

Performance Evaluation of Python Web Frameworks for Resource-Constrained Embedded Systems

Maroš Valášek, Tomáš Ranuša

University of Zilina

Faculty of Electrical Engineering
and Information Technology

Univerzitná 8215/1, 010 26 Zilina, Slovakia

{valasek, ranusa}@stud.uniza.sk

Gabriel Gašpar, Roman Budjac

University of Zilina

Research centre

Univerzitná 8215/1, 010 26 Zilina, Slovakia

{gabriel.gaspar, roman.budjac}@uniza.sk

Abstract. *The aim of this work is to analyze the performance characteristics of two popular Python web frameworks, FastAPI and Django. These frameworks are deployed on a Raspberry Pi 5 single-board computer. Testing is conducted using the wrk tool, a custom Python concurrent benchmarking script, and an asynchronous WebSocket client based on the websockets library. The study examines the impact of different WSGI/ASGI web servers on throughput, latency stability, and system resource consumption. The results show that FastAPI is a faster and less system-intensive choice compared to Django, although the violin plots indicate that while FastAPI has a lower average latency, Django exhibits a lower maximum latency.*

Keywords. Python, framework, embedded systems, resource-constrained

1 Introduction

With the growing number of IoT devices and embedded systems, there is a need for efficient data processing directly at the edge of the network. These devices have limited computing power, making it critical to choose a software architecture that maximizes throughput with minimal resource consumption. This necessitates the careful selection of lightweight yet robust web frameworks capable of handling high request volumes within such constrained environments. This paper evaluates the performance of Python web frameworks, specifically FastAPI(FastAPI, n.d.) and Django(Django Software Foundation, n.d.), for their suitability in such resource-constrained settings, focusing on metrics crucial for real-time edge processing (Dey et al., 2025). Specifically, the evaluation encompasses key performance indicators such as request throughput, response times, and resource utilization (e.g., CPU, memory) for basic operations (Bednarz and Miłosz, 2025).

The aim of this work is to analyze the performance

characteristics of two popular Python web frameworks: FastAPI and Django. While both are Python-based and widely utilized for backend development (KP and N, 2023), they differ significantly in their architectural paradigms and performance profiles (Choma et al., 2023), which are critical considerations for IoT deployments.

FastAPI is a modern, high-performance web framework for building Application Programming Interfaces (APIs) with Python, which natively supports asynchronous programming via asyncio(Python Software Foundation, n.d.), Python’s built-in library for asynchronous programming that enables concurrent handling of I/O operations through an event loop without blocking threads and the WebSocket protocol. FastAPI, known for its high performance and asynchronous capabilities, offers a compelling solution for microservices and real-time data processing, while Django, a more mature and feature-rich framework, provides a comprehensive suite for complex web applications (Poliakov and Yeremenko, 2024).

Django, on the other hand, is a high-level Python web framework following the Model-View-Template architectural pattern. Historically designed for a synchronous processing model via Web Server Gateway Interface (WSGI), Django was integrated with the Django Channels package to extend it with the Asynchronous Server Gateway Interface (ASGI) protocol and support WebSockets. This integration allows Django to handle long-lived connections and asynchronous tasks, positioning it as a contender for applications requiring real-time functionalities, despite its traditional synchronous core (Ruchi et al., 2026). While FastAPI is recognized for its API-first design and asynchronous capabilities, providing low-latency communication (Mabotha et al., 2025), Django’s extensive features and scalability often come with increased CPU utilization compared to more lightweight frameworks (Ruchi et al., 2026, Tusa and Clayman, 2021).

While both frameworks facilitate web application

development, their fundamental architectural differences, particularly regarding synchronous versus asynchronous operations, profoundly affect their performance in resource-constrained environments (Bednarz and Miłosz, 2025). This study examines how these architectural paradigms translate into measurable performance indicators, including throughput, latency, and resource utilization, on a Raspberry Pi 5, a representative platform for edge computing applications (Choma et al., 2023, Ruchi et al., 2026). This comparative analysis seeks to provide empirical data to guide developers in selecting the optimal Python web framework for IoT and embedded *systems*, where computational efficiency and responsiveness are paramount.

