Scalable Modular Deep Learning Framework for Earthquake Early Warning Systems

Satriawan Rasyid Purnama[1], Adi Wibowo[1,*], Arjuna Wahyu Kusuma[1], Liem Roy Marcelino[1], Indra Waspada[1], Cecep Pratama[2], Leni Sophia Heliani[2], David Prambudi Sahara[3], Sri Widiyantoro[3], Shindy Rosalia[3], Bondan Febriarta[3], Cahyo Adhi Hartanto[4]
[1]Department of Informatics, Universitas Diponegoro, Semarang, Indonesia
[2]Department of Geodetic Engineering, Universitas Gadjah Mada, Yogyakarta, Indonesia
[3]Global Geophysics Research Group, Faculty of Mining and Petroleum Engineering, Institute of Technology Bandung, Bandung, Indonesia
[4]Faculty of Computer Science, Universitas Indonesia, Depok, Indonesia
[*]Corresponding author: bowo.adi@live.undip.ac.id

Keywords: *Earthquake early warning system, deep learning, real-time seismic data processing, containersized deployment, WebSocket-based alert dissemination*

Received: April 12, 2026

Earthquake Early Warning Systems (EEWS) require fast, reliable, and scalable processing of seismic data to deliver timely alerts and reduce disaster impacts. This study presents a modular deep learning–based EEWS framework that introduces a latency-aware, event-driven processing pipeline for real-time seismic data handling. The proposed architecture restructures the EEWS workflow into decoupled modules that communicate asynchronously via Kafka topics, enabling efficient data streaming and scalable processing. A standardized ingestion mechanism converts raw seismic traces into uniform JSON messages, ensuring interoperability and reducing preprocessing overhead. In addition, parallel data ingestion combined with topic-based message partitioning allows concurrent processing of waveform streams, improving throughput under high data rates. The framework integrates Docker-based containersization, multiprocessing, NGINX load balancing, and WebSocket communication to support flexible deployment and low-latency alert dissemination. Experimental results demonstrate reduced end-to-end latency and improved resource efficiency across multiple deployment scenarios. These findings highlight the effectiveness of pipeline-level optimization in supporting reliable and scalable real-time EEWS applications.

Povzetek: Predlagani sistem z globokim učenjem omogoča hitrejšo, zanesljivejšo in bolj razširljivo obdelavo seizmičnih podatkov za pravočasno opozarjanje na potrese.

1 Introduction

Providing rapid alerts during the initial stages of an earthquake is essential for mitigating potential damage and reducing casualties. Earthquake Early Warning Systems (EEWS) have evolved from manual seismic wave detection toward increasingly automated and data-driven approaches, enabling faster and more reliable response mechanisms. Early systems relied heavily on human interpretation of seismic signals, which limited their responsiveness and scalability in real-time scenarios. The growing availability of continuous seismic data streams and advances in computational methods have significantly transformed EEWS into complex, real-time information systems capable of processing large-scale geophysical data.

Traditional automated methods such as Short-Term Average/Long-Term Average (STA/LTA) and matched-filter technique have been widely adopted for seismic phase detection due to their simplicity and low computational requirements [1–2]. However, these approaches often struggle to maintain high detection accuracy under noisy conditions and may produce false triggers or missed detections, particularly in complex seismic environments [3]. These limitations have motivated the exploration of more advanced techniques that can better capture the nonlinear characteristics of seismic signals while maintaining real-time performance.

Recent advances in machine learning and deep learning have led to the development of data-driven EEWS components, including models such as PhaseNet and PickNet, which demonstrate improved performance in seismic phase picking tasks. In addition, integrated systems such as LOC-FLOW and QuakeFlow combine deep learning models with distributed computing infrastructures to enable scalable and efficient real-time seismic data processing [2,4]. These systems represent a significant shift toward intelligent and automated EEWS pipelines, where detection, localization, and data streaming are tightly integrated within cloud-based environments.

Established operational systems include both monitoring software and dedicated EEW methods: GITEWS uses SeisComP3 for real-time earthquake monitoring and tsunami warning [5]; AQMS is an Earthworm-based operational monitoring system [6]; and ElarmS [7] and

PRESTo [8] are dedicated EEW systems. Nevertheless, many of these systems are primarily designed for centralized infrastructures and may lack the flexibility required to integrate modern distributed architectures and deep learning–based workflows. While some recent frameworks attempt to bridge this gap, challenges remain in achieving seamless interoperability, scalability, and efficient resource utilization in heterogeneous computing environments.

