Comparative analysis of HTTP request handling in different Python frameworks

Patrick Given Malatjie[1*], *Zimbini* Faniso-Myaka[2*],

[1]Council for Scientific and Industrial Research, Optronic and Sensor Systems, Pretoria, South Africa

[2]Council for Scientific and Industrial Research, Optronic and Sensor Systems, Pretoria, South Africa

Abstract. The Hypertext Transfer Protocol (HTTP) has become a fundamental component for client-server communication in web applications. This paper investigates the performance and suitability of HTTP request handling in three popular Python frameworks: Flask, Django, and FastAPI. Through comparative analysis, we evaluate the effectiveness of each framework based on speed, scalability, and ease of implementation, providing recommendations for its use in various development scenarios.

1 Introduction

The Hypertext Transfer Protocol (HTTP) is a fundamental application-layer protocol that facilitates two-way communication between client devices and web servers. It serves as the primary mechanism for data exchange on the World Wide Web and supports most contemporary web-based services. HTTP plays a central role in enabling structured request-response interactions essential for real-time control and system integration in Industry 4.0 and Industrial Internet of Things (IIoT) environments [1]. As industrial systems increasingly rely on microservices and RESTful APIs to link devices, platforms, and control systems, the performance of the underlying web framework becomes a critical factor in system responsiveness and stability. This paper investigates the performance characteristics of three widely adopted Python-based web frameworks—Flask, Django, and FastAPI—in handling HTTP requests under both constant and concurrent loads. The study introduces a realistic testing environment simulating real-world deployments, using a mixed operating system configuration (Windows client and Ubuntu server) and production-grade servers (Gunicorn, Gevent, Daphne, Uvicorn). By measuring execution times across 1,000 iterations for each framework and method (GET and POST), the study provides actionable insights for industrial engineers, system architects, and developers who must select web technologies for time-sensitive applications such as real-time monitoring, telemetry, and IIoT APIs. Python was chosen due to its simplicity, popularity in rapid prototyping, and extensive library support for industrial applications. The findings offer a comparative analysis focused on latency, scalability, and ease of deployment, guiding the making of informed architectural decisions during the design of industrial web services and APIs.

* Corresponding author: pmalatjie@csir.co.za

This study contributes to the field of industrial engineering by providing valuable insights into the selection of time-sensitive applications, such as real-time monitoring and control systems. It does so by simulating real-time operations and assessing the concurrency performance of various frameworks. These findings help engineers make informed decisions when choosing HTTP-based APIs for Industrial Internet of Things (IIoT) systems, with a clear understanding of their performance in terms of speed and overall suitability.

2 Related work

This section explores previous studies that explore the comparation of Flask, Django and Fast API with each other and other widely used web development frameworks in terms of HTTP execution speeds.

2.1 Flask

Flask is a lightweight framework written in Python; it provides a set of libraries for handling web development tasks, such as URL routing and template rendering with Jinja2 [2]. While Flask has been developed for the backend, its performance can be evaluated in various contexts. According to Nuruldelmia Idris, Flask tends to outperform Django in speed due to its lightweight design, allowing it to handle hundreds of requests per second efficiently in production environments. Another study by Eric Qvarnström et al. [4] compares Flask with ASP.NET Core and Express.js in terms of throughput, response time, and computer resource usage. The results indicated that Flask, even when used with Gunicorn, had the slowest response times compared to ASP.NET and Express.js.

2.2 Django

Hasan Rasulov [5] describes Django as a high-level Python web framework that makes it easy to create fast, dynamic websites. It uses the Model-View-Template (MVT) architectural pattern, which is comparable to the Model-View-Controller (MVC) structure, to encourage clean and efficient development. More recently, Dominik Choma et al. [6] conducted a comparative study evaluating the performance of Django against two other frameworks—Express and Spring Boot—based on request processing time and reliability. Using a shared dataset containing employee information, the study found that Spring Boot outperformed both frameworks, demonstrating the fastest request processing time and highest reliability. In contrast, Django ranked lowest, with a higher percentage of incorrectly processed requests.