The insights from this analysis help determine which framework is better suited for resource-limited edge deployments, particularly those requiring efficient data handling and real-time responsiveness.

2 Related works

Several studies have evaluated the performance of Python web *frameworks*. Some research focuses on comparing *frameworks* in standard environments, while other studies compare databases running on these *frameworks*, noting that much larger performance differences often arise specifically due to the database layer. Our study isolates the performance of the *frameworks* themselves for IoT-specific tasks directly on ARM architecture typical for single-board computers. For instance, previous works have explored general web application development with Python, specifically highlighting *frameworks* like Django and Flask for their simplicity and readability in web server applications (KP and N, 2023). However, these studies often overlook the unique constraints and performance implications of deploying such *frameworks* on resource-limited embedded *systems*, which is a primary focus of the current research. Furthermore, while many evaluations focus on general server-side performance, our investigation delves into the specific performance characteristics pertinent to IoT contexts, such as real-time responsiveness and efficiency under constrained computational resources. For example, (Morabito et al., 2017) highlights how management of HTTP requests can significantly impact gateway performance, indicating the necessity of evaluating *frameworks* within the context of their communication protocols. Other research explores the use of Flask for microservices on edge devices, finding it suitable for object detection functionalities due to its ease of integration and testing across diverse hardware (Alqahtani et al., 2024). However, these studies often concentrate on high-level application functionality rather than low-level performance metrics relevant to continuous, resource-intensive IoT operations. A study comparing Flask and web2py for AI microservices, for instance, found Flask to have lower latency

and higher throughput, underscoring the importance of empirical evaluations for framework selection in specialized contexts (Idowu and Omiyale, 2026). Conversely, while some studies discuss the general use of Python in embedded *systems* for programmability and code mobility (Morabito et al., 2017), they do not typically provide comprehensive benchmarking of web *frameworks* tailored for specific real-time performance metrics like inference latency, CPU utilization, and memory footprint in TinyML applications (Dey et al., 2025). Django and FastAPI were selected as representatives of two opposing architectural paradigms, a mature synchronous framework and a modern, natively asynchronous one, enabling a direct comparison of synchronous versus asynchronous processing. Although Flask is a widely used lightweight framework, it predates FastAPI and lacks native asynchronous and WebSocket support, making FastAPI a more suitable candidate for our real-time, sensor-data streaming use case. While Flask and web2py have been examined in prior studies, our focus was on the two most widely adopted frameworks offering native WebSocket support. This paper extends upon this body of work by conducting a rigorous comparative analysis of FastAPI and Django specifically on a Raspberry Pi 5, a ubiquitous platform for IoT and edge computing, to provide empirical data on their performance characteristics under scenarios typical of embedded *systems*.

3 Methodology and setup

This section outlines the experimental design, hardware and software configurations, and the metrics employed for evaluating the performance of FastAPI and Django. Our methodology encompasses controlled experimental conditions to ensure reproducibility and statistical validity, focusing on isolating the performance impacts of each framework on critical *system* resources. The experimental setup includes a Raspberry Pi 5 acting as the host for the web *frameworks*, meticulously configured to monitor CPU, memory, and network I/O during various load conditions. This approach facilitates a direct comparison of how each framework handles varying request loads and data processing demands, thereby offering insights into their suitability for edge computing environments.

3.1 Hardware specification

A Raspberry Pi 5 was used, which is a single-board computer based on a 64-bit ARM Cortex-A76 Broadcom BCM2712 processor with a frequency of 2.4 GHz. This unit is equipped with 8 GB of LPDDR4X SDRAM, providing ample memory for complex computational tasks and multitasking, typical of edge AI applications. This configuration enables robust processing capabilities suitable for demanding machine

learning inferences at the edge (Emanuilov and Dimov, 2025).

Both the server and client Raspberry Pi 5 units ran Raspberry Pi OS (Raspberry Pi Ltd, n.d.) (64-bit, Debian-based), while the development workstation used for some tests ran Windows 11 Pro (Microsoft, n.d.). The detailed technical specifications of the hardware used in this study are summarized in Table 1 (Raspberry Pi Ltd, n.d.). The Raspberry Pi 5's PCIe interface is accessible through an FPC (Flexible Printed Circuit) connector, allowing for external expansion.

