Politecnico di Torino

Master’s degree Program in

Digital Skills for Sustainable Societal Transitions

Master's Degree Thesis

Event-Driven microservices platform for Structural Health Monitoring of multiple systems

Supervisor:

Prof. Rosario Ceravolo

Co-Supervisors:

Prof. Gianvito Urgese

Prof. Gaetano Miraglia

Candidate:

Kamran Mosleh Khorrami

Academic Year:

2025-2026

To Sabri,

My love and companion, without whom my journey wouldn't have started!

Abstract

Structural Health Monitoring (SHM) today represents a fundamental tool for the continuous evaluation of the performance level of civil structures and infrastructures throughout their life cycle. In recent years, several advanced techniques have been developed to address the different phases of monitoring, including the optimization of the testing setup, the denoising and pre-processing of signals, the identification of mechanical properties, damage detection based on data-driven approaches, and integration with digital twinning procedures. However, such methodologies are often applied in a fragmented way, through heterogeneous tools, making their operational integration and long-term evolution complex. In real and long-term monitoring contexts, the need to modify, update or replace algorithms and analysis techniques is highly probable, due to changes in operating conditions, in the instrumentation configuration or in monitoring objectives, but also due to the evolution of research in this field. In the absence of an adequate framework, such modifications often require invasive interventions on the entire IT system, with negative consequences in terms of its maintainability and update.

This thesis addresses this issue by proposing the definition and implementation of a monitoring platform based on an event-driven microservices architecture, i.e., independently deployable services that progress the processing workflow via asynchronous events (messages), conceived as a framework for the decoupled implementation of the different phases of the SHM process. The objective is not limited to the management of data acquisition, but to provide a software infrastructure that enables structured management of data ingestion, pre-processing, analysis and visualization, allowing the independent updating of analysis techniques over the service life of the monitored structure.

The platform is designed according to a local-first approach, i.e., the platform runs on a single local machine via containerization, using local storage and local user interfaces without requiring cloud services for normal operation, and developed with independent services. In particular, the analysis is implemented as an autonomous service, so as to allow the introduction, replacement or extension of algorithms without impacting the rest of the system. The framework also supports operational workflows typical of structural monitoring, including automated data ingestion, management of pre-processing parameters, traceability of re-elaborations and visualization of results. Overall, the work provides a reproducible and maintainable basis for the development and experimentation of platforms for SHM, emphasizing the importance of software architecture as an enabling element for the effective and sustainable application of structural monitoring techniques in the long term, including the possibility of managing the monitoring of multiple structures simultaneously within a single environment.

Keywords: Structural Health Monitoring (SHM), microservices, event-driven architecture, ambient vibration, signal preprocessing, modal analysis, time-series database, metadata store, Grafana dashboards, Docker Compose

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Prof. Rosario Ceravolo, for his guidance, support, and valuable feedback throughout this thesis. I am also deeply grateful to my co-supervisors, Prof. Gianvito Urgese and Prof. Gaetano Miraglia, for their insightful suggestions and continued assistance during the development of this work.

Contents

1. Introduction and motivation

1.1. Why SHM needs “data systems” not only “algorithms.”

Structural Health Monitoring (SHM) is often introduced through the lens of damage detection algorithms - modal identification, novelty detection, statistical classifiers, or model updating. However, in operational settings, the limiting factor is frequently not the choice of algorithm, but the ability to acquire, curate, store, contextualize, and serve data reliably over long-time horizons. In other words, SHM is a data-to-decision process whose weakest link is often upstream of analytics.

A useful way to see this is like a “pipeline”: when the sensor network exists, SHM follows some consecutive steps: sensing, pre-processing, feature extraction, post-processing, pattern recognition, and decision making. This is often summarized as “data-to-decision” [1]. This viewpoint shows that algorithms are just one element in a larger practical chain, where the foundation is data handling.

As stated in Chapter 1 of the book “Structural Health Monitoring: A Machine Learning Perspective”, data cleansing, normalisation, compression and fusion can be part of the data acquisition process as well as the feature selection and statistical modelling portions of the SHM process [1].

Sensor fusion is the process of integrating data from a multitude of sensors with the objective of making a more robust and confident decision than is possible with any one sensor alone. There are numerous reasons why multi-sensor systems are desirable.

In the next chapters it will be shown how for an initial deployment the Centralized fusion architecture (case B in Figure 1.1) is used and how it can develop into either Pattern-level fusion architecture (case C) or arbitrarily connected sensors fusion tree (case D).

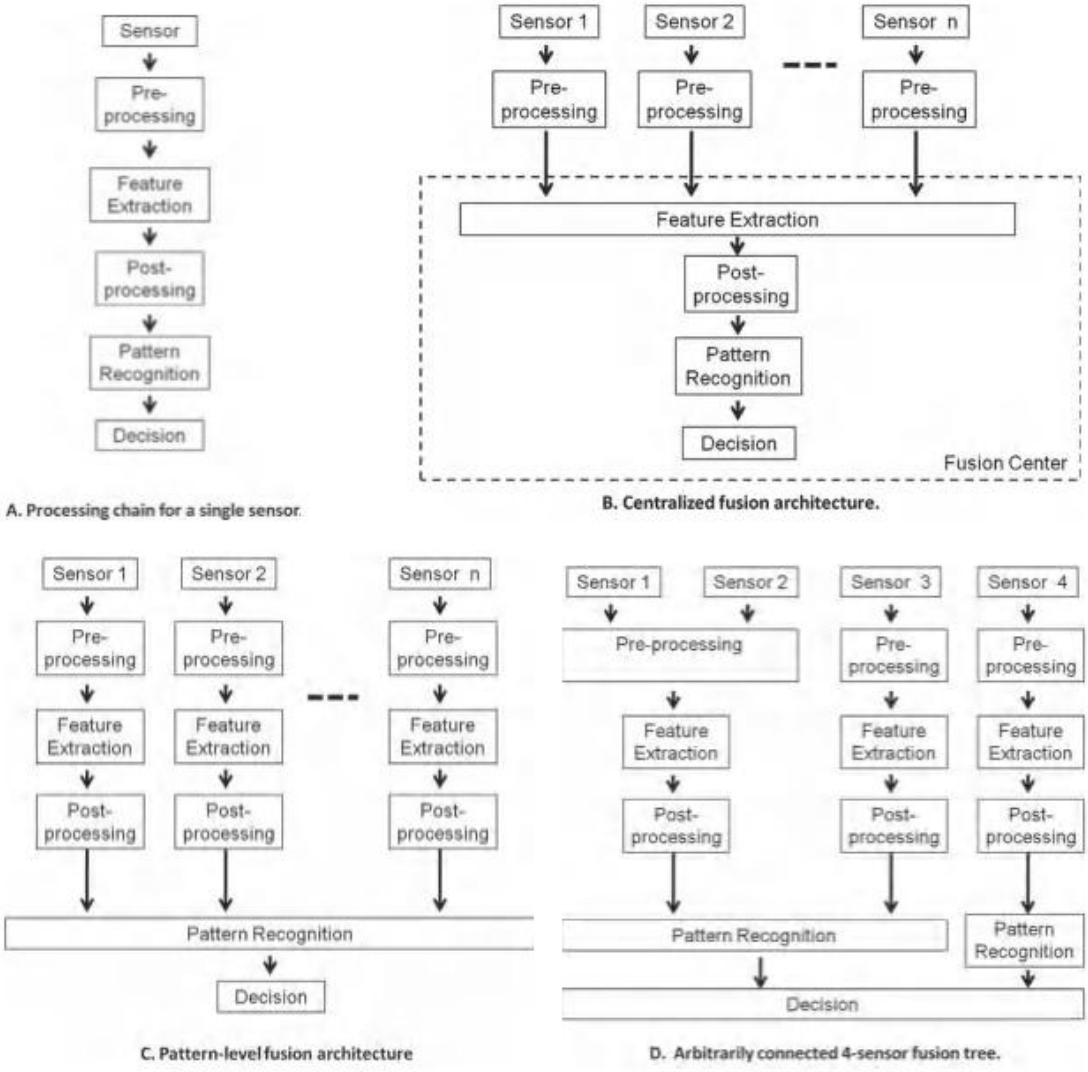


Figure 1.1 Sensor fusion strategies [1]

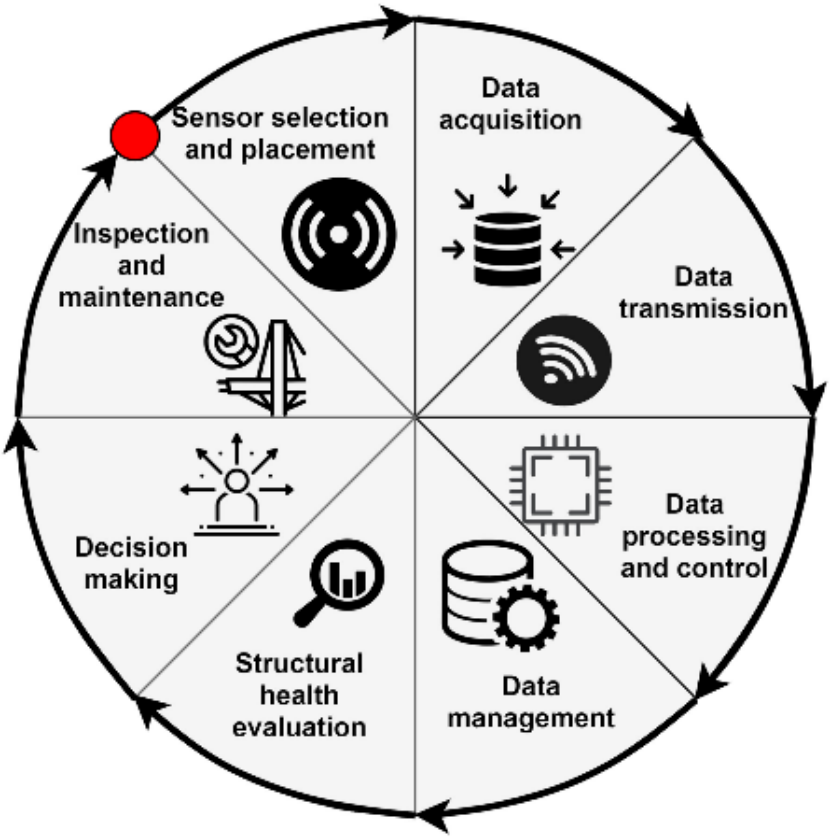


Figure 1.2 End-to-end SHM workflow ("data-to-decision"), highlighting the supporting data system functions (acquisition, metadata, storage, provenance, and delivery) [2].

1.1.1. Continuous monitoring implies continuous data operations

Long-term monitoring produces continuous (or regularly repeated) measurements, which means SHM quickly becomes an operational data problem: time synchronization, data loss handling, retention policies, and scalable storage. Classic SHM references explicitly note that operational implementations "inherently produce large amounts of data," making data reduction and robust handling necessary - especially when comparisons over the lifetime of a structure are required. In aerospace examples, it is similarly observed that large volumes of in-service measurements "necessitate a robust data management strategy."

This is why mature SHM system design discussions treat data archiving and telemetry as design requirements, not implementation details. In the sensing and acquisition design context, the ability to archive data consistently and retrievably for long-term monitoring and to transmit condition information are listed among the conceptual challenges that must be solved for effective SHM [1].

The motivation for this thesis is that the platform adds more than a software engineering tool. It is about supporting the main promises of SHM, like trendability, lifecycle comparisons, alarm auditing, and reliable long-term performance.

1.1.2. Reproducibility depends on effective data cleaning and management

Two SHM teams may use the same algorithm but get different results if their steps, such as sampling, filtering, windowing, sensor calibration, baseline setting, and missing-data policies, are different. The SHM practices emphasise that preprocessing is very important and not just cosmetic. Data cleaning requires carefully selecting which data to reject or correct, often based on knowledge of the acquisition environment, such as sensor mounting issues. Without documenting and standardising these decisions, the results cannot be reproduced accurately, even if the resulting model is well implemented.

Likewise, data normalisation is a core requirement because real structures are influenced by environmental and operational variability. The book [1] emphasises that without normalisation, benign variability may be misinterpreted as damage, and that normalisation issues are often "key challenges to the field deployment of a robust SHM system." This point is strongly echoed in a peer-reviewed review paper by Sohn (Philosophical Transactions A, 2007), which explains that environmental/operational variations can mask subtle damage-induced changes, motivating normalisation procedures to separate these effects [3].

From a "data system" perspective, preprocessing choices are viewed as versioned, analysable, and reproducible transformations that include stored parameters and are clearly defined as a family, rather than being mere scripts.

1.1.3. Traceability is required for auditability and for lifecycle management

SHM outputs can trigger actions, from extra inspections to a partial closure of infrastructure. For people to trust these outputs, every reported indicator should be traceable back to:

- the raw measurement interval, like sampling rate or data status
- the sensor setup and calibration
- the parameters used for preprocessing and feature extraction, including the software version
- the decision rule, based on statistics and the threshold that was applied

The SHM process description makes clear that data acquisition, cleansing, compression, fusion, and normalization are distinct and interrelated steps that shape what "features" mean in practice [1]. This is not merely an academic decomposition: traceability requires that each step be logged and reproducible so that an engineer can answer: "Why did the system raise this alert on that date?"

From a system-design standpoint, the same SHM reference that highlights algorithmic sophistication also emphasizes infrastructural requirements such as consistent archiving and the ability to transmit condition information to maintenance/control systems [1]. Those requirements directly translate into platform features: structured storage, metadata management, provenance tracking, and reliable dissemination (dashboards, APIs, reports).

This thesis describes the development of a modular SHM data platform. It ingests, logs, and time records measurement data streams, keeps metadata and data provenance, and provides standard interfaces for analysis and visualization.

1.1.4. Implications on the project implementation

SHM is an ongoing, longitudinal process, and its goals are achieved through trends over months or years rather than isolated analyses. This makes data management essential. Effective monitoring depends on managing environmental and operational effects through normalisation and consistent preprocessing, which should be established as repeatable data workflows. Responsibility in engineering requires traceability, with systematic archiving and transmission of monitoring data, as an essential challenge in SHM. A platform that organises these processes directly supports the implementation of SHM in the field.

1.2. Reference case: Vicoforte and Structural X monitoring project

This thesis uses the long term monitoring of the Sanctuary of Vicoforte as a case study. The system produces vibration data often and over long periods. The main value is that the data become comparable, because we apply consistent preprocessing, we keep analysis parameters traceable, and we use a method to track modal parameters. The platform presented here aims to automate this operational process. It does not try to replace structural models or expert judgement. The main goal is to reduce manual work between data acquisition and analysis, so trends in frequency, damping, and mode shape similarity can be checked with clear data provenance and less need for manual scripts.

A concrete example that motivates this thesis is the long-term monitoring of the Sanctuary of Vicoforte, also known as Regina Montis Regalis, which is a large monumental masonry building in Northern Italy. Here, monitoring is not treated as a short diagnostic campaign. It is a permanent infrastructure that supports preservation, safety assessment, and maintenance decisions over long time horizons. The structure has significant settlement and complex cracking patterns because of its massive weight. Its scale and historical importance make it an important case study for advanced, automated Structural Health Monitoring and modal identification [4].

Figure 1.3 Case-study context photo of the Sanctuary of Vicoforte (external view) (Vicoforte, Piedmont, Italy). [5]

1.2.1. From monitoring “campaigns” to a permanent observatory

The Vicoforte program is frequently cited as one of the most significant long-term SHM initiatives for Italian cultural heritage, with monitoring activities dating back decades. A conference paper describing the program frames it as a major monitoring effort that began in the early 1980s for conservation purposes [4]. This long-term perspective matters because it shifts the goal from “detect damage now” to “build a reliable history of structural behavior,” where trendability and comparability are central.

1.2.2. Real-time dynamic acquisition: distributed sensing and synchronized logging

A key element of the Vicoforte monitoring concept is permanent dynamic monitoring based on ambient vibrations (output-only). In the Vicoforte case, the dynamic system uses a network of twelve accelerometers installed across the structure to measure vibrations during normal operational conditions.

In fact, the acquisition is set up to establish repeatable measurement windows, such as scheduled acquisitions at a consistent sampling rate, and to enable remote data transfer to stakeholders [4].

At the system level, the “Structural-X” concept is relevant because it couples (i) field hardware for continuous acquisition, (ii) on-site supervision, and (iii) network connectivity for remote access and near-real-time use. In the Vicoforte deployment description, Structural-X is presented as a LabVIEW-based platform with an embedded component for low-level acquisition and a client component for configuration and supervision, designed to allow data access over an internet connection [6].

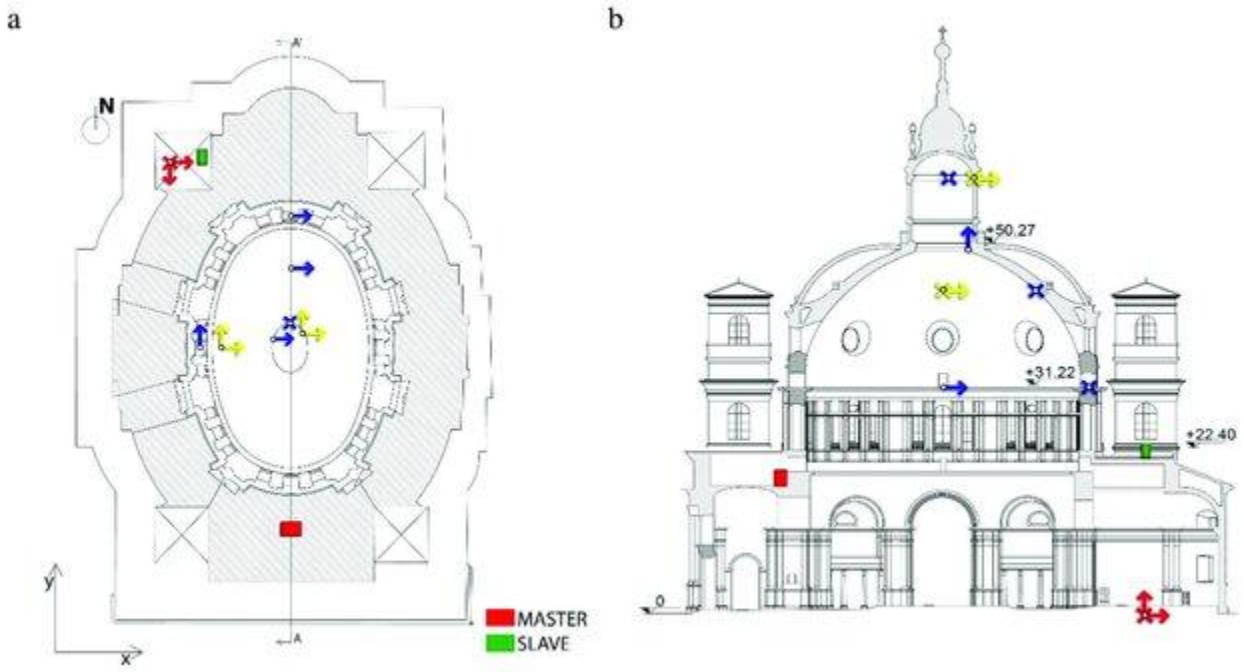


Figure 1.4 Sensor layout at Vicoforte: accelerometer locations [7] (field nodes → master controller → on-site PC → remote lab)

1.2.3. Long-term storage is not a by-product: it enables science and operations

A permanent monitoring system inevitably produces large datasets. The Vicoforte concept explicitly leverages long-term storage not only for archiving raw time histories but also for building reduced “diagnostic histories” (e.g., time series of modal frequencies). This aligns with the broader literature on operational SHM of monumental buildings, where continuous

monitoring provides an evidence base for understanding structural behavior across seasons, operational states, and extraordinary events.

1.2.4. Modal tracking as a "compressed representation" of structural behavior

In permanent dynamic monitoring, modal parameters (natural frequencies, damping ratios, mode shapes) are commonly treated as compact descriptors that can be tracked over time. For Vicoforte specifically, an automatic modal identification routine has been developed to run online and continuously extract modal parameters from ambient vibration data, forming the basis for long-term modal tracking [8].

This is especially important for monumental masonry, because detailed information about material properties and boundary conditions is often rare. In this case, techniques like output-only identification and long-term modal tracking help us analyze and interpret these structures better.

1.2.5. Expansion beyond accelerometers: integrating environmental and static sensing

A mature SHM installation tends to evolve from "dynamic-only" monitoring toward multi-physics, multi-rate sensing: combining accelerometers with static sensors (crack opening, displacement/tilt, strain) and environmental measurements (temperature, humidity). In the Vicoforte monitoring program, literature highlights the systematic role of environmental variables in long-term monitoring and analyses correlations between environmental time series and monitoring data at building scale [4].

Meanwhile, the Vicoforte deployment narrative highlights upgrades that include static instruments such as crack gauges, tilt or displacement sensors, temperature sensors, and strain gauges, as well as the dynamic measurements. This approach reflects the operational requirement to interpret modal variations considering structural state indicators [8].

1.2.6. The significance of this reference context here

The Vicoforte/Structural-X project serves as a helpful benchmark because it clarifies the requirements for a practical SHM system:

- Near-real-time data acquisition and remote access, because operators need to be able to monitor the system without being on site.
- Long-term storage and provenance (so that multi-year trends and post-event analyses are possible) [4].

- Modal tracking and automated feature extraction provide the possibility of a simple and clearly interpretable insight derived from high-frequency vibration data. [8].
- Supplementing different sensing methods to address environmental confounding and supplement rapid indicators for slower structural measurements. [4].

This project focuses on the software and data parts needed to run SHM workflows. It includes automated raw file ingestion, preprocessing, organised storage of raw and calculated data, and visualisation.

1.3. Problem statement

Operational SHM deployments produce large volumes of time-series data and require comparisons over long periods (weeks to years). The SHM "statistical pattern recognition" (SPR) workflow explicitly includes stages such as data cleansing, normalization, compression, and fusion before feature extraction and decision-making, and notes that operational implementations "inherently produce large amounts of data," making condensation and robust data reduction necessary for lifetime comparisons [1] [9].

Environmental and operational variability (EOV) also affects field measurements. Factors like temperature, traffic, and operating conditions can change the measured features. Sometimes these changes are similar to, or even larger than, the changes caused by damage. SHM literature says that normalizing the data, so we can separate harmless activity from real damage, is a key challenge in real world use. Peer reviewed studies also show that EOV can hide small damage signals, so managing it well is important for reliable damage detection [1] [3].

Another practical challenge is heterogeneity. Even within a single structure, monitoring frequently involves various sensor types and time scales—high-rate accelerometers for dynamic data, alongside lower-rate measurements like temperature, crack opening, displacement/tilt, or strain. The SHM pipeline explicitly addresses the integration of heterogeneous data (such as acceleration with temperature and operational parameters) through data fusion to enhance accuracy and interpretability [1]. Real world deployments, like the Vicoforte monitoring program, demonstrate this progression towards integrating dynamic, static, and environmental sensing within a single monitoring system. [4].

In many laboratory or early-stage deployments, the data lifecycle is handled through manual scripts and ad hoc file management (copying, renaming, filtering, windowing, exporting results). But SHM results can be highly sensitive to upstream choices (filter cutoffs, detrending order, window selection, baseline definition, missing-data handling). If these

decisions are not standardized and logged, two analyses of the "same" dataset can yield different indicators and alarms - reducing trust and making it difficult to audit decisions. Data cleansing itself is often guided by operator knowledge and judgment (e.g., rejecting records from a poorly mounted sensor), which further motivates capturing provenance so that decisions remain reviewable.

1.4. Thesis goal and scope

Motivated by the operational challenges above - data volume, variability, heterogeneity, and the need for reproducibility - this thesis aims to develop a local-first SHM data platform that turns exported monitoring files into a consistent, traceable "data-to-indicators" pipeline [3].