QuakeFlow, for instance, exemplifies a modern EEWS architecture that leverages Kafka-based streaming, containersized deployment, and cloud-native design principles to support both archived and real-time seismic data processing [4]. Despite its advantages, the integration of deep learning and distributed processing frameworks can introduce significant computational overhead and increase system latency, particularly under high data throughput conditions [6–8]. These trade-offs highlight the need for system-level optimization strategies that balance detection performance with real-time processing efficiency.

To address these challenges, this study proposes a modular and cloud-oriented EEWS architecture that emphasizes system-level optimization for real-time performance and resource efficiency. The proposed framework integrates multiprocessing for parallel data ingestion [9], NGINX-based load balancing for efficient request distribution [10], modular multi-containers deployment for scalability, and lightweight WebSocket communication using Express.js for low-latency alert dissemination [11].

The main contribution of this work lies in the design of a latency-aware, event-driven processing pipeline that restructures EEWS data flow into decoupled, asynchronously communicating modules using Kafka topics. The proposed system introduces a standardized streaming interface that transforms raw seismic traces into uniform JSON messages at the ingestion stage, enabling seamless interoperability between modules while reducing preprocessing overhead. Furthermore, the integration of parallel data ingestion with topic-based message partitioning enables concurrent processing of waveform streams, significantly improving throughput under high data rates. The architecture minimizes end-to-end latency through lightweight inter-service communication and optimized message routing, while maintaining extensibility by allowing independent scaling and replacement of detection, localization, and visualization components. Compared to existing frameworks that primarily emphasize component integration, this work focuses on optimizing the internal data pipeline to reduce communication overhead, enhance processing concurrency, and ensure consistent low-latency performance in real-time seismic monitoring environments.

2 System architecture

In this study, an Earthquake Early Warning System (EEWS) is developed using a modular and independent framework as illustrated in Figure 1. The system adopts a microservices architecture in which each module operates as an independent service with defined network interfaces [12]. Because service decomposition increases the number of network interfaces, deployment security must also be considered [13]. Key technologies include ObsPy for seismic data processing [14–16], Apache Kafka for message streaming [17], TensorFlow for model execution [18], and MongoDB is a document-oriented NoSQL database with a flexible schema and indexing support, and the document-oriented backend [19].

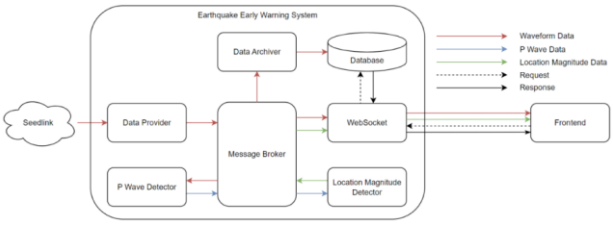


Figure 1: Proposed EEWS architecture

The proposed Earthquake Early Warning System (EEWS) is architecturally organized into several specialized modules that communicate via Kafka, allowing for flexible scaling and efficient real-time processing. Each module fulfills a critical role in the end-to-end detection and alert pipeline, including:

- **Data provider**, the system begins with the data provider module, which is responsible for ingesting real-time seismic data streams via SeedLink using ObsPy. This component emphasizes low-latency and low-resource data flow by immediately converting incoming seismic traces into standardized JSON format. These are then published to specific Kafka topics, such as trace_topic, enabling consistent and uniform data availability for downstream modules. This standardization is a crucial requirement for real-time EEWS performance [4].
- **P Wave detector**, this module applies a deep learning model with multiple convolutional layers to detect the onset of P Waves [20]. Implemented with TensorFlow and served through FastAPI, this module offers high-speed inference with minimal delay. FastAPI is selected for its asynchronous request handling and seamless integration with machine learning frameworks, facilitating fast and reliable phase detection, an essential capability for issuing timely warnings [21].
- **Message broker**, a central component in the architecture which functions as a distributed message broker utilizing a publish/subscribe pattern. Kafka handles real-time data transmission between containerss by organizing information into topics, trace_topic for waveform data, p_wave_topic for pick detections, and result_topic for output from the location-magnitude detector. This approach supports parallel processing by multiple consumers and ensures robust data distribution throughout the EEWS pipeline [17].
- **Data archiver and databases**, to ensure long-term data availability, a data archiver module subscribes to waveform streams and stores them in a persistent

backend. The system employs MongoDB, a flexible NoSQL document database, which excels at storing unstructured seismic data and supporting high-speed I/O operations. Its dynamic schema and advanced indexing capabilities make it ideal for storing and retrieving waveform information in time-critical environments [4,19].

