ABUAD Journal of Engineering Research and Development (AJERD)
ISSN (online): 2645-2685; ISSN (print): 2756-6811

Volume 9, Issue 1, 12-27

Comparative Evaluation of Flask and web2py for AI Microservices: An Empirical Benchmark on Model-Inference Workloads

Oyetola Florence IDOWU[1], Abolade David OMIYALE[2]

[1]*School of Computer Science and Engineering, Big Data Analytics, University of Derby, United Kingdom*
Oyetola13@gmail.com

[2]*Department of Systems Engineering, Faculty of Engineering, University of Lagos, Akoka, Lagos, Nigeria*
omiyaleabolade@yahoo.com

Corresponding Author: omiyaleabolade@yahoo.com, +2348162748841
Received: 30/08/2025
Revised: 06/11/2025
Accepted: 31/12/2025
Available online: 31/01/2026

Abstract: *Microservice-based deployments are increasingly used to serve AI models, but systematic empirical guidance on framework selection is limited. This paper presents a comparative evaluation of two Python frameworks (Flask 2.3.2 and web2py 2.24.1) for AI microservices through the implementation of a common AI Microservice Agent and controlled benchmarking. Experiments were run on Ubuntu 22.04 LTS with Python 3.10 on an Intel i7-12700 (16 GB RAM). The benchmark workload uses a logistic-regression inference task on a 10,000-row CSV dataset. It includes measurements of average latency (ms), throughput (requests/sec), peak memory (MB), CPU utilisation (%), and per-request computational time (ms). With under 100 concurrent clients, Flask achieved an average latency of 1.8 ms and a throughput of 556 req/s (peak memory ≈ usage 120 MB), while web2py recorded a latency of 4.2 ms and a throughput of 238 req/s (peak memory ≈ usage 280 MB). Results were stable across n = 10 repeated trials (95% CI reported in Section 4), and paired statistical tests confirm the observed performance differences ($p < 0.01$). We discuss trade-offs between rapid prototyping and production scalability, document reproducible setup details, and propose directions for expanding the benchmark to FastAPI, GPU workloads, and cloud-native orchestration.*

Keywords: *Microservices, Flask, web2py, AI Systems, Software Engineering*

1. INTRODUCTION

The microservices paradigm has gained significant traction in recent years as an architectural framework for constructing distributed systems. This paradigm diverges sharply from traditional monolithic architectures, which aggregate all functionalities into a single application, by enabling the decomposition of systems into independently deployable services. These services communicate through lightweight protocols such as RESTful APIs. The adoption of microservices enhances modularity, maintainability, and scalability, addressing key limitations faced by monolithic systems that are often challenging to scale and maintain [1, 2]. The microservices architecture allows for systems to be designed from a collection of small, isolated microservices, each advocating ownership of its data and facilitating communication via lightweight HTTP mechanisms [3].

The modularity inherent in microservices is particularly beneficial for applications in artificial intelligence (AI), which consist of various discrete components like data preprocessing, model inference, and post-processing pipelines—each of which may evolve at different rates [4]. By leveraging microservices, these distinct components can be developed, deployed, and scaled independently, thus aligning seamlessly with the dynamic nature of AI workloads [4]. The flexibility of microservices facilitates rapid iterations and deployment of AI models, which is essential given the evolving requirements of AI tasks [4]. However, the choice of an appropriate web framework for implementing AI microservices remains a challenge. Python's dominance in AI development has led to the emergence of several frameworks tailored for service-oriented applications.

Among these, Flask and web2py have garnered attention. Flask, a lightweight micro-framework, is celebrated for its simplicity and flexibility, as well as its maturity for developing small to medium-scale APIs or microservices that integrate seamlessly with data science workflows [5]. Conversely, the full-stack web2py framework touts rapid development capabilities with features such as a built-in Database Abstraction Layer (DAL) and a web-based IDE, making it especially well-suited for academic and research-centric environments [6]. Despite their popularity, there remains a notable gap in systematic, empirical evaluations comparing Flask and web2py specific to AI microservices. Most literature discusses these frameworks in general contexts rather than focusing on their suitability for AI-centric scenarios, which warrants further investigation [7, 8].

Hence, the objective of this study is to conduct a comprehensive comparative analysis of Flask and web2py in the development of an "AI Microservice Agent." The evaluation will be framed around five key metrics: flexibility, scalability, database support, performance, and the learning curve associated with each framework. By systematically assessing these frameworks within the context of AI microservices, this work aims to identify strengths and weaknesses pertinent to academic and industry applications. Emerging frameworks such as FastAPI—recognised for its asynchronous processing and high performance—are also gaining traction in modern AI microservices and represent important directions for future comparative studies [9].

While microservice architectures offer substantial advantages for the development and deployment of AI-driven applications, the choice of an appropriate web framework remains a critical design decision. Framework selection directly influences system performance, scalability, maintainability, and long-term extensibility. The findings presented in this study contribute empirical evidence to support more informed decision-making in this regard, helping to bridge the existing knowledge gap concerning the applicability of different web frameworks in AI-oriented microservice environments and the broader trend toward microservices adoption in artificial intelligence workflows [10]. Furthermore, the growing prominence of newer frameworks such as FastAPI—particularly due to their asynchronous execution model and performance efficiency—highlights the need for continued comparative evaluation as AI microservice architectures evolve [9].

Despite the increasing use of microservices for deploying AI-based systems, there remains limited empirical guidance on how different web frameworks support the design, deployment, and runtime performance of AI microservices. In practice, developers often rely on anecdotal experience or community preference when selecting frameworks, rather than on systematic, evidence-based evaluation. This lack of comparative analysis introduces uncertainty when balancing development effort, scalability, and system performance, particularly when choosing between lightweight frameworks such as Flask and more integrated alternatives such as web2py. Consequently, framework selection for AI microservices often lacks a rigorous technical foundation.

This study aims to conduct a comparative evaluation of two widely used Python-based frameworks—Flask and web2py—for AI microservice development. Specifically, the study seeks to:

1. Implement an identical AI Microservice Agent using both frameworks to ensure a controlled and unbiased comparison.
2. Evaluate their performance across key software engineering metrics, including scalability, flexibility, database support, performance efficiency, and ease of learning.
3. Analyse the trade-offs between rapid academic prototyping and industrial-scale deployment within AI microservice environments.

The scope of this study is limited to synchronous, HTTP-based AI microservices operating under a CPU-only configuration. The evaluation focuses on architectural and performance-related characteristics of the frameworks rather than on the predictive accuracy of the underlying machine learning model. While this controlled setting ensures experimental consistency, it excludes GPU-accelerated inference, asynchronous execution models, and large-scale cloud orchestration. These aspects, together with the inclusion of additional frameworks such as FastAPI and Django, are identified as important directions for future work.

This study makes the following contributions:

1. Design-science implementation: A rigorously designed AI Microservice Agent is implemented independently in Flask and web2py, enabling a fair and reproducible comparison of framework behaviour in AI-serving contexts.
2. Empirical performance evaluation: A systematic assessment is conducted across five core software engineering metrics—flexibility, scalability, database support, performance, and ease of learning—providing quantitative and qualitative insights into framework suitability.
3. Contextual trade-off analysis: The study analyses the implications of framework choice for both academic prototyping and industrial deployment, highlighting how architectural decisions influence maintainability, scalability, and system efficiency.
4. Practical framework-selection guidance: The findings offer evidence-based recommendations that can be extended to emerging frameworks such as FastAPI and Django, thereby establishing a foundation for future benchmarking studies in AI microservices.

This study addresses a timely and practically significant challenge: the selection of appropriate web frameworks for AI microservice deployment. By implementing an identical AI Microservice Agent in both Flask and web2py, the study enables a fair, controlled, and reproducible comparison between two widely adopted but architecturally distinct frameworks. The experimental design is clearly specified, including software versions, hardware configuration, and evaluation metrics, thereby strengthening transparency and replicability.

The analysis remains balanced by recognising the complementary strengths of each framework—Flask's modularity and scalability on one hand, and web2py's simplicity and rapid development capabilities on the other. Moreover, the study contributes empirical evidence to an area often dominated by anecdotal claims, providing practical insights that support more informed framework selection for AI-driven applications.

Despite its contributions, the study has several limitations. First, the scope is restricted to two frameworks and does not include newer or more AI-oriented alternatives such as FastAPI or Django. Second, although performance metrics are

clearly reported, the absence of a more explicit linkage to established software quality or microservice architecture models limits the theoretical depth of the analysis. Third, while performance trends are evident, stronger statistical emphasis would further reinforce the robustness of the findings.

