Session III

Performance, Cloud and Monitoring the Web API

Performance

Important Terminology:

- Scalability: handle increased traffic by adding more resources
  - Horizontal scaling: more instances; vertical scaling: better hardware
- Statelessness: requests are independent and include everything for processing
- Persistence: data that needs to be stored beyond runtime, e.g., using databases
- I/O efficiency: avoid blocking while waiting for external resources using async
- Caching: temporarily store often accessed data (e.g., in memory or Redis)
- Load balancing: distribute requests across multiple instances

ASGI and WSGI

WSGI (Web Server Gateway Interface):

- Defines standard communication between web servers and python apps
- Decouples web server from app framework
- Processes HTTP requests and responses and promotes scaling
- Used by, e.g., Flask

ASGI (Asynchronous Server Gateway Interface):

- Evolution of WSGI for asynchronous operations
- Used by, e.g., FastAPI

ASGI and WSGI

Production deployment of Python Web APIs typically requires running with a WSGI/ASGI server.

- Flask has a built-in development server, but for production it requires running with e.g., Gunicorn.
- FastAPI CLI (`fastapi run`) internally uses a production-ready ASGI server (uvicorn), but direct usage allows further configurations.

Deployment Options: Cloud

Serverless functions: fully managed execution of isolated functions

- Managed: scaling, networking, infrastructure, request handling
- Control: function logic

Serverless containers: run containerized APIs with automatic scaling

- Managed: scaling, networking, load balancing
- Control: API code, containerization

Managed Kubernetes: cloud-hosted Kubernetes with automated control plane

- Managed: cluster control plane, networking
- Control: deployments, scaling rules, API routing

Cloud VMs: full OS control with cloud flexibility

- Managed: hardware, optional networking
- Control: OS, scaling, networking, security, API deployment, updating

Deployment Options: On-Premise

Self-hosted servers: fully self-managed infrastructure

- Managed: nothing (fully self-hosted)
- Control: hardware, networking, security, scaling, maintenance

Private cloud: on-premise infrastructure with cloud-like management

- Managed: internal networking, storage, virtualization
- Control: compute, orchestration (e.g., Kubernetes), integrations

Additional requirements:

- Reverse proxy & networking: required for secure API exposure
- Ongoing maintenance: hardware replacements, security patches, system updates
- Scaling complexity: physical infrastructure must be expanded manually
- Monitoring & logging: no built-in cloud monitoring, needs self-hosted tools

Security

API Security Concerns:

- Authentication and authorization: limit access to protected endpoints
- Rate limiting: prevent abuse by limiting request frequency
- Data exposure: avoid leaking sensitive information in error messages or logs
- CORS (Cross-Origin Resource Sharing): restrict domains (from web browsers)
- HTTPS Enforcement: encrypt communication to protect against interception
- Logging and monitoring: detect unusual activity and security breaches

Levels of Monitoring

1. **System:** health of system, e.g., CPU, GPU, memory usage, system logs
2. **Network:** incoming/outgoing traffic, e.g., unusual patterns, traffic spikes
3. **Data storage:** database performance, data integrity
4. **Code:** code execution patterns, runtime errors, dependencies, misconfigurations
5. **Application:** response latency, HTTP status codes, authentication success/failures
6. **Model/Data:** running metrics, data/model drift, adversarial inputs
7. **User activity:** user behavior patterns

Metrics, Logs and Traces

Metrics:

- Time-series data representing system/application performance
- Useful for anomaly detection and trend analysis

Logs:

- Structured or unstructured records of discrete events (e.g., errors)
- Essential for analysis and debugging

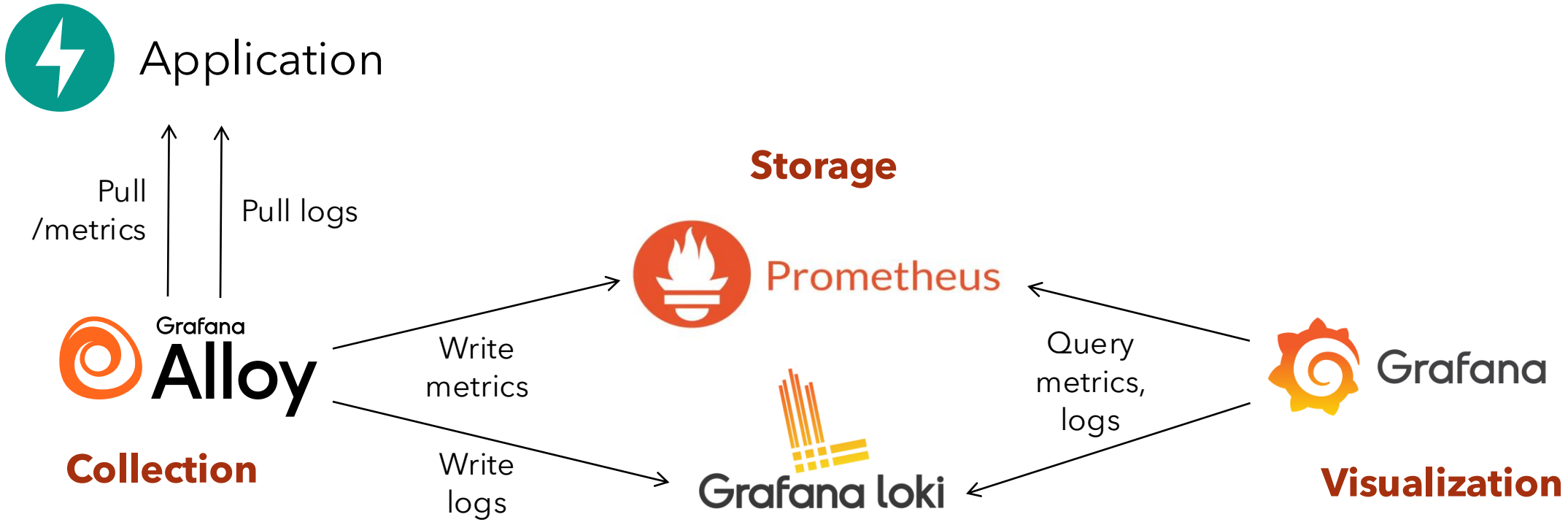
Traces:

- End-to-end tracking of requests through systems
- Insights into latency, dependencies, and failure points

Examples of Popular Monitoring Tools

- Grafana + Prometheus + Loki
  - Grafana: metrics/logs exploration
  - Prometheus: time-series database and collection of metrics
  - Loki: log aggregation
- ELK Stack
  - Elasticsearch: search and analytics engine
  - Logstash: log ingestion and transformation
  - Kibana: data visualization
- OpenTelemetry: Open standard for metrics, logs, traces
- Sentry: Code error tracking and performance

Grafana Monitoring Stack



Practice: Monitoring Stack

Add basic metrics endpoint:

```python
from prometheus_fastapi_instrumentator import Instrumentator

app = FastAPI()
Instrumentator().instrument(app).expose(app)
```

Task:

- Add the monitoring endpoint to your API (see above)
- Rebuild the image and start in the monitoring stack folder: `docker compose up -d`
- Send requests as before (`http POST` or /docs) to http://127.0.0.1:8000/chat
- Explore the resulting metrics and logs at http://127.0.0.1:3000
- Afterwards: shutdown with `docker compose down`
- Optional: add custom metrics (see in example folder)