1. Title: Python in Web Development: Building Dynamic Websites with Flask

Course name: Programming with Python

Course code: DLMDSPWP01

Course of Study: Computer Science (Master)

Date: August 2025

Author: Marzie Navaeilavasani

Matriculation Number: 10239549

Tutor: Ugur Ural

Table of Contents

2. List of Abbreviations

Abbreviation	Meaning / Context
Abadi et al.	Citation reference (not technical, included for context)
AI	Artificial Intelligence
AJAX	Asynchronous JavaScript and XML
API	Application Programming Interface
ASGI	Asynchronous Server Gateway Interface (Python async web standard)
AWS	Amazon Web Services
CI/CD	Continuous Integration / Continuous Deployment
CRUD	Create, Read, Update, Delete (database operations)
CSS	Cascading Style Sheets
CSRF	Cross-Site Request Forgery
DB / DBMS	Database / Database Management System
Django	Python full-stack web framework
Flask	Python microframework for web development
Flask-JWT-Extended	Flask extension for JWT-based authentication
Flask-Migrate	Flask extension for database migrations
Flask-PyMongo	Flask extension for MongoDB integration
Flask-RESTful / Flask-RestX	Flask extensions for building RESTful APIs
Gunicorn	Python WSGI HTTP server
HTTP	HyperText Transfer Protocol
Jinja2	Templating engine used by Flask and other frameworks
JSON	JavaScript Object Notation

Abbreviation	Meaning / Context
NumPy	Python numerical computing library
ORMs / ORM	Object-Relational Mapping / Mapper (e.g., SQLAlchemy)
Pandas	Python data analysis/manipulation library
REST	Representational State Transfer
SQL	Structured Query Language
SQLAlchemy	Python library for ORM-based database interaction
SPA	Single Page Application
SSR	Server-Side Rendering
TensorFlow	Machine learning library
uWSGI	WSGI server for deploying Python apps
UI / UX	User Interface / User Experience
Uvicorn	ASGI server for Python asynchronous applications
WSGI	Web Server Gateway Interface
XSS	Cross-Site Scripting

3. Introduction: Python in Web Development: Building Dynamic Websites with Flask

In recent decades, web development has become one of the most important areas in software engineering. This change is fueled by the fast growth of the internet and the rising need for interactive, data-driven websites and applications. Programming languages like JavaScript, PHP, Ruby, and Python have all been key in this shift. Python, in particular, has gained popularity because of its simplicity, readability, and versatility in various problem areas (Lutz, 2013). Python's clean syntax and extensive library ecosystem make it suitable for many applications, including data science, artificial intelligence, automation, and especially web development (Van Rossum & Drake, 2009). When it comes to creating dynamic websites, Python's effectiveness is improved with specific frameworks. Flask is one of the most popular microframeworks for web development.

Flask was introduced in 2010 by Armin Ronacher as part of the Pocoo team's projects. It was later included in the Pallets Projects group (Wikipedia, 2025). Unlike full-stack frameworks like Django, Flask has a minimalist, modular approach and is often called a microframework. This means Flask provides only the essential tools, such as routing, request handling, and templating. Developers can then add extensions or third-party libraries based on their project needs (Flask Documentation, 2023). This design principle reflects the Pythonic values of simplicity and clarity, allowing developers to create web applications that are flexible and scalable.

The importance of frameworks like Flask comes from their ability to connect static websites with modern, dynamic applications. A dynamic website can create content in real time, interact with databases, and customize user experiences based on input and context (Dhir & Mital, 2013). Flask provides this dynamism by working with both relational and non-relational databases. It also supports RESTful API development and uses the Jinja2 templating engine. This engine allows developers to include Python-like logic directly in HTML (Ronacher, 2010). Because of these features, Flask is widely used in academic settings for teaching web technologies and in industry for prototyping and deploying production-ready applications.

Flask's growing popularity is partly due to its ability to change with new technology. For example, as single-page applications (SPAs) and mobile-first designs become more common, Flask is often used as a backend framework. It provides RESTful or GraphQL APIs for JavaScript-based frontends like React or Vue.js (Miguel Grinberg, 2018). This flexibility helps Flask stay useful across different development situations, from small projects to large enterprise solutions. At the same time, Flask has added modern features like asynchronous programming (async/await) to meet the performance needs of today's web systems (Flask Documentation, 2023).

Security, scalability, and performance are important factors that highlight Flask's academic and practical value. While some view microframeworks as weaker than full-stack frameworks, Flask counters this

perception with a strong set of extensions that handle user authentication, input validation, and database management. Research on Python web frameworks points out the need for secure coding practices, especially given the common risks like injection attacks in web applications (Wang et al., 2018). By offering simplicity while still being expandable, Flask serves as a valuable tool for teaching web concepts and fulfilling professional software engineering needs.

In short, Python's popularity in web development closely relates to frameworks like Flask. As a microframework, Flask provides the simplicity, flexibility, and expandability needed to create dynamic websites and services. Its growing use in schools and businesses highlights its value as a teaching tool and a production resource. Looking at Flask in web development helps us understand larger trends in programming language use, framework design, and the changing needs of creating dynamic, user-focused web applications.

4. Literature Review: Python in Web Development and Flask's Role in Building Dynamic Websites

4.1. Python in Web Development (General Studies)

Python's popularity as a web development language comes from its readability, extensive ecosystem, and a well-established set of web interfaces and standards that allow frameworks and servers to work together. The Web Server Gateway Interface (WSGI) created a standard contract between web servers and Python applications. This helped bring about a range of frameworks that could run on various servers with minimal changes (see PEP 3333 discussion in practitioner-oriented sources). While the official PEP is available on python.org, accessible introductions clarify WSGI's function as a straightforward, universal interface connecting servers and applications (Liquid Web, n.d.; Plone Training, n.d.). Over the past ten years, the Asynchronous Server Gateway Interface (ASGI) has developed as WSGI's asynchronous counterpart. It supports concurrency-friendly protocols, like WebSockets, and event-driven servers (ASGI Team, n.d.; ASGI Team, n.d.). These gateway standards are vital to Python's flexibility. They let developers pick from various frameworks and deployment stacks without being tied to a specific vendor.

