incedo

Whitepaper

From Reactive to Proactive: Incedo's Playbook for Pre-Scaling APIs

Performance optimization and API Scaling

At Incedo, we worked with one of our enterprise clients to modernize their real-time security-intelligence API. Anticipating a substantial increase in data volumes, we proactively migrated the API service from Flask to FastAPI framework, which is a modern, high-performance Python web framework that handles requests more efficiently while keeping the entire infrastructure and API contracts unchanged. These behind-the-scenes changes preserved compatibility for existing clients and required no additional hardware.

The result: a roughly 40% reduction in typical response times, materially improved tail latency on heavy queries, and greater throughput — delivering faster, more reliable service to end users with no extra infrastructure cost.

System Infrastructure

Resources Specification

Characteristic	API Server	Relational Database Server	NoSQL Database
CPU Cores	2	4	N/A
Memory (GB)	4	16	N/A
Temporary Storage	500 MB	N/A	N/A
Read Operations/ Second	N/A	N/A	1,000
Write Operations/Second	N/A	N/A	350

Challenges in the Original System (Flask)

The Flask framework processes incoming requests in a **synchronous** manner (one request at a time for each worker). In practical terms, this meant each request had to wait for its turn. For example, if a request needed to wait for a slow database query, the worker handling that request was tied up, and other requests had to wait behind it. Under heavy loads with many requests at the same time, this caused queues and longer wait times.

Additionally, Flask added some processing overhead for every request. Each request involves parsing input, running the business logic, and formatting the output. When traffic is light, this overhead is small, but under heavy load it becomes noticeable. Together, these factors created performance bottlenecks underload:

- **Sequential processing:** Each worker handled one request at a time, so a slow operation blocked others.
- **Request overhead:** The framework's own processing for each request added extra latency when scaled up.
- **Limited concurrency:** There were a fixed number of worker threads. If too many users arrived, new requests would have to wait in line.

As a result, during peak usage the system became slow and less responsive. Response times would spike, and the service could not easily scale to more users without adding more hardware.

Migration to FastAPI

To address these issues, we migrated our code to FastAPI. FastAPI is a newer web framework that is built for high performance. It can support asynchronous processing, but for this migration we kept our implementation synchronous (like Flask) to ensure stability. In other words, we rewrote the service endpoints in FastAPI but left the request-handling model the same. This meant minimal changes and a smooth transition.

Key points of our migration approach:

- **Same APIs and data formats:** We kept all API endpoints, request parameters, and response formats identical. Existing clients continued to work without any changes.
- **Unchanged database logic:** Our data models and database queries were reused. FastAPI worked with the existing database connections and query patterns without modification.
- **Same server resources:** We did not change the server or database capacity. The same CPU, memory, and deployment settings were used before and after.

- **Maintain stability:** By running FastAPI in synchronous mode (the same as Flask), the internal behavior stayed consistent. The migration was essentially a swap of the web framework behind the scenes. Overall, the migration was mostly a behind-the-scenes upgrade: the core functionality and infrastructure remained the same, while the framework handling HTTP requests was replaced.

Why FastAPI Performs Better?

Performance Optimizations

Built on Starlette and Uvicorn, FastAPI uses high-performanc e ASGI servers for efficient request handling.

Faster Parsing

FastAPI uses Pydantic for optimized data validation, reducing overhead when handling requests.

Automatic Documentation

The framework autogenerates OpenAPI/Swagge r docs, cutting manual effort and reducing discrepancies.

Scalability Path

FastAPI makes it easy to move to asynchronous endpoints in the future without a major refactor.

Lower Latency

The internal architecture handles concurrent requests more efficiently, maintaining lower response times.

Cleaner Code

FastAPI encourages type hints and modular design, improving readability and debugging.

Async Compatibility

FastAPI supports async endpoints without architectural rewrites for non-blocking I/0 needs.

