GeoScienceEd *7(4) (2026)*

Jurnal Pendidikan, Sains, Geologi, dan Geofisika

http://jpfis.unram.ac.id/index.php/GeoScienceEdu/index

Modular Software Architecture for a Raspberry Pi-Based Geophysical Data Acquisition System

Ilham Muthahhari[1*], Benyamin Heryanto Rusanto[1], Suko Prayitno Adi[1], Nardi[1], Muhamad Dzakwan Firdaus[1]

[1] Department of Instrumentation-MKG, Sekolah Tinggi Meteorologi Klimatologi dan Geofisika, Tangerang, Banten 15119, Indonesia.

DOI: https://doi.org/10.29303/goescienceed.v7i4.3217

Article Info:
Received : July, 20 2026
Revised : August, 07 2026
Accepted : August, 29 2026
Published : September, 20 2026

Correspondence:

Ilham Muthahhari

Phone:

Abstract: Continuous geophysical data acquisition requires software that can maintain sampling, timing, data storage, and system operation when individual components become unavailable. This paper describes a modular software architecture developed for a Raspberry Pi-based 24-bit geophysical data acquisition system. The application is divided into six modules covering system orchestration, hardware abstraction, signal processing, data persistence, REST services, and the web interface. External configuration, hybrid timestamping using GPS, a DS3231 real-time clock, and a monotonic counter are used together with deadline-based scheduling and layered fault handling. The software was evaluated through dependency analysis, startup measurement, induced fault tests, dashboard checks, and a one-hour acquisition test at 100 SPS. The system reached a record-ready state in 7.840 s and recorded 360,000 samples per channel during the one-hour test, with no dropped samples, MiniSEED gaps, or overlaps. Acquisition also continued during watchdog access denial, temporary database disconnection, GPS unavailability, and REST service interruption. The results show that a modular software structure can support continuous geophysical data acquisition on a general-purpose Linux platform without a real-time kernel.

Keywords: Geophysical Data Acquisition; MEMS Accelerometer; Raspberry Pi; Software Architecture; Fault Tolerance

Citation: Muthahhari, I., Rusanto, B. H., Adi, S. P., Nardi, & Firdaus, M. D. (2026). Modular Software Architecture for a Raspberry Pi-Based Geophysical Data Acquisition System. *Jurnal Pendidikan, Sains, Geologi, Dan Geofisika (GeoScienceEd Journal)*, *7*(4), 5598–1610. https://doi.org/10.29303/goescienceed.v7i4.3217

Introduction

Low-cost microelectromechanical-system (MEMS) accelerometers, high-resolution analog-to-digital converters, and single-board computers have made geophysical data acquisition systems more accessible for research and monitoring applications. Recent studies have demonstrated the use of Raspberry Pi, MEMS accelerometers, and low-cost acquisition platforms for structural vibration and seismic monitoring, showing that useful waveform records can be obtained with relatively accessible hardware (Alessandro et al., n.d.; Ali Bakir et al., 2023; Crognale et al., 2024; Nof et al., 2019; Özcebe et al., 2022; Özdemir & Kömeç Mutlu, 2024; Ramdeane & Lynch, 2020). For continuous operation, however, the hardware is only one part of the acquisition system. The software must also maintain sampling, timestamps, data storage, and communication while the instrument is running unattended.

These requirements introduce problems that are not always visible from the hardware specifications. A recording process must avoid cumulative timing errors, handle communication timeouts, maintain consistent timestamps, and prevent failures in auxiliary services from stopping the main acquisition path. For geophysical waveform data, an interrupted recording or an unnoticed time discontinuity can affect the usefulness of the resulting record even when the sensing hardware remains operational (Ramdeane & Lynch, 2020; Ringler et al., 2023). Several low-cost acquisition studies focus

Email: ilhammuthahhari@gmail.com

mainly on sensor performance, converter characteristics, noise, or comparison with reference instruments, while the acquisition software is often described only briefly as part of the implementation (Crognale et al., 2024; Knapp & Bloom, 2022; Özdemir & Kömeç Mutlu, 2024; Vlachos et al., 2025). However, these studies provide limited discussion of the software architecture as a separate engineering concern. In particular, there is limited practical evaluation of how modular software components can coordinate sampling, timing, processing, storage, and auxiliary services while keeping failures in non-critical components from interrupting acquisition. This gap motivates the present study, which focuses specifically on the design and evaluation of a modular software architecture for continuous low-cost geophysical data acquisition.

Open-source platforms provide a practical basis for developing such systems. Ramdeane and Lynch (2020) demonstrated a low-cost seismic acquisition approach based on open-source hardware and software, while Knapp and Bloom (2022) showed the use of Raspberry Pi for high-resolution scientific data acquisition. Others have reported that the continued use of 24-bit converters in low-cost geophysical instrumentation (Pratama & Suharsono, 2025). These studies support the use of accessible hardware for research instrumentation, but the present work focuses on a different aspect: the organisation of the software that controls the acquisition process.

In this study, the software architecture was developed for a Raspberry Pi 3 Model B+-based 24-bit acquisition system using an ADXL335 MEMS accelerometer, ADS1256 delta-sigma converter, DS3231 real-time clock, and u-blox NEO-7M GPS receiver. The software is divided into modules for orchestration, hardware communication, signal processing, data persistence, REST services, and the web interface. The architecture was designed around three practical requirements: maintaining a 100 SPS acquisition rate, allowing acquisition to continue when non-essential components fail, and keeping changes such as replacing a converter or adding an output format localised within the software structure. The novelty of this work lies in treating the software architecture itself as the main object of evaluation. Rather than focusing on sensor or converter performance, the study evaluates how modular responsibility separation, hybrid timestamping, deadline-based scheduling, independent data persistence, and failure isolation can support continuous acquisition on a low-cost Linux platform. These aspects are evaluated through timing, fault-injection, dependency, and maintainability. The present work therefore evaluates the software architecture and acquisition workflow rather than sensor accuracy or comparison with reference instrumentation.

The main objective is to examine whether a responsibility-based software structure can support continuous geophysical waveform acquisition on a general-purpose Linux platform. The evaluation considers module dependencies, startup behaviour, timestamping, acquisition timing, data persistence, and responses to induced hardware and software failures. The resulting architecture is intended as a practical software structure for low-cost geophysical instrumentation, while measurement accuracy and long-term field performance remain outside the scope of this study.