Classic works in web engineering show that developing web applications needs careful processes and solid architecture, much like traditional software engineering. This is essential due to the web's scale, diversity, and fast changes (Dhir & Mital, 2013; Murugesan & Ginige, 2001; Pressman et al., 2001). Recent surveys still point out that quality factors—testability, security, maintainability, and performance—are important for evaluating web systems (Gerasimov et al., 2024). From this viewpoint, Python's framework landscape provides a wide range of architectural options, from "micro" libraries that promote composition to "batteries-included" platforms that offer a complete stack.

4.2. Comparison of Python Frameworks (Flask, Django, FastAPI, Pyramid)

Django. Django is a high-level framework with many built-in features. It includes routing, ORM, authentication, admin, and templating, all in one platform. This setup is meant for quick development, allowing users to go "from concept to completion" (Django Software Foundation, 2025; Django Software Foundation, 2025). This integration simplifies design choices for typical CRUD and content-heavy applications. It is supported by thorough official documentation and a strong community.

Flask. Flask takes a different approach. It has a lightweight core built on Werkzeug (WSGI) and Jinja. This design encourages developers to use extensions only when necessary (Pallets Projects, 2025; Pallets Projects, 2018). The "Design Decisions in Flask" document clearly states that these choices aim for simplicity in common tasks while still allowing options for complex needs (Pallets Projects, 2025). In

practice, Flask often acts as a base for RESTful APIs, small services, or modular backends. Teams can choose their own ORM, authentication method, and tools.

Pyramid. Pyramid sits between these poles with a “start small, finish big” philosophy. It aims to provide just the core necessities, such as URL mapping, security, and static assets, while remaining fast and thoroughly tested. Developers can add templating, database layers, and other components as needed (Pylons Project, 2023a; Pylons Project, 2023b). The Pyramid documentation emphasizes configurability. For example, it offers multiple view predicates, various routing strategies, and cookiecutter project templates that support Jinja2, Mako, Chameleon, and different persistence methods (Pylons Project, 2023c, 2023d, 2023e).

FastAPI (and the ASGI ecosystem). Django, Flask, and Pyramid came from the WSGI era. In contrast, newer frameworks like FastAPI are built directly on ASGI servers, such as Uvicorn and Hypercorn. They use async I/O to improve concurrency and make development easier. The ASGI documentation shows that the spec originated from Django Channels and points out the shift toward asynchronous stacks for web sockets and long-lasting connections. In this ASGI setting, Python web apps can manage streaming, background tasks, and real-time interactions more effectively. These abilities are increasingly important for today's dynamic web applications.

In summary, the choice of framework involves trade-offs. Django focuses on speed to feature with solid defaults. Flask and Pyramid emphasize composability and simplicity. ASGI-native frameworks prioritize concurrency and modern protocols. Teams usually assess these trade-offs based on project size, performance needs, and governance factors.

4.3. Academic Studies on Dynamic Websites

The theoretical foundations of dynamic web systems connect closely to architectural principles like REST. Fielding’s dissertation from 2000 outlined REST’s constraints, which include statelessness, a uniform interface, cacheability, and layered systems. These principles encourage growth and scalability in distributed hypermedia systems. REST is a key element in Python web development, seen in idiomatic routing, resource-oriented APIs, and content negotiation across different frameworks. Current web engineering research emphasizes that testing, quality assurance, and process improvement are crucial for dynamic sites. This is important due to the rapid increase in interactive behaviors and third-party integrations (Gerasimov et al., 2024).

From a process perspective, web engineering texts emphasize lifecycle coverage, including requirements, modeling, design, implementation, deployment, and maintenance. They compare web-specific methods, such as OOHDM, WebML, and WSDM, with general software processes (Dhir & Mital, 2013; Murugesan & Ginige, 2001). These works advocate for model-driven approaches, UI/UX

considerations, and iterative validation due to rapidly changing requirements. These factors fit well with Python frameworks that focus on quick prototyping, comprehensive testing ecosystems, and modular architecture. The move from purely server-rendered pages to hybrid and API-centric architectures, such as SPAs using JSON APIs, highlights REST's ongoing importance and the need for strong API frameworks and templating engines.

4.4. The Role of Templating Engines (e.g., Jinja2)

Templating engines are essential for creating dynamic websites, even as client-side frameworks take on more rendering tasks. Jinja2, used by Flask and supported by Pyramid, offers an expressive syntax along with filters, macros, and extension points that balance designer readability with programming power (Jinja Documentation, 2025a, 2025b, 2025c). Importantly, autoescaping serves as a key defense against cross-site scripting (XSS). Jinja's documentation urges developers to enable or configure autoescape policies properly and to treat HTML/XML templates differently from plain-text outputs (Jinja Documentation, 2025a). Jinja's extension mechanism allows for internationalization and reusable template logic, which helps with maintainability (Jinja Documentation, 2025c).

Templating also relates to software architecture. Server-side rendering (SSR) helps with SEO-friendly, low-latency first paint. Partial rendering and component-based layouts support progressive improvement. Template inheritance cuts down on duplication. In educational and prototyping settings, where quick changes and clarity matter, Jinja's simple mental model is a teaching benefit. Even in projects that focus on APIs, templating often drives admin consoles, emails, and tasks for generating static sites.

4.5. Flask in Educational Contexts (Teaching Web Programming)

Flask's simple core and clear concepts make it a popular choice in university courses and materials. Classes cover HTTP basics, routing, request and response objects, form handling, and templating through small, focused labs (University of Warwick, 2022). Practitioner texts and tutorials, like the well-known "Flask Mega-Tutorial," guide users from single-file apps to modular blueprints, authentication, and databases. This approach aligns with common learning outcomes (Grinberg, n.d.). Instructors often emphasize Flask's "opt-in" architecture to show how real projects only add the necessary layers, such as ORMs, migrations, authentication, and background tasks. This method effectively teaches composition and dependency management.

