# 0. Topics

Monday, June 16, 2025 10:16 PM

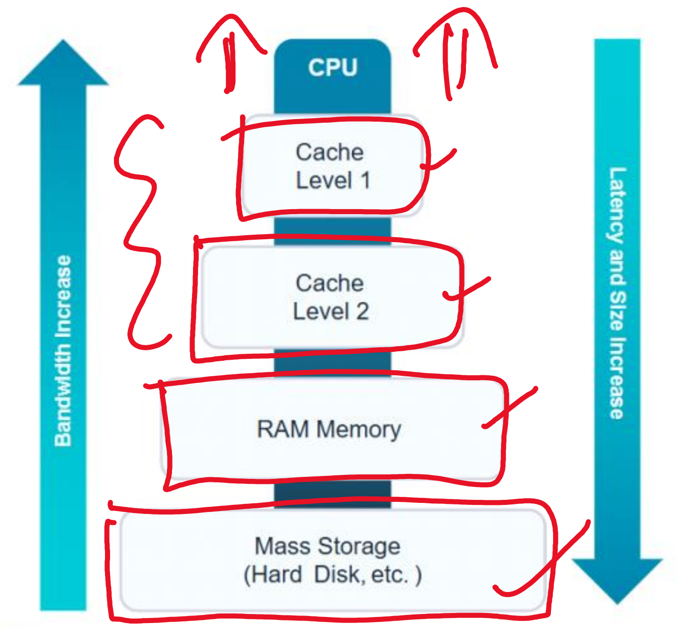
1. **Caching**
2. **Caching with Redis**
3. **Redis with FastAPI**
4. **Profiling FastAPI apps**
5. **Benchmarking APIs**
6. **Monitoring APIs**

# 1. Caching

Monday, June 16, 2025 10:20 PM

## What is Caching?

- Caching is the process of storing a copy of data or computational results in a temporary storage layer (**cache**)
- This is done so that future requests for that same data can be served much faster, without needing to recompute or fetch it from the original source



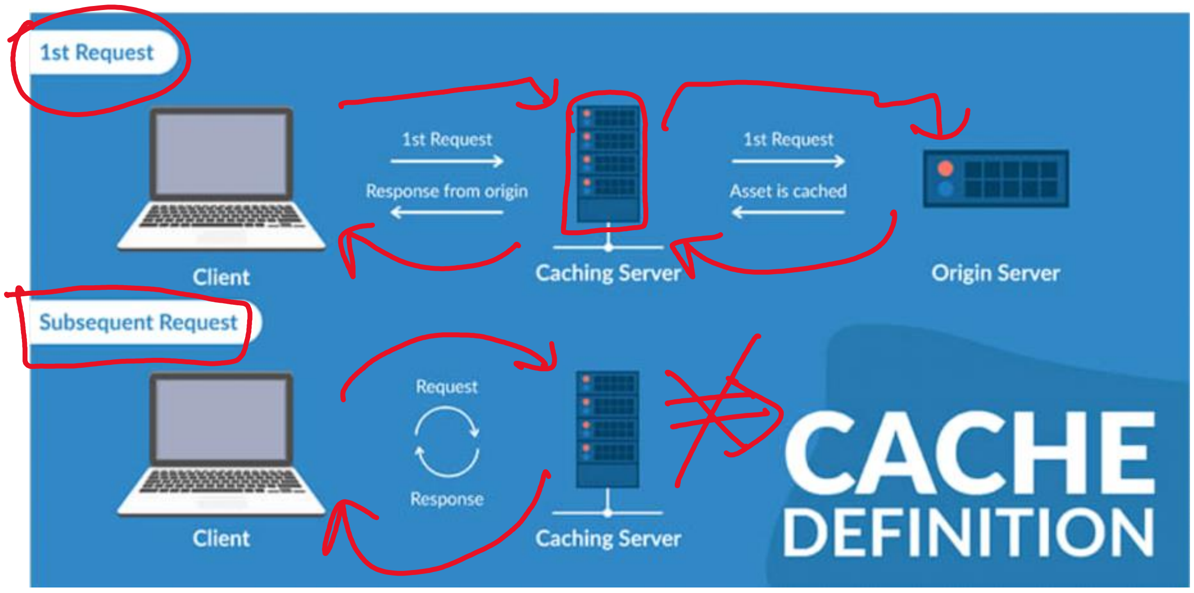
An Overview of Memory Caching

# 1.1 - Importance of Caching

Monday, June 16, 2025 10:31 PM

## Why is Caching Important?

- **Reduces Latency:** Cached responses are served from nearby or in-memory storage, which is significantly faster than from a database or an external API call
- **Improves Performance:** Applications become more responsive since frequently requested data is readily available
- **Reduces Load on Backend:** By reducing repeated data fetches or computations, the pressure on databases, ML models, or third-party APIs is minimized
- **Scalability:** Helps applications scale better under high load, as the same data doesn't need to be processed repeatedly

