Python

FastAPI **Latency**

Understanding and Optimizing Response Times

April 29, 2025

Source Code

1. Introduction to API Performance

In modern web development, speed and efficiency are crucial. Understanding fundamental concepts like latency and response time is essential to ensure APIs are scalable and provide an optimal user experience. This article uses FastAPI as a practical example to demonstrate these concepts, but the principles of latency, response time, and percentiles are applicable to all API systems, regardless of the technology used.

1.1. Latency vs Response Time: Understanding the Difference

Although often used interchangeably, **latency** and **response time** are distinct concepts:

- **Response Time:** What the client perceives—includes request processing time (service time), network delays, and queue wait times.
- **Latency:** The duration a request spends waiting to be processed—the time it remains latent before processing begins.

1.2. Measuring Performance: Beyond Averages

When evaluating API performance, average response time is commonly reported. However, averages are not ideal metrics for understanding "typical" system behavior because they don't reflect real user experience.
Using **percentiles** provides a more accurate picture:


Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com

- **50th Percentile (p50) or Median:** Half of requests complete in less time, half take longer.
- **High Percentiles** like p95, p99, and p999: Reveal how bad your outliers are and how they affect a small but significant percentage of users.

High percentiles, also known as "tail latencies," are crucial because they directly affect user experience. For example, Amazon describes response time requirements for internal services in terms of the 99.9th percentile, even though it affects only 1 in 1,000 requests, as these users often have the most valuable accounts.

2. Performance Monitoring with FastAPI

Let's build a small service with FastAPI and simulate load to analyze its behavior. Our goal is to measure and visualize different response time percentiles.

2.1. Creating a FastAPI Application

First, we'll create a simple API with different endpoints that simulate various processing times:

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com

```python
# api_server.py
import asyncio
import random
import time
from fastapi import FastAPI
from fastapi.middleware.cors import CORSMiddleware

app = FastAPI(title="Latency Demo API")

app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

@app.get("/")
async def root():
    """Fast endpoint with consistent response time"""
```

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com

```python
    return {"message": "Latency demonstration API"}

@app.get("/fast")
async def fast_endpoint():
    """Endpoint with fast response (10-30ms)"""
    await asyncio.sleep(random.uniform(0.01, 0.03))
    return {"response_type": "fast"}

@app.get("/medium")
async def medium_endpoint():
    """Endpoint with medium response time (50-150ms)"""
    await asyncio.sleep(random.uniform(0.05, 0.15))
    return {"response_type": "medium"}

@app.get("/slow")
async def slow_endpoint():
    """Endpoint with slow response time (200-500ms)"""
    await asyncio.sleep(random.uniform(0.2, 0.5))
    return {"response_type": "slow"}

```


Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com

PYTHON | **CONCURRENCY & PARALLELISM**

```python
@app.get("/variable")
async def variable_endpoint():
    """
    Endpoint with variable response time:
    - 80% of requests: fast (10-50ms)
    - 15% of requests: medium (100-300ms)
    - 5% of requests: very slow (500-1500ms)
    """
    random_value = random.random()

    if random_value < 0.8:
        await asyncio.sleep(random.uniform(0.01, 0.05))
        category = "fast (80%)"
    elif random_value < 0.95:
        await asyncio.sleep(random.uniform(0.1, 0.3))
        category = "medium (15%)"
    else:
        await asyncio.sleep(random.uniform(0.5, 1.5))
        category = "very slow (5%)"

```

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com

```python
    return {"response_type": "variable", "category": category}

if __name__ == "__main__":
    import uvicorn
    uvicorn.run("api_server:app", host="0.0.0.0", port=8000, reload=True)
```

2.2. Creating a Load Testing Agent

Now, let's create an agent that performs multiple requests to our API and collects performance data:

```python
# load_tester.py
import asyncio
import time
import statistics
import matplotlib.pyplot as plt
import numpy as np
from collections import defaultdict
```

PYTHON | **CONCURRENCY & PARALLELISM**

```python

class LoadTester:
    def __init__(self, base_url="http://localhost:8000"):
        self.base_url = base_url
        self.endpoints = {
            "fast": "/fast",
            "medium": "/medium",
            "slow": "/slow",
            "variable": "/variable"
        }
        self.results = defaultdict(list)

    async def make_request(self, session, endpoint):
        """Makes an HTTP request and measures response time"""
        url = f"{self.base_url}{self.endpoints[endpoint]}"
        start_time = time.time()

        try:
            async with session.get(url) as response:
                await response.json()
```

