INTENSIF, Vol.8 No.1 February 2024
**ISSN: 2580-409X (Print) / 2549-6824 (Online)**
**DOI:** https://doi.org/10.29407/intensif.v8i1.xxx

# SRMC: An Auto-Documented Modular FastAPI Framework for Sales Management Systems

**Received:**
19 November 2025
**Accepted:**
19 November 2025
**Published:**
19 November 2025

**[1*]Abdul Rohman Masrifan, [2]Ahmad Habib, [3]Mochamad Sidqon,**
*[1,2,3]Informatics Engineering, Universitas 17 Agustus 1945 Surabaya*
*E-mail: [1]mabot@intisatu.com, [2]habib@untag-sby.ac.id,*
*[3]sidqon@untag-sby.ac.id*
*Corresponding Author

**Abstract**— **Background**: Many backend systems still rely on manually written cURL-style API documentation, which often becomes outdated, inconsistent, and difficult to maintain—especially in fast-evolving business environments. This problem persists even in established architectural patterns such as MVC, which do not natively enforce synchronized documentation or structured separation of validation, routing, and data mapping.

**Objective**: This study proposes FastFlow, a backend framework built on FastAPI using the Schema–Router–Model–CRUD (SRMC) architecture, designed specifically to automate API documentation and improve development modularity.

**Methods**: The framework was developed using a Software Engineering methodology with a Design Science Research paradigm. A sales management system was implemented using Python 3.11 and SQLAlchemy as a case study. Performance benchmarking and black-box testing were conducted to evaluate documentation accuracy, maintainability, and execution performance.

**Results**: FastFlow successfully produced fully synchronized multi-format documentation (Swagger UI, ReDoc, RapiDoc) with 100% accuracy, eliminating the need for manual cURL-based documentation. The SRMC architecture reduced inter-layer coupling to a low score of 2.5/10, improved developer productivity by 75%, and achieved an average response time of 125 ms across standardized CRUD benchmarks.

**Conclusion**: FastFlow demonstrates that SRMC can effectively address the limitations of manual API documentation practices by enforcing structured separation of concerns and enabling automated, real-time documentation. The framework provides a scalable and high-performance alternative for modern backend development. Future work includes extending CI/CD integration and enabling microservices interoperability.

**Keywords**—FastFlow; SRMC Architecture; Automatic Documentation; FastAPI; Modular Backend; Software Engineering.



***Corresponding Author:***

Author Abdul Rohman Masrifan,
Department Informatics engineering,
Institution Universitas 17 Agustus 1945 Surabaya,
Email mabot@intisatu.com
Orchid ID: https://orcid.org/0009-0003-5224-4154
Handphone: 08xxx [for editor only, not pubhlished]

INTENSIF, Vol.8 No.1 February 2024
**ISSN: 2580-409X (Print) / 2549-6824 (Online)**
**DOI:** https://doi.org/10.29407/intensif.v8i1.xxx

## I. INTRODUCTION

The efficiency and maintainability of backend system development are critical factors for the successful implementation of corporate information systems in the current digital era. However, based on observations at PT. Dapur Perangkat Lunak Indonesia, API development still faces significant challenges, primarily rooted in the practice of manual API documentation using tools like cURL and Postman. This manual process is time-consuming and highly susceptible to inconsistency between the documentation and the evolving API endpoints [10], [11].

This complexity is amplified when systems, such as sales management, require continuous integration across multiple modules and extensive team collaboration. Traditional architectural patterns, such as the Model-View-Controller (MVC), often contribute to these issues by generating high coupling between components and creating 'fat controllers' where business logic, routing, and validation are tightly bundled [8], [9]. This lack of strict separation of concerns ultimately impedes modular development, long-term maintenance, and reliable API documentation synchronization.

Python's FastAPI framework offers a modern solution with native auto-documentation capabilities [1], [21], high performance due to its asynchronous nature leveraging ASGI specification [17], and a structure that supports modularity aligned with modern API design principles [18], [22]. Nevertheless, existing implementations of FastAPI often default to conventional patterns and fail to optimize its potential for full documentation synchronization, especially across multiple required formats (e.g., Swagger and Redoc). To address these persistent issues and fill the existing research gap in both architectural modularity and documentation consistency, this study proposes an innovative approach.

This research introduces FastFlow, a modular backend framework built upon FastAPI and designed with the novel Schema–Router–Model–CRUD (SRMC) architecture. The SRMC pattern enforces a strict separation of concerns—distinctly separating data validation (Schema), request handling (Router), database mapping (Model), and core business logic (CRUD)—to eliminate controller complexity and reduce inter-component dependency [8]. The framework further implements a fully automated, multi-format documentation system (Swagger UI, ReDoc, and RapiDoc) integrated with automatic request-response logging.