Table 1. Technical Specifications of Raspberry Pi 5 (8 GB version)

Parameter	Specification
Model	Raspberry Pi 5 (8 GB RAM variant)
SoC	Broadcom BCM2712
CPU	Quad-core ARM Cortex-A76 @ 2.4 GHz
GPU	VideoCore VII (OpenGL ES 3.1, Vulkan 1.2)
Memory	8 GB LPDDR4X SDRAM (up to 34 GB/s)
Storage	microSD (UHS-I), PCIe 2.0 external support
PCIe	1 × PCIe 2.0 lane (FPC connector)
USB	2 × USB 3.0, 2 × USB 2.0
Networking	Gigabit Ethernet (PoE+ support via HAT)
Wireless	Wi-Fi 802.11ac, Bluetooth 5.0 / BLE
Display	2× micro-HDMI (dual 4Kp60)
Camera	2× MIPI CSI (4-lane)
DSI	2× MIPI DSI (4-lane)
GPIO	40-pin header (backward compatible)
Power	5 V USB-C (recommended 5 V/5 A)
Dimensions	85 mm × 56 mm

3.2 Software specification

The work examines the impact of different WSGI/ASGI web servers (Gunicorn, Daphne, Uvicorn). Specifically, we deployed FastAPI applications using Uvicorn, a high-speed ASGI web server, and Django applications using Gunicorn with Uvicorn workers to ensure optimal asynchronous handling and performance comparability between the *frameworks* (Durán et al., 2024).

For FastAPI, Uvicorn was used as the web server, which is a lightweight ASGI server based on the uvloop and httptools *libraries*. This choice leverages Uvicorn's asynchronous capabilities to efficiently handle concurrent connections, crucial for high-throughput edge deployments.

For the Django framework, three web servers were initially tested: Gunicorn (Gunicorn, n.d.), Daphne (Django Software Foundation, n.d.), and Uvicorn (Uvicorn, n.d.). Gunicorn is a traditional WSGI server that delivered the highest HTTP throughput; however, it lacks native WebSocket support, rendering it unsuitable for applications that necessitate real-time communication. Daphne and Uvicorn are both ASGI servers and were assessed for their compatibility and performance with Django's asynchronous capabilities.

Daphne proved to be much slower and did not support multiple workers, so it was excluded from the test. Consequently, Uvicorn was selected as the deployment server for Django, running directly with multiple workers through the ASGI interface provided by Django Channels. This configuration was chosen as the optimal deployment setup for Django to facilitate a fair performance comparison with FastAPI, which is also served by Uvicorn. This configuration ensures that both *frameworks* operate under similar asynchronous serving mechanisms, thereby minimizing server-side performance discrepancies attributable to the underlying WSGI/ASGI implementation rather than the framework logic itself.

System statistics were collected using the psutil (psutil developers, n.d.) *library* in a separate thread, preventing blocking of HTTP requests. The *system* resource utilization metrics, including CPU load, memory consumption, and network throughput, were logged at regular intervals to provide a granular view of performance under various synthetic load conditions. Initial measurements showed a latency of 120 ms despite a 7 ms ping because the psutil function was blocking further requests, this was resolved by moving psutil to a secondary thread. This concurrent monitoring strategy allowed for precise, non-intrusive data collection, enabling an accurate assessment of framework performance under realistic load conditions.

The software stack consisted of Python 3.13.5, FastAPI 0.137.2, Uvicorn 0.49, Django 6.0.6, Django Channels 4.3.2, psutil 7.2.2, websockets 16.0, seaborne 0.13.2, matplotlib 3.10.8 pandas 2.3.3 and numpy 2.3.5. All communication was conducted over the HTTP/1.1 protocol, the default for both Uvicorn and the wrk benchmarking tool (Wrk developers, n.d.).