This educational approach reflects web engineering ideas that focus on process, modularity, and quality assurance in curricula (Dhir & Mital, 2013; Murugesan & Ginige, 2001). The Flask documentation clearly explains design choices like thread-locals and extension philosophy. It also provides material for

classroom discussions about the trade-offs in framework design (Pallets Projects, 2025; Pallets Projects, 2018).

4.6. Flask in Industry (Startups, Prototyping, APIs)

In the industry, Flask is commonly used for small services, prototypes that become production-ready, and lightweight APIs. Teams often value fast iteration and precise control over dependencies. Engineering blogs mention that Flask is used for internal tools and services. For instance, Netflix has shared its experiences with Flask-based services and patterns on its architecture blog, placing Flask within a broader context of microservices and data tools (Netflix Tech Blog, 2018, 2021). These accounts highlight practical benefits such as small codebases, clear routing, fast local development, and easy deployment with WSGI servers. They also recognize that larger systems may eventually move towards more integrated platforms or ASGI stacks for handling concurrency and streaming.

More broadly, the rise of ASGI has pushed some new API projects toward asynchronous frameworks. However, many production systems still use Flask with Gunicorn or uWSGI, as synchronous WSGI effectively handles request and response scenarios (ASGI Team, n.d.; Uvicorn, n.d.). Organizations that start with Flask for prototypes can keep much of the code as the product grows. They can do this by adding extensions such as SQLAlchemy, migrations, and authentication libraries. They can also break down the application using blueprints and gradually layer in observability and CI/CD. This method aligns with web engineering advice on adopting processes and evolving architecture in stages (Dhir & Mital, 2013).

4.7. Synthesis

The Python web ecosystem includes integrated frameworks like Django, composable microframeworks such as Flask and Pyramid, and async-native platforms like FastAPI, which are built on standardized gateways like WSGI and ASGI. Academic studies on web engineering and software architecture, especially REST, offer a way to evaluate these frameworks. They focus on quality attributes, lifecycle rigor, and architectural limits. In this context, Flask’s design is intentionally simple but flexible. It effectively teaches fundamental web concepts and serves startups and teams that value quick development, clear choices, and manageable services.

5. Methodology / Conceptual Approach

Flask's approach to web development focuses on simplicity, modularity, and composability. This reflects its design as a microframework. Unlike full-stack frameworks like Django, Flask offers a lightweight foundation with key components, such as routing, request handling, and templating. It leaves the decision about additional components, such as ORMs, authentication, and third-party integrations, to the developer (Pallets Projects, 2025). This method allows teams and individual developers to customize application architecture based on specific project needs. It also encourages a better understanding of web application mechanics in educational settings (Grinberg, 2018).

5.1. Implementation of Flask Applications

A typical Flask application begins by creating an application object from the Flask class. This object serves as the main registry for routing rules, configuration, and extensions. In Flask, routes link to view functions that handle HTTP requests and return responses as HTML, JSON, or other data types. Request and response objects are available through thread-local contexts, which simplifies access to user input, session data, and request metadata (Pallets Projects, 2025). This simple core process allows developers to build functional prototypes quickly while maintaining a modular approach with extensions.

5.2 Example Project Structures: Application Factory Pattern

For larger or more complex applications, the application factory pattern is widely recommended (Grinberg, 2018; Pallets Projects, 2025). This pattern defines a function that creates and configures the Flask application. It allows multiple instances of the app to be generated with different settings. This is especially helpful for testing, staging, and production environments. A typical factory-based structure may include the following components:

- app/__init__.py – contains the application factory function, initializes extensions, and registers blueprints.
- app/models.py – defines database models for ORM mapping.
- app/routes.py – handles routing and view functions.
- app/templates/ – stores Jinja2 templates for rendering dynamic content.
- app/static/ – contains CSS, JavaScript, and image assets.
- config.py – configuration settings for development, testing, and production.
- run.py – entry point to start the Flask application.

This structure promotes separation of concerns, testability, and scalability. It follows established software engineering principles for web applications (Dhir & Mital, 2013).

5.3 Integration with Databases

Flask does not require a specific database layer, which allows for flexibility in choosing relational or NoSQL databases based on project needs. SQLAlchemy is the most popular Object Relational Mapper (ORM) for relational databases. It supports complex queries, schema migrations, and session management (Grinberg, 2018). Flask extensions like Flask-Migrate and Flask-SQLAlchemy help with database integration. They offer tools for migrations, query abstractions, and connection pooling.

For non-relational databases, Flask can connect with MongoDB using extensions like Flask-PyMongo. This method allows for dynamic schemas, JSON-based document storage, and horizontal scaling. It's ideal for applications that require evolving data structures, high-throughput workloads, or fast prototyping (Pallets Projects, 2025). By keeping a modular approach to database integration, developers maintain control over how data is stored while keeping the core application simple.

5.4 RESTful API Design with Flask

A common method in modern web development is building RESTful APIs. These APIs let client applications interact with server resources using standard HTTP methods like GET, POST, PUT, and DELETE. Flask’s routing system, along with request parsing and response generation, makes it easier to create RESTful endpoints (Grinberg, 2018). Developers usually organize API routes based on resource models. They often use Flask extensions like Flask-RESTful or Flask-RestX to make serialization, request validation, and documentation simpler.

A typical RESTful design workflow in Flask includes:

1. Defining resource models, such as User and Post, in ORM classes.

2. Creating endpoints where routes match HTTP verbs to CRUD operations.

3. Serializing responses as JSON for web clients or mobile applications.

4. Adding authentication and authorization using extensions like Flask-JWT-Extended.

This modular approach follows standard web architecture principles. It helps with scalability, testability, and maintainability (Fielding, 2000; Gerasimov et al., 2024).

5.5 Dynamic Website Development Process

The website development process in Flask usually involves design iterations and gradual delivery:

1. Requirements and Planning: Define essential functionality, data models, and user interaction patterns.

2. Prototype Development: Create a basic working application with routes, templates, and a simple database.

3. Template Integration: Use Jinja2 to display dynamic content, implement template inheritance, and organize reusable components (Jinja Documentation, 2025b).