The platform is designed to automate the following steps:

1. Ingest monitoring raw txt files into a database, while recording the metadata needed for retrieval and auditing, such as structure, sensor, and acquisition parameters.
2. Preprocessing, or cleaning, acts as a repeatable transformation step. It can include filtering, detrending, or changing the sampling rate. This matches SHM practice, where preprocessing includes data cleansing and, when possible, steps that support reliable comparisons [1].
3. Feature/Index computation as a controlled reduction of high-rate time histories into low-dimensional indicators suitable for trend monitoring and alert logic (consistent with the SHM goal of condensing data while retaining sensitivity) [3].
4. Event generation uses clear decision logic, such as novelty or thresholding. It stores the outputs in a queryable format for future use and inspection.
5. Observability means we can see the pipeline status and results using dashboards or an operator UI.

Scope boundaries are set to remain realistic within delivery limits and to meet field SHM requirements. The focus is on automating data workflows, ensuring data provenance, and developing usable baseline indicators, rather than on fully replacing advanced modal-identification pipelines. Environmental effect considerations and more sophisticated analytics are considered as planned extensions. [3].

1.5. Contributions

This thesis contributes a practical SHM software system aligned with the SHM pipeline stages emphasized in the literature (acquisition → cleansing/ normalization/ compression/ fusion → features → decision).

main contributions:

- An automated, modular SHM processing tool, built as some loosely coupled services. This reduce the need for manual intervention and it improves consistency in preprocessing and analysis runs.
- A low-cardinality time-series data model for storing raw, cleaned, and analysed streams in a way that supports long-term querying and avoids schema explosion, reflecting the need for condensation and tractable long-term comparisons in operational SHM.
- It provides traceability and reproducibility through organized run logs, including uploads, cleaning, and analysis, and also through recorded event data. This supports alarm management and lets us reconstruct the processing settings, which is important because of variability and human cleansing decisions.
- Observability is provided through dashboards in Grafana and through APIs. This lets operators monitor pipeline performance and results. The setup fits real deployments, since it supports near real-time access and long-term storage of key diagnostic histories.
- Performance level additions
    - Add an operator UI to browse structures and sensors, check ingestion status, and view the output results.
    - Add containerized deployment using Docker Compose. This way the platform is reproducible and easy for demos and later extension.

The next chapter builds the technical background required to justify the chosen preprocessing and operational modal analysis approach, while keeping the discussion focused on what is actually implemented in current implementation.

2. Background: SHM principles and signal-processing toolbox

2.1. SHM fundamentals and the “axioms” that drive system design

This chapter introduces a small set of SHM principles that directly affect software design. Instead of presenting the axioms as abstract statements, they are used here to motivate concrete platform requirements: baseline management, explicit preprocessing logs, sensor/context metadata, and reproducible feature extraction. These needs are widely reported in SHM practice, especially when environmental and operational variability is present.

As mentioned in chapter 1, SHM can be framed as a “data-to-decision” process: sensors measure structural response, and only through signal processing + feature extraction + statistical inference can the system produce information about damage. A widely used way to summarize hard-won lessons from the field is the set of fundamental axioms of SHM proposed by Worden, Farrar, Manson, and Park and later reiterated in standard SHM references [10].

These Fundamental Axioms of Structural Health Monitoring (SHM) are basic principles that come from research experience and studies in the field. The axioms are meant as the basis for future SHM research, development, and applications

2.1.1. Why axioms matter here

In this thesis, the axioms are used as design constraints: they explain why a platform must support baseline handling, traceable preprocessing, and the integration of environmental context, rather than treating these as optional “engineering details.”

2.1.2. The eight axioms and what they imply for the platform

Axiom I - All materials have inherent flaws or defects

This axiom states that defects are always present, and that SHM ultimately concerns identifying when degradation becomes relevant for function/safety.

Platform implication: the platform should track long term trends and support reasoning about change over time, not only run single checks. This matters because, in practice, the main question is usually how the system evolves, not just if something is detected at one moment.

Axiom II - Damage assessment requires a comparison between two system states (baselines)

This is the basic requirement. Any damage inference needs a comparison to a reference state, such as a measured baseline, a learned normal model, or an implicit physics baseline. The book explains that claims like “no baseline” are often a misunderstanding of the term, because even novelty detection still depends on training data that represents the normal condition.

Platform mapping, main core:

- Baseline storage and versioning: store baseline IDs and the preprocessing parameters used to build them.
- Baseline traceability: every computed index or event should reference the baseline that was used.
- Support baseline updates: after maintenance, the system will need a new baseline.

Axiom III – Existence or location can be unsupervised; while, type or severity generally needs supervised learning

Identifying the existence and location of damage can be done in an unsupervised learning mode, but identifying the type of damage present and the damage severity can generally only be done in a supervised learning mode. Unsupervised learning algorithms, such as novelty detection, can be used to identify the existence and location of damage when data from only the undamaged structure is available (they can identify deviations from normal when only normal data exist) [11].

However, supervised learning algorithms, such as group classification and regression analysis, are required to identify the type and severity of damage when data from both the undamaged and damaged structures are available.

Platform mapping:

- For the thesis release, it is realistic to use a transparent unsupervised layer based on thresholded novelty indices.
- The platform must keep labels and specific data for later use. If future inspections assign a damage type or severity, these labels can be then used later in supervised training.

Axiom IVa - Sensors cannot measure damage

Feature extraction through signal processing and statistical classification are necessary to convert sensor data into damage information. Sensors measure the response of a system to its operational and environmental input. To identify damage, features must be extracted from the sensor data through signal processing and statistical classification. For example, in many wave propagation approaches to SHM and impedance measurements, changes in features derived from relative information obtained from an array of sensors can be used to locate damage.

Sensors measure the system response under operational and environmental inputs, so we must infer damage using features and models.

Platform mapping:

- Keep a clear separation from Raw_Data to Cleaned_Data to OMA_BUCKET, where OMA_BUCKET contains the modal parameters..
- Treat features as main outputs. The platform should store them and let users query them, not only show them in plots.
- Keep the full processing context, like which feature was used, which window was applied, and which sensor set was selected.

Axiom IVb: if we do not use intelligent feature extraction, then the more sensitive a measurement is to damage, the more sensitive it also becomes to changing operational and environmental conditions, meaning EOVs.

In simple terms, if we do not use intelligent feature extraction, higher damage sensitivity also brings higher sensitivity to EOVs. So, measurements that react strongly to damage often also react strongly to operational and environmental variability, such as temperature, humidity,

traffic, and wind. The book emphasises that feature extraction and normalisation have to separate changes driven by damage from changes driven by EOVs.

Without intelligent feature extraction, the more sensitive a measurement is to damage, the more sensitive it is to changing operational and environmental conditions. This axiom emphasizes the need for intelligent feature extraction to separate changes in features caused by damage from those caused by environmental and operational variations. This can be accomplished through data normalization procedures such as regression analysis, or by learning both the dependence on damage and the environment.

Platform mapping:

- EOV data model: let the platform accept temperature, humidity, and rainfall channels, and store them together with vibration data. Enough script manipulation should be done in the services.
- Why logs matter: if an alarm happens during a very extreme temperature change, the operator must be able to check that.

Axiom V: Damage length and time scales dictate sensing system requirements.

Damage can start and grow over very different time scales and spatial scales. This affects the sensing choices directly, including bandwidth, sampling, sensor type and placement, and also storage and telemetry constraints.

Platform mapping:

- Store acquisition metadata, such as sampling rate, dt, and sensor ID and location, because what damage we can detect depends on these.
- Support multi-rate data, so it can combine high-rate dynamic signals with low-rate environmental or static sensors.
- Support retention and aggregation policies, because raw data can be very large. While cleaned data and features can be kept for longer.

Axiom VI: There is a trade-off between an algorithm's sensitivity to damage and its noise rejection capability.

In general, higher noise increases the detectable damage threshold. The book shows that as noise grows, the damage has to be more severe to pass a detection threshold.

Cleaning logs

cleaning_id	filename	status	duration_seconds	hp_cutoff	lp_cutoff	downsample_fact...	original_fs	cleaned_fs	error_message
2	2025_10_26_06_04_18_Fast.txt	success	23.7	0.500	15	2	100	50	
1	2025_10_26_05_04_18_Fast.txt	success	29.2	0.500	15	2	100	50	

Figure 2.1 cleaning logs showing parameters and status

Platform mapping (core):

- Make preprocessing explicit and logged (filters, detrending, downsampling, window size).
- Record run parameters in cleaning_runs and analysis_runs so the results stay reproducible and comparable.
- Add basic data quality checks, such as dropouts, clipping, and spikes, since noise has a direct effect on detectability.

Axiom VII: Detectable damage size is inversely proportional to the excitation frequency range.

In general, smaller damage needs higher frequency content to become observable. If we only look at low frequency global dynamics, we may miss local damage.

Platform mapping:

- This explains why the system should preserve enough bandwidth, or at least store raw data so we can reprocess it later.
- It also supports extensibility. If future sensing or analysis adds higher frequency features, for example, wave-based ones, then the storage and ingestion model should be able to handle them [11].

Axiom VIII: Damage increases the complexity of a structure.

Damage can introduce nonlinear behavior and make the response more complex. For this reason, features related to complexity, such as entropy like measures, can become sensitive to damage.

Platform mapping:

- This facilitates future expansion of the feature layer beyond basic OMA or FDD modal tracking, so it can also include complexity or nonlinear indicators.

- It also reinforces why it helps to store cleaned signals and derived features in a structured way, so we can add new feature extractors without redoing ingestion.

Compact mapping table

- II baseline → baseline storage/versioning → planned/partial
- IVa sensors ≠ damage → feature layer (raw/cleaned/analyzed) → implemented
- IVb EOV confounding → temperature/humidity integration + normalization hooks → planned
- VI sensitivity vs noise → logged preprocessing + QC → implemented/expandable (and so on for I, III, V, VII, VIII)

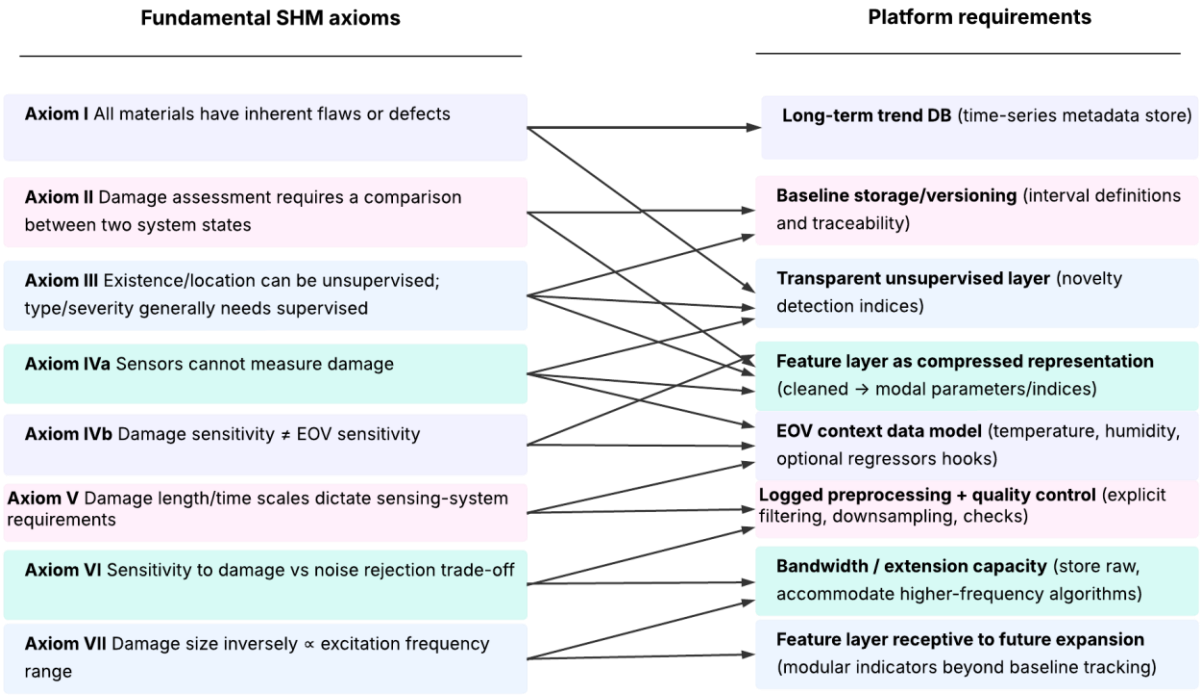


Figure 2.2 Mapping SHM axioms to platform requirements

2.2. Ambient vibration monitoring and modal analysis context (Vicoforte)

2.2.1. Case-study motivation: why Vicoforte is a “stress test” for SHM pipelines

The Sanctuary of Vicoforte (“Regina Montis Regalis”, Piedmont) is a benchmark case for long-term monitoring of cultural heritage structures because its oval masonry dome is

exceptionally large and structurally delicate: published case-study work reports internal axes on the order of ~37.23 m (major) × 24.89 m (minor), and emphasizes the complexity of the lantern-dome-drum system and its interaction with the rest of the building and the ground. [8]

Beyond geometry, Vicoforte is relevant because it has a documented history of cracking/settlement concerns and progressive conservation interventions, which makes long-term "trend interpretation" central (and non-trivial). A long-running monitoring and research program is described in the literature as starting in the 1980s (with major strengthening works in the mid-1980s) and later evolving into automated acquisition and remote analysis workflows. [12]

Vicoforte represents the kind of SHM deployment where (i) the diagnostic value comes from continuous, comparable time histories, and (ii) the data system becomes as important as the identification algorithm, because decisions depend on stable ingestion, consistent preprocessing, traceability of runs, and auditability of any "change point".

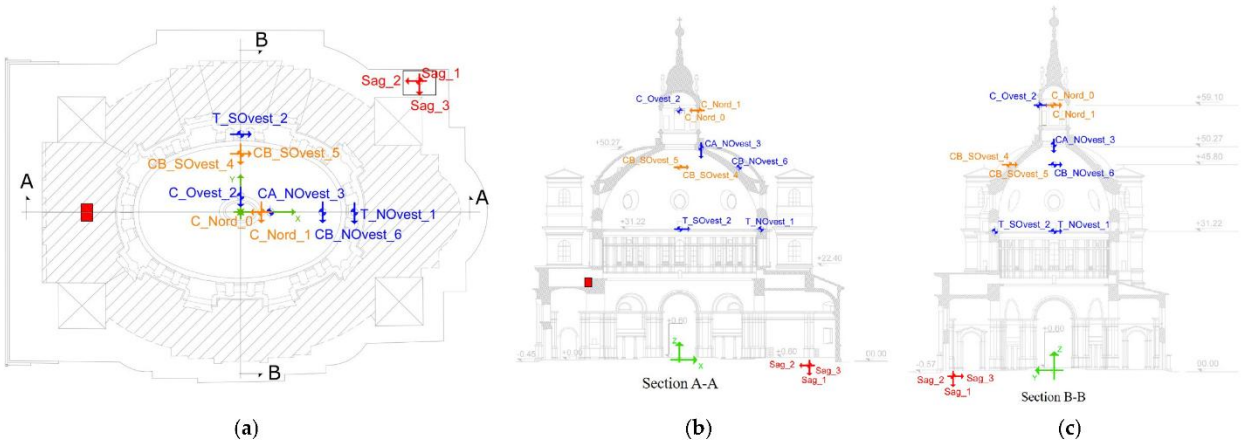


Figure 2.3 Schematic of the Sanctuary highlighting lantern–dome–drum macro-elements and key monitored zones [8]

2.2.2. Monitoring concept at Vicoforte: from long-term storage to modal tracking

- **Static monitoring (long-term, low-frequency, heterogeneous)**

Published work on the Vicoforte program describes a multi-sensor static monitoring system (crackmeters, extensometers, load cells on tie-bars, temperature/humidity and other geotechnical/structural instruments) with automated acquisition since the mid-2000s and a large number of channels (reported as ~112 instruments in one decade-scale study). [13]

This matters because static channels typically produce:

- slow time series (minutes-hours sampling),
- strong environmental coupling (temperature/moisture),

- long-term drift / missing-data episodes,
- "interpretation layers" (e.g., crack opening vs. temperature cycles) that must remain reproducible.

The platform's local first approach provides the right architecture design to manage heterogeneous static data without relying on improvised scripts.

- **Dynamic monitoring (high frequency, event or trigger based, or scheduled windows)**

For the dynamic system, open access documentation of the Vicoforte permanent monitoring describes a network of 12 accelerometers on the lantern, dome, and drum system. It includes a ground reference at the crypt and sensors at multiple elevations. It also describes a master and slave acquisition scheme to reduce distortion for long cable runs. The signals are collected locally and then transmitted online to the laboratory for automated processing [8].

Earlier open access reporting also describes a move from campaign testing, like the 2008 ambient vibration sessions, to a permanent dynamic monitoring installation around 2015. This installation was designed using optimal sensor placement, and it aims to better capture global behavior and dynamic interaction effects [14].

2.2.3. Ambient vibration and Operational Modal Analysis: why "output-only" identification is the default

For monumental buildings, forced excitation is often impractical; therefore, ambient vibration testing (AVT) and operational (output-only) modal analysis (OMA) are widely used. A classical review explains that in operational conditions the input is unmeasured and treated as stochastic; this motivates stochastic system identification and popular approaches such as stochastic subspace identification (SSI).

In the Vicoforte context, the identification problem is harder than in "textbook" cases because:

- there are many closely spaced modes and interacting subsystems, like soil and structure, and also the dome, drum, and lantern,
- the ambient signals have low amplitude, and the excitation quality changes over time,
- the response is not stationary because of environmental and operational variability.

Open access reporting on Vicoforte also states clearly that dynamic characterization is complex and needs careful processing and modeling, due to dynamic interactions, uncertainty in mass and material properties, and soil and structural coupling [14].

2.2.4. Automated modal identification and long-term mode tracking (the "real-time" challenge)

In continuous monitoring, the core deliverable is not a single modal estimate but a coherent time history of modal parameters. Recent open-access work on the Vicoforte system presents the Mode Tracking (MT) problem as: associating each newly identified mode to historical ones in order to build well-separated frequency time series, while accounting for environmental variability and identification scatter [8].

That work also shows a practical strategy based on Modal Assurance Criterion (MAC) thresholds. It also mentions that MT procedures need careful calibration for each structure and each mode, rather than using a single rule for all cases.) [8].

2.3. Time-frequency basics needed for the analysis

A key practical consequence is that anti aliasing has to happen before sampling, or before any downsampling. Once the signal is sampled, aliasing distortion is already in the data and a digital filter applied later cannot remove it.

Platform interpretation hook: when the cleaning stage downsamples the data, it needs to apply a proper low pass or band limit step before the decimation. If not, aliasing artefacts can distort the PSD estimates and can then bias OMA or FDD peak identification and mode tracking.

2.3.1. Sampling, Nyquist frequency, and aliasing

$$T_s = \Delta t, f_s = \frac{1}{T_S}, x[n] = x(nT_s), f_N = \frac{f_s}{2} \qquad \text{Eq.(2.1)}$$

Let a continuous-time signal (x(t)) be sampled uniformly with sampling interval (Δt) and sampling frequency (fs = 1/Δt). Sampling transforms (x(t)) into a discrete-time sequence (x[n] = x(nΔt)). In the frequency domain, sampling causes the spectrum of the original continuous signal to be periodically replicated at multiples of (fs). If the continuous-time signal contains energy above the Nyquist frequency (fN = fs/2), spectral replicas overlap and high-frequency components are folded back into ([0, fN]), producing aliasing [1].

2.3.2. Why FFT results have discrete bins (DFT grid, resolution, and "bin mismatch")

$$X[k] = \sum_{n=0}^{N-1} x[n] e^{-j2\pi kn/N}, f_k = \frac{k f_s}{N} \quad \text{Eq.(2.2)}$$

For a finite record of (N) samples, the discrete Fourier transform (DFT) evaluates the spectrum on a discrete frequency grid. The frequency spacing (bin width) is:

$$\Delta f = \frac{f_s}{N} \quad \text{Eq.(2.3)}$$

Hence, longer windows (larger (N)) yield finer frequency resolution. FFT outputs are organized into fixed frequency bins, and the bin spacing is controlled by sampling rate and record length.

A crucial point is that a sinusoid shows up as one sharp spectral line only when its frequency matches a DFT bin frequency exactly, so it is periodic over the DFT window. If the sinusoid frequency is between bins, the spectrum is not zero outside that bin and it spreads over several bins. This spreading of energy is one reason peaks look wider and can seem to shift.

Platform interpretation hook: in the implemented OMA/FDD stage, spectral resolution, leakage, and windowing influence how clearly modal peaks appear in the singular-value spectrum. Therefore; frequency estimates should be interpreted with an uncertainty that depends on the PSD estimator (e.g., Welch), record length, and preprocessing; MAC-based tracking reduces mode swapping when peaks are close [1] [7].

2.3.3. Spectral leakage and windowing (tapering)

When a finite segment of data is transformed, the DFT implicitly assumes the segment repeats periodically. If the signal does not complete an integer number of cycles within the segment, a discontinuity is introduced at the window edges, which injects additional frequency content and produces spectral leakage

Windowing (tapering) reduces edge discontinuities by forcing the time record smoothly toward zero at its ends. A widely used choice is the Hann/Hanning window, discussed in the reference SHM signal-processing appendix, which reduces sidelobes (leakage) at the cost of a wider main lobe (reduced ability to separate very close frequencies) [1]. The classic windowing survey by Harris formalizes these trade-offs (main-lobe width vs sidelobe levels) and remains a standard reference for selecting windows in DFT-based analysis [15].

Studies also emphasize that leakage can produce apparent peak broadening or small shifts in spectral estimates, especially when frequency content lies between bins and the window is effectively rectangular. In the context of OMA/FDD, this influences the clarity of singular-value peaks and can affect modal frequency estimates if the PSD resolution is insufficient.

2.3.4. Amplitude spectrum vs power spectrum vs power spectral density

Amplitude (magnitude) spectrum: The FFT output is complex valued, so we usually look at its magnitude (|X[k]|) to see the relative strength of the frequency components. If we apply the correct scaling, we can express the magnitude in the same amplitude units as the original signal, for example (m/s2) for acceleration. Studies point out that we can compare spectra in a relative sense directly, but recovering absolute amplitudes needs consistent scaling, often by normalising with N and converting to a one-sided spectrum for real-valued signals.

Power spectrum / PSD: For vibration signals that are stochastic, or only approximately stationary over a finite record, a single FFT magnitude can be a noisy estimate of the true spectral content. The power spectral density (PSD) describes how the signal power, or variance, is distributed over frequency. Unlike the “power spectrum”, which gives power in a bin, the PSD is normalized by bandwidth, so for acceleration it has units of ((m/s2)2/Hz). Standard signal processing texts discuss these definitions, along with how finite record transforms, scaling, and spectral estimation are connected [16].

Why PSD or Welch is often preferred for real vibration: the main idea is that Welch’s method estimates the PSD by splitting the signal record into overlapping segments. It then applies a window to each segment, computes a periodogram using the FFT, and averages the periodograms across segments to reduce variance. The original Welch paper defines this as a “time averaged modified periodogram” method and is the standard reference [17].

In the SHM signal processing appendix, the text makes the same point: when we take a finite realization of a random process, the spectral estimate includes an error term, and using averaging or PSD estimation reduces variability compared to using a single transform [1].

There is also a practical trade off. Welch averaging lowers variance, so the PSD becomes smoother, but it also lowers the effective frequency resolution because the segments are shorter. It can also introduce a window dependent bias. For this reason, Welch is useful when we need robust peak identification under noisy or non-stationary conditions, which is common in ambient vibration monitoring.

2.3.5. Time-frequency views for non-stationary data (STFT/spectrogram; optional wavelets)

Because ambient vibration is often non-stationary, a global spectrum computed over a long window may smear time-varying behavior. A common compromise is the short-time Fourier transform (STFT), which applies the FFT to sliding windows to produce a spectrogram (power vs. frequency vs. time). Standard SHM references include spectrograms and wavelet transforms as tools for detecting transients and time-localized changes.

2.3.6. Filtering and detrending (mean removal, high-pass, band-pass, Butterworth) and downsampling rationale

Field acceleration records often include low-frequency drift, a DC offset, and components that do not come from the structure. If we remove the mean and detrend, we reduce bias in spectral estimates and in time-domain information. A band pass filter is also helpful, since it keeps the analysis in the frequency range where we expect the structural modes.

- High-pass / band-pass filtering.