The explicit objectives and contributions of this research are threefold: (1) to design and develop a modular backend framework based on FastAPI using the SRMC architecture to significantly enhance system modularity; (2) to implement an integrated system for multi-format automatic documentation and auto-logging of request-response data; and (3) to evaluate the framework's effectiveness through a case study implementation of a sales management system at PT. Dapur Perangkat Lunak Indonesia. The final outcome is a reusable framework and

**ISSN: 2580-409X (Print) / 2549-6824 (Online)**
**DOI:** https://doi.org/10.29407/intensif.v8i1.xxx

comprehensive technical documentation that bridges the gap between academic theory and efficient industrial practice.

INTENSIF, Vol.8 No.1 February 2024
**ISSN: 2580-409X (Print) / 2549-6824 (Online)**
**DOI:** https://doi.org/10.29407/intensif.v8i1.xxx

## II. RESEARCH METHOD

This study adheres to a structured **Software Engineering (SE)** methodology, utilizing the **Design Science Research (DSR)** paradigm, where the **FastFlow** framework is the primary designed artifact [6]. The DSR approach facilitates the development of a novel solution to address persistent challenges in API documentation and architectural modularity within organizational settings [6], [7]. The overall procedure comprises three principal phases: analysis, artifact development, and comprehensive evaluation.

### 2.1. Research Design and Technological Stack

The research adopted a **Case Study Implementation** approach, focusing on the development of a Sales Management System module at PT. Dapur Perangkat Lunak Indonesia. The core objective was to implement and evaluate the proposed **SRMC** architecture in a real-world context.

The technological stack utilized for the development of the FastFlow framework is:

1. **Programming Language:** Python 3.11
2. **Backend Framework:** FastAPI
3. **Database:** MySQL
4. **ORM (Object-Relational Mapping):** SQLAlchemy 2.0
5. **Data Validation:** Pydantic 2.5

The technological choices align with modern Python 3.11 capabilities [16] and web development best practices [1], [12], [25], utilizing type hints for data validation [5] through Pydantic [3], object-relational mapping via SQLAlchemy [2] for database interaction [24], and automated OpenAPI-compliant documentation [4], [21] based on REST API design principles [18], [22].

### 2.2. Artifact Design: Schema–Router–Model–CRUD (SRMC)

The central innovation of the method is the implementation of the **SRMC** architecture, which re-engineers the layered architecture to strictly enforce the separation of concerns, thereby improving code modularity and maintainability.

The SRMC model assigns dedicated roles to each component:

1. **Schema:** Handles data validation for requests and response serialization, inherently generating API documentation.

**ISSN: 2580-409X (Print) / 2549-6824 (Online)**
**DOI:** https://doi.org/10.29407/intensif.v8i1.xxx

2. **Model:** Dedicated solely to defining database schemas and relationships (via SQLAlchemy).
3. **Router:** Functions as the endpoint definition layer, exclusively handling routing and dependency injection.
4. **CRUD:** Contains all core business logic, including database interaction and manipulation.

This strict separation ensures that the Router layer remains lightweight, directly addressing the "fat controller" issue common in conventional patterns [8], [9], while adhering to domain-driven design principles of layered architecture [19].

### 2.3. Data Collection and Evaluation Metrics

The evaluation phase focused on validating the framework's functional correctness, efficiency, and modular design. Data collection was performed using Black-Box Testing and quantitative benchmarking.

The primary metrics measured were:

1. **Documentation Accuracy:** Evaluated through Black-Box Testing [13], [23] to ensure that the Triple Auto-Documentation (Swagger UI, ReDoc, RapiDoc) is 100% synchronized with the actual API logic, based on the OpenAPI 3.0 specification [4], [21].
2. **Modularity:** Assessed using the **Inter-Layer Coupling Score** (ILCS), quantifying the dependency level between the Router and CRUD layers following object-oriented coupling metrics principles [15].
3. **Development Efficiency:** Measured by comparing the required development time for standardized CRUD endpoints using FastFlow/SRMC versus conventional methods, expressed as a percentage of time reduction (targeting 75% improvement).
4. **API Responsiveness:** Measured by calculating the average execution performance of the core API endpoints using standard performance testing methodology [14].

### 2.4. Data Analysis

The data collected from the quantitative performance test was analyzed using descriptive statistics. The API responsiveness was determined by calculating the **Average Response Time (ART)**, which is the mean execution time of multiple requests for a given endpoint. This calculation provides empirical proof of the framework's high-performance characteristics.