4. Database Integration: Connect ORM models or NoSQL collections to provide storage for dynamic data.

5. API and Service Layer: Provide RESTful endpoints for client applications, AJAX interactions, or mobile apps.

6. Testing and Validation: Conduct unit and integration tests, including route testing, form validation, and database queries.

7. Deployment: Launch to production servers using WSGI servers, like Gunicorn or uWSGI, or ASGI for asynchronous endpoints.

8. Maintenance and Iteration: Make updates, monitor performance, and refactor as needed to support new features or changing requirements.

This iterative method reflects agile development practices. It focuses on quick prototyping, ongoing integration, and modular growth (Grinberg, 2018; Dhir & Mital, 2013). Flask's lightweight design and modular extensions support this workflow. They allow small teams and individual developers to create strong, dynamic web applications efficiently.

6. Analysis and Discussion

6.1. Strengths of Flask

Flask focuses on simplicity, flexibility, and extensibility. This makes it a popular choice for teaching and professional web development. Its lightweight core lets developers select only the components needed for their application, keeping things straightforward. This simplicity leads to a lower learning curve for students and beginners. They can prototype and experiment quickly with minimal boilerplate code (Grinberg, 2018).

The flexibility of Flask shows in its support for various extensions and third-party integrations. These include SQLAlchemy for relational databases, Flask-PyMongo for NoSQL integration, Flask-JWT-Extended for authentication, and Flask-RESTful for building REST APIs (Pallets Projects, 2025). Developers can customize the stack to meet the specific needs of a project without being limited by strict defaults. Furthermore, Flask's extensibility allows for modularity. Applications can be organized using blueprints, which help in creating reusable components, independent modules, and multi-service architectures. This modularity is particularly useful for large projects that need incremental development, testing, and deployment.

Another strength of Flask is its connection to Python's ecosystem. Developers gain from easy integration with scientific computing and machine learning libraries like pandas, NumPy, TensorFlow, and PyTorch. This makes Flask suitable as a backend for AI-driven or data-driven applications (Chollet, 2021; Abadi et al., 2016). Along with Jinja2 templating for dynamic content rendering, Flask offers a solid base for interactive and smart web applications.

6.2. Weaknesses of Flask

Despite its strengths, Flask has several limitations. Its minimalist and less opinionated design can create inconsistent architecture when teams with different experience levels develop projects. Without standardized patterns, large projects face issues with fragmented codebases and maintainability challenges (Grinberg, 2018).

Security concerns are another drawback. Flask provides basic protections through autoescaping in Jinja2 and session management. However, developers must handle authentication, input validation, and protection against common web vulnerabilities such as SQL injection, cross-site scripting (XSS), and cross-site request forgery (CSRF) (Wang, Chen, & Wang, 2018). In contrast, full-stack frameworks like Django have built-in security features that reduce these risks by default.

Flask's asynchronous features are still limited when compared to ASGI-native frameworks like FastAPI. While Flask does support async/await starting from Python 3.8, its implementation relies on WSGI and

does not fully take advantage of non-blocking I/O. This could lead to performance issues for applications that need to handle many connections at once (Uvicorn, n.d.; Pallets Projects, 2025). Developers who need real-time updates, long-lived connections, or high-throughput API endpoints may find this limitation compared to modern asynchronous frameworks.

6.3. Case Studies

6.3.1. Flask for Academic Projects

Flask's educational value is well-known. Many university courses use Flask to teach important web development concepts. These concepts include routing, request and response handling, templating, and database integration (University of Warwick, 2022). The framework's straightforward design helps students understand basic web ideas without getting bogged down by the challenges of full-stack platforms. For instance, assignments using Flask often involve creating blogs, portfolio sites, or small REST APIs. These tasks show the complete web development process. This teaching method supports research on web engineering education, highlighting the need for hands-on, practical learning experiences (Dhir & Mital, 2013).

6.3.2. Flask in Startups and Prototyping

In startup environments, Flask's ability to quickly create prototypes is greatly appreciated. Its simple setup lets teams work on product ideas and launch functioning prototypes in days instead of weeks. For example, small tech startups often use Flask to build internal tools, dashboards, and lightweight services. They typically do this before they scale or move to larger frameworks if necessary (Netflix Tech Blog, 2018). The modular architecture makes it easy to connect with microservices, third-party APIs, and asynchronous job queues, which provides enough flexibility for changing needs.

6.3.3. Flask Powering APIs in Production

Flask has shown itself to be effective for production-grade applications, especially for API-driven services. RESTful APIs created with Flask use its routing system, request parsing, and JSON serialization. This allows for strong back-end services for web and mobile applications (Grinberg, 2018). Companies like Netflix and Lyft have turned to Flask to build internal services and microservices. Their experiences show that Flask is suitable for real-world, high-impact applications (Netflix Tech Blog, 2021). When paired with WSGI servers such as Gunicorn or uWSGI, along with containerization platforms like Docker, Flask can deliver high reliability and scalability while keeping code easy to manage.

6.3.4. Flask Compared with Django and FastAPI

Django is a complete framework that comes with built-in features, such as an integrated ORM, authentication, admin interfaces, and templating. Its structured approach speeds up development for

content-heavy applications, but it limits flexibility for custom setups (Django Software Foundation, 2025). In contrast, Flask takes a minimalist route, allowing for custom architecture, modularity, and selective use of extensions. For small to medium applications, Flask often lowers overhead and makes maintenance easier.

FastAPI is an async-native framework created for modern API-driven applications. It uses Python type hints for data validation and produces automatic documentation through OpenAPI. It also offers fully asynchronous endpoints (Uvicorn, n.d.). While FastAPI performs better than Flask in high-concurrency situations, Flask has its strengths in simplicity, abundant documentation, and educational use. Teams usually prefer Flask for projects that require fast iteration, modular design, or integration with synchronous codebases, while FastAPI is the choice for high-performance, API-focused services.

