT dashboards, with a strong focus on home automation. Both frameworks integrate a wide variety of protocols, including MQTT, HTTP and Zigbee, and provide user-facing dashboards for monitoring and controlling devices such as heating, lighting and security systems. They are lightweight enough to run on ARM-based platforms like Raspberry Pi, which makes them attractive for embedded contexts. However, their visualization capabilities are tailored toward discrete device states and home automation rules rather than continuous high-frequency sensor streams. As such, while they demonstrate how open-source dashboards can operate in constrained environments, they are not designed to directly interface with raw hardware FIFOs or register-level data paths.

2.1.2 Low-code and prototyping frameworks

Node-RED. Node-RED is a flow-based development tool built on Node.js. It allows users to connect devices, services and APIs via a visual editor; flows consist of nodes representing inputs, transformations and outputs. Node-RED was designed to democratize IoT programming and runs on low-cost hardware such as Raspberry Pi [12]. It is easy to set up and supports a range of protocols (MQTT, HTTP, WebSocket), making it attractive for quick prototypes. However, Node-RED flows are single-threaded and rely on Node.js event loops; they are not optimized for high-throughput raw data from hardware buffers. Furthermore, the `node-red-dashboard` add-on (for live charts and gauges) is based on an outdated AngularJS framework and has been deprecated [13], prompting users to migrate to alternative dashboard nodes or external UI frameworks.

Low-code edge dashboards. Several research and industrial projects present low-code dashboards specifically for edge devices. For example, the AccessiDashboard prototype focuses on making IoT data more accessible through semantic markup and user-friendly interactions [14]. Other projects combine MQTT telemetry with simple WebSocket views for industrial monitoring or building automation [15]. These efforts show that a lightweight server and browser-based UI can run on single-board computers, but they typically assume data decoupled from hrdware via a broker and do not address direct access to ring buffers or registers.

2.1.3 Transport protocols and streaming channels

IoT visualization systems rely on two transport channels: the uplink from devices to the backend and the downlink from backend to browser or dashboard.

MQTT for device uplink. MQTT is a lightweight publish/subscribe protocol standardized by OASIS. It features small transport overhead and configurable quality-of-service (QoS) levels for reliable message delivery [16]. Brokers can retain last-will messages and handle disconnections gracefully. MQTT is widely used for device-to-cloud telemetry and is supported natively by Node-RED, Grafana plugins and ThingsBoard. However, MQTT is not optimized for streaming large binary payloads such as 32-bit words from a ring buffer; rather it sends text or small binary values at moderate rates.

WebSockets for dashboard downlink. WebSockets provide a full-duplex channel over TCP established via an HTTP Upgrade handshake [17]. Once open, data can flow in both directions without the overhead of HTTP request/response cycles. WebSockets are ideal for pushing real-time updates to browsers; Grafana, ThingsBoard and Node-RED all offer WebSocket endpoints for streaming live data. Using a WebSocket connection with binary frames, a backend can send ring buffer samples directly to the browser. For resource-constrained devices, however, WebSockets require a persistent connection and memory buffers; careful management is needed to avoid drops or high latency. *A practical limitation is that binary WebSocket payloads are more difficult to debug than text-based formats, since raw frames cannot be easily inspected with developer tools or logs without additional decoding.* For this reason, many systems adopt JSON during development and switch to binary encoding only when performance requirements demand it.

2.1.4 Time-series storage and downsampling

Time-series databases. InfluxDB, TimescaleDB (a PostgreSQL extension) and TDengine are purpose-built TSDBs optimised for high write throughput and time-based queries [18, 19, 20]. InfluxData's Telegraf agent collects metrics from devices and writes them to InfluxDB; the integration is seamless, and Grafana can query the TSDB for plots. TimescaleDB adds time-series functionality to PostgreSQL and supports downsampling and retention policies. TDengine promises a high-performance, cloud-native TSDB with built-in caching and clustering [20]. Yet, these systems all assume that telemetry is ingested into a database via connectors or agents. In the HH-AMV architecture, however, this assumption does

not hold: each sensor channel is implemented in hardware and its data is exposed either through registers or a FIFO stream. A proper device driver provides user-space access to these hardware interfaces, eliminating the need for intermediate storage layers. This means that the visualization backend interacts directly with the ring buffer on the SoC rather than querying a database.

Downsampling algorithms. When visualizing high-frequency signals on a limited-resolution display, downsampling is essential to avoid clutter while preserving salient features. Simple methods such as uniform sampling (taking every k-th point) or averaging hide outliers and transient events. The Largest-Triangle-Three-Buckets (LTTB) algorithm addresses this by dividing data into buckets and selecting points that form the largest area triangles with adjacent points [4]. LTTB retains the global shape of the series and highlights peaks and valleys; it is used in many dashboards (the Plotly-Resampler library extends this concept [21]). Extensions such as MinMaxLTTB pre-select local extrema before applying LTTB [22].

2.1.5 Frontend charting libraries

Chart.js. Chart.js is a canvas-based library known for its simplicity and small footprint. Recent versions support mixing chart types and advanced features such as stacked axes and time scales [23]. A community plugin, `chartjs-plugin-streaming`, adds a real-time axis and scheduled refresh hooks so that new points can be appended while old points slide off the chart [24]. Chart.js is suitable for small to medium data sets but may struggle with hundreds of thousands of points; downsampling or progressive rendering may be necessary.

Highcharts. Highcharts is an SVG-based commercial library that provides interactive charting with features like zooming and panning and a large set of modules. It is widely used in enterprise dashboards but requires a license for commercial use and has a larger footprint than Chart.js [25].

ECharts and uPlot. Apache ECharts (formerly known as echarts.js) offers incremental rendering and supports streaming data [26]. It supports complex chart combinations and interactive features like data zoom. uPlot is a minimal library optimized for speed; it can render charts with hundreds of thousands of points and handle frequent updates without jank [27]. For an embedded dashboard, uPlot or a similar lightweight library may be appropriate because of its small size and high performance.

Gnuplot. Gnuplot [28]is a long-standing command-line driven plotting tool frequently used in scientific and engineering contexts. Unlike browser-based charting libraries, Gnuplot generates static or scripted figures directly from numerical output files (e.g., CSV or binary logs). Its lightweight nature and lack of external dependencies make it suitable for embedded development workflows, where measurements are captured on-device and plotted offline. In the HH-AMV development process, for instance, Gnuplot scripts were used to visualize ring buffer streams exported as `stream.csv`, enabling rapid inspection of signal quality without requiring a full web-based dashboard.

2.1.6 Academic contributions

Surveys and frameworks. Protopsaltis *et al.* survey IoT data visualization techniques and classify tools according to domains and user roles; they emphasise real-time visual analytics and the importance of aligning visualization with decision processes [29]. Shao *et al.* propose the IoT-based Efficient Data Visualization Framework (IoT-EDVF), which integrates data preparation, analytics and visualization for business intelligence contexts. Their experimental results show reduced delays and improved performance compared to baseline dashboards [30]. These works illustrate general principles but do not address direct hardware integration or ring buffer streaming.

Since direct integration with heterogeneous IoT devices is often impractical due to diverse protocols and hardware-specific interfaces, a hardware abstraction layer becomes necessary. Such an abstraction enables higher-level frameworks to interact with devices through standardized APIs, decoupling visualization and analytics modules from low-level device complexity and ensuring portability across different IoT hardware platforms.

Streaming visualization for resource-constrained devices. Research on edge and fog computing emphasises local analytics to reduce latency and network load. Gomes *et al.* developed STEAM++, a framework for processing and visualizing events on embedded devices; they report memory consumption below 500 kB and CPU usage around 1 % while processing nearly 240 packets per second [31]. Such results demonstrate that real-time dashboards are feasible on constrained hardware. Other studies incorporate streaming analytics and downsampling within the browser: for example, VanDerDonckt *et al.* adapt LTTB and min/max envelope algorithms to adjust level-of-detail dynamically according to the viewport [22].

Visualization in industrial IoT and smart manufacturing. Industry reports and experience papers describe custom dashboards for production lines and instrumentation. Grafana use cases in manufacturing show how companies unify data from MQTT brokers, PLCs and databases into central dashboards [8]. Node-RED flows integrate sensors with plant control systems and can push alerts to HMIs. These examples highlight the value of dashboards in operational settings but also reveal reliance on infrastructure absent in the thesis scenario.

An alternative approach could be to decouple low-level device communication from visualization frameworks by introducing a message broker such as MQTT. In this model, the Rust backend would read the FIFO directly via IOCTL and publish the decoded data streams to an MQTT broker running locally or remotely. Visualization dashboards such as Grafana, Node-RED, or ThingsBoard could then subscribe to the MQTT topics, benefiting from a widely adopted, standards-based pub/sub mechanism.

While this approach would increase interoperability and align the system with common /glsIoT practices, it would also introduce additional overhead. Running a broker on the resource-constrained ARM-based SoC might consume memory and CPU otherwise available for real-time tasks. Moreover, the design goal of the thesis was to establish a lightweight, direct path from hardware registers and FIFO buffers to the web frontend without relying on external middleware. For scenarios where device data must be integrated into larger enterprise or cloud infrastructures, however, an MQTT bridge could represent a natural extension of the current system.

2.2 Gap analysis

The literature and platforms reviewed above demonstrate a rich ecosystem of IoT visualization solutions, from full-stack platforms with TSDBs to lightweight flows on single-board computers. However, none directly address the scenario where measurement data come from an embedded SoC with a pre-existing ring buffer and register interface, where there is no database, broker or external server, and where hardware-level control (e.g., starting/stopping a streaming mode) is necessary. General-purpose dashboards assume data ingestion into a TSDB and cannot access ring buffers via IOCTL; low-code tools are not optimized for high-rate binary streams; and even edge-focused research frameworks often rely on message brokers or embedded caches.

The thesis therefore fills a gap by implementing a custom backend and frontend that run directly on the SoC and interface with the HH-AMV measurement architecture. The backend communicates with FPGA kernel modules to read measurement data from a hardware FIFO (which functions internally as a ring buffer) via IOCTL-based access and provides a REST API through which acquisition can be controlled and data retrieved. The service operates entirely on the SoC without relying on external servers or databases, ensuring a lightweight and self-contained deployment. While the current implementation focuses on REST endpoints for bounded captures and device control, the architecture is designed to be extensible: WebSocket streaming for continuous monitoring and downsampling algorithms such as Largest-Triangle-Three-Buckets (LTTB) are not part of the present system but are identified as future enhancements that would improve usability at high sampling rates. By combining insights from surveyed literature—such as streaming transport patterns and lightweight frontend design—and tailoring them to the HH-AMV hardware, the thesis provides a concrete contribution to IoT visualization for embedded measurement devices.

CHAPTER 3

Thesis Contribution

The challenges identified in Section 1.1 revolve around the constraints of embedded systems and the absence of an off-the-shelf solution capable of visualising high-rate measurement data from an FPGA on a Linux-based SoC without a database or message broker. This chapter explains how those challenges motivated the design of a custom backend and frontend. The backend executes on the hard-processor system (HPS) of the HH-AMV platform and interacts with the FPGA through two character devices provided by the operating system. It exposes a small REST API that allows a browser-based frontend to configure and retrieve measurement streams. The result is a self-contained system that runs on the SoC itself and delivers real-time visualisation over a network.

3.1 System Architecture

The HH-AMV platform consists of two tightly coupled components: the *Simulationsteil* (ST) and the *Digitalteil* (DT) (see Figure 3.1). Together, these form a closed-loop measurement and control system where the ST provides analog input signals, and the DT processes, stores, and forwards data to external interfaces. The overall architecture is summarized here to provide context for the implementation in this thesis.

3.1.1 Simulationsteil (ST)

The ST generates analog signals that emulate sensor data. It contains twelve channels that can either be connected to external sources or internally simulated. In laboratory setups, the HH-AMV_Valid board fulfils this role: it acts as a stimulus generator that produces twelve analog channels and forwards them to the DT via the same ADC-SPI interface used by real sensors. This ensures that the DT logic, FIFO buffering and register access are exercised under realistic conditions.

Waveform generation. The HH-AMV_Valid supports arbitrary waveform generation. Predefined signals (sine, square, ramp) can be selected via a web GUI 3.2, while custom signals

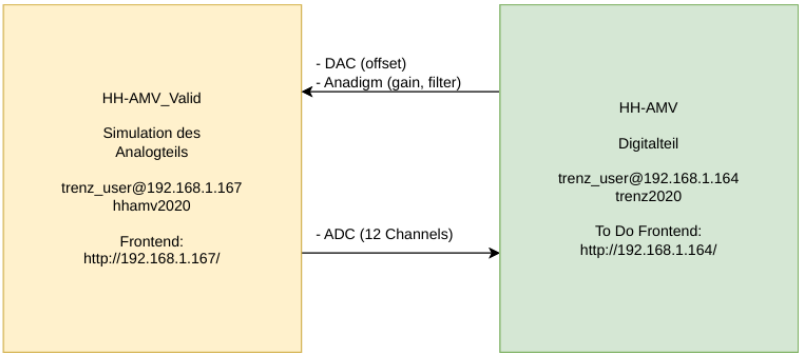


Figure 3.1: Interaction between the simulated analog part (HH-AMV_Valid) and the digital subsystem (HH-AMV). The analog unit provides digitised signals via 12 ADC channels, while the digital subsystem returns DAC offsets and Anadigm parameters (gain and filter). Both components are accessible over Ethernet, with separate frontends for configuration and visualization.