Method

System and Design Constraints

The work followed a design-and-build approach in which the software architecture was the main artefact and the digitizer provided the execution environment. The system was implemented on a Raspberry Pi 3 Model B+ running Raspberry Pi OS, a general-purpose Linux platform without a real-time kernel. Python was used for the acquisition and backend software because the system relies on scientific and seismological libraries available in the Python ecosystem.

The acquisition target was 100 samples per second (SPS) for three accelerometer channels, corresponding to a nominal 10 ms sampling interval. The software communicates with the ADS1256 converter through SPI, the DS3231 real-time clock through I²C, the NEO-7M GPS receiver through UART, and the Linux watchdog through its device interface. The instrument was designed for unattended operation and was required to preserve raw measurements, generate MiniSEED waveform files, provide operational status information, and continue acquisition when non-essential services became unavailable. The design therefore considered three main constraints. First, the acquisition loop had to maintain the 100 SPS target without allowing processing delays to accumulate over time. Second, failures in supporting components such as the GPS, database, watchdog, or REST service should not stop the measurement path. The ADC was treated differently because it is required to obtain samples. Third, configuration and software changes should remain local to the module responsible for the affected function.

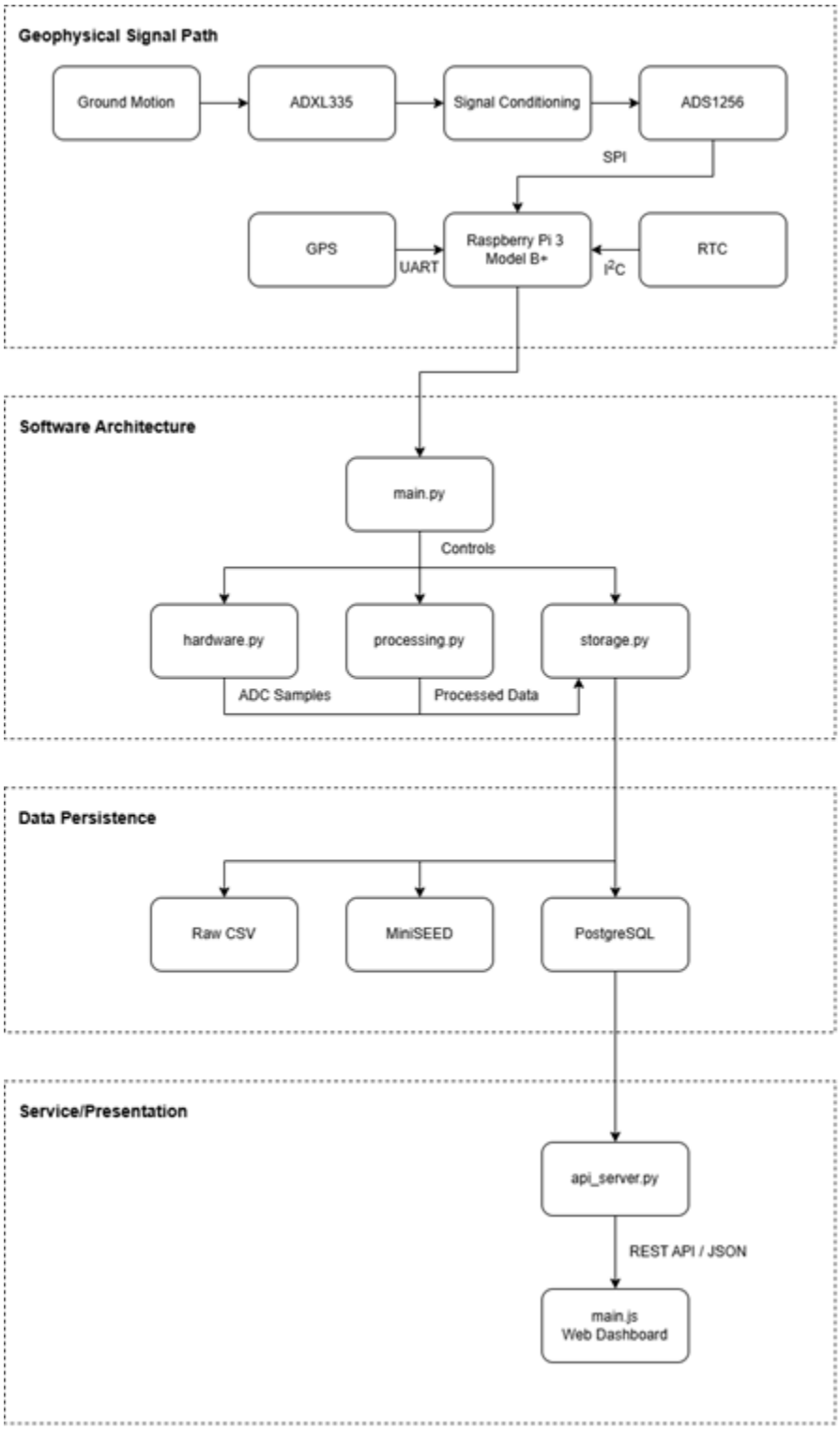

Figure 1. Geophysical signal path and modular software architecture

Architectural Decomposition

The software was divided according to responsibility rather than execution order. Six principal modules were used, organised into four functional layers, while main.py serves as the orchestration root. The four layers cover hardware abstraction, signal processing, persistence, and service/presentation. Dependencies are intended to flow from the orchestration layer toward the functional modules without cyclic imports.

Table 1. Software module, architectural layer, and primary responsibility

Module	Logical Placement	Primary Responsibility
main.py	Orchestration root	Loads configuration, performs startup, composes modules, and runs the acquisition loop
hardware.py	Hardware abstraction	Handles ADC, RTC, GPS, and watchdog interfaces
processing.py	Processing layer	Converts voltage to acceleration, applies filtering and deadband, and maintains statistics
storage.py	Persistence layer	Writes PostgreSQL records, CSV files, and MiniSEED waveforms
backend/api_server.py	Service/presentation layer	Provides REST endpoints and on-demand analysis
frontend/src/main.js	Service/presentation layer	Provides the browser dashboard, plots, maps, downloads, and connection status

The hardware layer isolates device-specific communication from the rest of the application. The processing layer handles measurement conversion and statistics, while the persistence layer manages the different output destinations. The service and presentation layers provide access to stored data without becoming part of the sampling path. This arrangement allows the acquisition process to operate independently from the web interface.