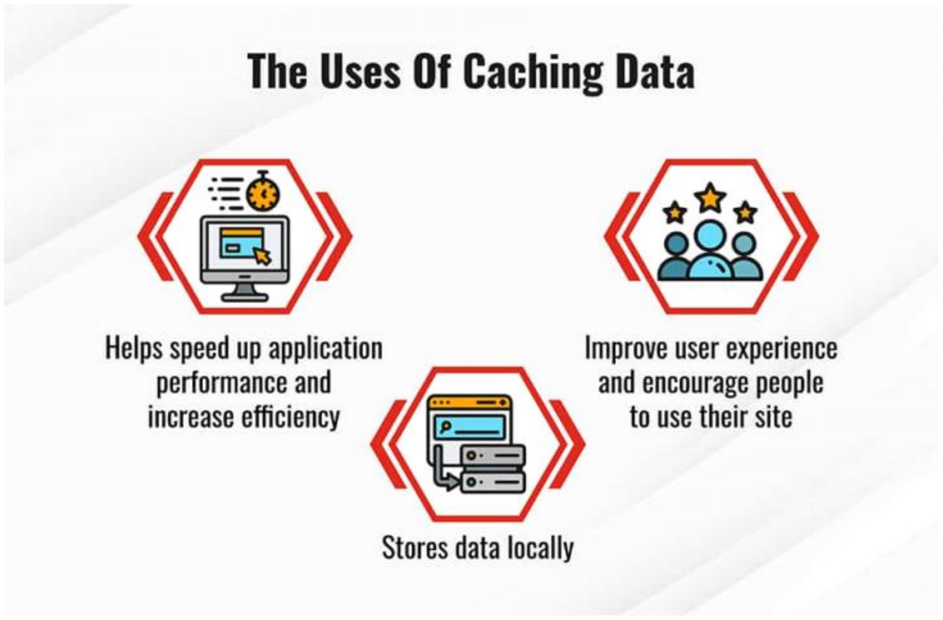


## 1.2 - Use Cases

Monday, June 16, 2025 10:32 PM

### Common Use Cases of Caching:

- **Web content:** Static assets (images, CSS, JS files) are cached by browsers or **CDNs** (Content Delivery Network)
- **Databases:** Frequently queried data are cached to avoid hitting the DB each time
- **API responses:** Slow or rate-limited third-party API calls are cached to avoid repeated calls
- **ML Predictions:** Expensive predictions (e.g., fraud scores, recommendations) are cached for identical inputs
- **Session Data:** In web apps, user sessions are often stored in a cache like **Redis** for fast retrieval

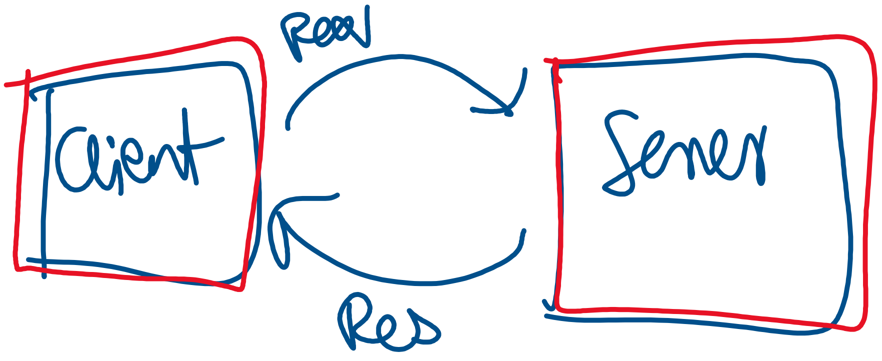

The Uses Of Caching Data
Helps speed up application performance and increase efficiency
Improve user experience and encourage people to use their site
Stores data locally

