### **System Design: The Pub-sub Abstraction**

#### **Introduction**

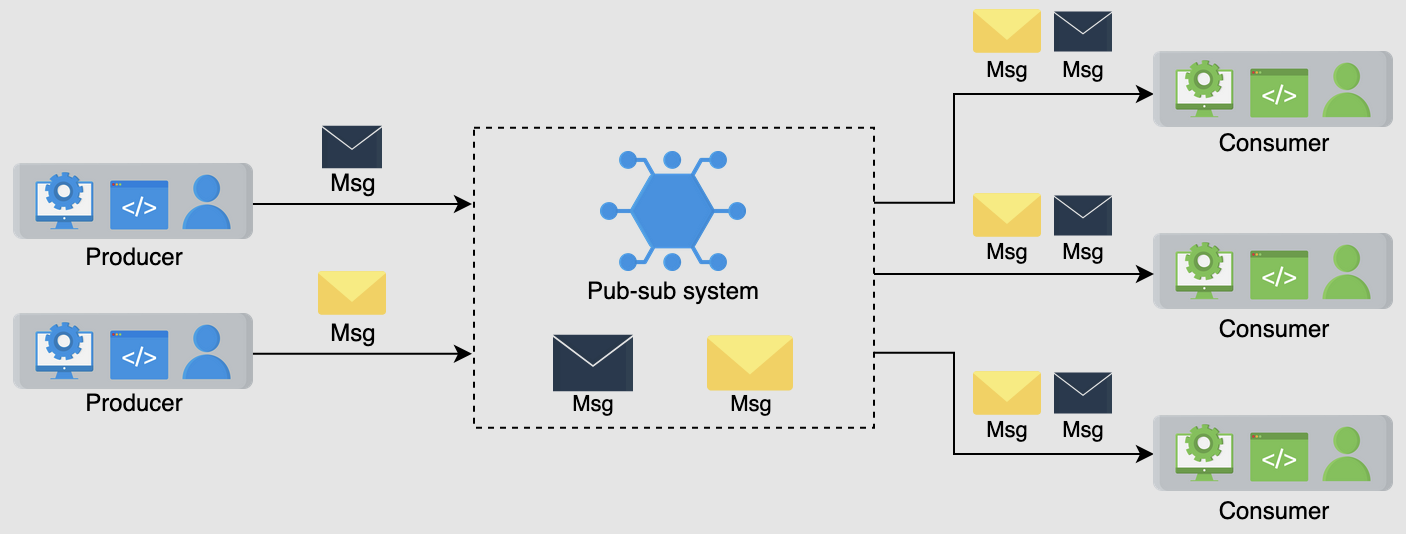
The document is part of the "Grokking Modern System Design Interview for Engineers & Managers" course and covers the fundamentals of designing a publish-subscribe (pub-sub) system, an essential concept in modern system architectures, especially within microservices and serverless environments.

#### **Pub-sub System Overview**

**Publish-subscribe (pub-sub) messaging**:

* An asynchronous communication method between services.
* Allows messages to be sent to different subsystems without requiring synchronous interactions.
* Typically used in microservices and serverless architectures to improve scalability and decouple producers (publishers) from consumers (subscribers).

**Example**: When a celebrity like Cristiano Ronaldo posts on social media, all followers (subscribers) receive the update without direct interaction between Ronaldo and each follower.



#### **Motivation**

* **Scalability**: Distributed systems often involve millions of machines. Using pub-sub systems enhances scalability by allowing independent operation and scaling of producers and consumers.
* **Decoupling**: By decoupling the components, changes or failures in one component do not impact others, leading to more robust systems.

#### **Design Considerations**

The document breaks down the pub-sub system design into specific lessons:

1. **Introduction**:
   * Discusses the use cases of pub-sub systems.
   * Defines the requirements for the system.
   * Outlines the API design.
2. **Design Approaches**:
   * Describes two primary designs for pub-sub systems:
     1. **With Messaging Queues**: Messages are sent to a queue where subscribers can pull them as needed.
     2. **With a Broker**: A broker manages the distribution of messages from publishers to subscribers, ensuring efficient and reliable delivery.

