Journal Homepage: www.ijarpr.com ISSN: 3049-0103 (Online)

International Journal of Advance Research Publication and Reviews

Vol 03, Issue 10, pp 32-40, October, 2026

Python Web Frameworks: A Review of Architecture, Performance, Scalability, and Development Features

Dr. D. Richard[1], Francis Xaviar J[2]

[1]Assistant Professor, Department of Information Technology, St. Joseph's College (Autonomous), Affiliated to Bharathidasan University, Tiruchirappalli, Tamil Nadu, India (richard_it2@mail.sjctni.edu)

[2]P.G. Student, Department of Information Technology, M.Sc. Computer Science Programme, St. Joseph's College (Autonomous), Affiliated to Bharathidasan University, Tiruchirappalli, Tamil Nadu, India (francisx424@gmail.com)

Abstract—

Python is one of the most widely used languages for building web applications and back-end services, and its ecosystem offers several mature web frameworks with very different design philosophies. This work presents a review and an empirical comparison of three of the most popular ones: Flask, a lightweight micro-framework; Django, a full-stack "batteries-included" framework; and FastAPI, a modern asynchronous framework built on the ASGI standard. The review part examines the architecture, request-handling model, and built-in development features of each framework. The experimental part implements an identical Task Manager REST API in all three frameworks and evaluates it under controlled conditions using the Locust load-testing tool. Throughput (requests per second), average and 95th-percentile latency, and memory consumption are measured at 50, 100, and 500 concurrent users and with 1, 2, and 4 server workers. Development effort is compared through lines of code, setup time, documentation quality, and available built-in features. The results are analysed to identify which framework best suits which class of application. In the sample results, FastAPI achieved the highest throughput and lowest latency, Django the lowest throughput but the richest set of built-in features, and Flask fell between them with the smallest memory footprint.

Index Terms—*Python, Flask, Django, FastAPI, WSGI, ASGI, web framework, REST API, performance benchmarking, scalability*

I. INTRODUCTION

A. Background

Web frameworks provide reusable building blocks such as URL routing, request parsing, data validation, templating, database access, and security features, so that developers do not need to write these components from scratch. Python is a popular choice for web development because of its readable syntax, large ecosystem, and strong presence in data science and machine learning. The choice of framework, however, strongly influences performance, scalability, maintainability, and developer productivity. The growth of API-driven applications, microservices, and I/O-bound workloads has also made asynchronous programming increasingly important in the Python community [1].

B. Problem Statement

Developers often select a framework because of its popularity or anecdotal reports rather than measured evidence. Published benchmarks differ in hardware, configuration, and level of optimisation, and they rarely combine runtime measurements with an assessment of the development experience. A reproducible, side-by-side comparison of the same application implemented in several frameworks is therefore useful for making an informed choice.

C. Objectives

The objectives of this project are:

To review the architecture and design philosophy of Flask, Django, and FastAPI.

To implement an identical REST API (Task Manager) in the three frameworks.

To measure throughput, latency, and memory usage under increasing numbers of concurrent users.

To evaluate how performance scales as the number of server workers increases.

To compare development features such as validation, documentation, ORM support, authentication, and testing.

To provide practical guidelines for selecting a framework for a given type of project.

D. Scope and Limitations

The study is limited to three frameworks and one type of application, a JSON-based REST API, tested on a single machine. Other frameworks such as Tornado, Pyramid, or Sanic, server-side HTML rendering, and security testing are outside the scope. The results are relative comparisons for the tested configuration and should not be treated as absolute figures for every environment.

II. BACKGROUND AND LITERATURE WORK

A. Web Application Architecture in Python

A web application follows the client–server model: a client sends an HTTP request, and the server returns a response. REST is the dominant architectural style for web APIs and defines constraints such as statelessness, a uniform interface, and resource-based URLs [2]. A framework sits between the web server and the application code and handles the repetitive parts of this cycle.

Python defines standard interfaces between servers and frameworks. The Web Server Gateway Interface (WSGI), specified in PEP 3333, is a synchronous callable interface: each worker process or thread handles one request at a time [3]. Servers such as Gunicorn use a pre-fork worker model to run several worker processes in parallel [4]. The Asynchronous Server Gateway Interface (ASGI) is a successor that supports asynchronous code, WebSockets, and long-lived connections [5]. ASGI servers such as Uvicorn run an event loop, built on Python's asyncio library, that can interleave many I/O-bound requests inside a single process while they wait for databases or network calls [1], [6]. Because the CPython Global Interpreter Lock prevents threads from executing Python bytecode in parallel, CPU-bound scaling is normally achieved with multiple processes [7].