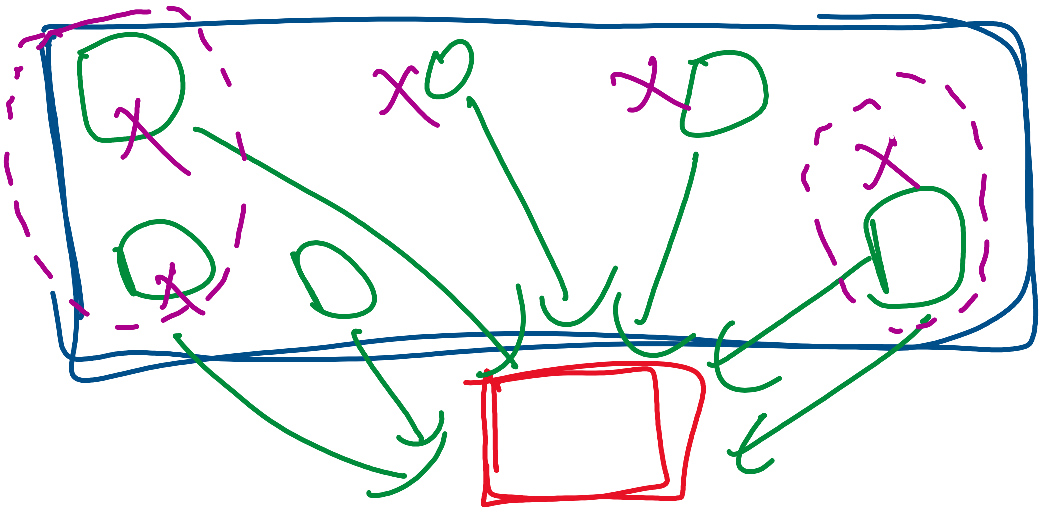
1.3 - Types of Caching

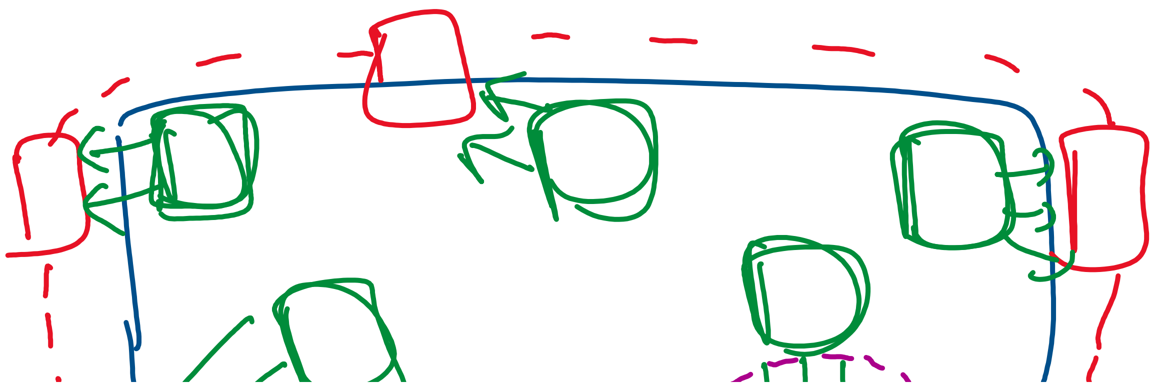
Monday, June 16, 2025 10:32 PM

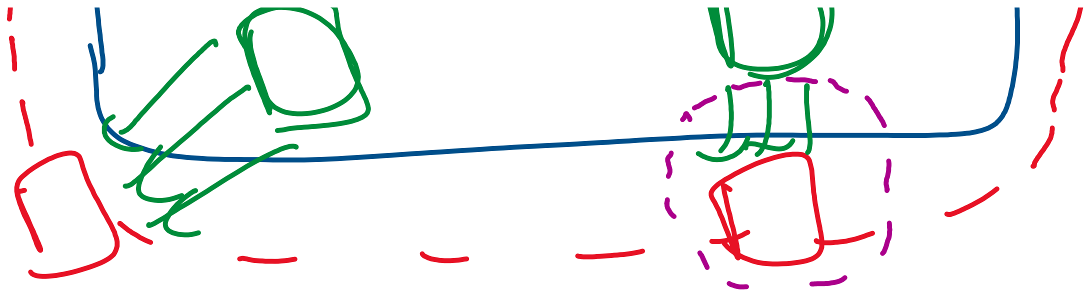
## Types of Caching:

- **Client-side caching:** Done in browsers or frontends using mechanisms like HTTP headers (**Cache-Control**)
- **Server-side caching:** Caching happens on the server using tools like **Redis**, **Memcached**, or **in-memory dictionaries**
- **CDN caching:** Content Delivery Networks cache static resources close to users for faster load times
  - A **Content Delivery Network (CDN)** is a network of geographically distributed servers that work together to deliver digital content (websites, videos, images, scripts, etc.) to users more efficiently and reliably
  - Instead of every user fetching content from a single central server (which can be slow or get overloaded), a CDN caches and serves content from a server that is geographically closer to the user
  - This reduces latency, speeds up loading times, and improves scalability and availability









## 1.4 - Key Considerations

Monday, June 16, 2025 10:32 PM

### Key Considerations with Caching:

- **Cache Invalidation:** When data changes, how do you ensure the cache is updated? This is a crucial challenge in caching systems
- **Eviction Policies:** Caches have limited memory, so older or less-used data must be removed using strategies like:
  - **LRU** (Least Recently Used)
  - **LFU** (Least Frequently Used)
  - **FIFO** (First In First Out)
- **Consistency:** Make sure cached data doesn't become stale or inconsistent with the source

## 1.5 - Tools

Monday, June 16, 2025 10:33 PM

### Commonly Used Tools:

- **Redis:** An in-memory key-value store, widely used for caching due to its speed and flexibility
- **Memcached:** Lightweight and fast in-memory caching system
- **CDNs:** Used to cache and serve static content globally (e.g., **Cloudflare**, **Akamai**)
- **Local Memory Cache:** Python dictionaries or FastAPI **lru_cache** decorators for lightweight scenarios

# 2. Caching with Redis

Monday, June 16, 2025 11:05 PM

## What is Redis?

- Redis (short for **REmote DIctionary Server**) is a fast, open-source, in-memory data structure store
- Redis stores everything in memory, which allows for blazing fast reads and writes — **often under 1 millisecond latency**
- **Redis is commonly used as:**
    - **Key-Value Database:** Stores data as key-value pairs, similar to a Python dictionary
    - **Cache:** Frequently used to cache database queries, API responses, and ML model predictions
    - **Message Broker:** Supports **publish/subscribe (pub/sub)**, streams, and queues for building messaging systems

## Why is Redis fast?

- **In-memory:** No disk I/O during reads/writes
- **Single-threaded:** Avoids context switching and locking overhead
- **Optimized C codebase**

## Redis Persistence:

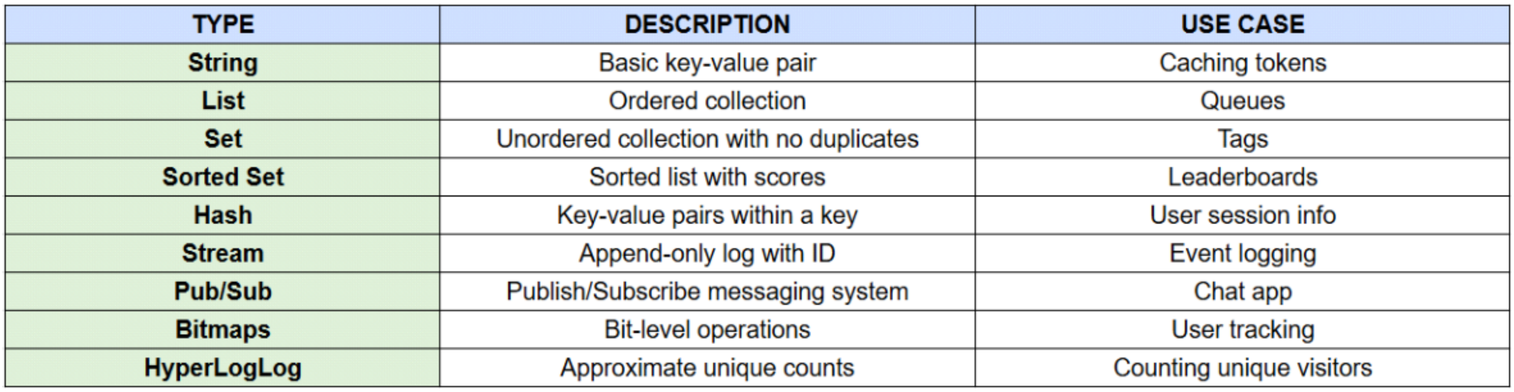
**Though Redis is an in-memory store, it supports data persistence through:**