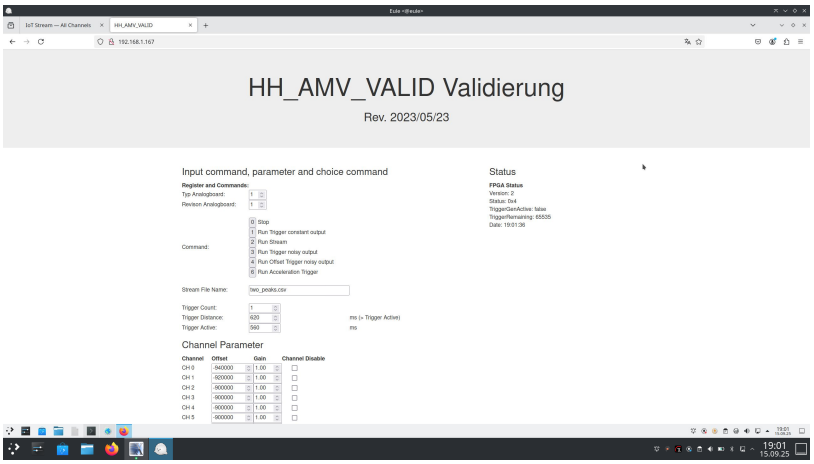


Figure 3.2: Web interface of the HH_AMV_VALID board. The GUI allows the user to select signal generation modes, upload waveform definitions, adjust trigger parameters, configure channel offsets and gains, and monitor FPGA status.

can be uploaded in CSV format. Each CSV file specifies the time series of the waveform, which the board replays as analog output. This enables replication of real measurement conditions and controlled stress-testing of the digital pipeline.

Closed-loop evaluation. Besides supplying ADC data streams, the ST also receives DAC outputs and re-injected signals after digital filtering and amplification by the DT. This feedback path allows closed-loop evaluation, where the effect of configurable gain, offset and filter settings can be observed on the generated waveforms.

From the perspective of the DT, the ST therefore acts as the environment delivering high-rate sensor data while also responding to digital control signals. This abstraction decouples the availability of real-world sensors from backend development and ensures reproducible test conditions.

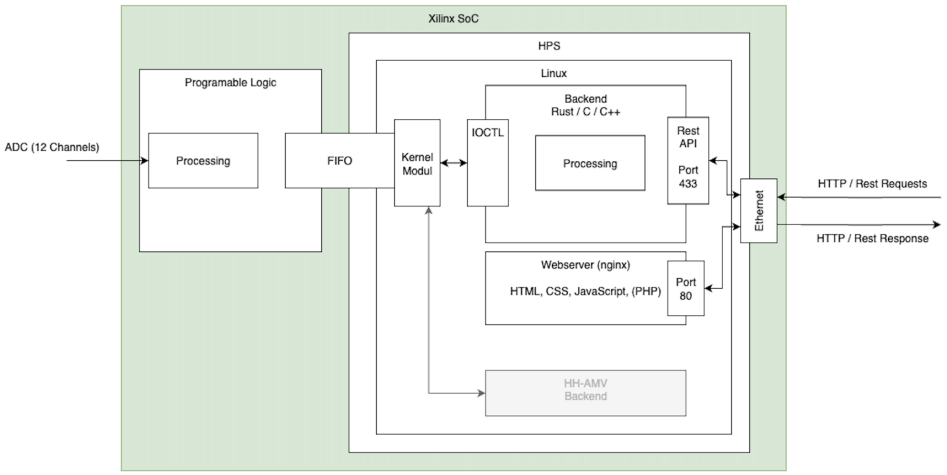


Figure 3.3: Data path from sensors to the web interface. The SoC collects digitised measurements from ADCs and writes them into a ring buffer. A kernel driver exposes this buffer as `/dev/hh_amv_psfifo`; a second driver provides register access via `/dev/hh_amv_psreg`. The backend service communicates with both devices using IOCTLs and exposes a REST API consumed by the browser frontend.

3.1.2 Digitalteil (DT)

The DT is based on a heterogeneous SoC that integrates an Programmable logic fabric with a Hard Processor System (HPS). The Programmable logic implements the following functional blocks:

- A ring buffer for streaming ADC samples.
- DAC and Anadigm interfaces for generating offset, gain, and filter responses.
- A set of memory-mapped registers controlling streaming modes, offsets, and error injection.

The HPS runs a Linux operating system and provides software access to these programmable logic blocks via character device drivers:

- `hh_amv_psreg` for register access,
- `hh_amv_psfifo` for FIFO data streaming,
- `hh_amv_psmem` for shared memory.

These drivers expose the FPGA resources as device files under `/dev/`, supporting access through `ioctl` calls and read/write operations.

3.1.3 Scope of This Thesis

The contribution of this thesis lies in building a complete data visualization stack on top of the existing HH-AMV platform. Specifically:

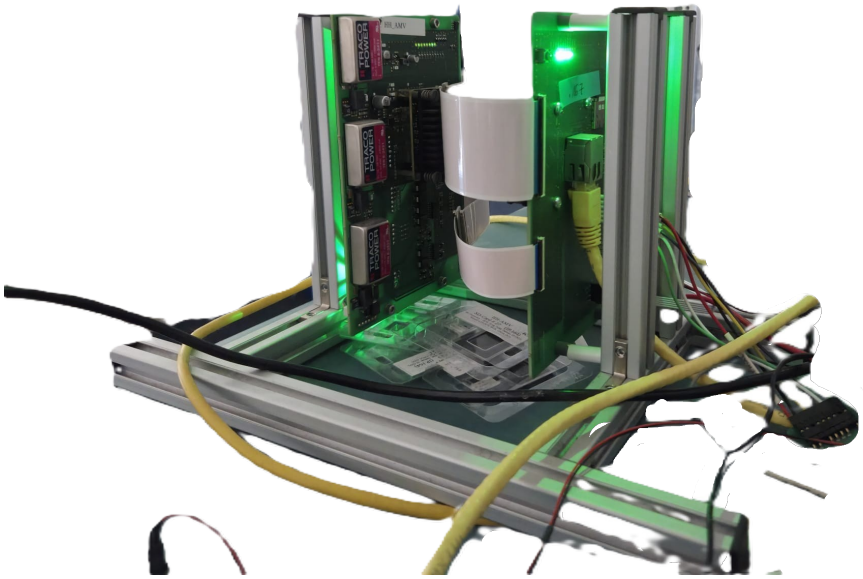


Figure 3.4: HH-AMV

- Development of a backend in Rust that interfaces with the kernel modules, using `ioctl` and read/write to access registers and the FIFO.
- Implementation of a REST-like API with Actix-web to expose measurement data and configuration endpoints.
- Design of a lightweight frontend in HTML, CSS, and JavaScript to visualize real-time data streams, using charting libraries for interactive plots.
- Evaluation of system performance in terms of latency, throughput, and resource usage.

Importantly, the work presented in this thesis focuses only on the HPS software layer; the FPGA logic, hardware registers, and kernel drivers were not modified but used as they were provided. The implemented middleware bridges the gap between the low-level SoC hardware and a human-readable, browser-based dashboard.

3.2 Implementation

The implementation consists of two cooperating components: a Rust backend service running on the SoC and a lightweight web frontend served to the client browser. The backend communicates with the Programmable logic via *PSREG* and *PSFIFO* kernel modules and exposes the data via HTTP. The frontend uses HTML, CSS and JavaScript (Chart.js) to poll the backend and render live plots. This section describes both parts in detail, beginning

with an architectural overview and then drilling down into the backend and frontend internals. An extensibility subsection summarises how WebSocket streaming and downsampling could be added in future without altering the core design.

3.2.1 Overview

At a high level, the contribution of this thesis is the creation of a *platform-independent visualisation pipeline* for the HH-AMV measurement platform. The pipeline starts with analogue signals acquired by the SoC, passes through a ring buffer, is exposed to user space via two character devices, and ends with interactive plots in the browser. The backend is implemented in Rust using the Actix-web framework and the `nix` crate for low-level system calls. It initialises the hardware, controls streaming modes via the command register, decodes fixed-size data packets from the FIFO, and serves a RESTful API. The frontend is a single-page web application using Chart.js, styled with CSS. It periodically polls the backend, updates line charts for all channels, and allows the user to configure the sampling mode, number of packets per poll and axis scaling.

The design satisfies the constraints identified in Section 1.1 because it (i) requires no external database or broker, (ii) executes on the SoC itself, and (iii) exposes a standards-compliant web interface for platform-independent access. The following sections describe how these goals are realised.

3.2.2 System Architecture

The HH-AMV platform integrates a Xilinx SoC in which programmable logic (PL) and a hard-processor system (HPS) operate together. On the PL side, measurement data are generated by the analogue front-end and digitised by twelve analogue-to-digital converters (ADCs). These ADC channels are first pre-processed in programmable logic before being streamed into a soft-logic FIFO buffer. This FIFO provides continuous high-rate data transfer to the processor domain.

On the HPS side, two kernel modules provide the interface to user space: one for register access and one for FIFO streaming. These modules expose character devices under `/dev`, and support IOCTL commands for configuration and data access. The IOCTL definitions were provided as part of the reference implementation and ensure low-latency interaction between user processes and the FPGA fabric.

The backend software stack, implemented in Rust with auxiliary C/C++ code, forms the central processing unit of the system. It opens the character devices, applies IOCTL commands to configure registers, and streams data from the FIFO. The backend exposes a REST API (port 433) that delivers measurement data and control functions to external clients. On top of this, an `nginx` web server (port 80) hosts the browser-based frontend, implemented in HTML, CSS and JavaScript. This frontend provides visualisation and control, consuming the REST API published by the backend.

From a development perspective, the backend and frontend modules were designed and implemented as part of this thesis. The kernel drivers, IOCTL interface and low-level FPGA integration were already provided by the supervisor and served as a stable foundation.

Figure 3.3 illustrates the overall architecture, showing the data path from ADC sampling through programmable logic, kernel drivers, backend processing, REST API, and finally to the web-based frontend.

The major components of the implementation and their responsibilities are summarised in Table 3.1. Each component is implemented in a specific language and interacts with a defined set of interfaces. For brevity, autogenerated modules such as `reg_bank.rs` and `reg_fifo.rs` contain thousands of IOCTL wrapper functions and are summarised rather than listed exhaustively.

3.2.3 Backend: SoC-resident service

The backend is implemented in Rust (edition 2021) and uses the Actix-web framework for asynchronous HTTP handling. The `Cargo.toml` lists dependencies such as `serde` for JSON serialization, `nix` for POSIX system calls, `env_logger` for logging, `chrono` for timestamps and `tokio` for the async runtime. The build system is Cargo; the release binary can be compiled with:

```
1  cargo build --release
```

Listing 3.1: Building the backend

The resulting executable can be deployed to the SoC by copying the `target/release/iot-backend` binary and installing a systemd unit that runs it on boot (assumed but not provided). The server binds to `192.168.1.164:433` and serves both the REST API and static frontend assets. When the program starts, it calls `hardware_init()` in `main.rs` to open the register and FIFO devices, reset the FIFO pointers and enable oscillators. Only after the hardware is in a known state does the Actix server begin accepting connections.

File structure

The `backend/` directory contains all source files. The key modules are summarised below; every file under `src/` is documented in Table 3.1.

`main.rs` Entry point that sets up logging, performs hardware initialisation via `hardware_init()`, and starts the Actix web server. It mounts the REST handlers configured in `handler.rs` and serves static files from `/var/www/html`.

`handler.rs` Contains the HTTP handlers for `/api/health`, `/api/stream/start`, `/api/stream/stop` and `/api/stream/once`. Each handler opens the device descriptors, calls appropriate functions in `fifo_stream.rs`, and returns JSON responses via Actix. For example, the `start_stream` handler maps query parameters to internal stream modes and writes the corresponding command bits before returning a JSON acknowledgement.

`fifo_stream.rs` Implements all low-level streaming operations. It provides functions to check whether the FIFO driver is present, reset and drain the FIFO, enable or disable

streaming and ramp generation, and decode incoming data packets. The key function `decode_datablock` reads one or more packets from the FIFO, aligns to packet headers, extracts 16-bit samples and demultiplexes them into per-channel vectors; a portion of this function is shown in Listing A.1 with line numbers taken from `src/fifo_stream.rs` lines 246–284. The decoding logic increments the channel index after each 16-bit sample and wraps around when reaching the configured number of stream channels.

`reg_bank.rs` and `reg_fifo.rs` Autogenerated modules created by the abaxor tool chain. They contain hundreds of IOCTL wrapper functions that perform 32-bit reads and writes on registers exposed by the kernel drivers. For example, `reg_bank::set_command()` writes the 16-bit command register and `reg_bank::get_command()` reads it. Similarly, `reg_fifo::get_axi4_str_rdfd()` reads a 32-bit word from the FIFO stream. These modules define no business logic; they simply marshal data into and out of the device nodes.