In reality, vibration monitoring often uses a high-pass cutoff, for example, to remove very low frequency drifts. They may also apply a band pass filter to focus on the frequency band where the structural modes are expected. The SHM reference appendix explains the main properties of common filters and discusses Butterworth filters in detail, including their smooth band pass behavior and how cutoff values are related to the sampling frequency.

- Butterworth family.

Butterworth filters are widely used because their magnitude response is monotonic, meaning there is no ripple, and because we can control the roll off through the filter order.

Butterworth filter transfer function (magnitude response or **s** domain polynomial form):

$$|H(j\omega)|^2 = \frac{1}{1 + \left(\frac{\omega}{\omega_c}\right)^{2p}} \qquad \text{Eq.(2.4)}$$

- Downsampling

Downsampling reduces storage needs and computational cost, and it can also make later feature extraction more stable, for example by focusing on lower frequency bands. However, it has to include suitable pre-filtering to avoid aliasing, consistent with the anti-aliasing requirement discussed earlier.

Downsampling reduces storage and computational cost and can stabilize subsequent feature extraction (e.g., by focusing on lower-frequency bands). However, it must be accompanied by appropriate pre-filtering to avoid aliasing, consistent with the anti-aliasing requirement discussed earlier.

2.3.7. Implications for the thesis indices (bridge to Chapter 7-8)

The modal estimation implemented later in this thesis inherits the above time-frequency properties: spectral resolution (Δf), leakage, and windowing influence how clearly peaks appear in the first singular value spectrum, and therefore influence the stability of identified modal frequencies and damping estimates.

- Outliers and slow drift can bias spectral estimates and can reduce the reliability of peak picking. For this reason, detrending and band-pass filtering are treated as mandatory preprocessing steps before OMA/FDD.
- Modal frequency estimation depends on record length (Δf), leakage, and the chosen window. Because of this, small changes in the estimated frequency should be read together with a tracking or similarity measure, like MAC, and also with a quality indicator. Using a Welch PSD can give more stable spectral density estimates under mild nonstationarity, and this helps FDD peak identification [17].
- PSD/Welch can produce more stable peak identification under non-stationarity, motivating it as a plausible future upgrade for frequency-tracking indices. [17]

The previous section established the basic signal-processing concepts that affect spectral estimates. The following section briefly surveys damage detection viewpoints, mainly to clarify what the current platform does and does not claim: it produces repeatable modal evidence and tracking metadata, while higher-level decision logic remains a configurable layer.

2.4. Damage detection approaches (current method's position)

The platform is not meant to fully automate the damage identification hierarchy. Instead, the current implementation focuses on a robust intermediate representation: operational modal parameters identified from ambient data and tracked over time. In this setup, the analysis stage provides evidence, such as frequency and damping trends and mode shape similarity, that can later be used in statistical decision rules. I chose this to keep the model complexity lower, while still keeping the pipeline compatible with more advanced detectors, for example, SSI-based identification or multivariate novelty detection, in future work [11].

2.4.1. From damage detection to making decisions: what SHM algorithms actually do

In vibration-based SHM, damage detection is rarely a direct measurement task; instead, it is a statistical decision problem that infers changes in structural condition from changes in measured response features. Even when damage-sensitive features are available (for instance, shifts in identified modal frequencies, changes in estimated damping, or reductions in mode-shape similarity), the monitoring system must still decide whether an observed deviation is consistent with normal variability (including environmental and operational variability) or whether it represents a structural anomaly requiring inspection.

A common way to organize the damage identification problem is the hierarchy of first used widely in vibration related inspection literature: detection (is damage present?), localization, assessment of severity, and prognosis [1]. In civil infrastructure, most operational SHM deployments usually start by aiming for reliable Level 1 detection, since the higher levels need denser sensing layouts, stable excitation conditions, and robust normalization for environmental effects [1]. This is why the thesis platform first focuses on robust ingestion, preprocessing, reproducible feature extraction, and conservative event logic, instead of directly targeting full modal shape based localization.

2.4.2. Feature-based monitoring as novelty detection

Most real-world SHM systems operate with abundant “normal” data and scarce (or absent) labeled damage data. For this reason, a large class of vibration-based damage detection methods are best described as novelty detection: the system learns a statistical description of the normal (baseline) condition and flags future observations that are unlikely under that normal model [1].

From a pattern-recognition viewpoint, novelty detection can be implemented through several different families of detectors:

1. Distance or outlier based detectors (parametric): if we approximate the baseline feature distribution as Gaussian, we can measure novelty as a scaled distance from the mean, such as a univariate z score, or as a multivariate distance like the Mahalanobis distance. We can then set thresholds from Gaussian confidence regions, with the usual caveat that the Gaussian assumption often breaks under environmental and operational variability [1].
2. Density estimation (nonparametric): KDE novelty scores. A kernel density estimate can approximate the baseline probability density p(x), and we can define novelty as a decreasing function of the likelihood, for example as -log p(x). KDE is attractive because it avoids strict parametric assumptions. At the same time, it becomes

sensitive when the training data are sparse and when the feature dimension grows, so it is better suited to low dimensional feature vectors or to cases where we apply dimensionality reduction first [1].

3. One-class classification (discriminative): one-class SVM (support vector data description) estimates the support of the baseline distribution and classifies points outside the learned boundary as novel. This approach is particularly useful when one wants a flexible decision boundary without explicitly estimating (p(x) [1].
4. Reconstruction based novelty detection. Auto associative neural networks, and more recently autoencoders, learn to reconstruct baseline features or signals. In this setting, a large reconstruction error becomes the novelty score. These methods work well when the baseline manifold is learnable and stable, but they need careful control of overfitting and drift [1].

Two points matter for framing the thesis:

- Unsupervised vs supervised: novelty detection is usually unsupervised, or semi supervised, because it trains only on baseline data. Supervised classifiers, instead, need labelled examples for several classes, like damage types or locations, and these are rarely available in monumental buildings. Still, if a labelled dataset exists, for example from lab tests or simulated data, we can use supervised models for detection. In this context, one class methods are in between the two approaches [1] [11].
- Why the suggested platform starts with low-dimensional features: high-dimensional density estimation and complex classifiers can get fragile when the baseline dataset is sparse. So, the current implementation stays with a small set of interpretable modal outputs, like frequencies, damping proxies, and MAC based similarity. This gives a practical first version that we can test in daily operation, and later it can be extended to richer multivariate detectors.

2.4.3. Thresholding and control plots for online detection

No matter how we define the novelty score, the monitoring system still has to turn it into an alarm decision. Classical statistical process control (SPC) provides a well-established way to do online thresholding on time series and features using control charts, such as Shewhart, CUSUM, and EWMA, while balancing sensitivity and false alarms [1]. In vibration based SHM, control charts are especially useful because they make three parts explicit: (i) baseline estimation, (ii) control limits, and (iii) run rules for sustained deviations..

For structural monitoring, we have to adapt this idea because vibration measurements, and many indices derived from them, are usually autocorrelated. This breaks the independence

assumptions that simple control limits rely on. A common fix is to build the charts using residuals from time series models, for example Autoregressive (AR) models, since this can reduce autocorrelation and make the control limits more reliable [1].

2.4.4. Extreme value statistics for robust alarm limits

When baseline feature distributions are non-Gaussian (as is common under environmental/operational variability), choosing thresholds based on Gaussian confidence intervals can be misleading. Extreme value statistics (EVS/EVT) offers a principled approach to setting alarm levels by modelling the distribution of extremes (e.g., maxima/minima of windowed features) rather than the full distribution [1].

In SHM, this is usually implemented by:

- Defining a single decision metric, such as the windowed maximum of an index or a residual,
- fitting an extreme value distribution from the GEV family or using peaks-over-threshold variants,
- selecting thresholds based on a target false alarm probability, like the 99.5th percentile for upper tail alarms.

The primary benefit is improved control over tail probabilities, which directly correspond to false alarm rates in online monitoring. This is especially important during extended platform operation, where maintaining consistent alarm behavior amidst large data volumes is crucial.

2.4.5. Modal feature based approaches (future extension aligned with Vicoforte)

While feature-based novelty detection provides a strong baseline, many civil SHM studies (including long-term monitoring of monumental buildings) aim to track modal parameters (natural frequencies, damping ratios, and mode shapes) via operational modal analysis (OMA). In these workflows, damage detection is often phrased as identifying persistent deviations in tracked modes, with additional steps to account for environmental correlation (notably temperature) [1].

A common approach for automated OMA under output only conditions is stochastic subspace identification (SSI). In SSI, stabilization diagrams are used to separate physical modes from spurious poles when we change the model order [1]. For mode tracking, similarity measures like the Modal Assurance Criterion (MAC) measure the correlation between mode shapes and help match modes over time [1]. For mode tracking, similarity measures like the Modal Assurance Criterion (MAC) measure the correlation between mode shapes and help match modes over time [1].

This direction fits the Vicoforte monitoring concept, which uses continuous ambient vibration, automated identification, stabilization diagram postprocessing, and early consideration of temperature effects. It can also be presented as a roadmap step beyond the thesis release. Once the platform can reliably produce cleaned, synchronized windows and store data over the long term, we can extend the same pipeline to compute SSI-based modal features and store them as time series together with the environmental sensor data [4].

2.4.6. Positioning the current thesis method in this landscape

Given the project constraints (local-first deployment, heterogeneous files), the current platform's method is positioned as:

- Feature based novelty detection for Level 1 damage detection. It uses low dimensional and interpretable modal features, like frequency evolution, MAC drops, and quality score changes. These are computed from cleaned windows and compared to a baseline or a historical distribution.
- Alarm logic inspired by SPC run rules. It uses consecutive window confirmation and cooldown based suppression to limit fake events and avoid alarm flooding.

Roadmap compatibility with more advanced detectors: KDE/one-class methods for richer feature vectors; EVT-based thresholding for improved false-alarm control; and SSI/MAC-based modal tracking for Vicoforte-style monitoring and environmental correlation studies. [18]

3. Requirements, scope, and project constraints

This chapter translates the SHM motivation and the chosen architecture into implementable requirements. The emphasis is on operational constraints: frequent file arrivals, mixed sensor sets, the need for provenance, and a local-first deployment that can be run by a small team without a dedicated DevOps stack. Requirements are therefore written as service contracts and data contracts, not only as user interface features.

3.1. Use cases

This thesis focuses on the operational side of SHM. The goal is to turn repeated monitoring exports into consistent, queryable indicators and events, with as little manual handling as possible. SHM system design discussions also stress that long term monitoring produces large and heterogeneous datasets, so we need a data management layer that supports archiving, retrieval, and access, not only signal processing algorithms.

Actors:

- Operator / Monitoring engineer: needs near real-time visibility, including pipeline status, dashboards, and alarms or events.
- Analyst / Researcher: needs reproducible preprocessing and feature extraction, the ability to rerun analyses with different parameters, and data export for further studies.
- System administrator (lightweight): needs a low ops setup, resilience, and simple recovery if something breaks.
- Developer: needs an architecture that is easy to extend, for example, by adding new indices, new structure, new sensors, environmental channels, or modal tracking.

3.1.1. “Drop StructuralX file → get cleaned signals + indices + events + dashboards”

Goal: A monitoring export is placed in a hot folder; the platform ingests it, preprocesses it, computes indices, generates events, and makes results visible.

Rationale: Long-term SHM needs consistent handling of high-rate data and effective access. File based practices often make querying and later use difficult, so this motivates a system that supports structured storage and access services.

Success criteria (operator view):

1. The system detects a file and performs ingestion only once (idempotent).
2. Users can query raw measurements by structure, sensor, and time.
3. The platform logs cleaning and analysis runs, and keeps them traceable.
4. Dashboards show new time windows and indices, and events show up in an event list.

3.1.2. Manage monitoring metadata (structures/sensors)

Goal: Define and maintain the metadata needed to interpret time series, including the structure identity, sensor IDs and locations, channel names, sampling rate expectations, and sensor activation or deactivation.

Rationale: SHM data has to stay interpretable months or years later. For this reason, metadata is essential for retrieval and auditing, and it is repeatedly highlighted as part of effective long-term monitoring and information utilization [19].

Typical operations: create or delete a structure, add sensors, map channels, enable or disable sensors, and annotate maintenance periods.

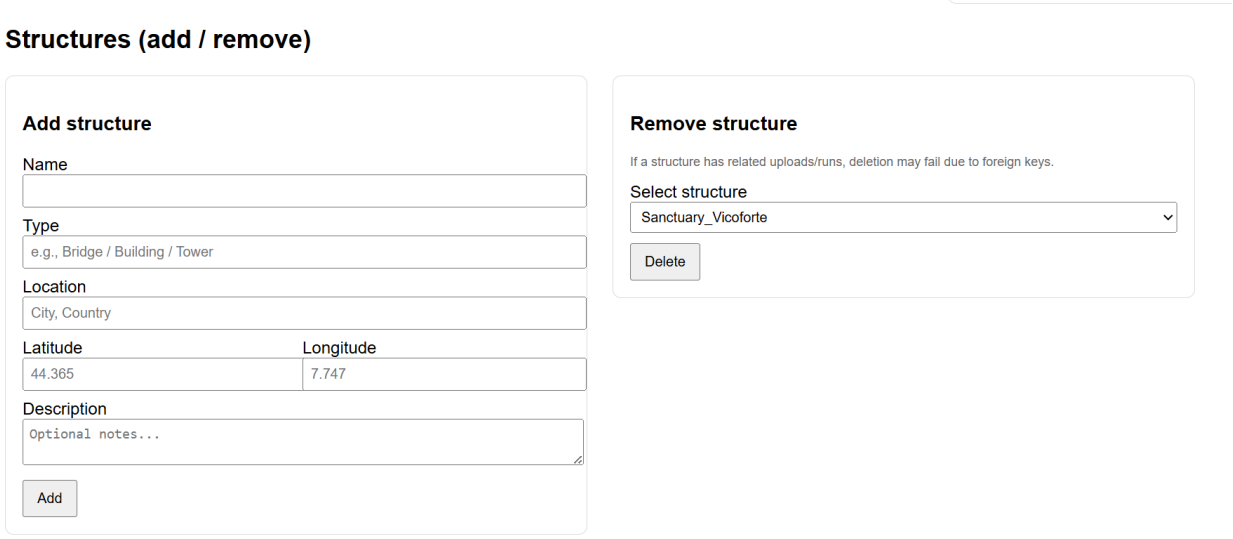


Figure 3.1 UI: Create Structure / Create Sensor.

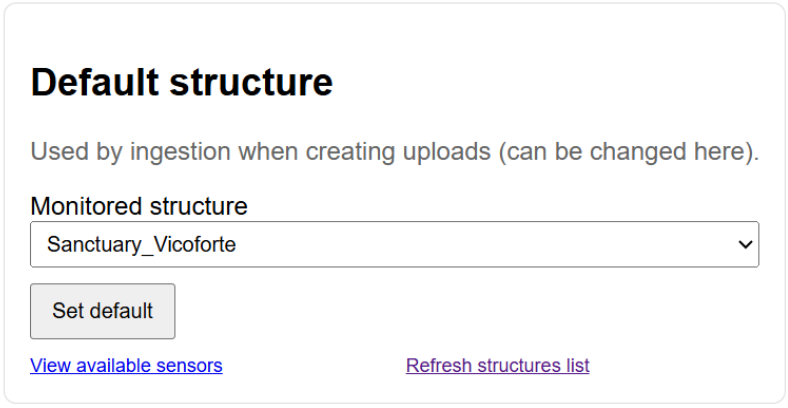


Figure 3.2 UI: Set Default Structure / List Sensors

3.1.3. Traceability & audit: "Why did the system raise an alert?"

Goal: Starting from an event, an engineer should be able to trace it back to the raw file, the exact time interval, the sensor set, the preprocessing parameters, the feature computation parameters, and the decision rule.

Rationale: SHM decision making has to be accountable. The SHM workflow also separates acquisition, cleansing and normalization, feature extraction, and decision making, and it stresses consistent archiving and transmission of condition information.

Outputs required: run IDs, timestamps, parameter snapshots, logs, and a stable link between events and the analysis runs.

3.1.4. Visualization for operations (dashboards + operator UI)

Goal: Provide a single pane view of pipeline health, meaning ingestion, cleaning, and analysis status, and also SHM outputs such as raw and cleaned plots, indices, and an events timeline.

Rationale: Real time visualization is often where SHM systems run into problems, mainly because of integration gaps and reliance on offline or manual processing. Web and database backed access can improve operational usability.

3.2. Functional requirements

The functional requirements come directly from the use cases above and from the operational needs of long-term SHM. This includes handling high-rate data flows, enabling structured storage and access, and keeping traceability and visibility.

Below, each requirement is written in an implementation ready form, meaning what the system must do, and it can later be mapped to specific services in Chapter 4.

3.2.1. File ingestion

- The system shall monitor a configurable directory for new StructuralX exports and ingest each file exactly once (idempotency).
- The ingestion pipeline shall parse the acquisition metadata needed for correct time indexing, such as t0, dt, sampling rate, and channel definitions.
- The ingestion pipeline shall record an upload log entry, including timestamp, source path, structure mapping, parse success or failure, and a checksum or an equivalent identifier.

3.2.2. Raw time-series storage

- The system shall store raw monitoring streams in a time indexed data store that supports queries by time range and tag filters, such as structure and sensor or channel.
- The storage format shall preserve enough metadata to allow later retrieval and auditing.

Data intensive SHM needs long-term archiving and effective data access. File based approaches often make data hard to use, which motivates database supported services.

3.2.3. Preprocessing and cleaning as a reproducible pipeline stage

- The system should apply a defined preprocessing pipeline to raw signals, for example mean removal, detrending, bandpass filtering, and optional downsampling, and write cleaned signals to a dedicated storage namespace.
- The system should log each run, including parameters, timestamps, input range, number of samples processed, and success or failure.

Rationale: preprocessing choices have a real impact on SHM features, so they must stay repeatable and auditable to support trustworthy long-term comparisons.

3.2.4. Feature and index computation (analysis stage)

- The system should compute operational modal parameters from cleaned signals, such as natural frequencies, damping ratios, and mode shape similarity metrics like MAC. It should store these outputs as time series and link each one to a specific analysis run for provenance. The Modal Assurance Criterion (MAC) is used for mode tracking.

$$\mathrm{MAC}(\phi_i, \phi_j) = \frac{|\phi_i^H \phi_j|^2}{(\phi_i^H \phi_i)(\phi_j^H \phi_j)} \qquad \text{Eq.(3.1)}$$

- The system shall support adding new indices without modifying ingestion logic (extensible analysis layer).

3.2.5. Event generation (decision logic)

- The system should generate discrete “events” based on index excursions (thresholding or novelty logic) with operator-friendly safeguards (e.g., consecutive-window confirmation, cooldown suppression).
- Each event shall store: start/end time, severity/score (if defined), the triggering index, and references to the producing analysis run.

3.2.6. Metadata CRUD API (structures/sensors) and status endpoints

- The system shall provide an API to create/read/update/delete (or partially manage) structures and sensors.
- The system shall expose endpoints that let users retrieve the pipeline status for each upload, including ingested, cleaned, and analyzed flags, and the last error.

3.2.7. Visualization layer

- The system shall provide dashboards to visualize raw vs cleaned signals, indices time histories, and events.
- The system shall provide pipeline observability panels, showing ingestion counts, the last successful run, and failures, to support operational monitoring.

Rationale: real time SHM visualization works better with a unified digital framework and web or database backed querying, instead of offline or manual [20].

3.2.8. Logging and provenance (end-to-end traceability)

- The system shall produce structured logs for each service and store run metadata so results can be reproduced and audited.
- The system shall link artifacts across stages, from upload to cleaning run to analysis run to events.

Rationale: SHM deployments explicitly list consistent, retrievable archiving and the transmission of condition information as key challenges. Provenance is required if we want to trust the alerts.

3.3. Non-functional requirements

Beyond the functional pipeline, from ingest to clean to analyze to events, an SHM platform has to meet deployment grade qualities, including long term operation, traceability, and reproducibility. The SHM process also treats data acquisition, cleansing, normalization, compression, and fusion as core steps for making features and decisions meaningful. It also states that operational SHM produces large amounts of data and needs robust data reduction and handling for lifecycle comparisons.

3.3.1. Local-first operation and low operational burden (low-ops)

Requirement: The system shall run on a single workstation or server with minimal external dependencies and minimal administration effort, meaning it should be easy to bring up and run.

Motivation: Permanent monitoring systems prioritize reliability and low maintenance cost. Monitoring requirements also commonly emphasize reliability, maintenance planning, and system flexibility across the lifecycle.

Design implications:

- A self contained stack, including databases, services, and visualization, that can run locally.

- Automatic recovery from common faults, such as service restart and idempotent ingestion for database.
- Clear operational documentation and sensible defaults.

3.3.2. Reproducibility of preprocessing and analysis runs

Requirement: Given the same raw input and configuration, the platform shall reproduce the same cleaned streams, indices, and events (within numerical tolerance) and record the parameters that produced them.

Motivation: The SHM workflow emphasizes that data cleansing and normalization are not cosmetic: they shape the meaning of extracted features and must be treated as controlled steps.
Design implications.

- Persisted run registries (uploads, cleaning runs, analysis runs) with parameter snapshots.
- Versioning hooks (service version/git commit hash in run metadata).
- Deterministic windowing and consistent time indexing.

3.3.3. Traceability (provenance) from event → raw data

Requirement: For any generated event, the platform shall allow an engineer to trace it back to the raw file, the exact time interval, the sensor set, and the processing parameters. Motivation: SHM references explicitly include "archive data consistently and retrievably" and "transmit condition information" among core challenges from a sensing/data-acquisition perspective; both imply provenance and auditability.

Design implications.

- Foreign key links, from event to analysis_run to cleaning_run to upload.
- Unique identifiers for ingested files, using a hash or a unique constraint.
- Stored time ranges for each run window.

3.3.4. Reliability and robustness in long-term operation

Requirement: The platform shall handle continuous arrivals of files and transient failures without corrupting stored data or duplicating processing.

Motivation: For permanent systems, reliability and minimal maintenance are repeatedly treated as practical requirements; the monitoring system should also be able to self-report status to the engineer.

Design implications.

- Idempotent ingestion (detect duplicates; “ingest exactly once”).
- Safe retry behavior (retry without double-writing).
- Health checks and clear failure modes (errors recorded in logs + run tables).

3.3.5. Performance and throughput (practical latency)

Requirement: The platform shall process typical StructuralX files with acceptable end-to-end latency for operational dashboards (from file arrival to indices/events becoming visible), on commodity hardware.

Motivation: Data-intensive SHM is explicitly framed as a “big data” challenge where long-term deployments incur large volumes of heterogeneous data and require a data-management framework for archiving and access [19].

Design implications.

- Use batch or window processing, for example 10 s windows, instead of per sample logic.
- Apply data compression or feature reduction in line with SHM practice, by storing reduced indices as a “diagnostic history”.
- Provide configurable downsampling and retention policies.

3.3.6. Scalability within the “single node” scope

Requirement: The design shall remain functional as data volume grows (more files, longer monitoring periods) within a single-node deployment.

Motivation: SHM literatures mention that operational use produces large datasets, so it benefits from data condensation and strong handling for long period comparisons.

Design implications:

- Use a low cardinality time series schema, so we avoid tag explosion and store only essential tags.
- Define a retention strategy that separates raw, cleaned, and index data.
- State a clear scale out roadmap, such as a stream bus or distributed workers, as future work.

3.3.7. Observability (logs, metrics, dashboards)

Requirement: The system should have enough operational signals to tell (i) true structural anomalies apart from (ii) pipeline failures or data quality issues.

Motivation: SHM needs to separate environmental and operational effects from damage, and it often relies on extra measurements. In the same way, the platform has to separate "data issues" from "structure issues."

Design implications:

- Use structured logs for each service, including ingestion, cleaning, analyzer, and APIs.
- Provide dashboard panels for pipeline status, such as last success, failures, and backlog.
- Add explicit QC notes, for example, channel anomaly flags and missing segments.

3.3.8. Portability and repeatable deployment (containerization)

Requirement: The platform shall be deployable in a repeatable way across machines, from a developer laptop to a lab workstation to a demo machine.

Motivation: Containerization is widely used to package an application together with its dependencies. This helps avoid dependency hell and improves portability and consistency across environments [21].