- **RDB (Redis Database Backups)** – snapshots at intervals
- **AOF (Append Only File)** – logs every write operation

## 2.1 - Redis Data Structures

Monday, June 16, 2025 11:11 PM

**Redis supports a wide variety of data structures:**



| TYPE | DESCRIPTION | USE CASE |
|---|---|---|
| **String** | Basic key-value pair | Caching tokens |
| **List** | Ordered collection | Queues |
| **Set** | Unordered collection with no duplicates | Tags |
| **Sorted Set** | Sorted list with scores | Leaderboards |
| **Hash** | Key-value pairs within a key | User session info |
| **Stream** | Append-only log with ID | Event logging |
| **Pub/Sub** | Publish/Subscribe messaging system | Chat app |
| **Bitmaps** | Bit-level operations | User tracking |
| **HyperLogLog** | Approximate unique counts | Counting unique visitors |

## 2.2 - Use Cases

Monday, June 16, 2025 11:14 PM

### Common use-cases of Redis:

- **Caching** (e.g., ML predictions, DB queries)
- **Session Management** (e.g., storing user sessions in web applications)
- **Rate Limiting** (e.g., API throttling)
- **Real-time Analytics**
- **Leaderboards and Ranking Systems** (sorted sets)
- **Pub/Sub Messaging**

# 2.3 - Setup

Monday, June 16, 2025 11:25 PM

## 1. Install Redis Server (One-time setup):

- **Run command in bash: docker run -d -p 6379:6379 redis**

## 2. Install Redis Python Client:

- **There're generally two options:**
    - **redis-py: pip install redis**
    - **aioredis: pip install redis[async]** (recommended)

## 3. Test Redis Setup Using Python:

| CODE | PURPOSE |
|---|---|
| **redis.Redis(...)** | Connects Python code to Redis |
| **.ping()** | Tests the connection |
| **.set(key, value)** | Stores a value in Redis |
| **.get(key)** | Retrieves a value |
| **.decode()** | Converts bytes to a readable string |

# 3. Redis with FastAPI

Tuesday, June 17, 2025 10:25 PM

1. **Caching ML Predictions**

2. **Caching DB Query Results**

3. **Caching External API Call**

## 3.1 - Caching ML Predictions

Tuesday, June 17, 2025 10:27 PM

1. **Start Redis with Docker:**
   - **docker run -d -p 6379:6379 redis**
2. **Start the application**
3. **Test with identical inputs**

### Code:

- **to_list():**
  - Converts features into a list so they can be passed to the model
- **cache_key():**
  - Serializes the input into JSON (sorted for consistency)
  - Hashes it using **SHA-256** to generate a unique Redis key like **predict:<hash>**

### App Workflow:

- Input received at **/predict** as JSON and validated against **IrisFlower**
- A unique cache key is generated using a **SHA-256** hash of the input
- The app checks if the result is already in Redis:
  - If **yes** → return the cached result (parsed from JSON)
  - If **no** → make a prediction using the ML model
- The prediction result is cached in Redis with a **TTL (Time To Live)** of 1 hour (3600 seconds)
- The result is returned as JSON

# 3.2 - Caching DB Query Results

Tuesday, June 17, 2025 10:54 PM

## Code:

- **get_db_connection():**
  - Opens a connection to sqlite3 database
  - **row_factory = sqlite3.Row** makes rows behave like dictionaries

## App Workflow:

- Request comes in with **user_id**
- Redis is checked first
  - If **hit** → result returned from cache
  - If **miss** → query DB, store in cache, and return result
- Cache avoids repetitive DB load for same inputs

## 3.3 - Caching External API Call

Tuesday, June 17, 2025 11:01 PM

- This application will fetch data from a public API, cache the result in Redis, and return the cached result for repeated calls with the same input
- **API:** https://jsonplaceholder.typicode.com/posts
- Ensure to have **httpx** installed if not already

### App Workflow:

- Generates a **unique cache key** for the given **post_id**
- Checks Redis if data exists
- If **found** → Returns cached data
- If **not** → Calls external API, caches result, and returns it
- Subsequent calls with the same ID hit **Redis** only