B. Flask

Flask is a micro-framework maintained by the Pallets project. Its core provides routing, request and response handling, and sessions, and it is built on the Werkzeug WSGI toolkit and the Jinja2 template engine [8]. Database access, authentication, and validation are deliberately left to extensions such as Flask-SQLAlchemy, Flask-Login, and marshmallow, which gives developers freedom to choose their own components [8], [9]. This minimal design gives Flask a gentle learning curve and high flexibility. Recent versions allow async view functions, but Flask remains a WSGI framework, so its concurrency model is still based on workers rather than a full event loop [8].

C. Django

Django is a high-level, full-stack framework that follows the Model–View–Template (MVT) pattern. It ships with an object–relational mapper (ORM) with migrations, an automatic administration interface, an authentication system, form handling, a template engine, and built-in protection against common attacks such as cross-site request forgery, cross-site scripting, and SQL injection [10]. The Django REST Framework (DRF) adds serializers, viewsets, and a browsable API for building REST services [11]. Django supports ASGI deployment and asynchronous views, and its async support has been extended progressively to other parts of the framework [10].

D. FastAPI

FastAPI is a modern framework for building APIs, created by Sebastián Ramírez and first released in 2018. It is built on Starlette for the web layer [13] and Pydantic for data validation [14], and it uses standard Python type hints to declare request and response schemas [12]. It runs on ASGI, supports both async and regular endpoint functions, provides dependency injection, and automatically generates OpenAPI documentation with interactive Swagger UI and ReDoc pages [12], [15], [16]. FastAPI does not include an ORM or an admin interface; SQLAlchemy is commonly used for database access [16], [17].

E. Comparative Overview

Table I summarises the main architectural differences between the three frameworks.

Aspect	Flask	Django	FastAPI
Design philosophy	Micro-framework; minimal core	Full-stack; batteries included	API-first; driven by type hints
Server interface	WSGI	WSGI and ASGI	ASGI
Core architecture	Blueprints and extensions	MVT, apps, and settings	Starlette routing, Pydantic models, dependency injection
Data validation	Via extensions (e.g., marshmallow)	Forms and DRF serializers	Built in through Pydantic
ORM	Via extension (SQLAlchemy)	Built-in Django ORM	Not included (SQLAlchemy or others)
Admin interface	Not included	Built in	Not included
API documentation	Via extensions	Via DRF or third-party tools	Automatic (OpenAPI, Swagger UI, ReDoc)
Async support	Limited (async views on WSGI)	Supported (async views, ASGI)	Native
Typical use	Small services, prototypes	Database-driven, content-rich sites	High-concurrency APIs, microservices

F. Related Work and Research Gap

The TechEmpower Framework Benchmarks are the best-known public comparison of web frameworks and include tests such as JSON serialisation, single and multiple database queries, and plaintext responses [18]. Their results, however, depend on hardware, configuration, and implementation-level optimisations, and highly tuned entries may not resemble typical application code. Many informal comparisons also rely on a simple “hello world” endpoint, which measures little more than framework overhead. This work addresses that gap by combining a realistic database-backed CRUD API, a controlled load-testing procedure, scalability tests with different worker counts, and a structured comparison of development features.

III. METHODOLOGY

A. Research Approach

The study is a comparative experimental investigation with two parts: a qualitative review of framework architecture and features based on official documentation, and a quantitative benchmark of identical applications. Functionality, database schema, hardware, and server configuration are kept the same for all frameworks, and only one variable (concurrency or worker count) is changed at a time.

B. Application Under Test

A simple Task Manager REST API is implemented in each framework. Each task has the fields id, title, description, completed (Boolean), and created_at. Table II lists the endpoints.