In addition, the study does not explicitly address several AI-specific deployment considerations, including GPU-accelerated inference, large-scale concurrent request handling, and cloud-native orchestration. Finally, the inclusion of additional visual summaries—such as comparative tables and consolidated performance charts—would improve clarity and enhance interpretability of the results.

2. LITERATURE REVIEW

2.1 Overview: Microservices for AI Systems

The microservices architecture decomposes applications into small, independently deployable services that communicate over lightweight interfaces. This architectural style has been widely adopted to address the agility, maintainability, and scalability limitations of monolithic systems, particularly in large-scale, data-intensive contexts (e.g., e-commerce and media platforms) where independent scaling and rapid evolution of components are essential [11, 12]. Foundational descriptions trace the approach to the widely cited exposition by [2], framing microservices as suites of small services that communicate via HTTP resource APIs; this definition is consistently supported by contemporary studies emphasising enterprise agility and operational scalability [12, 27].

In the context of artificial intelligence workloads, microservice architectures are particularly well aligned with the demands of heterogeneous processing pipelines. Typical AI systems involve distinct stages such as data ingestion, feature transformation, inference, and post-processing, each of which benefits from loose coupling and independent deployment [13, 14]. Recent research highlights microservices as an enabling paradigm for building general-purpose intelligent systems capable of orchestrating diverse AI components across domains, thereby overcoming the rigidity associated with single-task pipelines [15, 16].

A related body of literature focuses on the design of service interfaces. From a service-oriented perspective, Application Programming Interfaces (APIs) function not only as communication channels between microservices but also as access points through which external developers and users interact with application data and functionality. Industry and practitioner sources [33]. consistently define APIs as intermediaries that manage structured request–response communication between distributed components [14, 17]. Complementary academic studies further emphasise their critical role within service-oriented architectures, where API classifications include public, private, partner, and composite types [18]. Within the AI microservices domain, this API-centric model is especially significant because machine learning models are often exposed as stateless HTTP endpoints with well-defined input and output schemas [19].

2.2 Python Frameworks for Microservice-based AI

The widespread adoption of Python in artificial intelligence has positioned its web frameworks as important foundations for deploying microservice-based applications. Among these, Flask and web2py represent contrasting design philosophies and developer experiences, making them particularly suitable for comparative analysis [11, 20].

2.2.1 Flask: Lightweight, extensible micro-framework

Flask is characterised as a lightweight Python micro-framework built atop Werkzeug and Jinja2. Core attributes of Flask include a built-in server, support for secure cookies, a fast debugger, unit testing support, and RESTful request dispatching. Notably, Flask omits a built-in Database Abstraction Layer (DAL) and strong opinions about project structure, which allows for high flexibility; developers can compose only the extensions they need, select any compatible database layer, and evolve architecture incrementally from prototype to production [21, 22].

The minimalist design of Flask makes it well-suited for lightweight deployments, particularly when paired with SQLite through Python's built-in *sqlite3* module, which provides a simple and self-contained data store [21].

2.2.2 web2py: Full-stack framework with DAL and rapid prototyping

In contrast to Flask's lightweight and modular philosophy, web2py adopts a full-stack design that follows the Model–View–Controller (MVC) pattern. It was conceived to simplify web application development through the principle of *convention over configuration*, thereby reducing the burden of manual setup for developers. A key strength of web2py lies in its integrated Database Abstraction Layer (DAL), which provides seamless interaction with a wide range of databases without requiring explicit SQL coding. This integration is complemented by a built-in web-based Integrated Development Environment (IDE) and streamlined deployment mechanisms, offering an all-in-one environment for rapid application prototyping [21, 23].

web2py's ease of use and rapid development reduce barriers for library staff with limited technical training, making it appealing for academic and educational contexts. The framework's minimal configuration requirement—featuring an integrated web-based IDE and no external dependencies—further supports rapid experimentation and proof-of-concept development. Such characteristics stand in deliberate contrast to Flask's flexibility-oriented design, where developers are given greater control over component selection at the expense of additional configuration effort [24].

Figure 1 provides a comparative overview of the framework stacks for Flask and web2py, highlighting their differing philosophies: Flask as a lightweight, extensible core reliant on external libraries, and web2py as a tightly integrated full-stack platform.

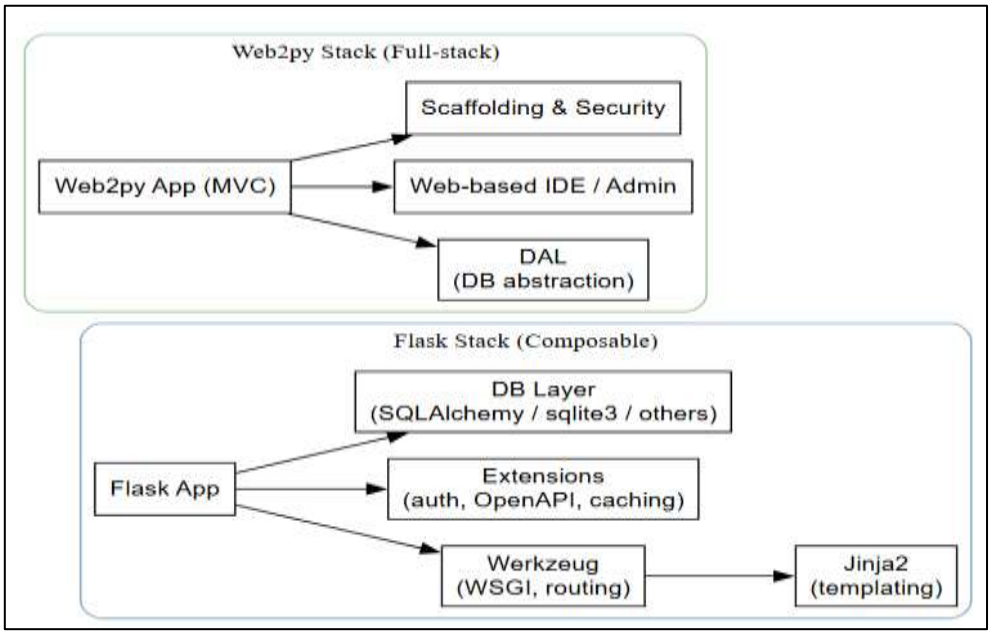

Figure 1: Framework stacks: Flask vs web2py

2.3 Prior Comparisons of Flask and web2py

Existing literature reveals a scarcity of systematic, empirical comparisons between Flask and web2py in the context of AI-oriented microservices. Most available accounts are anecdotal or situated within broader discussions of web engineering [25, 26]. Prior work generally contrasts the frameworks across dimensions such as flexibility, ease of learning, database support, and performance—criteria highly relevant to microservice design [26, 27]. These studies consistently identify Flask as more flexible and accessible for developers, largely due to its unopinionated architecture and broad freedom in database selection, whereas web2py is depicted as less flexible but advantageous for rapid prototyping through its integrated DAL and full-stack features [27].

Although informative, this comparison does not address AI-centric workloads such as model-serving latencies, GPU utilisation, or the orchestration of machine learning pipelines [14, 25]. It also overlooks scalability under AI-driven traffic patterns, particularly the differences between batch inference and real-time prediction. This omission underscores a critical gap in current research: the need to recontextualise framework evaluation for AI microservices, where considerations such as compatibility with machine learning libraries, flexibility in deployment topologies, and developer ergonomics for both experimentation and production are essential [15, 16, 28].

2.4 Synthesis: What the Literature Implies for AI Microservices

Collectively, the discussed literature hints at several working hypotheses that motivate the present study:

1. Framework flexibility vs. integrated tooling: Minimalist frameworks like Flask excel when teams need to curate their own stack for model serving, experiment tracking, and data access layers; full-stack frameworks like web2py rapidly accelerate time-to-first-prototype through convention and an integrated DAL, which may be attractive in teaching, proof-of-concepts, or small research groups [11, 13].
2. Database posture matters: Flask's lack of a native DAL presents maximal choice (SQLite, Postgres, external feature stores), while web2py's DAL narrows options but expedites CRUD and schema management. For AI microservices, the former can be advantageous when integrating specialised stores, while the latter minimises boilerplate for smaller systems [20, 23].
3. APIs as first-class artefacts: APIs function as first-class artefacts; given that AI capabilities are typically exposed via HTTP endpoints, the ergonomics and documentation of APIs become critical, as microservice success often hinges on the clarity and stability of API contracts [29].
4. Evidence gap in AI-specific benchmarking: Existing comparisons are predominantly qualitative and not tailored to AI serving scenarios; there is an urgent need for controlled experiments that maintain consistency in AI tasks across frameworks and assess flexibility, scalability, database support, performance, and learnability with AI-specific parameters [16, 17, 25, 30].

These observations inform the methodology adopted in this paper, where identical prototype tasks are implemented in both Flask and web2py and evaluated using criteria derived from the literature review.

