Testing is a critical phase in software development that ensures the reliability, functionality, and performance of an application. It is applied throughout the development lifecycle, from early unit tests to final system validation. Testing is useful because it identifies defects early, reduces maintenance costs, and improves user satisfaction by delivering a stable product. By systematically verifying each component and interaction, testing helps maintain high-quality standards and minimizes risks in production environments.

The application in my Bachelor thesis is a microservices-based e-commerce system with key features such as order processing, payment handling, and inventory management. It leverages Kubernetes for orchestration, Prometheus for monitoring, and KEDA for auto-scaling. The system is designed for high scalability (handling variable workloads) and fault tolerance (resilience to service failures). Additional features include distributed tracing (Jaeger) for performance analysis and event-driven communication (Kafka) for asynchronous workflows.

Testing Techniques for Development
To ensure robustness, the following testing techniques should be used:

Unit Testing: Validates individual components (e.g., API endpoints, business logic) in isolation. For the Order Service, unit tests should verify core logic such as Order Creation, which ensures the API correctly validates and persists orders.

Integration Testing: Checks interactions between services (e.g., Order Service ↔ Payment Service). Integration tests should focus on critical service interactions:

1. Order → Payment Service: Confirm payment processing succeeds/fails as expected. Strategy: Big Bang Testing

2. Order → Inventory Service: Validate stock reservation and rollback on failure. Strategy Incremental Integration (testing initially just the order and inventory service, and afterwards adding the payment service and Kafka for event driven communication)

3. API Gateway Routing: Ensure requests are correctly forwarded to microservices. Strategy: Use contract testing (Pact) to verify APIs meet expectations without full deployment.

System Testing: Verifies end-to-end workflows (e.g., placing an order → payment → inventory update).

1. Happy Path: Successful order placement → payment → inventory deduction → confirmation email.

2. Failure Recovery: Payment service downtime triggers order cancellation and inventory rollback.

3. Load Testing: Spike traffic (via Locust) to validate auto-scaling (KEDA) and monitor latency.