JOURNAL
COMPUTER SCIENCES INSTITUTE

JCSI 36 (2025) 336–341
Received: 13 May 2025
Accepted: 1 July 2025

Benchmarking the performance of Python web frameworks

Analiza porównawcza wydajności webowych szkieletów programistycznych w języku Python

Bartłomiej Bednarz*, Marek Miłosz

Department of Computer Science, Lublin University of Technology, Nadbystrzycka 36B, 20-618 Lublin, Poland

Abstract
This article presents a comparative performance analysis of three the most popular Python web frameworks - Django (with Django Rest Framework), Flask, and FastAPI - in the context of developing Web APIs. The evaluation focused on key performance metrics such as request throughput for basic GET and POST operations, and response times for database-related CRUD operations. FastAPI consistently outperforms the other frameworks in most scenarios. The study also highlights how ORM usage can incur performance costs, even outside database use. A notable outlier was found in FastAPI when using SQLModel for large-scale data retrieval. Overall, the findings provide practical insights that can assist developers in selecting the most suitable framework.

Keywords: benchmarking; performance; Flask; Django Rest Framework; FastAPI; sqlite; crud

Streszczenie
W artykule przedstawiono analizę porównawczą wydajności trzech najpopularniejszych webowych szkieletów programistycznych w języku Python - Flask, Django (z Django Rest Framework) i FastAPI - w kontekście tworzenia Web API. Ocenę oparto na metrykach, takich jak przepustowość zapytań dla operacji GET i POST oraz czas odpowiedzi dla operacji CRUD na bazie danych. FastAPI przewyższa wydajnością pozostałe szkielety w większości scenariuszy testowych. Badanie wskazuje też, że użycie ORM może wpływać na wydajność, nawet poza operacjami na bazie danych. Zaobserwowano także przypadek znacznie pogorszonej wydajności FastAPI przy użyciu SQLModel podczas pobierania dużych zbiorów danych. Wyniki dostarczają praktycznych wskazówek, które mogą pomóc programistom w wyborze najodpowiedniejszego szkieletu.

Słowa kluczowe: analiza wydajności; Flask; Django Rest Framework; FastAPI; sqlite; crud

*Corresponding author
Email address: s90866@pollub.edu.pl (B. Bednarz)

1. Introduction

Since the launch of the first website by Tim Berners-Lee in 1991, the world of web development has undergone rapid evolution [1]. Today, nearly every company or institution maintains a website or web application, and the quality of these platforms often plays a crucial role in shaping a brand's market position. Fast and reliable applications enhance user satisfaction, which can lead to greater engagement and customer loyalty. Moreover, efficient web applications consume fewer server resources, resulting in lower operational costs. Therefore, selecting the right technology is essential - not only for building applications that meet these demands but also for streamlining the development process. This is where web development frameworks come into play.

A framework is a high-level software solution that promotes code reuse beyond what is achievable through libraries alone. It enables developers to share common features and application logic within a given domain. Frameworks also contribute to higher product quality, as significant portions of the application are prebuilt and thoroughly tested [2]. According to the 2023 Stack Overflow Developer Survey, Python remains one of the most popular programming languages for web development [3]. Modern web architectures are largely based on a client-server model using APIs for communication [4]. In fact, web platforms are increasingly built around Web APIs [5]. This study focuses on three of the most widely used Python web frameworks - Flask, Django, FastAPI - and evaluates their performance in the context of building a typical API [3].

1.1. Literature Review

Prior to conducting the study, a review of selected academic publications from conferences, journals, technical blogs, and student theses was carried out. The focus was primarily on performance and technical evaluations of Python web frameworks, along with common criteria used for performance assessment. Given the wide range of available technologies and the relatively recent emergence of FastAPI, no existing study was found that directly compares the performance of Flask, Django, and FastAPI.

One of the earliest sources [6], published in 2011, presents six popular frameworks from that time, offering only a theoretical comparison by describing their features and use cases. Although Django was included, the age of the article makes its content largely outdated.

