Trade-offs in Gmail Service Design

Designing a large-scale email system requires balancing performance, scalability, consistency, and maintainability. Below are some key trade-offs I considered while designing this system.

1. Separate Service Manager vs. Keeping it Inside the Gateway Service

Pros of a Separate Service Manager:

- Decouples responsibilities: The Gateway Service only handles routing, while the Service Manager manages service discovery.
- Reduces complexity in the Gateway, making it more scalable.
- Improves maintainability by allowing independent updates and scaling of services.

Cons of a Separate Service Manager:

- Increases network latency since every request to the Gateway requires an additional call to the Service Manager.
- Introduces additional infrastructure overhead by requiring a separate component to monitor and scale.

I decided to go with a separate Service Manager despite the minor performance loss. This approach improves fault isolation and enables independent scaling of services.

2. Global Redis Cache vs. Local Redis for Each Microservice

Pros of a Global Redis Cache:

- Provides a single source of truth for caching across all services, improving consistency.
- Simplifies management by eliminating the need to handle multiple Redis instances.
- Utilizes resources efficiently by centralizing caching in a single optimized cluster.

Cons of a Global Redis Cache:

- Adds slight latency compared to a local Redis instance in each microservice.
- Increases the load on the Redis cluster, potentially creating bottlenecks.
- Creates a potential single point of failure, where a Redis crash could impact multiple services.

I chose a global Redis cache for better consistency and easier management. With Redis clustering and replication, the risk of failure is minimized.

3. Kafka for Event-Driven Communication

Pros of Using Kafka:

- Enables asynchronous processing, reducing API response times.
- Improves fault tolerance by ensuring email events are not lost if a service goes down.
- Handles spikes in email traffic efficiently, ensuring scalability.

Cons of Using Kafka:

- Increases system complexity due to message failures, retries, and consumer lag.
- Requires additional computing resources, leading to higher infrastructure costs.

Kafka was selected because an event-driven architecture is essential for handling email traffic efficiently. Without a message queue, sudden traffic spikes could overload the system.

4. Kubernetes Persistent Volumes vs. Cloud Object Storage (S3/MinIO)

Pros of Using Kubernetes Persistent Volumes:

• Provides better integration with containerized workloads.

• Reduces latency since data is stored within the same cluster.

Cons of Using Kubernetes Persistent Volumes:

• Less scalable than object storage solutions like S3, which are designed for large-scale storage.

• Offers lower redundancy compared to cloud storage options.

Since the system is deployed using Kubernetes and Docker, I opted for Kubernetes Persistent Volumes for initial deployment.

Rate Limiting Strategy

I implemented the **Token Bucket** algorithm for rate limiting in the Gateway Service.

How it Works:

- Each user gets a bucket with a fixed number of tokens.
- Each request consumes one token.
- Tokens are refilled at a fixed rate (e.g., 100 requests per minute).
- If a user exhausts their tokens, additional requests are rejected or delayed.

Pros of Using the Token Bucket Algorithm:

- Allows short bursts of high request rates while maintaining a long-term limit.
- Ensures a smooth user experience by permitting occasional spikes (e.g., sending multiple emails at once).
- Works efficiently with Redis for distributed rate limiting.

Cons of Using the Token Bucket Algorithm:

- Requires additional memory, as each user has a separate token bucket in Redis.
- Does not strictly enforce per-second rate limits, as users can make multiple requests in quick succession until their bucket is empty.

I chose this approach because Gmail needs to support burst requests while ensuring that users do not exceed long-term limits. The Token Bucket algorithm balances flexibility and control, making it an effective choice.