6.3.5. Flask and Emerging Trends

Flask works well with current development methods like microservices, cloud deployment, and containerization. Its modular architecture and blueprint system let developers break applications into separate services. This supports microservices designs that can be deployed independently on Docker or managed through Kubernetes (Grinberg, 2018; Baldini et al., 2017). Serverless deployment platforms like AWS Lambda, Google Cloud Functions, and Azure Functions also support Flask through adapters such as Zappa and the Serverless Framework. These tools enable scalable web services that respond to events without requiring direct management of infrastructure.

These emerging trends show how adaptable Flask is. While it doesn't provide native asynchronous performance like FastAPI, its ecosystem and user-friendly design keep it relevant in cloud-native, AI-driven, and data-heavy applications. Flask's ability to work with modern CI/CD pipelines, observability tools, and cloud infrastructure means it can effectively support both startups and established organizations.

6.4. Synthesis

In summary, Flask balances simplicity and extensibility, making it a great choice for education, prototyping, and production APIs. Its strengths, including flexibility, modularity, and integration with the Python ecosystem, enable rapid development. However, its weaknesses, such as limited async support, potential security gaps, and reliance on developer choices, require careful management. Comparing Flask with Django and FastAPI shows its niche as a lightweight, versatile framework that fits various project sizes and architectural styles. As microservices, cloud deployment, and AI integration become more common, Flask’s modular design ensures it remains relevant for modern web development.

6.5. Future Directions

6.5.1. Flask and Async/Await Adoption

The evolution of web applications toward high concurrency, real-time interactions, and non-blocking operations has made it necessary to adopt asynchronous programming models. Traditionally, Flask was designed as a synchronous WSGI-based framework, which limited its ability to handle long-lived or concurrent connections efficiently (Pallets Projects, 2025). However, the growing need for asynchronous features has led to the integration of async/await patterns in Flask, allowing developers to create asynchronous view functions. While Flask's support for async relies mainly on synchronous WSGI workflows through background thread execution, it offers a practical solution for applications that need some level of concurrency without completely moving to ASGI-native frameworks like FastAPI (Uvicorn, n.d.; Pallets Projects, 2025).

This change helps Flask manage use cases like real-time notifications, chat applications, and long-running I/O tasks better while keeping backward compatibility. However, the asynchronous model in Flask is still not as efficient as frameworks built specifically for ASGI. Developers need to weigh the trade-offs between code simplicity, backward compatibility, and the concurrency needs of modern applications.

6.5.2. Role in AI-Driven and Data-Driven Web Applications

The combination of web development and artificial intelligence (AI) has created new opportunities for dynamic web services. Flask's simple and modular design makes it a good fit for AI-driven applications. The backend often needs to merge complex machine learning models or data pipelines (Grinberg, 2018; Chollet, 2021). For example, Flask can act as an API layer to host predictive models, perform inference, and send results to client applications through RESTful endpoints. Its ability to work with Python's extensive scientific computing ecosystem—including libraries like NumPy, pandas, TensorFlow, and PyTorch—supports the easy integration of AI features into web applications (Abadi et al., 2016; Raschka & Mirjalili, 2021).

In data-driven settings, Flask supports real-time dashboards, dynamic reporting, and interactive visualization platforms. By using templating engines like Jinja2 and asynchronous extensions, developers can create websites that update content based on user input or streaming data. This makes Flask a flexible and practical option for projects in areas like finance, healthcare, and e-commerce, where responding to data and using predictive analytics are becoming more important.

6.5.3. Serverless Deployment

Serverless computing has changed how we deploy and scale web applications. By simplifying infrastructure management, serverless platforms let developers concentrate on application logic instead of handling server setup and upkeep (Baldini et al., 2017). Flask applications can be deployed to

serverless environments like AWS Lambda, Google Cloud Functions, or Azure Functions with tools like Zappa or Serverless Framework. These tools turn Flask WSGI applications into handlers that work with serverless systems, enabling event-driven execution and on-demand scaling (Baldini et al., 2017; AWS, 2023).

Serverless deployment provides several benefits, such as lower costs, automatic scaling, and easier maintenance. However, developers need to deal with latency issues, cold-start times, and possible limits on long-lived connections or asynchronous operations. As cloud providers keep improving their serverless services, Flask's modular design helps applications adjust well, staying relevant in the age of event-driven cloud-native services.

6.5.4. Security Challenges in Python Web Frameworks

Security is a major concern in the design and operation of web frameworks. Python web frameworks, like Flask, face common vulnerabilities such as SQL injection, cross-site scripting (XSS), cross-site request forgery (CSRF), and poor authentication or session management (Wang et al., 2018). While Flask offers basic protections like automatic escaping in Jinja2 templates and session management, it depends on developers to add further security measures through extensions or best practices (Pallets Projects, 2025).

New security threats include API abuse, insecure dependencies, and attacks targeting asynchronous endpoints in hybrid synchronous/asynchronous applications. Research highlights the importance of continuous monitoring, dependency auditing, automated testing, and following security guidelines for Python web applications (Almorsy et al., 2018; Wang et al., 2018). The modular and clear nature of Flask encourages developers to intentionally integrate security measures, which can be both beneficial and challenging, especially for large-scale projects.

6.5.5. Flask's Relevance in 2030 and Beyond

Looking toward 2030, Flask's relevance will depend on how well it adjusts to changing technological trends and developer expectations. Its main ideas, simplicity, flexibility, and clarity, match up with educational goals and agile development methods. This alignment ensures that it will continue to be used in academia and small-to-medium production environments (Grinberg, 2018; Pallets Projects, 2025).

The expected growth of microservices architectures, serverless computing, and AI-driven web applications suggests that Flask will remain a useful backbone for modular, API-focused services. Competing frameworks, especially those designed for asynchronous processing and type-safe APIs like FastAPI, may become more popular for high-performance applications. However, Flask's large ecosystem, community support, and educational value make it a solid choice for flexible and

maintainable web applications. Its development will likely emphasize better asynchronous support, easy integration with cloud-native services, and stronger security tools. This will help maintain its relevance into the next decade.