2.3 Fast API

FastAPI [7] is a modern, fast web framework for developing APIs in Python 3.6 and higher. It uses conventional Python type hints to make API creation more efficient, intuitive, and resilient. According to Bill Lubanovic [8], as of October 2023, FastAPI ranks just behind Django and Flask in popularity, with 64,000 GitHub stars compared to Django's 73,800 and Flask's 64,800. Another study by Priya Bansal [9] evaluated machine learning models for continuous authentication using behavioural biometrics, comparing implementations in FastAPI and Flask. FastAPI delivered faster execution times than the Flask-based application and accelerated development through features such as editor autocompletion and built-in error detection.

3 Methodology

To analyse the frameworks, we implement a basic HTTP server in each framework, capable of handling both GET and POST requests. The performance evaluation metrics include latency, scalability under concurrent requests, ease of implementation, and configuration. The client-server model represents a system where a client device sends requests to a server, which processes them and returns responses. This communication forms the backbone of internet and application interactions, enabling functions like browsing websites, accessing databases, and online gaming. The arrows in the diagram illustrate this exchange: the client initiates an input, the server processes it, and an output is returned. This model ensures efficient data handling and resource distribution across networks. Refer to Fig. ***1*** for client-server connection architecture.

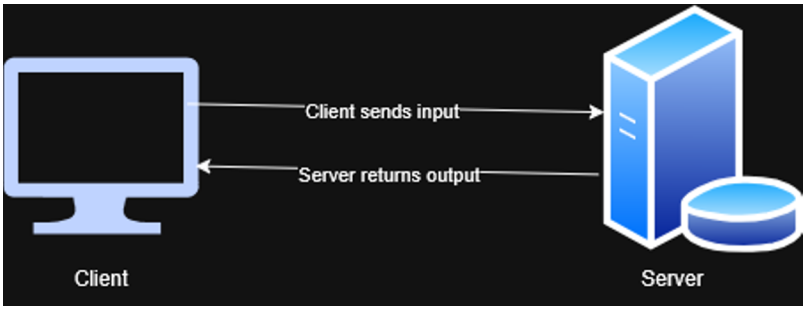


Fig. 1. Client-server connections architecture.

4 Evaluation

4.1 Evaluation architecture

The table below outlines the infrastructure utilised for the tests. The server computer hosts the HTTP methods, which are accessed and interacted with by a separate machine referred to as the client.

Different operating systems are used for the client and server to simulate a real-world environment, where clients typically run on Windows and servers are commonly deployed using Ubuntu.

Table 2. Device architecture

Component	Server	Client
Processor	Intel(R) i7-5820K (12) @ 3.600GHz	Intel(R) Core (TM) i7-7700HQ CPU @ 2.80GHz, 2.81GHz
Ram	31.9 GB	16 GB
GPU	NVIDIA GeForce GTX TITAN X	NVIDIA GeForce GTX 1070
Operating system	Ubuntu 18.04.6 LTS x86 64	Windows 64 bit

Kernel/System type	5.4.0-150-generic / x64	64-bit operating system, x64-based processor

4.2 Experiment setup

In this section, server script and client script are both presented to show the reader, how this experiment is conducted. **Fig. *2*** and **Fig.** 3 depicts the server script for Flask and client script for Django respectively. The complete repository containing all scripts for each framework is available upon request.

```
START

IMPORT necessary modules:
    - os module
    - Flask, request, jsonify from flask
    - datetime from datetime

INITIALIZE Flask application

DEFINE route for GET request at "/get":
    When this route is accessed:
        - Create a JSON response containing:
            - a message: "This is a GET request!"
            - current UTC timestamp
            - process ID of the server
        - Return the response

DEFINE route for POST request at "/post":
    When this route receives a POST:
        - Parse the incoming JSON payload
        - Create a JSON response containing:
            - the received data
            - current UTC timestamp
            - process ID of the server
        - Return the response

IF this script is the main program:
    RUN the Flask server on host 0.0.0.0, port 5000, with debug mode enabled

END
```

Fig. 2. Pseudocode of Flask server implementation for HTTP request handling.