Design implications:

- Use a Docker Compose stack for InfluxDB, PostgreSQL, Grafana, services, and the UI.
- Use persistent volumes for database state.
- Use environment based configuration, with a .env template.

3.3.9. Basic security and data protection (thesis level)

Requirement: For a local first deployment, the system shall include basic controls to prevent accidental data loss or exposure, for example non default passwords and limited open ports.

Scope note: Full authentication and authorization is outside the scope of the thesis release as stated in section 3.4. However, containerized deployments still need basic hardening.

3.4. Scope boundaries and project constraints

3.4.1. Thesis release scope (implemented deliverables)

The thesis-release target is a complete operational loop on a single machine:

1. Automated ingestion of StructuralX exports into a time-indexed store (raw bucket).
2. Repeatable preprocessing (cleaning) with run logging and cleaned bucket storage.
3. Analysis/indices computation and writing of reduced time series (analyzed bucket).
4. Event generation with transparent, conservative alarm logic (debounce/cooldown) and event persistence.
5. Grafana dashboards for raw/cleaned/indices + pipeline status (to be added before submission, per your plan).
6. Dockerized deployment (Compose) to make the platform reproducible and demo-ready. (seltzer.com)

3.4.2. Deployment constraints

- Single-node, local-first deployment; reliability is achieved via restartability and idempotency, not via redundant clusters.

4. System architecture and workflow

4.1. High-level architecture

4.1.1. Architectural style: microservices with storage-mediated coupling

The platform is organized as a set of small services (FastAPI APIs and background workers) that communicate through explicit persistence and explicit events. Time-series signals and derived modal outputs are persisted in InfluxDB, while metadata and run provenance are persisted in PostgreSQL. Stage-to-stage coordination is handled through Redis Streams: ingestion publishes an ingested_ready event, cleaning publishes a cleaned_ready event, and analysis publishes an analysis_done event. This combination keeps services loosely coupled while still providing a clear notion of pipeline progress.

- InfluxDB (time series backbone): it holds raw signals, cleaned signals, and OMA and FDD derived modal time series in separate buckets.
- PostgreSQL (relational backbone): it holds metadata for structures and sensors, plus the run and event registries, including uploads, cleaning runs, analysis runs, and events.

I chose this design for resilience. Each service can start or restart on its own, and the system state stays easy to inspect because the databases and the event streams show the current progress. The event layer avoids tight calls between services, and the storage layer keeps durable ground truth for auditing and reprocessing.

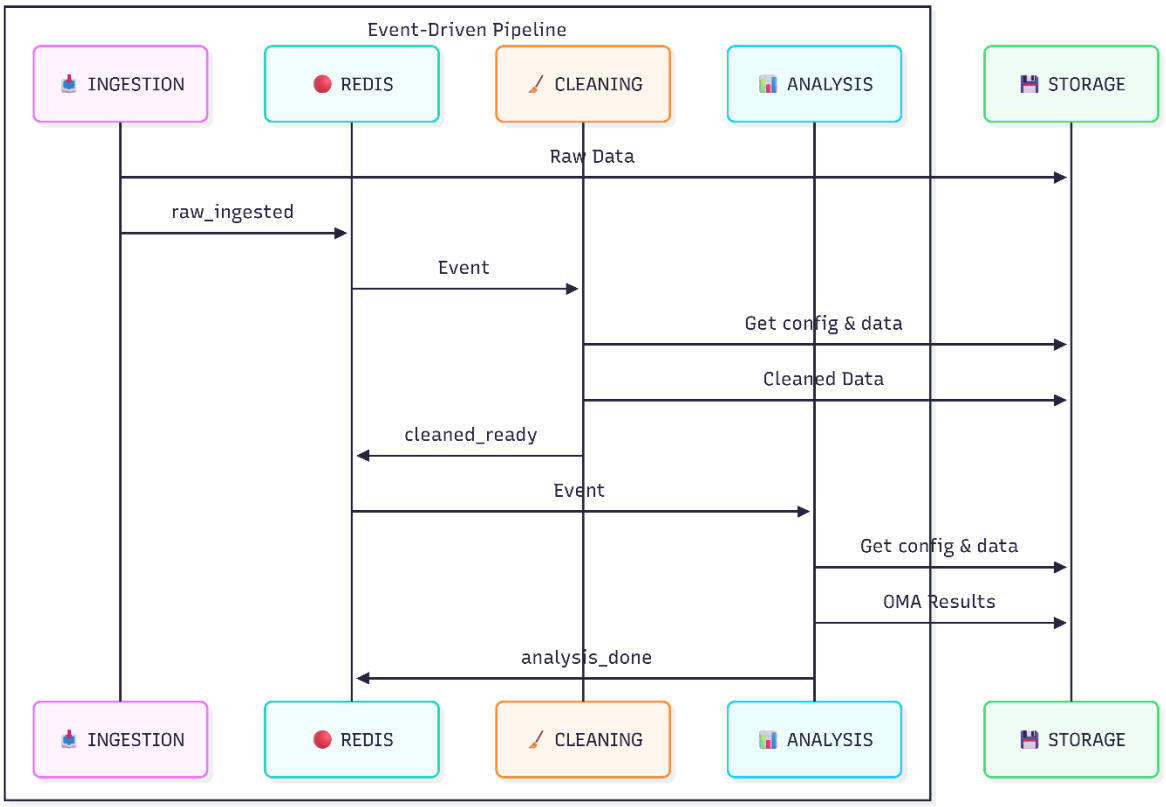


Figure 4.1 Sequence of inter-service events (raw_ingested, cleaned_ready, analysis_done) and data movement across pipeline stages

4.1.2. Main runtime components

- Ingestion Service (FastAPI + watchdog)
  - Watches a Windows "hot folder" for finalized monitoring exports.
  - Parses each file and writes raw accelerometer time series to InfluxDB (Raw_Data).
  - Logs the upload outcome to PostgreSQL (uploads_log).
  - Publishes an ingested_ready event to the Redis stream when ingestion succeeds, enabling the cleaning worker to start asynchronously.
- Cleaning stage (repeatable preprocessing, implemented as a worker consuming Redis events)
  - Reads the raw window from InfluxDB (Raw_Data).
  - Applies mean removal, detrending, Butterworth bandpass, and downsampling.
  - Writes cleaned time series to InfluxDB (Cleaned_Data).
  - Records parameters + status in PostgreSQL (cleaning_runs) for reproducibility.
- Analyzer Worker

    - Reads cleaned signals from InfluxDB (Cleaned_Data).
    - Stores modal results in InfluxDB (OMA_BUCKET), using controlled cardinality tags like Structure and ModeID, and records run provenance in PostgreSQL (analysis_runs).
    - Generates OMA and FDD modal observations and writes them to InfluxDB (OMA_BUCKET) and PostgreSQL (mode_observations). It also logs each analysis execution in analysis_runs.

- Metadata API (FastAPI)
    - Provides CRUD endpoints for structures and sensors.
    - Provides query endpoints for uploads, runs, and events so operators, and UI, can inspect the pipeline state.
- Grafana (observability and monitoring dashboards)
    - Planned visualization layer that queries:
        - InfluxDB for raw, cleaned, and index time series
        - PostgreSQL, either directly or through the API, for upload logs, run status, and event timelines
- Minimal UI (for future improvements use React/Vue)
    - A thin operator console that uses the Metadata API to:
        - browse structures and sensors
        - view upload pipeline status
        - list detected events
    - This is not meant to replace Grafana. The UI provides management and traceability views that can be used alongside the dashboards.

4.1.3. Data stores and “integration contracts”

- InfluxDB buckets
    - Raw_Data: raw accelerometer series (measurement accel, field value)
    - Cleaned_Data: pre-processed/ downsampled series (same measurement/ field, but post-cleaning)
    - OMA_BUCKET: compact modal outputs (measurement oma_modes; tags include ModeID).
- PostgreSQL schema
    - structures, sensors (metadata)
    - uploads_log (ingestion provenance + file metadata + outcome)
    - cleaning_runs (parameterized preprocessing provenance)
    - analysis_runs (parameterized analysis provenance)

- mode_observations (modal outputs)

This separation is the core reason the platform is “local-first but still operational”: compute services can be restarted without losing provenance, and it can audit any event back to the exact file and processing parameters.

4.2. End-to-end workflow

Step 0 - File production and drop (StructuralX export)

StructuralX produces hourly *_Fast.txt exports. The platform assumes a consistent format with:

- header fields providing timing (t0, dt)
- a time column used for timestamps
- multiple accelerometer channels (dynamically detected)
- possible trailing zero “tail” that must be trimmed

```
t0  2025/10/26 06:04:07
dt  0.010000
tempo    c_Nord_0    c_Nord_1    c_Ovest_2   CA_NOvest_3 CB_SOvest_4

0.000    0.004060    0.008676    -0.002767   -0.000109   -0.007181
0.010    0.003997    0.009240    -0.002968   -0.000198   -0.005905
0.020    0.005185    0.009943    -0.002866   -0.000170   -0.006682
0.030    0.005678    0.009809    -0.002220   -0.000184   -0.006783
0.040    0.004948    0.009578    -0.003031   0.000096    -0.006577
0.050    0.004746    0.008764    -0.003786   -0.000565   -0.006698
0.060    0.006171    0.009163    -0.002513   0.000189    -0.006998
0.070    0.005457    0.009415    -0.003474   0.000141    -0.006346
0.080    0.006162    0.009178    -0.001897   0.000340    -0.006784
0.090    0.005406    0.009653    -0.002697   0.000481    -0.006536
```

Figure 4.2 Example input file structure

Step 1 - Watcher detects a finalized file (hot-folder gate)

A watchdog watcher monitors the configured HOT_FOLDER and applies a finalization gate before ingestion:

- only ingest files matching the final suffix (e.g., *_Fast.txt)

- only ingest if file is stable (size not changing for N seconds)
- only ingest if file size is above a minimum threshold (MB)

This protects the pipeline from partial writes and duplicate ingestion attempts typical of real-time export workflows.

Step 2 - Ingestion service parses and writes raw time series

Once the watcher calls POST /ingest:

1. Parse file → numeric matrix
    - read time axis and channels
    - trim trailing all-zero tail
    - validate "non-empty after trimming."
2. Compute acquisition time bounds
    - start_time = t0
    - end_time = start_time + N/fs
    - These bounds become the integration key used by later stages (cleaner/analyzer windowing).
3. Write to InfluxDB (Raw_Data)
    - measurement: accel
    - tags: Structure, Sensor
    - field: value
    - timestamps: ns precision
4. Write upload registry row in PostgreSQL (uploads_log)
    - filename, path, size, time bounds, points written, status
    - idempotency: duplicate detection via uniqueness constraints (duplicate file returns "duplicate" without re-writing)

Step 3 - Cleaning stage transforms raw → cleaned (repeatable preprocessing)

After successful ingestion, the platform initiates the cleaning stage. Currently, this occurs immediately after ingestion; if it is later separated into a different worker or container, the logical workflow will remain the same.

For each channel in the upload time range:

1. Query raw sensor time series from Raw_Data.
2. Apply the cleaning pipeline:
    - mean removal
    - polynomial detrending (linear by default)
    - Butterworth bandpass (e.g., 0.5-15 Hz, zero-phase)

    - downsampling (e.g., 100 Hz → 50 Hz)
3. Write cleaned series to Cleaned_Data with the same tags.
4. Record a cleaning_runs entry with:
    - parameter snapshot (hp/lp, downsample factor, filter order, detrend order)
    - original/cleaned sampling frequency
    - run status and points written

This makes preprocessing an explicit, replayable stage rather than an "invisible script".

Step 4 - Analysis computes indices and writes compact results

The current implementation focuses on producing a consistent time history of modal parameters. I treat the step of turning these observations into debounced alarms as a future layer, since it needs structure specific thresholds, handling of environmental and operational variability (EOV), and careful validation. A later extension can map changes in frequency and or MAC to alert states, using debounce and cooldown logic that is common in SHM decision pipelines.

Step 5 - Alerting and event generation (future extension)

Each execution is recorded in PostgreSQL under analysis_runs so we keep track of when and how the modal estimation was done. The identified modes and the per run observations are also stored in relational tables, modes and mode_observations. This makes traceability and UI queries easier, for example "show the last 30 days of Mode 3 frequency".

- fields: frequency_hz, mac, quality
- tags: Structure=..., ModeID=...
- measurement: oma_modes

The system writes results to InfluxDB in a dedicated bucket called OMA_BUCKET.

For each upload, meaning each time range, the analysis stage estimates a set of modes and stores at least the natural frequency and some quality descriptors, such as modal coherence or a quality score. To keep continuity across runs, the pipeline assigns a stable ModeID and evaluates similarity using the Modal Assurance Criterion (MAC). This supports mode tracking across consecutive analyses [7].

The analyzer reads from Cleaned_Data, not Raw_Data, and runs an automated Operational Modal Analysis (OMA) step using Frequency Domain Decomposition (FDD). The goal is to extract interpretable, time invariant modal parameters, rather than compute energy based or FFT peak based scalar indices.

Step 6 - Visualization and operator access (Grafana + optional UI)

Finally, operators consume outputs via:

- Grafana dashboards
    - Raw signals (Influx Raw_Data)
    - Cleaned signals (Influx Cleaned_Data)
    - Modal outputs (Influx OMA_BUCKET)
    - Pipeline status (Postgres tables or API)
    - Events timeline (Postgres)
- Minimal UI (improved UI by React or Vue planned for future)
    - “Structures/Sensors” metadata page (CRUD or partial CRUD)
    - “Uploads & Runs” page (traceability view)
    - “Events” page (filterable list)

4.3. Component responsibilities

This section summarizes “who does what” in the platform. Components are intentionally small and communicate primarily through the two databases (InfluxDB for time series, PostgreSQL for metadata + provenance), so each piece can be restarted independently without losing system state.

4.3.1. Metadata API (FastAPI)

- It owns the domain metadata, meaning structures, sensors, and the identifiers used as tags in InfluxDB.
- It exposes read endpoints for the operational state, including uploads, cleaning runs, analysis runs, and events.
- It acts as the integration point for a future UI, and also for external tools or scripts.

4.3.2. Ingestion service (watcher plus ingest endpoint)

- It watches the hot folder and applies a file stability gate to avoid reading partial files.
- It parses StructuralX exports and writes raw time series into Raw_Data.
- It creates or updates uploads_log with status and key metadata, including time bounds and the file id or path.

4.3.3. Cleaner worker (for preprocessing)

- It reads raw windows from Raw_Data and applies the configured cleaning pipeline, including detrending, filtering, and downsampling.
- It writes cleaned windows to Cleaned_Data.

- It writes a cleaning_runs record with the parameters and the outcome.

4.3.4. Analyzer worker (OMA and FDD plus mode tracking)

- It reads aligned windows from Cleaned_Data and runs automated OMA and FDD to estimate modal frequencies, damping ratios with a prototype estimate, and mode shapes. It then tracks modes over time using MAC and frequency tolerances.
- It writes modal outputs to OMA_BUCKET as oma_modes.
- It writes an analysis_runs record and stores per mode observations, including frequency_hz, mac, and quality, to support traceability.

4.3.5. Grafana (dashboards)

- It queries InfluxDB for raw, cleaned, and index time series.
- It queries PostgreSQL, or the API, for uploads, runs, and events to build pipeline health panels and events panels.
- It provides the main operator facing visualization for the thesis release.

4.3.6. Minimal UI

- A lightweight operator console that uses the Metadata API:
    - Structure and sensor management, with basic CRUD or read only access
    - Upload and run status list
    - Event list with filtering capability (in development)
- It complements Grafana by focusing on management and traceability views, not signal plots.

4.4. Traceability design

A key design option is that every output, whether it is an index or it is an event, remains auditable. An engineer should be able to answer which data produced a given result, and which parameters were used to get it. I implement this using three provenance registries in PostgreSQL.

4.4.1. uploads_log: Ingestion provenance, meaning what entered the system

Why it exists: file based SHM pipelines can fail without clear signs if there is no registry that records what the system processed.

It guarantees:

- Each file is either processed once, or it is explicitly marked as duplicate or failed.
- Every raw time series block can be traced back to the file identity, the arrival time, the parsed time bounds, and the ingestion outcome.

4.4.2. cleaning_runs: Preprocessing provenance, meaning how it was cleaned

Why it exists: preprocessing choices directly affect spectral content and modal estimates, so they have to be reproducible.

Guarantees:

- Cleaned data in Cleaned_Data can be traced back to the exact preprocessing parameters, including filter cutoffs and order, detrend settings, and downsampling.
- Failures are recorded explicitly through status and error fields, which supports safe retries and debugging.

4.4.3. analysis_runs: Feature provenance, meaning how indices were computed

Why it exists: indices and events need to link to a specific run configuration, including the baseline definition, window size, and thresholds.

Guarantees:

• Every observation set in OMA_BUCKET can be linked to an analysis run with recorded parameters in analysis_runs.

• Events reference analysis runs, so alarms remain explainable and reproducible.

4.4.4. End-to-end lineage (the audit chain)

In combination, these tables form a simple lineage chain:

Figure 4.3 End-to-end lineage chain

This design provides:

- Reproducibility: rerun with the same parameters and confirm results.

- Debuggability: isolate whether anomalies come from data issues, preprocessing, or genuine changes.
- Operational accountability: every alarm has a documented provenance.

5. Data model and storage design

This chapter describes how the platform represents monitoring exports as (i) time-indexed signals and indices and (ii) relational metadata, and provenance logs. The storage layer is split into two complementary databases:

- InfluxDB stores high-frequency time series (raw and cleaned accelerations) and lower-rate time series (OMA/FDD-derived modal outputs such as per-mode frequency and quality).
- PostgreSQL stores long lived metadata, including structures and sensors, and it also stores the pipeline audit trail, meaning uploads, run logs, and events.

This separation facilitates scalable queries on signals, while keeping system state and traceability clear and easy to inspect.

5.1. Input format (StructuralX export)

5.1.1. File structure and time reconstruction

The platform ingests monitoring exports from StructuralX in text form. Each export is assumed to include:

- a header with timing information, including:
  - t0, the acquisition start time as an absolute timestamp
  - dt, the sampling interval in seconds
- a time column that is used either as:
  - an explicit time axis, which is preferred for validation
  - a consistency check against the reconstructed sample index n.dt
- multiple acceleration channels, where each one represents a sensor channel

Given t0, dt, and the number of samples N, the platform reconstructs sample timestamps as:

t[n] = t0 + n.dt, n = 0, ..., N-1

This way, the time series written to InfluxDB are fully time-indexed, and we can query them by time range in a consistent way across the pipeline stages, or using the databases.

5.1.2. Trimming trailing tails (all zero segments)

In real exports, a file may end with a trailing segment where the channels become zeros, for example because the acquisition shuts down or the export adds padding. During ingestion, the platform trims this trailing all zero tail so it does not affect later preprocessing and spectral features.

Assumption: the system trims only the trailing all zero segment. It keeps internal zero segments, since these can be real sensor dropouts and should be handled through quality checks and logs.

5.1.3. Error handling assumptions

The ingestion step treats the following as hard failures (logged in PostgreSQL and not written as valid raw series):

- malformed file structure or missing required timing fields
- non-numeric values where numeric samples are expected
- file becomes empty after tail trimming
- missing or inconsistent number of channels (if the project assumes a fixed sensor layout)

5.2. InfluxDB model

InfluxDB stores all time-indexed numeric streams produced by the platform. In the current codebase, bucket names are not hardcoded; they are read from environment variables:

- RAW_BUCKET, by default set to Raw_Data
- CLEANED_BUCKET by default set to Cleaned_Data
- OMA_BUCKET by default set to OMA_Data

5.2.1. Buckets: Raw_Data, Cleaned_Data, OMA_Data

Raw signals (RAW_BUCKET)

Writer: services ingester uploader.py, using to_points() and write_raw_to_influx()

- Measurement: "accel"
- Tags:
    - Structure = <structure_name> (e.g., "Sanctuary_Vicoforte")
    - Sensor = (e.g., "CH01"...)
- Field:
    - value (float) - the acceleration sample
- Time precision: nanoseconds, using WritePrecision.NS

Timestamp generation (as implemented): samples get timestamps as t0 + i * dt_ms, where dt_ms = int(1000 / sampling_hz), and then the code converts this to ns. This works well when sampling_hz divides 1000, for example 100 Hz gives 10 ms. This is an important assumption to document.

Cleaned signals (CLEANED_BUCKET)

Writer: services/cleaning_service/worker.py (_write_cleaned_data())

- Measurement: "accel" (same unit as raw; bucket distinguishes stage)
- Tags:
- Structure = "Sanctuary_Vicoforte", this is set in setting.py and can be overridden from UI or .env as environment variable.
- Sensor = "CH01"...
- Field:
- value (float)
- Time precision: nanoseconds (WritePrecision.NS)
- Sampling frequency: reduced by integer factor CLEANING_DOWNSAMPLE (default 2)

Timestamp downsampling: The cleaner reuses raw timestamps and samples them again (timestamps_ns[::ss]) to match the downsampled signal.

- Field names are fixed. The signal field is called value, and the OMA outputs use frequency_hz, mac, and quality.
- Measurement names are fixed. The raw and cleaned signals use accel, and the modal results use oma_modes.
- Structure tagging: ingestion writes Structure as the monitored structure name. The cleaner and analyzer use configured STRUCTURE filters and writers.

- OMA_BUCKET is low volume, so it can have the longest retention. This supports longer period or even multi month trend dashboards and reporting. We can also do a sensitivity analysis on mode tracking to select an efficient value for the retention policy.
- Field names are fixed. For signals, the field is value. For OMA outputs, the fields are frequency_hz, mac, and quality.
- Measurement names are fixed. Raw and cleaned signals use accel, and modal results use oma_modes.
- Structure tagging works as follows. Ingestion writes Structure as the monitored structure name. The cleaner and analyzer use configured STRUCTURE filters and writers.
- OMA_BUCKET has low volume, so it can keep the longest retention. This supports multi month trend dashboards and reporting.
- CLEANED_BUCKET has medium volume, so it can keep longer retention. This supports re analysis without re ingestion.
- RAW_BUCKET has the highest volume, so it needs shorter retention. It should still be long enough for debugging and reprocessing.

Because the bucket separation already shows the processing stage, for retention a tiered plan still makes sense.

Retention strategy discussion:

Run specific parameters, like filter cutoffs, downsampling factor, FDD settings, and tracking thresholds, are not stored as tags. They are stored in PostgreSQL run tables, cleaning_runs and analysis_runs.

- OMA: measurement "oma_modes" with tags Structure and ModeID. This gives, for each structure, k time series, where k is the number of tracked modes and it is usually small.
- Raw and Cleaned: measurement "accel" with tags Structure and Sensor. This gives, for each structure, N time series per bucket.

In InfluxDB, a series is defined by the measurement name and the tag set.

Influx series are determined by a measurement and a tag set.

oma_modes, Structure=..., ModeID=M02 frequency_hz=..., mac=..., quality=...

oma_modes, Structure=..., ModeID=M01 frequency_hz=..., mac=..., quality=...

Time precision is seconds, using WritePrecision.S. The system writes one observation for each pair of analysis run and mode, and it timestamps it at the midpoint of the analyzed time span.

- quality (float) is an algorithm specific quality indicator, for example peak prominence or a coherence proxy
- mac (float) is the similarity score used for tracking, based on the Modal Assurance Criterion
- frequency_hz (float) is the estimated natural frequency of the mode

Fields:

- quality
- mac
- frequency_hz

Tags:

- ModeID = "M01" ... "Mk", which are stable identifiers assigned by the tracking logic
- Structure = "Sanctuary_Vicoforte", currently set by configuration and used in Flux filters and writers

Measurement: "oma_modes"

Writer: services analysis_service worker.py, in _write_oma_modes()

OMA results are stored in OMA_BUCKET.

5.3. PostgreSQL model

PostgreSQL stores the relational “control plane” of the platform: (i) monitoring metadata (what is being monitored) and (ii) the audit trail of pipeline execution (what was ingested, how it was processed, and what events were produced). In your codebase, these tables are defined in services/metadata_api/models.py and created at API startup via Base.metadata.create_all(...).