7. Conclusion

Flask has become a flexible and influential framework in the Python web development world. Its simple, modular, and adaptable design enables developers to create dynamic websites, RESTful APIs, and microservices effectively. It suits many use cases, from academic projects to production-level applications. This essay has highlighted Flask's benefits, such as its easy learning curve, modularity, seamless integration with databases and third-party libraries, and compatibility with Python's broad scientific and machine learning resources (Grinberg, 2018; Pallets Projects, 2025). These strengths keep Flask relevant in both educational and professional settings. It offers developers a solid foundation to learn web development concepts while providing reliable solutions for real-world applications.

The discussion of Flask compared to other frameworks highlights the need to select the right tool for the project. Full-stack frameworks like Django provide a complete, ready-to-use setup with built-in authentication, ORM, and admin interfaces. This can speed up development for large, content-heavy applications. In contrast, Flask's microframework approach gives developers more freedom and control. They can choose the features they need, resulting in lighter and easier-to-maintain applications, especially where flexibility and modularity are important (Django Software Foundation, 2025; Grinberg, 2018). FastAPI, a new asynchronous framework, shows the rising importance of designing high-concurrency, type-safe APIs. Still, Flask's ongoing popularity shows that many people value simplicity and composability, particularly for educational purposes, prototyping, and small to medium-sized applications.

Python's lasting popularity in web development comes from its clear syntax, wide library support, and large developer community. The language's flexibility allows it to connect web development with data science, machine learning, and artificial intelligence. This creates a rich space for innovative web applications (Chollet, 2021; Abadi et al., 2016). Flask, with its simple but adaptable design, acts as a link between fundamental web development concepts and new trends like microservices, cloud-native deployment, serverless computing, and AI-driven interfaces. Its use in schools, startups, and production systems highlights its importance as both a teaching tool and a practical framework that meets today's development challenges.

In conclusion, Flask remains relevant because it balances simplicity and power. It offers a framework that is easy for beginners and flexible for professionals. As web development changes, Flask's modularity and compatibility with new technologies help it keep up with future trends, such as asynchronous programming, AI integration, or cloud-native architectures. By allowing developers to create dynamic, maintainable, and scalable applications, Flask showcases the lasting strengths of Python as a web development language. This reinforces its role as an essential tool for teaching and professional use.

8. References

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., … Zheng, X. (2016). *TensorFlow: Large-scale machine learning on heterogeneous systems*. https://www.tensorflow.org

Almorsy, M., Grundy, J., & Müller, I. (2018). Security challenges in cloud applications. *Journal of Cloud Computing*, 7(1), 1–18. https://doi.org/10.1186/s13677-018-0127-4

ASGI Team. (n.d.). *ASGI documentation* (3.0). https://asgi.readthedocs.io/

ASGI Team. (n.d.). *ASGI specifications.* https://asgi.readthedocs.io/en/stable/specs/index.html

AWS. (2023). *AWS Lambda: Serverless computing*. https://aws.amazon.com/lambda/

Baldini, I., Castro, P., Chang, K., Cheng, P., Fink, S., Ishakian, V., ... & Suter, P. (2017). Serverless computing: Current trends and open problems. *Research Advances in Cloud Computing*, 1–20.

Chollet, F. (2021). *Deep learning with Python* (2nd ed.). Manning Publications.

Dhir, S., & Mital, A. (2013). *Web Engineering and Applications*. Springer.

Django Software Foundation. (2025). *Meet Django*. https://www.djangoproject.com/

Django Software Foundation. (2025). *Overview*. https://www.djangoproject.com/overview/

Fielding, R. T. (2000). *Architectural styles and the design of network-based software architectures* (Doctoral dissertation, University of California, Irvine). https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

Flask Documentation. (2023). *Flask: Design Decisions*. Pallets Projects. Retrieved from https://flask.palletsprojects.com

Grinberg, M. (2018). *Flask Web Development: Developing Web Applications with Python*. O'Reilly Media.

Gerasimov, A., et al. (2024). Web application testing—Challenges and opportunities. *Journal of Systems and Software*. https://www.sciencedirect.com/science/article/pii/S0164121224002309

Grinberg, M. (n.d.). *The Flask Mega-Tutorial*. https://blog.miguelgrinberg.com/post/the-flask-mega-tutorial-part-i-hello-world

Grinberg, M. (2018). *Flask web development: Developing web applications with Python*. O'Reilly Media.

Jinja Documentation. (2025a). *API (autoescape guidance)* (v3.1.x). https://jinja.palletsprojects.com/en/stable/api/

Jinja Documentation. (2025b). *Template designer documentation* (v3.1.x). https://jinja.palletsprojects.com/en/stable/templates/

Jinja Documentation. (2025c). *Extensions* (v3.1.x). https://jinja.palletsprojects.com/en/stable/extensions/

Liquid Web. (n.d.). *What is WSGI (Web Server Gateway Interface)?* https://www.liquidweb.com/blog/what-is-wsgi/

Lutz, M. (2013). *Learning Python* (5th ed.). O'Reilly Media.

Pallets Projects. (2025). *Design decisions in Flask*. https://flask.palletsprojects.com/en/stable/design/

Pallets Projects. (2018). *Flask documentation (PDF compilation)*. https://media.readthedocs.org/pdf/pallet/latest/pallet.pdf

Pallets Projects. (2025). *Welcome to Flask* (v3.1.x). https://flask.palletsprojects.com/

Pallets Projects. (2025). *Welcome to Flask*. https://flask.palletsprojects.com/

Pallets Projects. (2025). *Design decisions in Flask* (v3.1.x). https://flask.palletsprojects.com/en/stable/design/

Pylons Project. (2023d). *Advanced Pyramid design features (view predicates)*. https://docs.pylonsproject.org/projects/pyramid/en/latest/narr/advanced-features.html

Pylons Project. (2023a). *Pyramid introduction* (v2.0.2). https://docs.pylonsproject.org/projects/pyramid/en/latest/narr/introduction.html

Pylons Project. (2023c). *Creating a Pyramid project (cookiecutter options)*. https://docs.pylonsproject.org/projects/pyramid/en/latest/narr/project.html

Pylons Project. (2023b). *Quick tour of Pyramid* (v2.0.2). https://docs.pylonsproject.org/projects/pyramid/en/latest/quick_tour.html

Pylons Project. (2023e). *URL dispatch*. https://docs.pylonsproject.org/projects/pyramid/en/latest/narr/urldispatch.html

Netflix Technology Blog. (2018). *The Netflix TechBlog on Microservices and Infrastructure.* https://netflixtechblog.com/

Netflix Technology Blog. (2021). *Selected engineering posts referencing Flask-based services*. https://netflixtechblog.com/

Netflix Technology Blog. (2021). *Selected engineering posts referencing service patterns.* https://netflixtechblog.com/

Ronacher, A. (2010). *Flask: Introduction and Design Philosophy*. Pocoo Projects.