`parser.rs` Provides a convenience function to convert a slice of raw 32-bit FIFO words into a `Measurement` struct containing a timestamp and 12 channel values. Each 32-bit word is sign-extended and scaled to a voltage; this function is reserved for future use and is not currently exposed via the API.

`models.rs` Defines serialisable structures used in API responses, such as `ChannelData` (channel index and value) and `Measurement`. These are leveraged by Actix when returning JSON.

`settings.rs` Contains compile-time constants such as `MAXIMALE_ANZAHL_AN_MESSSTELLEN` (the number of measurement channels, set to 12), `FIFO_STREAM_DATA_PACKET_SIZE` (1024 bytes), `FIFO_STREAM_DATA_HEADER_SIZE` (8 bytes) and `FIFO_NUM_STREAM_CH` (number of stream channels including two auxiliary channels). These constants drive buffer sizing and loop bounds throughout the code.

`flags.rs` Declares bit masks for the command register (e.g., `OSCILLATOR_EN`, `STREAM_TRANSFER_EN`, `STREAM_RAMP_EN`, `STREAM_SELECT`) and status register (e.g., `NEW_MEAS_RESULTS`). These constants allow the code to atomically set or clear bits when controlling the FPGA.

`sys_log.rs` Initialises logging to the system logger using the `syslog` crate. It configures messages at debug and error levels and is called early in `hardware_init()`.

Kernel interface (IOCTL)

The backend interacts with the FPGA exclusively through two character devices: `/dev/hh_amv_psreg_dev` (register bank) and `/dev/hh_amv_psfifo_dev` (stream FIFO). Both devices are opened in non-blocking read/write mode using the POSIX `open` system call. Because the drivers expose a large number of registers, the abaxor tool chain generates wrapper functions via `nix` macros. Selected aspects of the interface are summarised here.

Command register. The command register is 16 bits wide. The backend reads it via `reg_bank::get_command()` and writes it via `reg_bank::set_command()`. Bit masks de-

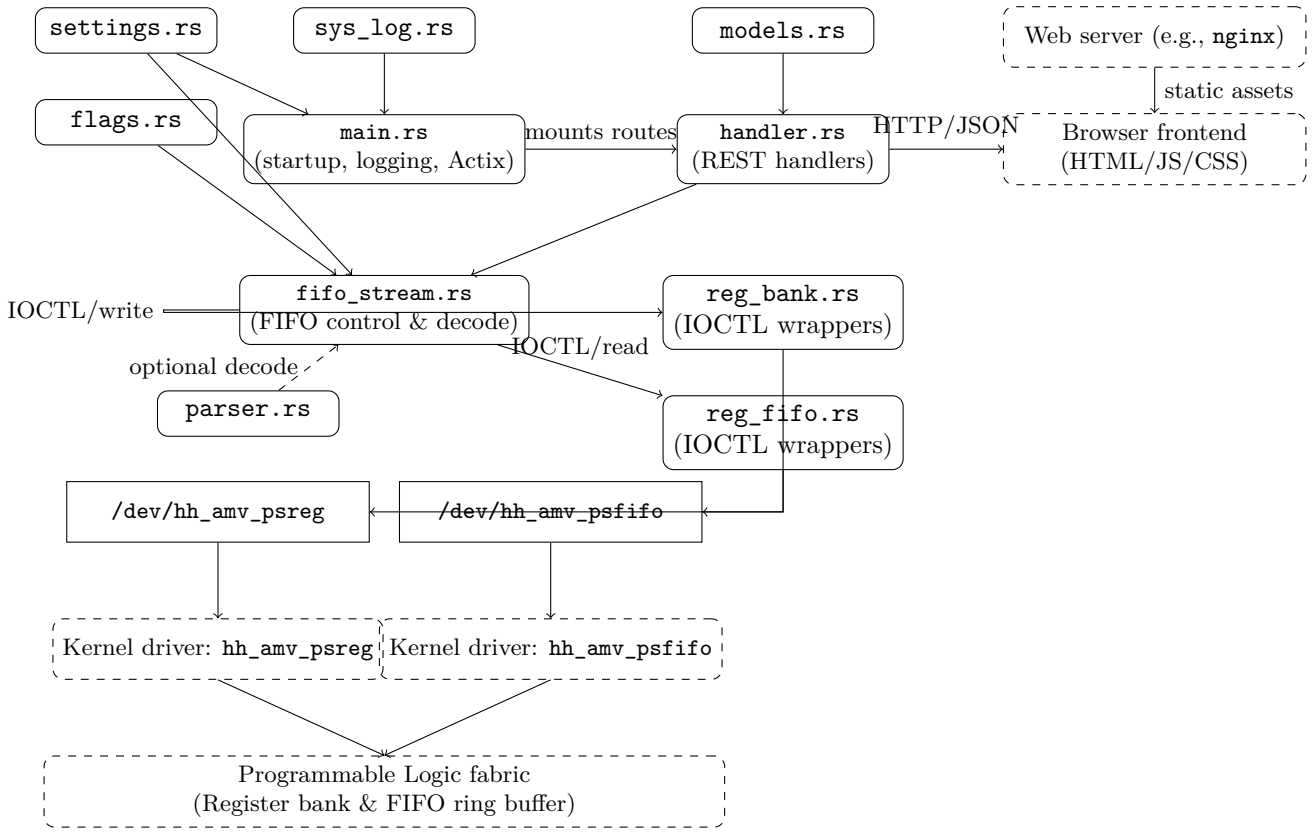


Figure 3.5: Module interaction in the backend. The entrypoint `main.rs` initialises logging and starts Actix; `handler.rs` implements the REST API consumed by the browser. Low-level streaming and control reside in `fifo_stream.rs`, which uses autogenerated IOCTL wrappers (`reg_bank.rs`, `reg_fifo.rs`) to access the device nodes `/dev/hh_amv_psreg` and `/dev/hh_amv_psfifo`. The kernel drivers bridge to the FPGA register bank and FIFO ring buffer. Configuration (`settings.rs`) and bit masks (`flags.rs`) parameterise behaviour; `sys_log.rs` provides system logging; `models.rs` defines JSON response types; `parser.rs` is an optional decoder.

fined in `flags.rs` denote specific functions: `OSCILLATOR_EN` enables the oscillator, `STREAM_TRANSFER_EN` starts streaming, `STREAM_RAMP_EN` toggles ramp generation, and `STREAM_SELECT` selects the raw stream without offset. Setting multiple bits results in combined behaviour (e.g., ramp streaming sets both `STREAM_TRANSFER_EN` and `STREAM_RAMP_EN`). The backend clears bits using bitwise AND with the negation of the mask before writing.

FIFO control. The FIFO module exposes registers such as `RDFO` (occupancy), `RDFD` (read data), `RDFR` (reset FIFO) and others. In practice, the backend uses `reg_fifo::get_axi_str_rdfo()` to obtain the number of words available in the FIFO and `reg_fifo::get_axi4_str_rdfd()` to retrieve a 32-bit word. Resetting the FIFO pointers is accomplished by writing `flags::RESET_KEY` to the `RDFR` register via `reg_fifo::set_axi_str_rdfr()`. This operation is used both during initialisation and whenever streaming is stopped.

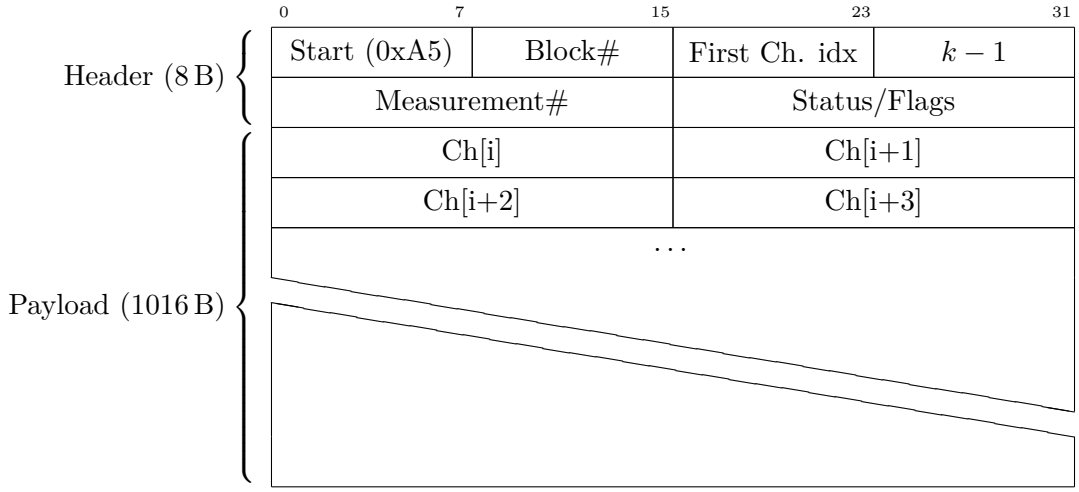


Figure 3.6: FIFO block structure. Each block contains 1024 B: an 8 B header and 1016 B payload. The header encodes the start marker (0xA5), block number (modulo 256), index of the first channel, channel count $k-1$, status/flags, and measurement number. The payload holds interleaved 16-bit signed samples, packed two per 32-bit word, wrapping across channel indices modulo `settings::FIFO_NUM_STREAM_CH`.

Ring buffer semantics. Each FIFO block contains 1024 bytes (`settings::FIFO_STREAM_DATA_PACKET_SIZE`) 3.6 . The *first words within the block* encode the header fields—fixed start marker (`0xA5`), block number (incremented modulo 256), index of the first channel present in this block, and the number of sensor channels k—1 as defined by the streaming format; the remaining bytes carry interleaved samples that may wrap across block boundaries. The implementation reads 32-bit words and reconstructs samples from the documented byte layout (for payload words this typically means unpacking two signed 16-bit values). After emitting each sample, the code advances a local `current_channel` and wraps modulo (k+2) (in code: `settings::FIFO_NUM_STREAM_CH`) to maintain alignment even if a block starts mid-channel. The *measurement number* is parsed from the header/status words and used to terminate bounded captures once the required count is reached; it is not treated as an extra channel. Overrun and underrun are mitigated via bounded waits and draining loops: if no data arrives within the configured deadline, the reader abandons the current block to avoid blocking. When visiting auxiliary channel k (vibration/temperature), the lower two selector bits are masked before interpreting the value.

3.2.4 FIFO Block Layout

REST API

The backend exposes a minimal REST interface over HTTP. Endpoints and their semantics are summarised in Table 3.2. All responses are JSON and use standard HTTP status codes. The API is stateless: requests include all necessary parameters and the backend does not maintain session data. No authentication is performed; the service is intended to run on a

private network.

For example, issuing

```
curl -X POST "http://<ip>:433/api/stream/start?mode=ramp"
```

Listing 3.2: Example API call

returns a JSON response `{ "status": "started", "mode": "ramp" }`. Subsequent `GET /api/stream/once?packets=2` reads two packets, decodes them and returns the per-channel samples. After streaming, the client should issue `POST /api/stream/stop` to release hardware resources.

Streaming and acquisition control

The acquisition workflow follows a simple state machine: (1) prepare the hardware, (2) start streaming, (3) acquire and decode packets, and (4) stop streaming. Preparation occurs in `init_fifo()` where the backend drains any leftover data from the FIFO, resets the pointers and ensures that the oscillator is enabled. Starting streaming is achieved by setting bits in the command register using either `start_streaming()` (offset mode), `start_streaming_without_offset()` or `start_ramp_gen()`. These functions simply OR the current command with the appropriate masks and write it back. Acquisition is performed by `decode_datablock()` which reads a specified number of packets and returns when either the requested number of packets have been decoded or, in *without_offset* mode, when a certain number of measurements have been collected.

Listing A.1 shows the inner loop of `decode_datablock` that reads 32-bit words, splits them into 16-bit samples and assigns them to channels. The code demonstrates how the measurement number is stored separately, how channels wrap around `settings::FIFO_NUM_STREAM_CH` and how glitch detection is incremented only when the `STREAM_GLITCHES_TEST` option is selected.

Stopping streaming is accomplished by clearing the transfer and select bits using `stop_streaming()` and `stop_ramp_gen()`, then resetting the FIFO again via `reg_fifo::set_axi_str_rdfr()`. The backend also drains residual data to avoid stale samples when the next stream starts.

Error handling and resilience

The implementation anticipates several error cases and handles them gracefully:

- *Device missing.* The health endpoint tests whether the FIFO device node exists using `std::fs::metadata`. If not, `/api/health` returns `fifo_device_present=false` and status `"fifo_device_missing"`. Other handlers return HTTP 500 if device opening fails.
- *Timeouts.* All reading loops include deadlines; if the occupancy does not exceed a threshold within 20–100 ms the loop breaks and the function returns partial data. This prevents blocking when there is no stream or when the FIFO pointer is misaligned.

- *Alignment loss.* During header search the code checks whether the header marker is 0xA5 and whether the first channel index is within range. If not, it resets the alignment search and drains until a valid header is found. If no header appears before a deadline, decoding terminates with whatever has been collected.
- *Glitch detection.* In ramp mode, the code compares successive samples modulo 2^{16} and counts jumps that deviate from the expected +1 increment. This allows the frontend to display the number of anomalies in a ramp test.
- *JSON errors.* All API handlers serialise responses via `serde_json`; any serialisation error causes an internal server error which Actix reports automatically. Inputs are validated by parsing query parameters via `serde::Deserialize`.