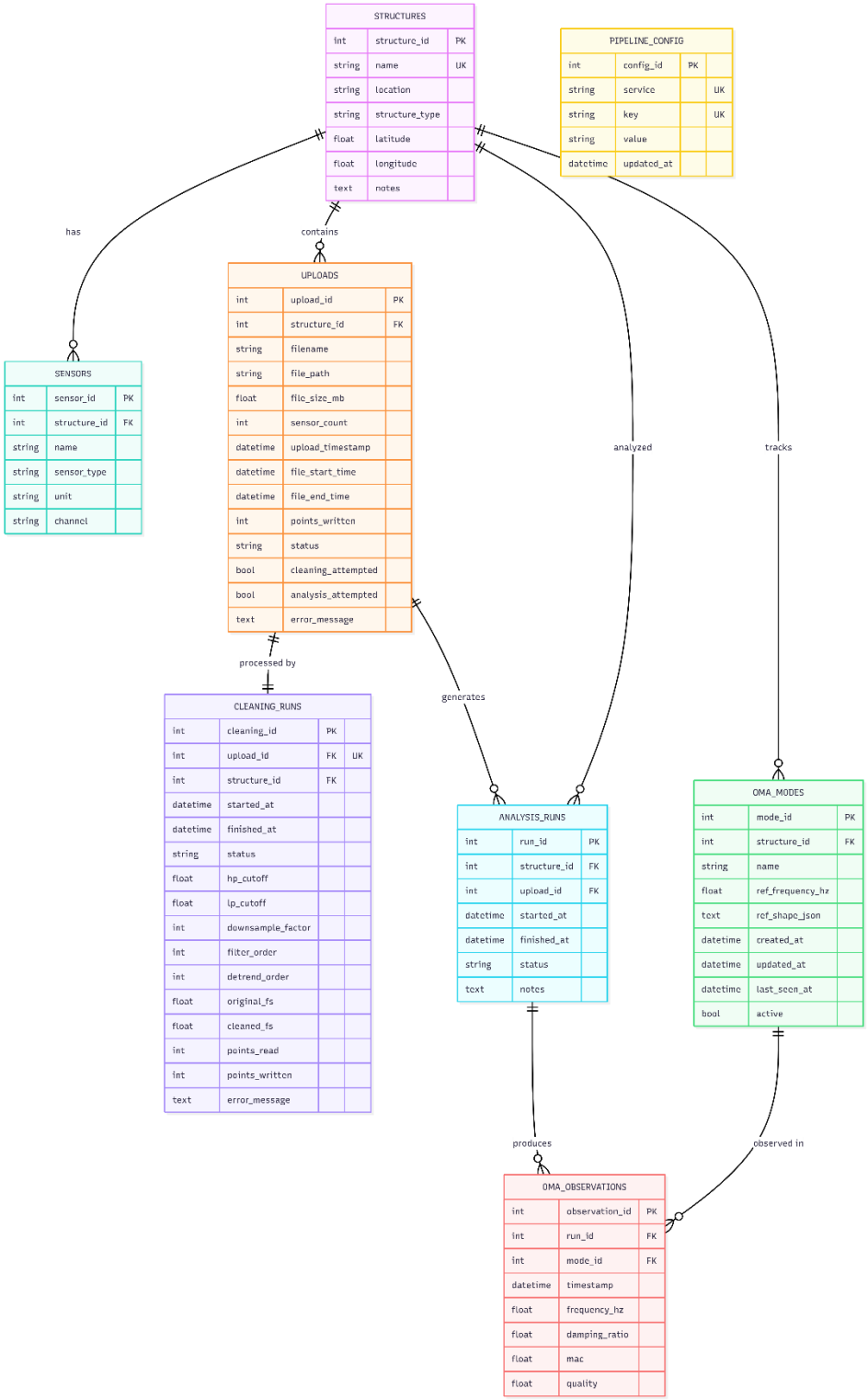


Figure 5.1 Relational data model in PostgreSQL, including structures and sensors, ingestion, cleaning, and analysis run logs, and the modal results tables.

5.3.1. Domain metadata tables

Structures: Represents a monitored asset (e.g., “Vicoforte sanctuary”).

Columns:

- structure_id, primary key, auto increment, indexed
- name, unique, not null, indexed
- location, nullable
- structure_type, nullable
- latitude and longitude, nullable
- notes, nullable

Key constraint

- name is unique, so there is one canonical structure record for each name

Sensors: this table represents sensors or channels attached to a structure, for example accelerometer channels.

Columns:

- sensor_id (PK, autoincrement)
- structure_id (FK → structures.structure_id, ON DELETE CASCADE, indexed, NOT NULL)
- name (NOT NULL, indexed)
- sensor_type (nullable, e.g., "accelerometer")
- unit (nullable, e.g., "m/s^2")
- channel (nullable label)

Key constraint

- uq_sensor_per_structure: UNIQUE(structure_id, name) (sensor names must be unique inside one structure)

5.3.2. Provenance and run registries

uploads_log, ingestion registry: this table is the ground truth of what entered the system and if ingestion succeeded.

Columns:

- upload_id, primary key, auto increment
- structure_id, foreign key to structures.structure_id, on delete cascade, indexed, not null

- filename, not null
- file_path, nullable
- file_size_mb, nullable
- sensor_count, nullable
- upload_timestamp, TIMESTAMPTZ, default utcnow, not null
- file_start_time, TIMESTAMPTZ, nullable, this is t0 parsed from the file
- file_end_time, TIMESTAMPTZ, nullable, computed from t0 plus duration
- points_written, nullable
- started_at, TIMESTAMPTZ, default utcnow, not null
- finished_at, TIMESTAMPTZ, nullable
- status, default "pending", not null, expected values are pending, success, failed
- cleaning_attempted, boolean, default false, not null
- analysis_attempted, boolean, default false, not null
- error_message, nullable
- notes, nullable

Key constraint

- uq_structure_file_timestamp: UNIQUE(structure_id, filename, upload_timestamp)

This constraint supports basic deduplication and prevents exact duplicates for the same structure, filename, and timestamp.

cleaning_runs (preprocessing provenance): This table makes cleaning an explicit, reproducible pipeline stage and avoids “hidden” preprocessing.

Column:

- cleaning_id (PK, autoincrement)
- upload_id (FK → uploads_log.upload_id, ON DELETE CASCADE, indexed, NOT NULL)
- structure_id (FK → structures.structure_id, ON DELETE CASCADE, indexed, NOT NULL)
- started_at (default utcnow, NOT NULL)
- finished_at (nullable)
- status (default "running", NOT NULL) - expected: running | success | failed
- Cleaning parameters:
    - hp_cutoff (nullable)
    - lp_cutoff (nullable)
    - downsample_factor (nullable)

    - filter_order (nullable)
    - detrend_order (nullable)
    - original_fs (nullable)
    - cleaned_fs (nullable)
- Output tracking:
    - points_read (nullable)
    - points_written (nullable)
- error_message (TEXT, nullable)
- notes (TEXT, nullable)

Key constraint

- uq_cleaning_per_upload: UNIQUE(upload_id)
- → enforces at most one cleaning run per upload (important for idempotency).

analysis_runs (analysis provenance): Stores when analysis was executed and links analysis outputs/events back to the upload.

Columns:

- run_id (PK, autoincrement)
- structure_id (FK → structures.structure_id, ON DELETE CASCADE, indexed, NOT NULL)
- upload_id (FK → uploads_log.upload_id, ON DELETE CASCADE, indexed, NOT NULL)
- started_at (default utcnow, NOT NULL)
- finished_at (nullable)
- status (default "running", NOT NULL) - expected: running | success | failed
- notes (TEXT, nullable)

Note: unlike cleaning_runs, there is no UNIQUE constraint on upload_id, so the schema allows multiple analysis runs per upload (useful if it will later support re-analysis with different parameters).

mode_observations (modal outputs): Discrete events produced by the analyzer, designed for operator review and auditability.

Columns:

- event_id (PK, autoincrement)
- run_id (FK → analysis_runs.run_id, ON DELETE CASCADE, indexed, nullable)
- timestamp (default utcnow, indexed, NOT NULL)
- severity (VARCHAR(24), NOT NULL) - e.g., Low | Medium | High

- index_value (FLOAT, NOT NULL) - “the value that triggered the event”
- description (nullable)
- confirmed (BOOLEAN, default false)
- comments (nullable)

Reserved fields for future analysis extensions (all nullable):

  - reserved_metric
  - reserved_metric
  - reserved_metric

5.3.3. Idempotency and duplicate-ingestion prevention logic (as implemented)

The platform is designed so that restarts or repeated watcher notifications do not corrupt the database state.

Ingestion idempotency:

- The main guard in the schema is that uploads_log has uq_structure_file_timestamp. It is unique on structure_id, filename, and upload_timestamp.
- What it guarantee is this. Inside one structure, the same filename cannot be logged twice with the same upload_timestamp.

Practical limitation: upload_timestamp is the time of ingestion, not t0 from the file. So the unique rule does not give full deduplication for the same physical file if it is ingested again later with a different upload_timestamp. If we want stronger deduplication, common options are:

- store a file_hash and enforce UNIQUE(structure_id, file_hash)
- enforce uniqueness on structure_id, filename, and file_start_time, since this is often stable for monitoring exports

This can be improved later, but the current behavior should be written clearly.

Cleaning idempotency:

- cleaning_runs uses uq_cleaning_per_upload, which is unique on upload_id.
- Guarantee: only one cleaning run can exist for one uploaded file record. This avoids accidental re cleaning writes that could create inconsistent multiple cleaned

versions, unless we later change the schema on purpose to support versioned cleaning.

Analysis repeatability: The schema allows multiple analysis_runs per upload (no uniqueness constraint), which is useful for supporting re-analysis with improved algorithms or baselines while preserving provenance.

6. Implementation (codebase-driven)

This chapter explains the implementation of the platform in the provided repository, highlighting the specific modules, data flow, and methods used to ensure the pipeline remains reproducible and easy to debug. The codebase adopts a “storage-mediated” microservice architecture: services interact indirectly via InfluxDB (for time-series data) and PostgreSQL (for metadata and trace logs), with only minimal HTTP calls for ingestion triggers.

Repository structure, implementation view. The implementation is organized into:

- shared folder, which has settings, logging, and Influx utilities that all services use.
- services metadata_api folder, which is a FastAPI CRUD service that exposes structures, sensors, logs, and events.
- services ingestion folder, which includes the ingestion API, the file watcher, the parser, and the uploader.
- services cleaning_service folder, which has the signal cleaning worker, and services analysis_service folder, which has the OMA and FDD analysis worker.
- scripts folder, which contains helper scripts for seeding the monitored structure and sensors.

6.1. Shared utilities (settings, logging, Influx helpers)

The shared package covers the common parts that all services need, like configuration, structured console logging, and InfluxDB write utilities.

The configuration is done through .env and Pydantic settings.

shared/settings.py defines a Settings object using pydantic_settings.BaseSettings. Required environment variables include:

- PostgreSQL connection string (DATABASE_URL)
- Influx parameters (INFLUX_URL, INFLUX_ORG, INFLUX_TOKEN)
- bucket names (RAW_BUCKET, CLEANED_BUCKET, OMA_BUCKET)
- ingestion parameters (HOT_FOLDER, SIZE_MIN_MB, STABLE_SECONDS, INGEST_URL)
- cleaning parameters (CLEANING_HP, CLEANING_LP, CLEANING_DOWNSAMPLE, CLEANING_FILTER_ORDER, CLEANING_DETREND_ORDER)

This design keeps the same code runnable in multiple environments (local dev now; Docker later) by changing only the runtime configuration.

The logging helper in shared/logging.py provides a simple setup_logging() wrapper around Python's standard logging module.

All services log using the same format: timestamp, level, logger_name, message. This uniform format matters when we try to line up watcher, ingestion, cleaning, and analysis actions using timestamps.

Influx helpers (client + chunked writer).

shared/influx.py includes:

- get_influx_client(), which creates an InfluxDB client using the configured URL, token, and org
- write_points(), which writes influxdb_client.Point objects in chunks, default 50_000, so it avoids high memory use when uploading large hourly files.

6.2. Metadata API (schemas, routers, CRUD)

The Metadata API is a FastAPI service in services/metadata_api/. It provides CRUD endpoints for platform metadata and traceability logs. It uses SQLAlchemy ORM models and Pydantic schemas.

Database layer: database.py defines:

- a SQLAlchemy engine created from DATABASE_URL
- a SessionLocal factory
- Base = declarative_base()
- get_db() as a dependency for the FastAPI endpoints

models.py defines the domain models that the platform uses in the relational database:

- structures and sensors describe the monitored asset and the measurement channels.
- uploads_log tracks ingestion attempts, timestamps, and status.
- cleaning_runs stores the cleaning parameters and the run outcome, success or fail.
- analysis_runs stores the bookkeeping for analysis runs.
- mode_observations stores the tracked modal parameters for later review and for trend dashboards.

The schema enforces several platform invariants through unique constraints. For example, it allows only one sensor name per structure, and only one cleaning run per upload.

Field mapping happens in the CRUD operations. For example, StructureCreate.type is manually mapped to the database column structure_type in the CRUD layer. This keeps the external API clean, while the internal schema stays explicit.

Routers and CRUD separation:

- routers folder files define the REST endpoints. They handle FastAPI routing, validation, and error handling.
- crud.py contains the real database operations, like create and list structures and sensors, insert upload logs, mark upload success or failure, insert and list events, create and mark cleaning runs, and similar tasks.

This separation keeps HTTP concerns, like request and response, separate from the persistence logic.

6.3. Ingestion service

The ingestion API (services/ingestion/app.py) is the entry point for new StructuralX hourly files detected by the watcher. Its responsibilities are:

1. validate the file path,
2. parse the StructuralX export format,
3. create a trace entry in PostgreSQL (uploads_log),
4. stream raw time-series points into InfluxDB (Raw_Data),
5. update the upload log status, and
6. trigger cleaning automatically (current implementation).

Endpoint contract: the ingestion API exposes POST /ingest and it expects a JSON payload like { "filepath": "..." }.

Structure resolution: when it starts, the ingestion service looks up the monitored structure in Postgres by name, using MONITORED_STRUCTURE, default is "Sanctuary_Vicoforte". If it does not find it, it falls back to a default id. This is the current fallback behavior in the implementation.

Idempotency and duplicate handling: before it writes to InfluxDB, ingestion tries to insert a new uploads_log record. If it hits a uniqueness violation, meaning a duplicate, ingestion returns a duplicate response and it does not reprocess the same file.

Raw write strategy: Raw data are written as Influx points:

- measurement: accel
- tags: Structure=<name>, Sensor=CH01…
- field: value
- timestamp: nanosecond precision derived from t0 and sample index

The uploader uses chunked writing and includes retry logic (3 retries with incremental backoff) to tolerate transient InfluxDB failures.

6.4. watcher logic, parser, uploader

The ingestion pipeline starts from a local file watcher in services/ingestion/watcher.py. It monitors a configured hot folder. The design is Windows friendly, but it is still portable.

File watcher policy:

- The system processes only files that end with "_Fast.txt". This helps filter out partial or temporary files.
- A candidate file must meet these rules:
    - minimum size, set by SIZE_MIN_MB, default 10 MB
    - stability window, meaning the file size stays unchanged for STABLE_SECONDS, with a default set to 60 s

The stability rule is implemented in windows_file_utils.is_file_stable(). It checks the file size, waits for the configured stable window, and then checks again that the size did not change.

Parser, StructuralX format: services/ingestion/parser.py parses:

- t0 from line 1 (local time in Europe/Rome converted to UTC),

- dt from line 2,
- a table with a tempo (time) column and 12 sensor columns,
- a known “trailing zeros tail” (trimmed by detecting last non-zero row).

The sampling frequency is taken by priority from the tempo column when it is stable. If not, the code derives it from dt.

Uploader, point generation: services/ingestion/uploader.py converts the N by M matrix (N rows of data for M channels) into Influx points by looping over samples and channels. It computes the timestamp as t0 + i * dt.

6.5. Cleaning service

Signal cleaning is implemented in services/cleaning_service (pipeline.py and worker.py). In the current version, cleaning can be run in two ways:

- automatically as a post-ingestion step (ingestion triggers process_cleaning()), and/or
- as a standalone polling worker (main_loop()), which scans Postgres for uploaded files that have not yet been cleaned.

Cleaning objective: the cleaning stage turns raw accelerometer data into a consistent analysis ready form. It removes offsets and trends, limits the signal to a target frequency range, and reduces the sampling rate to improve efficiency.

6.5.1. Algorithm (SASI style cleaning)

For each sensor channel, the cleaner does:

1. mean removal,
2. polynomial detrending, default is linear,
3. Butterworth filtering, high pass, low pass, or bandpass, using zero phase filtering with filtfilt,
4. downsampling by an integer factor, default is 2.

Influx read and write:

- The cleaner queries Raw_Data with Flux for each sensor between upload.file_start_time and upload.file_end_time.
- It downsamples the timestamps in the same way as the downsample factor, and writes the cleaned series to Cleaned_Data. It keeps the same tags, Structure and Sensor, and the same measurement name, accel.

Cleaning run traceability: each cleaning operation writes one row into cleaning_runs table that includes:

- the parameters used, such as hp_cutoff, lp_cutoff, downsample_factor, filter_order, and detrend_order,
- the sampling frequencies, meaning original_fs and cleaned_fs,
- the number of points read and written, plus status.

6.6. pipeline implementation, run logging

The overall pipeline is implemented as a sequence of stages connected through the storage backbones (InfluxDB + PostgreSQL), with explicit run logging and stage flags.

Stage flags in uploads_log: Each upload evolves through:

- status = success|failed (raw ingestion outcome),
- cleaning_attempted = True after the cleaner processes the upload (even if it fails, to avoid infinite retries),
- analysis_attempted = True after the analyzer processes the upload (also set even on failure to avoid repeated crashes).

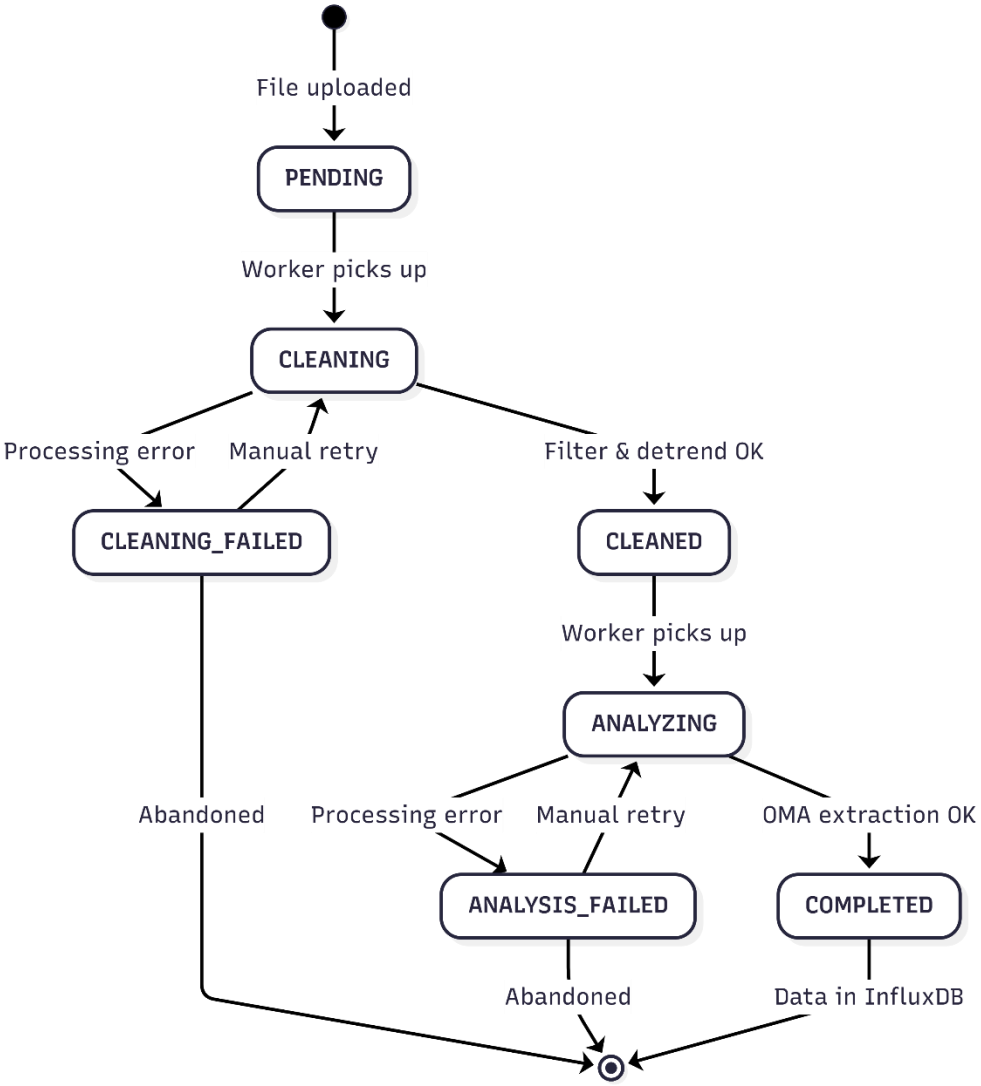


Figure 6.1 Upload processing states from pending to completed, including failures and retries.

This design makes the pipeline restartable. Services can crash and restart, and the next run can pick up where it left off by scanning for uploads that are still not processed.

Mode tracking is performed by comparing estimated mode shapes using MAC. In practice this helps to avoid false 'frequency jumps' due to mode swapping or closely-spaced peaks, which is a known risk when relying only on peak picking in spectra [7].

The analysis_runs table links each observation set to the related upload and records the parameter choices, so the results stay reproducible.

• PostgreSQL: modes (stable list of ModeID per structure) and mode_observations (frequency, mac, quality per analysis_run_id).

• InfluxDB uses the measurement oma_modes, with tags Structure and ModeID, and fields frequency_hz, mac, and quality.

For each upload time range, the worker estimates a set of modes and writes the outputs per mode, not as a single scalar index. It also assumes that a successful cleaning run exists for that upload.

```
uploads_log.analysis_attempted == False
```

```
uploads_log.cleaning_attempted == True
```

```
uploads_log.status == "success"
```

In the current implementation, the analyzer (services/analysis_service/worker.py) is implemented as a polling worker that selects uploads ready for analysis, reads cleaned signals from InfluxDB, performs automated OMA using Frequency Domain Decomposition (FDD), and stores modal results in both InfluxDB and PostgreSQL [7].

The script migrate_cleaning.py shows how the platform integrated a new pipeline stage, by adding flags and creating the cleaning_runs table. The platform uses a similar migration pattern when it adds the modal tables needed for OMA based monitoring.

Migration and schema evolution:

- modes and mode_observations store the identified modal set for a structure, plus per run observations of frequency and tracking similarity, using MAC.
- analysis_runs is created when OMA and FDD analysis starts and it is updated when the run finishes. It stores the algorithm version and parameter snapshots.
- cleaning_runs stores one record for each upload and cleaning attempt. A uniqueness constraint on upload_id enforces this.

The platform uses explicit run registries so the pipeline can restart safely and stay auditable.

6.7. Configuration and local operation

Environment variables configuration: A complete example is provided in .env.example. The most important operational settings are:

- database endpoints for Postgres and Influx,
- bucket names
- the hot folder path and the raw data file stability setting
- cleaning parameters

Local startup sequence (current non-Docker workflow): A typical run sequence is:

1. Start Postgres and InfluxDB (native or Docker).
2. Create DB tables (Metadata API auto-create or one-liner Base.metadata.create_all()).
3. Seed the monitored structure and 12 sensors (either manually via Swagger or via scripts/seed_sanctuary.py).
4. Start services:
    - Metadata API
    - Ingestion API
    - Watcher module (python -m services.ingestion.watcher)
    - Analysis worker (python -m services.analysis_service.worker)
    - (optional) Optional: Use a cleaner worker for periodic reprocessing; otherwise, cleaning is automatically triggered.

Operational checks.

- Health endpoints exist for metadata and ingestion, at /healthz.
- Successful successful ingestion, cleaning, and analysis can be verified by checking:
    - the status and flags in uploads_log,
    - cleaning_runs and analysis_runs,
    - that time series exist in the Influx buckets.