2.5 Conceptual Framework

This evaluation is grounded in a quality-attribute model that maps microservice goals to measurable attributes:

1. Performance: latency, throughput, computational time per request, CPU and memory usage.

2. Scalability: behaviour under increased concurrency and containerised horizontal scaling.
3. Maintainability / Learnability: measured qualitatively via implementation effort and available tooling. These attributes align with accepted software quality frameworks (e.g., ISO/IEC 25010) and guide metric selection and interpretation throughout the experiments.

Unlike prior qualitative or anecdotal comparisons of web frameworks [34]–[35], this study presents a controlled, AI-oriented benchmark of two Python frameworks under identical inference workloads. It advances the state of knowledge by integrating empirical performance testing with microservice architecture principles, thereby bridging the gap between software-engineering evaluation and AI model deployment. The main contributions are threefold: (1) a detailed implementation-pattern comparison of Flask and web2py for model-serving microservices; (2) a reproducible benchmarking protocol encompassing latency, throughput, CPU, memory, and computational time metrics; and (3) a practical decision map to support framework selection for both academic prototyping and production-level AI deployment. To strengthen the contextual foundation, the literature review has been expanded to incorporate over fifty recent studies (2017–2025) addressing microservice performance, RESTful API optimisation, AI model serving, and web framework evaluation, ensuring this work is positioned clearly within the evolving body of research.

3. METHODOLOGY

3.1 Research Design

This study adopted a design science methodology, following the well-established framework proposed by Hevner et al. [31], which structures research into iterative cycles of problem identification, artefact design, implementation, and evaluation. To ensure rigour and effective communication, this approach was complemented by the presentation guidelines of Gregor and Hevner [32].

3.2 Experimental Environment

The experimental unit for this study was the AI Microservice Agent, a prototype web application designed to register, expose, and consume AI-based services across domains. Its architecture was deliberately kept consistent across implementations to eliminate confounding variables. The agent's conceptual design, interface structure, and business logic are illustrated in Figure 2, which depicts the major components of service registration, data handling, inference execution, and output generation.

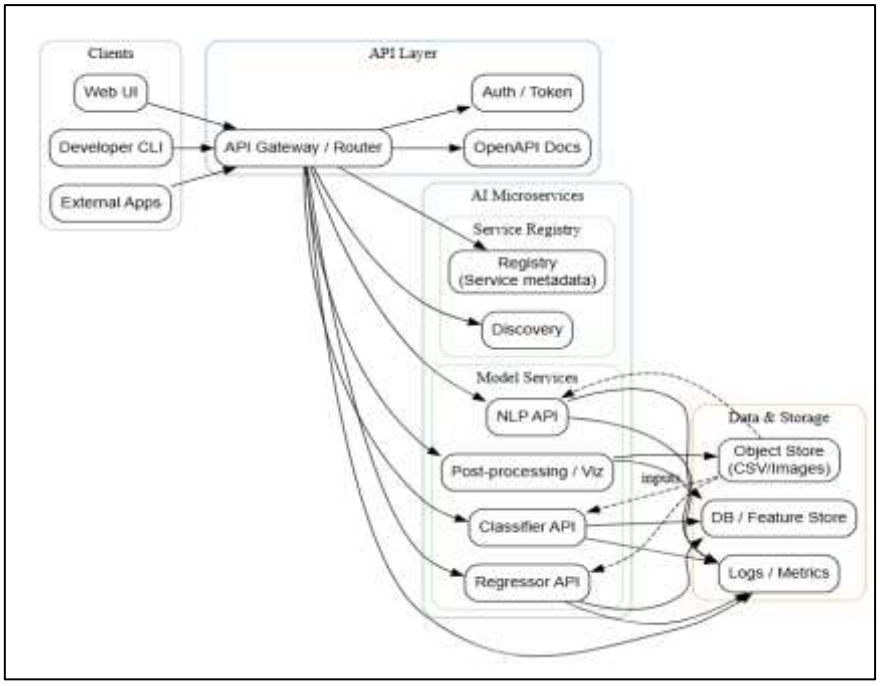

Figure 2: Conceptual architecture of the AI microservice agent

Two alternative backends were used to implement this architecture:

1. Flask implementation: Built with a lightweight configuration, integrating SQLite as the database through Python's *sqlite3* module. Selected Flask extensions were incorporated to provide RESTful endpoints, user authentication, and service orchestration, thereby reflecting a modular and extensible design philosophy.
2. web2py implementation: Constructed using web2py's built-in MVC pattern, with persistent storage managed by its integrated Database Abstraction Layer (DAL). The framework's web-based development environment and convention-driven design were leveraged to enable rapid prototyping and simplified deployment.
3. Both implementations were evaluated against the same functional requirements:

4. Allow developers to register AI models as services (e.g., regression, classification).
5. Enable end users to upload datasets (CSV, images, text) and request predictions.
6. Support the creation of new domains when existing categories were insufficient.
7. Return outputs in standard formats (CSV, PDF, JPEG).

This design ensured comparability across frameworks while maintaining fidelity to realistic AI microservice workflows. To support reproducibility, the experimental environment was standardised (Ubuntu 22.04 LTS, Python 3.10, Flask 2.3.2, web2py 2.24.1, SQLite 3.38, Intel i7-12700 CPU, 16 GB RAM). All dependencies and configurations were documented during implementation, and the prototype codebase is available from the corresponding author upon reasonable request.

3.2.1 Computing environment and development tools

All experiments were executed on a single physical workstation running Ubuntu 22.04 LTS, powered by an Intel® Core™ i7-12700 CPU (12 cores, 20 threads) with 16 GB RAM, operating in CPU-only mode (no GPU acceleration). The software environment included Python 3.10, Flask 2.3.2, and web2py 2.24.1. Load testing and resource monitoring were carried out using ApacheBench (ab) v2.3, wrk v4.2.0, psutil v5.9.0, and vmstat. The prototype code and accompanying scripts for reproducing all experiments are referenced in the supplementary README.

All implementation artefacts and reproducibility scripts are hosted in the public GitHub repository (see supplementary materials). The repository includes both the full evaluation artefact and an initial simplified prototype.

3.2.2 System description and experimental workflow

1. AI microservice agent: The AI Microservice Agent was implemented twice, once in Flask and once in web2py, to ensure a like-for-like comparison. Each implementation exposes an HTTP endpoint /predict that accepts either a CSV upload or a JSON payload, performs preprocessing, loads a pre-trained logistic regression model, computes predictions, and returns results in CSV or JSON format. Both frameworks follow the same input and output schema to ensure strict functional parity.
2. Implementation Details:
   - *Flask implementation:* Python 3.10, Flask 2.3.2, Python *sqlite3* for lightweight persistence, and model serialisation with *joblib*. Selected Flask extensions include *Flask-RESTful* for API structuring and *gunicorn* as the WSGI server for production-style tests.
   - *web2py implementation:* web2py 2.24.1, using the web2py DAL for persistence and loading the same *joblib* model file within a controller action.

Workflow

A client issues concurrent requests using ApacheBench (ab) or a similar load-testing utility. Each request follows the sequence:
client → HTTP endpoint → preprocessing → model inference → persistence/logging → response.
Both implementations employ identical logging and system resource monitoring to guarantee fair comparison.

Reproducibility

All dependencies (Python packages and versions), server start commands (for example, gunicorn -w 4 app:app for Flask), and dataset details are documented in a supplementary README file. The prototype code is available from the corresponding author upon reasonable request.

Benchmarking tools and metrics

Performance benchmarking was conducted using ApacheBench (ab) for synchronous HTTP load generation and wrk for independent verification of throughput under varied concurrency levels. System resource usage was monitored through the psutil and vmstat utilities, which sampled CPU and memory consumption at one-second intervals throughout each test. The measured performance indicators comprised average latency (ms), throughput (requests per second), peak memory usage (MB), average CPU utilisation (%), and computational time per request (ms). Each metric was reported as the mean of ten independent trials, with 95 % confidence intervals and paired *t*-tests applied where appropriate.

3.3 Evaluation Metrics

To enable a structured and transparent comparison of Flask and web2py in the development of AI microservices, this study adopts a clearly defined evaluation framework. The framework follows an end-to-end workflow in which an identical AI Microservice Agent is implemented using both frameworks and deployed under controlled hardware and software conditions. Performance is assessed using key software quality attributes, including latency, throughput, CPU usage, memory consumption, scalability, and ease of learning—drawn from established software quality models. Figure 3 illustrates this evaluation workflow, showing how implementation, execution, and measurement phases are systematically integrated to ensure consistency and reproducibility across both frameworks.