This script shows the Flask server script implementation, designed for benchmarking due to its minimal use of extra time, memory, or resources beyond what is needed to handle requests efficiently (overhead).

```
DEFINE base_url AS "http://localhost:5000"

FUNCTION log_to_csv(file_name, data_list):
    OPEN file in write mode
    INITIALIZE CSV writer
    WRITE header row ["Iteration", "Execution Time (seconds)"]
    FOR each data_row in data_list:
        WRITE data_row to CSV
    CLOSE file
FUNCTION test_get_request():
    INITIALIZE empty list called results
    REPEAT 1000 TIMES:
        RECORD start_time
        TRY:
            SEND GET request to base_url + "/get/" with timeout of 5 seconds
            RECORD end_time
            IF response status code IS NOT 200:
                PRINT error with iteration number and status code
                APPEND [iteration, 0] to results
            ELSE:
                CALCULATE duration = end_time - start_time
                APPEND [iteration, duration] to results
        EXCEPT network or timeout errors:
            PRINT error message with iteration number
            APPEND [iteration, 0] to results
        WAIT for 5 milliseconds
    CALL log_to_csv with "get_django_dephne_times.csv" and results
                |
    PRINT "GET request execution times logged."
```

```
FUNCTION test_post_request():
    SET data payload to {"key": "value"}
    INITIALIZE empty list called results
    REPEAT 1000 TIMES:
        RECORD start_time
        TRY:
            SEND POST request to base_url + "/post/" with JSON data and timeout of 5 seconds
            RECORD end_time
            IF response status code IS NOT 200:
                PRINT error with iteration number and status code
                APPEND [iteration, 0] to results
            ELSE:
                CALCULATE duration = end_time - start_time
                APPEND [iteration, duration] to results
        EXCEPT network or timeout errors:
            PRINT error message with iteration number
            APPEND [iteration, 0] to results
        WAIT for 5 milliseconds
    CALL log_to_csv with "post_django_dephne_times.csv" and results
    PRINT "POST request execution times logged."

MAIN FUNCTION:
    PRINT "Starting GET requests..."
    CALL test_get_request()

    PRINT "Starting POST requests..."
    CALL test_post_request()
```

Fig. 3. Client-side pseudocode for sending HTTP requests to Django server.

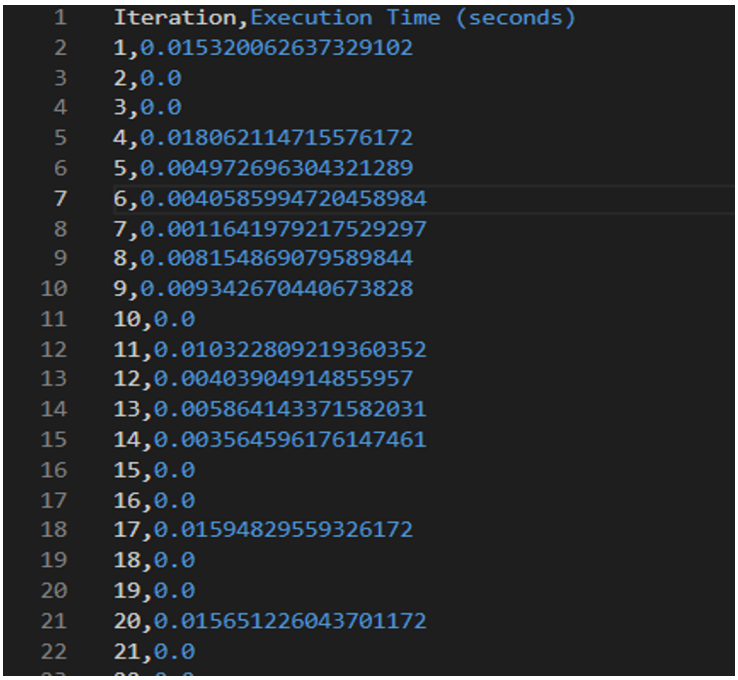
Fig. 3 represents the client-side script used to send GET and POST requests to the server. Stores the response time from the server in a CSV file which is then used for further analysis. This script is valuable for this experiment because it allows the client to send multiple requests to the server in fixed intervals, thanks to time.sleep(0.005) which delays each iteration by 5 milliseconds. Through this implementation, the client sends 200 requests per second to the server, allowing for a controlled and predictable load.

4.3 Evaluation method

The server system implements multiple GET and POST HTTP endpoints for each framework, handling different load conditions. These endpoints process requests initiated by the client system. On the client side, a script measures the execution time for each request over 1,000 iterations, recording individual time instances in a CSV file for further analysis and computation.