3.3 Test scenarios

The server was implemented in the main.py file and consisted of four HTTP endpoints and a WebSocket handler:

- GET ping: The simplest possible endpoint returning only "message": "pong" with no processing or *system* info. This endpoint serves as a baseline for measuring raw network and framework overhead, isolating the effects of data serialization and deserialization.
- GET hello: Similar to ping but additionally returns *system* information from the Pi (CPU, RAM, temperature, timestamp). It simulates a real endpoint returning device status. The only difference from the ping endpoint is the additional overhead of retrieving and serializing cached system information, which is continuously updated by a background thread.
- POST echo: Receives a JSON payload with a random name, number, and a 64-character string and returns it back. It simulates sending sensor data to the server, typical for research center projects. This endpoint specifically tests the framework's efficiency

in handling data ingress, parsing, and egress under varying payload sizes, directly relevant to IoT data acquisition pipelines.

- **GET heavy:** The server calculates the sum of the squares of numbers from 0 to 10000 before responding. It simulates an endpoint that must process data, such as sensor data aggregation. This computational task evaluates the *frameworks'* capacity for CPU-intensive operations, reflecting scenarios common in edge analytics or localized machine learning inference.
- **WS ws:** A persistent two-way connection in which the client can continuously stream sensor data to the server independently of individual server requests. The server immediately returns each received message as an echo response. This simulates a live sensor-data stream where new data are pushed by the client as they become available, rather than being repeatedly requested by the server. This WebSocket interaction assesses the *frameworks'* real-time communication capabilities and their efficiency in maintaining continuous, low-latency data streams, which is critical for interactive edge applications.

The detailed performance comparison across all tested web framework configurations, including GET, POST, and WebSocket throughput alongside latency percentiles, is presented in Table 3, where req/s denotes requests per second, WS represents WebSocket throughput, and p50 and p99 correspond to the median and 99th percentile latency, respectively. The best-performing results in the table are highlighted in bold. The data was collected using a custom Python concurrent benchmark script running 12 parallel workers over a duration of 10 minutes per configuration.

The results of Django + Daphne are included in Tables 3 and ?? solely to demonstrate its performance inadequacy during the initial evaluation phase and to justify its exclusion from the primary benchmark, rather than as a viable deployment configuration.

Table 2. Performance Comparison of Web Framework Configurations

Configuration	GET req/s	POST req/s	WS req/s	p50 [ms]	p99 [ms]
Django + Daphne	82	82	52	37	88
Django + Uvicorn	165	82	55	23	50
FastAPI + Uvicorn	187	93	54	19	45

3.4 Benchmarking tools

To evaluate the performance characteristics of each framework under heterogeneous workloads, a combination of industry-standard benchmarking tools and custom-developed test scripts was employed to ensure consistent and objective data collection. An initial

benchmark consisted of a 10-minute stress test executed via a Python script on a Windows-based *system*, utilizing the concurrent futures module to generate parallel tasks. This approach was designed to simulate high-concurrency conditions and to assess sustained *system* performance under load. However, experimental configurations with 50, 100, 300, and 500 worker threads were constrained by operating *system* throughput, with a maximum observed rate of approximately 190 requests per second. To address these constraints, subsequent experiments were conducted using Apache JMeter 5.5, deployed within a dedicated benchmarking container. This setup enabled the generation of controlled, scalable workloads, facilitating the evaluation of server response times under varying network conditions while minimizing client-side bottlenecks.

To further mitigate the limitations of the Windows environment, an auxiliary Raspberry Pi device was used to run the wrk utility. This configuration allowed for a more accurate assessment of server performance under unconstrained load conditions. The wrk tool, implemented in C, can generate high request rates, reaching tens of thousands of requests per second. Additionally, its extensibility via Lua scripting enables customized request generation, response handling, and reporting, thereby providing flexibility for a wide range of performance testing scenarios.

All benchmarking scripts, including the concurrent Python script, the asynchronous WebSocket client, and the visualization scripts, were developed by the authors. Each request was logged to a CSV file with its timestamp, test type, latency, and system metrics, and subsequently processed using the pandas and statistics libraries. The complete set of benchmark results, including raw CSV outputs and supplementary measurements, is available in the project GitHub repository DePolarBear, n.d.

4 Results and discussion

This section presents the empirical results and a comparative analysis of the performance metrics, focusing on latency, throughput, and resource utilization, derived from the rigorous benchmarking of FastAPI and Django under various simulated loads.