#### **Key Points in System Design**

* **Non-functional Characteristics**: Scalability, reliability, and decoupling of components.
* **Building Blocks**: Involves components such as load balancers, databases, content delivery networks (CDNs), and distributed caches.

#### **Practical Applications**

The document likely includes practical examples and case studies on how pub-sub systems are applied in real-world scenarios, such as:

* **Designing Newsfeed Systems**: For social media platforms.
* **Designing Notification Systems**: For applications like WhatsApp and Instagram.
* **Collaborative Tools**: Such as Google Docs for real-time document editing.

#### **Conclusion**

The detailed breakdown of pub-sub system design highlights the importance of asynchronous communication in modern distributed systems. By focusing on scalability and decoupling, the pub-sub abstraction provides a robust framework for developing and maintaining large-scale applications efficiently.

### **Additional Topics Covered in the Document**

* **Rate Limiter**: Ensuring fair resource usage and preventing abuse.
* **Distributed Search and Logging**: For efficient data retrieval and monitoring.
* **Task Scheduling and Sharded Counters**: For managing distributed tasks and counting operations.
* **Case Studies**: Examples of system design for popular services like YouTube, Quora, Google Maps, Yelp, Uber, and Twitter.

### **Summary**

This document serves as a comprehensive guide for engineers and managers to understand and design pub-sub systems, emphasizing practical implementation, scalability, and decoupling principles.

### **2. Introduction to Pub-sub**

#### **Overview**

The document titled "Introduction to Pub-sub" is part of the "Grokking Modern System Design Interview for Engineers & Managers" course. It delves into the essentials of the publish-subscribe (pub-sub) system, which is a critical aspect of modern system architectures. The content includes the definition, use cases, requirements, API design, and the building blocks necessary for implementing a pub-sub system.

#### **Pub-sub System Basics**

**Pub-sub Messaging**:

* **Asynchronous Communication**: It enables services to communicate asynchronously, which is particularly beneficial in serverless and microservices architectures.
* **Decoupling**: Publishers and subscribers are decoupled, meaning they operate independently. This allows systems to scale and evolve without tightly coupling components.

**Example**: A celebrity posting on social media where all followers get notified. Here, the celebrity is the publisher, the post is the message, and the followers are the subscribers.

#### **Use Cases of Pub-sub**

1. **Improved Performance**:
   * Push-based distribution eliminates the need for recipients to poll for updates, reducing latency and improving response times.
2. **Handling Log Ingestion**:
   * Pub-sub systems handle large volumes of user-interaction data, which can be used for analysis. For example, Meta uses a pub-sub system called Scribe for log ingestion.
3. **Real-time Monitoring**:
   * It allows real-time system monitoring by distributing raw or processed messages to multiple monitoring applications.
4. **Replicating Data**:
   * Changes can be distributed efficiently, such as in leader-follower protocols where the leader sends updates to followers. This ensures data consistency across distributed caches and multiple views of the same conversation in applications like WhatsApp.

#### **Requirements for a Pub-sub System**

**Functional Requirements**:

1. **Create a Topic**: Producers can create new topics.
2. **Write Messages**: Producers can send messages to topics.
3. **Subscription**: Consumers can subscribe to topics to receive messages.
4. **Read Messages**: Consumers can read messages from topics.
5. **Specify Retention Time**: Consumers can set retention periods for messages.
6. **Delete Messages**: Messages are deleted after their retention period.

**Non-functional Requirements**:

1. **Scalability**: The system must handle increasing numbers of topics and messages.
2. **Availability**: It should be accessible at all times for both producers and consumers.
3. **Durability**: Messages must not be lost and should reach the intended subscribers.
4. **Fault Tolerance**: The system should remain operational even during failures.
5. **Concurrency**: It should manage simultaneous read and write operations efficiently.

#### **API Design**

**Creating a Topic**:

text

create(topic\_ID, topic\_name)

* **Parameters**: topic\_ID (unique identifier), topic\_name (name of the topic).
* **Response**: Acknowledgment or error.

**Writing a Message**:

text

write(topic\_ID, message)

* **Parameters**: topic\_ID, message (max size 1 MB).
* **Response**: Acknowledgment or error.

**Reading a Message**:

text

read(topic\_ID)

* **Parameters**: topic\_ID.
* **Response**: Object containing the message.