Server implementation. Since Flask's built-in server is not suitable for production environments, it becomes necessary to opt for Gunicorn, which delivers the performance and scalability lacking in Flask's development server. Implementing Gunicorn helps bridge the gap between frameworks by enabling Flask to efficiently handle concurrent requests, making it more capable in real-world scenarios.

Fig. 4 illustrate the challenges of using Flask's development server in this experiment. The results show multiple execution times recorded as 0, which might confuse the reader into thinking Flask has a better performance under the development server. However, these zero values do not represent fast response times—instead, they represent Flask's inability to handle a high volume of concurrent requests in its development mode. Consequently, exactly 26% of all GET and POST requests made to the Flask development server were either skipped or dropped entirely, leading to no response and a recorded execution time of zero. Due to this limitation, Flask in development mode is excluded from further analysis in this study. The GitHub repository containing the script used to calculate this percentage is referenced in the Results and Analysis section of this paper.

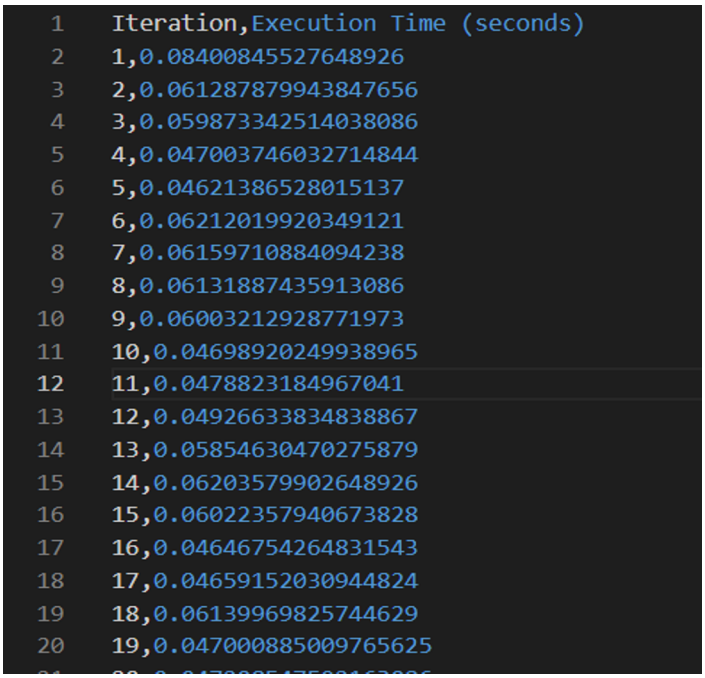


```
Iteration,Execution Time (seconds)
1,0.015320062637329102
2,0.0
3,0.0
4,0.018062114715576172
5,0.004972696304321289
6,0.0040585994720458984
7,0.0011641979217529297
8,0.008154869079589844
9,0.009342670440673828
10,0.0
11,0.010322809219360352
12,0.00403904914855957
13,0.005864143371582031
14,0.003564596176147461
15,0.0
16,0.0
17,0.01594829559326172
18,0.0
19,0.0
20,0.015651226043701172
21,0.0
```

Fig. 4. Execution times using Flask's built-in development server.

The experimental setup for FastAPI also needs to be adjusted to use Uvicorn since FastAPI is an ASGI (Asynchronous Server Gateway Interface) framework. Uvicorn provides FastAPI with the freedom to fully utilise its async capabilities. This implementation will help improve the performance of FastAPI. It is important to note that FastAPI cannot run on its own without an ASGI server like Uvicorn. Since Django version 3.2.25, the latest release used for the experiment as of May 8, 2025, supports ASGI [11], it will be treated as an ASGI framework. Daphne, an ASGI server, is used to improve the framework's performance in terms of stability and handling asynchronous tasks.

The generated CSV files are stored in seconds for each web framework with a thousand iterations showing execution times for each iteration in seconds. **Fig. 5** shows an example of how the CSV files look for each framework.



```
Iteration,Execution Time (seconds)
1,0.08400845527648926
2,0.061287879943847656
3,0.059873342514038086
4,0.047003746032714844
5,0.04621386528015137
6,0.06212019920349121
7,0.06159710884094238
8,0.06131887435913086
9,0.06003212928771973
10,0.04698920249938965
11,0.0478823184967041
12,0.04926633834838867
13,0.05854630470275879
14,0.06203579902648926
15,0.06022357940673828
16,0.04646754264831543
17,0.04659152030944824
18,0.06139969825744629
19,0.047000885009765625
```

Fig. 5. Example of CSV file generated from benchmarking client.