- **Location and magnitude detector**, this module estimates key earthquake parameters namely the hypocenter and magnitude based on the detected P Waves. It also leverages TensorFlow and outputs results to the result_topic in Kafka. Its modular design supports later integration of alternative algorithms or more complex models for enhanced precision, offering flexibility for continuous system evolution.
- **WebSocket and frontend**, the system includes a WebSocket service that listens to Kafka topics and relays real-time data to the frontend. This allows for instant updates of seismic waveforms, detection outputs, and earthquake parameters in client interfaces. Whether built with Express.js, FastAPI, or other frameworks, the frontend can dynamically render seismograms and display warning messages, ensuring users receive immediate situational awareness [11].

3 Optimization methods

The performance evaluation of the proposed Earthquake Early Warning System (EEWS) focuses on identifying the most effective technological combination to optimize real-time data processing and reduce system latency. Several optimization components were developed and tested under various scenarios to assess their impact on key performance metrics: CPU usage, memory usage, and data delay. The evaluation was conducted using 500-sample trials for each test case, with resource monitoring via Docker Desktop. The experimental modules are as follows:

- **Multithreading and multiprocessing**, the data provider module was evaluated using four execution strategies: sequential, multithreading, multiprocessing, and a hybrid multithreaded–multiprocessing model, to assess their effects on data ingestion efficiency and real-time system performance. As the entry point of the EEWS pipeline, this module directly influences downstream processing latency. In sequential mode, waveform data are processed one at a time on a single thread, which often results in higher latency and limited throughput under increasing data loads. The multithreaded strategy improves responsiveness by allowing concurrent operations within a single process, making it suitable for I/O-intensive tasks [9]. Multiprocessing further enhances scalability by distributing tasks across multiple independent processes and CPU cores, reducing contention and improving performance under continuous streaming conditions [9,22]. The hybrid approach combines both techniques to balance throughput, resource utilization, and delay, offering greater flexibility for real-time processing. The architectural details of each execution strategy are illustrated in Figure 2.
- **Configurated Kafka with NGINX**, in this configuration, as shown in figure 3, Kafka serves as the message broker for real-time waveform distribution. To enhance data throughput and support scalable processing, Kafka is deployed with three brokers and partitioned topics (trace_topic, p_wave_topic, result_topic), enabling parallel consumption by components such as data archivers and P Wave detectors. To bolster throughput and high availability, the system may employ three Kafka brokers alongside NGINX, which can function as a reverse proxy and load balancer [23]. By incorporating NGINX as a load balancer, the system can effectively distribute client requests, reducing bottlenecks and improving the efficiency of the data flow. This configuration maximizes throughput, minimizes delay, and reduces performance bottlenecks under high traffic.
- **Multicontainers**, to improve responsiveness, throughput, and deployment scalability, the data archiver and P Wave detector modules were implemented using a multi-containers architecture, in which multiple containers instances of the same service operate concurrently under a distributed execution scheme. In the experimental setup, the data archiver module was deployed using 1 to 5 containerss to evaluate the effect of horizontal scaling on waveform storage performance and overall pipeline efficiency. This configuration enabled parallel write operations and reduced storage-side bottlenecks, thereby increasing the rate at which incoming seismic traces could be archived and made available to downstream services [24]. A similar strategy was applied to the P Wave detector module to support concurrent inference on multiple waveform segments under continuous real-time data streams. It should be emphasized that the P Wave detection model used in this study is not a newly proposed model, but the previously published deep learning model described in [20], which is integrated here into a distributed containersized EEWS pipeline. The technical contribution of this work therefore lies in the architectural design and deployment strategy, particularly in the use of replicated service containerss to improve workload distribution, reduce service contention, and maintain low-latency processing across ingestion, archival, detection, and alert dissemination stages. Such a design is consistent with the established advantages of containersization for lightweight deployment, service isolation, portability, and efficient scaling in distributed computing environments [25-27].
- **Express.js**, the WebSocket module is designed to transmit processed seismic information, including waveform data, estimated event location, and magnitude, from the backend processing pipeline to the frontend interface in real time. This module serves as the final communication layer of the EEWS