4.1 Throughput

Using the wrk tool, we achieved up to 1660 req/s with 12 workers compared to the 180 req/s limit on the Windows PC. This significant increase underscores the impact of client-side resource limitations on benchmark accuracy and validates the necessity of utilizing robust load generation tools and environments. The utilization of wrk, known for its efficiency in generating high-volume traffic due to its asynchronous I/O model and C-based implementation, allowed for a more realistic evaluation of server capabilities under sustained

throughput. Furthermore, this disparity in throughput highlights the critical importance of selecting appropriate benchmarking tools and environments to accurately assess the maximum transaction processing capacity of an application.

The detailed results of the ping and request throughput benchmark across all tested configurations are presented in Table 3, where all measurements were obtained using the wrk HTTP benchmarking tool with 4 threads, 100 concurrent connections, and a test duration of 30 seconds per endpoint. The best-performing results in the table are highlighted in bold.

Table 3. Ping and Request Throughput Benchmark

Configuration	ping req/s	hello req/s	heavy req/s	localhost ping
Django + Daphne	299	289	223	295
Django + Uvicorn	634	622	549	568
FastAPI + Uvicorn	1401	1597	1133	1660

FastAPI with Uvicorn reached 1,660 req/s on the localhost ping, while Django with Uvicorn reached 568 req/s. This substantial difference in throughput for a trivial endpoint suggests fundamental architectural distinctions in request handling and processing overhead between the two *frameworks*. Specifically, FastAPI's asynchronous nature and Uvicorn's efficient ASGI server likely contribute to its superior performance in I/O-bound operations, whereas Django's more synchronous foundation, even when paired with an ASGI server, may introduce additional overhead.

On the "hello" endpoint, FastAPI processed 1,597 req/s, whereas Django handled 622 req/s. This indicates that even with minimal additional processing for retrieving *system* information, FastAPI maintains a significant performance advantage, suggesting superior efficiency in managing concurrent requests and integrating *system*-level data.

During the WebSocket test with 200 clients and 20,000 messages, FastAPI achieved a throughput of 8,751.5 req/s, while Django reached 7,606.7 req/s. This demonstrates FastAPI's marginally superior efficiency in managing persistent, real-time communication streams, even though both *frameworks* exhibit robust performance under high WebSocket load. This finding suggests that while both *frameworks* are competent in handling synchronous and asynchronous tasks, FastAPI's design, often leveraging asynchronous programming models, provides a slight edge in scenarios requiring high concurrency and real-time data exchange.

The comprehensive WebSocket load test results across varying numbers of concurrent clients for both *frameworks* are presented in Table 4, where higher throughput values indicate better performance, lower latency values indicate better responsiveness, p50, p95, and p99 represent latency percentiles, max denotes the

worst observed latency, and the best values per column are highlighted in bold. All measurements were obtained using a custom Python asyncio benchmark script executed on the second Raspberry Pi acting as the client, with each configuration tested using 10 messages per client.

4.2 Latency

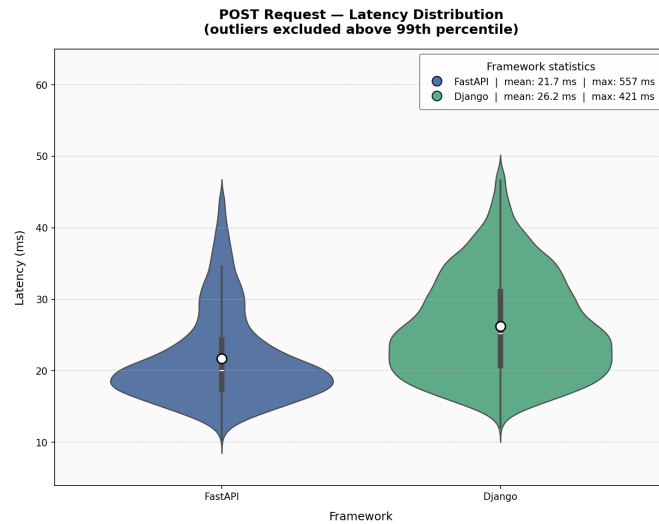
We visualized the latency distribution using violin plots. These plots offer a comprehensive view of response-time variability under different load conditions, thereby providing more insight into performance consistency than simple mean or median metrics (Hintze and Nelson, 1998).