Performance considerations

Although no explicit benchmarking is performed in this chapter, the implementation makes several choices to minimise overhead. The backend reads data in 32-bit words and demultiplexes two samples at a time, thus halving the number of IOCTL calls compared with 16-bit reads. It allocates channel vectors once per `decode_datablock` call and reuses them across packets. Non-blocking I/O and bounded waits avoid kernel blocking and keep the Actix event loop responsive. Logging is optional and can be disabled at compile time via the `verbose` feature. The use of Actix-web ensures that HTTP requests are processed asynchronously, preventing the web server from blocking on device reads.

Security considerations

The current backend does not implement authentication or transport encryption. It assumes operation on a private network behind a firewall. Cross-Origin Resource Sharing (CORS) is not enabled; therefore, only clients served from the same domain (the SoC's IP) can access the API. Input parameters are parsed via `serde` and numeric bounds are checked to prevent uncontrolled memory allocations. Nevertheless, if the backend were exposed to untrusted clients, additional mechanisms such as token-based authentication, rate limiting and HTTPS termination would be required.

Build and deployment

To build the backend for the target architecture, run the command shown in Listing 3.1. When deploying to the HH-AMV SoC running Linux, the binary can be placed in `/usr /local/bin` and a systemd service file can ensure it starts after the device drivers are loaded. The systemd unit would typically set the working directory to `/var/www/html` for static file serving and redirect logs to journald.

3.2.5 Frontend: browser-based UI

The frontend is a single-page application served by the backend. It comprises an HTML document, a CSS stylesheet and a JavaScript module. The application uses Chart.js for

plotting and the Fetch API for communication with the backend. Static assets are served via the Actix `Files` service or through an external HTTP server such as nginx.

File structure and stack

The `frontend/` directory contains three files:

- `index.html` defines the page structure. It loads the Chart.js library from a CDN, links `style.css` for styling, includes form controls for mode selection, packet count and polling interval, and defines a `canvas` for the chart and a `pre` element for status logs.
- `style.css` provides responsive styling via CSS grid layouts, custom properties for colours, and a light/dark mode. It arranges the controls, chart and scaling card and ensures the chart occupies an appropriate portion of the viewport.
- `script.js` implements the application logic. It defines helper functions to read form inputs, build a Chart.js line chart, apply axis scaling, fetch JSON from the API and update datasets. The script uses `setInterval` to poll `/api/stream/once` at a user-selected interval and logs messages in the status console.

The frontend does not use a build tool or bundler; instead, the files are served as-is. Chart.js is loaded from a CDN to minimise the footprint on the SoC. No additional JavaScript frameworks are used.

UI architecture and components

Upon loading, the script constructs a Chart.js line chart with a data set for each measurement channel. Colours are assigned by computing evenly-spaced hues on the HSL circle. The user can select one of three streaming modes (with offset, without offset or ramp), specify how many 1024-byte packets to request per poll and choose the polling interval in milliseconds. Two buttons control streaming: "Start Stream" posts to `/api/stream/start` and begins periodic polling, while "Stop Stream" posts to `/api/stream/stop` and cancels the polling timer. A separate card allows manual adjustment of the x- and y-axis limits.

The script parses numeric inputs, applies basic validation (e.g., minimum poll interval of 50 ms), and logs changes in the status console. When new data arrive, `updateAllChannels()` ensures that the chart has the correct number of datasets and samples, updates each dataset, applies axis scaling and triggers a chart redraw.

API integration

All network access is encapsulated in `fetchJson()`, which wraps the Fetch API and checks for HTTP errors and the `application/json` content type. Polling is implemented via a timer that calls `fetchOnceAndPlot()` at the user-selected interval. Each call builds a query string containing the number of packets and the mode, sends a GET request to

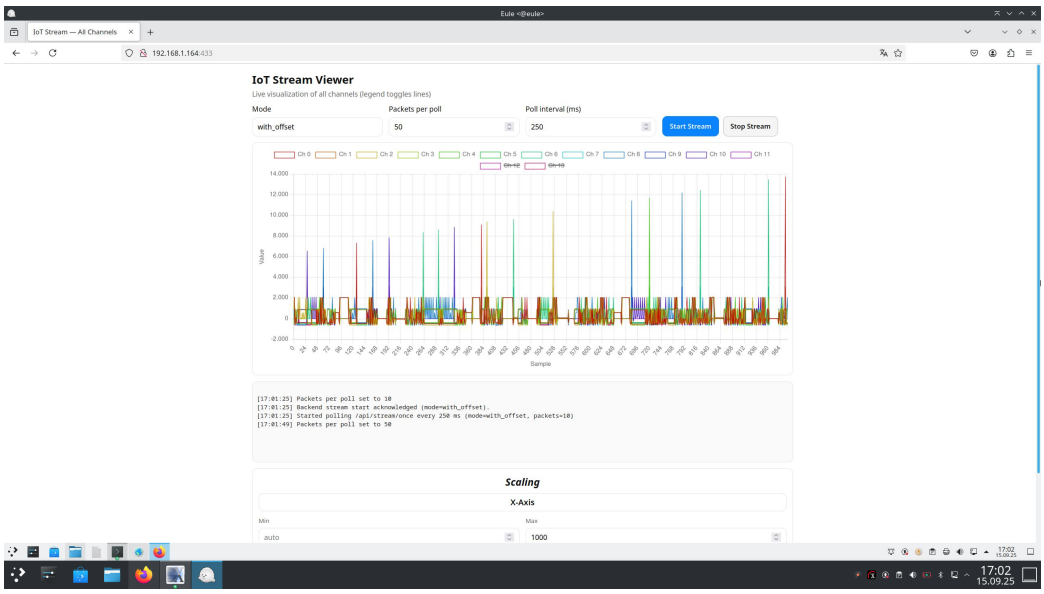


Figure 3.7: Frontend UI

`/api/stream/once`, decodes the JSON response and hands it to `updateAllChannels()`. Error messages are appended to the status console and printed to the JavaScript console.

Data visualisation and performance

Chart.js renders a separate line series for each channel and supports interactive tooltips (see Figures 3.10 - 3.14). Animation is disabled and the point radius is zero to maximise rendering performance. When new data arrive, the script computes the maximum channel length and pads shorter series with `null` values so that all datasets have equal length. Scaling inputs call `applyAxisScaling()`, which updates the axis limits immediately. For large data sets, Chart.js may suffer performance degradation; the planned downsampling extension (Section 3.2.9) would mitigate this by pre-aggregating samples before plotting.

Responsiveness and deployability

The CSS uses grid layouts and viewport units to ensure that the UI adapts to different screen sizes (see Figure 3.15 - 3.18). The colour scheme also adapts to the user's preferred colour scheme via a media query. Because the frontend consists only of static files, it can be served by any HTTP server. During development, the files are served by the Actix backend itself via `Files::new("/", "/var/www/html")`; for production, they may be placed in an nginx document root. The absence of a build step means that updating the UI only requires

Figure 3.8: Scaling X and Y axis

copying the files to the SoC.

3.2.6 Configuration and parameters

Most parameters controlling the system are defined as compile-time constants in `settings.rs`. These include the number of measurement channels (12), packet size (1024 bytes), header size (8 bytes), the number of additional FIFO channels for vibrations and trigger data (`ANZAHL_AN_ZUSATZ_KANAELE_FIFO_STREAM`=2), and default oscillator settings. Changing these values requires recompilation. At runtime, the user can adjust only the REST parameters (mode, number of packets, polling interval) and the axis scaling on the frontend. The command register bits provide further control (e.g., high-resolution mode), but the current API does not expose them.

3.2.7 Logging, diagnostics and observability

The backend initialises logging via `env_logger` and `syslog`. When run with default settings, it prints coloured messages to standard output indicating hardware initialisation steps and server startup. If compiled with the `verbose` feature, the backend writes debug information to syslog (facility `LOG_USER`). The `/api/health` endpoint allows clients to verify that the FIFO device is present and that the backend is running. In case of failures, error messages such as “Failed to init oscillator” are emitted via `stderr` and logged.

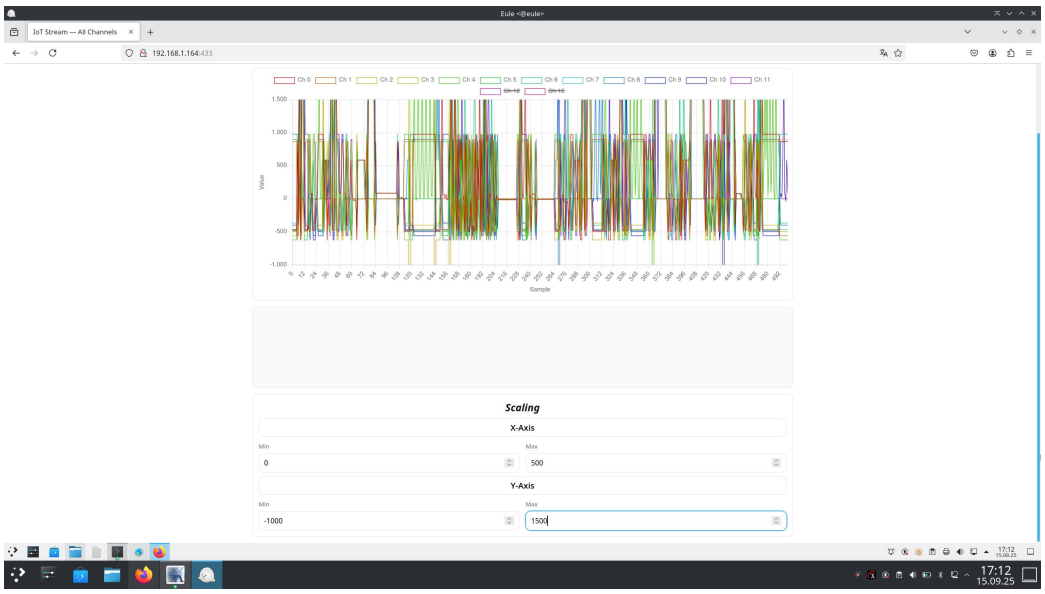


Figure 3.9: Scaling X and Y axis

3.2.8 Validation hooks

Several sanity checks are built into the code. The FIFO initialisation function drains residual data and ensures that the occupancy drops to zero before streaming begins. The header search verifies the start marker and channel count; failure to find a header within 250 ms causes the function to return without delivering data. In ramp mode, the `glitch_if_out_of_sequence()` function checks that successive samples increase by exactly one modulo 2^{16} and increments a counter otherwise. These checks help detect misconfiguration or signal integrity issues early.

3.2.9 Extensibility

Although this chapter does not evaluate future enhancements, two extensions are planned. First, the backend could expose a WebSocket endpoint to stream data to the browser continuously rather than relying on HTTP polling. Actix-web supports WebSockets; implementing this would involve moving the packet decoding loop into a streaming task and pushing JSON frames over the socket. On the frontend, the polling timer would be replaced with a WebSocket listener that updates the chart as messages arrive. Second, downsampling algorithms such as Largest-Triangle-Three-Buckets (LTTB) could be inserted into `fifo_stream.rs` or `script.js` to reduce the number of points plotted for long time windows. LTTB selects representative points that preserve the global shape of the waveform. Integrating LTTB would require grouping raw samples into buckets, computing area metrics and emitting only three points per bucket. Both extensions are straightforward additions to the existing architecture and would enhance scalability without altering the system's core behaviour.

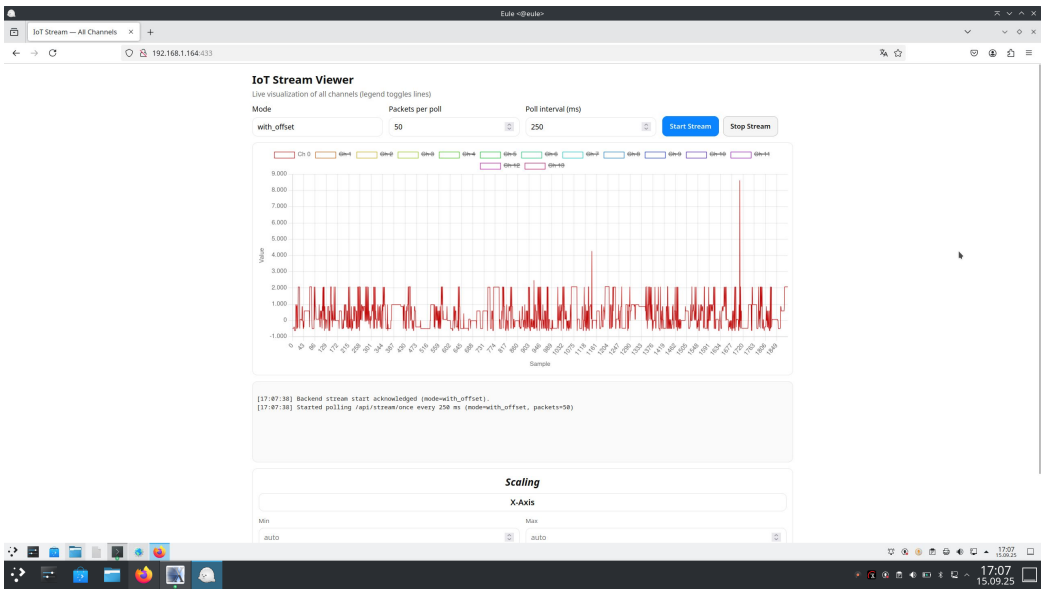


Figure 3.10: Seprate line series for Channel 1 in Red color

3.3 Experiments

In order to evaluate the correctness and performance of the implemented system, we designed a series of experiments that exercise all major features of the backend and frontend described in Section 3.2. The experiments are designed to be reproducible and to allow quantitative comparisons between different streaming modes, API parameters and polling strategies. All results are deferred to Chapter 4; this section merely details the setups, tools and data collection procedures.