Another study [7] reviews different types of Python web frameworks, highlighting eight commonly used options - both full-stack and minimalist. It ultimately

compares Django and Flask based on their features, strengths, and weaknesses. However, this analysis remains theoretical and does not include FastAPI.

In the bachelor's thesis "Comparative study on Python web frameworks: Flask and Django" [8], the author examines the advantages and limitations of both frameworks. The theoretical part discusses various web development technologies, while the practical portion involves building two sample applications: a social networking site using Flask and an e-commerce site using Django. Flask was praised for its simplicity, flexibility, precise control, and ease of learning, while Django was recognized for its rich built-in functionality and scalability - making it suitable for large-scale projects. Nevertheless, the comparison was limited to two frameworks and remained theoretical in nature.

Another bachelor's thesis, "Concurrent benchmark system for web-frameworks on Python" [9], focuses on database operations and JSON (de)serialization performance. Flask, Django, and Pyramid were tested across multiple CRUD scenarios with varying numbers of requests (5, 20, 50, 100, 200, 500). Django delivered the best results for CRUD operations, followed by Pyramid and Flask. For serialization, Flask and Pyramid performed equally well, with Django being the slowest - similar results were observed during deserialization. FastAPI was not included in the study.

In article [10], the authors performed load testing using repeated requests at varying levels of concurrency (1 to 20 simulated users). Hardware and software resource usage - CPU, memory, disk, I/O - were monitored alongside user-perceived response times. Each concurrency level was tested 2,000 times for statistical stability. This paper offers valuable insight into best practices for performance testing.

The article [11] offers a comparative assessment of five widely used web frameworks - Django, Flask, Laravel, Slim, and Spring Boot - focusing in part on performance as measured by Core Web Vitals (LCP, FID, CLS). Laravel demonstrated the fastest load times, while Django provided superior visual stability. Although the study is oriented toward individual learning choices, it provides useful insights for evaluating efficiency of these frameworks in lightweight, real-world scenarios.

Overall, the reviewed literature primarily focuses on theoretical analysis or suggests optimal frameworks for specific use cases. Only three studies conducted performance testing, and none of them included FastAPI in the evaluation.

1.2. Aim and scope of the study

The aim of this study is to conduct a comparative performance analysis of three Python web frameworks - FastAPI, Flask, and Django (with Django Rest Framework) - and to identify their respective strengths and weaknesses in the context of API development. The performance evaluation is based on two key metrics: the speed of handling simple GET and POST requests, measured in requests per second, and the response time for database CRUD operations. The study is intended to provide practical insights for developers choosing the most suitable framework for their projects.

The hypothesis defined for this study is as follows: FastAPI is the most efficient Python web framework.

2. Overview of the selected frameworks

This chapter presents a brief overview of the web frameworks evaluated in the study.

2.1. Flask

Flask is a lightweight, minimalist web framework based on the WSGI protocol [12]. It is well-suited for applications requiring a high level of flexibility. Thanks to its simple and intuitive interface, Flask allows developers to build web applications and APIs quickly and efficiently. It does not impose a specific project structure, making it ideal for small to medium-sized projects. Flask supports a wide range of extensions, such as Flask-SQLAlchemy and Flask-WTF, allowing for additional functionality as needed. It is commonly used in environments where rapid prototyping and full control over the application are important.

2.2. Django (with Django Rest Framework)

Django Rest Framework (DRF) is a robust toolkit that extends Django's capabilities for building APIs [13, 14]. It simplifies the creation of complex APIs by offering built-in tools for data serialization, authentication, and RESTful request handling. DRF integrates easily with databases and includes features such as pagination and filtering. It is especially suitable for large-scale projects that require scalability and integration with other systems, while maintaining readable and maintainable code.