University of Warwick. (2022). *CS139 Web Applications—Flask and Jinja introduction (course material).* https://warwick.ac.uk/fac/sci/dcs/teaching/material/cs139/

Uvicorn. (n.d.). *Uvicorn ASGI server*. https://www.uvicorn.org/

Uvicorn. (n.d.). *Uvicorn ASGI server (notes on Django Channels and ASGI/Quart)*. https://www.uvicorn.org/

Van Rossum, G., & Drake, F. L. (2009). *Python 3 Reference Manual*. CreateSpace Independent Publishing.

Wang, H., Chen, J., & Wang, Y. (2018). "Security Vulnerabilities in Python Web Applications: An Empirical Study." *Proceedings of the 2018 IEEE Symposium on Security and Privacy Workshops (SPW)*.

Wang, H., Chen, J., & Wang, Y. (2018). Security vulnerabilities in Python web applications: An empirical study. *Proceedings of the IEEE Symposium on Security and Privacy Workshops (SPW)*, 142–149.

Wikipedia. (2025). *Flask (web framework)*. Retrieved from https://en.wikipedia.org/wiki/Flask_(web_framework)

9. Appendix

9.1. The main project code(emtehan)

```
import pandas as pd
import numpy as np
from sqlalchemy import create_engine
from sqlalchemy.exc import SQLAlchemyError
from bokeh.plotting import figure, show
from bokeh.io import output_notebook
import math

# ----------------- Exceptions -----------------
class DataError(Exception):
    """Custom exception for data handling errors"""
    pass



# ----------------- Base Class -----------------
class DataLoader:
    """Base class for loading CSV files into pandas DataFrames"""
    def __init__(self, filepath: str):
        self.filepath = filepath

    def load(self) -> pd.DataFrame:
        """Load CSV file"""
        try:
            return pd.read_csv(self.filepath)
        except FileNotFoundError:
            raise DataError(f"File not found: {self.filepath}")
        except pd.errors.EmptyDataError:
            raise DataError(f"File empty: {self.filepath}")



# ----------------- Inherited Class -----------------
class TrainingDataLoader(DataLoader):
    """Loads training datasets (inherits DataLoader)"""
```

```
    pass


# ---------- Database Handler ----------
class DatabaseHandler:
    """Handles saving and loading data to SQLite"""

    def __init__(self, db_name="project.db"):
        # Check if db_name is a full SQLAlchemy URL
        if db_name.startswith("sqlite://"):
            self.engine = create_engine(db_name)
        else:
            self.engine = create_engine(f"sqlite:///{db_name}")

    def save_dataframe(self, df: pd.DataFrame, table_name: str):
        """Save DataFrame into SQLite table"""
        try:
            df.to_sql(table_name, self.engine, if_exists="replace", index=False)
        except SQLAlchemyError as e:
            raise DataError(f"Database error: {e}")


# ---------- Model Selector ----------
class ModelSelector:
    """Selects best fit ideal functions using least squares"""
    def __init__(self, training_df: pd.DataFrame, ideal_df: pd.DataFrame):
        self.training_df = training_df
        self.ideal_df = ideal_df

    def find_best_fits(self) -> dict:
        """Return dict of 4 best matching ideal functions"""
        best_funcs = {}
        for col in self.training_df.columns[1:]:  # skip X
            y_train = self.training_df[col].values
            min_error = float("inf")
```

```
            best_func = None
            for ideal_col in self.ideal_df.columns[1:]:
                y_ideal = self.ideal_df[ideal_col].values
                error = np.sum((y_train - y_ideal)**2)
                if error < min_error:
                    min_error = error
                    best_func = ideal_col
            best_funcs[col] = best_func
        return best_funcs


# ----------------- Test Data Mapper -----------------
class TestDataMapper:
    """Maps test data to chosen functions if within allowed deviation"""
    def __init__(self, test_df: pd.DataFrame, ideal_df: pd.DataFrame, mapping: dict, max_dev: dict):
        self.test_df = test_df
        self.ideal_df = ideal_df
        self.mapping = mapping
        self.max_dev = max_dev

    def map_points(self):
        """Map test data points to best ideal functions if within allowed deviation"""
        import math
        import pandas as pd

        results = []

        # Normalize column names
        self.test_df.rename(columns=str.upper, inplace=True)
        self.ideal_df.rename(columns=str.upper, inplace=True)

        for _, row in self.test_df.iterrows():
            if pd.isna(row["X"]) or pd.isna(row["Y"]):
                continue  # skip empty rows
```

```
            x, y = row["X"], row["Y"]

            for train_col, ideal_col in self.mapping.items():
                subset = self.ideal_df.loc[self.ideal_df["X"] == x, ideal_col]
                if subset.empty:
                    continue  # skip if no matching X
                y_ideal = subset.values[0]
                deviation = abs(y - y_ideal)
                if deviation <= self.max_dev[train_col] * math.sqrt(2):
                    results.append({
                        "X": x,
                        "Y": y,
                        "DeltaY": deviation,
                        "IdealFunction": ideal_col
                    })

        return pd.DataFrame(results)


# ---------- Visualizer ----------
from bokeh.plotting import figure, show
from bokeh.models import HoverTool, ColumnDataSource
from bokeh.palettes import Category10
from bokeh.io import output_file  # or output_notebook if in notebook


class Visualizer:
    """Visualizes training, ideal, and test mappings"""
    def plot(self, training_df, ideal_df, test_df):
        p = figure(title="Training vs Ideal vs Test Data", x_axis_label="X", y_axis_label="Y")
        # Training data
        for col in training_df.columns[1:]:
            p.line(training_df["X"], training_df[col], legend_label=f"Train {col}")
        # Test data
        p.scatter(test_df["X"], test_df["Y"], legend_label="Test Data", size=6, color="red")
```

```
        show(p)



# ---------- Main ----------
def main():
    # Load data
    train =
TrainingDataLoader(r"C:\Users\Marzieh\PycharmProjects\PythonProject\training.CSV").load()
    ideal = DataLoader(r"C:\Users\Marzieh\PycharmProjects\PythonProject\ideal.csv").load()
    test = DataLoader(r"C:\Users\Marzieh\PycharmProjects\PythonProject\test.csv").load()

    # Save to DB
    db = DatabaseHandler()
    db.save_dataframe(train, "training_data")
    db.save_dataframe(ideal, "ideal_functions")
    db.save_dataframe(test, "test_data")

    # Select best fits
    selector = ModelSelector(train, ideal)
    best_mapping = selector.find_best_fits()
    print("Best mapping:", best_mapping)

    # Compute max deviations
    max_dev = {}
    for train_col, ideal_col in best_mapping.items():
        max_dev[train_col] = np.max(abs(train[train_col].values - ideal[ideal_col].values))

    # Map test data
    mapper = TestDataMapper(test, ideal, best_mapping, max_dev)
    mapped = mapper.map_points()
    db.save_dataframe(mapped, "mapped_test_data")

    # Visualize
    vis = Visualizer()
```

```
    vis.plot(train, ideal, test)


if __name__ == "__main__":
    main()
```