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com

PYTHON | CONCURRENCY & PARALLELISM

```python
                response_time = (time.time() - start_time) * 1000
                self.results[endpoint].append(response_time)
                return response_time
        except Exception as e:
            print(f"Error in request to {url}: {e}")
            return None

    async def generate_load(self, endpoint, num_requests, concurrency):
        """Generates load for a specific endpoint"""
        async with aiohttp.ClientSession() as session:
            tasks = []
            for _ in range(num_requests):
                tasks.append(self.make_request(session, endpoint))
                if len(tasks) >= concurrency:
                    await asyncio.gather(*tasks)
                    tasks = []

            if tasks:
                await asyncio.gather(*tasks)
```

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com

PYTHON | **CONCURRENCY & PARALLELISM**

```python

    def calculate_percentiles(self, endpoint):
        """Calculates percentiles for response times"""
        if not self.results[endpoint]:
            return {}

        data = sorted(self.results[endpoint])
        return {
            "min": min(data),
            "p50": statistics.median(data),
            "p90": np.percentile(data, 90),
            "p95": np.percentile(data, 95),
            "p99": np.percentile(data, 99),
            "p999": np.percentile(data, 99.9) if len(data) >= 1000 else None,
            "max": max(data),
            "mean": statistics.mean(data),
            "stdev": statistics.stdev(data) if len(data) > 1 else 0
        }

```


Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com

```python
    def plot_results(self):
        """Generates charts to visualize the results"""
        fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(12, 14))

        endpoints = list(self.results.keys())
        metrics = ["p50", "p90", "p95", "p99"]

        x = np.arange(len(endpoints))
        width = 0.2

        for i, metric in enumerate(metrics):
            values = [self.calculate_percentiles(ep)[metric] for ep in endpoints]
            ax1.bar(x + i*width, values, width, label=f’{metric}’)

        ax1.set_ylabel(’Response Time (ms)’)
        ax1.set_title(’Response Time Percentiles by Endpoint’)
        ax1.set_xticks(x + width * 1.5)
        ax1.set_xticklabels(endpoints)
        ax1.legend()
```

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com

PYTHON | **CONCURRENCY & PARALLELISM**

```
        ax1.grid(axis='y', linestyle='--', alpha=0.7)

        if "variable" in self.results and self.results["variable"]:
            data = self.results["variable"]
            bins = np.logspace(np.log10(min(data)), np.log10(max(data)), 50)
            ax2.hist(data, bins=bins, alpha=0.7, color='green')
            ax2.set_xscale('log')
            ax2.set_title('Response Time Distribution (variable endpoint)')
            ax2.set_xlabel('Response Time (ms) - logarithmic scale')
            ax2.set_ylabel('Number of Requests')

            percentiles = self.calculate_percentiles("variable")
            for metric, value in [(k, v) for k, v in percentiles.items()
                                if k in ["p50", "p90", "p95", "p99"] and v is not None]:
                ax2.axvline(x=value, color='red', linestyle='--', alpha=0.6,
```


Alejandro Sánchez Yalí

Software Developer | AI & Blockchain Enthusiast

www.asanchezyali.com

```python
                           label=f"{metric}: {value:.2f}ms")

            ax2.legend()

        plt.tight_layout()
        plt.savefig('latency_results.png', dpi=300)
        plt.close()

    def print_summary(self):
        """Prints a summary of the results"""
        for endpoint in self.results:
            print(f"\n=== Endpoint: {endpoint} ({len(self.results[endpoint])} requests) ===")
            percentiles = self.calculate_percentiles(endpoint)
            for metric, value in percentiles.items():
                if value is not None:
                    print(f"{metric}: {value:.2f} ms")

async def main():
    tester = LoadTester()
```


Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com

```python

    print("Starting load tests...")

    for endpoint in tester.endpoints:
        requests = 1000 if endpoint == "variable" else 200
        print(f"Testing endpoint '{endpoint}' with {requests} requests...")
        await tester.generate_load(endpoint, requests, concurrency=50)

    tester.print_summary()
    tester.plot_results()
    print("Tests completed. Results saved in 'latency_results.png'")

if __name__ == "__main__":
    asyncio.run(main())

# Output
# Starting load tests...
# Testing endpoint 'fast' with 200 requests...
# Testing endpoint 'medium' with 200 requests...
```