2.3. FastAPI

FastAPI is a modern Python web framework designed with performance and ease of use in mind [15]. Its full support for type hints allows for automatic generation of OpenAPI-compliant documentation. FastAPI is exceptionally fast due to its ASGI foundation and native support for asynchronous programming. It is particularly effective for building microservices and high-performance applications, such as real-time data processing systems. With its clean syntax, FastAPI is beginner-friendly while also offering powerful features for experienced developers.

3. Research Methodology

3.1. Test environment

All performance tests were conducted on a single, dedicated test environment. The technical specifications of this setup, shown in Table 1, ensured reliable measurements free from hardware-related limitations.

Table 1: Specification of the test environment

Component	Specification
Processor	Intel Core i7-8750H
RAM	16 GB DDR4
Storage	1 TB, Samsung SSD 970 EVO
Operating System	Windows 10 Home 64-bit

To minimize performance discrepancies caused by server configuration and to produce results reflective of real-world conditions, all applications were deployed using the production-grade Gunicorn server. The configuration used 2 workers and 2 threads, selected experimentally to guarantee stable performance across all frameworks. The only difference was in the type of worker utilized: gthread for Flask and Django applications, and UvicornWorker for FastAPI, due to compatibility requirements. All applications were also containerized using Docker.

3.2. Test applications

For the purposes of this study, five equivalent applications were developed, each offering the same functionality necessary to carry out the testing scenarios, but implemented using different technology stacks:
- FastAPI + SQLModel,
- FastAPI + SQLite3,
- Flask + SQLAlchemy,
- Flask + SQLite3,
- Django + Django Rest Framework.

All applications used SQLite as the database engine. Flask and FastAPI were each implemented in two variants. One used an ORM (SQLAlchemy for Flask, SQLModel for FastAPI, which is built on top of SQLAlchemy), while the other used the sqlite3 library, providing direct SQL-level access to the database through a DB-API 2.0-compliant interface [16]. Table 2 presents the version numbers of all tools and libraries used in the study.

Table 2: Software versions used

Tool/Library	Version
Python	3.11
FastAPI	0.115.12
Flask	3.1.0
Django	5.2.1
Django Rest Framework	0.1.0
SQLModel	0.0.24
Flask-SQLAlchemy	3.1.1
Gunicorn	23.0.0
Docker Engine	24.0.2

3.3. Testing procedure and scenarios

An automated testing script was created to ensure consistent and reproducible performance measurements. The process began by launching a Docker container with the application under test. After a 20-second period to allow the environment to stabilize, a test database containing 1,500 records was loaded. This was followed by a short pause, after which a warm-up phase began. This involved 20 virtual users performing GET requests for 10 seconds, followed by a series of requests retrieving all records from the database. A 5-second idle period was then enforced to allow the system to settle before benchmarking.

The main testing phase included the following scenarios, executed sequentially with a 3-second delay between each:

- 50 users sending GET requests for 10 seconds, receiving a simple JSON response (GET /hello),
- 100 users sending POST requests with a JSON payload for 15 seconds, receiving the same payload in response (POST /echo),
- 10 users collectively performing 100 requests to retrieve all records (GET /items/),
- 10 users collectively performing 1,000 requests to retrieve a single item (GET /items/{id}),
- 10 users collectively performing 1,000 requests to create new items (POST /items/),
- 10 users collectively performing 1,000 requests to update randomly selected items (PUT /items/{id}),
- 1 user performing 100 delete operations (DELETE /items/{id}).

The open-source tool k6 was used to generate load and collect performance metrics, such as response times and overall stability. Each test run was repeated five times under identical conditions, and the results were averaged to reduce the impact of anomalies and ensure reliability.

During each test, system resource usage was monitored to maintain comparable test conditions. At the beginning of each run, RAM usage ranged between 52% and 55%, while CPU load hovered around 5%. Notably, CPU usage never reached 100% during any test, suggesting that the processor was not a bottleneck and did not negatively impact the results.

4. Results

4.1. Simple GET requests

In the simple GET request test, FastAPI achieved the best results (Figure 1).