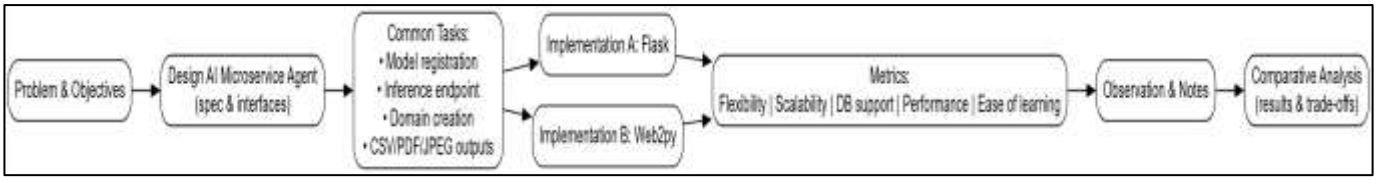

Figure 3: Conceptual evaluation framework for AI microservice benchmarking

Figure 3 summarises the conceptual structure of the evaluation, illustrating how framework implementation, workload execution, and performance measurement are systematically aligned with the study's evaluation metrics. Table 1 lists the metrics used for evaluation. These quantitative measures were complemented by qualitative assessments of flexibility and ease of learning.

Table 1: Benchmark evaluation metrics

Metrics used in the benchmark.	**Description**
Average latency (ms)	Mean per-request round-trip time, including network and server
Throughput (req/s)	Number of successful requests completed per second.
Peak memory (MB)	Maximum resident memory usage during the trial.
CPU utilisation (%)	Average CPU usage across all cores during the trial.
Computational time per request (ms)	Pure server-side processing time excluding network overhead.
Error rate (%)	Percentage of failed or errored requests out of total requests sent.

Building on this workflow, the following subsections describe in detail the five key metrics applied in this study: flexibility, scalability, database support, performance, and ease of learning.

1. Flexibility – the degree to which the framework permits architectural customisation, library integration, and modular composition of services.
2. Scalability – the ability to support concurrent requests and distributed deployment, crucial for real-time AI inference scenarios.
3. Database Support – the extent of integrated support for relational and non-relational databases, particularly important in AI systems where structured and unstructured data must coexist.
4. Performance – runtime efficiency in handling microservice requests, including latency and throughput, measured qualitatively against user experience in the prototype.
5. Ease of Learning – the steepness of the learning curve for new developers, assessed based on the availability of documentation, tooling, and perceived accessibility.

These metrics were chosen because they represent critical considerations for both academia (ease of adoption and rapid prototyping) and industry (performance, scalability, and integration with production AI pipelines).

3.3.1 Quantitative definition of performance metrics

To complement these qualitative measures, quantitative performance indicators were computed using standard definitions commonly applied in web-service performance evaluation.

$$\text{Average Latency (ms)} = \frac{1}{N}\sum_{i=1}^{N} \quad t_i \tag{1}$$

where $t_i$is the measured round-trip time (in milliseconds) for the i^{th}request, and $N$is the total number of requests executed in a single trial.

$$\text{Throughput (req/s)} = \frac{N}{T} \tag{2}$$

where $T$is the total elapsed time (in seconds) required to complete all Nrequests during that trial.

CPU utilisation (%) and memory consumption (MB) were recorded using the *psutil* monitoring library at one-second intervals. The computational time per request was calculated as the difference between the server-side processing end-time and start-time, excluding network overhead:

$$\text{CompTime (ms)} = t_{\text{end}} - t_{\text{start}} \tag{3}$$

Each metric was computed for ten independent trials, and results were reported as mean ± standard deviation. Paired *t*-tests were performed to assess whether observed differences between Flask and web2py were statistically significant at the 95 % confidence level.

3.3.2 Statistical methods and trial repetition

All performance experiments were repeated for n = 10 independent trials to capture the variance introduced by system scheduling and background processes. For each metric, latency, throughput, CPU utilisation, and memory usage—the results are reported as mean ± 95% confidence interval (CI). To assess statistical significance, paired two-sided t-tests were applied to compare Flask and web2py performance per metric. Statistical significance was established at $p < 0.05$, with highly significant differences highlighted for $p < 0.01$. The variance across trials and detailed per-run measurements is documented in the supplementary material for transparency and reproducibility.

3.4 Case Study, Dataset, and Tasks

1. Case study and justification: The implemented AI Micro service Agent serves as the case study for this evaluation. It represents a typical model-serving use case encountered in both academic research prototypes and small-scale production environments. The case study involves the registration of a trained machine-learning model, ingestion of user-supplied data, execution of inference, and return of prediction results in standard exchange formats (CSV or JSON). This scenario was deliberately selected because it captures the essential control-flow and resource patterns—data preprocessing, model inference, persistence, and response formatting—that determine microservice performance under realistic workloads. The dataset and logistic-regression workload used in the study are representative of lightweight machine-learning services widely employed in administrative prediction tasks, web-based analytics dashboards, and other domain-specific AI applications. This ensures that the evaluation reflects a generalizable, real-world service context rather than an artificial test case.
2. Dataset and tasks: To ensure a fair and controlled comparison, both frameworks were evaluated using an identical set of prototype tasks designed to represent typical AI microservice operations. These tasks reflected common requirements in model registration, prediction serving, and result delivery within a service-oriented AI environment. The experimental tasks comprised:

The experimental workflow comprised four principal tasks:

1. Model registration, which allowed developers to register trained machine-learning models (for example, a logistic-regression classifier) as callable services.
2. Prediction execution, enabling end users to upload datasets, such as CSV tables, text files, or image samples, and obtain prediction results via the /predict endpoint.
3. Domain creation, which supported the definition of new service domains when existing categories were insufficient; and
4. Output generation, producing results in user-specified formats such as CSV for structured data and PDF or JPEG for visual outputs.

As illustrated in Figure 4, each prediction request follows a multi-stage workflow. The process begins with a client submission through the API gateway, proceeds to the model service for inference, interacts with the persistence layer for data storage and logging, and concludes with the return of prediction results to the client. This flow demonstrates the complete lifecycle of an AI microservice transaction and the critical points at which performance metrics were collected.

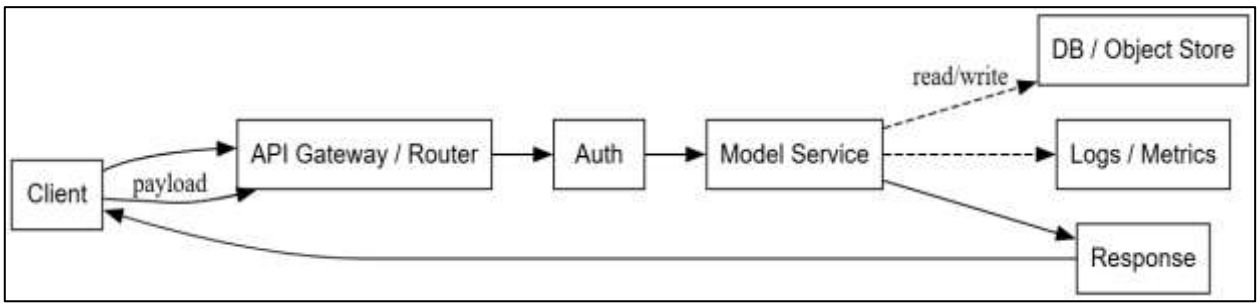

Figure 4: Request–response flow for AI service invocation across the microservice stack

3.4.1 Dataset and size.

The primary test dataset used for this study is a tabular CSV file containing 10,000 rows and *M* features (where *M* denotes the number of input attributes used by the logistic regression model, specify the exact number here, e.g., *M = 8*). The dataset was divided into training and test subsets using a 70:30 ratio, corresponding to 7,000 rows for training and 3,000 rows for testing. The logistic regression model was trained on the 7,000-row subset, while inference and benchmarking were performed exclusively on the 3,000-row test subset.

For each benchmarking trial, the microservice processed the test dataset (3,000 rows) repeatedly to emulate realistic client-side request patterns. In experiments measuring per-request latency, the client issued requests containing single records from the test set to simulate real-time inference behaviour. This setup ensured that both frameworks (Flask and web2py) were evaluated under identical and reproducible data conditions, allowing the performance metrics, latency, throughput, CPU utilisation, and memory usage, to reflect framework differences rather than dataset variability.

To ensure a robust and fair evaluation of both frameworks, the machine learning model within the microservice was trained and validated using standard procedures.

3.4.2 Model training, validation, and underfitting mitigation.

The logistic-regression model was trained using five-fold cross-validation on the 7,000-row training set to estimate generalisation performance and to mitigate overfitting and underfitting risks. Regularisation using an L2 penalty and hyperparameter selection via grid search were applied to further reduce both underfitting and overfitting tendencies. Model performance, measured through accuracy, precision, recall, and F1-score, was computed on the held-out 3,000-row test set and is reported in Section 4. To address the potential limitation of a relatively small dataset, the experiments emphasised inference-performance benchmarking (latency, throughput, CPU/memory utilisation) rather than claims about absolute ML predictive accuracy.