**Subscribing to a Topic**:

text

subscribe(topic\_ID)

* **Parameters**: topic\_ID.
* **Response**: Acknowledgment.

**Unsubscribing from a Topic**:

text

unsubscribe(topic\_ID)

* **Parameters**: topic\_ID.
* **Response**: Acknowledgment.

**Deleting a Topic**:

text

delete\_topic(topic\_ID)

* **Parameters**: topic\_ID.
* **Response**: Acknowledgment.

#### **Building Blocks**

1. **Database**: To store subscription details and topic information.
2. **Distributed Messaging Queue**: To store messages sent by producers.
3. **Key-value Store**: To maintain information about consumers.

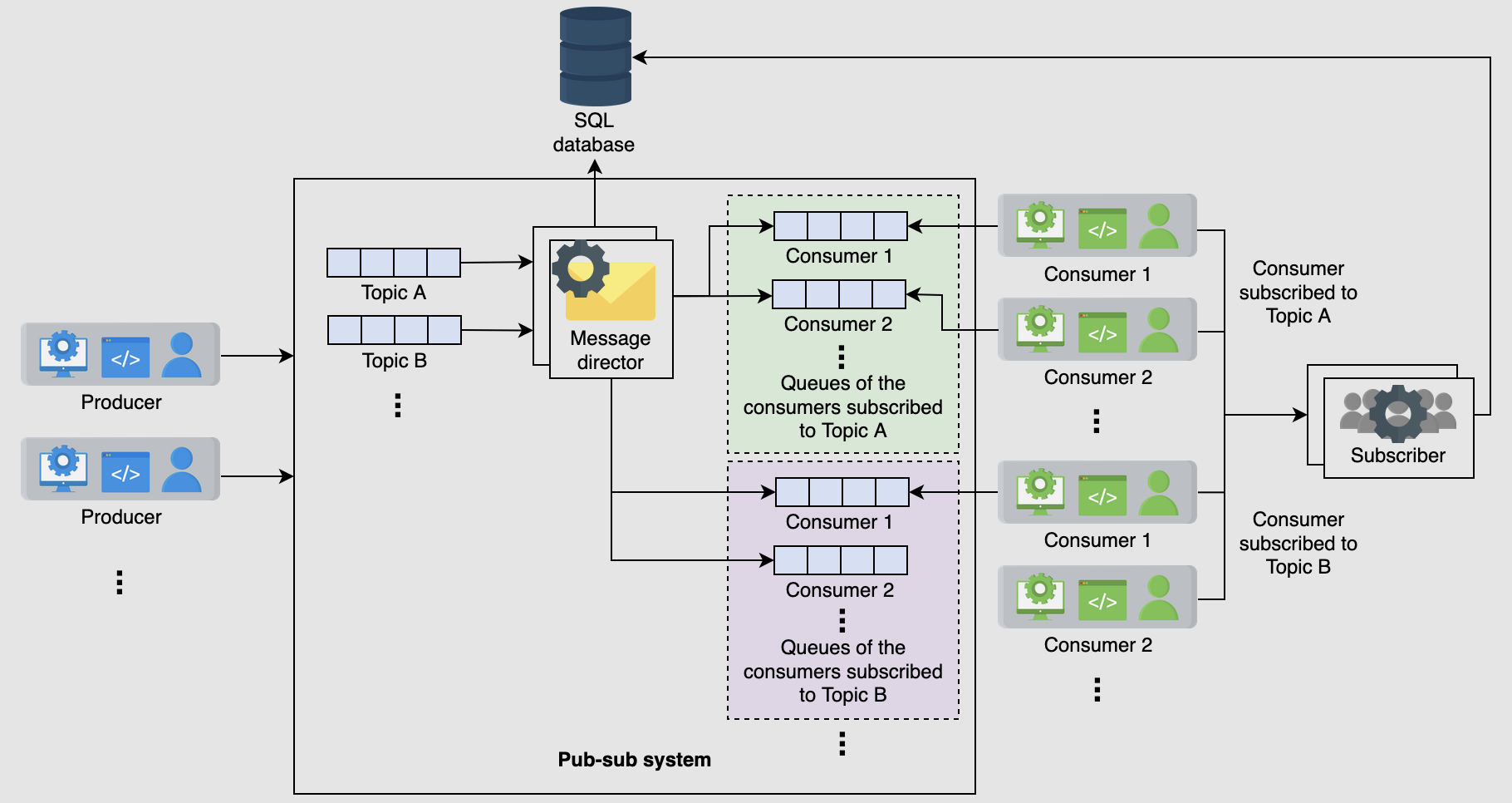
### **Conclusion**

This document serves as a foundational guide for understanding and implementing a pub-sub system. It covers the theoretical aspects, practical use cases, detailed functional and non-functional requirements, and API design, ensuring a comprehensive grasp of the pub-sub abstraction necessary for modern system design.

### **3. Design of a Pub-sub System**

#### **Overview**

The document titled "Design of a Pub-sub System" is part of the "Grokking Modern System Design Interview for Engineers & Managers" course. It provides a comprehensive guide to designing a publish-subscribe (pub-sub) system, exploring two main design approaches, their components, and the considerations involved in building a scalable, reliable, and efficient system.