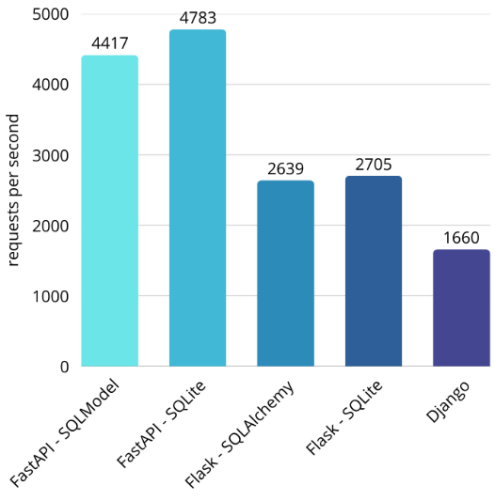

Figure 1: GET request test results.

Its asynchronous architecture enabled the handling of up to 4,783 requests per second with SQLite, and 4,417 req/s when using SQLModel. Flask-based applications reached throughput levels around 2,700 req/s. Django performed the weakest, with a result of only 1,660 req/s - roughly 35% of FastAPI's maximum. This clearly demonstrates the higher overhead imposed by Django's architecture. Interestingly, even though the test scenario

did not involve any interaction with the database, applications using raw SQLite outperformed their ORM-based counterparts. While the difference was negligible in the case of Flask, for FastAPI it exceeded 8%.

4.2. Echo test

FastAPI with SQLite once again achieved the best performance, processing 4,024 requests per second (Figure 2). FastAPI with SQLModel followed with a still-impressive 3,627 req/s. Despite the lack of database interaction, the non-ORM version once again outperformed the ORM-based version - this time by approximately 11%. Flask scored significantly lower, achieving 2,194 req/s with SQLite and 2,143 req/s with SQLAlchemy. The difference between Flask variants was marginal. Django once again ranked lowest at 1,551 req/s, although the gap between Django and Flask was smaller than in the previous test.

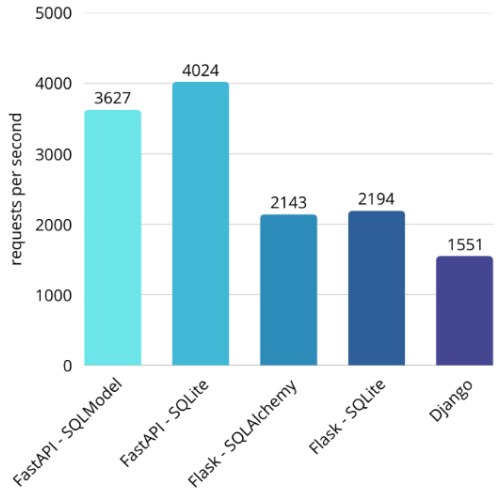

Figure 2: Echo test results.

4.3. Retrieving all records from the database

The test measuring the average response time of a full SELECT query produced notably diverse results (Figure 3). FastAPI with SQLModel was the most striking outlier, with an average response time of 168.8 ms - over two times higher than the median result. The best performance was achieved by Flask with SQLite, at only 28.2 ms, indicating highly efficient data retrieval. Flask with SQLAlchemy (69.2 ms) and FastAPI with SQLite (67.4 ms) yielded very similar results. Django was slightly slower at 75.0 ms. Although most frameworks showed relatively small differences in this scenario, the SQLModel result stands out as a significant negative deviation, suggesting performance bottlenecks under these specific conditions.

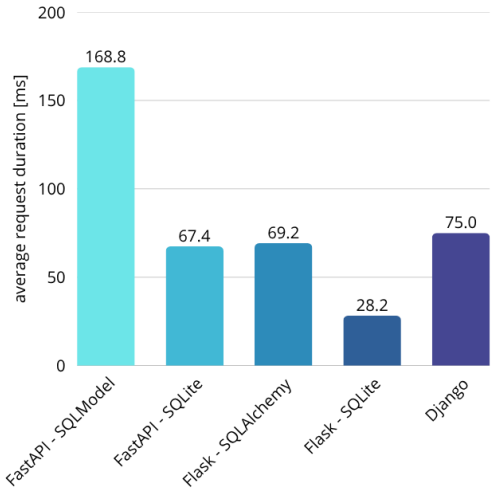

Figure 3: Average response time for fetching all records.