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com

```
# Testing endpoint 'slow' with 200 requests...
# Testing endpoint 'variable' with 1000 requests...

# === Endpoint: fast (200 requests) ===
# min: 10.22 ms
# p50: 22.67 ms
# p90: 31.31 ms
# p95: 34.34 ms
# p99: 37.17 ms
# max: 38.25 ms
# mean: 22.79 ms
# stdev: 6.84 ms

# === Endpoint: medium (200 requests) ===
# min: 53.42 ms
# p50: 102.49 ms
# p90: 143.29 ms
# p95: 150.27 ms
# p99: 161.89 ms
# max: 163.58 ms
```

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com

```
# mean: 103.06 ms
# stdev: 30.29 ms

# === Endpoint: slow (200 requests) ===
# min: 206.44 ms
# p50: 361.78 ms
# p90: 476.99 ms
# p95: 489.22 ms
# p99: 501.95 ms
# max: 505.02 ms
# mean: 360.42 ms
# stdev: 84.46 ms

# === Endpoint: variable (1000 requests) ===
# min: 11.65 ms
# p50: 38.14 ms
# p90: 239.78 ms
# p95: 297.38 ms
# p99: 1240.78 ms
# p999: 1452.90 ms
```

```
# max: 1491.94 ms
# mean: 103.07 ms
# stdev: 213.06 ms
# Tests completed. Results saved in 'latency_results.png'
```

2.3. Running and Analyzing the Results

To run our experiment:

1. Start the FastAPI server:

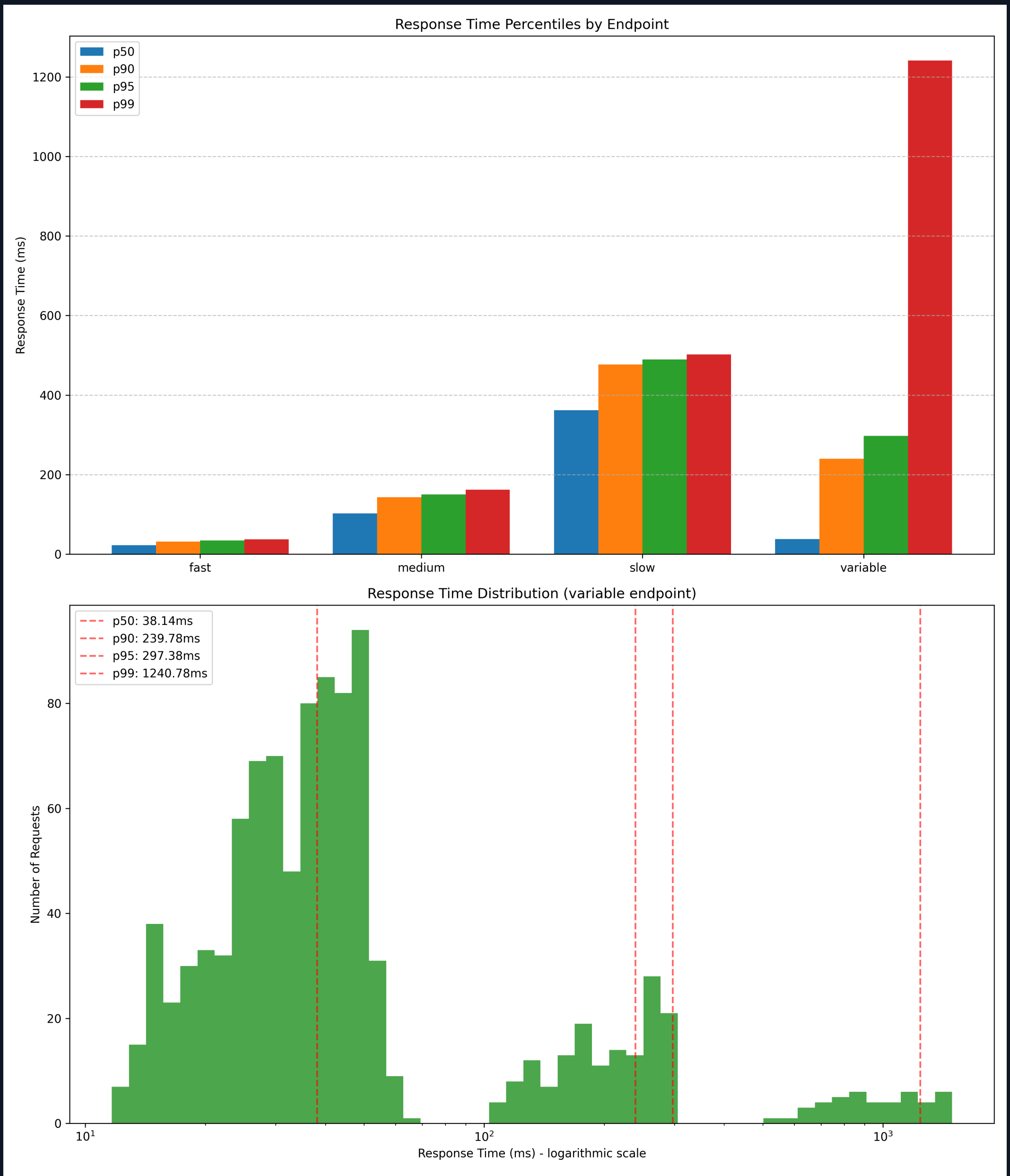
```
python api_server.py
```

2. In another terminal, run the load testing agent:

```
python load_tester.py
```

The agent will generate a file called `latency_results.png` with visualizations of the results, and will also print a summary to the console.

PYTHON | CONCURRENCY & PARALLELISM



Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com

3. Analyzing the Results

The visualizations demonstrate why percentiles provide superior insights for API performance compared to averages:

3.1. Understanding the Variable Endpoint

The most revealing insights come from examining the **variable** endpoint:

- The **median (p50)** is only 38.14ms, indicating most users receive fast responses.
- The **p90** jumps to 239.78ms, revealing that 10% of requests experience significantly slower performance.
- The **p95** at 297.38ms shows further degradation for 5% of requests.
- Most critically, the **p99** at 1240.78ms demonstrates that 1% of users experience response times over 32 times slower than the median.
- The mean (103.07ms) obscures this reality, appearing deceptively moderate despite the extreme outliers.

The histogram's logarithmic distribution confirms these observations, showing three distinct clusters corresponding to the programmed response time categories (80% fast, 15% medium, 5% slow).

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com

3.2. Comparative Analysis Across Endpoints

Comparing the four endpoints reveals important performance patterns:

- **Predictable endpoints** (fast, medium, slow) show relatively consistent behavior where p99 is only 1.3-1.4 times greater than p50.
- The **variable endpoint** exhibits dramatic tail latency, with p99 being 32.5 times higher than p50.
- While the **mean response time** of the variable endpoint (103.07ms) is nearly identical to the medium endpoint (103.06ms), their performance profiles are entirely different—a fact that would be missed by relying solely on averages.

3.3. Practical Benefits of Percentile-Based Monitoring

Using percentiles for performance monitoring offers several concrete advantages:

- **Detecting Hidden Issues:** The variable endpoint's mean suggests acceptable performance, while percentiles reveal severe degradation affecting a minority of requests.
- **User-Centric Metrics:** Percentiles directly correspond to user experience—p95 represents what 5% of your users are experiencing or worse.

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com

- **Early Warning System:** Changes in high percentiles often precede system-wide degradation and can signal emerging problems before they affect most users.
- **SLA Alignment:** Service Level Agreements based on percentiles protect all users, while average-based SLAs may hide systematic failures affecting a subset of users.
- **Infrastructure Sizing:** Understanding tail latency helps properly size infrastructure for peak demands rather than average conditions.

The logarithmic histogram of the variable endpoint particularly emphasizes how response times are not normally distributed—they follow a multi-modal distribution with long tails. This pattern is common in real-world systems due to factors like cache misses, garbage collection pauses, or resource contention.

4. The Tail Latency Amplification Problem

Imagine that your frontend application needs to make several calls to these endpoints to compose a single page. If a page requires 5 API calls:

- With a 5% probability that each call is slow, the probability that at least one call is slow is:

$$1 - (0.95)^5 \approx 23\%$$

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com

This means that although only 5% of individual calls are slow, approximately 23% of page loads will experience delays. This phenomenon is known as "tail latency amplification."

5. Conclusion

When developing and monitoring APIs, it's critical to:

- **Measure Percentiles, Not Just Averages:** High percentiles reveal problems affecting real users that might go unnoticed in means.
- **Understand the Complete Distribution:** Visualize response time histograms to better understand system behavior.
- **Consider Tail Latency Amplification:** Optimizing high percentiles is essential for multi-call user actions.
- **Establish SLOs and SLAs Based on Percentiles:** For example, "p99 must be less than 300ms" is more meaningful than "average time must be less than 100ms."

Modern applications must be designed with these principles to provide consistent, high-quality experiences to all users. Future posts will explore techniques like priority queues, circuit breakers, and caching to mitigate tail latency amplification.

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com

6. References

- Sentry. (2024). *What's the difference between API Latency and API Response Time?*. Link
- Catchpoint. *API Performance Monitoring—Key Metrics and Best Practices*. Link
- FastAPI Documentation. *Benchmarks*. Link
- DeCandia, G., et al. (2007). *Dynamo: Amazon's Highly Available Key-value Store*. ACM SIGOPS Operating Systems Review, 41(6), 205-220.
- Kleppmann, M. (2017). *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. O'Reilly Media, Inc.