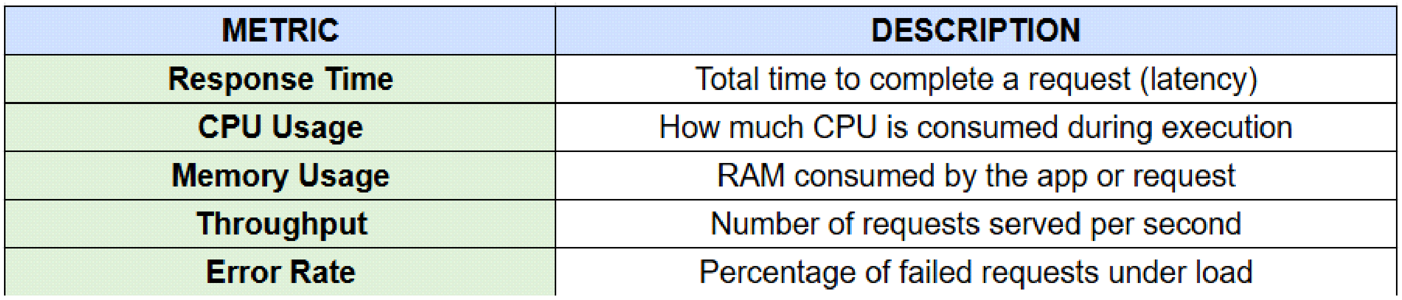
# 4. Profiling FastAPI Apps

Tuesday, June 17, 2025 11:17 PM

## Why Profiling is Important:

- Identify performance bottlenecks in business logic or API calls
- Minimize latency and maximize throughput
- Reduce CPU, memory usage, and I/O wait times
- Make informed architectural decisions (e.g., sync vs async)
- Prepare APIs for scale

## Key Metrics to Observe:



| METRIC | DESCRIPTION |
|---|---|
| **Response Time** | Total time to complete a request (latency) |
| **CPU Usage** | How much CPU is consumed during execution |
| **Memory Usage** | RAM consumed by the app or request |
| **Throughput** | Number of requests served per second |
| **Error Rate** | Percentage of failed requests under load |

## Profiling Tools:

- **time:** Quick & dirty timing of functions
- **cProfile:** Built-in profiler to capture function calls & time
- **line_profiler:** Line-by-line profiling

## 4.1 - Profiling using time

Tuesday, June 17, 2025 11:39 PM

## 4.2 - Profiling using cProfile

Tuesday, June 17, 2025 11:40 PM

- Install **snakeviz** for interactive visualization:
  - **snakeviz** is a browser-based visualization tool for Python's **cProfile** output
  - It provides interactive graphs to see how much time each function takes and how functions call each other
- Run **snakeviz**:
  - **snakeviz profiles\<profile_name>.prof**

## 4.3 - Profiling using line_profiler

Friday, June 20, 2025 10:46 PM

- Install **line_profiler**
- Write code for **app.py**
- Write code for **benchmark_test.py**
- Run the profiler --> kernprof -l -v profiling_test.py
    - **-l**: line-by-line profiling
    - **-v**: verbose output

**Note:**

- **line_profiler** doesn't run inside an ASGI server like **uvicorn** directly; needs isolated functions for benchmarking
- **line_profiler** installs a command-line tool called **kernprof**
- Run the profiler with **kernprof**
- **@profile** is not a built-in python decorator, it tells **line_profiler** - **Profile this function line by line**

# 5. Benchmarking APIs

Friday, June 20, 2025 11:08 PM

## What is Benchmarking of APIs?

- **Benchmarking** is the process of **measuring the performance** of an application or system under specific conditions
- In the case of APIs, it provides **quantitative insights** into how they behave under different levels of usage, helping make **data-driven** decisions for improvement

## 5.1 - Advantages

Friday, June 20, 2025 11:15 PM

### 1. Identify Performance Bottlenecks:

- Benchmarking allows you to pinpoint **slow endpoints**, **memory-heavy operations**, and **latency-inducing logic**
  - **Are certain API routes taking longer than expected?**
  - **Is the database query slowing down the response?**
  - **Are your background tasks affecting throughput?**
- By benchmarking, you gain visibility into which part of the stack needs optimization — whether it's the application layer, the database, or external integrations

### 2. Assess Scalability:

- Benchmarking simulates varying loads to evaluate how well your system scales
  - **How many concurrent users can your API support?**
  - **How does the performance degrade as the load increases?**
  - **What's the tipping point before errors or timeouts occur?**
- These insights are critical when preparing for **traffic spikes**, **product launches**, or **promotional campaigns**

### 3. Validate Service Level Agreements (SLAs):

- SLAs often define maximum response times, uptime, and error rates
- Benchmarking helps ensure:
  - Response times remain within the acceptable threshold (e.g., under 200ms)
  - Error rates stay below critical limits under heavy usage
  - Availability is maintained across different endpoints and services
- Failing to meet SLAs can result in **penalties, lost trust** and **poor user experience**

### 4. Compare Different Implementations:

- When considering design or technology changes, benchmarking helps you compare performance across versions
  - Comparing REST vs. GraphQL
  - Evaluating different FastAPI configurations (sync vs async)
  - Switching databases (PostgreSQL vs MongoDB)
  - Assessing the impact of caching layers like Redis
- This ensures that new implementations offer tangible benefits and don't degrade performance

## 5. Optimize Cost and Resources:

- Cloud providers charge based on usage
  - Tune your application to use resources efficiently
  - Avoid over-provisioning compute instances
  - Spot memory leaks or CPU-hungry operations
- It ensures you're not overpaying for underperforming infrastructure

## 6. Improve End-User Experience:

- End-users expect fast and consistent performance
  - Low latency under real-world usage conditions
  - Predictable performance regardless of geography or concurrency
  - Resilience under peak demand
- This directly influences user retention, satisfaction, and product adoption

## 7. Prepare for Production Readiness:

- Before going live, benchmarking gives confidence that:
  - Your system is production-ready
  - It can survive worst-case scenarios (e.g., DDoS mitigation, flash sales)
  - You can confidently scale your services

## 5.2 - Key Metrics

Friday, June 20, 2025 11:38 PM

1. **Latency: The time taken by the API to respond to a request, typically measured in milliseconds**
   - **Importance:**
     - Indicates user experience
     - **High tail latency** often causes **performance jitter** under load
   - **Use Cases:**
     - Detect **performance regression** after code changes
     - Optimize specific endpoints for faster responsiveness