4.4. Retrieving a single record

In the test focused on retrieving a single record from the database (Figure 4), significant performance differences between stacks were observed. FastAPI with SQLite led by a wide margin, with an average response time of 2.5 ms. Flask with SQLite followed with a still impressive, yet two-times slower result of 5.0 ms. FastAPI with SQLModel recorded a response time of 7.6 ms, while Flask with SQLAlchemy came in at 8.5 ms. Django had the slowest response time at 13.4 ms, making it over five times slower than FastAPI with SQLite. Although all recorded times remained below 13.5 ms and might seem negligible in isolation, such differences can become significant in high-load systems or environments requiring high availability.

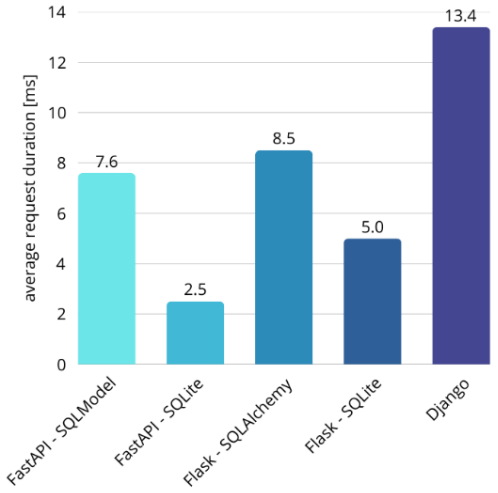

Figure 4: Average response time for fetching a single record.

4.5. Creating a new record

In the test measuring the time to create a new record in the database (Figure 5), moderate variation was observed

across the stacks. FastAPI with SQLite once again delivered the best performance, with an average response time of 78.6 ms. FastAPI with SQLModel followed at 92.4 ms, and Django came in slightly slower at 99.6 ms. Flask-based applications were the slowest in this test: 111.3 ms with SQLite and 116.0 ms with SQLAlchemy, making them the least efficient in this scenario.

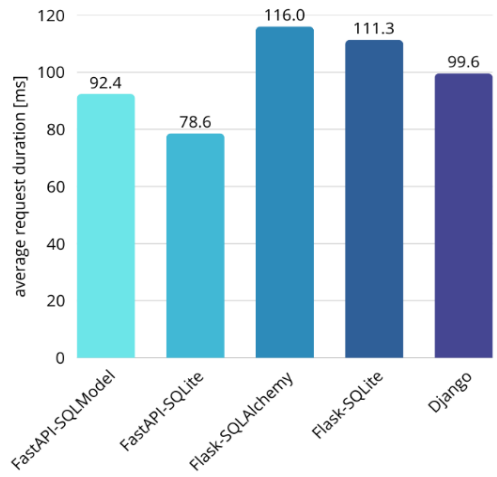

Figure 5: Average response time for create operations.

4.6. Updating a record

Figure 6 shows the average response times recorded during the update test for a single record. FastAPI again achieved the best performance, with 78.4 ms for the SQLite version and 95.3 ms for SQLModel. Flask with SQLAlchemy followed closely at 98.6 ms. Interestingly, Flask with SQLite - which bypasses ORM overhead - was actually slower, matching Django's result of 109.9 ms. This suggests that in this particular scenario, SQLAlchemy's optimizations may have offset the expected simplicity of raw SQL access.