- This article was translated, edited and written in collaboration with AI. If you find any inconsistencies or have suggestions for improvement, please don't hesitate to open an issue in our GitHub repository at github or reach out directly.

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com

7. Explore My Other Posts

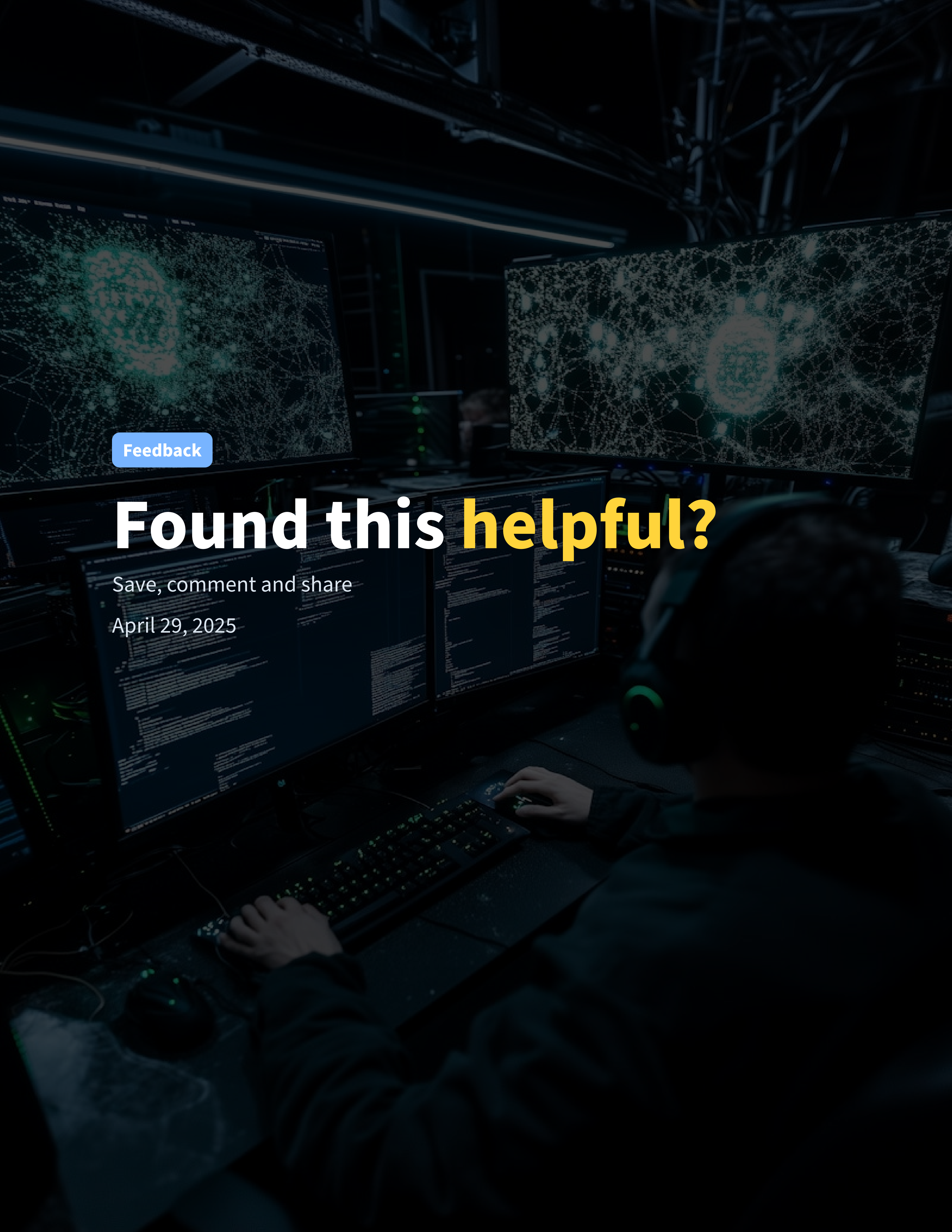
Enjoyed This Content?

Don't miss my previous post about:

Python Generators: Elegant, Memory-Efficient Iterations

Discover how Python Generators can help you process large datasets efficiently, create elegant data pipelines, and write cleaner code with minimal memory footprint.

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com



Feedback

Found this helpful?

Save, comment and share

April 29, 2025