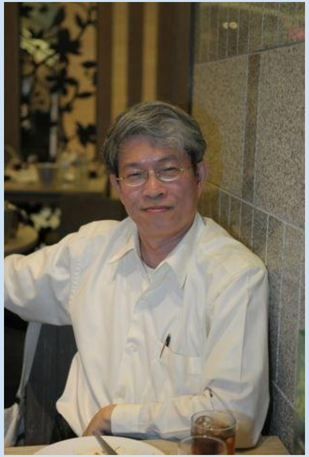
Distributed Application
Devlopment & Tools



Dr. Yunyong Teng-amnuay

(Retired)

Dept. of Computer Engineering

Faculty of Engineering

Chulalongkorn University

January 13, 2025

Examples of Distributed Applications

https://www.tutorchase.com/answers/ib/computer-science/what-are-examples-of-distributed-systems-in-real-life-applications

Online Banking

- Transaction processing, customer services, security
- Large volume of transactions, reliability, fault tolerant

Social Networking (Facebook, X)

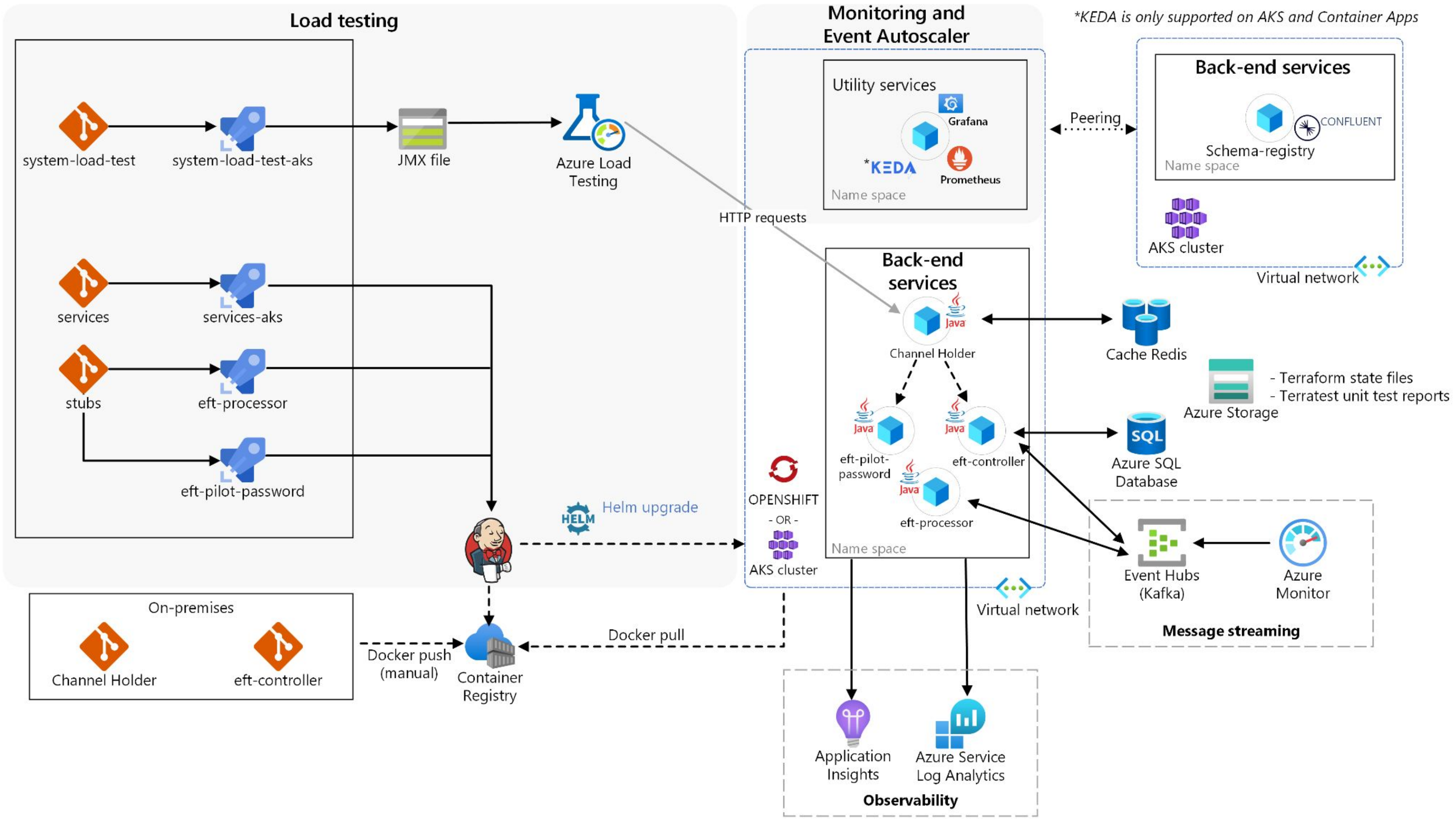
- Vast amount of data, millions of users worldwide
- Multiple servers in various locations
- High traffic, high availability

Cloud (Amazon Web Service - AWS, Google Cloud, Microsoft Azure)

- Remote & multiple servers
- Different geographical locations
- Scalability, reliability, cost-effectiveness

Telecommunications

- Mobile phone services through interconnected servers & switches
- Large number of calls & texts, reliability



Microsoft Azure - An Example of Banking in the Cloud

https://learn.microsoft.com/en-us/azure/architecture/example-scenario/banking/banking-system-cloud-transformation

How to develop distributed application?