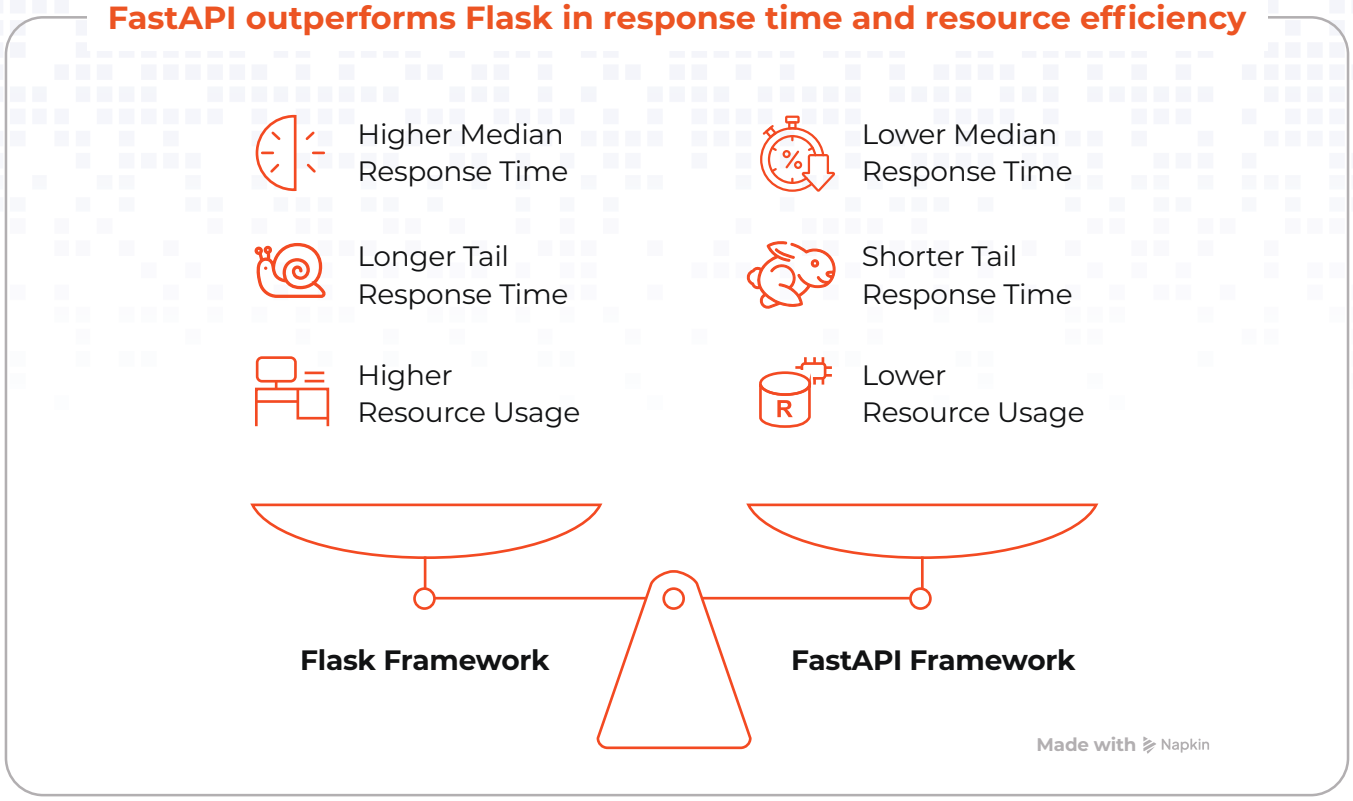
Performance Testing Methodology

We ran performance tests to compare the old Flask system and the new FastAPI system under identical conditions. We simulated multiple requests at the same time, ranging from tens to hundreds of simultaneous requests. For each load scenario, we measured:

- **Median response time (P50):** the typical time to handle a request.
- **Tail response time (P99):** the time taken by the slowest 1% of requests, to capture worst-case delays.

Each scenario was tested multiple times to ensure consistency. We also monitored CPU and database usage to see how resource efficiency has changed.

Performance Improvements



The results showed clear benefits after migration. Under the same loads, the FastAPI-based service was significantly faster. **On average, response times were about 40% lower with FastAPI.** For example, a request that took 1.0 seconds with Flask might take only 0.6 seconds after migration. Some of the heaviest operations became much faster in the order of 80–90% faster. For instance, a complex query that once took over 6 seconds is now completed in under 1 second.

The database saw similar benefits: since queries were completed sooner, each database connection was held for less time, reducing contention and effectively increasing throughput without changing database settings. Overall, migration allowed the service to handle higher traffic more efficiently. The following table summarizes the typical improvements observed:

Metric	Improvement
Median (typical) response time	~40% lower (faster)
Peak response time (99th percentile)	~30% lower (faster)
Throughput (requests/sec)	~25% higher

(Lower is better for latency; higher is better for throughput.)

These improvements mean the API can serve many more concurrent users with the same hardware, and users experience faster response times in practice.