All plots were generated using matplotlib, with violin plots additionally relying on seaborn. The 99th percentile latency represents the value below which 99% of all measured latencies fall, characterizing the worst-case response time while excluding extreme outliers.

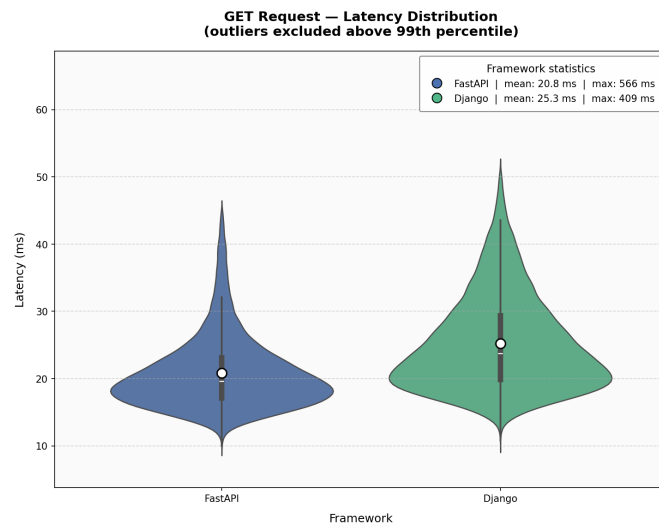
The plots show that FastAPI has a lower average latency, but Django has a lower maximum latency. The latency distributions for GET, POST, and WebSocket requests are visualized using violin plots in Fig. 1 a), b), c) respectively, with outliers above the 99th percentile excluded to better illustrate the core distribution of response times. In the overall benchmark, the average latency for FastAPI was 16.38 ms with a maximum of 565.79 ms. In contrast, the average latency for Django was 19.34 ms, but its maximum latency was lower at 420.99 ms. This discrepancy suggests that while FastAPI generally offers faster responses, Django exhibits greater stability under peak load conditions, potentially due to different internal queuing mechanisms or resource management strategies. These observed trade-offs between CPU and RAM utilization, alongside thermal performance, provide critical insights for deployment strategies, particularly for resource-constrained environments or applications requiring sustained high performance.

4.3 Resource utilization

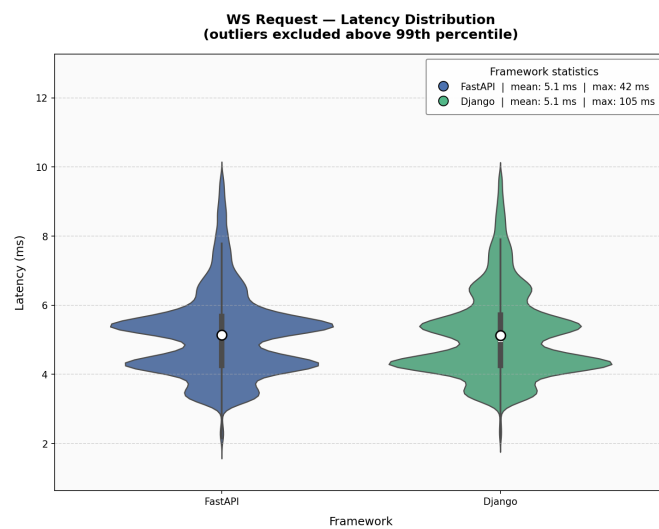
In the case of FastAPI, there is lower CPU utilization but higher RAM utilization compared to Django. These metrics were collected over a 10-minute benchmark session using a custom Python concurrent script with 12 parallel workers, continuously logging CPU utilization, RAM consumption, and CPU temperature at 1-second intervals via the psutil *library*, resulting in approximately 600 samples per configuration. No thermal throttling was observed during any of the test runs, as the CPU temperature remained below 48 °C across all configurations, well under the Raspberry Pi 5 throttling threshold of 85 °C. The average CPU utilization for Django with Uvicorn was 15.5% (max 29.6%), whereas FastAPI averaged 11.8% (max 18.9%). This indicates that FastAPI, despite its higher throughput,



(a)



(b)



(c)

Figure 1. Latency distribution of requests for FastAPI and Django (outliers above 99th percentile excluded) a) POST requests b) GET requests c) WS requests.