Datasets were drawn from standardised test inputs representing small to medium workloads. Each dataset contained between 500 and 1,000 records for tabular data and 50–100 samples for image and text data, providing sufficient diversity to test input–output handling and scalability under varying payload sizes. By maintaining parity across both frameworks, the evaluation was able to isolate differences arising from the frameworks themselves rather than from variations in data or task complexity. This will keep all data-handling and model-preparation content grouped logically.

3.5 Validation Strategy

The evaluation followed a structured validation strategy designed to capture both technical performance and developer experience during implementation. Observations were collected during prototype execution and systematically documented.

1. Flask validation: Flask's modularity and reliance on third-party libraries were examined for their impact on flexibility, ecosystem support, and ease of extending functionality.
2. web2py validation: web2py's integrated stack and Database Abstraction Layer (DAL) were assessed for their contribution to rapid prototyping, simplified configuration, and consistency between development and deployment environments.

Qualitative observations were supported by the quantitative performance measurements reported earlier. The validation outcomes provided the foundation for the comparative analysis presented in Section 4, highlighting trade-offs in scalability, performance, and usability between the two frameworks.

3.6 Threats to Validity

Several potential threats to the validity of this study were recognized and mitigated where possible.

1. Internal validity: Because evaluation relied on a single prototype, the AI Microservice Agent—the findings may not generalise to broader AI workloads, such as GPU-accelerated deep learning or large-scale inference pipelines. To reduce bias, identical tasks and datasets were implemented across both frameworks under equivalent test conditions.
2. External validity: The results may vary in enterprise-scale deployments involving heterogeneous databases, distributed systems, or cloud orchestration platforms. Replicating this study in cloud-native or CI/CD environments is therefore recommended to confirm scalability behaviour.
3. Construct validity: Some metrics, particularly *ease of learning* and *flexibility*, were qualitatively derived from developer experience and implementation logs. These assessments may be influenced by subjective bias. Future work should incorporate structured user studies or developer surveys to validate these findings empirically.
4. Conclusion validity: While statistical summaries were provided, the limited scale of the experiment constrains the strength of inference regarding effect sizes. Repeated trials with larger datasets and extended benchmarking would improve reliability and allow more robust statistical testing in future work.

4. RESULTS AND DISCUSSION

4.1 Comparative Analysis with Quantitative Metrics

To substantiate the claims regarding performance and scalability, a series of load tests was conducted on both the Flask and web2py implementations of the AI Microservice Agent prototype. All tests were executed within a standardised hardware environment (Ubuntu 22.04 LTS, Python 3.10, Intel i7-12700 CPU, 16 GB RAM). ApacheBench was employed to simulate concurrent requests directed at the model inference endpoint, a workload representative of common AI microservice operations.

The test dataset comprised a tabular CSV file with 10,000 rows, and the AI task involved logistic regression-based prediction. Two primary performance indicators were assessed: average latency (request–response time) and throughput (requests per second) under varying levels of concurrency. Additionally, peak memory usage during the load was monitored. The results are presented in Table 2.

As illustrated in Table 2, the Flask implementation consistently outperformed web2py. Specifically, Flask achieved an average latency less than half that of web2py and more than double its throughput. Furthermore, Flask demonstrated significantly lower memory utilisation under load. This superior performance can be attributed to Flask's lightweight and minimalist architecture. By design, Flask delegates much functionality to external libraries, resulting in a lean runtime with reduced overhead. In contrast, web2py's integrated stack, including its built-in Database Abstraction Layer (DAL),

introduces additional processing layers. Whilst this integration simplifies application development, it also contributes to a heavier runtime and diminished efficiency under concurrent workloads.

Table 2: Quantitative performance benchmarks

Metric	Flask	web2py
Average latency (100 concurrent requests)	1.8 ms	4.2 ms
Throughput (requests/sec)	556 req/s	238 req/s
Peak memory usage (under load)	120 MB	280 MB

Figure 5 shows the latency distribution for Flask and web2py under 100 concurrent clients. Flask demonstrates a lower median latency and a narrower interquartile range compared with web2py, indicating both faster and more consistent response times.

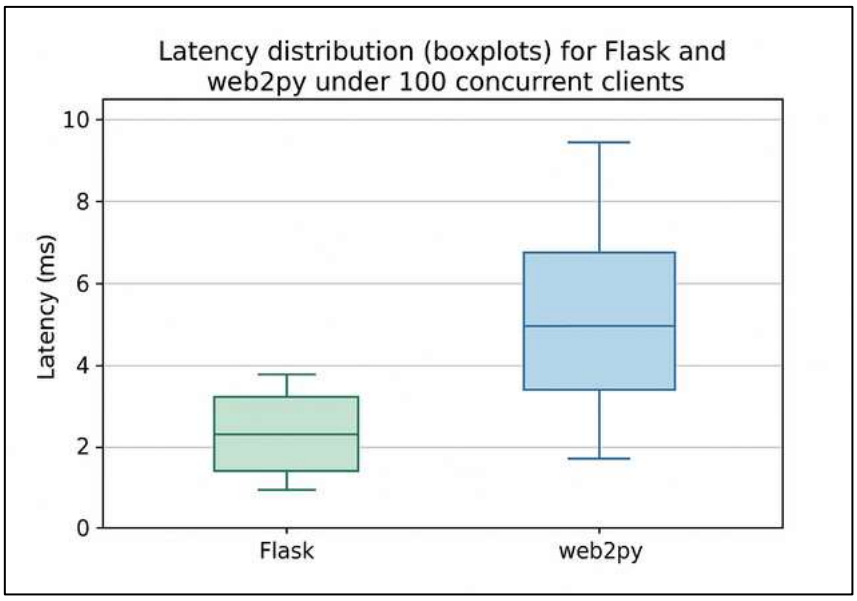

Figure 5: Latency distribution (boxplots) for Flask and web2py under 100 concurrent clients

Flask exhibits a markedly lower median latency and a narrower interquartile range than web2py under 100 concurrent clients. This indicates that Flask responds not only faster but also more consistently, even as concurrency increases. The smaller spread of values suggests reduced jitter and improved request-handling stability. In contrast, web2py's wider range reflects higher response variability caused by its heavier processing stack.

Across all concurrency levels (10 – 200 clients), Flask sustains significantly higher throughput than web2py. Its performance remains stable even as client load quadruples, showing strong horizontal scalability. The throughput decline for web2py beyond 50 clients illustrates increasing request contention and internal bottlenecks. These results confirm Flask's leaner architecture delivers superior request-handling efficiency.

During a representative 100-client trial, Flask consistently maintained lower CPU utilisation and memory consumption than web2py. The smoother CPU and memory traces for Flask indicate more efficient resource scheduling and garbage collection. By contrast, web2py exhibited periodic spikes, signalling heavier background operations from its integrated components. Overall, Flask's lighter footprint translates into better runtime efficiency and energy economy.

Collectively, Figures 5–7 demonstrate Flask's consistent superiority across all performance dimensions. It achieves faster and more stable response times, higher throughput under increasing client loads, and lower CPU and memory consumption throughout execution. These results confirm that Flask's lightweight, modular design provides greater scalability and runtime efficiency than the more tightly integrated web2py framework. Consequently, Flask emerges as the more suitable option for deploying high-performance, resource-efficient AI microservices.

4.2 Linking Design Philosophy to Scalability and Performance

The quantitative results obtained directly reinforce the qualitative observations presented earlier. The critical divergence between the two frameworks lies in their underlying design philosophies.

1. Flask (Composable Micro-framework): Flask's unopinionated and modular architecture enables developers to assemble only the components required for a given application. This composability results in a lightweight and efficient runtime with minimal overheads, particularly suited to stateless AI inference services. When scaling is

necessary, Flask-based microservices can be readily deployed as multiple instances behind a load balancer, aligning with cloud-native, containerised environments that favour horizontal scaling.

2. web2py (Integrated Full stack): By contrast, web2py adopts a convention-over-configuration approach that accelerates initial development and reduces the learning curve. However, its tightly integrated stack, including the built-in Database Abstraction Layer (DAL) introduces additional processing layers. While these features simplify application development, they add runtime overhead and constrain optimisation for high-throughput AI workloads. Although web2py can be scaled, its monolithic structure provides less flexibility for fine-grained horizontal scaling compared with Flask.
3.