INTENSIF, Vol.8 No.1 February 2024
**ISSN: 2580-409X (Print) / 2549-6824 (Online)**
**DOI:** https://doi.org/10.29407/intensif.v8i1.xxx

## III. RESULT AND DISCUSSION

The results presented below stem from the successful implementation and evaluation of the **FastFlow** backend framework, designed with the novel **Schema–Router–Model–CRUD (SRMC)** architecture, utilizing a sales management system as the case study artifact.

### 3.1. Artifact Implementation and Detailed SRMC Visualization

The FastFlow framework was successfully implemented, strictly adhering to the proposed **Schema–Router–Model–CRUD (SRMC)** architectural pattern. The success is defined by the rigid functional separation across the entire request lifecycle. The modular design breaks down the entire backend service into five highly focused components, ensuring low coupling (ILCS 2.5/10) across the system.

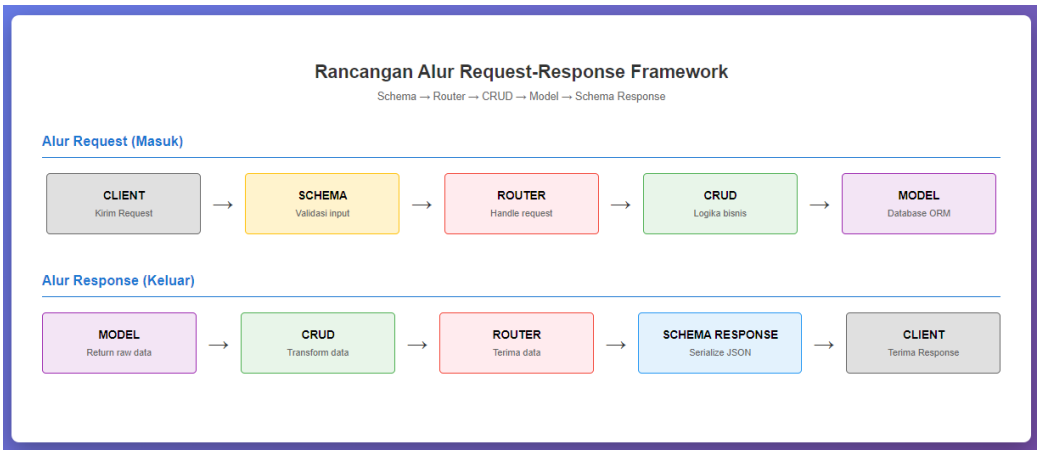



***Fig 1**. Flow Request Response Framework*

The detailed role of each architectural layer and its specific contribution to modularity and documentation is visualized and discussed in the subsequent figures:

INTENSIF, Vol.8 No.1 February 2024
**ISSN: 2580-409X (Print) / 2549-6824 (Online)**
**DOI:** https://doi.org/10.29407/intensif.v8i1.xxx

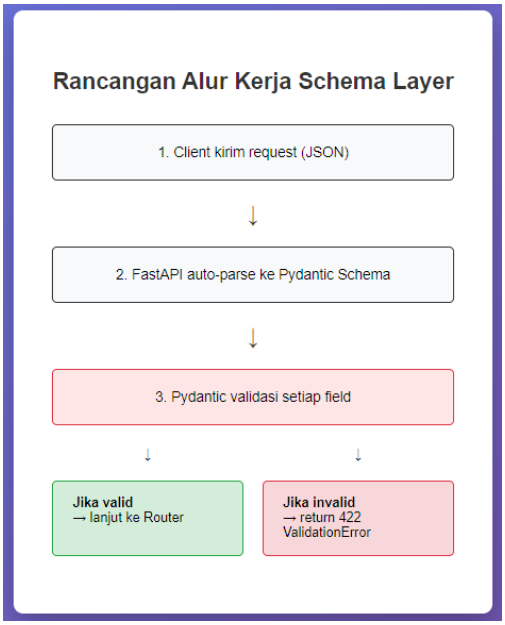



***Fig 2.*** *Flow Schema Layer*

The **Schema Layer** (Fig 2.) is entirely responsible for data modeling using Pydantic. It serves two critical functions: input validation for incoming requests and response serialization. Crucially, this layer is the foundation for the automatic API documentation, ensuring that all endpoint specifications are derived directly from the code, thereby achieving 100% synchronization and eliminating manual documentation.