Runtime and Acquisition Process

The application uses two independent processes. The acquisition process loads the configuration, initialises the hardware and storage components, performs the required checks and calibration, and then starts continuous sampling. A separate FastAPI process provides the REST interface and dashboard data. The two processes communicate through stored data products rather than direct function calls, so restarting the web service does not require restarting the acquisition process. During continuous operation, the acquisition loop is divided according to the frequency of each task. ADC reading, timestamp assignment, signal processing, raw-data buffering, watchdog servicing, and sample counters are performed for each sample. Statistical calculations, database insertion, and MiniSEED block assembly are performed every 100 samples, or approximately once per second. The absolute time reference is refreshed from the RTC every 60 s. This arrangement keeps relatively expensive operations outside the most timing-sensitive part of the loop. The sampling loop uses an absolute deadline rather than sleeping for a fixed duration after each iteration. The next deadline is advanced by 0.01 s for every sample. When an iteration takes longer than expected, the following deadline is still based on the original schedule rather than on the completion time of the previous iteration. This prevents delays from accumulating as a persistent phase shift.

Timestamping and Synchronization

Timestamp generation separates the absolute time reference from short-term interval measurement. Reading the DS3231 for every sample would introduce repeated I²C transactions into the 10 ms acquisition interval. Instead, the software stores the latest RTC time together with the corresponding value of Python's perf_counter_ns() at each 60 s refresh. For each sample, the timestamp is obtained by adding the elapsed monotonic-counter interval to the latest RTC reference. When available, UTC information from the GPS NMEA $GPRMC sentence is used during startup to initialise or calibrate the RTC. This approach reduces repeated RTC access while avoiding dependence on the adjustable system clock for inter-sample timing.

Evaluation Procedure

The evaluation focused on the software architecture and acquisition workflow rather than sensor measurement accuracy. Five aspects were examined: module dependencies, startup behaviour, runtime acquisition, fault handling, and data continuity. First, the application import structure was inspected to identify cyclic dependencies and upward dependencies between modules. Second, startup time was obtained from timestamped application logs, from process initiation until the acquisition loop entered the recording state. Third, runtime behaviour was monitored using the system's per-second health statistics and the resulting waveform files.

Fault handling was evaluated under five conditions: watchdog access denial, an unavailable ADC during startup, temporary PostgreSQL interruption, absence of a GPS position fix, and termination of the REST service process. The response of the acquisition process was recorded for each condition. A one-hour acquisition run at the configured 100 SPS rate was then used to check sample count, dropped-sample counters, ADC timeout counters, and MiniSEED gaps or overlaps. The dashboard was checked using a functional checklist, while generated MiniSEED files were opened using ObsPy and PQL II to verify file accessibility and interoperability. These checks were treated as software and data-format verification rather than as sensor-performance validation.

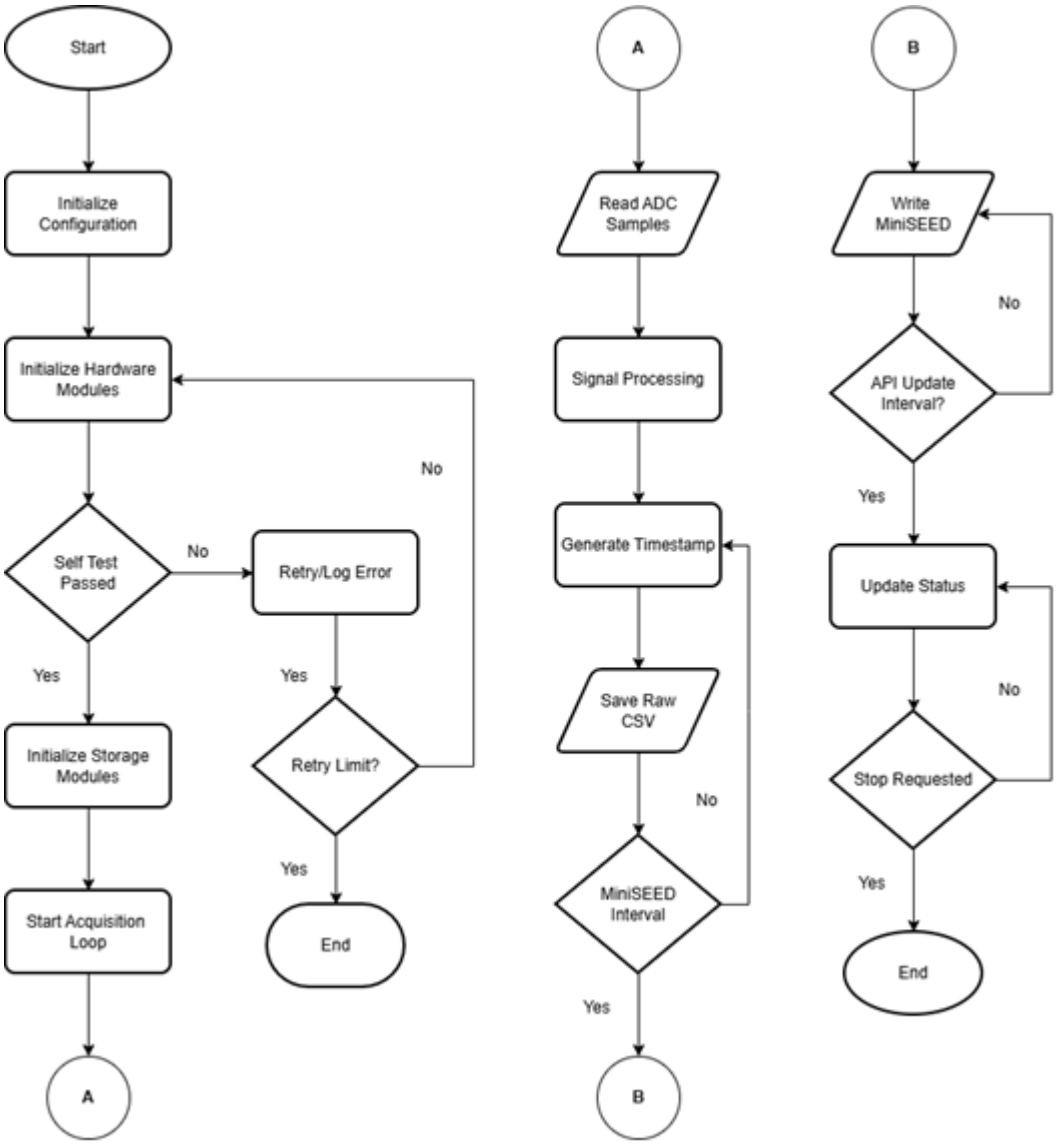

Figure 2. Runtime workflow of the acquisition process

Result and Discussion

Realised Structure and Code Organisation

Figure 1 shows the implemented software structure and its relationship with the geophysical signal path. Ground motion is converted into an analog signal by the ADXL335, conditioned, digitized by the ADS1256, and transferred to the Raspberry Pi. From this point, the software separates hardware communication, processing, data persistence, service functions, and presentation. This arrangement places software boundaries close to the corresponding physical interfaces. As a result, a change to a device interface can be handled mainly within the corresponding hardware module rather than throughout the acquisition program.