Docker/Grafana readiness (what is implemented vs planned): At this stage, runtime configuration is already centralized in environment variables, which is the key prerequisite for Dockerization. Grafana can be added as a deployment component that queries the existing buckets and the Postgres tables; however, provisioning files are not currently part of the repository and would be added during the deployment packaging step.

7. Signal preprocessing (Cleaning) and data quality issues

7.1. Why vibration signals are non-stationary and why preprocessing matters

In operational SHM, acceleration recordings are rarely "clean laboratory signals." Even when the structure is undamaged, the measured response changes over time because the excitation changes (wind, microtremors, traffic, local activities) and because environmental/operational variability (EOV) modifies boundary conditions and material properties (e.g., temperature gradients, humidity/moisture, mass/loading changes). These effects can shift modal frequencies by several percent - often comparable to early damage-related shifts - so the same feature value can drift even in healthy conditions [1].

From a signal processing viewpoint, this means that:

- low-frequency drift and offsets can bias band limited spectral estimates and make modal trends harder to interpret,
- transient spikes can dominate statistics and can create false events,
- frequency features, like modal frequency estimates and their similarity or quality measures, are sensitive to non-stationarity, windowing, and the effective band kept by preprocessing.

Because of this, a repeatable preprocessing stage is not optional. It is what makes feature extraction comparable across days, sensors, and operating conditions. The SHM reference text also treats data cleansing as a core step of the SHM workflow, not something added at the end.

7.2. Cleaning pipeline (mean removal, detrend, Butterworth bandpass, downsample)

This thesis implements a deterministic, parameterized cleaning pipeline in the platform's cleaning worker (services/cleaning_service/pipeline.py). Conceptually, the pipeline maps directly to standard SHM/DSP motivations:

Step A - Mean removal (DC offset removal)

The pipeline first subtracts the sample mean channel-wise. This is a standard normalization step used to remove DC offsets when the damage mechanism is not expected to manifest as a DC component in the measured response - typical for accelerometers [1].

Design caveat important to say clearly: for strain signals, the mean can have physical meaning, like permanent deformation. So, if we remove the mean, we may throw away damage information. The SHM reference gives an example where mean shifts show plastic strain [1].

Step B - Polynomial detrend, default is linear

After mean removal, the cleaner estimates and subtracts a low order polynomial trend. The default order is 1, so it is linear, and it uses least squares. The goal is to remove slow drift, like thermal drift, sensor drift, or slow tilt. If we do not remove it, it can leak into the low frequency spectral content and it can overestimate broadband energy measures.

Step C - Butterworth high pass, low pass, or band pass filtering, zero phase

Next, the pipeline applies a Butterworth filter using the imported "filtfilt" function, so the filtering is zero phase and it does not add phase distortion. This is useful when later steps depend on the waveform shape, or when we want to change the magnitude content without shifting peaks in time.

Filtering is also the main tool the platform uses for data cleansing in the SHM sense. It suppresses frequency components that are known to be non-structural, like very low drift, or that are mostly measurement noise [1].

Anti-aliasing principle (critical when downsampling): the SHM book illustrates how aliasing creates a fictitious spectrum and motivates filtering to suppress out-of-band energy before reducing the sampling rate [1].

Step D - Downsampling (decimation by integer factor)

Finally, the pipeline downsamples by an integer factor ss (simple decimation after filtering), reducing storage and compute costs and focusing subsequent analysis on the preserved band. This aligns with the platform's goals (continuous monitoring and long-term storage).

There is a small but important theory point. Once a continuous time signal is sampled, we cannot undo aliasing that already happened during the acquisition. A digital filter applied later cannot fix it after the fact [1].

However, for downsampling, digital low-pass/band-limiting before decimation is exactly what prevents additional aliasing from the decimation step.

Table 1 Cleaning parameters stored per run and rationale

Parameter	Typical value	Rationale
hp (Hz)	0.5	This value is a high-pass filter and removes low frequency drift and sensor bias.
lp (Hz)	15.0	This is a low-pass filter that reduces high-frequency noise and helps with anti-aliasing.
filter order	4	It keeps a balance between attenuation strength and signal distortion.
detrend order	1	It removes linear trends before spectral analysis.
downsample factor	2	It reduces data volume while keeping modal content.
original_fs (Hz)	100	It defines the real signal time base and how to read the cutoff.
cleaned_fs (Hz)	50	It is the effective sampling rate after resampling for later analysis.

All parameters are saved for each cleaning_run, so results can be reproduced and traced even if the global configuration changes later.

7.3. Practical issues observed

(i) Early/late spikes and “tail artifacts”

In real monitoring exports, the first and the last part of a file can contain artifacts, like start up transients, acquisition discontinuities, or operator caused disturbances.During the operational exercise, a more stable segment was often seen after an initial period, around 2 minutes into the file. This suggests either:

- cropping rules, like dropping the first and last T seconds, or
- a smarter baseline or stable window selector, that picks the most stationary sub window by minimizing variance or maximizing spectral consistency.

This is not only data cleaning polish. One spike can bias the estimated spectra and can change FDD peak picking, especially if the analysis window is short or if the spike is not removed by filtering or cropping.

(ii) EOV-driven drift that masquerades as damage

Even perfect signal cleaning does not remove true EOV effects: temperature/loading changes can shift modal properties by ~5-10% in field observations, and the SHM literature emphasizes that we should measure EOV parameters where possible (temperature gradients, humidity, operational state) to avoid incorrect diagnosis [22].

This connects directly to platform evolution: the current pipeline is ready to ingest such channels later (and we already planned the expansion to environmental/static sensors), Overal, the feature interpretation without EOV context is inherently uncertain.

7.4. Reproducibility: why parameters must be logged (cleaning_runs)

Preprocessing is a step where small hidden differences can enter an SHM pipeline. Different filter cutoffs, detrending orders, and downsampling rates can lead to different features even from the same raw file. Because of this, the platform logs every cleaning execution in cleaning_runs, including:

- cutoff frequencies and filter order
- detrend order
- downsample factor
- original and cleaned sampling rates
- status and counts of points read and written

This is the minimum needed for traceability. If an anomaly shows up later, it is possible to repeat the exact same transformation and see what produced the cleaned data. It is also possible to compare runs in an objective way, and to tell apart algorithm updates from real structural modifications. This fits the wider SHM workflow, where data acquisition, cleansing, and normalization are formal steps in the pattern recognition process, not just small implementation details.

8. Analysis, indices, and damage-event logic

This chapter describes the analysis stage of the SHM platform. The analyzer consumes cleaned acceleration time series from InfluxDB.

8.1. Purpose and position of the analysis stage in the SHM pipeline

The analysis stage takes cleaned vibration data and makes a small set of modal parameters. These can be saved, tracked over time, and checked by an operator. In the current implementation, this stage is an event driven microservice called the analysis worker. It runs only after the cleaning stage finishes successfully. The worker reads cleaned_ready messages from a Redis Streams channel, and each message tells it to analyse one time window for a specific upload and structure.

Two design decisions shape this chapter. First, the analyzer reads only from the cleaned time series bucket in InfluxDB, with measurement name accel and tags Structure and Sensor. This keeps concerns separate. Filtering, detrending, and resampling happen before, so the analyzer can assume the window is already good for modal identification.

Second, the analyzer writes outputs in two forms. It writes structured metadata records to PostgreSQL, and it writes time series points to an InfluxDB bucket that is dedicated to OMA outputs. This split is done on purpose. PostgreSQL keeps traceable run level facts, like which upload was analysed, which parameters were used, and which modes were found. InfluxDB stays the natural place for trend plots and for looking at changes over time.

At the start of each run, the worker creates an analysis_run record (status = running). It then queries the cleaned series for the requested window, computes modal estimates using an automated Frequency Domain Decomposition (FDD) routine, and persists the resulting

modal observations. When the run succeeds, the worker marks the analysis_run as completed and publishes an analysis_done event that includes summary fields such as the number of modes detected. If a failure occurs (for example, missing data in the window), the worker records an error status and leaves the message unacknowledged so it can be reclaimed later, which is a pragmatic reliability choice in a stream-based pipeline.

8.2. Inputs, time window alignment, and operational assumptions

Each analysis execution is defined by a time interval $[t_{start}, t_{end}]$ received from the upstream cleaning stage. The worker queries the cleaned bucket for all acceleration samples in this interval, filtered by the monitored structure name and the acceleration measurement. The query returns multiple sensor streams, each represented as a pair (ts, xs), where ts is the timestamp vector (in nanoseconds), and xs is the cleaned acceleration sequence for sensor s.

The analyzer then constructs a multi-channel data matrix $X \in R^{N \times M}$, where M is the number of sensors and N is the number of samples used per channel. Because the database query may return slightly different lengths across sensors (for example, due to missing samples or writing delays), the current implementation uses a conservative alignment rule: it truncates every sensor series to the minimum available length across the selected sensors, and it requires this common length to be at least 10 samples. The reference time vector is taken from the first sensor after truncation; this is acceptable for visualization and for frequency-domain processing, but it also reveals a limitation: the current analyzer does not resample or re-synchronize signals if the channels are not already aligned by the cleaning stage.

The analyzer uses the cleaned sampling rate, cleaned_fs, carried in the event payload (with a fallback default when absent). This is important because the modal identification routine operates in the frequency domain and relies on a consistent mapping between frequency bins and physical frequency values. In operational terms, the analyzer assumes:

- the window contains quasi-stationary data suitable for spectral estimation (at least within the chosen segment lengths);
- all channels correspond to the same physical structure and window interval;
- the cleaned signals are already de-trended and filtered in a way that does not destroy the modal content of interest.

These assumptions are realistic for a prototype SHM platform, yet they should be read as a contract between services. If later deployments require robustness to dropped samples or

asynchronous sensors, the alignment step must evolve from simple truncation to time-aware synchronization (e.g., interpolation onto a shared grid, or discarding channels that violate timing quality thresholds).

8.3. Automated operational modal analysis by Frequency Domain Decomposition

The implemented analysis method is a lightweight operational modal analysis procedure built around Frequency Domain Decomposition (FDD). In its classical form, FDD exploits the fact that, under broad-band excitation and linear response assumptions, the output cross-spectral density matrix contains modal information that can be separated by a singular value decomposition. A practical and influential presentation of the approach is given by Brincker, Zhang, and Andersen in the context of output-only identification [23] [7].

8.3.1. Cross-spectral density matrix estimation (Welch-type averaging)

Cross-spectral density matrix S(f) and its Hermitian property:

$$S_{xx}(f) = E[X(f)X(f)^H] \qquad \text{Eq.(8.1)}$$

Given the aligned multi-channel matrix X, the analyser estimates the cross-spectral density (CSD) between all channel pairs. In implementation terms, the CSD is computed by applying a Welch-type averaged periodogram procedure for each pair (i,j), producing a complex-valued spectrum Sij(f). The Welch estimator reduces the variance of raw periodograms at the cost of frequency resolution, a trade-off that is suitable for routine automated processing [17].

The set of all pairwise estimates forms a spectral density matrix for each frequency bin,

$$\mathrm{S}(\mathrm{f_k}) = \begin{bmatrix} \mathrm{S_{11}}(\mathrm{f_k}) & \cdots & \mathrm{S_{1M}}(\mathrm{f_k}) \\ \vdots & \ddots & \vdots \\ \mathrm{S_{M1}}(\mathrm{f_k}) & \cdots & \mathrm{S_{MM}}(\mathrm{f_k}) \end{bmatrix} \qquad \text{Eq.(8.2)}$$

where $S(f_k)$ is Hermitian by construction, since $S_{ji}(f_k) = \overline{S_{\imath\jmath}(f_k)}$ in the code path.

A small but relevant implementation detail is that the analyzer uses several segment lengths to compute spectra at multiple resolutions. In the current version, the default list is {1024,2048,4096} samples (subject to the available window length). This is not presented as an optimal choice. It is an engineering compromise: a short segment length gives more

averaging (lower variance), while a longer segment length improves frequency resolution and can separate closer peaks.

8.3.2. Singular value decomposition and peak picking

SVD of the CSD matrix and definition of the first singular value/vector:

$$S_{xx}(f) = U(f) \sum(f)\, U(f)^H, \qquad \sigma_1(f) = \max diag\left(\sum(f)\right) \quad \text{Eq.(8.3)}$$

For each frequency bin f_k, the analyzer computes an SVD: and extracts the first singular value $\sigma_1(f_k)$ and the first singular vector $u_1(f_k)$. In FDD, peaks in $\sigma_1(f)$ are interpreted as candidate natural frequencies, while $u_1(f)$ provides an estimate of the corresponding mode shape, up to a scaling and phase factor [7].

Mode candidates are detected by peak-picking in $\sigma_1(f)$ over a configurable frequency band $[f_{min}, f_{max}]$. The current analyzer selects up to N_{max} (max_modes) peaks per spectrum resolution. For each selected peak, it stores:

- frequency f_p at the peak corresponding to the location of the maximum in $\sigma_1(f)$
- mode shape ϕ_p , derived from $u_1(f_p)$ after a normalization step;
- quality score, computed as a simple ratio between peak amplitude and a robust baseline level (median of σ_1 over the frequency band).

The normalization step needs a short explanation because it affects repeatability. The first singular vector can be complex valued, and it can have an arbitrary sign or phase. The implementation sets the phase by rotating the vector, so the component with the largest absolute magnitude becomes real. After that, it takes the real part as a simple prototype choice and scales it to have Euclidean norm equal to one. This gives a consistent real valued shape vector that can be stored and used later for tracking. It is not a full solution for complex modes, but it fits the prototype first approach of this thesis.

Finally, since the analyzer does peak picking at multiple spectral resolutions, it can output duplicate mode candidates close to the same frequency. To reduce this repetition, it groups detected peaks using relative frequency distance and keeps the best quality one in each group. This gives a compact list of modal estimates for the time window, ready for the next mode tracking step.

After peak picking, the implementation performs a lightweight consolidation step. Because spectra are computed at several resolutions (multiple n_{perseg} values), the same physical mode can appear as nearby peaks. Detected candidates are therefore clustered by relative

frequency proximity, and the representative with the highest quality score (peak-to-median ratio) is retained for persistence.

A prototype damping estimate is also calculated for each retained mode using a half power bandwidth rule. This estimate is marked as approximate on purpose, because ambient excitation, limited record length, and close modes can break the assumptions behind the half power method. Still, it gives an understandable damping trend that can be checked together with frequency and similarity metrics.

Half-power bandwidth damping approximation:

$$\varsigma \approx \frac{f_2 - f_1}{2f_n} \qquad \text{Eq.(8.4)}$$

Mode tracking across successive files is implemented by comparing newly detected modes with a set of tracked reference modes stored in PostgreSQL (in the oma_modes table). Each reference mode includes a nominal frequency and a reference (normalized) real mode-shape vector. For each detected mode, candidates are filtered by relative frequency and proximity, then scored using the Modal Assurance Criterion (MAC). If the best MAC exceeds a configurable threshold, the mode is associated with the existing tracked identifier; otherwise, a new tracked mode is created.

Frequency proximity rule used before MAC scoring:

$$|f_{cand} - f_{track}| \leq \Delta f_{max} \qquad \text{Eq.(8.4)}$$

When a detected mode matches an existing tracked mode, the current implementation updates the reference values with a light exponential moving average (EMA). The goal is to follow slow operational drift (for example, due to temperature) without letting single-window noise redefine the reference. A small update factor α is used in the code (α = 0.2 in the current prototype).

Exponential moving average update for tracked modal frequency and mode shape:

$$\begin{aligned} f_t &= \alpha f_{t-1} + (1-\alpha) f_{new}, \\ \phi_t &= \alpha \phi_{t-1} + (1-\alpha)\phi_{new} (then\ normalize\ \phi_t\ ) \end{aligned} \qquad \text{Eq.(8.5)}$$

Persistence is split between the time-series and relational stores. Per-window observations (frequency_hz, damping_ratio when available, MAC, quality) are written to PostgreSQL (table mode_observations) linked to an analysis_runs record, while a compact time series is also written to InfluxDB (bucket OMA_Data, measurement oma_modes). In InfluxDB, ModeID and

Structure are stored as tags and the numerical fields are stored as measurements; the timestamp is set to the end of the analyzed window. This dual persistence supports both dashboarding (InfluxDB) and auditable provenance queries (PostgreSQL).

Finally, when the analysis run finishes successfully, the worker updates the analysis_runs record with status = completed and sends an analysis_done event on the Redis stream. Downstream consumers are not implemented in the current implementation, but this event gives a stable point to connect later alert logic, reporting, or automated exports.

9. Visualization and user interaction layer

This chapter describes the operator-facing layer of the platform. In long-term monitoring, the analytical results are rarely used as a single report. Instead, operators need continuous visibility into the data pipeline status (file arrivals, preprocessing outcomes, analysis outcomes) and into the evolution of modal indicators over time. Current implementation supports this through two complementary interfaces: Grafana dashboards for time-series exploration and a lightweight Management UI for configuration and manual diagnostic actions.

A key design choice is to keep trend exploration separate from pipeline control. Grafana is used to explore time series and compare plots. The Management UI is used to manage parameters, check runs, and handle data manually. Keeping them separate, reduces coupling and makes it easier to validate each interface independently.

9.1. Grafana dashboards

Grafana is the main tool for visualizing time series outputs. In the current platform, raw and cleaned signals are stored in InfluxDB buckets, named Raw_Data and Cleaned_Data, while modal results are stored in a separate bucket called OMA_Data. This separation helps the operator understand the data. Raw data stays unchanged, cleaned data depends on the preprocessing choices, and OMA outputs depend on the selected analysis settings and tracking rules.

Dashboards are organized around three operational questions. First, did the pipeline run successfully for each uploaded file? Second, do the cleaned signals look physically plausible and consistent across time windows? Third, do the extracted modal indicators show stable trends, or do they exhibit abrupt changes that require inspection? Grafana panels address these questions by combining status timelines (upload and run logs) with

quantitative plots of modal frequency, damping estimates, and mode-shape similarity indicators.

For modal monitoring, the most useful view is a trend plot for each mode. It shows frequency in Hz and damping ratio over time, grouped by ModeID. At the same time, MAC values are plotted to check if the mode stays the same. If MAC drops and stays low, it can mean mode swapping, poor excitation, or bad windows, not a real change in the structure. This matches common SHM practice, where clear and traceable results are preferred over one unclear score [1], [8].

Figure 9.1 Grafana dashboard with sample visuals

From a system view, Grafana also works as a light observability layer. By plotting timestamps and run durations for ingestion, cleaning, and analysis, the operator can notice backlogs, for example when the delay between ingestion and analysis keeps growing. It also helps to find resource problems without needing direct shell access to the containers.

Grafana is set up as a container in the same Compose stack, and it connects to InfluxDB through the internal Docker network. The operator opens Grafana using the published host port. This port can be opened only locally or also on a LAN, depending on the binding settings in docker-compose.yml [21]. Official documentation is used for integration details and dashboard provisioning [24].

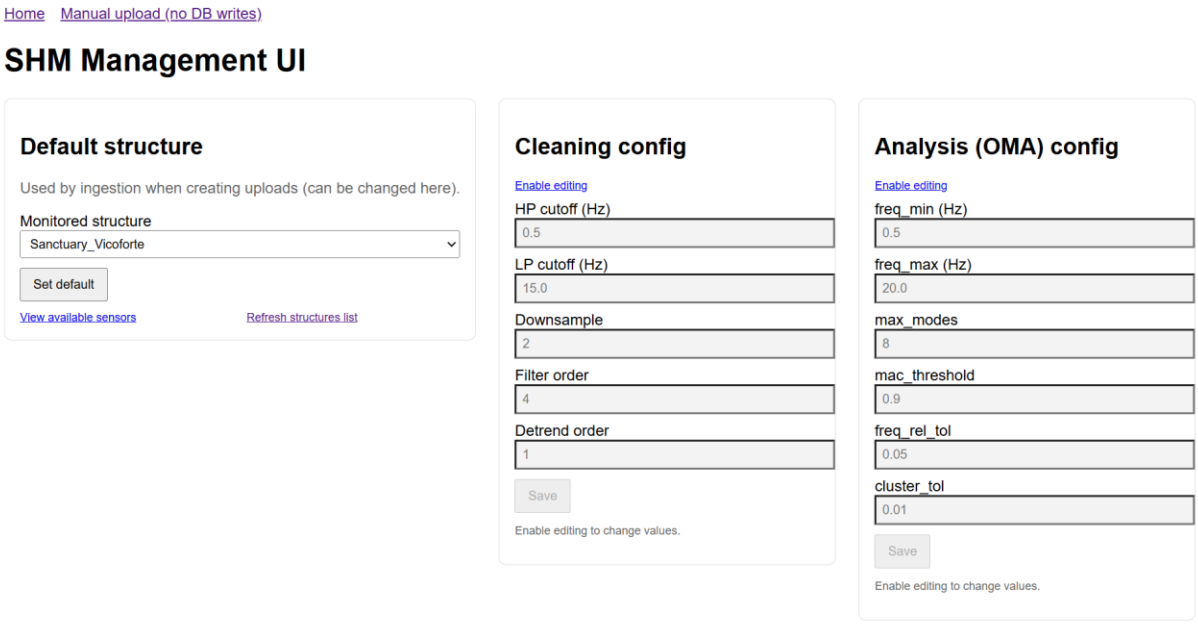
9.2. Web Management UI

In addition to dashboards, current implementation includes a Management UI intended for configuration and controlled manual actions. The UI is implemented as a FastAPI service that serves HTML templates and communicates with the Metadata API for persistent

configuration. This approach keeps the UI lightweight while still providing a structured interface for editing parameters that affect preprocessing and analysis [25].

The UI supports three main task groups. First, it is used for structure and sensor management. This means selecting the default monitored structure, looking at the detected sensors, and updating basic structure metadata. Second, it provides configuration panels to change cleaning parameters, such as filter cutoffs, detrend order, and downsampling factor. It also lets the user change analysis parameters, like frequency range, maximum modes, MAC threshold, and tolerance settings. These parameter changes are stored in PostgreSQL through the Metadata API, so they remain traceable in a recorded run.

Third, the UI includes a manual file cleaning pipeline and a diagnostic page. On this page, an operator can upload one TXT export and create quick plots of the raw signal, FFT and PSD diagnostics, and also a cleaned version. It uses the same preprocessing functions as the automated pipeline. This manual tool does not write to the operational databases. It is treated like a laboratory view, to understand one window and to check that the selected cleaning settings are reasonable. The interactive plots are made with Plotly, which is a practical compromise between responsiveness and implementation complexity [26].