2. **Throughput:** The number of requests the API can handle per second
   - **Importance:**
     - Tells you how much load the system can sustain
     - Key for **capacity planning** and **scaling decisions**
   - **Use Cases:**
     - Compare system performance under various configuration
     - Ensure infrastructure can handle expected traffic peaks

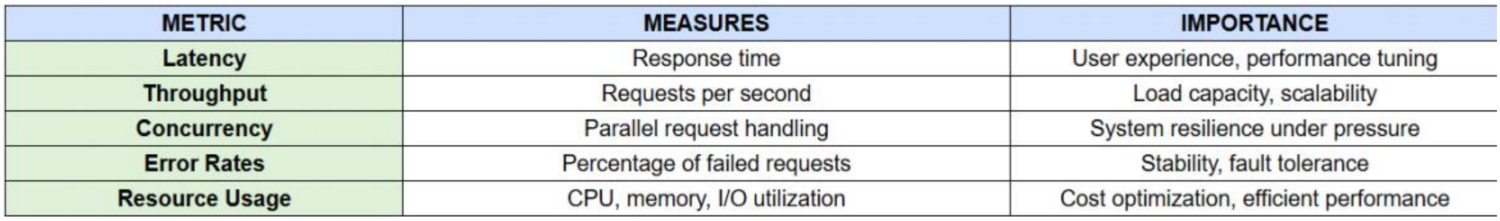
3. **Concurrency Handling:** The ability of API to process multiple simultaneous requests without degradation in performance or errors
   - **Importance:**
     - Real-world users hit APIs simultaneously
     - Poor concurrency support leads to timeouts and failed transactions
   - **Use Cases:**
     - Test **async vs sync** performance in FastAPI
     - Ensure proper **thread/worker configurations** in production environments

4. **Error Rates:** The percentage of requests that result in errors (HTTP 4xx or 5xx responses)
   - **Importance:**
     - Indicates system instability or poor error handling under load
     - Helps distinguish between **functional errors** and **load-induced failures**

- **Use Cases:**
    - Validate rate-limiting, authentication, or fail-safes
    - Ensure robust fallback mechanisms and exception handling in production

5. **Resource Usage:** The consumption of system resources (CPU, RAM, disk I/O) by API server during execution
    - **Importance:**
        - High CPU usage can indicate inefficient code or unnecessary computation
        - Memory bloat may lead to signal leaks or overuse of objects in memory
        - Important for **cloud cost optimization** and **container-based deployments**
    - **Use Cases:**
        - Optimize code paths or data processing
        - Determine resource requirements for autoscaling in **Kubernetes** or **cloud VMs**
        - Tune configurations

## Summary:



| METRIC | MEASURES | IMPORTANCE |
|---|---|---|
| **Latency** | Response time | User experience, performance tuning |
| **Throughput** | Requests per second | Load capacity, scalability |
| **Concurrency** | Parallel request handling | System resilience under pressure |
| **Error Rates** | Percentage of failed requests | Stability, fault tolerance |
| **Resource Usage** | CPU, memory, I/O utilization | Cost optimization, efficient performance |

## 5.3 - Tools

Friday, June 20, 2025 11:57 PM

1. **wrk:** **CLI**
    - **Pros:**
        - High performance
        - Lightweight and fast
        - Ideal for simple stress testing
    - **Cons:**
        - No detailed report
        - No scripting logic for real user scenarios

2. **ApacheBench (ab):**
    - **Pros:**
        - Very simple to use
        - Comes pre-installed on many systems
    - **Cons:**
        - Single-threaded (not suitable for high-scale)
        - Basic reporting
        - No user behavior scripting

3. **Locust:** **Python-based, Web UI**
    - **Pros:**
        - Web-based control panel
        - Easy scenario scripting (supports login, flows, etc.)
        - Supports distributed testing
    - **Cons:**
        - Heavier than CLI tools
        - Slightly more setup

4. **k6:** **Modern CLI, CI/CD Friendly**
    - **Pros:**
        - Great for automation
        - JavaScript-based scripting
        - Detailed metrics, CI/CD integration
        - Cloud options available (k6 Cloud)
    - **Cons:**
        - Slightly complex setup for beginners

| TOOL | DESCRIPTION |
|---|---|
| **wrk** | High-performance, multi-threaded HTTP benchmarking tool with customizable Lua scripts. Ideal for simple performance tests |
| **ab** | ApacheBench, a single-threaded CLI tool that's easy to use for quick checks |
| **Locust** | Python-based, event-driven tool with a web interface and support for user behavior scripting. Great for real-world simulation |
| **k6** | Modern, developer-centric load testing tool with JS scripting. Easy CI/CD integration |

## 5.4 - Locust Demo

Saturday, June 21, 2025 12:31 AM

- Install **locust** using **pip**

- Implement the script:
  - **main.py**
  - **locustfile.py**
    - **HttpUser:** Represents a simulated user making HTTP requests
    - **@task:** Used to mark methods as tasks that Locust will execute
    - **between:** Used to set wait time between tasks (to simulate real user behavior)

- Run **locust**:
  - Execute -- **locust / locust -f locustfile.py**
  - Open -- http://localhost:8089/

- Provide details:
  - **Number of users to simulate** - 100
  - **Spawn rate** - 10
  - **Host** - http://127.0.0.1:8000