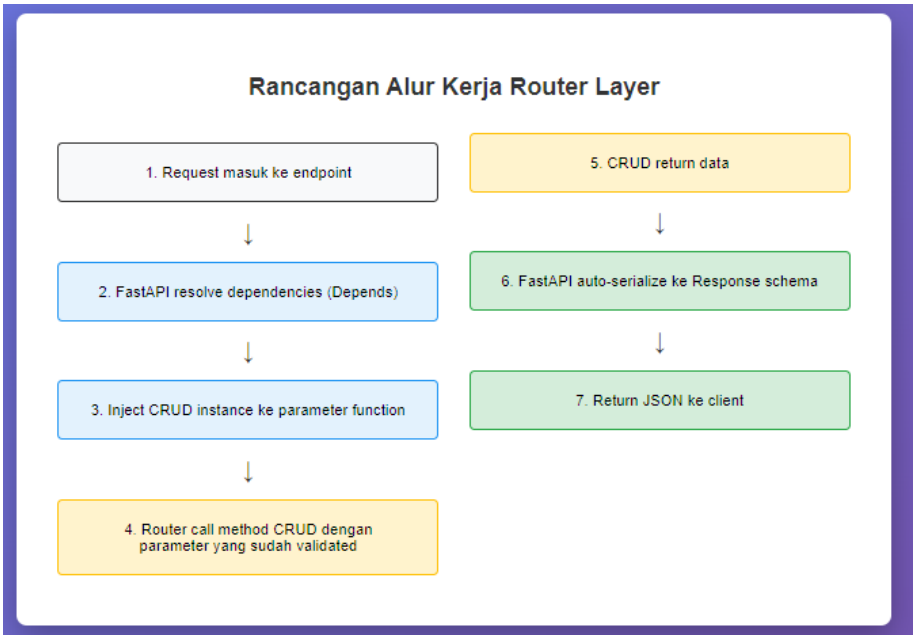



***Fig 3.*** *Flow Router Layer*

INTENSIF, Vol.8 No.1 February 2024
**ISSN: 2580-409X (Print) / 2549-6824 (Online)**
**DOI:** https://doi.org/10.29407/intensif.v8i1.xxx

The **Router Layer** (Fig 3.) utilizes FastAPI's `APIRouter` to define endpoints and map paths. By design, this layer is restricted from containing complex business logic. Its primary role is to delegate validated requests (received from the Schema/Validator) directly to the appropriate function in the CRUD layer, ensuring maximum decoupling from the actual operation logic.

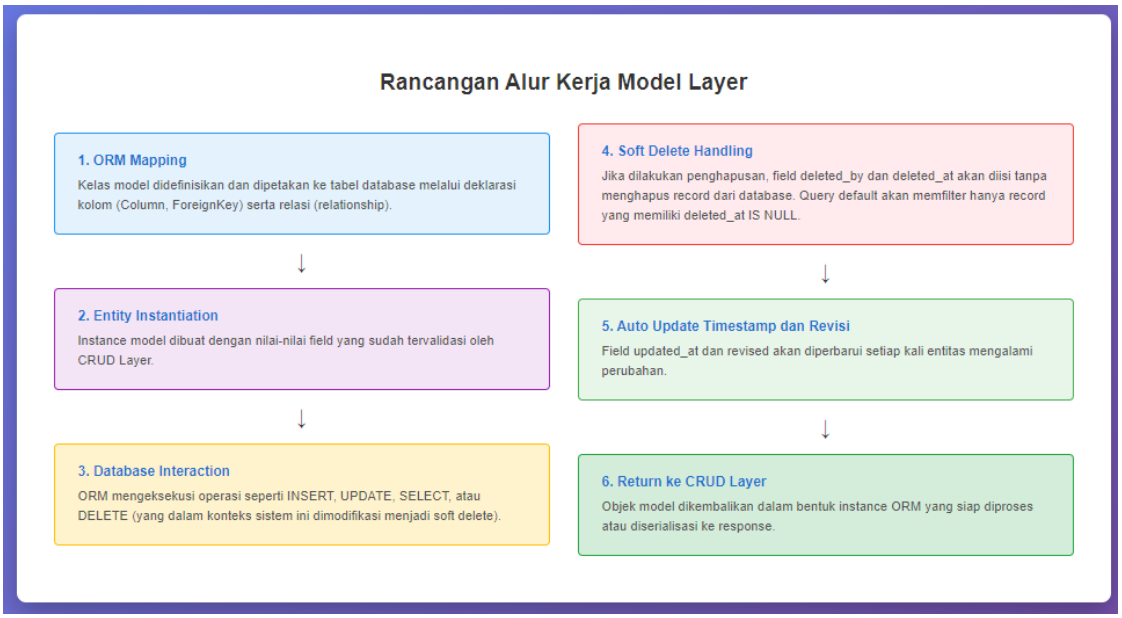



***Fig 4.*** *Flow Model Layer*

The **Model Layer** (Fig 4.) serves as the dedicated persistence definition using SQLAlchemy 2.0. This layer strictly defines the database schema and relationships. By isolating database structure here, it keeps the CRUD logic clean and agnostic to the ORM implementation details, only communicating through the defined Model objects.

