Python

FastAPI Latency & Response Times

Understanding Performance Metrics and Optimizing API Speed in Modern Web Applications

April 15, 2025

Source Code

1. Introduction to API Performance

In modern web development, speed and efficiency are crucial. When developing APIs or web services with FastAPI, understanding fundamental concepts like latency and response time is essential to ensure our applications are scalable and provide an optimal user experience.

1.1. Latency vs Response Time: Understanding the Difference

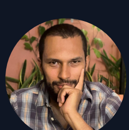
Although often used interchangeably, **latency** and **response time** are distinct concepts:

- **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:

- **50th Percentile (p50) or Median:** Half of requests complete in less time,



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

half take longer

- **High Percentiles** like p95, p99, and p999 (95th, 99th, and 99.9th percentiles): Reveal how bad your outliers are and how they affect a small but significant percentage of users

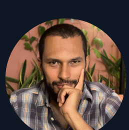
High percentiles, also known as "tail latencies," are crucial because they directly affect user experience. For example, Amazon describes response time requirements for internal services in terms of the 99.9th percentile, even though it affects only 1 in 1,000 requests. Why? Because customers with the slowest requests often have the most data in their accounts—they're the most valuable customers.

2. Performance Monitoring with FastAPI

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

2.1. Creating a FastAPI Application

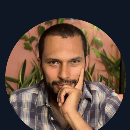
First, we'll create a simple API with different endpoints that simulate various processing times:



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

PYTHON | **CONCURRENCY & PARALLELISM**

```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")

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

```

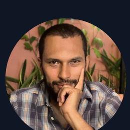


Alejandro Sánchez Yalí

Software Developer | AI & Blockchain Enthusiast

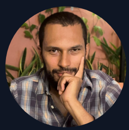
www.asanchezyali.com

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



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

```python
    await asyncio.sleep(random.uniform(0.2, 0.5))
    return {"response_type": "slow"}

@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:
```



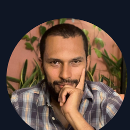
Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com

```python
        await asyncio.sleep(random.uniform(0.5, 1.5))
        category = "very slow (5%)"

    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

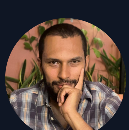
Now, let's create an agent that performs multiple requests to our API and collects performance data:

```python
# load_tester.py
import asyncio
import time
import statistics
```



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

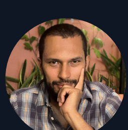
```python
import matplotlib.pyplot as plt
import numpy as np
import aiohttp
from collections import defaultdict

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()
```



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

PYTHON | **CONCURRENCY & PARALLELISM**

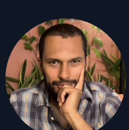
```python

        try:
            async with session.get(url) as response:
                await response.json()
                response_time = (time.time() - start_time) * 1000  # in milliseconds
                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:
```



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

PYTHON | **CONCURRENCY & PARALLELISM**

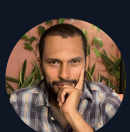
```python
                    await asyncio.gather(*tasks)
                    tasks = []

            if tasks:
                await asyncio.gather(*tasks)

    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,
```



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

PYTHON | **CONCURRENCY & PARALLELISM**

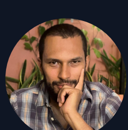
```python
            "max": max(data),
            "mean": statistics.mean(data),
            "stdev": statistics.stdev(data) if len(data) > 1 else 0
        }

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

        # Percentile bar chart
        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}')
```



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

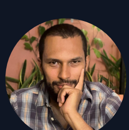
PYTHON | **CONCURRENCY & PARALLELISM**

```python

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

        # Histogram for the variable endpoint
        if "variable" in self.results and self.results["variable"]:
            data = self.results["variable"]
            # Use logarithmic scale for data
            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’)
```



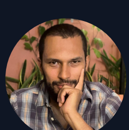
Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com

```python

            # Add vertical lines for percentiles
            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,
                           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:
```



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

```python
            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()

    # Generate load for each endpoint
    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)

    # Print results and generate charts
```



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

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

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

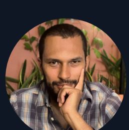
2.3. Running and Analyzing the Results

To run our experiment:
1. Start the FastAPI server:

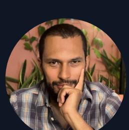
```
python api_server.py
```

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

```
python load_tester.py
```


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

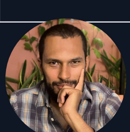
The agent will generate a file called `latency_results.png` with visualizations of the results, and will also print a summary to the console.



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

2.4. Visualization of Results

latency_results.png



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

3. Analyzing the Results

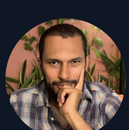
The results clearly show why percentiles are more informative than averages:

- For the **variable** endpoint:
  - The median (p50) is quite low, showing that most requests are fast
  - p95 and p99 reveal the slower requests affecting a small percentage of users
  - The average could be distorted by extreme outliers
- Comparing different endpoints:
  - We observe how response time distribution varies considerably
  - We see the difference between endpoints with predictable vs. variable behavior

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:



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

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

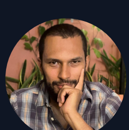
This means that although only 5% of individual calls are slow, approximately 23% of page loads will experience delays. This phenomenon is known as "tail latency amplification" or "tail at scale."

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:** If your frontend or client makes multiple API calls to complete a user action, optimizing high percentiles is essential.
- **Establish SLOs and SLAs Based on Percentiles:** For example, "p99 must be less than 300ms" is a more meaningful metric than "average time must be less than 100ms."

Modern applications must be designed and optimized with these principles in mind to provide consistent, high-quality experiences to all users, not just the average user.



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

In a future post, we'll explore techniques for optimizing these high percentiles, including implementing priority queues, circuit breakers, and caching strategies to mitigate tail latency amplification.

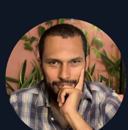
6. Explore My Other Posts

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

Ready to Optimize Your APIs?

You’ve learned how to measure and analyze response times.

Try implementing percentile monitoring in your next project

and watch your applications de-
liver a better experience for all users!