## 5.5 - Best Practices

Saturday, June 21, 2025 12:00 AM

- Warm up the server before starting tests
- Test different endpoints separately
- Try **increasing concurrency** gradually (ramp-up)
- Use production-like data if possible
- Benchmark **locally** and in **staging/cloud**
- Use monitoring tools (**Grafana + Prometheus**) to correlate performance with system usage

# 6. Monitoring APIs

Saturday, June 21, 2025 12:56 AM

## What is API Monitoring?

- Monitoring APIs refers to the continuous observation and tracking of an API's performance, availability, and functionality to ensure it behaves as expected and delivers a seamless experience to users or connected systems
- It is a critical aspect of maintaining reliable and high-performing software applications, especially in production environments

## Why API Monitoring?

1. **Availability Monitoring:**
   - Ensures the API is reachable and responsive
   - Uptime checks at regular intervals
   - Alerts when the API is down or returns errors
2. **Performance Monitoring:**
   - Measures response time, latency, throughput
   - Identifies slow endpoints or inefficient logic
   - Helps in tuning APIs for better user experience
3. **Error Tracking:**
   - Logs and analyzes status codes
   - Tracks frequency and type of errors
   - Useful for debugging and **root cause analysis (RCA)**
4. **Usage Analytics:**
   - Monitors request volume, endpoints usage, and consumer behavior
   - Helps in capacity planning and scaling
5. **Resource Monitoring:**
   - Tracks CPU, memory, I/O utilization at the infrastructure level
   - Useful in understanding backend stress caused by API calls
6. **Alerting & Incident Management:**
   - Sends real-time notifications on abnormal behavior
   - Helps reduce **Mean Time To Detect (MTTD)** and **Mean Time To Resolve (MTTR)**

## Key Metrics to Track: **Similar to Benchmarking**

# 6.1 - Prometheus

Saturday, June 21, 2025 10:03 AM

## What is Prometheus?

- Prometheus is an open-source systems monitoring and alerting toolkit, originally developed by **SoundCloud**
- It's designed for reliability and scalability, and is especially strong in environments like cloud-native applications, microservices, and containerized deployments (like **Kubernetes**)

## Characteristics:

1. **Pull-based Model:** Prometheus **pulls (scrapes)** metrics from instrumented targets (applications/services) at specified intervals
2. **Time Series Storage:** Metrics are stored as **time series** and they're indexed by :-
   - A metric name
   - One or more labels
3. **Flexible Query Language: PromQL** lets you perform complex queries to **aggregate, filter, and compute metrics**
4. **Built-in Alert Manager:** Allows to configure alert rules and send alerts to email, Slack, PagerDuty, etc.

## How Prometheus scrapes Metrics?

1. **Configuration:** Prometheus reads a **YAML** configuration file (**prometheus.yml**) to determine what to scrape, when, and how often
2. **Targets:** Each target is an HTTP endpoint that exposes metrics, commonly at the **/metrics** route
3. **Scrape Format:** The data should be exposed in **Prometheus text** format or **OpenMetrics** format

## Use Cases:

- Monitoring web services
- Observing containerized applications
- Tracking application health, traffic, and performance
- Generating real-time alerts and dashboards

## 6.2 - Prometheus with FastAPI

Saturday, June 21, 2025 10:21 AM

### prometheus-fastapi-instrumentator:

- **prometheus-fastapi-instrumentator** is a plug-and-play library that enables **automatic instrumentation** of FastAPI applications
- It helps collect **runtime metrics** (request count, latency and status codes) and expose them in a **Prometheus-compatible** format, typically at **/metrics**

- **Installation:**
  - **pip install prometheus-fastapi-instrumentator** (Python 3.9+)
  - **pip install prometheus-fastapi-instrumentator==5.9.1** (up to Python 3.8)

### FastAPI Integration: http://127.0.0.1:8000/metrics

- **instrument(app):**
  - Hooks into FastAPI's **routing layer**
  - Captures HTTP-level metrics:
    - Request count
    - Request duration
    - Status code distribution
    - Method and endpoint path
  - Wraps every route handler with logic to collect and export the metrics

- **expose(app):**
  - Adds a **/metrics** route (by default)
  - This endpoint serves metrics in a **Prometheus-compatible** format

### Default Metrics Generated:

- **method:** HTTP method (GET, POST, etc.)
- **path:** API route
- **status:** HTTP status code
- **_bucket{le="0.1"}:** Histogram bucket showing how many requests completed in ≤0.1 seconds
- **_count** and **_sum:** Total number and cumulative duration of requests

## 6.3 - Prometheus, FastAPI & Docker

Saturday, June 21, 2025 11:02 AM

### File Structure:

**project-folder/**

- **app/**
  - **main.py**
- **prometheus/**
  - **prometheus.yml**
- **docker-compose.yml**
- **Dockerfile**

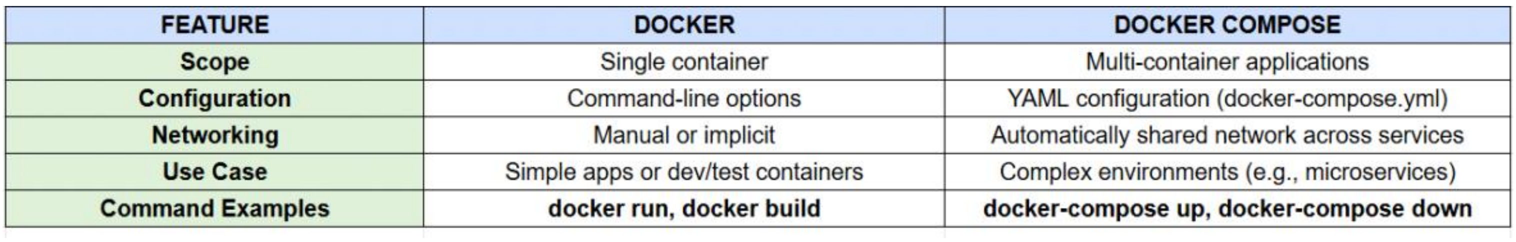
### Run the app: docker-compose up --build