9.2. The project test code

```
import pytest
import pandas as pd
import numpy as np
from io import StringIO
from emtehan import (
    DataLoader,
    TrainingDataLoader,
    DatabaseHandler,
    ModelSelector,
    TestDataMapper,
    DataError,
    Visualizer
)

# ------------------ DataLoader Tests ------------------
def test_load_csv_success(tmp_path):
    csv_file = tmp_path / "data.csv"
    csv_file.write_text("X,Y\n1,2\n3,4")
    loader = DataLoader(str(csv_file))
    df = loader.load()
    assert df.shape == (2, 2)
    assert list(df.columns) == ["X", "Y"]

def test_load_csv_file_not_found():
    loader = DataLoader("nonexistent.csv")
    with pytest.raises(DataError, match="File not found"):
        loader.load()

def test_load_csv_empty_file(tmp_path):
    csv_file = tmp_path / "empty.csv"
    csv_file.write_text("")
    loader = DataLoader(str(csv_file))
```

```
    with pytest.raises(DataError, match="File empty"):
        loader.load()

# ------------------ DatabaseHandler Tests ------------------
def test_save_dataframe_in_memory():
    db = DatabaseHandler("sqlite:///:memory:")
    df = pd.DataFrame({"X": [1,2], "Y": [3,4]})
    # Should not raise
    db.save_dataframe(df, "test_table")

# ------------------ ModelSelector Tests ------------------
def test_find_best_fits():
    train_df = pd.DataFrame({"X": [1,2], "A": [1,2], "B": [2,3]})
    ideal_df = pd.DataFrame({"X": [1,2], "I1": [1,2], "I2": [2,3]})
    selector = ModelSelector(train_df, ideal_df)
    mapping = selector.find_best_fits()
    assert mapping == {"A": "I1", "B": "I2"}

# ------------------ TestDataMapper Tests ------------------
def test_map_points_within_deviation():
    test_df = pd.DataFrame({"X": [1], "Y": [1.1]})
    ideal_df = pd.DataFrame({"X": [1], "I1": [1.0]})
    mapping = {"A": "I1"}
    max_dev = {"A": 0.2}
    mapper = TestDataMapper(test_df, ideal_df, mapping, max_dev)
    mapped = mapper.map_points()
    assert mapped.shape[0] == 1
    assert mapped.iloc[0]["DeltaY"] == pytest.approx(0.1)

def test_map_points_out_of_deviation():
    test_df = pd.DataFrame({"X": [1], "Y": [1.5]})
    ideal_df = pd.DataFrame({"X": [1], "I1": [1.0]})
    mapping = {"A": "I1"}
    max_dev = {"A": 0.2}
    mapper = TestDataMapper(test_df, ideal_df, mapping, max_dev)
```

```
    mapped = mapper.map_points()
    assert mapped.shape[0] == 0


# ------------------ Visualizer Tests ------------------
def test_visualizer_runs():
    training_df = pd.DataFrame({"X": [1,2], "A": [1,2]})
    ideal_df = pd.DataFrame({"X": [1,2], "I1": [1,2]})
    test_df = pd.DataFrame({"X": [1], "Y": [1]})
    vis = Visualizer()
    # Should not raise
    vis.plot(training_df, ideal_df, test_df)

def test_find_best_fits():
    train_df = pd.DataFrame({"X": [1,2], "A": [1,2], "B": [2,3]})
    ideal_df = pd.DataFrame({"X": [1,2], "I1": [1,2], "I2": [2,3]})
    selector = ModelSelector(train_df, ideal_df)
    mapping = selector.find_best_fits()
    print("Mapping found:", mapping)  # 👈 This will show only if you run pytest -s
    assert mapping == {"A": "I1", "B": "I2"}
```

9.3 The GitHub commands

```
# 1. Clone the repository and checkout develop branch
git clone -b develop https://github.com/username/project.git && cd project

# 2. Create and switch to a new feature branch
git checkout -b feature/new-function

# 3. Stage all changes
git add .

# 4. Commit changes with a descriptive message
git commit -m "Add new function for X feature"

# 5. Push feature branch to GitHub
git push -u origin feature/new-function

# 6. (PR step) Create a Pull Request via GitHub UI
# 7. After merge, update local develop branch
git checkout develop && git pull origin develop
```

9.4. My project GitHub Address:

https://github.com/marzienavaeilavasani/my-project