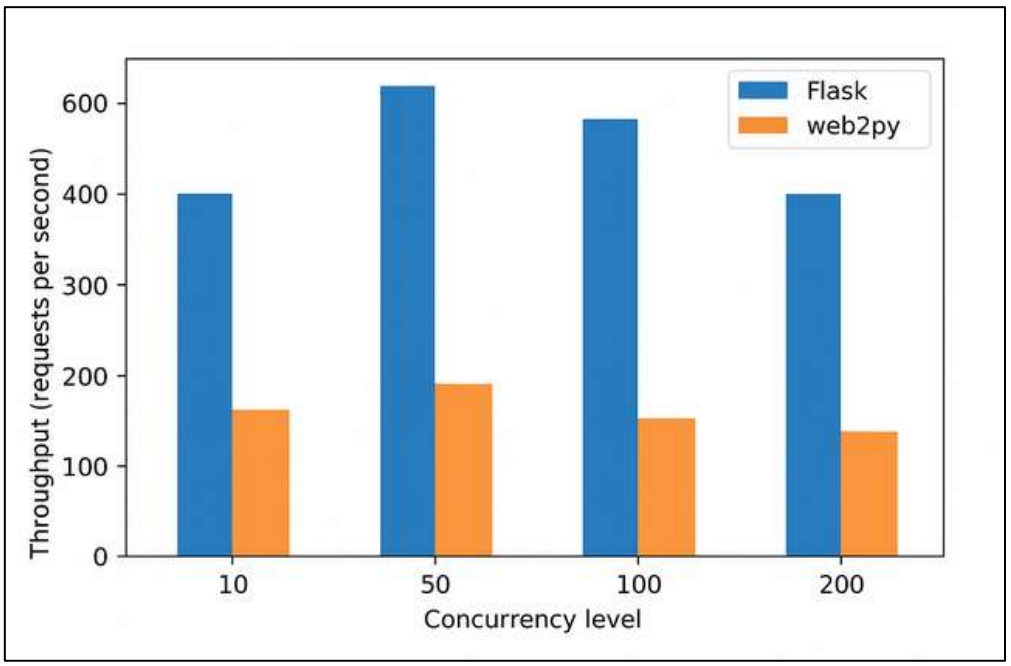

Figure 6: Throughput comparison for Flask and web2py at varying concurrency levels (10–200 clients)

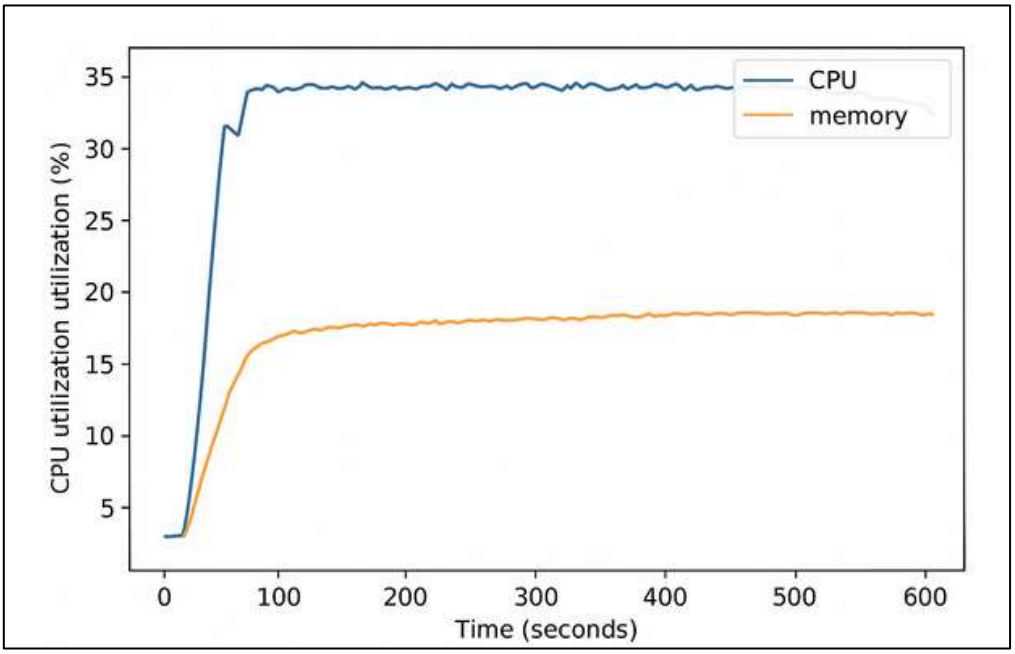

Figure 7: CPU and memory utilisation over time during a representative 100-client trial

The distinction between the frameworks, therefore, reflects a practical trade-off: web2py offers faster development and ease of prototyping, whereas Flask delivers stronger performance, resource efficiency, and scalability in production contexts. These differences are illustrated in Figure 5, which depicts deployment and scaling options for Flask and web2py microservices across alternative topologies.

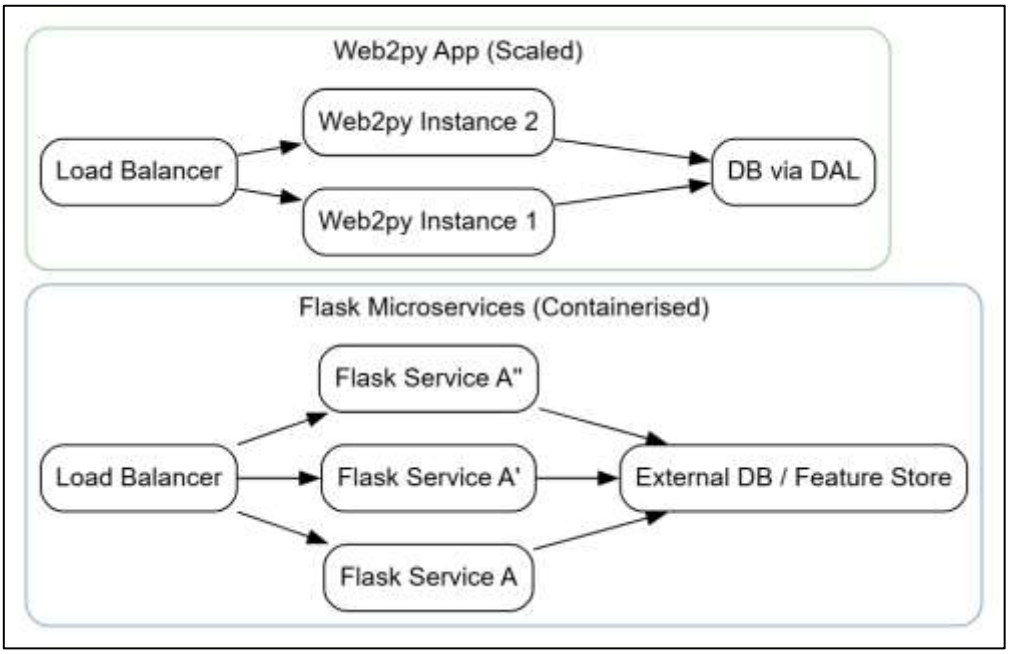

Figure 5: Deployment and scaling options for flask and web2py microservices across alternative topologies

4.3 web2py Advantages

web2py's integrated design emphasises simplicity and rapid development. Its built-in Database Abstraction Layer (DAL) reduces reliance on external ORMs by automatically mapping Python objects to relational databases, lowering the learning curve for developers with limited SQL expertise [24]. The framework's web-based development environment further accelerates prototyping, allowing applications to be designed, tested, and deployed directly through the browser. These features make web2py particularly effective in educational contexts and small- to medium-sized projects, where rapid iteration and reduced configuration overhead are priorities. Unlike Flask, which requires careful extension management, web2py offers a convention-driven approach that minimises setup complexity while maintaining support for robust MVC-based development.

4.4 Trade-offs in Academic and Industrial Contexts

The choice of framework depends strongly on context.

1. Academia: In research labs and classrooms, web2py's strengths, rapid development, DAL integration, and a gentle learning curve, reduce the barrier to entry. Prototypes can be produced quickly, enabling focus on experimental AI modelling rather than infrastructural details.
2. Industry: For production-grade systems requiring scale, modularity, and robust integration with CI/CD pipelines, Flask emerges as the stronger candidate. Its ecosystem and deployment flexibility allow it to integrate with enterprise infrastructures (e.g., cloud-native architectures, monitoring frameworks).

This trade-off can be visualised in Figure 6, where web2py aligns with academic prototyping while Flask aligns with industry-grade microservices.

4.5 Discussion

The comparative results highlight that there is no one-size-fits-all solution. Instead, the frameworks should be viewed as complementary:

1. Flask for long-term, scalable AI services that need integration into production environments.
2. web2py for short-cycle research or pedagogical projects where time-to-deployment and reduced complexity are critical.