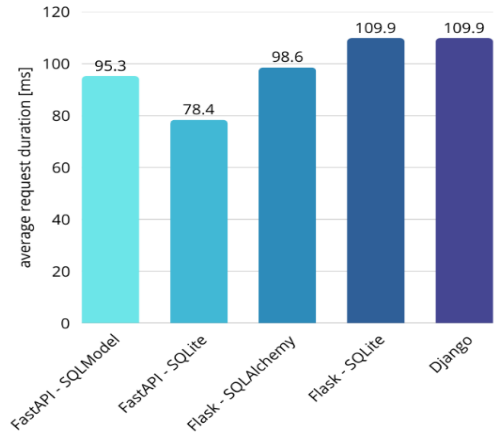

Figure 6: Average response time for update operations.

4.7. Deleting a record

The delete operation test produced consistently low response times across all frameworks (Figure 7), ranging from 9.0 to 11.2 ms. FastAPI with SQLite achieved the lowest response time at 9.0 ms, followed closely by Flask with SQLite at 9.3 ms. FastAPI with SQLModel and Flask with SQLAlchemy posted results of 10.2 ms and 10.4 ms respectively. Django recorded the highest average response time - 11.2 ms - but even this was still well within acceptable performance boundaries.

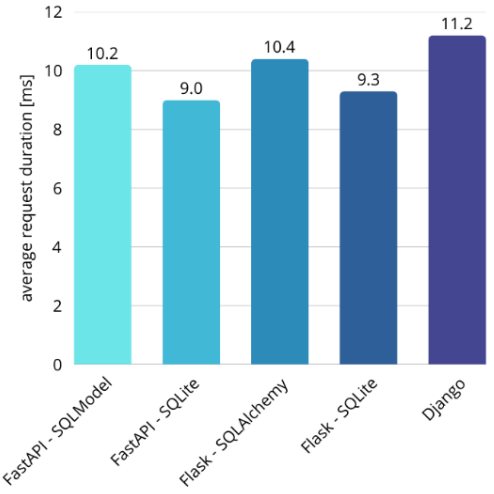

Figure 7: Average response time for delete operations.

5. Conclusion

The conducted study enabled a detailed comparison of the performance of selected Python web frameworks in the context of building Web APIs. The tests covered both typical CRUD operations and basic HTTP requests. To ensure consistency, all applications were deployed in Docker containers and executed under identical conditions using a production-grade Gunicorn server configured with 2 workers and 2 threads. FastAPI used the UvicornWorker to accommodate its asynchronous architecture, while Flask and Django used the gthread worker due to compatibility with WSGI-based applications.

The research hypothesis assumed that FastAPI would be the most efficient of the tested frameworks - and the results confirmed this. FastAPI consistently delivered the best performance in nearly all scenarios, with the exception of the single-record read test. In throughput-oriented benchmarks, it achieved on average 80% higher performance than Flask and nearly three times that of Django. It also performed better in CRUD operations, although the differences in that area were less dramatic. An exception was the create single item test, in which FastAPI achieved results twice as fast as Flask and five times faster than Django. This level of responsiveness can be attributed to FastAPI's lightweight and asynchronous architecture, along with the low overhead associated with simple data models.

A notable exception to the overall trend was observed in the test involving the retrieval of all records from the database. FastAPI with SQLModel recorded an average response time of nearly 170 ms, while the other stacks performed the same task in the 28-75 ms range. This points to inefficiencies in how the query is constructed or

executed when using SQLModel - likely due to the concurrent use of Pydantic for data validation and SQLAlchemy for ORM functionalities. However, this performance issue did not appear in other CRUD operations. Another outlier was Flask with SQLite, which achieved a remarkably low response time of 28.2 ms - more than twice as fast as its FastAPI counterpart (67.4 ms).

Additional insights were drawn from the hello and echo tests. Even though these scenarios did not involve any database operations, variants using ORMs performed worse than their counterparts using raw SQLite. In Flask, the performance gap was marginal, but in FastAPI, differences of 8-11% were observed. This suggests that simply including an ORM in the application may introduce measurable overhead - even in endpoints returning static or near-static JSON responses.