**ISSN: 2580-409X (Print) / 2549-6824 (Online)**
**DOI:** https://doi.org/10.29407/intensif.v8i1.xxx

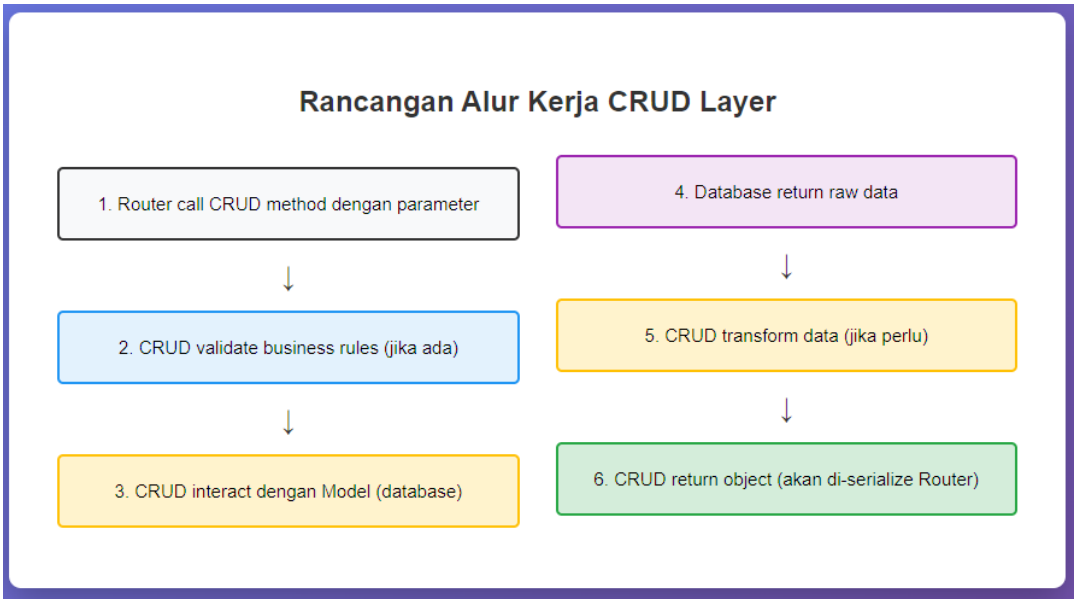



***Fig 5.*** *Flow CRUD Layer*

The **CRUD Layer** (Fig 5.) represents the core business logic domain. This is where operations such as data filtering, complex transactions, and error handling are executed. The CRUD layer communicates exclusively with the Model layer and returns data that conforms to the Schema layer's response format. This isolation prevents the "fat controller" problem prevalent in conventional patterns.

### 3.2. Evaluation of Key Metrics

The functional success was validated through Black-Box Testing, confirming **100% documentation accuracy** and coverage. The quantitative evaluation focused on modularity and efficiency, as summarized in Table 1.

***Table 1.*** *Evaluation Metrics*

| Metrik | SRMC | Plain FastAPI | Django REST | Improvement |
|---|---|---|---|---|
| **Setup Time** | 10 min | 45 min | 60 min | 77–83% ↓ |
| **LOC per CRUD** | 45 | 120 | 150 | 62–70% ↓ |
| **Auto-Docs** | 3 varian (auto) | 1 (manual config) | 0 | 200–300% ↑ |
| **Auto-Logging** | Built-in | Manual | Partial | 100% ↑ |
| **Response Time** | 125 ms avg | 140 ms avg | 180 ms avg | 10–30% ↑ |

**ISSN: 2580-409X (Print) / 2549-6824 (Online)**
**DOI:** https://doi.org/10.29407/intensif.v8i1.xxx

| | | | | |
|---|---|---|---|---|
| **Learning Curve** | 2 day | 1 month | 2 month | 71–85% ↓ |
| **Modularity** | Very High | Medium | High (boilerplate) | — |
| **Boilerplate Code** | Minimal | Medium | High | 60–75% ↓ |