(Answer by Copilot.)

1. Define the Requirements
2. Choose the Architecture
3. Select the Technologies
4. Set Up the Infrastructure
5. Design the Application
6. Implement the Code
7. Testing
8. Deployment
9. Monitoring and Maintenance
10. Scaling

1. Define the Requirements:

- Determine the purpose of the application and its key features.
- Identify the components that will be distributed across different nodes.

2. Choose the Architecture:

- **Monolithic**: All components in one package (not truly distributed).
- **Microservices**: Each component runs as a separate service.
- **Serverless**: Functions run in response to events, managed by cloud providers.

3. Select the Technologies:

- **Programming Languages**: Java, Python, Go, etc.
- **Frameworks**: Spring Boot (Java), Django (Python), Express (Node.js), etc.
- **Communication**: REST, gRPC, message queues (RabbitMQ, Kafka).

4. Set Up the Infrastructure:

- **Cloud Providers**: AWS, Azure, Google Cloud.
- **Containerization**: Docker.
- **Orchestration**: Kubernetes.

5. Design the Application:

- **Database**: Choose between SQL (e.g., PostgreSQL, MySQL) or NoSQL (e.g., MongoDB, Cassandra).
- **APIs**: Define clear interfaces for communication between components.
- **Security**: Implement authentication, authorization, and data encryption.

6. Implement the Code:

- Write clean, modular, and well-documented code.
- Develop and test each component independently.

7. Testing:

- **Unit Tests**: Test individual components.
- **Integration Tests**: Ensure components work together.
- **Load Testing**: Test the system under high load.

8. Deployment:

- Set up CI/CD pipelines for automated testing and deployment.
- Deploy the application incrementally, monitoring each step.

9. Monitoring and Maintenance:

- Use tools like Prometheus, Grafana, and ELK Stack for monitoring and logging.
- Regularly update and patch the application.

10. Scaling:

- Scale horizontally by adding more nodes.
- Ensure the system can handle increased traffic and data volume.

Essential Tools

https://www.linkedin.com/advice/3/what-essential-tools-developing-distributed-zm9hf

1. Architecture Patterns
2. Communication Protocols
3. Serialization Formats
4. Development Frameworks
5. Testing Tools
6. Deployment Tools