Method	Endpoint	Description	Purpose
GET	/ping	Returns a static JSON message	Baseline overhead
GET	/tasks	Returns the list of tasks	Database read

Method	Endpoint	Description	Purpose
POST	/tasks	Creates a new task	Write & validation
GET	/tasks/{id}	Returns one task	Single-record read
PUT	/tasks/{id}	Updates a task	Database update
DELETE	/tasks/{id}	Deletes a task	Database delete

C. System Architecture

Fig. 1 shows the layered architecture used to test each framework. A load generator (Locust) plays the role of the client and sends HTTP requests to an application server (Gunicorn for the WSGI frameworks, Uvicorn for the ASGI framework), which forwards them to the framework layer. The framework layer performs routing, request validation, and response serialisation, and it delegates data access to an ORM/data-access layer that reads from and writes to a PostgreSQL database. The same layered structure is kept identical across Flask, Django, and FastAPI; only the internal implementation of the framework and data-access layers differs, which is what the experiment measures.

D. Technology Stack

SQLite is used during development and PostgreSQL for the final measurements. Table III lists the components used with each framework. Exact versions must be recorded because performance can change between releases.

Component	Flask	Django	FastAPI
Framework version	Flask 3.1.x	Django 5.2.x, DRF 3.16.x	FastAPI 0.115.x
Data layer	Flask-SQLAlchemy	Django ORM with DRF	SQLAlchemy with Pydantic
Database access	Synchronous	Synchronous	Asynchronous (asyncpg)
App server	Gunicorn	Gunicorn	Uvicorn
Database	PostgreSQL	PostgreSQL	PostgreSQL

E. Experimental Environment

Table IV outlines the hardware and software specifications of the environment used for conducting the benchmarks.

Parameter	Value
Processor (model, cores)	Intel Core i5, 4 cores / 8 threads
Memory (RAM)	16 GB
Operating system	Ubuntu 22.04 LTS
Python version	3.12
PostgreSQL version	16
Load-testing tool	Locust 2.x [19]
Load generator location	Same machine as the server
Containerisation	Docker not used

F. Test Procedure

1) Implement the API in each framework with the same endpoints, schema, and validation rules.

2) Verify functional equivalence with a common automated test suite (for example, pytest with an HTTP client).

3) Deploy each application with its production server (Gunicorn or Uvicorn) [4], [6] connected to PostgreSQL, seeded with the same number of records (for example, 1,000).

4) Warm up each application for 30–60 seconds before measuring.

5) Run Locust [19] at 50, 100, and 500 concurrent users for a fixed duration (for example, 2–5 minutes) and repeat each run three times, reporting the mean.

6) Repeat the test with 1, 2, and 4 server workers at a fixed concurrency level.

7) Record memory usage per worker using a tool such as psutil or docker stats.

8) Record development metrics: lines of code (using cloc), setup time, and a checklist of built-in features.

G. Evaluation Metrics

Table V details the metrics collected across performance, scalability, resource usage, and development effort.

Category	Metric	Measurement Method
Performance	Requests per second (RPS)	Reported by Locust
Performance	Avg. and p95 latency (ms)	Reported by Locust
Reliability	Failure rate (%)	Failed total requests
Scalability	Throughput vs. workers	Runs with 1, 2, 4 workers
Resource use	Memory per worker (MB)	psutil or docker stats
Development	LOC, setup time, docs, features	stopwatch, cloc, checklist

H. Feature Comparison

Table VI summarizes the development and architectural features across all three frameworks.

Feature	Flask	Django	FastAPI
Built-in ORM	No	Yes	No
Admin panel	No	Yes	No
Authentication	Via extensions	Built in	Security utilities; libraries
Data validation	Via extensions	Forms/DRF serializers	Automatic via Pydantic
Automatic API docs	No	Via DRF/third-party	Yes (Swagger, ReDoc)
Async support	Limited	Partial and growing	Native
DB migrations	Via Alembic	Built in	Via Alembic

Testing support	Built-in test client	Test framework/client	TestClient (Starlette)
Learning curve	Gentle	Moderate to steep	Moderate
Community	Large, mature	Very large, mature	Large, fast-growing

I. Threats to Validity

Several factors can affect the fairness of the comparison. If the load generator runs on the same machine as the server, it competes for CPU time. Differences in ORM efficiency, connection pooling, and the use of synchronous or asynchronous database drivers can influence results more than the framework itself. SQLite and PostgreSQL behave differently under concurrent writes. To reduce these effects, all tests use identical configurations, each run is repeated, software versions are reported, and conclusions are limited to the tested setup.