### Access the Interfaces:

- **FastAPI endpoint:** http://localhost:8000/
- **FastAPI metrics:** http://localhost:8000/metrics
- **Prometheus UI:** http://localhost:9090/
  - **Run query: http_requests_total**

### Docker-compose:

- Higher-level tool used to define and run **multi-container** Docker applications
- Uses a **YAML file (docker-compose.yml)** to configure application services, networks, volumes, etc.
- Allows to start all defined services with one command: **docker-compose up**
- **Key Benefits:**
  - Simplifies running multi-container environments
  - Handles networking and volume mounts between containers automatically



| FEATURE | DOCKER | DOCKER COMPOSE |
|---|---|---|
| **Scope** | Single container | Multi-container applications |
| **Configuration** | Command-line options | YAML configuration (docker-compose.yml) |
| **Networking** | Manual or implicit | Automatically shared network across services |
| **Use Case** | Simple apps or dev/test containers | Complex environments (e.g., microservices) |
| **Command Examples** | **docker run, docker build** | **docker-compose up, docker-compose down** |

# 6.4 - Grafana

Saturday, June 21, 2025 11:46 AM

## What is Grafana?

- Grafana is an open-source analytics and interactive data visualization platform
- It allows users to query, visualize, alert on, and understand metrics no matter where they are stored
- It is widely used for monitoring infrastructure, applications, and data pipelines in real time
- It supports pluggable data sources, meaning it can connect to a wide range of backends like Prometheus, InfluxDB, Elasticsearch, MySQL, PostgreSQL, and many more

## How Grafana Works:

1. **Data Sources:**
    - Grafana connects to databases and time-series backends via **data source plugins**
    - Common sources include: **Prometheus, InfluxDB, Loki, Elasticsearch, PostgreSQL, MySQL**
2. **Queries:**
    - You can write **custom queries** or use **built-in query builders** to extract data from the source
3. **Dashboards:**
    - Dashboards are made of **panels** (charts, tables, gauges, heatmaps, etc.) where the data is visualized
    - You can save and share dashboards with teams
4. **Alerting:**
    - Grafana provides **rule-based alerting** on your metrics
    - You can integrate with **Slack, PagerDuty, Microsoft Teams, Email**, etc., for notifications

## Use Cases of Grafana:

1. **Infrastructure Monitoring:**
    - Monitor server CPU, RAM, disk usage, network I/O
    - Integrate with **Prometheus, Telegraf, InfluxDB**, or **Node Exporter**
2. **Application Performance Monitoring (APM):**
    - Track API response times, throughput, error rates
    - Useful with **Prometheus + FastAPI/Django**, or APM tools like **Jaeger**
3. **Database Monitoring:**
    - Connect to **MySQL, PostgreSQL, MongoDB**, etc., and visualize query performance, connections, latency
4. **DevOps and CI/CD Pipeline Monitoring:**
    - Visualize deployment frequency, failure rates, build times using data from **Jenkins, GitHub Actions, CircleCI**, etc.

5. **IoT and Sensor Data Dashboards:**
    - Ingest data from IoT sensors using **MQTT** or **InfluxDB**, visualize in Grafana with time-series plots

6. **Business KPIs Monitoring:**
    - Track sales performance, user engagement, or churn rates using **Google Sheets, PostgreSQL,** or **Excel files** as data sources

7. **Security Monitoring:**
    - Visualize login attempts, suspicious activity using **Elasticsearch** or **Loki** for log aggregation

## Advantages of Grafana:

- **Open-source and free:** Core Grafana is free to use and has a large community of contributors
- **Custom Dashboards:** Easy to build and customize dashboards tailored to specific needs
- **Plugin Ecosystem:** Rich library of community-developed plugins for new data sources or visuals
- **Multiple Data Sources:** Combine metrics from multiple tools in a **single unified dashboard**
- **Time-Series Friendly:** Especially powerful for time-series and real-time streaming data
- **Integrations:** Works with **Prometheus, Loki, Elasticsearch, InfluxDB, AWS CloudWatch**, etc.
- **Alerting System:** Allows users to receive notifications when metrics cross thresholds
- **Collaboration:** Share dashboards with teams and define permissions for access control
- **Templating:** Use variables to create dynamic and reusable dashboards

# 6.5 - Grafana, Prometheus, FastAPI & Docker

Saturday, June 21, 2025 12:10 PM

## File Structure:

**project-folder/**

- **app/**
    - **main.py**
    - **Dockerfile**
- **prometheus/**
    - **prometheus.yml**
- **docker-compose.yml**
- **requirements.txt**

## Run the app: **docker-compose up --build**

## Access the Interfaces:

- **FastAPI endpoint:** http://localhost:8000/
- **FastAPI metrics:** http://localhost:8000/metrics
- **Prometheus UI:** http://localhost:9090/
    - **Run query: http_requests_total**
- **Grafana:** http://localhost:3000
    - **Username: admin**
    - **Password: admin**

## Configure Grafana:

- Navigate to **Grafana -> Add data source -> Prometheus**
- **Set URL:** http://prometheus:9090
- Save & Test
- Create dashboards using metrics such as:
    - **http_server_requests_total**
    - **http_request_duration_seconds_bucket**
    - **http_request_duration_seconds_sum**