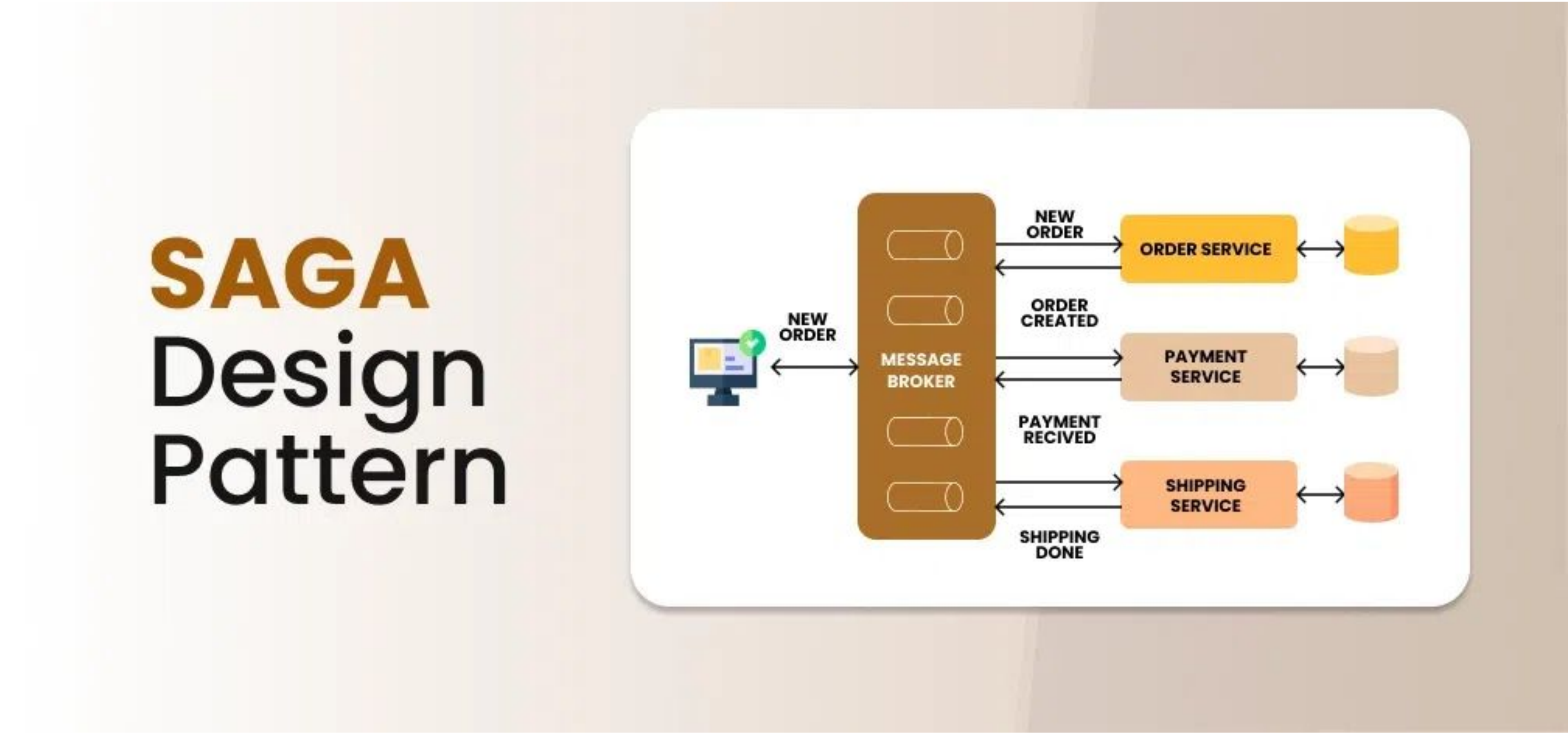
Architectural Patterns

One of the first steps in developing a distributed application is choosing an architecture pattern that suits your requirements and constraints. **An architecture pattern is a high-level design that defines the components, interactions, and responsibilities of your system.** Some common architecture patterns for distributed applications are client-server, peer-to-peer, publish-subscribe, microservices, and serverless. Each pattern has its own advantages and disadvantages, so you should consider factors such as scalability, availability, performance, security, and complexity when selecting one.

Architectural Patterns

Client-server (Basic/fundamental)
Peer-to-peer (No central control)
Layered (Request from top, reply from bottom)
API Gateway (Ambassador)
ETL (Extract-transform-load - data from various sources)
CQRS (Command Query Responsibility Seggregation)
Publish/subscrib (Producer/consumer)
Orchestration (Central coordinator)
Streaming (Real-time data stream processing)
Batching (compare to streaming)
Event-sourcing (Journal of events for audit & replay - financial system)
Saga (Multi-transactions control - like series)

Architectural Patterns



Communication Protocols

Communication protocols are the rules and formats that define how the nodes send and receive messages over the network. There are many communication protocols available for different purposes and scenarios, such as HTTP, TCP, UDP, WebSocket, MQTT, AMQP, RabbitMQ, and gRPC. You should select the communication protocols that match your system's functionality, reliability, and efficiency needs.

Serialization Formats

When you use communication protocols to send and receive data, you need to convert the data into a format that can be transmitted and understood by the nodes. This process is called serialization, and it involves encoding and decoding the data according to a specific format. Some common serialization formats for distributed applications are JSON, XML, Protobuf, Thrift, and Avro. Each format has its own pros and cons, such as readability, compactness, schema support, and compatibility.

Development Frameworks

Development frameworks provide you with ready-made tools and libraries for creating, testing, and running your system. Development frameworks are software packages that abstract and automate some of the common tasks and challenges of developing distributed applications, such as concurrency, fault tolerance, load balancing, and service discovery. Some popular development frameworks for distributed applications are Spring Boot, Django, Node.js, Akka, and Spark.

Testing Tools

Testing helps you ensure the quality, functionality, and performance of your system. Testing a distributed application can be challenging, as you need to simulate and verify the behavior of multiple nodes under different conditions and scenarios. To help you with testing, you can use testing tools that can generate, execute, and analyze test cases for your distributed application. Some examples of testing tools for distributed applications are JMeter, Postman, Selenium, Gatling, and PyTest.

Deployment Tools

Once you have developed and tested your distributed application, you need to **deploy it to the target environment** where it will run and serve its users. Deploying a distributed application can be complex, as you need to **manage and coordinate the configuration, installation, and update of multiple nodes** across the network. To make the deployment process easier and more reliable, you can use deployment tools that can automate and orchestrate the deployment of your distributed application. Some common deployment tools for distributed applications are Docker, Kubernetes, Ansible, Terraform, and AWS CloudFormation.

Q & A

Thank You !!

17