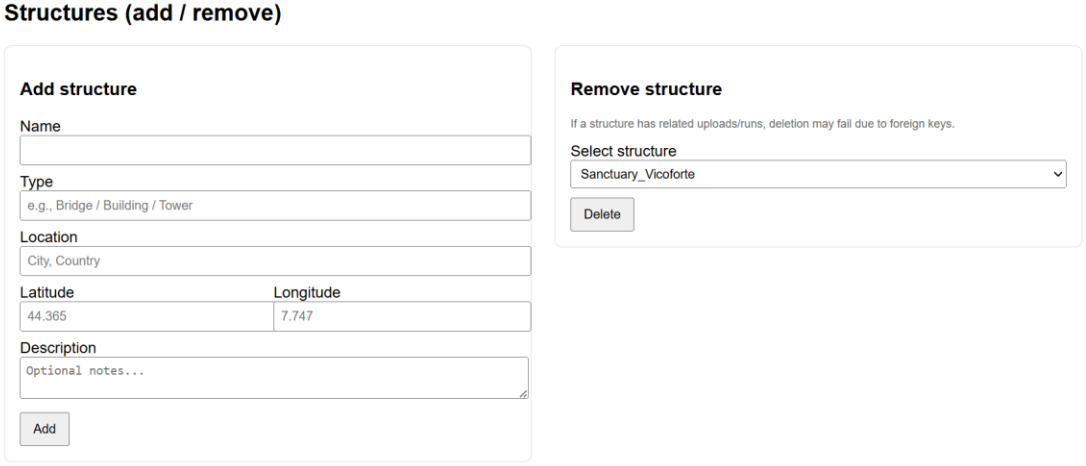


Figure 9.2 Management UI Homepage: default structure selection and parameter configuration editing panel

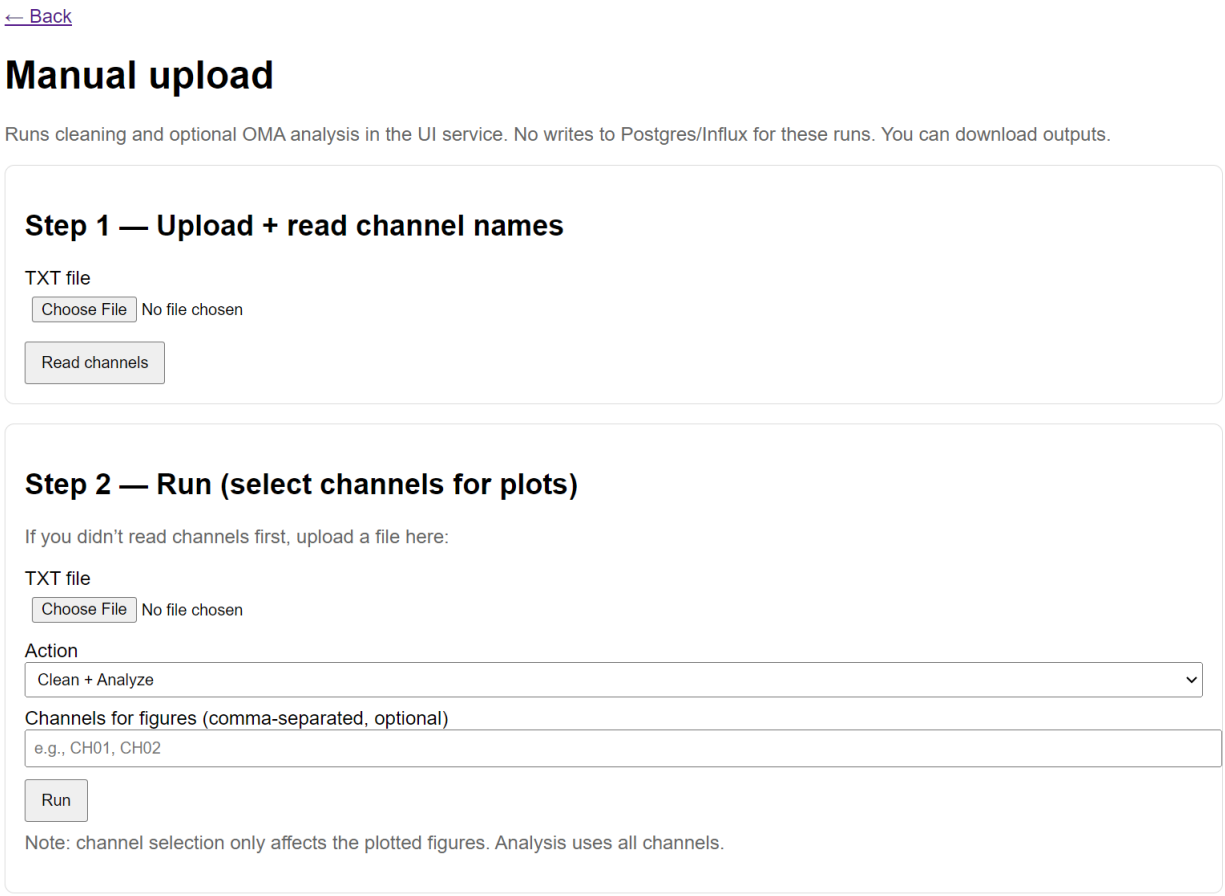


Figure 9.3 Management UI Homepage: Manual pipeline control page (file selection, dynamic channel detection button, cleaning and analysis options)

The UI intentionally does not call worker internals or implement any asynchronous orchestration logic. The automated pipeline remains the responsibility of background workers and the Redis event bus. This separation reduces the risk that UI actions change pipeline state in unexpected ways, and it makes it easier to reason about failure modes in an operational setting.

9.2.1. How parameter configuration works

The parameter configurations inserted via the Management UI are used by both automatic and manual pipelines. Because both pipelines retrieve configurations from the same source (a Postgres database via a metadata API), ensuring consistency between automatic and manual processing. When parameters are updated in the UI, they are immediately available to both pipelines on their next run.

1. Configuration Storage

When the parameters are updated in the Management UI (implemented by @app.post("/config/cleaning") in app.py), they are stored in the Postgres database via the metadata API's /config/cleaning and /config/analysis endpoints.

2. Automatic Pipeline Uses UI configs

Cleaning Service: The automatic cleaning worker (worker.py) calls clean_upload_influx() and this loads parameters using get_cleaning_params() in (pipeline.py)

Analysis Service: The automatic analysis worker in (worker.py:108) calls get_config("analysis") to get parameters from the database

Both services get the latest configs from the metadata API each time they process data

3. Manual Pipeline Uses UI configs

In the /manual/run endpoint (app.py), the code directly fetches the same configs:

cleaning_cfg = await _get_json(f"{meta}/config/cleaning", {})

analysis_cfg = await _get_json(f"{meta}/config/analysis", {})

Then it uses these values for processing.

9.2.2. Performance of the management UI

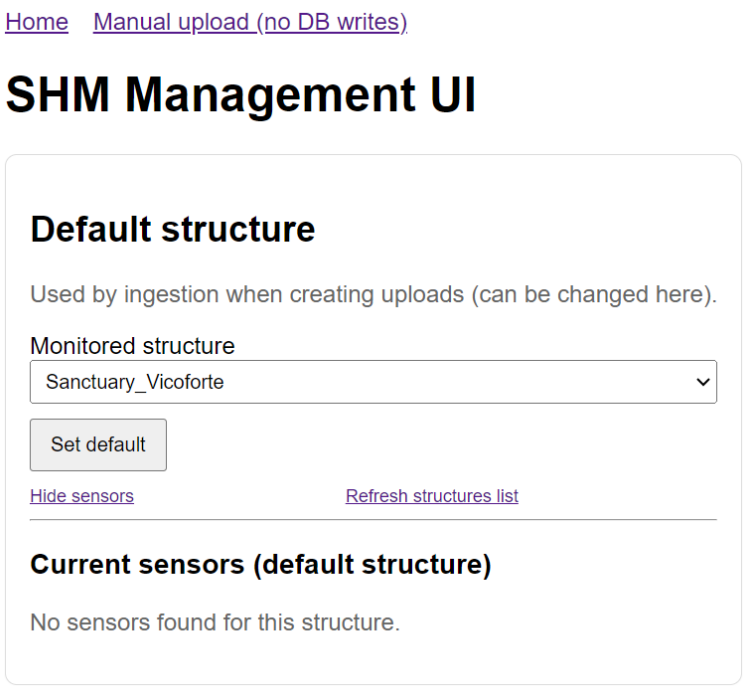


Figure 9.4 No sensors before any automatic ingestion for a default or newly added building

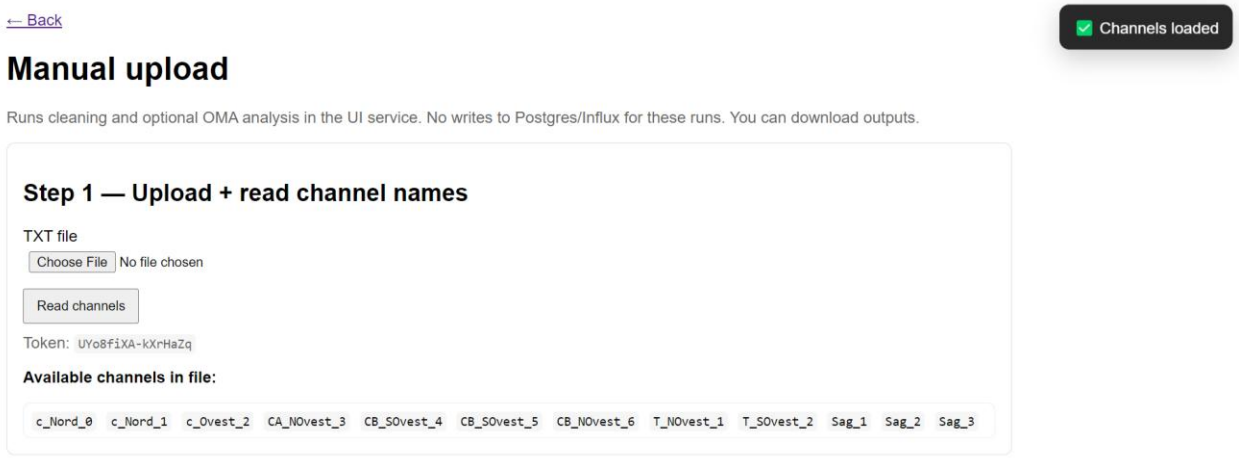


Figure 9.5 Sample channels read from raw data file

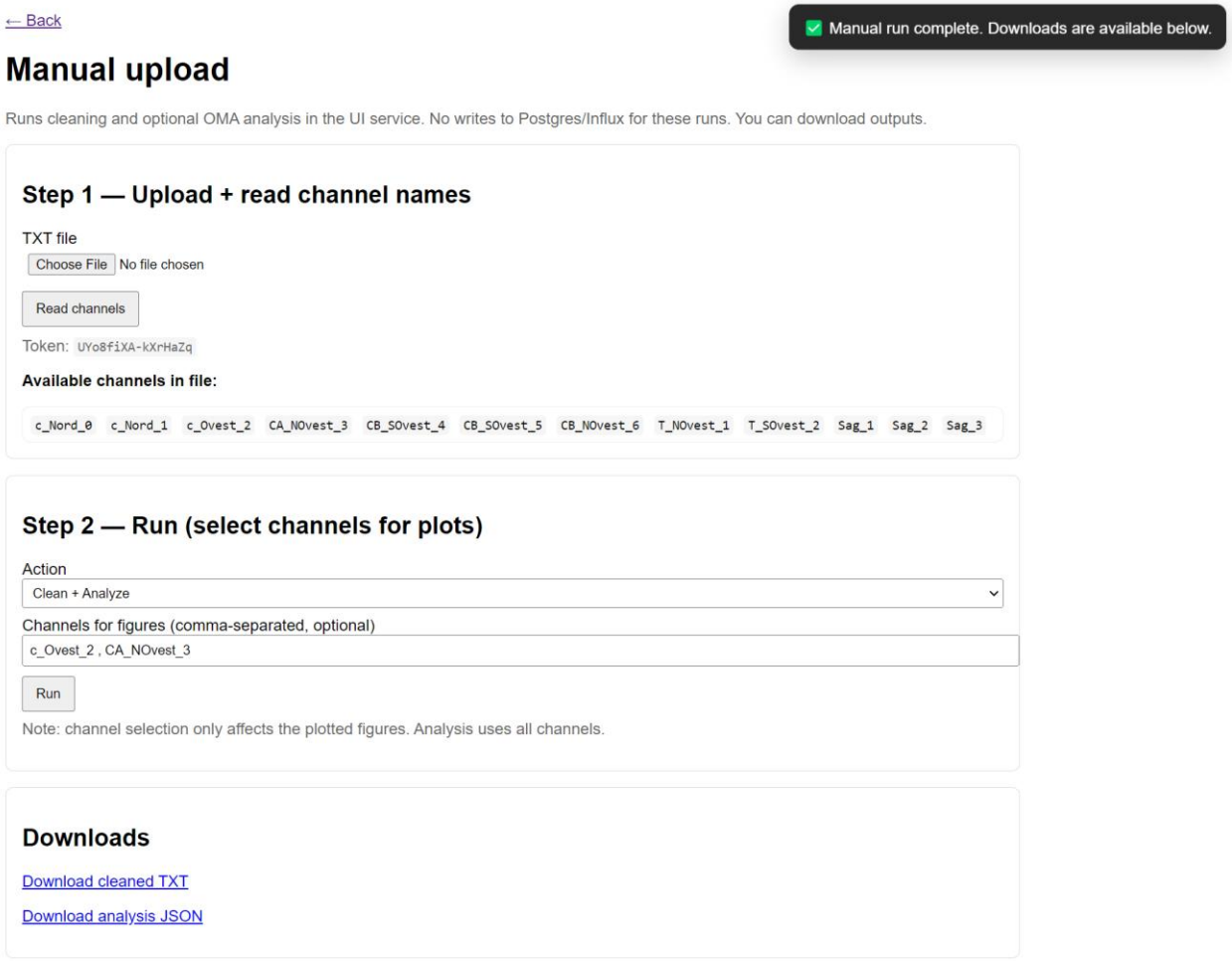


Figure 9.6 channel insertion method for running and output visuals

Figure 9.7 Analysis run output plots in the manual pipeline for the selected channels

Figure 9.8 Analysis output plots in the manual pipeline (shown by default for all channels when no channels are inserted)

Detected modes (preview)

Frequency (Hz)	Quality	Damping (rough)
0.6470	18.08	0.1132
0.6592	16.74	0.2222
0.7324	16.98	0.2333
0.8057	15.16	0.1970
1.0742	92.53	0.0341
1.0864	115.19	0.0225
1.2695	12.19	0.3462
1.9043	11.06	0.0256
2.3560	21.40	0.0104
9.7656	1476.63	0.0013
12.1094	7.60	0.0202
12.3291	24.32	0.0059

Figure 9.9 Mode detection using the OMA (FDD) method that produces rough damping

Figure 9.10 Plotly to create interactive visualizations plots.

10. Containerization and deployment

The platform is delivered as a Docker Compose application to support repeatable execution on a single workstation. This choice is not only a software-engineering convenience: in SHM prototypes, the ability to reproduce results depends on controlling library versions, database schemas, and service configurations over long monitoring campaigns. Containers provide an explicit and testable deployment boundary for each microservice, which aligns with common microservices practice [27].

In the current implementation, local first workstation is the deployment target. All services run on one host, and only the operator endpoints, the Management UI and Grafana, are exposed to the local network. The time series database, InfluxDB, and the relational database, PostgreSQL, are bound to localhost by default to lower the risk of accidental exposure. This limitation matches the thesis goal, which is reproducibility and easy setup for evaluation, and not deployment on the public internet.

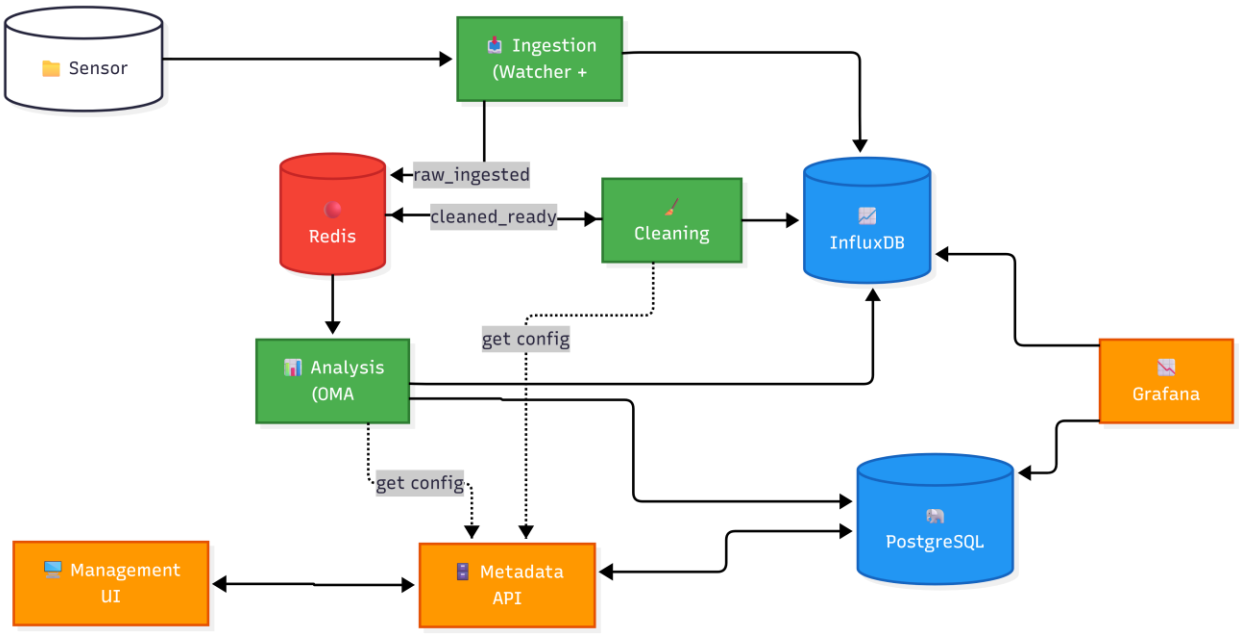


Figure 10.1 General System Overview

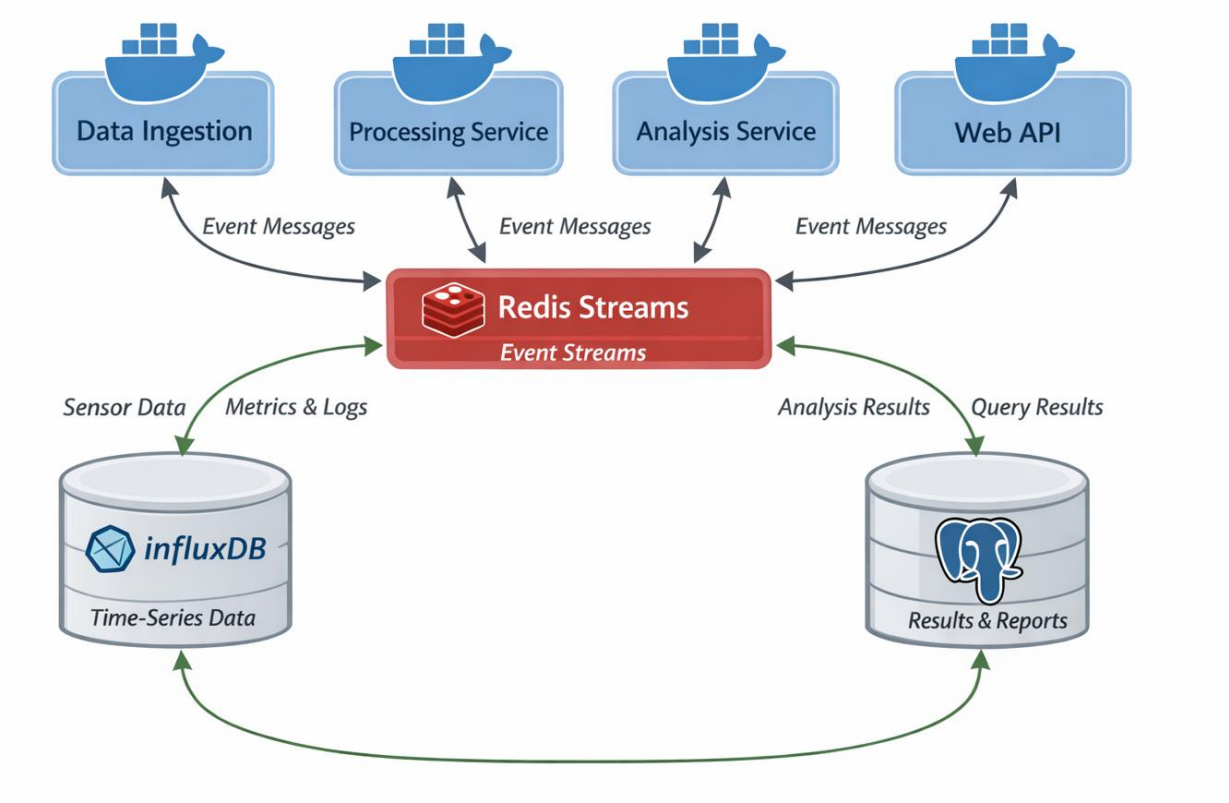


Figure 10.2 Containerized System Overview

10.1. Why containerization matters for SHM prototypes

Containerization addresses three recurring issues in SHM software prototypes. First, numerical processing often depends on compiled scientific libraries and specific Python package versions. Second, the platform combines heterogeneous infrastructure components (databases, dashboards, web services), which are error-prone to install and align manually. Third, operational handover to another engineer or laboratory typically fails when setup steps are undocumented or environment-specific. Docker reduces these sources of variability by specifying build and runtime requirements in version-controlled files [28].

Another benefit is that Docker Compose makes the service order clear and also checks if services are healthy. For example, the ingestion service should not start until InfluxDB is ready and it can be reached. Also, the Management UI should not start until the Metadata API and the Cleaning API are healthy. This makes the startup more deterministic, and it reduces race condition failures that often happen during demonstrations.

10.2. Docker architecture

The Compose stack consists of three infrastructure services and five application services. The infrastructure layer includes PostgreSQL for metadata storage, InfluxDB for time-series data storage, and Redis as an event bus. The application layer includes the Metadata API, Ingestion API (plus a watcher process), Cleaning API (plus a worker), the Analysis worker, and the Management UI. Grafana is deployed as an additional visualization service.

10.2.1. Docker Port Mappings

The port configuration is defined in the docker-compose.yml file:

Table 2 Example Docker port mapping in Docker Compose

Service	Internal Port	Published Port	Bind Address	Access Type
Management UI	8020	**18020**	0.0.0.0 (all interfaces)	**LAN accessible**
Grafana	3000	**3001**	0.0.0.0 (all interfaces)	**LAN accessible**
InfluxDB	8086	**8086**	127.0.0.1 (localhost)	**Local only**
PostgreSQL	5432	**15432**	127.0.0.1 (localhost)	**Local only**
Redis	6379	*(none)*	Internal network only	**Docker internal**
Metadata API	8001	*(none)*	Internal network only	**Docker internal**
Ingestion API	8010	*(none)*	Internal network only	**Docker internal**
Cleaning API	8030	*(none)*	Internal network only	**Docker internal**

10.2.2. How to change port mappings, in the docker-compose.yml:

Format: "`<bind_address>:<host_port>:<container_port>`"

Examples:

- **To change Management UI to port 8080:**
  - Change "18020:8020" to "8080:8020"
- **To make InfluxDB accessible on LAN:**
  - Change "127.0.0.1:8086:8086" to "0.0.0.0:8086:8086"
- **To restrict Grafana to localhost:**
  - Change "0.0.0.0:3001:3000" to "127.0.0.1:3001:3000"
- **To change PostgreSQL port:**
  - Change "127.0.0.1:15432:5432" to "127.0.0.1:5433:5432"

10.3. docker-compose orchestration

Docker Compose is used to orchestrate startup order, networking, and persistence. Persistent volumes are configured for PostgreSQL, InfluxDB, and Grafana to keep data available across container restarts. Environment variables are loaded from a single .env file, which centralizes parameters such as bucket names and service URLs, reducing configuration drift across services.

Health checks and depends_on conditions give a practical gate to make sure services are ready. For example, the ingestion service depends on InfluxDB and Redis, while the Management UI depends on the Metadata and Cleaning APIs. This matters a lot on Windows hosts, because file system mounting and the first service startup can add delays that are not always the same.