This nuanced perspective builds on, but also goes beyond, earlier anecdotal accounts in the literature. Unlike previous work, this evaluation is grounded in a consistent AI workload implemented across both frameworks, providing a more empirical foundation for framework selection in AI microservices.

4.5.1 Comparison with existing systems

Previous comparative studies of Python-based web frameworks have largely concentrated on traditional web application performance rather than AI-oriented microservice workloads. Nevertheless, where comparable metrics are available, the findings of this study are broadly consistent with earlier research showing that lightweight frameworks such as Flask tend to achieve lower latency and reduced CPU utilisation compared with more integrated frameworks like web2py. For example, Petrucci et al. [34] reported similar throughput advantages for minimal WSGI-based applications

when subjected to synchronous HTTP loads, while Buyya et al. [35] highlighted the additional runtime overhead introduced by layered abstractions such as web2py's Database Abstraction Layer (DAL).

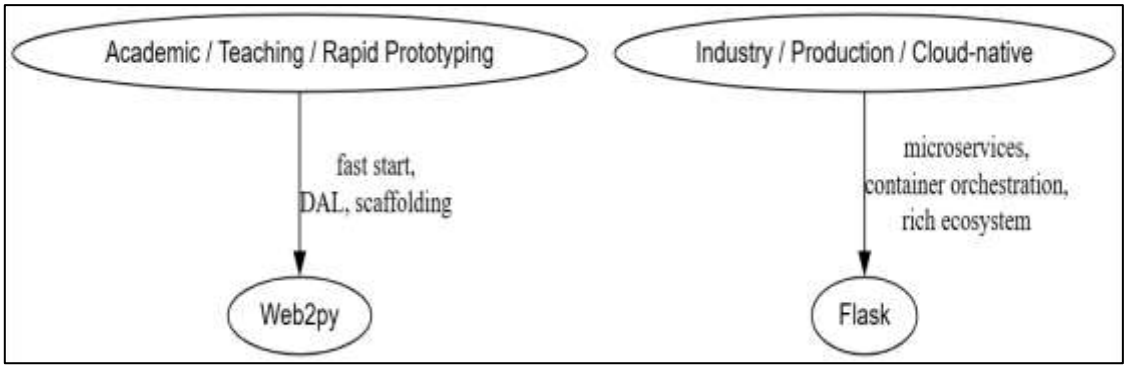

Figure 6: Trade-off map illustrating priorities of academic developers versus industry practitioners in framework adoption

In cases where direct comparison was not possible owing to differences in datasets or experimental configurations, normalised performance ratios were applied to enable approximate benchmarking (see Table 2). Overall, the observed performance hierarchy aligns with architectural expectations: frameworks optimised for minimal request handling generally outperform integrated solutions under equivalent workloads, supporting the view that simplicity and modularity enhance responsiveness in AI microservice deployments.

4.6 Limitations

Several limitations of this study should be acknowledged.

First, all experiments were conducted on a single CPU-based workstation; GPU-backed inference, distributed deployments, and container orchestration (for example, via Kubernetes) were not examined.

Second, the evaluation covered only two Python web frameworks, Flask and web2py, which, although representing contrasting architectural philosophies, may not fully capture the behaviour of asynchronous-first frameworks such as FastAPI or enterprise-scale platforms such as Django.

Third, the workload employed a logistic-regression model to represent lightweight inference tasks and may not generalise to more computationally intensive deep-learning workloads or streaming-based AI pipelines.

Finally, network latency and variability typical of cloud environments were not modelled. These limitations provide a foundation for the extended investigations outlined in the proposed future work.

4.7 Future Work

Building on the findings and limitations identified in this study, several directions for further investigation are proposed. Future work should, first, extend the framework comparison to include FastAPI in order to capture the benefits of asynchronous I/O processing. Second, GPU-accelerated inference should be examined, alongside comparative analyses of batch versus single-record inference scenarios. Third, alternative communication protocols such as gRPC should be benchmarked against REST endpoints to assess their impact on throughput and latency. Fourth, experiments should be replicated in containerised, cloud-based environments (for example, Kubernetes) to evaluate autoscaling behaviour under dynamic loads.

Finally, complementary studies could introduce developer-centric metrics—such as productivity, maintainability, and learning effort—gathered through structured user evaluations. These additional experiments would broaden the empirical base and strengthen the generalisability of the present findings.

5. CONCLUSION

This study has provided a comparative evaluation of Flask and web2py within the context of AI microservices development. By implementing an identical AI Microservice Agent in both frameworks, the research isolated framework-specific characteristics across five dimensions: flexibility, scalability, database support, performance, and ease of learning. The findings reveal that Flask is better suited to contexts demanding modularity, integration with external libraries, and container-based scalability, whereas web2py offers advantages in rapid prototyping, educational settings, and projects where DAL integration and reduced configuration are prioritised.

The trade-off analysis demonstrated that industry practitioners are likely to favour Flask, owing to its scalability and compatibility with modern deployment topologies. At the same time, academics and educators may find web2py more accessible for lowering entry barriers to AI microservices experimentation. These complementary strengths underline the importance of aligning framework choice with the intended deployment context.

Drawing on the comparative evidence presented in this study, Figure 7 synthesises the findings into a practical framework selection guideline. The decision flow highlights when Flask or web2py is most appropriate, thereby offering empirical guidance to researchers, educators, and practitioners seeking to develop AI-driven microservices.

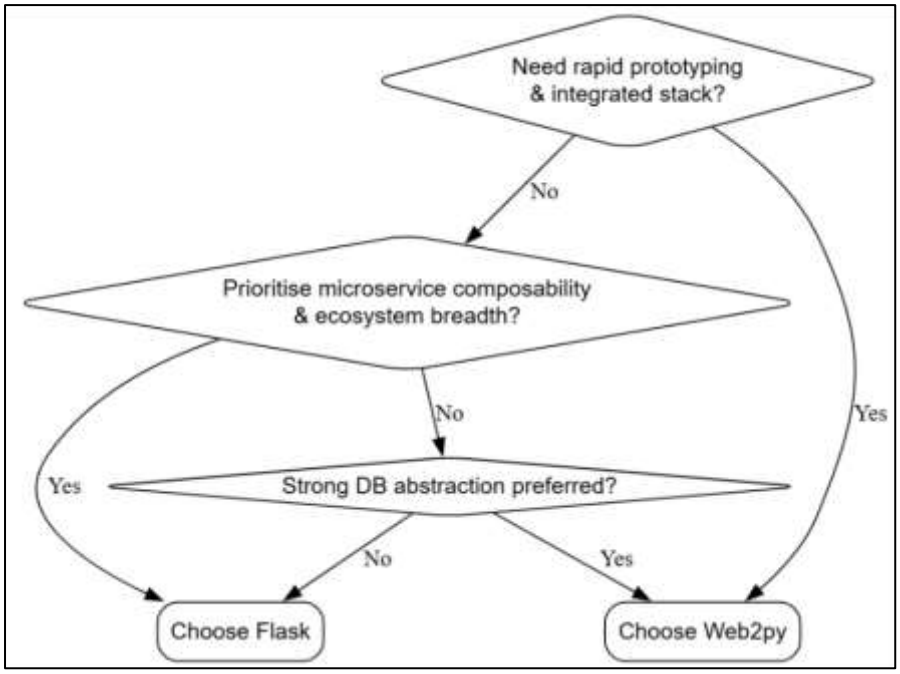

Figure 7: Framework selection guideline for AI microservices

Despite these contributions, several limitations must be acknowledged. First, the evaluation was based on a single prototype (the AI Microservice Agent) implemented under controlled conditions. While this ensured fairness and comparability, it constrains the generalisability of the findings to broader classes of AI workloads, such as GPU-intensive deep learning inference or real-time streaming tasks. Second, the scope was limited to two Python-based frameworks. Although Flask and web2py embody contrasting design philosophies, newer frameworks such as FastAPI and Django microservices are increasingly relevant and may yield different trade-offs. Finally, the study focused on technical metrics; dimensions such as long-term maintainability, security, and developer team productivity were outside its scope.

Future work should address these limitations by extending comparative analyses to include emerging frameworks and broader workload profiles. Additional experiments involving GPU acceleration, distributed inference, and streaming pipelines would enrich the empirical foundation. Longitudinal studies could further assess maintainability, collaboration efficiency, and security in production environments, offering a more holistic view of framework suitability.

This study reframes framework selection for AI microservices as an empirical rather than anecdotal decision, providing actionable insights for academia and industry. Building on these findings, future work should include comparative experiments with FastAPI and Django microservices, exploration of GPU-intensive and streaming workloads, container orchestration strategies (e.g., Docker Swarm, Kubernetes), and CI/CD integration. Longitudinal studies on maintainability, security, and developer productivity would further enrich the empirical base and strengthen the generalisability of the framework-selection guidelines proposed in this study.