The same separation is reflected in the project directory. The six principal source modules are kept at the project root together with the configuration file and dependency manifest, while the backend, frontend, data, and log directories are separated into their respective subdirectories. The ADS1256 vendor driver is not accessed directly by the acquisition logic; communication with the driver is handled through hardware.py. This keeps device-specific code within the hardware abstraction layer. The implemented modules and their responsibilities are summarised in Table 1. main.py acts as the composition and orchestration root, while the remaining modules handle specific functions. The processing and storage modules do not depend on the web interface, and the service process accesses data that has already been produced by the acquisition process. This structure keeps dashboard and service activity outside the sampling path.

Configuration and Startup

Operational parameters that may change between deployments are loaded from an external .env file into a shared configuration object. These parameters include station and channel identity, target sampling rate, ADC channel mapping and timeout, RTC refresh interval, GPS wait limit, database settings, output directories, buffer size, processing parameters, sensor sensitivity, and API settings. The acquisition logic therefore does not need to be edited when these values are changed.

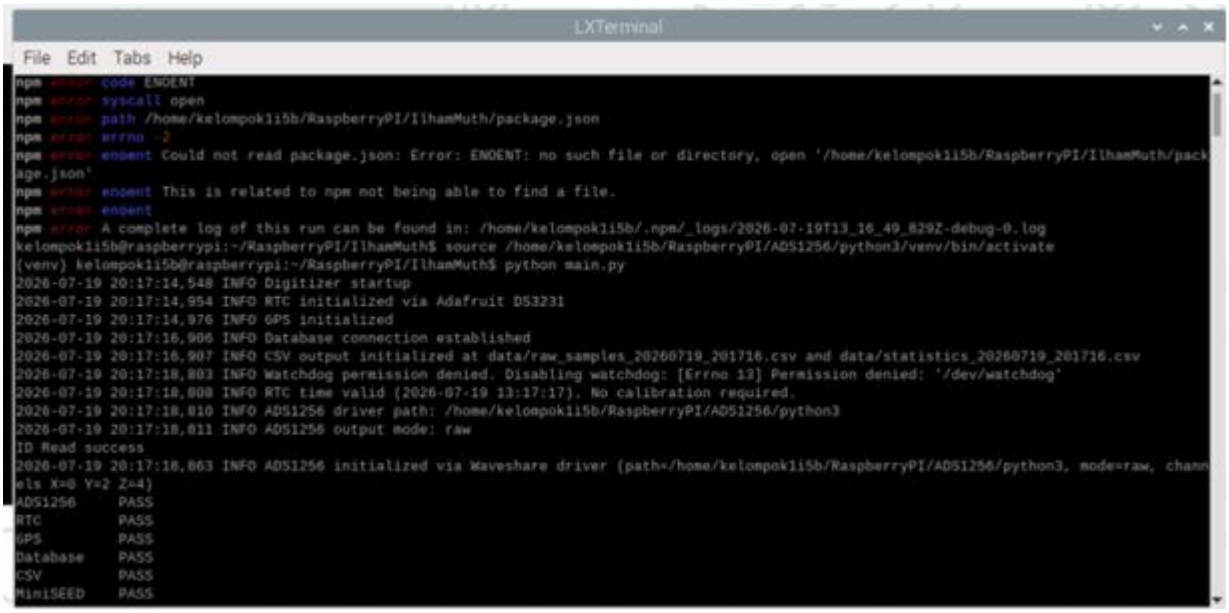

Figure 3. Instrumented startup log showing staged initialisation, component self-test results, and watchdog permission degradation

The startup process is staged rather than performed as a single initialisation block. Figure 3 shows an actual startup sequence in which the RTC, GPS, database, CSV writers, watchdog, MiniSEED writer, and ADC are initialised before offset calibration and entry into the acquisition loop. The status of each component is reported separately, which makes it possible to distinguish an unavailable supporting service from a failure that prevents acquisition.

Table 2 gives the measured duration of each startup stage. The complete sequence required 7.840 s from process start until the acquisition loop reported that it was running. The database connection took 1.930 s, making it the longest connection-related stage. Offset calibration required 3.311 s, which was the longest individual stage overall because the procedure acquired 1,000 samples before calculating the offset and noise characteristics. Local peripheral initialisation was considerably shorter. The startup duration was not further optimised because the instrument is intended for long continuous recording periods. More importantly, the staged sequence provides information about where startup is delayed or interrupted. This is useful for an unattended instrument because a failure can be associated with a specific component rather than being reported only as a general application failure.

Table 2. Measured startup sequence timing derived from consecutive application log timestamps

Stage	Completed at	Duration (s)
Process start	20:17:14.548	—
RTC initialization (DS3231, I²C)	20:17:14.954	0.406
GPS initialization (UART)	20:17:14.976	0.022
Database connection established	20:17:16.906	1.930
CSV writers initialized	20:17:16.907	0.001
Watchdog attempt and MiniSEED setup	20:17:18.803	1.896
RTC time validation	20:17:18.808	0.005
ADC driver initialization (SPI)	20:17:18.863	0.055
Offset calibration (1000 samples)	20:17:22.174	3.311
Acquisition loop running	20:17:22.174	0.214
Total time to record-ready	—	7.840

Hardware Abstraction and Device Handling

The hardware.py module contains the interfaces for the ADS1256 ADC, DS3231 RTC, NEO-7M GPS receiver, and Linux watchdog. Device-specific communication and errors are handled within this module. The rest of the application therefore receives higher-level results rather than directly managing SPI, I²C, UART, or watchdog operations.

The separation also allows different functions of the same device to be handled independently. GPS time obtained from the $GPRMC sentence can be available before a valid position fix is obtained. Consequently, the absence of a GPS position does not prevent the system from using the available time information for timestamp initialisation. During the observed test, GPS position availability therefore affected metadata rather than the waveform acquisition path.The ADC is treated as an essential component because no measurement can be obtained when it is unavailable. Its failure therefore produces a different response from failures in supporting components.