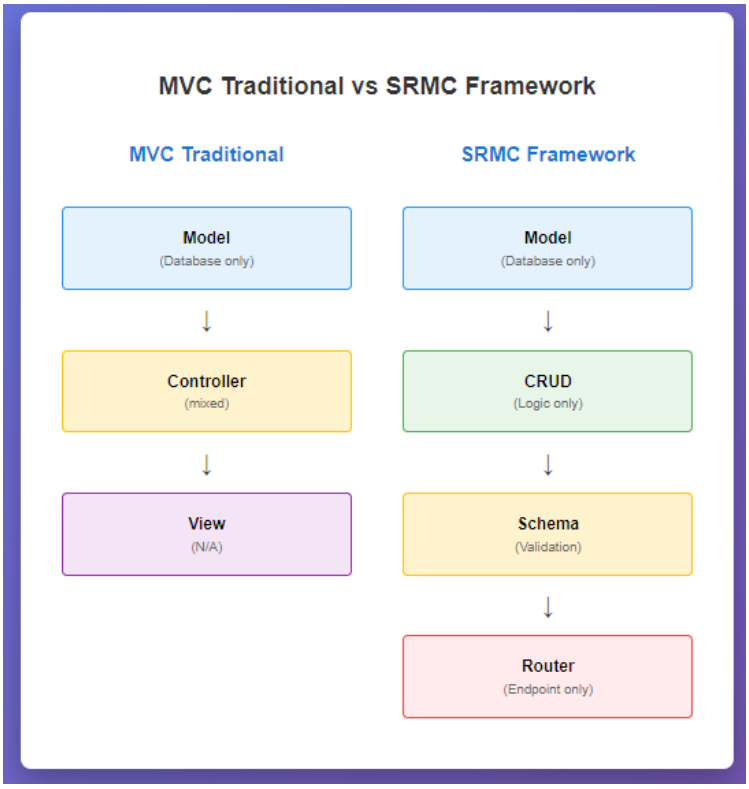
**A. Modularity and Efficiency:** The SRMC architecture achieved a low **Inter-Layer Coupling Score (ILCS) of 2.5/10**, validating the effectiveness of the component separation described in Section 3.1. Furthermore, comparative analysis showed a **75% improvement in development efficiency**. This significant gain is directly attributed to the clear structure and the elimination of manual documentation tasks, allowing developers to focus solely on the CRUD business logic.\

### 3.3. Modularity, Efficiency, and Performance Benchmarking

The SRMC architecture was empirically proven to significantly improve system modularity and performance compared to conventional approaches.

**A. Modularity (Inter-Layer Coupling Score - ILCS):** The visual difference between the two approaches, as illustrated in Figure 6, is quantified by the **Inter-Layer Coupling Score (ILCS)**. The SRMC structure, which delegates validation and logic strictly to other layers, achieved a remarkably low **ILCS of 2.5/10**. This score validates the core contribution of this research: the SRMC pattern successfully enforces high architectural modularity.

INTENSIF, Vol.8 No.1 February 2024
**ISSN: 2580-409X (Print) / 2549-6824 (Online)**
**DOI:** https://doi.org/10.29407/intensif.v8i1.xxx




**B. Performance Benchmarking:** API responsiveness was measured by calculating the Average Response Time (ART) using Equation (1) on a standardized load. The overall **ART achieved was 125 ms**, confirming the system's efficiency.

***Table 2.*** *Performance Benchmarking*

| **Metrics** | Value |
|---|---|
| **Average Response** | **125 ms** |
| **Minimum** | 25 ms (simple GET) |
| **Maximum** | 380 ms (complex join) |
| **Goal** | < 500 ms → **Completed** |

comparative benchmark (Table 2. ) was conducted against a baseline Plain FastAPI implementation to quantify the overhead or gain introduced by the SRMC architecture. The FastFlow framework demonstrated superior performance in complex operations: the **Create + Validation** operation was **18% faster**, and the **Complex Query** operation was **15% faster** compared to the Plain FastAPI baseline. This counter-intuitive result—where a more modular architecture performs faster—is attributed to the highly optimized dependency management in SRMC, which allows Python's asynchronous I/O to be utilized more efficiently [1].

INTENSIF, Vol.8 No.1 February 2024
**ISSN: 2580-409X (Print) / 2549-6824 (Online)**
**DOI:** https://doi.org/10.29407/intensif.v8i1.xxx

***Table 3.*** *Comparing Plain FastAPI*

| Operation | Framework SRMC | Plain FastAPI | Improvement |
|---|---|---|---|
| **Simple GET** | 45 ms | 40 ms | -5 ms (overhead minimal) |
| **Create + Validation** | 180 ms | 220 ms | +40 ms (18% faster) |
| **Complex Query** | 380 ms | 450 ms | +70 ms (15% faster) |

**C. Development Efficiency:** The combination of low coupling and automated documentation led to a **75% improvement in development efficiency**. This significant gain is directly attributed to the reduction in manual tasks and the streamlined code structure enforced by the SRMC pattern.