architecture, ensuring that the results generated by upstream processing components can be delivered to users with minimal delay. In this study, two implementation frameworks, FastAPI and Express.js, were evaluated to determine their suitability for real-time alert dissemination. FastAPI was considered due to its native compatibility with python-based services and its efficient support for asynchronous communication, which makes it well aligned with machine learning inference pipelines implemented in python [21]. This characteristic is advantageous in architectures where model inference, data processing, and service delivery are closely integrated within a unified software environment. Express.js, in contrast, was selected as an alternative lightweight framework built on the event-driven and non-blocking execution model of Node.js. Such an architecture is particularly appropriate for communication-intensive workloads involving frequent message broadcasting and multiple simultaneous client connections [28].

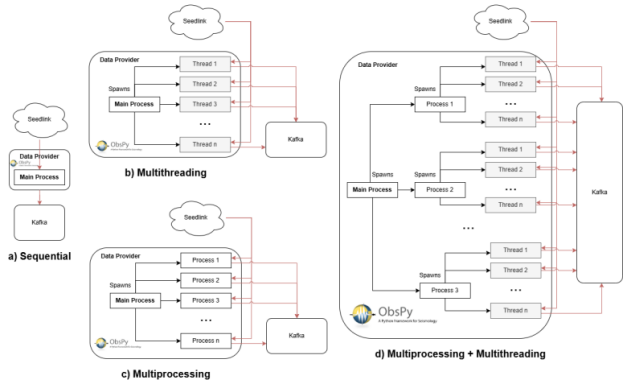


Figure 2: Proposed data provider architecture

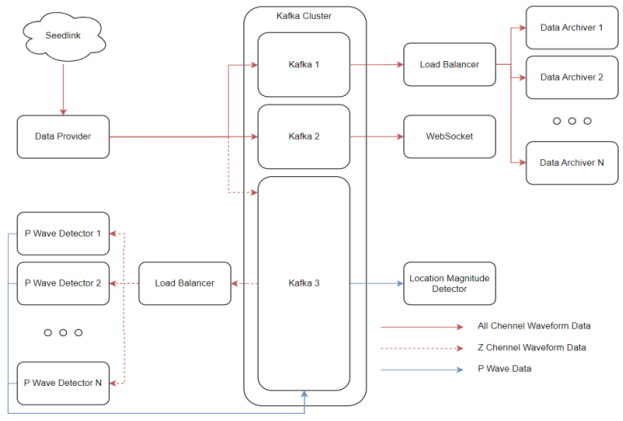


Figure 3: Proposed message broker architecture with NGINX.

4 Results and discussion

To execute the system, users are advised to utilize Docker Compose, which facilitates containers orchestration in a streamlined and reproducible manner. The system can be initiated in detached mode using the command “docker-compose up -d”, enabling background execution. Alternatively, a specific configuration file can be specified with “docker-compose -f <configuration file> up -d”, allowing for customized setups. This flexibility supports systematic testing across multiple scenarios. The test cases are defined through various Docker Compose configuration files, each tailored to evaluate particular aspects of the system, such as parallel data processing methods, load balancing mechanisms, multi-containers orchestration, and WebSocket-based communication. In particular, the test cases targeting parallel data processing on the data provider component are structured to assess the system's capacity to simultaneously manage diverse data handling strategies. These scenarios offer a controlled environment for comparative analysis and performance evaluation of the system under different architectural and operational conditions. The complete source code and configuration files are available in the official GitHub repository at https://github.com/ArjunaWahyu/MDLBEEWS.

In this section, we describe the optimization goal and experimental setup designed to evaluate the performance of each component in our EEWS. Our goal is to identify the most effective combination of technologies for enhancing the speed and reliability of EEWS. By evaluating various data storage methods, real-time communication frameworks, and data management solutions, we aim to optimize the system’s performance to ensure timely and accurate seismic warnings.