Acquisition Timing and Timestamp Behaviour

The acquisition loop uses three practical execution rates. ADC reading, timestamp assignment, signal processing, raw-data buffering, watchdog servicing, and sample counters are handled for each sample. Statistical calculations, database insertion, and MiniSEED block assembly are performed every 100 samples, approximately once per second. The RTC time base is refreshed every 60 s. This arrangement limits the amount of work performed during the 10 ms sampling interval.

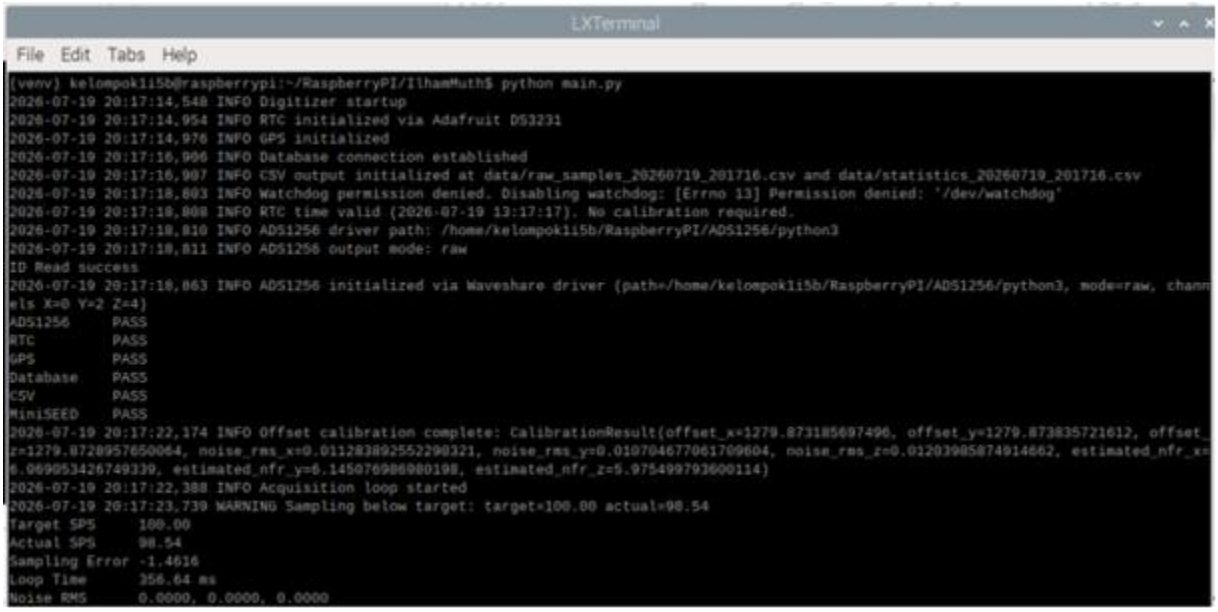

Figure 4. Completion of offset calibration and entry into the acquisition loop, including the elevated execution time of the first iteration

The sampling loop uses an absolute deadline. After each iteration, the next deadline is advanced by 0,01 s instead of being calculated from the time at which the previous iteration finishes. This distinction becomes visible during the first acquisition iteration. Figure 4 shows a loop duration of 356,64 ms and a reported rate of 98,54 SPS during the initial transient. This delay is associated with first-use operations such as file creation, buffer allocation, and code paths that are executed for the first time.

The first-iteration delay did not produce a persistent sampling offset. After the transient, the loop returned to its normal schedule. Under a relative-delay approach, the execution time of an iteration would be added to the following sleep interval and could accumulate as a long-term timing error. With the absolute-deadline approach, a late iteration does not redefine the subsequent schedule. Figure 5 and Table 3 show the steady-state behaviour. The loop execution time settled at 11,87 ms, while the software-reported acquisition rate varied between 99,30 and 99,55 SPS. The corresponding deviation from the 100 SPS target was –0,45% to –0,70% . Both dropped samples and ADC read timeouts remained at zero.

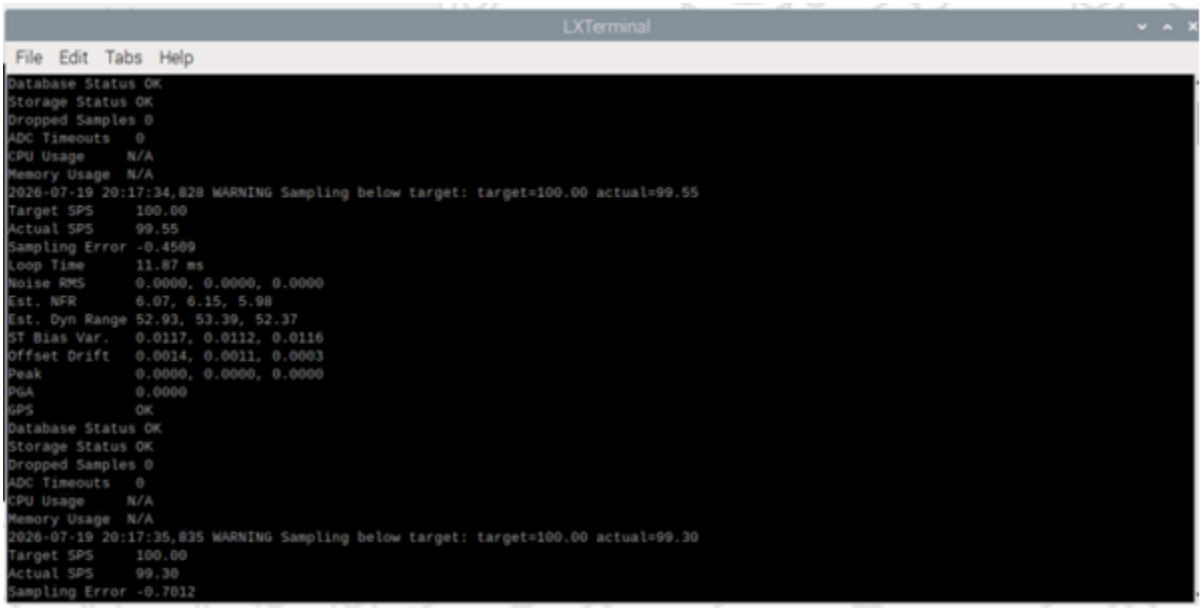

Figure 5. Steady-state health telemetry emitted once per second by the acquisition loop

Table 3. Acquisition timing and continuity metrics

Metric	Observed Value
Target sampling rate	100 SPS
Software-reported steady-state rate	99,30–99,55 SP
Sampling-rate deviation	−0,45% to −0,70%
Steady-state loop execution time	11,87 ms
First-iteration loop execution time	356,64 ms
File-level effective sampling rate, 1 h	100 SPS
Samples recorded per channel in 1 h	360.000 of 360.000 expected
Dropped samples	0
ADC read timeouts	0
MiniSEED gaps / overlaps	0 / 0
Storage status	Operational across PostgreSQL, CSV, and MiniSEED

During the one-hour acquisition run, 360.000 samples per channel were recorded, exactly matching the expected number at 100 SPS. No dropped samples, MiniSEED gaps, or overlaps were detected. The file-level effective sampling rate was 100 SPS, even though the short-window software telemetry fluctuated slightly below the nominal rate. This difference is important: the short-window rate reflects user-space scheduling variation, whereas the file-level result confirms the continuity of the complete recorded sequence.

