Exercises 02 & 03

Guidelines for implementing components of Lecture 03-06 in Group Project

Module 5: Non-Functional Requirements: Performance Optimization

Objective: Implement monitoring and profiling.

Steps:

1. **Performance Monitoring**:
    - Set up **Prometheus** for monitoring.
        i. Set up metrics collection (e.g., response times, message throughput, database query latency, CPU, memory, request latencies) in each microservice (gRPC, REST FastAPI/Django, Kafka).
        ii. Expose metrics endpoints (e.g., /metrics) in FastAPI and gRPC services.

Note: *"Before you can monitor your services, you need to add instrumentation to their code via one of the Prometheus client libraries. These implement the Prometheus metric types."* https://prometheus.io/docs/instrumenting/clientlibs/

    - Use **Grafana** to visualize the metrics. https://grafana.com/docs/grafana/latest/getting-started/get-started-grafana-prometheus/
2. **Performance Profiling:**
    - Use Py-Spy or cProfile to analyze Python microservices.
    - Identify bottlenecks and optimize code.
3. **Load Testing**:
    - Use **Locust** (ideal for Python) to simulate load and distributed testing.
        i. Test the performance of synchronous (gRPC/REST) and asynchronous (Kafka) components under high loads

Note: Another popular load-testing tool in the Finnish ICT industry is k6 if your choice of language is JavaScript. You can use either Locust or K6.

Testing:

- Verify the system's throughput under varying loads.
- Check latency and error rates for each microservice.
- Confirm that metrics are correctly captured and visualized.

Module 6: Non-Functional Requirements: Elasticity and Scalability

Objective: Enable the system to scale dynamically based on load.

Steps:

1. **Deploy on Kubernetes**
    - Containerize each microservice using Docker.
        i. Step-by-step instructions from Linode on building and deploying microservices using Docker and Docker Compose, covering essential aspects such as creating Dockerfiles, setting up Docker Compose configurations, and testing the microservices: https://www.linode.com/docs/guides/deploying-microservices-with-docker
    - Deploy on **Kubernetes** for orchestration.
        i. Stepwise guide (you need to have containerized microservices first): https://www.digitalocean.com/community/tech-talks/how-to-deploy-your-application-or-microservice-as-a-kubernetes-deployment
2. **Horizontal Scaling:**
    - Set up **Horizontal Pod Autoscaler (HPA)** for scaling based on CPU, memory, or custom metrics (e.g., Kafka lag).
        i. Official walkthrough on configuring HPA based on CPU utilization: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale-walkthrough
        ii. how to implement HPA using custom metrics, specifically focusing on Kafka metrics like consumer lag https://sunnykrgupta.github.io/kubernetes-hpa-autoscaling-with-kafka-metrics.html
        iii. how to scale Kafka consumer groups using custom metrics in Kubernetes https://developers.redhat.com/articles/2023/12/29/autoscaling-kafka-workloads-custom-metrics-autoscaler
3. **Load Balancing:**
    - Integrate **NGINX** as a reverse proxy and load balancer.
    - Configure NGINX to distribute requests to multiple replicas.
4. **Elastic Database Scaling** (suggestion from CSC colleague.):
    - Use CockroachDB or Cassandra, which support distributed scaling
    - Implement database connection pooling for performance.

If you have already chosen another database (like not Cassandra or CockroachDB), please feel free to skip elastic database scaling.

5. **Elastic Messaging:**
    - Configure **Kafka partitions** and brokers to scale the messaging system.
    - Use **KRaft mode** for Kafka cluster management

Testing:

- Simulate varying traffic levels using Locust.
- Validate that Kubernetes scales pods dynamically.
- Test database performance with concurrent read/write operations.

Module 7: Replication and Consistency

Objective: Ensure data availability, fault tolerance, and consistency across distributed components.

Steps:

1. **Database Replication**:
    - Configure MongoDB/Apache Cassandra with replication enabled (e.g., Replica Set for MongoDB or multi-datacenter replication for Cassandra, CockroachDB multi-region deployment).
2. **Consistency Management**:
    - Design your services to handle **CAP theorem trade-offs** based on use case (e.g., eventual consistency for Kafka consumers, strong consistency for critical operations).
        i. Implement eventual consistency in Cassandra or MongoDB.
        ii. Demonstrate strong consistency using CockroachDB or MariaDB.
3. **Distributed Coordination with etcd:**
    - Integrate etcd for leader election and service discovery.
    - Use etcd's watch mechanism for consistent state updates across microservices.
4. **Replication in Messaging**
    - Enable **Kafka topic replication** across brokers for fault-tolerant message delivery.
5. **Testing Fault Tolerance**:
    - Simulate node failures using Chaos Engineering tool: **Chaos Mesh**.

        i. **Chaos Mesh** is easy to integrate into Kubernetes infrastructure without any changes to deployment logic.
    - Verify data integrity and recovery mechanisms.

Another popular tool for chaos engineering in the Finnish ICT sector, Gremlin, is not open source!

6. **Conflict Resolution:**
    - Implement a last-write-wins (LWW) policy or vector clocks to resolve replicated data conflicts.

Testing:

- Simulate node failures to verify data replication and recovery.
- Test read/write operations to ensure consistency.
- est etcd coordination by introducing artificial network partitions and monitoring service recovery.
- Use **Jepsen** to test distributed consistency under network partitions.

Module 8: Resource Management and Scheduling

Objective: Optimize resource allocation and scheduling for efficient system utilization.

Steps:

1. **Dynamic Resource Allocation:**
    - Use **Kubernetes Resource Quotas** and **LimitRanges** to manage resource allocation for each microservice.
        i. Configure pod-level resource requests and limits for CPU and memory.
    - Use **KEDA** (Kubernetes-based Event Driven Autoscaling) to scale Kafka consumers or other services based on event volume.
2. **Scheduling Optimization:**
    - Implement Kubernetes custom schedulers for critical tasks: Configure **Kubernetes affinity/anti-affinity rules** and **priority-based scheduling.**
    - Use Kubernetes **taints and tolerations** to segregate workloads (e.g., database on high-memory nodes).
3. **Cluster Autoscaling**:
    - Enable Kubernetes **Cluster Autoscaler** to dynamically add/remove nodes based on demand.

Testing:

- Deploy services with resource limits and test behavior under constrained resources.
- Simulate node unavailability and verify service redistribution across nodes.
- Scale workloads and validate autoscaling at the container and cluster levels.

Integration Testing

Objective: Test the fully integrated system for functionality and performance.

Steps:

1. **Integrated Deployment:**
    - Deploy all components (gRPC, FastAPI, Kafka, database) on Kubernetes.
    - Use Helm charts for consistent configurations.
    - Integrate monitoring tools (Prometheus, Grafana) across all components.
        i. Measure latency, throughput, and resource utilization under integrated workloads.
        ii. Use Grafana dashboards to analyze system behavior and identify bottlenecks.
    - Ensure proper resource allocation and scaling policies.
2. **Workflow Validation:**
    - Simulate end-to-end workflows combining synchronous and asynchronous communication.
    - Include database operations to test replication and consistency.
3. **Fault Injection:**
    - Use **Chaos Mesh** to inject failures (e.g., pod crashes, network delays).
    - Observe system behavior and verify recovery.
4. **Full Load Test:**
    - Use Locust to generate realistic traffic across all services.
        i. Conduct load tests using Locust to simulate hybrid workloads (synchronous + asynchronous communication).
    - Monitor system performance and resource usage.

Success Criteria:

- Services remain functional under stress and failures.
- Data consistency is maintained across replicas.

- The system scales dynamically with workload.