Table 1: Data provider experimental results

Scenario	**Data Delay (seconds)**	**CPU Usage (%)**	**Memory Usage (MB)**	**Note**
Sequential	N/A	N/A	N/A	Terminated after ±40 minutes because memory exhausted; benchmark not completed.
Multithreading	3.150	31.24	112	Became unstable with a large number of threads.
Multiprocessing	2.955	44.50	476	Stable
Multiprocessing + Multithreading	4.768	62.50	600	Stable

The results in Table 1 show clear differences among the execution strategies. The sequential configuration did not complete the benchmark because its memory consumption progressively increased during continuous Seedlink ingestion until the available system memory was exhausted. The process was consequently terminated after approximately 40 minutes, and complete-run averages for delay, CPU usage, and memory usage could not be calculated. Multithreading achieved a data delay of 3.150 seconds but became unstable when a large number of threads were used. Multiprocessing remained stable and produced the lowest measured delay of 2.955 seconds, with 44.50% CPU usage and 476 MB of memory. The hybrid multiprocessing–multithreading configuration also remained stable but produced higher delay and resource consumption. Therefore, multiprocessing provided the most favorable balance of stability, latency, and resource utilization.

Table 2: Message broker experimental results

Scenario	**Data Delay (seconds)**	**CPU Usage (%)**	**Memory Usage (MB)**
Kafka 3 Containers	0.006329	27.24	3,108
Kafka 3 Containers + NGINX	0.015902	25.68	2,591

Table 2 highlights differences between using Kafka alone and Kafka with NGINX as a load balancer. Kafka-only setup results in lower delay due to internal message partitioning. However, integrating NGINX introduces additional delay because of overhead in routing and communication layers. Despite this, CPU usage remains stable across both setups. Memory usage is higher in the Kafka-only configuration, which reflects Kafka's responsibility for both brokering and load management. Adding NGINX reduces Kafka's memory load, shifting routing tasks to the proxy. The decision between setups depends on whether latency or memory efficiency is the higher priority.

Table 3: Databases experimental results

Scenario	**Process (seconds)**	**CPU Usage (%)**	**Memory Usage (MB)**
InfluxDB	0.007740	32.91	433.92
MongoDB	0.004206	20.16	1,925.22

Table 3 compares the performance of InfluxDB and MongoDB under the evaluated database workload. MongoDB recorded the shorter process time of 0.004206 seconds and lower CPU usage of 20.16%, but required substantially more memory at 1,925.22 MB. InfluxDB recorded a process time of 0.007740 seconds and CPU usage of 32.91%, while using only 433.92 MB of memory. These results demonstrate a trade-off between processing speed and memory consumption: MongoDB is preferable when lower processing time is prioritized, whereas InfluxDB provides a more memory-efficient alternative.

Table 4: Data archiver experimental results

Scenario	**Data Delay (seconds)**	**CPU Usage (%)**	**Memory Usage (MB)**
5 Containers	0.015763	197.90	518.59
4 Containers	0.015903	185.81	418.59
3 Containers	0.017323	173.48	321.34
2 Containers	0.018164	160.55	242.68
1 Containers	0.019274	150.37	151.34

Table 5: P Wave detector experimental results

Scenario	**Data Delay (seconds)**	**CPU Usage (%)**	**Memory Usage (MB)**
5 Containers + FastApi 2 Workers	0.033676	192.14	20,240
4 Containers + FastApi 2 Workers	0.034843	177.18	19,836
3 Containers + FastApi 2 Workers	0.035214	162.55	18,749
2 Containers + FastApi 2 Workers	0.035951	157.21	17,685
5 Containers	0.032647	280.00	16,530
4 Containers	0.033197	257.94	15,528
3 Containers	0.034010	228.17	14,214
2 Containers	0.034936	195.73	13,530

The performance evaluation of multi-containers execution highlights the trade-offs between latency, scalability, and resource usage. Increasing the number of containerss improves processing speed and reduces delay by distributing tasks in parallel. However, CPU and memory usage also increase accordingly. For the Data Archiver, as shown in Table 4, performance improves with containers count, but resource consumption also increases. The P Wave Detector shows similar trends, as presented in Table 5. Integration with FastAPI helps scale requests efficiently; however, it slightly increases delay and memory usage while reducing CPU utilization. The appropriate configuration therefore depends on whether the system prioritizes processing performance or resource conservation.

Table 6: WebSocket experimental results

Scenario	Data Delay (seconds)	CPU Usage (%)	Memory Usage (MB)
Express.js 1 client	0.001324	12.02%	95.49MB
Express.js 5 client	0.001452	13.38%	96.05MB
FastApi, 1 client	0.001356	17.71%	72.67MB
FastApi, 5 client	0.001578	39.05%	72.97MB

Based on the analysis of Table 6, Express.js offers better CPU efficiency and lower delay in both low and high connection scenarios, making it suitable for real-time, high-concurrency applications. However, it uses more memory. FastAPI consumes less memory but incurs significantly higher CPU usage, especially when scaling to multiple clients. The trade-off is between memory efficiency (FastAPI) versus CPU efficiency and latency (Express.js). Framework selection should be aligned with system priorities Express.js for low-latency high-traffic environments, or FastAPI for memory-constrained deployments.