The health telemetry also provides an operational advantage. Sampling rate, loop duration, counters, storage status, and per-axis statistics are generated periodically and stored with the other operational records. These values provide information about acquisition behaviour while the instrument is running, rather than requiring the operator to inspect waveform files after a session.

Timestamping and Time Synchronization

The hybrid timestamping mechanism was designed to avoid reading the DS3231 for every sample. The RTC provides an absolute time reference, but accessing it through I²C for every 10 ms sample would add unnecessary bus activity. The software therefore records the RTC time together with the corresponding perf_counter_ns() value every 60 s. Timestamps between two RTC updates are generated from the elapsed monotonic-counter interval.

GPS UTC information from the NMEA $GPRMC sentence is used as the preferred startup reference when available. A valid position fix is not required for this time information, so a loss of GPS position after startup does not invalidate the timestamping mechanism as long as the RTC and monotonic counter remain available.

The observed behaviour supports the intended separation between absolute time and short-term interval measurement. The method reduces repeated I²C transactions and avoids using the adjustable Linux

system clock as the only timing source. However, the present evaluation does not establish externally calibrated UTC accuracy, hardware-level channel simultaneity, or quantitative timestamp jitter.

Processing and Data Persistence

The processing.py module converts the three ADC voltage values into acceleration products using the session calibration parameters. The processing sequence includes conversion using sensor sensitivity and calibrated offset, a fifth-order moving-average filter, and an adaptive deadband based on three times the measured noise standard deviation. The processing module also maintains statistics used by the monitoring interface. Raw ADC-derived records are retained separately from processed monitoring values. This distinction prevents filtering or deadband operations intended for display and monitoring from replacing the original waveform data.

The persistence layer uses three output destinations for different purposes. CSV files provide raw and statistical records that can be inspected or processed with general numerical tools. MiniSEED provides waveform files that can be read by seismological software. PostgreSQL stores recent statistics and historical summaries used by the REST service and dashboard. MiniSEED files produced during the evaluation were opened successfully in ObsPy and PQL II for the three axes, providing file-level evidence of interoperability.

The three destinations use different failure policies. PostgreSQL records are temporarily buffered in memory when the database connection is unavailable. The application attempts reconnection every 30 s and flushes buffered records after the connection is restored. The configured buffer holds 10.000 statistical records, equivalent to approximately 2,8 h at one record per second. This policy is intended for temporary network or database interruptions rather than prolonged outages. CSV output does not use the same buffering approach because a failure of the local storage medium would not be resolved by accumulating more records in memory. MiniSEED data are assembled in 100-sample blocks for the three waveform channels.

Service and Presentation Layer

The service layer is implemented as a separate FastAPI process. It provides access to recent status, real-time data, historical records, recording sessions, file downloads, and on-demand spectral previews. The browser dashboard accesses these functions through the REST endpoints listed in Table 4.

Table 4. REST endpoints exposed by the service layer

Endpoint	Purpose	Access Pattern
`/api/latest`	Most recent statistics or status record	Polled once per second
`/api/realtime`	Recent window of samples	Requested by chart view
`/api/history`	Historical range query	On demand
`/api/sessions`	Available recording sessions	Requested when views load
`/api/download`	CSV or MiniSEED file retrieval	User initiated
`/api/spectrum`	FFT/PSD preview from selected raw file	User initiated; outside acquisition loop

The separation between the API process and the acquisition process is important for availability. Web requests, file retrieval, and spectral-preview calculations do not execute inside the ADC sampling loop. If the REST process is stopped, the acquisition process can continue because the two processes do not call each other directly. The API reads data products already generated by the acquisition process.

The /api/spectrum endpoint performs the most computationally intensive service operation. It reads a selected raw CSV file, removes the mean, applies a Hanning window, and calculates the transform and power spectral density. This calculation is performed on demand rather than periodically so that it does not become part of the sampling path.

The frontend is implemented as a single-page application using HTML, CSS, and JavaScript, with Chart.js for plotting and Leaflet for the station map. The dashboard provides station information, system status, sampling counters, measurement statistics, waveform and spectral displays, file downloads, and connection status. Functional testing against the documented checklist passed for the tested dashboard functions.

Figure 6 shows the dashboard home view. The interface polls the latest status once per second. When the REST service becomes unavailable, the connection indicator changes to a disconnected state while the acquisition process continues independently. This provides a distinction between a problem with data acquisition and a problem with the monitoring interface.