5 Results and analysis

Investigating factors affecting framework performance. We present quantitative and qualitative results that highlight factors affecting the performance of the python framework systems. The insights drawn from the investigation could serve as a general guideline for selecting python framework for software development system. All related scripts are available on GitHub at: https://github.com/GivenM2021/http-requests-web-development-frameworks.git.

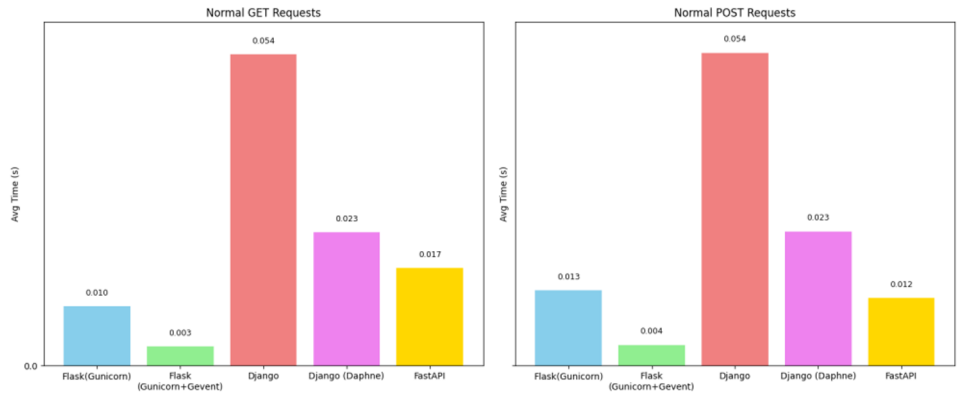


Fig. 6. Bar chart representing average execution times for constant load.

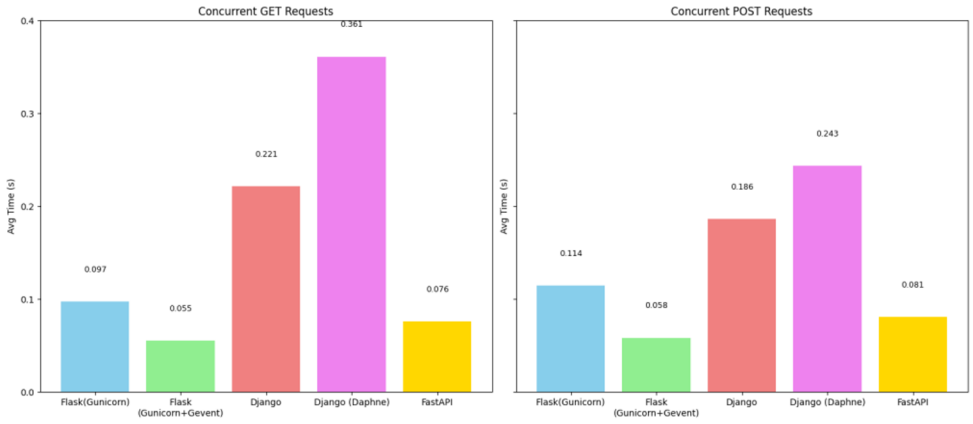


Fig. 7: Bar chart representing average execution times for concurrent load.

Fig. 6 represents the average execution times under normal (constant) load for each framework. Based on the results, we observed that Flask with Gunicorn and FastAPI performed similarly, ranking among the best in terms of speed. Flask's performance improved further when configured with both Gunicorn and Gevent, achieving even lower execution times. In contrast, Django recorded the highest average execution times for both GET and POST requests. However, its performance notably improved when paired with the Daphne.

Fig. 7 shows the average execution times under concurrent (asynchronous) load. In this scenario, FastAPI continued to perform strongly, maintaining low execution times and demonstrating excellent responsiveness. Only Flask configured with Gevent outperformed FastAPI, thanks to enhancements that increased its ability to manage concurrent requests more effectively.