5 Conclusion

This study evaluated parallel processing, message brokering, multi-containers deployment, and WebSocket communication in a modular EEWS architecture. Multiprocessing provided the best balance of stability and performance, achieving a 2.955-second delay with 44.50% CPU usage and 476 MB of memory. Kafka alone produced the lowest messaging latency, whereas adding NGINX reduced memory usage at the cost of additional delay. Increasing the number of services containerss improved processing speed but required more CPU and memory. For WebSocket communication, Express.js achieved lower latency and CPU usage, while FastAPI offered better memory efficiency. Overall, the results demonstrate that no single configuration is optimal for every deployment; component selection should reflect the required balance among latency, scalability, and resource availability. Future work should validate the complete architecture using end-to-end latency, reliability, and realistic multi-station workloads.

Acknowledgements

The authors would like to acknowledge to Badan Meteorologi, Klimatologi, dan Geofisika, Jakarta, Indonesia for the Seismic Data and the Directorate General of Higher Education Research and Technology and Dikti AI Centre for the NVIDIA DGX A100 access. This work was supported by the Riset Kolaborasi Indonesia (RKI) NonAPBN Universitas Diponegoro Indonesia under Grant 391-04/UN7.D2/PP/V/2023 and the World Class Research UNDIP (WCRU) Universitas Diponegoro Grant 2024.

References

[1] Tunç S, Tunç B, Çaka D, Budakoğlu E. An Overview of Traditional and Next-Generation Earthquake Early Warning Systems. J Adv Res Nat Appl Sci. 2024;10(3):747–760. https://doi.org/10.28979/jarnas.1481067

[2] Zhang M, Liu M, Feng T, Wang R, Zhu W. LOC-FLOW: An End-to-End Machine Learning-Based High-Precision Earthquake Location Workflow. Seismol Res Lett. 2022;93(5):2426–2438. https://doi.org/10.1785/0220220019

[3] Liu M, Tan YJ. Evaluating the performance of machine-learning-based phase pickers when applied to ocean bottom seismic data: Blanco oceanic transform fault as a case study. Geophys J Int. 2025;242(3). https://doi.org/10.1093/gji/ggaf256

[4] Zhu W, Hou AB, Yang R, Datta A, Mousavi SM, Ellsworth WL, Beroza GC. QuakeFlow: a scalable machine-learning-based earthquake monitoring workflow with cloud computing. Geophys J Int. 2023;232(1):684–693. https://doi.org/10.1093/gji/ggac355

[5] Hanka W, Saul J, Weber B, Becker J, Harjadi P, Fauzi, GITEWS Seismology Group. Real-time earthquake monitoring for tsunami warning in the Indian Ocean and beyond. Nat Hazards Earth Syst Sci. 2010;10:2611–2622. https://doi.org/10.5194/nhess-10-2611-2010

[6] Hartog JR, Friberg PA, Kress VC, Bodin P, Bhadha R. Open-Source ANSS Quake Monitoring System Software. Seismol Res Lett. 2020;91(2A):677–686. https://doi.org/10.1785/0220190219

[7] Brown HM, Allen RM, Hellweg M, Khainovski O, Neuhauser D, Souf A. Development of the ElarmS methodology for earthquake early warning: Realtime application in California and offline testing in Japan. Soil Dyn Earthq Eng. 2011;31(2):188–200. https://doi.org/10.1016/j.soildyn.2010.03.008

[8] Satriano C, Elia L, Martino C, Lancieri M, Zollo A, Iannaccone G. PRESTo, the earthquake early warning system for Southern Italy: Concepts, capabilities and future perspectives. Soil Dyn Earthq Eng. 2011;31(2):137–153. https://doi.org/10.1016/j.soildyn.2010.06.008

[9] Aziz ZA, Abdulqader DN, Sallow AB, Omer KH. Python Parallel Processing and Multiprocessing: A Review. Acad J Nawroz Univ. 2021;10(3):345–354. https://doi.org/10.25007/ajnu.v10n3a1145

[10] NGINX. Using NGINX as HTTP load balancer. NGINX Documentation. Available from: https://NGINX.org/en/docs/http/load_balancing.html (accessed 1 June 2024).