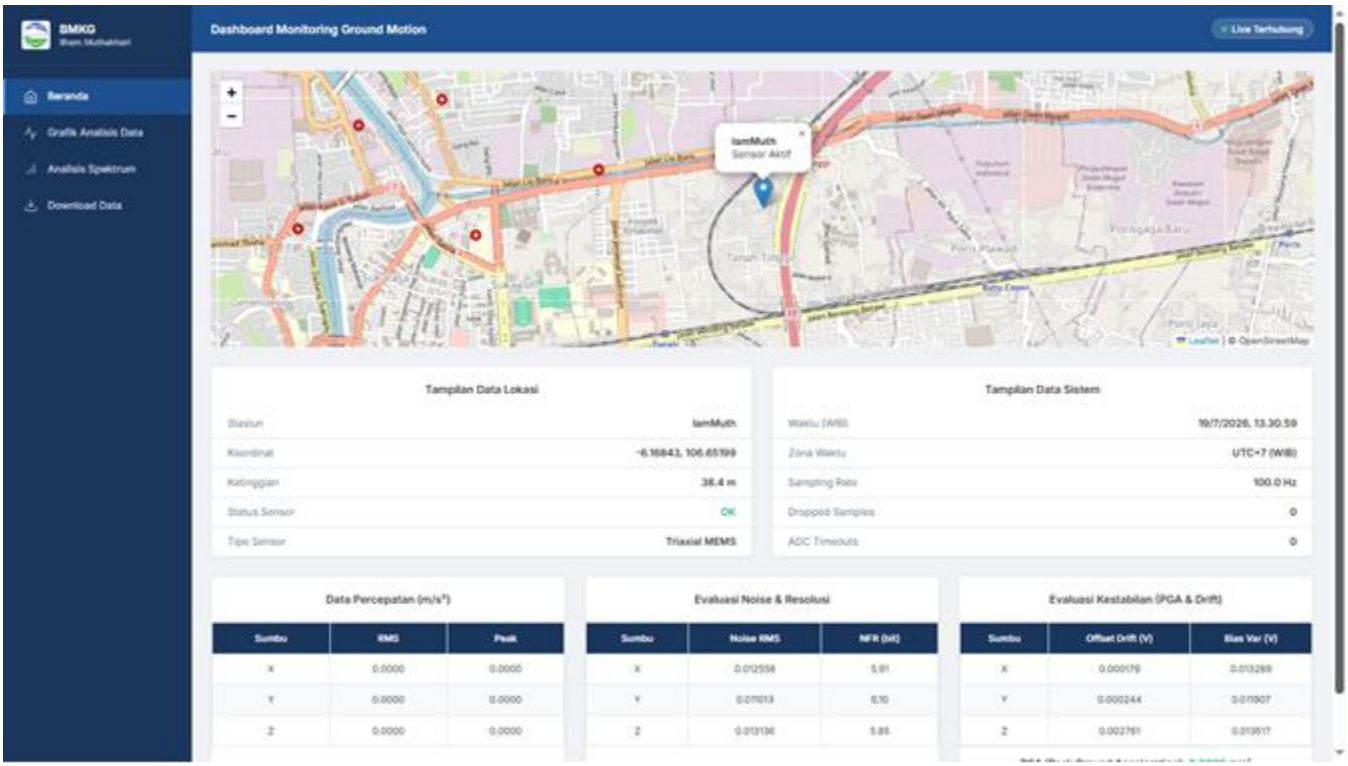

Figure 6. Dashboard home view showing station location, system status, and per-axis measurement and stability information

Fault Tolerance and Recovery

Fault handling is distributed according to component responsibility. Failures affecting essential measurement functions can block or defer acquisition, whereas failures in supporting functions are handled through degraded operation. Table 5 summarises the five tested or observed conditions.

Table 5. Induced or observed failure scenarios and system response

Condition	Observed Response	Effect on Acquisition
Watchdog access denied by OS	Failure logged; watchdog disabled; startup continued	Unaffected
ADC unavailable at startup	Retry every 5 s until converter responded	Deferred, then normal
PostgreSQL server stopped during session	Records buffered; reconnection every 30 s; buffer flushed after recovery	Unaffected
No GPS position fix	Position metadata unavailable; time retained from `$GPRMC` or RTC	Unaffected
REST/API process terminated	Dashboard reported disconnection; acquisition process remained separate	Unaffected

The watchdog case is particularly relevant because the access failure occurred during normal startup rather than being artificially introduced. The application did not have sufficient permission to open /dev/watchdog, so the operation returned a permission error. The failure was recorded and the watchdog was disabled, but startup continued and acquisition subsequently operated normally. This behaviour prevents a non-essential supervision mechanism from stopping an otherwise functional recorder.

The ADC is handled differently. Because the converter is required to obtain samples, an unavailable ADC cannot simply be ignored. The application instead retries the device every five seconds. Acquisition is therefore deferred until the required hardware becomes available.

The database test demonstrates a different recovery strategy. When PostgreSQL was stopped during acquisition, records were retained in the configured buffer while reconnection attempts continued. After the database became available again, the buffered records were flushed. The acquisition path therefore continued while the persistence service was temporarily unavailable.

The GPS case produced a smaller degradation. When a position fix was unavailable within the configured waiting period, position metadata was omitted, but acquisition continued because the RTC and

previously obtained time information were still available. Similarly, termination of the REST process affected dashboard access but did not stop the acquisition process.

These results show that fault handling is not uniform across all components. The response depends on whether the failed component is required for measurement, required only for supporting functions, or recoverable after a temporary interruption. This distinction is one of the main practical effects of separating the software into functional responsibilities.

The watchdog implementation also has a known limitation. A watchdog can detect a failure of the loop that services it, but it cannot by itself determine whether the samples being produced are scientifically valid. A loop may continue running and servicing the watchdog while producing corrupted or implausible values.

Dependency Structure, Maintainability, and Extensibility

Figure 7 shows the import dependency graph of the implemented software. The graph is acyclic, with main.py acting as the orchestration root and the functional modules below it. No cyclic dependency or upward dependency was identified during inspection of the import structure.

The resulting structure also makes individual modules easier to exercise independently. For example, the processing module can be tested with synthetic voltage values without connecting the ADC, while storage classes can be supplied with fabricated records. Hardware classes can also be tested separately from the processing and presentation layers. This separation reduces the amount of software that needs to be involved when investigating a specific module.

Three practical change scenarios were traced through the architecture. Replacing the ADC would require a new hardware interface that provides the same expected output to the processing layer, together with the associated configuration change. Adding another output format, such as HDF5, would require a new writer in the persistence layer. Adding a dashboard view would primarily involve the frontend and, where necessary, an additional REST endpoint. These examples show how the responsibility boundaries can localise common modifications.

The architecture does not isolate every possible change. For example, increasing the sampling rate substantially could affect assumptions in several modules at once. A target of 1,000 SPS would require reconsideration of the processing window, statistics interval, MiniSEED block size, and execution-time budget. Therefore, the modular structure helps with changes that fit the existing interfaces, but it does not remove dependencies created by fundamental changes to system requirements. The implementation uses a Python virtual environment with a pinned dependency manifest. Containerisation was considered but not used for the current instrument. The present system is deployed on a constrained ARM platform, and the reproducibility requirements were considered sufficiently covered by the dependency manifest and externalised configuration. This choice is specific to the current deployment and does not imply that containerisation would be unsuitable for a larger managed instrument fleet.