IV. RESULTS AND DISCUSSION

This section presents the qualitative comparison, the measurement tables, and a discussion. Tables VII and VIII report results with 4 server workers and no think time between requests; Table IX reports throughput and memory versus worker count.

Note: the values in Tables VII, VIII, IX, and X are illustrative sample values for demonstration and must be replaced with the author's own measured results before submission.

A. Baseline Endpoint Performance

Table VII reports the baseline /ping endpoint, which isolates framework overhead.

Fwk	Users	RPS	Avg (ms)	p95 (ms)	Fail %
Flask	50	2400	20.8	38	0.0
Flask	100	2450	40.8	72	0.0
Flask	500	2400	208.3	390	0.1
Django	50	1500	33.3	62	0.0
Django	100	1520	65.8	120	0.0
Django	500	1480	337.8	640	0.2
FastAPI	50	5200	9.6	18	0.0
FastAPI	100	5350	18.7	33	0.0
FastAPI	500	5100	98.0	180	0.0

B. Database Endpoint Performance

Table VIII reports performance on the database-backed /tasks endpoints (read and write).

Fwk	Users	RPS	Avg (ms)	p95 (ms)	Fail %
Flask	50	620	80.6	140	0.0
Flask	100	640	156.3	260	0.0
Flask	500	610	819.7	1350	0.4

Fwk	Users	RPS	Avg (ms)	p95 (ms)	Fail %
Django	50	410	122.0	210	0.0
Django	100	420	238.1	390	0.1
Django	500	395	1266.0	2100	0.9
FastAPI	50	1350	37.0	65	0.0
FastAPI	100	1420	70.4	120	0.0
FastAPI	500	1380	362.3	610	0.1

C. Scalability Results

Table IX details throughput and memory usage per worker across varying worker counts.

Fwk	Workers	RPS	Speed-up	Mem/worker (MB)
Flask	1	175	1.00	48
Flask	2	335	1.91	49
Flask	4	640	3.66	49
Django	1	115	1.00	72
Django	2	220	1.91	73
Django	4	420	3.65	74
FastAPI	1	390	1.00	58
FastAPI	2	750	1.92	59
FastAPI	4	1420	3.64	60

D. Development Effort

Table X presents comparative development metrics including lines of code, setup time, and documentation quality.

Metric	Flask	Django	FastAPI
Lines of code (cloc)	95	140	85
Setup time (min)	10	20	12
Number of files	4	9	4
Documentation quality (1-5)	4	5	5
Learning curve (1-5, steepest)	2	4	3

E. Discussion

Throughput and latency. Framework design suggests that FastAPI should perform well on I/O-bound workloads, because its ASGI event loop can serve

many concurrent requests in one process while they wait for the database, particularly at high concurrency such as 500 users and when an asynchronous database driver is used [1], [12]. Flask and Django running under synchronous Gunicorn workers handle a limited number of requests per worker, so their throughput at high concurrency depends mainly on the number of workers [3], [4]. Django adds middleware and DRF serialisation layers, which can increase per-request overhead compared with a minimal Flask endpoint but also provide more built-in functionality [10], [11]. These expectations should be compared with actual measurements; if they differ, blocking database calls inside async code, ORM overhead, and worker settings should be checked.

Scalability. Throughput is expected to grow with the number of workers until the available CPU cores or the database become the bottleneck, after which the gains flatten. Memory per worker reflects the amount of framework code loaded, so Django, which loads more built-in components, is likely to have a higher baseline than Flask.

Development experience. Django provides the most functionality out of the box (ORM, migrations, admin, authentication), which reduces the code needed for standard CRUD applications. FastAPI reduces boilerplate through automatic validation and interactive documentation generated from type hints. Flask offers the most freedom, but the developer must select and integrate extensions, which adds decisions and configuration.

In the sample results, FastAPI delivered about 2.2 times the throughput of Flask on both /ping (about 5,200 versus 2,400 requests per second at 50 users) and the database-backed /tasks endpoints (1,350 versus 620), while Django was about one third slower than Flask on /tasks (410 versus 620). Throughput scaled almost linearly with workers, reaching a speed-up of about 3.6 to 3.7 with four workers for all frameworks, and memory per worker was lowest for Flask (about 49 MB) and highest for Django (about 74 MB). Latency and failure rates rose sharply for the synchronous frameworks at 500 users, which supports the expectation that the event-loop model handles high concurrency better for I/O-bound work.