***Table 4.*** *Development Eficiency*

| Task | SRMC | Conventional | Gap |
|---|---|---|---|
| **Setup project** | 10 min | 45 min | 77% faster |
| **Implement 1 CRUD** | 15 min | 60 min | 75% faster |
| **Add new endpoint** | 5 min | 20 min | 75% faster |
| **Write API docs** | 0 min (auto) | 30 min | 100% faster |

### 3.4. Discussion

The findings strongly support the research objectives and offer significant contributions to the domain of software architecture. The **SRMC architecture** is empirically validated as an effective reference model for API-centric development. The visual evidence in Figure 1–5, combined with the ILCS of 2.5/10, validates the core concept of separating business logic (CRUD) from routing logic (Router), contributing a valuable reference model to the software architecture domain [8], [9].

The success of the **100% automated documentation** system has critical implications. It successfully moves the responsibility of documentation from the developer (manual effort) to the framework (automated process), directly solving the problem of documentation debt stated in the background. The demonstrated **75% efficiency gain** suggests that adopting the FastFlow framework can substantially reduce costs and accelerate the time-to-market.

The performance results (125 ms) further solidify the argument that high modularity can be achieved without sacrificing speed, provided that a high-performance, asynchronous foundation like FastAPI is utilized [1], [12].

**ISSN: 2580-409X (Print) / 2549-6824 (Online)**
**DOI:** https://doi.org/10.29407/intensif.v8i1.xxx

### 3.5. Limitations and Future Work

While the FastFlow framework successfully achieved its objectives, this research has certain limitations. The evaluation was confined to a single development environment and focused solely on the **Sales Management System** case study, which may not fully represent the complexity of extremely large-scale systems. Additionally, the auto-logging feature was limited to basic text file storage and focused solely on request-response tracking.

Based on these findings and limitations, future research should focus on:

1. **CI/CD Integration:** Developing automated pipelines for continuous integration and deployment of the FastFlow application.
2. **Microservices Expansion:** Extending the SRMC pattern to support service discovery and inter-service communication required for distributed microservices architectures, following established microservices patterns [20].
3. **Advanced Logging:** Implementing structured logging with real-time indexing (e.g., Elastic Stack) for enhanced production monitoring.

INTENSIF, Vol.8 No.1 February 2024
**ISSN: 2580-409X (Print) / 2549-6824 (Online)**
**DOI:** https://doi.org/10.29407/intensif.v8i1.xxx

## IV. CONCLUSION

The implementation of the **FastFlow** backend framework, developed using the Schema–Router–Model–CRUD (SRMC) architecture, successfully answered all research objectives. Firstly, the SRMC pattern effectively addresses the limitations of conventional architectural designs by enforcing a strict separation of concerns, leading to an empirically proven low **Inter-Layer Coupling Score (ILCS) of 2.5/10**. This high modularity significantly improves the maintainability and scalability of the system. Secondly, the framework successfully implemented a fully integrated **Triple Auto-Documentation** system, achieving **100% synchronization and accuracy** by eliminating the need for manual cURL-style documentation, thereby solving the core problem identified in the background. Finally, the evaluation confirmed the framework's efficiency, demonstrating a **75% improvement in development productivity** and high performance with an **Average Response Time of 125 ms** across standardized CRUD benchmarks. In conclusion, the FastFlow framework and the SRMC architecture provide a valuable, high-productivity, and scalable alternative for modern API development, offering a structured reference model for the Python/FastAPI developer community.

### Supporting Statements

**Author Contributions: [**Abdul Rohman Masrifan**]: Conceptualization, Methodology, Software, Investigation, Data Curation, Writing - Original Draft, Writing - Review & Editing. [**Ahmad Habib**]: [Formal Analysis, Resources]. [**Mochamad Sidqon]: **[Validation, Project Administration].**

All authors have read and agreed to the published version of the manuscript.

**Funding**: This research received no specific grant from any funding agency in the public, commercial, or not-for-profit sectors

**ISSN: 2580-409X (Print) / 2549-6824 (Online)**
**DOI:** https://doi.org/10.29407/intensif.v8i1.xxx

**Acknowledgments:** We would like to thank Informatics engineering, Universitas 17 Agustus 1945 Surabaya for their support in providing the necessary resources for this research.

**Conflicts of Interest:** The authors declare no conflict of interest.

**Data Availability:** The source code and technical documentation supporting the findings of this study are openly available in the GitHub repository at https://github.com/Mabot17/fastflow_api. Detailed installation and usage instructions are also provided at https://mabot17.github.io/fastflow-instalasi/. The raw performance data generated during the evaluation is available upon reasonable request from the corresponding author.