While these findings are based on averages shown in bar charts, it's important to recognize that averages can mask inconsistencies in performance. To provide a clearer picture of stability and variability, we also included a box plot analysis, which better highlights the spread and potential outliers in each framework's performance.

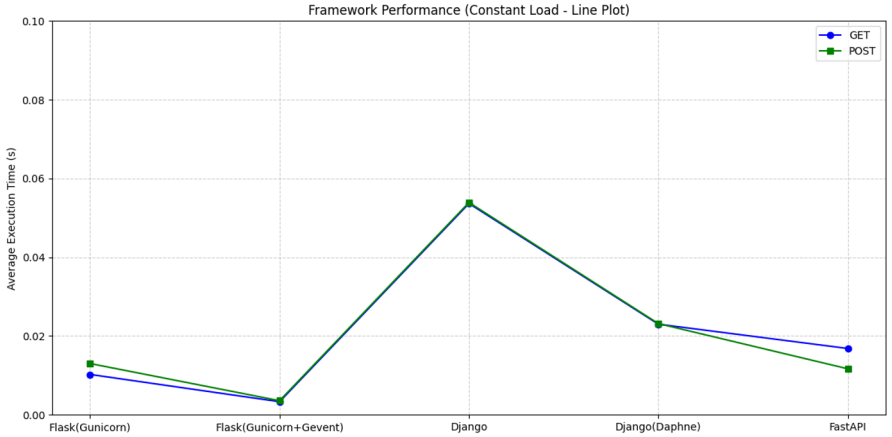


Fig. 8. Line plot representation for each framework under constant load.

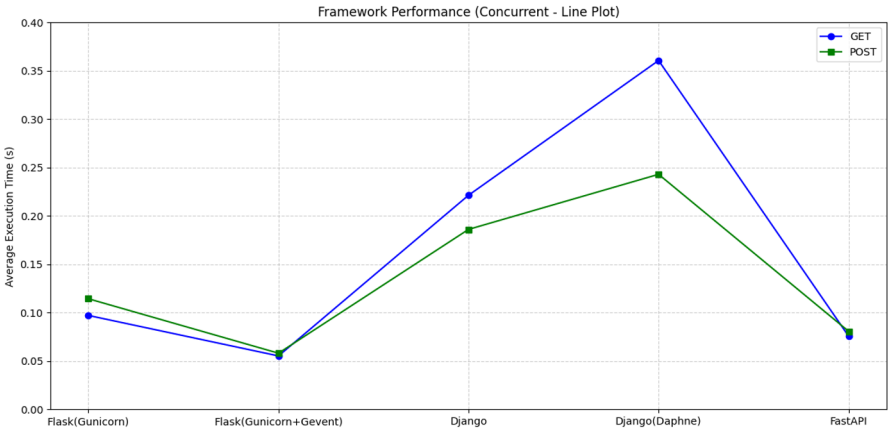


Fig. 9. Line plot representation for each framework under concurrent load.

Fig. 8 shows the performance of each framework under constant load. We observed that Flask with Gunicorn and FastAPI demonstrate low execution times, indicating efficient request handling. Flask configured with both Gunicorn and Gevent outperforms all others, gaining an edge from Gevent's asynchronous capabilities—even in a non-concurrent environment. In contrast, Django shows the highest average execution time, reflecting the weakest performance under normal load. However, its performance improves noticeably when used with the Daphne server.

Fig. 9 illustrates performance under concurrent load. Flask with Gunicorn and Gevent, along with FastAPI, emerged as the top performers, maintaining minimal execution times. In this scenario, Flask slightly outperformed FastAPI. Although Django with Daphne performed better than standard Django, it still showed relatively high execution times—especially for

GET requests—indicating that Django continues to struggle the most under concurrent conditions.