#### **Key Components and Concepts**

**Pub-sub Messaging**:

* **Publish-subscribe (pub-sub) messaging**: An asynchronous communication model where producers publish messages to topics, and consumers subscribe to those topics to receive messages.
* **Asynchronous communication**: Enables independent operation of producers and consumers, enhancing scalability and decoupling components.

#### **First Design: Using Distributed Messaging Queues**

1. **Components**:
   * **Topic Queue**: Each topic is represented as a distributed messaging queue where producers write messages.
   * **Database**: Stores subscription details, such as which consumer is subscribed to which topic.
   * **Message Director**: Reads messages from the topic queue, fetches subscriber details from the database, and sends messages to the consumer queue.
   * **Consumer Queue**: Each consumer has a separate distributed queue to read messages.
   * **Subscriber Service**: Adds subscription entries to the database when a consumer subscribes to a topic.
2. **Process**:
   * Consumers subscribe to topics, and the subscriber service updates the database.
   * Producers write messages to topic queues.
   * The message director reads messages from the topic queue, retrieves subscriber details, and sends messages to consumer queues.
   * Consumers read messages from their queues.
3. **Challenges**:
   * Managing millions of queues for numerous topics and subscribers is resource-intensive.
   * Duplicating messages across multiple subscriber queues leads to unnecessary data duplication.