3.3.1 Experimental setup and tools

All experiments are executed on the HH AMV platform with the backend compiled in release mode and running on the SoC's Linux environment. The SoC exposes two character devices, `/dev/hh_amv_psreg_dev` and `/dev/hh_amv_psfifo_dev`, which are opened by the backend at startup. The frontend is served from the backend and accessed via a modern browser on a laptop connected to the same Ethernet segment as the SoC. Because the hardware has no permanent storage, all telemetry is streamed from the FPGA ring buffer; each packet consists of 1024 bytes with an 8 byte header and 256 32 bit words (512 samples)[1]. The oscillator period and ADC configuration remain at their default values throughout.

Metrics are collected using standard tools: the wall clock request time is measured with the Unix `time` utility when invoking the REST API via `curl`; CPU and memory usage on the SoC are monitored with `top` and sampled every 0.5 s; network throughput is recorded via `ifstat`. Glitch counts are returned directly in the JSON response from `GET`

[1] These values correspond to the constants `FIFO_STREAM_DATA_PACKET_SIZE`, `FIFO_STREAM_DATA_HEADER_SIZE` and `FIFO_STREAM_DATA_WORDS_PER_PACKET` defined in `backend/src/settings.rs`, lines 67–74.

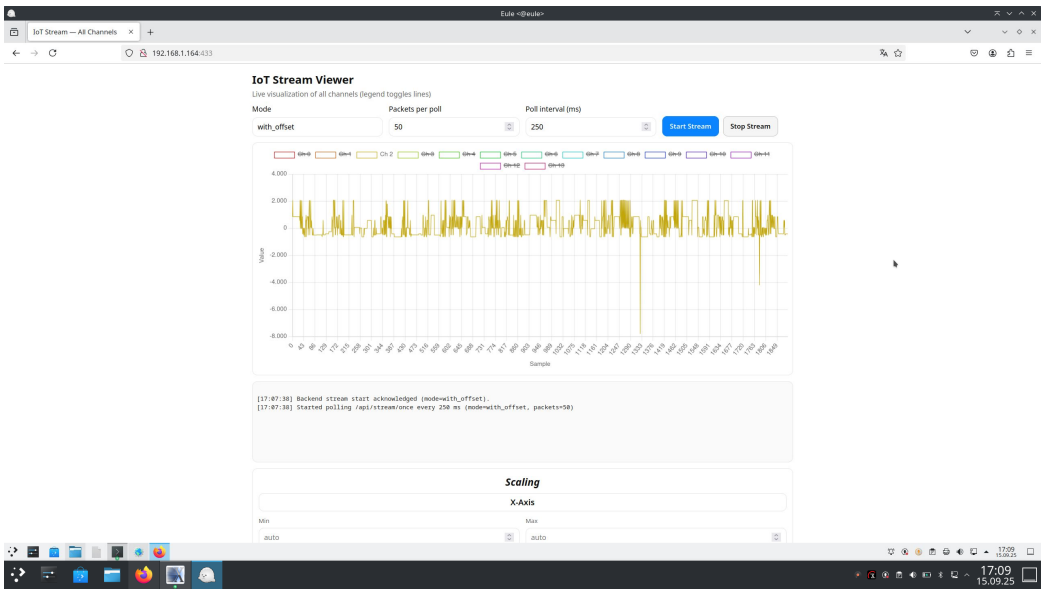


Figure 3.11: Seprate line series for Channel 2 in Yellow color

`/api/stream/once`.

3.3.2 Throughput versus number of packets

The first experiment studies how the throughput and latency of the bounded capture endpoint `/api/stream/once` depend on the number of packets requested. The independent variable is the query parameter `packets`$\in\{1,2,5,10,20\}$; all other parameters remain fixed (streaming mode `with_offset`, no concurrent clients, poll interval ignored because this endpoint returns immediately). For each value, `curl` invokes the endpoint, and the time between sending the HTTP request and receiving the complete JSON response is recorded. The script logs the length of the `channel_data` vectors and the size of the `raw_words` array to confirm that each packet yields 512 samples. CPU and memory usage are measured on the SoC during each request. It is expected that the total capture time increases roughly linearly with the number of requested packets, whereas throughput (samples per second) increases sub linearly and plateaus when the FIFO or network becomes the bottleneck.

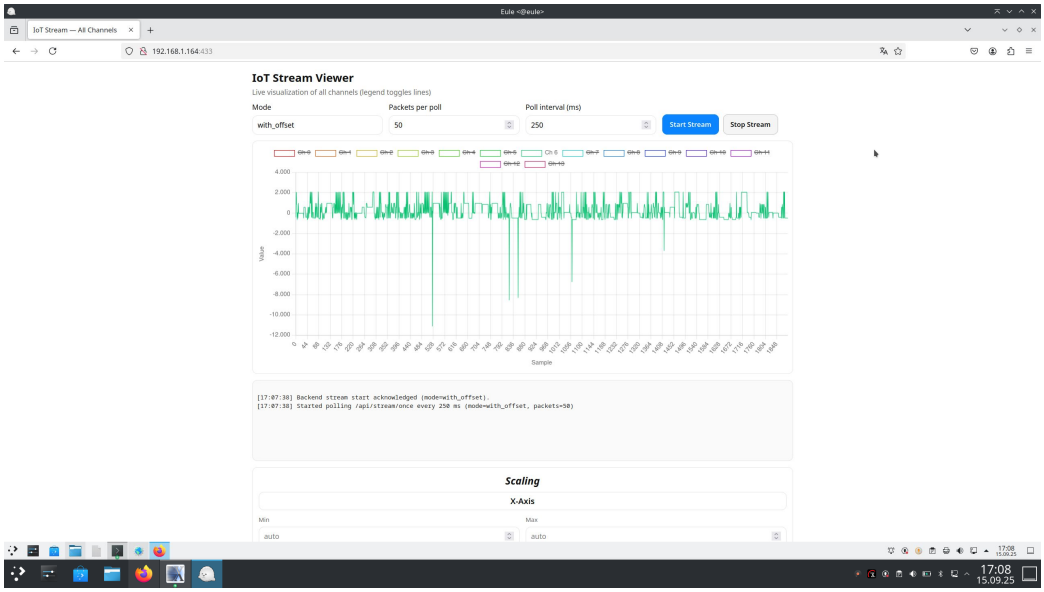


Figure 3.12: Seprate line series for Channel 6 in Green color

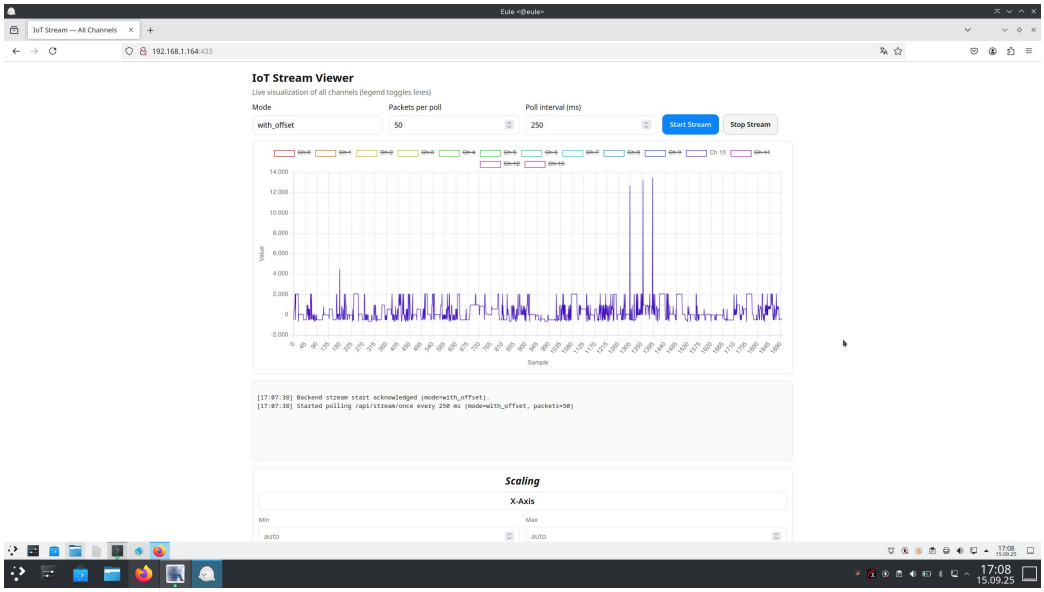


Figure 3.13: Seprate line series for Channel 10 in Purple color

Module/Component	Language	Responsibility	Interfaces
`main.rs`	Rust	Entry point; initialises hardware and launches Actix server	Opens `/dev/hh_amv_psreg_dev`, `/dev/hh_amv_psfifo_dev`; HTTP on port 433
`reg_bank.rs`	Rust	Autogenerated IOCTL wrappers for the register bank	`/dev/hh_amv_psreg_dev`
`reg_fifo.rs`	Rust	Autogenerated IOCTL wrappers for the stream FIFO	`/dev/hh_amv_psfifo_dev`
`fifo_stream.rs`	Rust	Controls streaming, initialises the FIFO, starts/stops transfer and ramp generator, decodes data packets	Uses IOCTLs via `reg_bank.rs` and `reg_fifo.rs`
`handler.rs`	Rust	Defines REST endpoints and binds them into Actix	JSON over HTTP
`parser.rs`	Rust	Converts raw words to `Measurement` structs (used for future API)	Internal only
`models.rs`	Rust	Structures for API responses	Internal only
`settings.rs`	Rust	Compile-time constants (channel count, packet size, etc.)	Internal only
`flags.rs`	Rust	Bit masks for command/status flags	Internal only
`sys_log.rs`	Rust	Configures syslog logging	Syslog facility `LOG_USER`
Frontend (HTML)	HTML	Entry page served to the browser	Loads Chart.js, links script and stylesheet
Frontend (JS)	JavaScript	Polls the REST API, updates charts and UI	Fetches `/api/` endpoints
Frontend (CSS)	CSS	Styles the UI	None

Table 3.1: Overview of implementation components. Paths are relative to the repository roots `backend/` and `frontend/`.

Table 3.2: REST endpoints exposed by the backend. Query parameters are optional unless noted otherwise.

Method / Path	Request parameters	Description / Response schema
`GET /api/health`	none	Returns {`fifo_device_present`: *bool*, `status`: *string*} to indicate whether the FIFO device node exists. Useful for diagnostics.
`POST /api/stream/start`	`mode`∈{`with_offset`, `without_offset`, `ramp`}	Resets the FIFO, configures the command register and starts streaming or ramp generation accordingly. Returns {`status`: "started", `mode`: *string*}.
`POST /api/stream/stop`	none	Stops streaming and ramp generation, resets the FIFO and returns {`status`: "stopped"}. Idempotent.
`GET /api/stream/once`	`packets` (default 1), `mode` as above	Performs a bounded capture of one or more packets without leaving the board in streaming mode. Returns {`glitches`: *number*, `channel_data`: array of channel arrays of $i16$, `raw_words_len`: *number*}. The last channel array contains the measurement number stream for alignment.

Table 3.3: Summary of throughput experiment parameters.