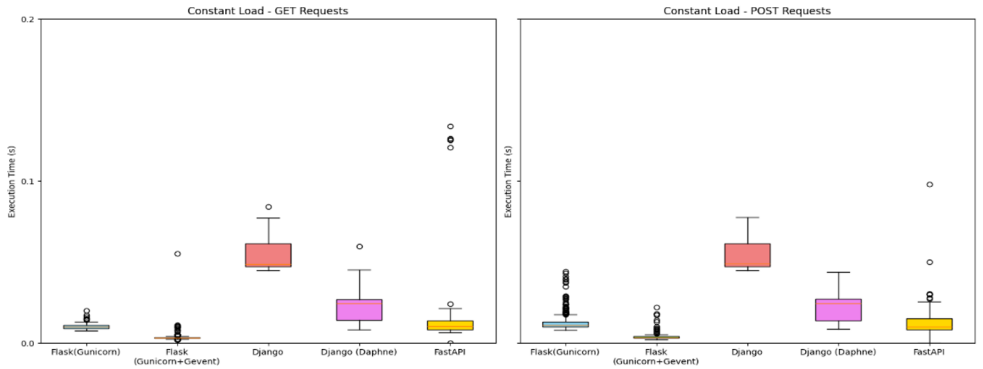


Fig. 10. Plot box diagram representing the performances for each framework under constant load.

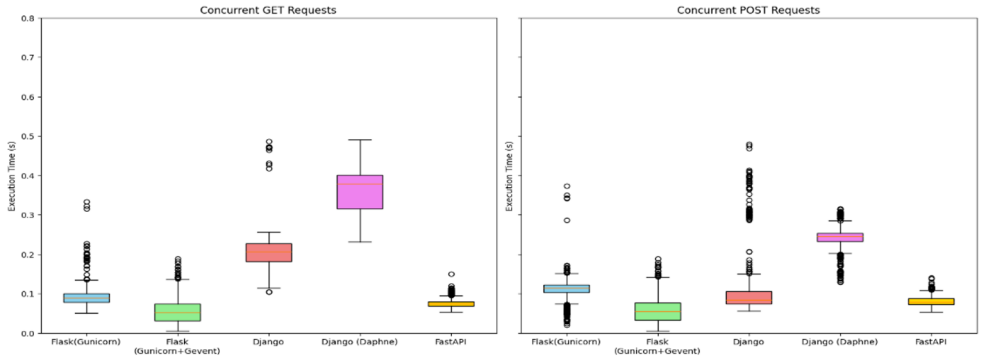


Fig. 1111. Plot box diagram representing the performances for each framework under concurrent.

Fig. 10 highlights clear differences in execution time distributions among various web frameworks under constant conditions. Flask paired with Gunicorn delivers consistently low and stable execution times, indicating dependable performance. On the other hand, Flask with Gevent shows greater variability—occasionally underperforming compared to standalone Flask—but its performance improves considerably under concurrent load (as illustrated in **Fig. 11**), showcasing its strength in handling asynchronous tasks. FastAPI consistently demonstrates efficient performance, maintaining low and steady execution times across both request types and load conditions. Django, by contrast, records the highest execution times and the greatest variability under constant load, suggesting less reliable responsiveness. Nevertheless, when Django is used with the Daphne server, its performance significantly improves in concurrent scenarios. These box plots not only convey average performance but also reveal variability and outliers, providing a more nuanced understanding than simple bar charts.

5.1 Summary of the findings

Django, while powerful and feature-rich, was found to be the slowest in terms of performance among all the frameworks tested. This is likely due to its monolithic design which, although excellent for rapid design and built-in features, introduces performance latency that is not present in lightweight frameworks. Flask with Gunicorn showed moderate performance — better than Django in terms of execution times, but still inferior compared to when implemented with both Gunicorn and Gevent. While it showed stability under normal load, it still had some inconsistencies when tested with concurrent load. Flask with Gunicorn and Gevent displayed lower execution times in all scenarios, particularly under concurrent load conditions, but still displayed slightly higher variability (outliers). This suggests that while Gevent improves Flask execution speeds, it may still be unpredictable. Overall, FastAPI remained the most consistent and efficient across all tests, reinforcing its position as the fastest and most stable choice for high-performance web applications.

Table 1. Performance summary table under normal load.

Framework & Method	Average Time (s)	Improvement	Number Outliers
flask (Gevent) - GET	0.00333	93.82%	61
flask (Gevent) - POST	0.00359	93.34%	16
FastAPI - GET	0.01026	68.44%	6
FastAPI - POST	0.01176	78.18%	9
Flask - GET	0.01026	80.98%	6
Flask – POST	0.01302	75.85%	57
Django - GET	0.02302	0.48%	1
Django - POST	0.02317	0.00%	0
Django (Daphne) – POST	0.02302	57.31%	0
Django (Daphne) - GET	0.02317	57.03%	1

Table 2. Performance summary table under concurrent load.