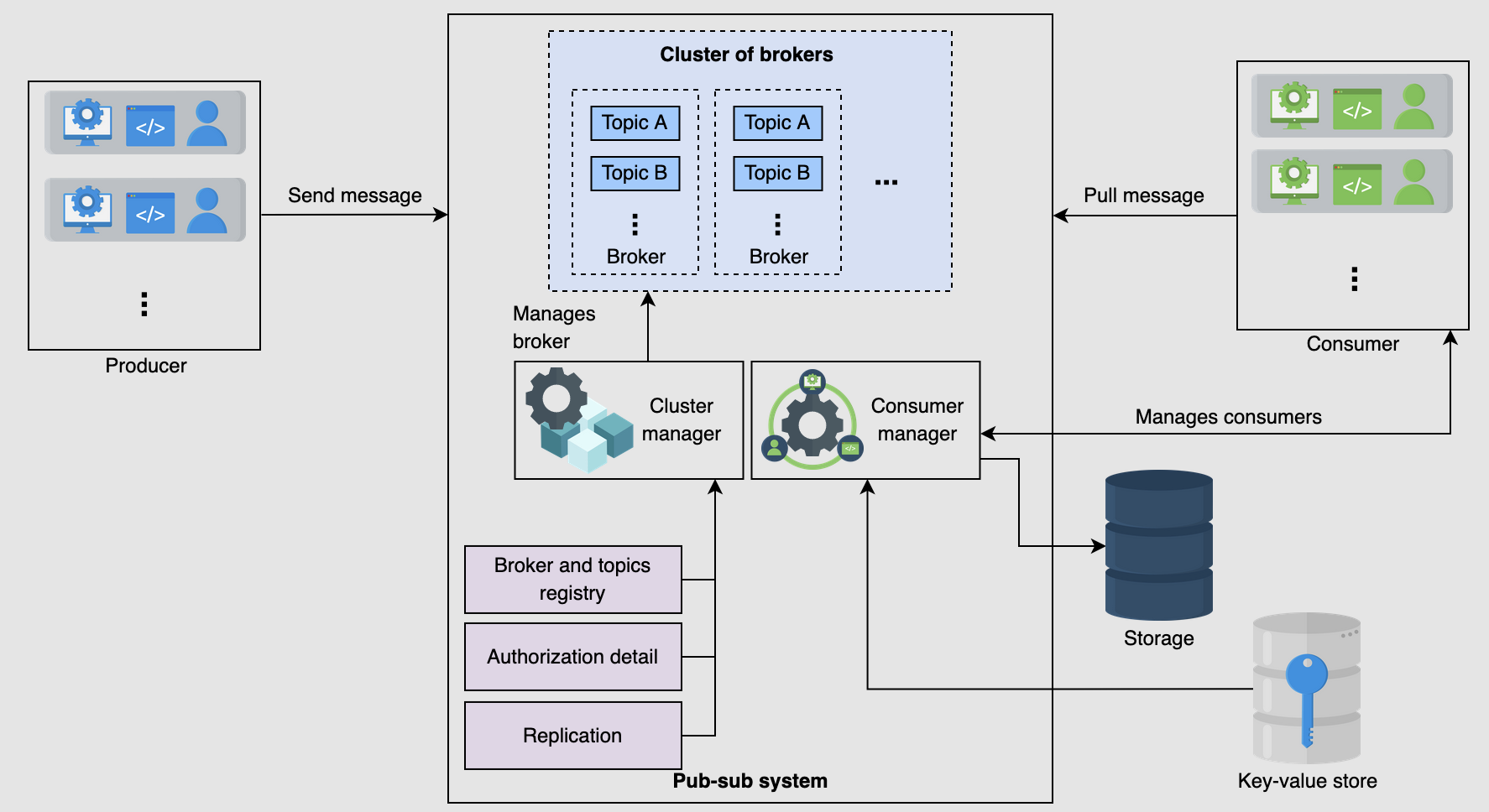
#### **Second Design: Using Brokers and Cluster Management**

1. **High-level Design**:
   * **Broker**: Manages messages, stores them from producers, and allows consumers to read them. Topics are partitioned for scalability and availability.
   * **Cluster Manager**: Oversees multiple broker servers, manages replication, and handles broker failures.
   * **Storage**: Uses a relational database for consumer details, such as subscription information and retention periods.
   * **Consumer Manager**: Manages consumer authorizations, retention times, and message delivery options.
2. **Broker Details**:
   * **Partitions**: Topics are divided into partitions, which are stored in the broker's local storage. Partitions contain messages segmented by offset addresses.
   * **Segments**: Help identify the start and end of messages. Consumers can read messages from specific offsets.
   * **Replication**: Managed by the cluster manager using a leader-follower approach. Ensures data availability and fault tolerance.
3. **Consumer Manager**:
   * **Verification**: Ensures consumers are authorized to read messages from specific topics.
   * **Retention Time Management**: Enforces message retention policies based on consumer requirements.
   * **Message Delivery Options**: Supports both push and pull methods for message delivery to consumers.
   * **Offset Management**: Maintains the read offset for each consumer to ensure messages are read sequentially and efficiently.
4. **Design Considerations**:
   * **Acknowledgment**: Producers are notified when messages are successfully stored. Consumers acknowledge receipt of messages.
   * **Concurrency**: The system handles simultaneous read and write operations efficiently.
   * **Scalability**: The system scales with increasing topics and message loads.
   * **Fault Tolerance**: The system continues to operate despite failures.

#### **Finalized Design**

The finalized design integrates the components and considerations discussed:

* **Brokers** manage topics, partitions, and segments, ensuring message persistence and availability.
* **Cluster Manager** handles replication and broker health, ensuring system resilience.
* **Consumer Manager** oversees consumer authorizations, retention policies, and delivery options, maintaining efficient and secure message distribution.



#### **Conclusion**

The document provides a detailed exploration of two designs for a pub-sub system, highlighting the benefits and challenges of each approach. The final design leverages brokers and cluster management to address scalability, fault tolerance, and efficient message delivery, making it suitable for large-scale, data-intensive applications.