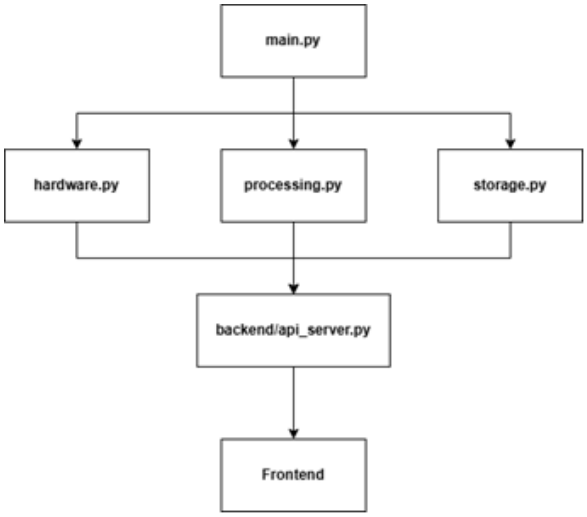

Figure 7. Module dependency graph showing downward dependencies and the absence of cyclic imports

Limitations

Several limitations should be considered when interpreting these results. The architecture was evaluated on one hardware configuration consisting of the ADS1256, ADXL335, DS3231, NEO-7M, and Raspberry Pi 3 Model B+. Although the hardware abstraction structure is intended to support replacement of individual devices, portability to another ADC or acquisition platform has not been experimentally demonstrated.

Then, the acquisition loop is single-threaded. This simplifies state management and makes the timing path easier to inspect, but it does not use all available Raspberry Pi CPU cores. The health statistics are generated and stored but are not currently connected to an automatic alert or shutdown policy. Similarly, the system does not perform content-level waveform validation capable of identifying plausible but corrupted measurements while the acquisition loop continues to run. Maintainability and extensibility were assessed qualitatively through dependency analysis and traced change scenarios. Quantitative software-quality metrics such as coupling, complexity, or change-impact

measurements were not applied. The continuous acquisition test lasted one hour. The result confirms the behaviour of the tested run but does not establish reliability over several weeks or under long-term field conditions.

Finally, the timing evaluation verifies sampling continuity and software-generated timestamps but does not provide an external measurement of UTC accuracy, sampling jitter, or channel simultaneity. These properties require a separate timing reference and a dedicated experimental setup.

Conclusion

This work developed and evaluated a modular software architecture for a Raspberry Pi-based 24-bit geophysical data acquisition system. The implemented software separates hardware communication, signal processing, data persistence, REST services, and the user interface into distinct modules, allowing the acquisition process to operate independently from non-essential services. The architecture also uses externalised configuration, hybrid timestamping, deadline-driven scheduling, and component-specific failure handling to support continuous unattended operation.

The evaluation showed that the system maintained the configured 100 SPS acquisition rate during the one-hour test, recording 360,000 samples per channel without dropped samples, MiniSEED gaps, or overlaps. The system reached a record-ready state in 7.840 s and continued acquisition when the watchdog was unavailable, PostgreSQL was temporarily disconnected, GPS position was unavailable, or the REST service was stopped. These results indicate that modularity and graceful degradation can be implemented on a general-purpose Raspberry Pi Linux platform without requiring a real-time kernel. The separation of responsibilities also provides a practical basis for extending individual parts of the software without directly modifying the acquisition loop.

The evaluation was limited to one Raspberry Pi-based hardware configuration and a one-hour continuous acquisition test. External validation of timestamp accuracy, channel simultaneity, long-term field reliability, and quantitative software-quality metrics was not performed. Future work should therefore evaluate the architecture over longer deployment periods, on additional hardware configurations, and with an external timing reference.

Acknowledgements

The authors acknowledge the Sekolah Tinggi Meteorologi Klimatologi dan Geofisika for supporting the development environment and the instrumentation facilities used in the underlying undergraduate final-year project.

References

Alessandro, A. D., B, S. S., & Vitale, G. (n.d.). Optimization of Low-Cost Monitoring Systems for On-Site Earthquake Early-Warning of Critical Infrastructures. 1, 1–13.

Ali Bakir, R., Elksasy, M., Salam, M., Saraya, M., & Abdelsalam, M. (2023). Low Cost MEMS accelerograph: structure, operation and application to seismology. Delta University Scientific Journal, 6(1), 193–204. https://doi.org/10.21608/dusj.2023.291042

Crognale, M., Rinaldi, C., Potenza, F., Gattulli, V., Colarieti, A., & Franchi, F. (2024). Developing and Testing High-Performance SHM Sensors Mounting Low-Noise MEMS Accelerometers. Sensors, 24(8), 1–19. https://doi.org/10.3390/s24082435

Knapp, A., & Bloom, A. J. (2022). Easy as piadcs: A low-cost, ultra-high-resolution data acquisition system using a Raspberry Pi. Applications in Plant Sciences, 10(3), 1–6. https://doi.org/10.1002/aps3.11485

Nof, R. N., Chung, A. I., Rademacher, H., Dengler, L., & Allen, R. M. (2019). MEMS accelerometer mini-array (MAMA): A low-cost implementation for earthquake early warning enhancement. Earthquake Spectra, 55(1), 21–38. https://doi.org/10.1193/021218EQS036M

Özcebe, A. G., Tiganescu, A., Ozer, E., Negulescu, C., Galiana-merino, J. J., Tubaldi, E., Toma-danila, D., Molina, S., Kharazian, A., Bozzoni, F., Borzi, B., & Balan, S. F. (2022). Raspberry Shake-Based Rapid Structural Identification of Existing Buildings Subject to Earthquake Ground Motion: The Case Study of Bucharest. Sensors, 22(13), 1–24. https://doi.org/10.3390/s22134787

Özdemir, K., & Kömeç Mutlu, A. (2024). Cost-Effective Data Acquisition Systems for Advanced Structural Health Monitoring. Sensors, 24(13). https://doi.org/10.3390/s24134269

Pratama, Y. A., & Suharsono, S. (2025). Performance Evaluation of ADS 1256 for Geoelectric Data Acquisition System: Laboratory Scale Comparative Study. Jurnal Geofisika, 23(1), 23. https://doi.org/10.36435/jgf.v23i1.674

Ramdeane, A., & Lynch, L. (2020). Low Cost Seismic Data Acquisition System Based on Open Source

Hardware and Software Tools. 496–505. https://doi.org/10.47412/vycb8830

Ringler, A. T., Anthony, R. E., Bastien, P., & Pascale, A. (2023). Introduction to the Digitization of Seismic Data : A User ' s Guide. 94(4). https://doi.org/10.1785/0220220158.Introduction

Vlachos, I., Anagnostou, M. N., Avlonitis, M., & Karakostas, V. (2025). Upgrading a Low-Cost Seismograph for Monitoring Local Seismicity. GeoHazards, 6(1), 1–32. https://doi.org/10.3390/geohazards6010004