Parameter	Values	Fixed settings	Measured metrics	Repetitions
`packets`	1, 2, 5, 10, 20	mode: `with_offset`	Response time, CPU%, memory usage, samples returned	30 each

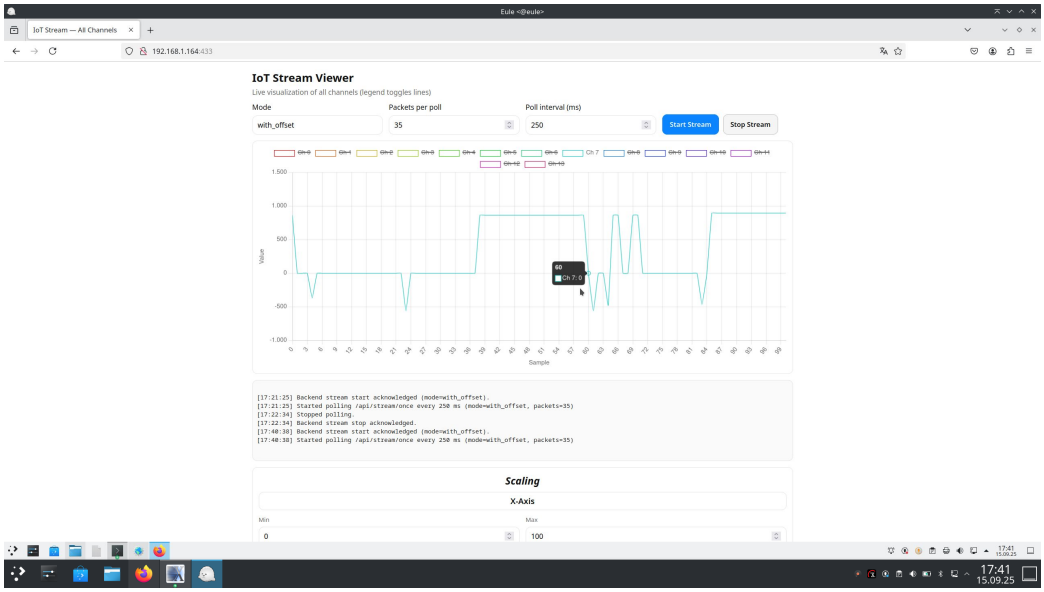


Figure 3.14: Interactive tooltip

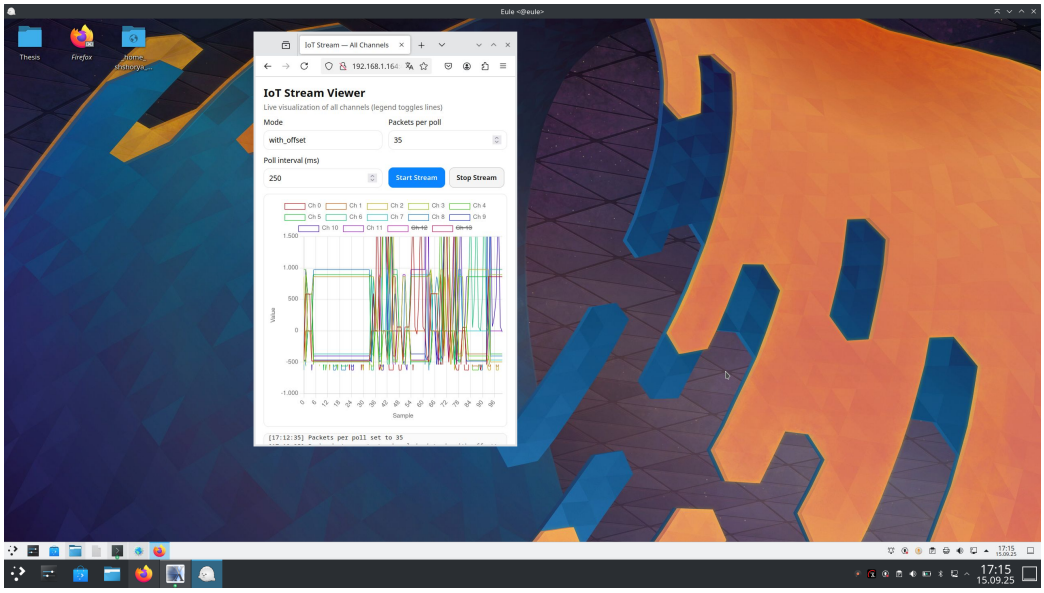


Figure 3.15: Frontend UI Responsivness to screen size

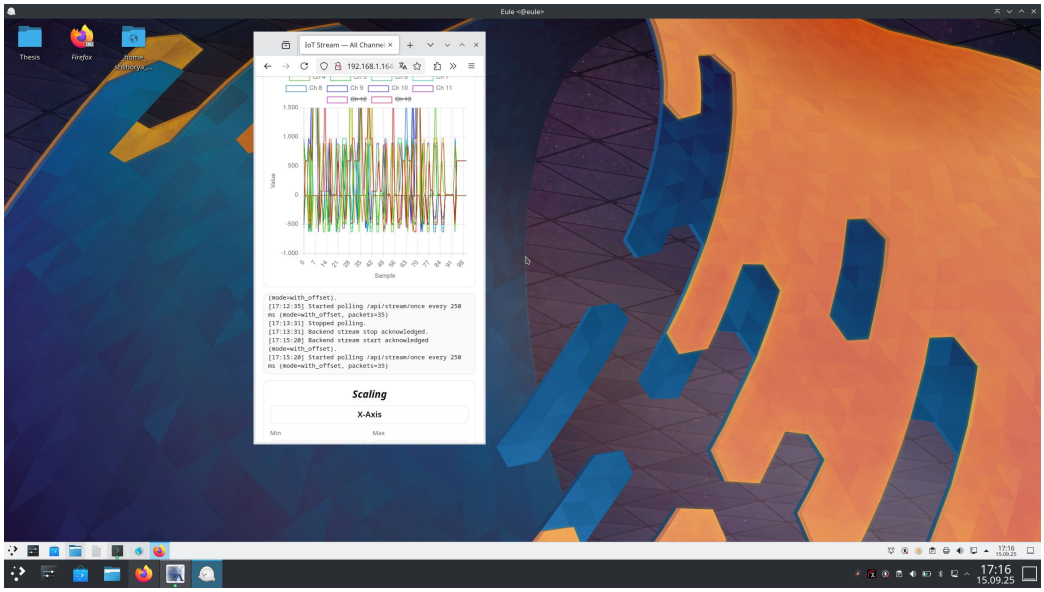


Figure 3.16: Frontend UI Responsivness to screen size

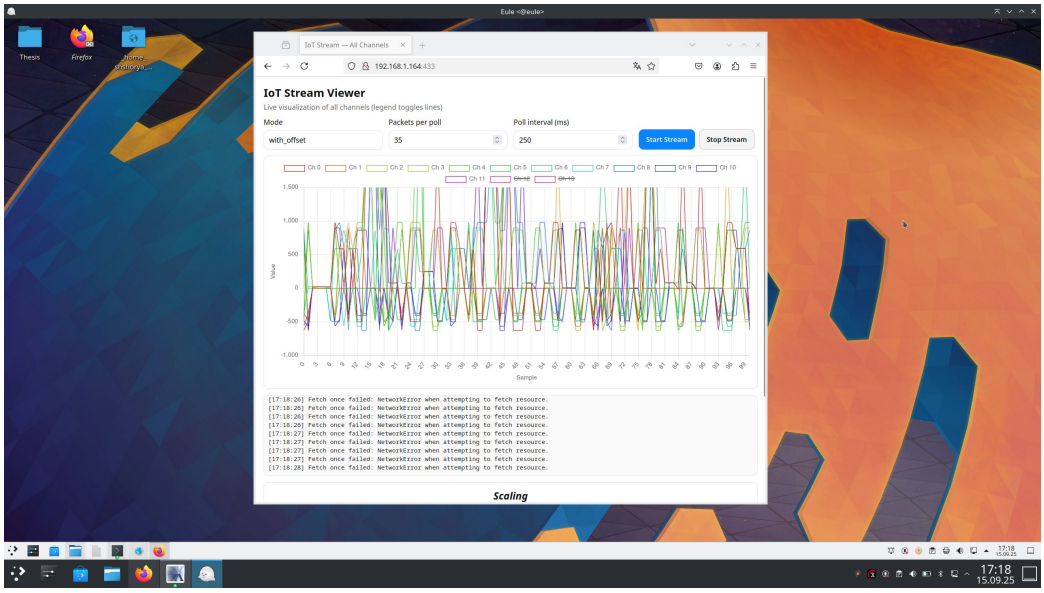


Figure 3.17: Frontend UI Responsivness to screen size

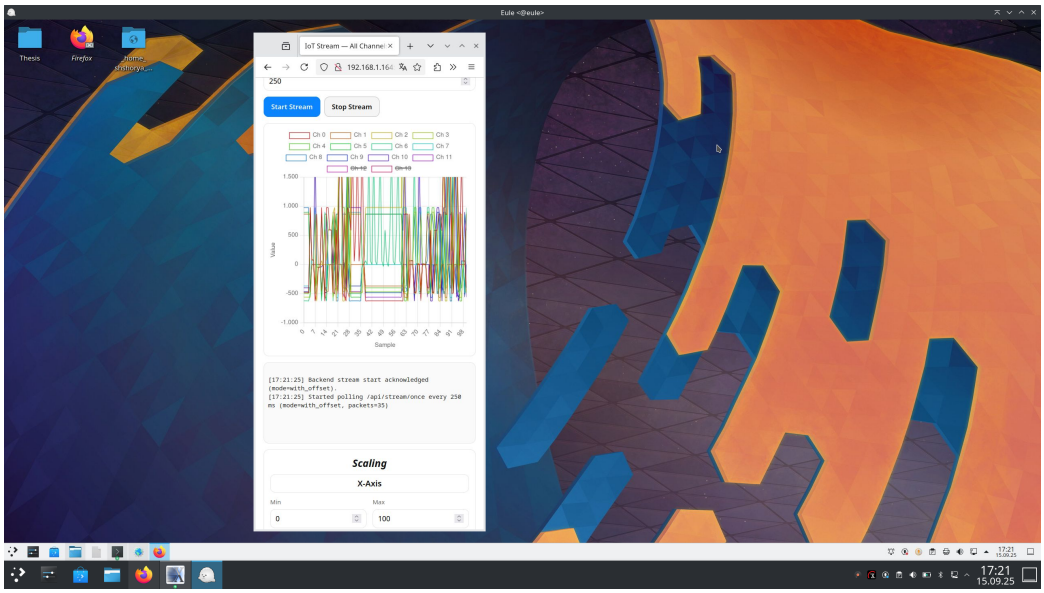


Figure 3.18: Frontend UI Responsivness to screen size

CHAPTER 4

Thesis Outcome

This chapter interprets the experiments described in Section 3.3 and draws conclusions about the effectiveness of the implemented system. In keeping with the thesis structure, Section 4.1 focuses on the evaluation of the system, while the concluding remarks and future work are presented in Chapter 5.

4.1 Evaluation

The evaluation of the HH-AMV backend and frontend is based on three experimental axes: functional correctness, request latency, and resource usage on the target SoC. The experiments follow the methodology described in Section 3.3. All data were collected on the target system configured with the custom FPGA bitstream and the backend service running on port 433. The client machine communicated with the SoC over a local network. Raw data from the experiments are provided in following subsections.

4.1.1 Functional checks and data integrity

To confirm that the backend initialises correctly and recognises the FPGA devices, the client first queried the `/api/health` endpoint 4.1. The service returned a JSON document with the fields `status`: `"ok"` and `fifo_device_present`: `true`, verifying that the *PSFIFO* character device is accessible and that the backend has successfully opened it. A subsequent call to `/api/stream/once` in the default streaming mode captured one data block from the ring buffer. The payload reported `glitches = 0` and `raw_words_len = 256` alongside twelve channel vectors of equal length 4.2. This observation indicates that the header marker, block number and channel mapping were decoded correctly and that the driver did not detect any anomalies (underflow or overrun) for that block. Together, these calls confirm that the pipeline from the FPGA ring buffer through the backend's parser and REST API to the client is operational.

Figure 4.1: Snapshot of `apihealth`.

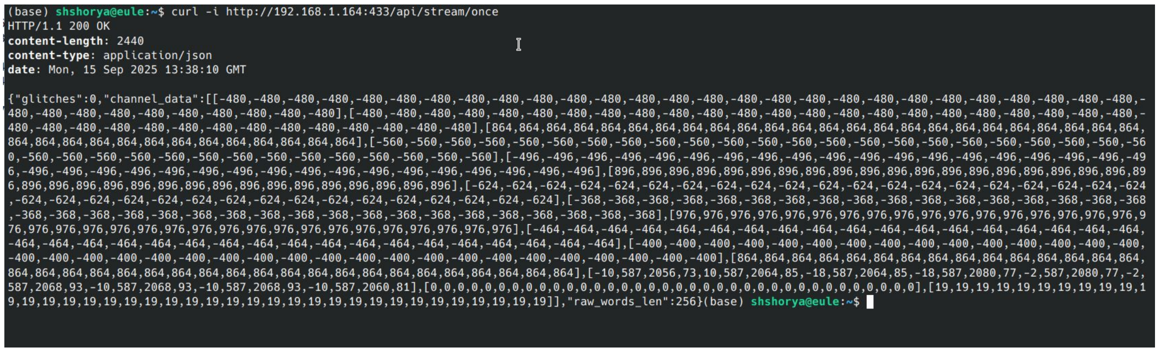


Figure 4.2: Snapshot of `api/stream/once` payload.

4.1.2 Latency of `/api/stream/once`

Latency was measured using a shell harness that issued back-to-back HTTP requests to `/api/stream/once`. Each run consisted of n requests ($n \in \{10, 15, 20, 30, 50, 60, 90, 150, 180$ $, 200, 300, 400, 500\}$) with a 500 ms pause between runs to drain the FIFO. The harness aggregated the `curl` timing statistics to compute the average ($\bar{t}$), median (p50), 90th percentile (p90) and maximum latency across the run. Table 4.1 summarises the latency distributions. Each request read twelve channels and returned a JSON payload of 256 16 bit samples per channel.