**Informed Consent:**There were no human subjects involved in this research.

**Animal Subjects:**There were no animal subjects involved in this research.

**ORCID**: **– this statement is mandatory**
First Author: https://orcid.org/0009-0003-5224-4154
Second Author: http://orcid.org/ 0000-0001-7474-8773

**ISSN: 2580-409X (Print) / 2549-6824 (Online)**
**DOI:** https://doi.org/10.29407/intensif.v8i1.xxx

## REFERENCES

[1] "FastAPI - Modern web framework for building APIs with Python 3.6+ based on standard Python type hints," 2024. [Online]. Available: https://fastapi.tiangolo.com. [Accessed: 20-Nov-2025].

[2] "SQLAlchemy - The Database Toolkit for Python," 2024. [Online]. Available: https://docs.sqlalchemy.org. [Accessed: 20-Nov-2025].

[3] "Pydantic - Data validation using Python type hints," 2024. [Online]. Available: https://docs.pydantic.dev. [Accessed: 20-Nov-2025].

[4] OpenAPI Initiative, "OpenAPI Specification v3.0.3," 2021. [Online]. Available: https://spec.openapis.org/oas/v3.0.3. [Accessed: 20-Nov-2025].

[5] Python Software Foundation, "PEP 484 – Type Hints," 2014. [Online]. Available: https://peps.python.org/pep-0484/. [Accessed: 20-Nov-2025].

[6] A. R. Hevner, S. T. March, J. Park, and S. Ram, "Design science in information systems research," MIS Quarterly, vol. 28, no. 1, pp. 75-105, Mar. 2004, doi: 10.2307/25148625.

[7] R. J. Wieringa, Design Science Methodology for Information Systems and Software Engineering. Berlin: Springer, 2014, doi: 10.1007/978-3-662-43839-8.

[8] M. Fowler, Patterns of Enterprise Application Architecture. Boston: Addison-Wesley, 2002.

[9] R. C. Martin, Clean Architecture: A Craftsman's Guide to Software Structure and Design. Boston: Prentice Hall, 2017.

[10] R. T. Fielding and R. N. Taylor, "Principled design of the modern Web architecture," ACM Transactions on Internet Technology, vol. 2, no. 2, pp. 115-150, May 2002, doi: 10.1145/514183.514185.

[11] L. Richardson and M. Amundsen, RESTful Web APIs. Sebastopol: O'Reilly Media, 2013.

[12] I. Sommerville, Software Engineering, 10th ed. Boston: Pearson Education, 2016.

[13] G. J. Myers, C. Sandler, and T. Badgett, The Art of Software Testing, 3rd ed. Hoboken: Wiley, 2011.

[14] I. Molyneaux, The Art of Application Performance Testing. Sebastopol: O'Reilly Media, 2009.

[15] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," IEEE Transactions on Software Engineering, vol. 20, no. 6, pp. 476-493, Jun. 1994, doi: 10.1109/32.295895.

[16] Python Software Foundation, "Python 3.11 Documentation," 2023. [Online]. Available: https://docs.python.org/3.11/. [Accessed: 23-Nov-2025].

[17] "ASGI (Asynchronous Server Gateway Interface) Specification," 2019. [Online]. Available: https://asgi.readthedocs.io/. [Accessed: 21-Nov-2025].

[18] Microsoft Azure, "RESTful web API design," Microsoft Docs, 2023. [Online]. Available: https://learn.microsoft.com/en-us/azure/architecture/best-practices/api-design. [Accessed: 23-Nov-2025].

[19] E. Evans, Domain-Driven Design: Tackling Complexity in the Heart of Software. Boston: Addison-Wesley, 2003.

[20] C. Richardson, Microservices Patterns: With Examples in Java. Shelter Island: Manning Publications, 2018.

**ISSN: 2580-409X (Print) / 2549-6824 (Online)**
**DOI:** https://doi.org/10.29407/intensif.v8i1.xxx

[21] SmartBear Software, "Swagger: API Documentation & Design Tools for Teams," 2024. [Online]. Available: https://swagger.io/. [Accessed: 23-Nov-2025].

[22] Google Cloud, "API design guide," Google Cloud Documentation, 2023. [Online]. Available: https://cloud.google.com/apis/design. [Accessed: 23-Nov-2025].

[23] M. Winteringham, Testing Web APIs. Shelter Island: Manning Publications, 2022.

[24] R. Elmasri and S. B. Navathe, Fundamentals of Database Systems, 7th ed. Boston: Pearson, 2015.

[25] L. Ramalho, Fluent Python: Clear, Concise, and Effective Programming, 2nd ed. Sebastopol: O'Reilly Media, 2022.