Table 4. Performance Comparison of FastAPI and Django Under Load

Framework	Clients	Requests	Throughput [req/s]	p50 [ms]	p95 [ms]	p99 [ms]	max [ms]
FastAPI	10	1000	1255.5	5.59	9.11	10.96	13.95
FastAPI	50	5000	5414.7	7.46	10.23	11.82	35
FastAPI	100	10000	6777.7	10.03	15.25	19.97	115.45
FastAPI	200	20000	8751.5	16.35	26.27	31.92	171.12
Django	10	1000	1051.0	7.38	10.80	13.58	16.21
Django	50	5000	4229.3	9.14	14.40	20.54	42.56
Django	100	10000	6164.6	13.00	20.13	26.64	88.70
Django	200	20000	7606.7	18.63	31.05	43.56	206.13

is more CPU-efficient, likely due to its asynchronous architecture and optimized I/O handling, which minimizes blocking operations.

Conversely, average RAM usage was 13.8% (1113.18 MB) for Django and 15.5% (1247.31 MB) for FastAPI. The average temperature was slightly lower for FastAPI (42.2 °C) than for Django (43.2 °C). This suggests that while FastAPI leverages memory more intensively, possibly for maintaining asynchronous task states or larger event loops, its optimized CPU usage contributes to overall cooler operation, indicating a more efficient thermal profile under load.

The overall *system* resource utilization comparison between FastAPI and Django, including average and peak CPU and RAM consumption recorded during the benchmark, is presented in Table 5, where lower values indicate more efficient resource usage and the best values per column are highlighted in bold.

Table 5. Benchmark Comparison of FastAPI and Django

Metric	FastAPI	Django
Avg CPU [%]	11.82	15.49
Max CPU [%]	18.9	29.6
Avg RAM [%]	15.47	13.82
Max RAM [%]	15.8	14.7
Avg RAM [MB]	1247.31	1113.18
Max RAM [MB]	1274.8	1188.3

5 Conclusions

After the tests, FastAPI proved to be faster and less *system*-intensive choice compared to Django. This holds true for both synchronous and asynchronous operations, aligning with findings that asynchronous architectures generally outperform synchronous ones in terms of lower tail latencies under high concurrency. We tested "plain" FastAPI against Django, which basically has everything implemented, which may be a reason for Django's higher latency. It should be noted that

during testing, Django was deployed with its default configuration, meaning that built-in middleware components such as session management, authentication, Cross-Site Request Forgery (CSRF) protection, and the admin panel were not explicitly disabled. These components introduce additional processing overhead on every request, even when not actively utilized, and represent a limitation of this study. Therefore, the results should be interpreted in the context of a default Django deployment compared to a minimalist FastAPI setup, rather than as an absolute performance comparison between the two *frameworks*. For our purposes of controlling and receiving data from embedded *systems*, we chose FastAPI for its speed and modularity, which make it adaptable to any task. This decision is further supported by evidence highlighting the benefits of asynchronous architectures in high-transaction *systems*, where minimizing blocking I/O leads to superior tail latencies.

Acknowledgments

This research was supported by the project VEGA through the Research on motion data analysis methods for applications in diagnosis and therapy of gnostically relevant symptoms under Grant 1/0147/25.