F. Framework Selection Guidelines

These guidelines are general recommendations, not strict rules. Team experience, existing infrastructure, and project requirements should also influence the final decision. Table XI lists suggested frameworks for various project scenarios.

Scenario	Fwk	Reason
High-concurrency, I/O-bound APIs and microservices	FastAPI	Native async, automatic validation and documentation
Database-driven websites needing admin/auth	Django	Built-in ORM, admin, authentication, security
Small services, prototypes, custom architectures	Flask	Minimal core and maximum flexibility
Rapid CRUD and content-management apps	Django	Admin interface and migrations reduce dev. time
Serving ML models through an API	FastAPI	Typed request schemas, async request handling

V. CONCLUSION

This work reviewed and compared three leading Python web frameworks, Flask, Django, and FastAPI, from the perspectives of architecture, performance, scalability, and development features. The review showed that the frameworks represent three different trade-offs: Flask favours minimalism and flexibility, Django favours completeness and rapid development, and FastAPI favours modern asynchronous design, type-driven validation, and automatic documentation. An identical Task Manager REST API was implemented in each framework and evaluated with a controlled load-testing procedure. In the sample results, FastAPI led on I/O-bound throughput, Django offered the most built-in features at the cost of throughput and memory, and Flask was the lightest and most flexible, which matches the expected pattern.

Overall, no single framework is best in every situation. The most suitable choice depends on the workload, the required built-in features, the scalability targets, and the experience of the development team.

A. Limitations

The study covers three frameworks, one application type, and one test machine. Results depend on the hardware, software versions, database configuration, and the synchronous or asynchronous approach chosen for database access.

B. Future Work

Include additional frameworks such as Sanic, Tornado, or Litestar.

Evaluate asynchronous database drivers and the async ORM features of Django.

Add caching (for example, Redis), authentication overhead, and CPU-bound workloads to the tests.

Repeat the experiments in containerised or orchestrated deployments and on newer Python releases.

REFERENCES

[1] Python Software Foundation, "asyncio — Asynchronous I/O," Python 3 Documentation. [Online]. Available: https://docs.python.org/3/library/asyncio.html

[2] R. T. Fielding, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, Dept. Inf. Comput. Sci., Univ. California, Irvine, CA, USA, 2000.

[3] P. J. Eby, "PEP 3333 — Python Web Server Gateway Interface v1.0.1," Python Enhancement Proposals, 2010. [Online]. Available: https://peps.python.org/pep-3333/

[4] "Gunicorn documentation." [Online]. Available: https://docs.gunicorn.org/

[5] "ASGI (Asynchronous Server Gateway Interface) specification." [Online]. Available: https://asgi.readthedocs.io/

[6] "Uvicorn documentation." [Online]. Available: https://www.uvicorn.org/

[7] Python Software Foundation, "Glossary: global interpreter lock," Python 3 Documentation. [Online]. Available: https://docs.python.org/3/glossary.html#term-global-interpreter-lock

[8] Pallets Projects, "Flask documentation." [Online]. Available: https://flask.palletsprojects.com/

[9] M. Grinberg, Flask Web Development: Developing Web Applications with Python, 2nd ed. Sebastopol, CA, USA: O'Reilly Media, 2018.

[10] Django Software Foundation, "Django documentation." [Online]. Available: https://docs.djangoproject.com/

[11] "Django REST framework." [Online]. Available: https://www.django-rest-framework.org/

[12] S. Ramírez, "FastAPI documentation." [Online]. Available: https://fastapi.tiangolo.com/

[13] "Starlette." [Online]. Available: https://www.starlette.io/

[14] "Pydantic documentation." [Online]. Available: https://docs.pydantic.dev/

[15] OpenAPI Initiative, "OpenAPI Specification." [Online]. Available: https://spec.openapis.org/oas/latest.html

[16] B. Lubanovic, FastAPI: Modern Python Web Development. Sebastopol, CA, USA: O'Reilly Media, 2023.

[17] "SQLAlchemy documentation." [Online]. Available: https://docs.sqlalchemy.org/

[18] TechEmpower, "TechEmpower Framework Benchmarks." [Online]. Available: https://www.techempower.com/benchmarks/

[19] "Locust documentation." [Online]. Available: https://docs.locust.io/