Interpretation. Across all Requests/run sizes, the median (p50) and p90 values clustered tightly between roughly 12.5 ms and 15.6 ms, while the averages remained between 13 ms and 15 ms. Occasional outliers pushed the maximum latency above 50 ms, and up to 212 ms for the longest runs, but these events were rare. The tight clustering of p50 and p90 suggests the HTTP polling path is sufficiently responsive for a front-end refresh rate of 60–80 Hz. The few long-tail outliers would manifest as sporadic frame skips in the dashboard; these could be mitigated by using a persistent streaming transport (e.g., WebSocket) or by reading larger batches per request to amortise fixed overheads.

Visualisation. For presentation, Figure **??** depicts the latency distributions with box plots. The box spans the interquartile range (p25–p75) and the median is marked by a horizontal line; whiskers extend to the minimum and maximum. Outliers above 100 ms are annotated. This figure makes it evident that most requests complete within 20 ms.

Table 4.1: Summary statistics for `/api/stream/once` latency across repeated requests. Each row aggregates n back-to-back requests.

Requests/run	$\bar{t}$ [s]	p50 [s]	p90 [s]	**Max** [s]
10	0.017468	0.013303	0.050257	0.050257
15	0.013040	0.013105	0.014619	0.014657
20	0.013987	0.014305	0.014435	0.014697
30	0.013674	0.013496	0.014369	0.015516
50	0.014760	0.014329	0.015578	0.043848
60	0.013317	0.012551	0.014322	0.042884
90	0.013977	0.014318	0.015645	0.016748
150	0.014302	0.014299	0.015552	0.048856
180	0.014004	0.014319	0.014790	0.045061
200	0.014630	0.013297	0.015466	0.166376
300	0.014462	0.013299	0.014400	0.212584
400	0.014545	0.014394	0.015581	0.054836
500	0.014353	0.013192	0.014445	0.214083

4.1.3 Resource usage on the SoC

CPU and memory usage were monitored with `top` during the experiments. Figures 4.3–4.5 show representative snapshots of the process `iot-backend` while serving HTTP requests. The backend consumed between roughly 9 % and 13 % of one CPU core and between 1.0 and 1.9 MiB of resident memory. Given that the SoC has 753 MiB total memory, the service footprint is negligible and leaves headroom for the operating system and other tasks.

4.1.4 Throughput vs. Packets per Request

Another important aspect of system performance is how throughput changes with the number of packets requested in a single call. Table 4.2 summarises the results of our benchmark, and Figure 4.6 visualises the relationship.

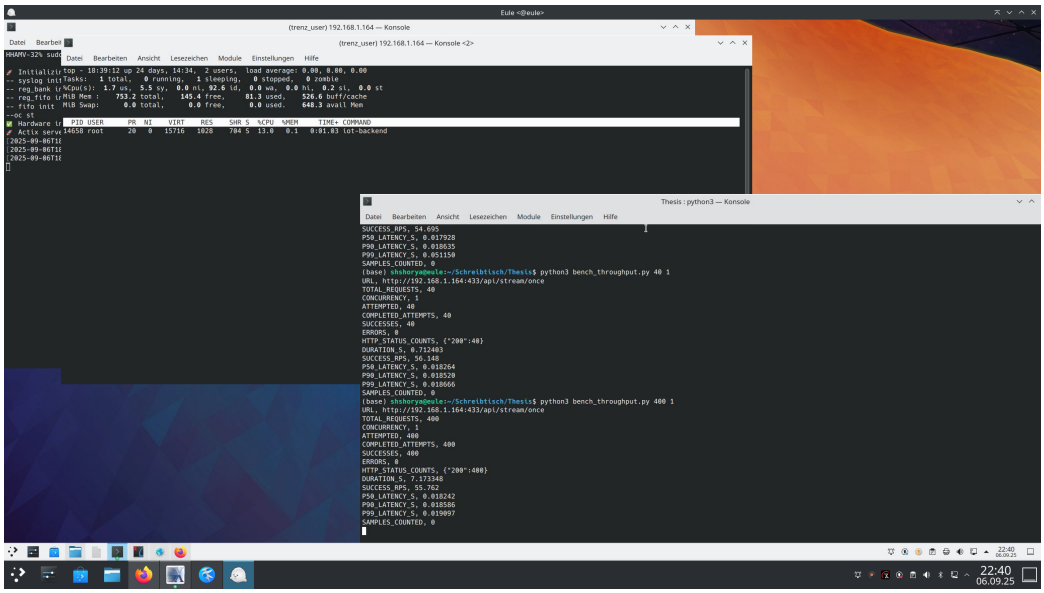


Figure 4.3: Snapshot of `top` on the SoC showing `iot-backend` using about 13 % CPU on 400 requests.

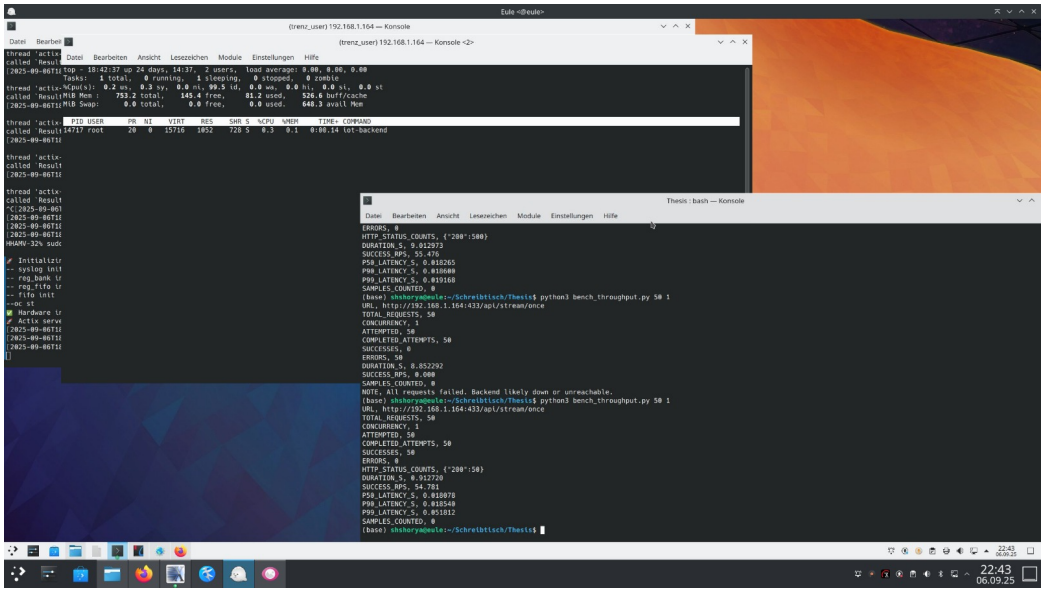


Figure 4.4: Snapshot of `top` on the SoC showing `iot-backend` using about 0.3 % CPU on 50 requests

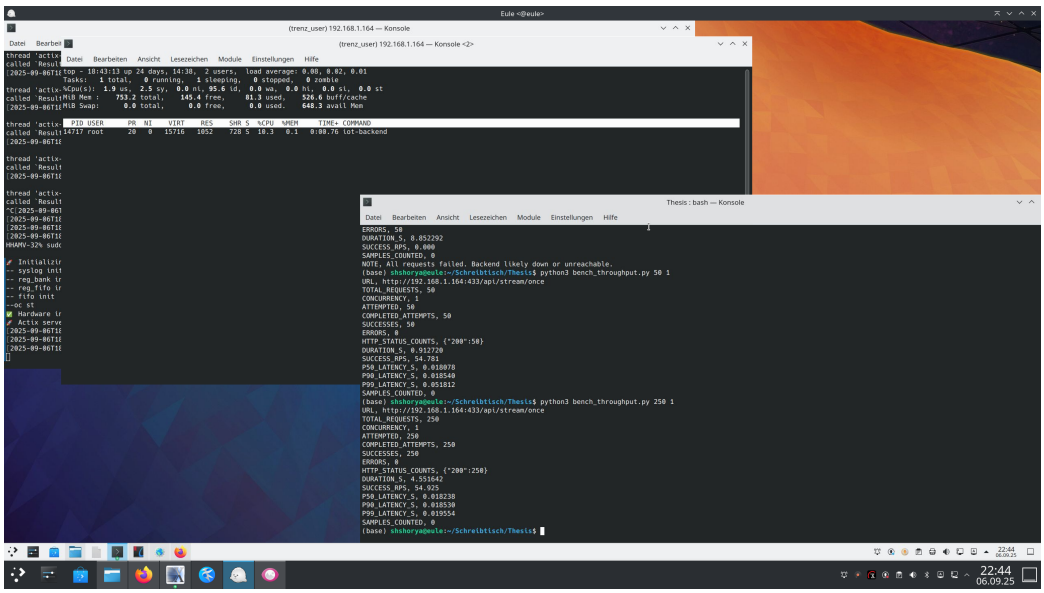


Figure 4.5: Snapshot of `top` on the SoC showing `iot-backend` using about 10.3 % CPU on 250 requests

Table 4.2: Throughput and latency percentiles for varying packets per request.

packets	throughput [RPS]	p50 [s]	p90 [s]	p99 [s]
1	52,900	0.018	0.019	0.062
2	37,000	0.027	0.028	0.028
5	18,300	0.054	0.055	0.083
10	10,000	0.100	0.101	0.101
20	5,200	0.191	0.191	0.223
50	2,100	0.470	0.470	0.471
100	1,100	0.938	0.938	0.939

Interpretation. The data show an inverse relationship: very small request sizes (1–2 packets) enable extremely high request rates (over 50 000 RPS), while larger batch sizes reduce throughput but increase per-request latency. For example, at 100 packets per request, throughput falls to about 1 000 RPS, with median latency nearly one second. This reflects the trade-off between fine-grained responsiveness and bulk transfer efficiency. For interactive visualization, smaller packet sizes yield faster response times, whereas larger sizes may be more suitable for offline or aggregated logging.

4.1.5 Implications and conclusions

Functional readiness. The health endpoint and sample capture confirm that the backend correctly discovers the devices and decodes blocks from the FIFO without errors. The parameter `glitches` remains at zero in steady-state conditions, indicating that the FIFO

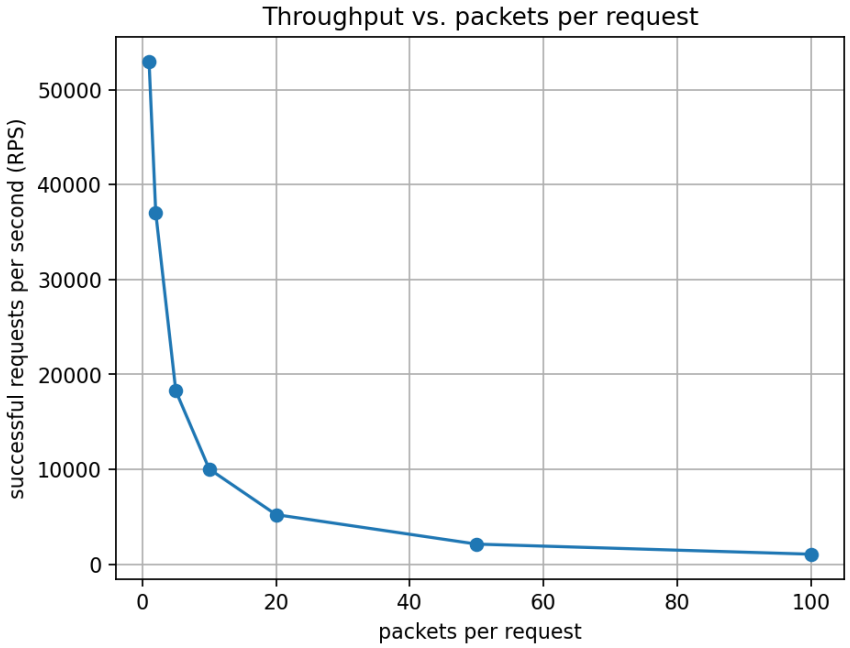


Figure 4.6: Throughput as a function of packets per request.

and parser handle streaming without overrun or underrun.

Interactivity. Request latencies of 12–16 ms (median/p90) mean that the front-end can safely poll the backend at frame rates up to about 60–80 Hz without saturating the CPU. Occasional outliers, likely caused by scheduler pre-emption or cache misses, could result in visual hiccups; switching to WebSockets or batching more samples per call would reduce these tails.

Resource footprint. The backend's CPU and memory usage remain comfortably low ($<13\,\%$ CPU, <2 MiB RAM), making the solution viable for embedded deployment on the HH-AMV SoC. The measured footprint also suggests that additional features such as downsampling or WebSocket support could be added without exhausting the platform's resources.

Limitations. These measurements represent a single device and network configuration; performance may vary with different SoC revisions, network conditions or concurrent workloads. The experiments also used HTTP polling rather than a persistent streaming channel; evaluating a WebSocket implementation would provide insights into further reducing latency and smoothing outliers. Finally, the ramp mode was not extensively tested under high sampling rates; future work should assess glitch detection and throughput in that mode.

CHAPTER 5

Conclusion

This chapter concludes the thesis by summarising the problem, the solution developed and the results obtained, and by outlining directions for future work. The previous chapters introduced the motivation for real-time data visualisation on embedded SoCs, surveyed related work, described the implementation of a custom backend and frontend for the HH-AMV architecture, and evaluated the system's functional correctness, latency and resource usage. Here, we reflect on those findings and consider how the system could be extended.