[11] Fette I, Melnikov A. The WebSocket Protocol. RFC 6455. Internet Engineering Task Force; 2011. https://doi.org/10.17487/RFC6455

[12] Jamshidi P, Pahl C, Mendonça NC, Lewis J, Tilkov S. Microservices: The journey so far and challenges ahead. IEEE Softw. 2018;35(3):24–35. https://doi.org/10.1109/MS.2018.2141039

[13] Hannousse A, Yahiouche S. Securing microservices and microservice architectures: A systematic mapping study. Comput Sci Rev. 2021;41:100415. https://doi.org/10.1016/j.cosrev.2021.100415

[14] Beyreuther M, Barsch R, Krischer L, Megies T, Behr Y, Wassermann J. ObsPy: A Python Toolbox for Seismology. Seismol Res Lett**.** 2010;81(3):530–533. https://doi.org/10.1785/gssrl.81.3.530

[15] Krischer L, Megies T, Barsch R, Beyreuther M, Lecocq T, Caudron C, Wassermann J. ObsPy: A Bridge for Seismology into the Scientific Python Ecosystem. Comput Sci Discov**.** 2015;8(1):014003. https://doi.org/10.1088/1749-4699/8/1/014003

[16] Megies T, Beyreuther M, Barsch R, Krischer L, Wassermann J. ObsPy - What can it do for data centers and observatories? Ann Geophys**.** 2011;54(1):47–58. https://doi.org/10.4401/ag-4838

[17] Martín C, Langendoerfer P, Zarrin PS, Díaz M, Rubio B. Kafka-ML: Connecting the data stream with ML/AI frameworks. Future Gener Comput Syst. 2022;126:15–33. https://doi.org/10.1016/j.future.2021.07.037

[18] Abadi M, Barham P, Chen J, et al. TensorFlow: A System for Large-Scale Machine Learning. In: 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16). Savannah, GA: USENIX Association; 2016. p. 265–283. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi

[19] Syafrudin M, Alfian G, Fitriyani NL, Rhee J. Performance Analysis of IoT-Based Sensor, Big Data Processing, and Machine Learning Model for Real-Time Monitoring System in Automotive Manufacturing. Sensors. 2018;18(9):2946. https://doi.org/10.3390/s18092946

[20] Wibowo A, Heliani LS, Pratama C, et al. Deep learning for real-time P Wave detection: A case study in Indonesia's earthquake early warning system. Appl Comput Geosci**.** 2024;24:100194. https://doi.org/10.1016/j.acags.2024.100194

[21] Ramírez S. FastAPI: Concurrency and async/await [software documentation]. Available from: https://FastAPI.tiangolo.com/async/ (accessed 1 June 2024).

[22] Python Software Foundation. threading—Thread-based parallelism. Python 3 Documentation. Available from: https://docs.python.org/3/library/threading.html (accessed 1 June 2024).

[23] Ma C, Chi Y. Evaluation test and improvement of load balancing algorithms of NGINX. IEEE Access. 2022;10:14311–14324. https://doi.org/10.1109/ACCESS.2022.3146422

[24] Pahl C. Containersization and the PaaS Cloud. IEEE Cloud Comput. 2015;2(3):24–31. https://doi.org/10.1109/MCC.2015.51

[25] Dalal YM, Supreeth S, Rohith S, Shruthi G, Sowmya BJ. Towards smarter IoT through taxonomy and prospective directions for microservices placement in fog computing paradigms. Discov Artif Intell. 2026;6:89. https://doi.org/10.1007/s44163-025-00601-5

[26] Cantiello P, De Cesare W, Di Filippo A, Peluso R. A scalable data-driven architecture for real-time multidisciplinary volcano monitoring. Earth Sci Inform. 2026;19:68. https://doi.org/10.1007/s12145-026-02116-8

[27] Joyce JE, Sebastian S. Inference-Time-Driven Autoscaling for Inference Workloads: A Comparative Study of Latency-Variant Models in Kubernetes. Technologies. 2026;14(6):350. https://doi.org/10.3390/technologies14060350

[28] Tilkov S, Vinoski S. Node.js: Using JavaScript to Build High-Performance Network Programs. IEEE Internet Comput. 2010;14(6):80–83. https://doi.org/10.1109/MIC.2010.145