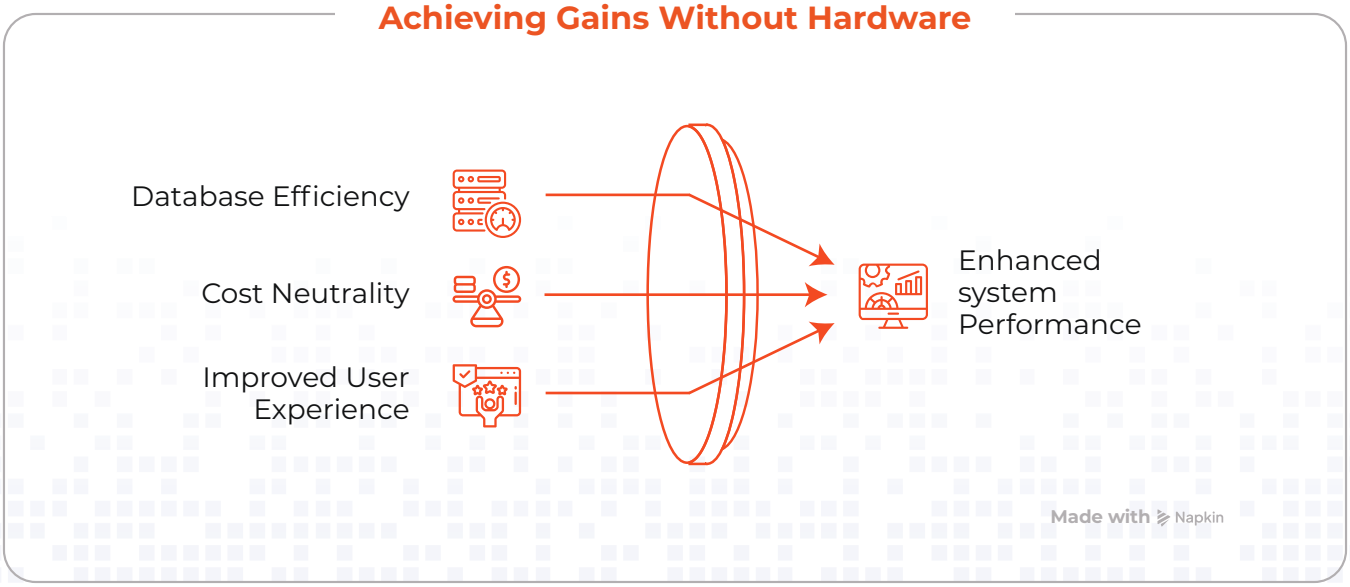
Business Impact

All gains were achieved with no additional hardware. In practice, this meant:

- **Database efficiency:** Faster request handling meant shorter database queries and fewer timeouts. The database handled the same workload with lower CPU load.
- **Cost neutrality:** We continued using the same servers and database capacity. There were no additional infrastructure costs.
- **Improved user experience:** Faster API responses make the system feel more responsive and reliable to end-users.

Conclusion

By modernizing the client's API framework from Flask to FastAPI, Incedo helped the client realize substantial performance improvements while keeping infrastructure costs flat. The approach preserved client interfaces, minimized risk, and unlocked immediate operational and user-experience benefits — proving that framework modernization is a high-impact, low-cost first step in platform optimization.



Call To Action

Modern SaaS businesses demand APIs that scale effortlessly without compromising reliability. At Incedo, we help organizations modernize applications and unlock measurable performance gains.

Looking to scale your SaaS APIs with confidence? Book a quick consultation with us—we'll walk you through tailored optimization strategies. Want to see how these techniques fit into your stack? Get in touch, and we'll share a deep-dive playbook for API performance at scale.

Connect with us today.

About the Author

Ashish Agrawal

Senior Director, Hi-Tech

Ashish has two decades of experience working on cloud native saas application for Cybersecurity domain. Prior to Incedo Ashish has worked with companies like Motorola and Google. In his current role Ashish is working on AI/ML initiatives at scale.