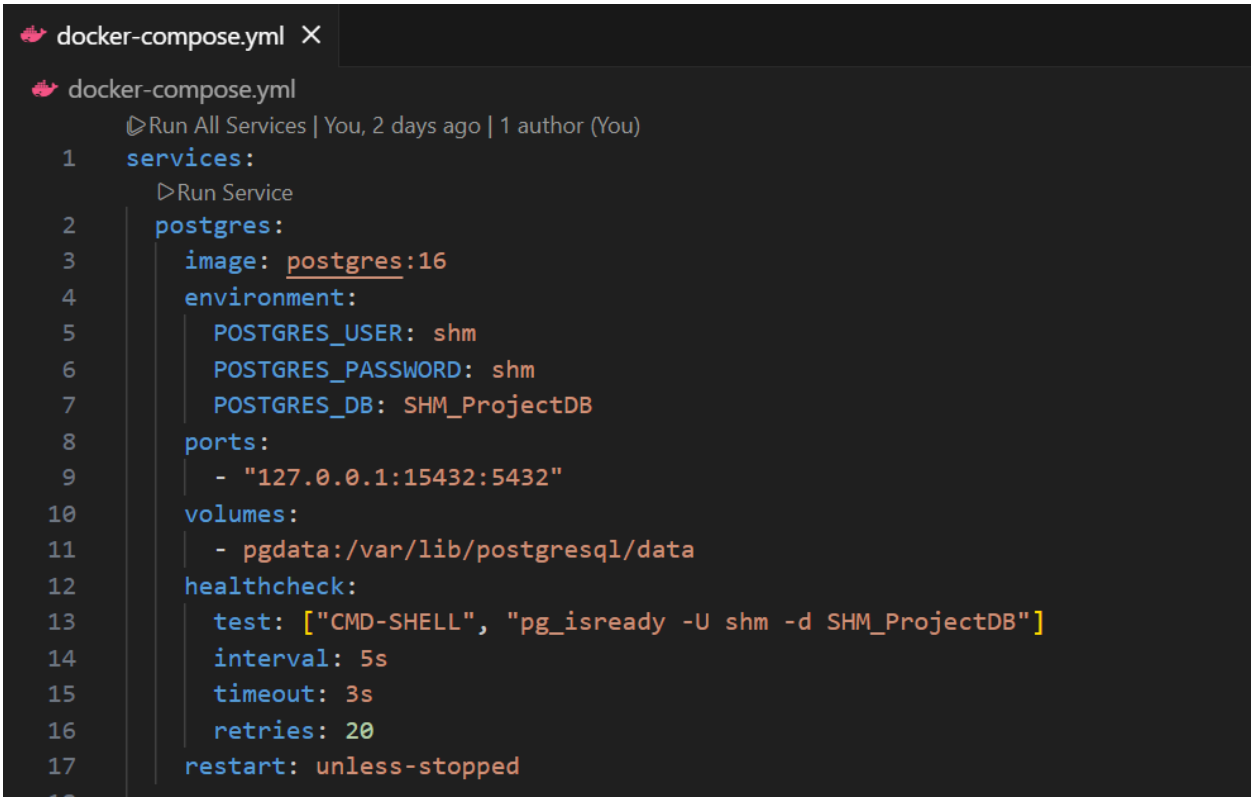


Figure 10.3 Excerpt of docker-compose.yml

In the current implementation, the ingestion “hot folder” is mounted at /StructuralX_Data/Data/incoming on both the ingestion service and the watcher container. A key operational detail is that the watcher must detect completed file copies rather than

partial writes. The current implementation uses a stability check (unchanged file size for a short interval) before triggering ingestion, which reduces parse failures on large files.

10.3.1. Password settings and changes instructions

To update passwords, secrets, or keys in the system, modify them in two places and ensure both files stay synchronized.

1. **.env file (Application-level credentials)**

Table 3 Variables and credentials to change in the .env file

Credential Type	Variable Name	Current Default	Purpose
Management UI Login	UI_USERNAME	admin	Web interface username
	UI_PASSWORD	changeme	Web interface password
PostgreSQL Connection	DATABASE_URL	postgresql+psycopg2://shm:shm@...	Database credentials (embedded in URL)
InfluxDB Token	INFLUX_TOKEN	super-secret-token	Time-series database authentication

Example changes in .env:

```
UI_USERNAME=myuser

UI_PASSWORD=MySecureP@ssw0rd!

DATABASE_URL=postgresql+psycopg2://shm:newpassword@postgres:5432/SHM_ProjectDB

INFLUX_TOKEN=my-very-long-secure-token-12345
```

2. **docker-compose.yml (Infrastructure-level credentials)**

Edit docker-compose.yml to change credentials for the database containers themselves:

```
POSTGRES_USER: shm
POSTGRES_PASSWORD: shm           # ← Change this
POSTGRES_DB: SHM_ProjectDB


DOCKER_INFLUXDB_INIT_USERNAME: admin      # ← Change this
DOCKER_INFLUXDB_INIT_PASSWORD: adminadminadmin  # ← Change this
DOCKER_INFLUXDB_INIT_ADMIN_TOKEN: super-secret-token  # ← Change this
```

3. **Synchronisation**

If credentials in docker-compose.yml are changed, the matching values in .env must update:

PostgreSQL password in docker-compose.yml → Update DATABASE_URL in .env

InfluxDB token in docker-compose.yml → Update INFLUX_TOKEN in .env

10.4. Deployment profiles

Although the thesis version targets local-first execution, it is useful to describe three deployment profiles. A development profile prioritizes rapid iteration (frequent rebuilds, verbose logs). A demonstration profile prioritizes deterministic startup and stability (health checks and persistent volumes). A future production profile would require authentication, encrypted transport, hardened database exposure rules, and operational monitoring. The current Compose file represents a demonstration-oriented profile, while the internal service boundaries are compatible with a later split across multiple hosts if required [27], [21].

11. Discussion, limitations, and future work

11.1. Validation summary

To assess whether the platform behaves predictably in daily operation, a set of functional and performance checks was executed on representative monitoring exports. The purpose was to verify the repeatability and traceability of the end-to-end workflow rather than to claim certified damage detection performance on a real structure. Evidence was collected from service logs, PostgreSQL run registries, and time-series queries in InfluxDB.

Upload Logs

upload_id	filename	status	sensor_count ↑	file_size_...	points_written	upload_timestamp	file_start_time	file_end_time	duration_seconds
2	2025_10_26_	success	12	15.1	1573488	2026-02-03 21:13:25	2025-10-26 06:04:07	2025-10-26 06:25:58	24.1
1	2025_10_26_	success	12	15.1	1573488	2026-02-03 21:08:31	2025-10-26 05:04:05	2025-10-26 05:25:56	30.0

Figure 11.1 Validation dataset summary (file name and size, channel count, upload timestamps, notes, ...)

unctional validation focused on stage correctness. For each file dropped into the hot folder, the expected artifacts were: a new upload record, a completed cleaning run with parameter snapshot, and a completed analysis run with modal outputs written to the OMA bucket. This check is intentionally mechanical. It ensures that failures in ingestion, preprocessing, or analysis are observable and that reruns do not silently create inconsistent states.

Performance was measured using simple timing for each stage. The end to end latency was split into parts for watcher, ingestion, cleaning, analysis, and persistence. Storage growth was estimated from sampling rate, channel count, and window duration, and then compared with the size of the stored modal feature layer.

End-to-end latency decomposition:

$$T_{tot} = T_{watch} + T_{ingest} + T_{clean} + T_{oma} + T_{persist} \quad \text{Eq.(11.1)}$$

Raw time series storage estimate:

$$S_{raw} \approx N_{ch}.f_S.T.b_{ps}(bytes), where\ b_{ps}\ is\ bytes\ per\ sample \quad \text{Eq.(11.2)}$$

Data reduction ratio (raw versus modal features):

$$R = \frac{S_{raw}}{S_{feat}}\left(or\ equivalently\ \rho = \frac{S_{feat}}{S_{raw}} as\ a\ compression\ fraction\right) \quad \text{Eq.(11.3)}$$

This chapter discusses what the current implementation platform achieves, its limitations, and how it could be extended to a more complete SHM system. The discussion is framed around the thesis scope: a local-first, microservices-based data platform that implements a robust preprocessing and OMA-based analysis pipeline, rather than a full structural diagnosis system.

11.2. What the current platform does well

The primary strength is automation with traceability. The system transforms ad hoc file handling into a repeatable workflow with explicit stages and run logs. For each uploaded file, it is possible to recover which cleaning parameters were used, which analysis parameters were active, and which outputs were written. This directly supports auditability, which is an important requirement for operational SHM, where indicators may influence maintenance decisions [1].

A second strength is the separation of concerns. By isolating ingestion, cleaning, analysis, and visualization into separate services, the platform makes failures easier to detect and localize. Redis Streams provide a simple and effective pipeline mechanism for decoupling stage execution and for enabling retries without tight service coupling. This architecture is consistent with general microservices guidance, where independently deployable units and explicit interfaces reduce operational risk [27].

Finally, a local first Compose deployment fits the thesis context well. Evaluators can reproduce the whole system with one command, and they do not need to manage external cloud credentials or distributed infrastructure. For research prototypes, this type of reproducibility is often more useful than theoretical scalability claims.

11.3. Key limitations

The analysis remains lightweight and does not include full stabilization-diagram processing. While FDD provides a practical output-only method for extracting modes from ambient vibration data, it is sensitive to spectral resolution, peak proximity, and excitation conditions. The current implementation mitigates some mode-swapping risks through MAC-based association, but it does not perform the full physical-mode selection process typically used in automated SSI pipelines [7].

A second limitation is how the system deals with environmental and operational variability. In principle, the platform can record temperature or other slow measurements as extra streams, but the current thesis version does not include clear compensation or normalization models. This is important because EOV can change the features in a way that looks like damage effects, and robust monitoring often needs clear methods to separate these influences [3].

In practice, the platform still does not provide a complete workflow for alarm delivery and acknowledgement. The event stream shm_analysis_done is a clean point for integration, but a production system would need authenticated user management, notification mechanisms, and an audit trail for operator decisions.

11.4. Future work roadmap

Future work can be grouped into three layers. At the analysis layer, a clear next step is to add SSI-based identification with stabilization diagrams and clustering, matching long-term monitoring practice in monumental structures. This would allow more robust separation of physical and numerical modes and support a richer set of quality indicators. The current storage model (ModeID tracking, MAC, per-run logs) is compatible with this extension.

At the decision layer, threshold selection could be more statistically clear. For example, extreme value based thresholds can control false alarm probability better for heavy tailed indices. Also, run rules can be adapted to the autocorrelated nature of vibration derived features. Still, these improvements should keep interpretability and traceability for engineering users.

At the platform layer, two improvements are needed. First, security should be stronger, including authentication, authorization, and transport security. Second, observability should be better, with centralized logs, health dashboards, and clearer error reporting. If the project goes beyond one workstation, container orchestration and distributed processing could be explored, but this scaling should be justified by monitoring volume and operational constraints, not by architectural fashion.

The following Table 4 shows a future road map:

Table 4 Future work mapping

Layer	Proposed Feature	Motivation	Required Code Changes	Expected Risk / Benefit
Analysis	SSI with stabilization diagrams	More robust modal separation and quality indicators	Add SSI to shm_shared/oma.py; extend mode metadata; update analysis worker	**Benefit:** Higher identification reliability **Risk:** Low
Decision	Extreme-value-based thresholds	Statistical control of false alarms	Add threshold module; extend config tables; update decision logic	**Benefit:** Better alarm control **Risk:** Medium
Decision	Run rules for autocorrelated features	Reduce spurious alerts	Implement run rules; update alert logic	**Benefit:** Fewer false positives **Risk:** Low
Platform	Security hardening	Production-ready deployment	Add auth, TLS, and role control	**Benefit:** High **Risk:** Medium
Platform	Improved observability	Easier debugging and monitoring	Centralized logs and health dashboards	**Benefit:** High **Risk:** Low
Platform	Orchestration and scaling	Support higher monitoring volumes	Add orchestration and distributed tasks	**Benefit:** High if needed. Risk**:** High

The proposed roadmap is intentionally incremental. Analysis- and decision-layer extensions build on the existing data model and processing pipeline and can be introduced without structural changes to the platform. Platform-level enhancements, particularly orchestration and distributed execution, are considered conditional and should be driven by actual monitoring scale and operational requirements rather than by architectural preference alone.

References

[1] C. R. F. a. K. Worden, Structural Health Monitoring: A Machine Learning Perspective, Chichester, U.K.: John Wiley & Sons, 2013.

[2] O. E. A. M. G. M. Malekloo A, "Machine learning and structural health monitoring overview with emerging technology and high-dimensional data source highlights," *Structural Health Monitoring,* vol. 21, no. 4, pp. 1906-1955, 2021.

[3] H. Sohn, "Effects of environmental and operational variability on structural health monitoring," *Philos Trans A Math Phys Eng Sci,* vol. 365, no. 1853, p. 539–560, 2007.

[4] R. Epicoco, "Identificazione modale automatica di tipo Stochastic Subspace per il monitoraggio di grandi edifici monumentali = Automatic modal identification with Stochastic Subspace for monumental structures monitoring," Politecnico di Torino, Turin, Italy, 2022.

[5] C. Facelli, "Cupola Vicoforte," 2024. [Online]. Available: https://www.cascinafacelli.com/wp-content/uploads/2024/12/panorama-santuario-vicoforte-mondovi.jpg. [Accessed Jan. 2026].

[6] Q. Magazine, "Reducing Seismic Risk for an Ancient Roman Amphitheater," 11 Oct. 2012. [Online]. Available: https://www.qualitymag.com/articles/90796-reducing-seismic-risk-for-an-ancient-roman-amphitheater. [Accessed Jan. 2026].

[7] M. L. Pecorelli, R. Ceravolo, G. De Lucia and R. Epicoco, "A vibration-based health monitoring program for a large and seismically vulnerable masonry dome," in *Proc. 12th Int. Conf. on Damage Assessment of Structures (DAMAS)*, Kitakyushu, Japan, 2017.

[8] S. Coccimiglio, G. Miraglia, G. Coletta, R. Epicoco and R. Ceravolo, "Balanced Definition of Thresholds for Mode Tracking in a Long-Term Seismic Monitoring System," *Geosciences,* 2023.

[9] C. R. Farrar and K. Worden, "An introduction to structural health monitoring," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences,* vol. 365, no. 1851, pp. 303-315, 2007.

[10] C. R. F. G. M. G. P. Keith Worden, "The fundamental axioms of structural health monitoring," *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences,* vol. 463, no. 2082, pp. 1639-1664, 2007.

[11] S. Gholizadeh, "Damage Analysis and Prediction in Glass Fiber Reinforced Polyester Composite Using Acoustic Emission and Machine Learning," *Research Square,* 2022.

[12] C. Casalegno, R. Ceravolo, M. A. Chiorino, M. L. Pecorelli and L. Zanotti Fragonara, "Soil-structure modeling and updating of the" Regina Montis Regalis" basilica at Vicoforte, Italy," in *9th International Conference on Structural Analysis of Historical Constructions (SAHC 2014)*, Mexico City, 2014.

[13] R. Ceravolo, M. L. Pecorelli, L. Zanotti Fragonara and A. De Marinis, "Monitoring of masonry historical constructions: ten years of static monitoring of the world's largest oval dome," *Structural Control and Health Monitoring,* vol. 24, no. 10, 2017.

[14] R. Ceravolo, G. De Lucia and M. L. Pecorelli, "Issues on the modal characterization of large monumental structures with complex dynamic interactions," *Procedia Engineering,* vol. 199, p. 3344–3349, 2017.

[15] F. J. Harris, "On the use of windows for harmonic analysis with the discrete Fourier transform," in *Proceedings of the IEEE*, 1978.

[16] A. V. S. R. W. Oppenheim, Discrete-Time Signal Processing, 3 ed., Upper Saddle River: Prentice Hall, 2009.

[17] P. Welch, "The use of fast Fourier transform for the estimation of power spectra: A method based on time averaging over short, modified periodograms," *IEEE Transactions on Audio and Electroacoustics,* vol. 15, no. 2, pp. 70-73, 1967.

[18] F. Magalhães, Á. Cunha and E. Caetano, "Vibration based structural health monitoring of an arch bridge: From automated OMA to damage detection," *Mechanical Systems and Signal Processing,* vol. 28, p. 212–228, 2012.

[19] S. Jeong, R. Hou, J. P. Lynch, H. Sohn and K. H. Law, "A scalable cloud-based cyberinfrastructure platform for bridge monitoring," *Structure and Infrastructure Engineering,* vol. 15, no. 1, pp. 82-102, 2019.

[20] M. S. a. A. Sadhu, "Visualization of structural health monitoring information using Internet-of-Things integrated with building information modeling," *Journal of Infrastructure Intelligence and Resilience,* vol. 2, no. 3, 2023.

[21] I. Docker, "Docker Compose documentation," 2026. [Online]. Available: https://docs.docker.com/compose/. [Accessed Jan 2026].

[22] B. Peeters, J. Maeck and G. De Roeck, "Vibration-based damage detection in civil engineering: excitation sources and temperature effects," *Smart Materials and Structures,* vol. Vol. 10, no. 3, pp. 518-527, 2001.

[23] L. Z. a. P. A. Rune Brincker, "Modal identification of output only systems using Frequency Domain Decomposition," *Smart Materials and Structures,* vol. 10, no. 3, 2001.

[24] G. Labs, "Grafana documentation," 2026. [Online]. Available: https://grafana.com/docs/. [Accessed Jan 2026].

[25] S. Ramírez, "FastAPI documentation," [Online]. Available: https://fastapi.tiangolo.com/. [Accessed Jan 2026].

[26] P. T. Inc., "Plotly Python documentation," [Online]. Available: https://plotly.com/python/. [Accessed Jan 2026].

[27] N. S.-L. A. M. F. R. L. Dragoni, "Microservices: Yesterday, Today, and Tomorrow," in *Present and Ulterior Software Engineering*, Cham, Switzerland, Springer, 2017, p. 195–216.

[28] D. Merkel, "Lightweight Linux Containers for Consistent Development and Deployment," *Linux Journal,* vol. 2014, no. 239, 2014.

Appendices

Full API endpoint list and example requests

This appendix documents the internal REST APIs exposed by the platform services. The APIs are designed for service-to-service communication within the containerized environment and for limited operator interaction through the Management UI. Unless otherwise stated, endpoints are not intended for direct public access and are reachable only within the Docker network.

Table 5 Service Overview

Service	Base URL	Intended Access
Metadata API	http://metadata_api:8001	Internal services
Ingestion API	http://ingestion_api:8010	Internal services
Cleaning API	http://cleaning_api:8030	Internal services
Management UI	http://localhost:18020	Local operator (Basic Auth)

A.1 Metadata API

The Metadata API manages persistent configuration, structural metadata, and run-level bookkeeping.

Structures

```
GET    /structures/
GET    /structures/by-name/{name}
POST   /structures/
```

```
DELETE /structures/{structure_id}
```

Sensors

```
GET  /sensors/?structure_id={id}
POST /sensors/
POST /sensors/upsert
```

Uploads

```
GET  /uploads/?skip=0&limit=100
GET  /uploads/{upload_id}
POST /uploads/
POST /uploads/{upload_id}/success
POST /uploads/{upload_id}/failed
```

Cleaning Runs

```
GET  /cleaning/runs
GET  /cleaning/runs/{cleaning_id}
GET  /cleaning/runs/upload/{upload_id}
POST /cleaning/runs
POST /cleaning/runs/{cleaning_id}/success
POST /cleaning/runs/{cleaning_id}/failed
```

Analysis Runs

```
GET  /analysis-runs/
GET  /analysis-runs/{run_id}
POST /analysis-runs/
POST /analysis-runs/{run_id}/success
POST /analysis-runs/{run_id}/failed
```

OMA Modes and Observations

```
GET  /oma/modes
POST /oma/modes
POST /oma/modes/{mode_id}/seen
GET  /oma/observations
```

```
POST /oma/observations
```

Configuration

```
GET /config/{service}

PUT /config/{service}
```

Configuration values are stored as strings and resolved at runtime by the consuming services.

A.2 Ingestion API

The Ingestion API registers new monitoring files and starts the processing pipeline. It offers these endpoints.

```
POST /ingest

POST /ingest-upload
```

A typical request involves providing a file path in the monitored hot folder. When ingestion succeeds, the raw data is stored in InfluxDB, a record of the upload is added to PostgreSQL, and a raw_ingested event is triggered.

A.3 Cleaning Service API

The Cleaning API offers a small set of endpoints to query parameters and to start cleaning..

```
GET  /params

POST /clean/upload/{upload_id}
```

The cleaning service reads raw data from InfluxDB, applies the configured preprocessing steps, writes the cleaned data back to the time series store, and then sends a cleaned_ready event.

A.4 Management UI Endpoints

The Management UI gives a browser interface for monitoring and manual experiments. Access is protected with HTTP Basic Authentication.

Key endpoints include:

```
GET  /                          (dashboard)

GET  /manual                    (manual processing)

POST /config/cleaning

POST /config/analysis

POST /manual/run

GET  /manual/download/{token}/{kind}
```

Manual processing is not saved, and it skips the automated pipeline.

A.5 Pipeline Events (Redis Streams)

Inter-service coordination is implemented using Redis Streams.

Table 6 Publisher-Consumer for Redis Streams

Stream	Publisher	Consumer
shm_raw_ingested	Ingestion API	Cleaning Worker
shm_cleaned_ready	Cleaning Worker	Analysis Worker
shm_analysis_done	Analysis Worker	User

Each event carries minimal metadata rather than a bulk data payload.

A.6 Operational Notes

- Configuration priority is: PostgreSQL values first, then environment variables, then defaults.
- Deleting a structure also deletes dependent entities.
- All timestamps are stored in UTC in ISO 8601 format.
- Internal services can be reached only inside the container network.
- Manual UI execution does not change the database.

Container sample logs for automatic pipeline (ingestion-cleaning-analysis) verification

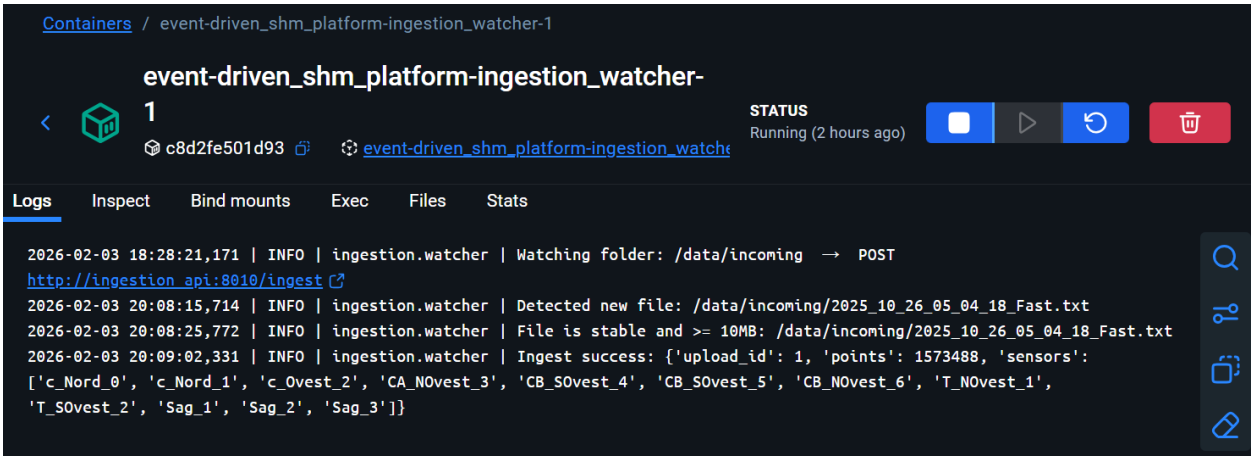


Figure 0.1 ingestion logs

Figure 0.2 cleaning logs

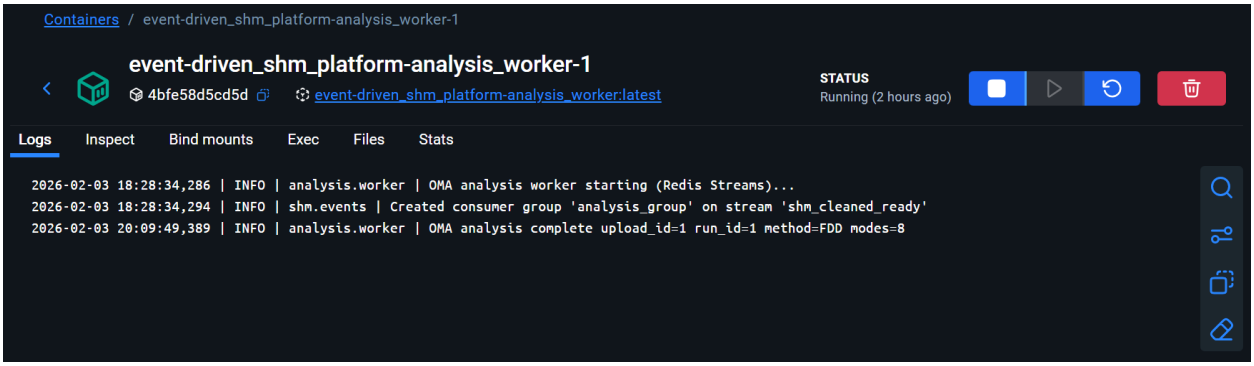


Figure 0.3 analysis logs