Framework & Method	Average Time (s)	Improvement	Number of Outliers
flask (Gevent) - GET	0.05525	84.68%	12
flask (Gevent) - POST	0.05807	83.90%	15
FastAPI - GET	0.07562	79.04%	62
FastAPI - POST	0.08071	77.63%	27
Flask - GET	0.09709	73.08%	34
Flask - POST	0.11435	68.30%	97
Django - GET	0.18611	48.41%	156
Django - POST	0.22142	38.62%	32
Django (Daphne) - POST	0.24303	32.63%	96

Django (Daphne) - GET	0.36074	Baseline	0

Table 1 and **Table 2** illustrate the performance of each framework in terms of execution speed, where a higher percentage indicates better efficiency under load. The results reveal that Flask (Gevent) and FastAPI deliver the best performance overall. However, these high-performing frameworks tend to exhibit less stability, as Django consistently demonstrates fewer outliers, indicating more consistent response times despite its slower execution.

6 Conclusion

This paper explores performance variations of different web development frameworks Flask, Django and FastAPI on HTTP requests handling. FastAPI consistently proved to be a go-to-framework in terms of speed under both normal and concurrent loads, showing faster execution times and better efficiency. Flask when implemented with Gunicorn using Gevent workers follows at the second-best performing framework with visible improvements over its default setup which in this case is when it's implemented with Gunicorn. Django displayed higher latency, particularly under concurrent load.

These findings highlight the compromise between ease of use, performance, and scalability, enabling developers to make informed decisions when selecting frameworks for their needs. This study lays the groundwork for future research into the performance of various web development frameworks when used in combination to solve specific types of problems. Further investigation could help identify the most effective framework combinations for different application scenarios.

The authors sincerely acknowledge the Council for Scientific and Industrial Research (CSIR), specifically the Optronic Sensor Systems cluster, for their invaluable support and funding, which enabled this research. The authors also thank Mr. Edwin Magidimisha for guidance in understanding the research process, and Dumisani Kunene for providing hands-on experience and support in carrying out technical tasks and producing results. The data, in the form of execution times for each framework stored in CSV files, is available in the project repository.

References

1. M. Trevisan, D. Giordano, I. Drago and A. S. Khatouni, Measuring HTTP/3: Adoption and Performance, 1-8 (2021). doi: 10.1109/MedComNet52149.2021.9501274
2. M. Copperwaite, C. Leifer, Learning Flask Framework, Packt Publishing Ltd. (2015)
3. E. Chopra, Creating live dashboards for data visualization: Flask vs. React, Int. J. Eng. Res. 8(9), a1–a12 (2021). https://tijer.org/tijer/papers/TIJER2109001.pdf
4. E. Qvarnström, M. Jonsson, A performance comparison on REST-APIs in Express.js, Flask and ASP.NET Core (2022)
5. H. Rasulov, Django: A Comprehensive Framework For Python Web Development,166-169. (2024)
6. D. Choma, K. Chwaleba, M. Dzieńkowski, The efficiency and reliability of backend technologies: Express, Django, and Spring Boot, Informatyka, Automatyka, Pomiary w Gospodarce i Ochronie Środowiska (2023). https://doi.org/10.35784/iapgos.4279
7. V. François, Building Data Science Applications with FastAPI: Develop, manage, and deploy efficient machine learning applications with Python, Packt Publishing Ltd. (2023)

8. B. Lubanovic, FastAPI, O'Reilly Media, Inc. (2023)
9. P. Bansal, A. Ouda, Study on integration of FastAPI and machine learning for continuous authentication of behavioral biometrics, 2022 Int. Symp. Netw. Comput. Commun. (ISNCC), 1–6, IEEE (2022)
10. A. Novikau, Evaluative Comparison of ASGI Web Servers: A Systematic Review, Int. J. Sci. Eng. Appl. 13(3), 30–35 (2024). https://doi.org/10.7753/IJSEA1303.100
11. I. Nuruldelmia, C. Feresa M. Foozy, and P. Shamala. A generic review of web technology: Django and flask, 34-40 (2020)