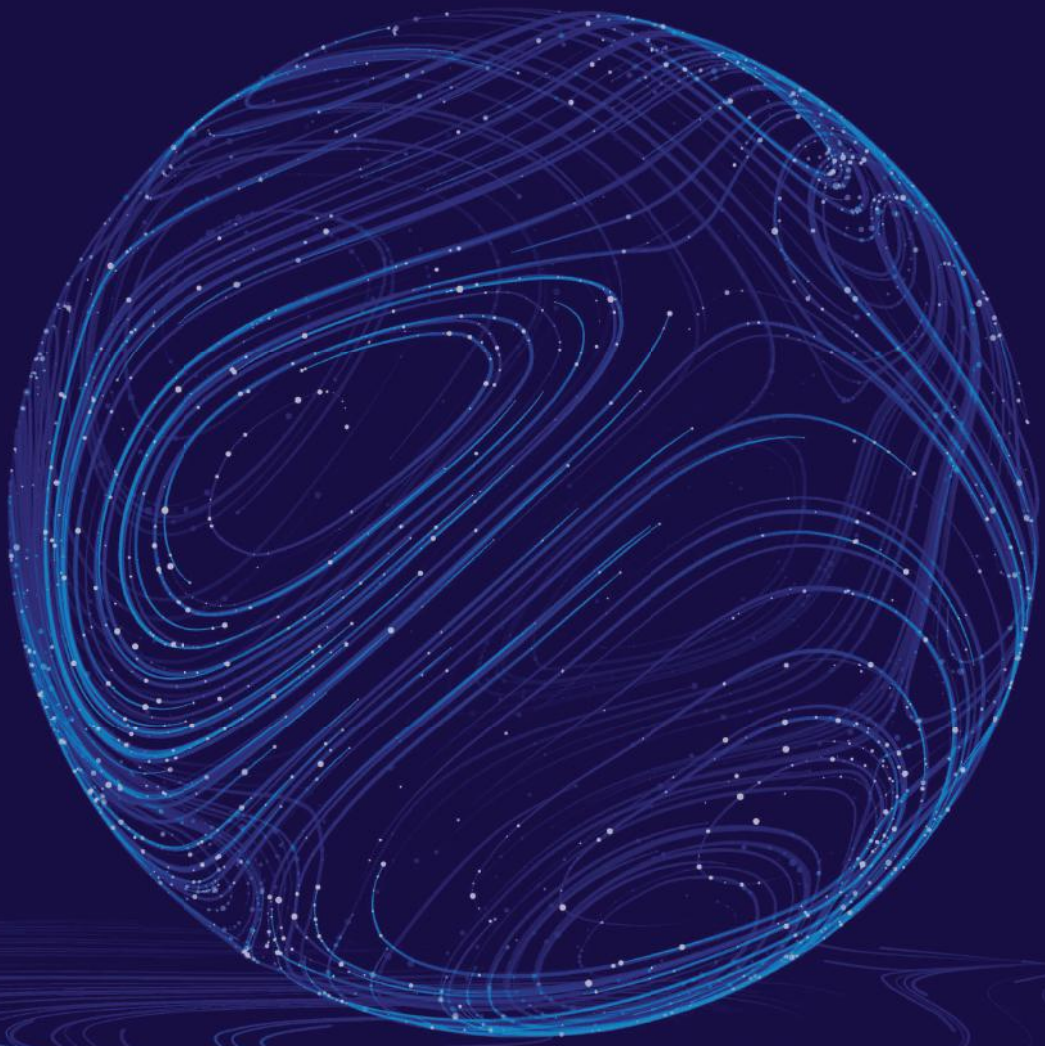
Kishore Singaraju

Sr. Software Engineer, Hi-Tech

Kishore has expertise in backend development and scalable SaaS solutions. Skilled in optimizing and securing high-performance applications. Current focus: Python, Flask/FastAPI, Go, AWS, Microservices, API performance, and automation.

To learn more about how our solutions and platforms can drive your success, please email us at inquiries@incedoinc.com

incedo | Win in the Digital Age



About Incedo

Incedo is a digital transformation expert empowering companies to realize sustainable business impact from their digital investments. Our integrated services and platforms that connect strategy and execution, are built on the foundation of Design, AI, Data, and strong engineering capabilities blended with our deep domain expertise from digital natives.

With over 4,000 professionals in the US, Canada, Latin America, and India and a large, diverse portfolio of long term, Fortune 500 and fast-growing clients worldwide, we work across financial services, telecom, product engineering, and life sciences industries.

9+
Fortune 500 Customers

10+
Global Locations

4k+
Employees Globally

Our Global Presence

India
Gurugram
Chennai
Pune
Bengaluru
Hyderabad

USA
Santa Clara
New Jersey
Dallas
Boston

Canada
Ontario

Mexico
Guadalajara

S/03/10/2025