The study also confirmed what is generally expected: applications that interact directly with the database layer tend to be faster than those relying on ORMs - though at the expense of abstraction, maintainability, and long-term scalability.

It should be noted that the server configuration used in this study, while consistent across all applications, was not necessarily optimal for each framework. The parameters were selected to ensure test stability, but they were not specifically tuned for maximum performance. Architectural differences suggest that each framework might benefit from its own tailored configuration, including different numbers of workers, threads, or concurrency models. Future studies could explore such optimizations to determine the peak performance of each framework under ideal conditions.

For developers evaluating which framework to use, the results of this study offer several practical takeaways. If maximum performance and low latency are critical - especially in high-throughput systems with relatively simple business logic - FastAPI is a strong choice. Its asynchronous model and minimal overhead make it particularly well-suited for modern, scalable APIs. Flask remains a good option for projects that do not require asynchronous behavior and value simplicity, rapid prototyping, and clean code. Django, while showing lower raw throughput, continues to be an excellent choice for larger-scale projects that benefit from a complete out-of-the-box ecosystem. Its built-in features - such as authentication, admin interface, and ORM - can significantly speed up development, especially in complex business applications. Ultimately, the choice of framework should be informed not only by benchmark results, but also by the nature of the project, its scale, and the needs of the development team.

References

[1] The birth of the Web. CERN, https://home.cern/science/computing/birth-web, [10.01.2025].

[2] R. A. Santelices, M. Nussbaum, A framework for the development of videogames, Software Practice and Experience 31 (2001) 1091-1107, https://doi.org/10.1002/spe.403.

[3] Stack Overflow Developer Survey 2023, https://survey.stackoverflow.co/2023/#technology-most-popular-technologies, [11.05.2025].

[4] W. W. Eckerson, Three tier client/server architectures: Achieving scalability, performance, and efficiency in client server applications, Open Information Systems 10 (1995) 46-50.

[5] J. Kopecky, P. Fremantle, R. Boakes, A history and future of Web APIs, it - Information Technology 56(3) (2014) 90-97, https://doi.org/10.1515/itit-2013-1035.

[6] Pillars of Python: Six Python Web frameworks compared, https://www.infoworld.com/article/2273961/pillars-of-python-six-python-web-frameworks-compared-2.html, [11.05.2025].

[7] F. Fuior, Introduction in Python frameworks for web development, Romanian Journal of Information Technology and Automatic Control 31(3) (2021) 97–108, https://doi.org/10.33436/v31i3y202108.

[8] D. Ghimire, Comparative study on Python web frameworks: Flask and Django, Bachelor thesis, Metropolia University of Applied Sciences, Helsinki, 2020.

[9] A. Pankiv, Concurrent benchmark system for web-frameworks on Python, Bachelor thesis, Ukrainian Catholic University, Lviv, 2019.

[10] V. Apte, T. Hansen, P. Reeser, Performance comparison of dynamic web platforms, Computer Communications 26(8) (2003) 888-898, https://doi.org/10.1016/s0140-3664(02)00221-9.

[11] L. N. Hyseni, A. Dermaku, Z. Dika, Evaluating Web Frameworks for Personal Learning Decision-Making: A Comparative Analysis, International Journal of Computational and Experimental Science and Engineering 11(2) (2025) 3365-3374, https://doi.org/10.22399/ijcesen.1845.

[12] Welcome to Flask - Flask Documentation, https://flask.palletsprojects.com/en/stable/, [11.05.2025].

[13] The web framework for perfectionist with deadlines, Django, https://www.djangoproject.com, [11.05.2025].

[14] Home - Django REST Framework, https://www.django-rest-framework.org, [11.05.2025].

[15] FastAPI, https://fastapi.tiangolo.com, [11.05.2025].

[16] sqlite3 - DB-API 2.0 interface for SQLite databases, https://docs.python.org/3/library/sqlite3.html, [11.05.2025].