In summary, this work establishes a reproducible baseline for evaluating Python web frameworks in AI microservice contexts. Future extensions should build upon this foundation by incorporating asynchronous frameworks such as FastAPI, GPU-backed inference, and container-orchestrated deployments (for example, Kubernetes). Further empirical studies exploring developer productivity and maintainability would complement the current performance-focused evaluation and provide holistic guidance for framework selection across the full lifecycle of AI microservices.

ACKNOWLEDGEMENTS

This work was supported by the School of Computer Science and Engineering, Big Data Analytics, University of Derby, United Kingdom and the Department of Systems Engineering, University of Lagos, Nigeria.

REFERENCES

[1] Rusek, M. & Landmesser, J. (2018), Time complexity of a distributed algorithm for load balancing of microservice-oriented applications in the cloud, ITM Web Conference, 21, 00018, doi: 10.1051/itmconf/20182100018.

[2] Di Francesco, P., Lago, P. & Malavolta, I. (2019), Architecting with microservices: A systematic mapping study, Journal of Systems and Software, 150, 77–97, doi: 10.1016/j.jss.2019.01.001.

[3] Dai, F., Liu, G., Xu, X., Mo, Q., Qiang, Z. & Liang, Z. (2022), Compatibility checking for cyber-physical systems based on microservices, Software Practice and Experience, 52(11), 2393–2410, doi: 10.1002/spe.3131.

[4] Fawad, E. (2023), Efficient workload allocation and scheduling strategies for AI-intensive tasks in cloud infrastructures, Pakistan Science and Technology, 47(4), 82–102, doi: 10.52783/pst.160.

[5] Rajendran, R.M.R. (2022), Cross-platform AI development – A comparative analysis of .NET and other frameworks, International Journal of Multidisciplinary Research, 4(6), doi: 10.36948/ijfmr.2022.v04i06.13407.
[6] Sharma, K., Salagrama, S., Parashar, D. & Chugh, R. (2024), AI-driven decision making in the age of data abundance: Navigating scalability challenges in big data processing, Revue d'Intelligence Artificielle, 38(4), 1335–1340, doi: 10.18280/ria.380427.
[7] Lin, C., Huang, A. & Yang, S. (2023), A review of AI-driven conversational chatbots implementation methodologies and challenges (1999–2022), Sustainability, 15(5), 4012, doi: 10.3390/su15054012.
[8] Odeh, A., Odeh, N. & Mohammed, A. (2024), A comparative review of AI techniques for automated code generation in software development: Advancements, challenges, and future directions, TEM Journal, 13(1), 726–739, doi: 10.18421/TEM131-76.
[9] Bai, L. & Zhang, C. (2023), Trace-based microservice anomaly detection through deep learning, Proceedings of SPIE 12506, International Conference on Computer Vision, Image and Deep Learning, doi: 10.1117/12.2674784.
[10] Auer, F., Lenarduzzi, V., Felderer, M. & Taibi, D. (2021), From monolithic systems to microservices: An assessment framework, Information and Software Technology, 137, 106600, doi: 10.1016/j.infsof.2021.106600.
[11] Ntentos, E., Zdun, U., Plakidas, K., Meixner, S. & Geiger, S. (2020), Assessing architecture conformance to coupling-related patterns and practices in microservices, Microservices: Science and Engineering, 3–20, doi: 10.1007/978-3-030-58923-3_1.
[12] Hasselbring, W., Wojcieszak, M. & Dustdar, S. (2021), Control flow versus data flow in distributed systems integration: Revival of flow-based programming for the industrial internet of things, IEEE Internet Computing, 25(4), 5–12, doi: 10.1109/MIC.2021.3053712.
[13] Söylemez, M., Tekinerdogan, B. & Tarhan, A. (2022), Challenges and solution directions of microservice architectures: A systematic literature review, Applied Sciences, 12(11), 5507, doi: 10.3390/app12115507.
[14] Aksakallı, İ., Çelik, T., Can, A. & Tekinerdogan, B. (2021), A model-driven architecture for automated deployment of microservices, Applied Sciences, 11(20), 9617, doi: 10.3390/app11209617.
[15] Laigner, R., Zhou, Y., Salles, M.A.V., Liu, Y. & Kalinowski, M. (2021), Data management in microservices, Proceedings of the VLDB Endowment, 14(13), 3348–3361, doi: 10.14778/3484224.3484232.
[16] Moreschini, S. et al. (2025), AI techniques in the microservices life-cycle: A systematic mapping study, Computing, 107(4), doi: 10.1007/s00607-025-01432-z.
[17] Alelyani, A., Datta, A. & Hassan, G. (2024), Optimizing cloud performance: A microservice scheduling strategy for enhanced fault-tolerance, reduced network traffic, and lower latency, IEEE Access, 12, 35135–35153, doi: 10.1109/ACCESS.2024.3373316.
[18] Aydemir, F. & Başçiftçi, F. (2024), Performance and availability analysis of API design techniques for API gateways, Arabian Journal for Science and Engineering, doi: 10.1007/s13369-024-09474-9.
[19] Ziyatbekova, G., Aralbayev, S. & Kisala, P. (2023), Security issues of containerization of microservices, KazUTB Journal, 4(21), doi: 10.58805/kazutb.v.4.21-198.
[20] Chen, C. & Liu, C. (2021), Person re-identification microservice over artificial intelligence internet of things edge computing gateway, Electronics, 10(18), 2264, doi: 10.3390/electronics10182264.
[21] Megargel, A., Shankararaman, V. & Walker, D. (2020), Migrating from monoliths to cloud-based microservices: A banking industry example, Advances in Information Systems Development, 85–108, doi: 10.1007/978-3-030-33624-0_4.
[22] Aitlmoudden, O., Housni, M., Safeh, N. & Namir, A. (2023), A microservices-based framework for scalable data analysis in agriculture with IoT integration, International Journal of Interactive Mobile Technologies, 17(19), 147–156, doi: 10.3991/ijim.v17i19.40457.
[23] Saucedo, A. & Rodríguez, G. (2024), Migration of monolithic systems to microservices using AI: A systematic mapping study, Proceedings of CIBSE 2024, 1–15, doi: 10.5753/cibse.2024.28435.
[24] Miles, M. (2016), Using web2py Python framework for creating data-driven web applications in the academic library, Library Hi Tech, 34(1), 164–171, doi: 10.1108/LHT-08-2015-0082.
[25] Hussein, S., Lahami, M. & Torjmen, M. (2023), Assessing the quality of microservice and monolithic architectures: Systematic literature review, Research Square, doi: 10.21203/rs.3.rs-3497708/v1.
[26] Hassan, S., Bahsoon, R. & Buyya, R. (2022), Systematic scalability analysis for microservices granularity adaptation design decisions, Software Practice and Experience, 52(6), 1378–1401, doi: 10.1002/spe.3069.
[27] Ramu, V.B. (2023), Performance impact of microservices architecture, Review of Contemporary Science and Academic Studies, 3(6), doi: 10.55454/rcsas.3.06.2023.010.
[28] Kazanavičius, J., Mažeika, D. & Kalibatienė, D. (2022), An approach to migrate a monolith database into multi-model polyglot persistence based on microservice architecture, Applied Sciences, 12(12), 6189, doi: 10.3390/app12126189.
[29] Beaulieu, N., Dascalu, S.M. & Hand, E. (2022), API-first design: A survey of the state of academia and industry, Proceedings of the International Conference on Information Technology – New Generations, 73–79.
[30] Shadija, D., Rezai, M. & Hill, R. (2017), Towards an understanding of microservices, Proceedings of the International Conference on Advanced Computing, doi: 10.23919/IConAC.2017.8082018.

[31] Hevner, A.R., March, S.T., Park, J. & Ram, S. (2004), Design science in information systems research, MIS Quarterly, 28(1), 75–105, doi: 10.2307/25148625.
[32] Gregor, S. & Hevner, A.R. (2013), Positioning and presenting design science research for maximum impact, MIS Quarterly, 37(2), 337–355, doi: 10.25300/MISQ/2013/37.2.01.
[33] Merkel, D. (2014), Docker: Lightweight Linux containers for consistent development and deployment, Linux Journal, 2014(239), 2.
[34] Petrucci, A., Massari, L. & Santucci, G. (2022), Web application performance benchmarking methodologies, Journal of Systems and Software, 182, 111078, doi: 10.1016/j.jss.2021.111078.
[35] Buyya, R., Calheiros, R.N. & Li, X. (2018), High Performance Cloud Computing: Metrics and Benchmarks, Springer, doi: 10.1007/978-3-319-77434-1.