References

- Alqahtani, D. K., Cheema, A., & Toosi, A. N. (2024). Benchmarking deep learning models for object detection on edge computing devices. *arXiv (Cornell University)*, (242). <https://doi.org/10.48550/arxiv.2409.16808>
- Bednarz, B., & Miłosz, M. (2025). Benchmarking the performance of python web frameworks. *Journal of Computer Sciences Institute*, 36, 336–341. <https://doi.org/10.35784/jcsi.7738>
- Choma, D., Chwaleba, K., & Dzieńkowski, M. (2023). The efficiency and reliability of backend technologies: Express, django, and spring boot. *Informatyka Automatyka Pomiary w Gospodarce i Ochronie*

- Środowska, 13, 73–78. <https://doi.org/10.35784/iappos.4279>
- DePolarBear. (n.d.). Framework-Benchmark-Data [Accessed: 2026-06-30].
- Dey, A., Srivastava, S. P., Singh, G., & Pettit, R. G. (2025). Real-time performance benchmarking of tinyml models in embedded systems (pico: Performance of inference, cpu, and operations). *arXiv (Cornell University)*, 279–284. <https://doi.org/10.48550/arxiv.2509.04721>
- Django Software Foundation. (n.d.). Daphne [Accessed: 2026-06-29].
- Durán, F., Martínez, M., Lago, P., & Martínez-Fernández, S. (2024). Insights into resource utilization of code small language models serving with runtime engines and execution providers. <https://doi.org/10.48550/ARXIV.2412.15441>
- Emanuilov, S., & Dimov, A. (2025). A quantitative framework for evaluating architectural patterns in ml systems. <https://doi.org/10.48550/ARXIV.2501.11543>
- FastAPI. (n.d.). FastAPI. Retrieved June 30, 2026, from <https://fastapi.tiangolo.com/>
- Gunicorn. (n.d.). Gunicorn - Python WSGI HTTP Server for UNIX [Accessed: 2026-06-30].
- Hintze, J. L., & Nelson, R. D. (1998). Violin plots: A box plot-density trace synergism. *The American Statistician*, 52(2), 181–184. <https://doi.org/10.1080/00031305.1998.10480559>
- Idowu, O. F., & Omiyale, A. D. (2026). Comparative evaluation of flask and web2py for ai microservices: An empirical benchmark on model-inference workloads. *ABUAD Journal of Engineering Research and Development (AJERD)*, 9, 12–27. <https://doi.org/10.53982/ajer.2026.0901.02-j>
- KP, B., & N, D. (2023). Leveraging django: A web framework for efficient infrastructure monitoring tools. *International Journal for Research in Applied Science and Engineering Technology*, 11, 407–412. <https://doi.org/10.22214/ijraset.2023.55414>
- Mabotha, E., Mabunda, N., Ali, A., & Khan, B. (2025). Exploring dynamic restful api implementation in iot environments using docker. *Scientific Reports*, 15(34267). <https://doi.org/10.1038/s41598-025-16460-0>
- Microsoft. (n.d.). Download Windows 11 [Accessed: 2026-06-30].
- Morabito, R., Farris, I., Iera, A., & Taleb, T. (2017). Evaluating performance of containerized iot services for clustered devices at the network edge. *IEEE Internet of Things Journal*, 4, 1019–1030. <https://doi.org/10.1109/jiot.2017.2714638>
- Poliakov, M., & Yeremenko, B. (2024). Information communication system of the electronic estimation of adolescents' special abilities. *Management of Development of Complex Systems*, 139–145. <https://doi.org/10.32347/2412-9933.2024.58.139-145>
- psutil developers. (n.d.). psutil documentation [Accessed: 2026-06-30].
- Python Software Foundation. (n.d.). asyncio — Asynchronous I/O [Accessed: 2026-06-30].
- Raspberry Pi Ltd. (n.d.). Raspberry Pi 5 [Accessed: 2026-06-29].
- Ruchi, R., Saroj, S., Joshi, H., & Thareja, D. Y. (2026). A comparative analysis of fastapi and django frameworks: Evaluating performance, scalability, and concurrency efficiency in modern web applications. *International journal of scientific research in engineering and management*, 10, 1–9. <https://doi.org/10.55041/ijserem58536>
- Tusa, F., & Clayman, S. (2021). The impact of encoding and transport for massive real-time iot data on edge resource consumption. *Journal of Grid Computing*, 19(32). <https://doi.org/10.1007/s10723-021-09577-9>
- Uvicorn. (n.d.). Uvicorn [Accessed: 2026-06-30].
- Wrk developers. (n.d.). wrk - a HTTP benchmarking tool [Accessed: 2026-06-30].