5.1 Summary

The goal of this thesis was to design and implement a lightweight, platform-independent web-based solution for visualising measurement data generated by an FPGA on a Linux-based SoC. As argued in Section 1.1, existing IoT dashboards such as Grafana and Node-RED require external databases or brokers and are therefore ill-suited to resource-constrained embedded devices. To overcome this, the thesis contribution (Chapter 3) consisted of two main components:

- A *Rust backend* running on the SoC that communicates directly with the FPGA via the `PSREG` and `PSFIFO` kernel modules. Using low-level `ioctl` calls, the service controls measurement modes, reads 256-word blocks from a ring buffer and decodes them into channel-wise sample arrays. The backend exposes REST endpoints for starting and stopping continuous streaming and for capturing bounded acquisitions, and serves static assets for the frontend. The entire service is self-contained and requires no external database or message broker.
- A *web frontend* implemented in HTML, CSS and JavaScript (Section 3.2). This single-page application fetches measurement data from the backend, renders XY curve plots for up to twelve channels, and provides UI controls for selecting the streaming mode, the number of packets per request and the polling interval. The dashboard can be accessed from any modern browser and requires no client-side installation.

To evaluate whether the solution meets its design objectives, Section 4.1 presented experiments that probed functional correctness, latency and resource usage. A health check

confirmed that the backend correctly detects the FIFO device and that a single acquisition produced a well-formed payload with zero glitches. Latency measurements using repeated calls to `/api/stream/once` showed median and 90th-percentile response times between approximately 12 ms and 16 ms (Table 4.1), with occasional long-tail outliers. CPU and memory usage captured via `top` indicated that the service occupies between 9 % and 13 % of a single CPU core and less than 2 MB of RAM.

These results support the conclusion that the implementation successfully meets the primary goal: it enables real-time visualisation of FPGA-generated data on an embedded SoC with minimal overhead. Users can view live plots in a browser without installing additional software; the service maintains low latency suitable for human-interactive dashboards; and the resource footprint is compatible with the constraints of the HH-AMV platform. The evaluation also exposes limitations: the HTTP polling mechanism occasionally incurs high-latency outliers; the ramp mode and glitch detection were not thoroughly exercised at high sampling rates; and the solution lacks advanced features such as downsampling and persistent connections. Nevertheless, the core objective of providing a lightweight end-to-end path from hardware to browser has been achieved.

5.2 Future work

Several avenues exist to extend and improve the presented system. First, while HTTP polling proved sufficient for low-latency operation, adopting a persistent transport such as WebSockets would eliminate request overhead and reduce the long-tail latencies observed in Section 4.1. A WebSocket API would allow the backend to push new samples to the frontend as soon as they are available, improving responsiveness and simplifying the client logic. Implementing this would involve adding a WebSocket endpoint to the Actix server and refactoring the JavaScript code to maintain a single connection and update plots on receipt of data frames.

Second, the current frontend plots every sample returned by the backend, which could overwhelm the browser if the sampling frequency or acquisition window increases. Integrating a downsampling algorithm such as Largest-Triangle-Three-Buckets (LTTB) would allow the frontend to render representative points while preserving visual fidelity. This change would require implementing LTTB in JavaScript and providing configuration options for the sampling density.

Third, the evaluation focused on functional correctness and latency but did not compare the system quantitatively with off-the-shelf platforms. Future work could involve deploying Grafana or Node-RED on the same SoC (with an external TSDB if necessary) and measuring end-to-end latency, resource usage and ease of deployment. Such a comparison would more fully substantiate the advantages of the custom solution.

Fourth, an alternative integration path would be to publish measurement data from the FIFO to an MQTT broker, rather than exposing it only via a REST or WebSocket API. In such a design, the Rust backend would act as a lightweight bridge, decoding hardware samples and pushing them into MQTT topics. Standard dashboards such as Grafana, ThingsBoard or Node-RED could then subscribe to these topics without custom integration. This would align the system more closely with mainstream IoT practices, improve interop-

erability with existing monitoring platforms, and enable multi-subscriber use cases where several clients consume the same stream simultaneously. However, this approach would also introduce additional overhead, and its feasibility on an ARM-based SoC with limited resources would require careful benchmarking. Exploring such a hybrid design therefore represents a natural next step.

Fifth,additional hardware features could be exposed via the backend, such as adjusting amplifier gains, switching filter banks, or enabling high-resolution modes described in the register map. Incorporating these controls would increase the utility of the dashboard for engineers and researchers.

Finally, from a robustness and usability perspective, future versions should incorporate authentication and authorisation mechanisms, log rotation and structured logging, and a more modular frontend architecture. Comprehensive testing, including stress tests on the ramp mode and under high sampling rates, would help ensure reliability. Together, these enhancements would transform the prototype into a production-ready visualisation tool for embedded measurement systems.

Bibliography

[1] Statista. *Number of Connected IoT Devices Worldwide from 2015 to 2025*. Accessed ccessed August 1, 2025. 2025. URL: https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/.

[2] Grafana Labs. *Grafana Documentation*. https://grafana.com/docs/. Accessed August 5, 2025. 2025.

[3] Node-RED Community. *Node-RED - Low-code programming for event-driven applications*. https://nodered.org/. Accessed August 5, 2025. 2025.

[4] Sveinn Steinarsson. "Downsampling Time Series for Visual Representation". M.Sc. thesis. Reykjavik, Iceland: University of Iceland, 2013.

[5] openHAB Foundation e.V. *openHAB Documentation*. https://www.openhab.org/docs/. Accessed September 10, 2025. 2025.

[6] FHEM. *FHEM Home Automation*. https://fhem.de/. Accessed September 10, 2025. 2025.

[7] Wireshark Foundation. *Wireshark Documentation*. https://www.wireshark.org/docs/. Accessed September 10, 2025. 2025.

[8] Grafana Labs. *Grafana Documentation*. Accessed August 5, 2025. 2025. URL: https://grafana.com/docs/.

[9] Elastic. *The Elastic Stack*. https://www.elastic.co/docs/get-started. Accessed August 5, 2025. 2025.

[10] Sourav Choudhary. *ElasticSearch Aggregation & Queries*. Accessed August 6, 2025. 2023. URL: https://medium.com/@souravchoudhary0306/elasticsearch-aggregation-queries-557131ef5ea4.

[11] ThingsBoard Authors. *ThingsBoard IoT Platform*. Accessed August 6, 2025. 2025. URL: https://thingsboard.io/docs/.

[12] Node-RED. *Running on Raspberry Pi*. Accessed August 7, 2025. 2025. URL: https://nodered.org/docs/getting-started/raspberrypi.

[13] Node-RED Flows Community. *node-red-dashboard (deprecated)*. Accessed August 10, 2025. 2023. URL: https://flows.nodered.org/node/node-red-dashboard.

[14] George Alex Stelea, Livia Sangeorzan, and Nicoleta Enache David. "Accessible IoT Dashboard Design with AI-Enhanced Descriptions for Visually Impaired Users". In: *Future Internet* 17.274 (2025). Ed. by Rui Yu et al., p. 274.

[15] Cristian D'Ortona, Daniele Tarchi, and Carla Raffaelli. "Open-Source MQTT-Based End-to-End IoT System for Smart City Scenarios". In: *Future Internet* 14.57 (2022). Ed. by Joel J. P. C. Rodrigues.

[16] OASIS. *MQTT Version 5.0*. Accessed August 10, 2025. 2019. URL: https://docs.oasis-open.org/mqtt/mqtt/v5.0/.

[17] IETF. *The WebSocket Protocol (RFC 6455)*. Accessed August 12, 2025. 2011. URL: https://datatracker.ietf.org/doc/html/rfc6455.

[18] InfluxData. *InfluxDB Documentation*. Accessed August 16, 2025. 2025. URL: https://docs.influxdata.com/influxdb/.

[19] Timescale, Inc. *TimescaleDB Documentation*. Accessed August 16, 2025. 2025. URL: https://docs.timescale.com/.

[20] TDengine Authors. *TDengine Time-Series Database*. Accessed: August 16, 2025. 2025. URL: https://docs.tdengine.com/introduction/.

[21] Jonas Van Der Donckt et al. "Plotly-resampler: Effective visual analytics for large time series". In: *2022 IEEE Visualization and Visual Analytics (VIS)*. IEEE. 2022, pp. 21–25.

[22] Jeroen Van Der Donckt et al. "MinMaxLTTB: Leveraging MinMax-Preselection to Scale LTTB". In: *arXiv* 2305.00332v1 (2023). IDLab, Ghent University - imec, Belgium. cs.HC: 2305.00332.

[23] Chart.js Community. *Chart.js Documentation*. Accessed August 18, 2025. 2025. URL: https://www.chartjs.org/docs/.

[24] Chart.js Community. *chartjs-plugin-streaming*. Accessed August 18, 2025. 2023. URL: https://nagix.github.io/chartjs-plugin-streaming/.

[25] Highsoft AS. *Highcharts JS*. Accessed August 18, 2025. 2025. URL: https://www.highcharts.com/docs.

[26] Apache Software Foundation. *Apache ECharts*. Accessed August 18, 2025. 2025. URL: https://echarts.apache.org/.

[27] Leon Sorokin. *uPlot: A fast, lightweight chart library*. Accessed August 18, 2025. 2025. URL: https://github.com/leeoniya/uPlot.

[28] Thomas Williams, Colin Kelley, and many others. *Gnuplot 4.4: an interactive plotting program*. http://gnuplot.info/. Accessed September 12, 2025. 2010.

[29] Antonis I. Protopsaltis et al. "Data Visualization in Internet of Things: Tools, Methodologies, and Challenges". In: *Proceedings of the 15th International Conference on Availability, Reliability and Security*. ARES 2020. Virtual Event, Ireland: ACM, 2020.

[30] Cuili Shao et al. "IoT data visualization for business intelligence in corporate finance". In: *Information Processing & Management* 59.1 (2022), p. 102736.

[31] Márcio Miguel Gomes, Cristiano André da Costa, and Dalvan Griebler. "STEAM++ An Extensible End-to-end Framework for Developing IoT Data Processing Applications in the Fog". In: *International Journal of Computer Science and Information Technology* 14.1 (2022). Ed. by Rodrigo da Rosa Righi. URL: https://www.researchgate.net/publication/359201187.

Appendix

A.1 Sourcecode Fifo Stream

```
// Read the packet payload: number of 32-bit words per packet (excl. header)
        let mut words_needed: usize = settings::FIFO_STREAM_DATA_WORDS_PER_PACKET;
        let packet_deadline = Instant::now() + Duration::from_millis(100);

        while words_needed > 0 {
            let occ = fifo_occupancy(fifo);
            if occ == 0 {
                if Instant::now() >= packet_deadline { break; }
                sleep(Duration::from_millis(1));
                continue;
            }

            let to_read: usize = std::cmp::min(occ as usize, words_needed);
            for _ in 0..to_read {
                // If first_channel == 0, store the measurement number (as i16) aligned to special index
                if header.first_channel == 0 {
                    channel_data[settings::FIFO_NUM_STREAM_CH as usize].push(meas_nr as i16);
                }

                let word = reg_fifo::get_axi4_str_rdfd(fifo);
                raw_words.push(word);
                let b = word.to_le_bytes();

                // Two 16-bit samples packed little-endian
                let value0 = i16::from_le_bytes([b[0], b[1]]);
                let value1 = i16::from_le_bytes([b[2], b[3]]);

                // Push value0 into current channel
                if (header.first_channel as usize) < channel_data.len() {
                    if option == flags::STREAM_GLITCHES_TEST {
                        glitches += glitch_if_out_of_sequence(&channel_data[header.first_channel as usize], value0);
                    }
                    channel_data[header.first_channel as usize].push(value0);
                }

                // Advance channel index (wrap on total number of stream channels)
                header.first_channel = header.first_channel.wrapping_add(1);
                if header.first_channel >= settings::FIFO_NUM_STREAM_CH {
                    header.first_channel = 0;
                }

```

```
42                 // Push value1 into (possibly next) channel
43                 if (header.first_channel as usize) < channel_data.len() {
44                     if option == flags::STREAM_GLITCHES_TEST {
45                         glitches += glitch_if_out_of_sequence(&channel_data[
                               header.first_channel as usize], value1);
46                     }
47                     channel_data[header.first_channel as usize].push(value1);
48                 }
49
50                 // Advance again for next word placement
51                 header.first_channel = header.first_channel.wrapping_add(1);
52                 if header.first_channel >= settings::FIFO_NUM_STREAM_CH {
53                     header.first_channel = 0;
54                 }
55
56                 words_needed -= 1;
57                 if words_needed == 0 { break; }
58             }
59         }
```

FIFO Stream in Rust

I herewith assure that I wrote the present thesis titled *Data Visualization for IoT* independently, that the thesis has not been partially or fully submitted as graded academic work and that I have used no other means than the ones indicated. I have indicated all parts of the work in which sources are used according to their wording or to their meaning.

I am aware of the fact that violations of copyright can lead to injunctive relief and claims for damages of the author as well as a penalty by the law enforcement agency.

Magdeburg, September 30, 2025

(Shashank Shorya)