High-Level Design (HLD) for Gmail Service

Overview

The Gmail service is a highly scalable email system designed using a microservices architecture. It ensures efficient email handling with features such as authentication, email storage, search, and distributed processing.

Architecture Overview

The system consists of multiple microservices to handle different aspects efficiently. Below is the refined architecture, including key services:

Gateway & Traffic Management

Gateway Service – Acts as an entry point, handles authentication, rate limiting, and routes requests to appropriate microservices.
Load Balancer – Distributes traffic across multiple instances of each service.
Rate Limiter – Prevents abuse by limiting API requests (uses Redis + Token Bucket).

Email Management

Email Service – Handles sending and receiving emails via SMTP & IMAP.
Search Engine – Indexes emails and enables fast email search.
Message Queue (Kafka) – Ensures reliable email event processing (e.g., new email arrival).

Storage & Security

Distributed Drive (Kubernetes PVC) – Stores email attachments.
Virus Checker – Scans attachments for malware before storage.
Space Detector – Monitors storage consumption and optimizes space usage.

User & Profile Management

Auth Service – Manages user authentication and JWT-based authorization.
Profile Service – Manages user details such as name, profile picture, and bio.
Contacts Manager – Manages the user’s contact list, enabling auto-suggestions.
Preference Store – Stores user settings like themes, email filters, notification preferences.

Monitoring & Logging

Monitoring Service (Prometheus, Grafana) – Tracks system health, API latency, and failures.
Logging Service (ELK Stack) – Collects logs from all microservices for debugging and analysis.

Trade-offs in Gmail Service Design

Separate Service Manager vs. Keeping it Inside the Gateway Service

Pros:

- Decouples responsibilities: The Gateway Service only handles routing, while the Service Manager handles service discovery.
- Reduces complexity in the Gateway: The Gateway Service doesn't need to track service health, making it more scalable.
- Improved maintainability: Services can be managed, updated, or scaled independently.

Cons:

- Additional network latency due to a separate call to the Service Manager.
- More infrastructure overhead with a dedicated Service Manager.

The decoupling of concerns outweighs the slight performance loss. Having a separate Service Manager ensures better fault isolation and allows independent scaling of services.

Global Redis Cache vs. Local Redis for Each Microservice

Pros of Global Redis:

- **Consistency:** A single source of truth for caching across all services.
- **Easier management:** One centralized cache is easier to maintain.
- **Efficient resource utilization:** Eliminates the need for multiple Redis instances.

Cons:

- **Higher latency:** Slightly slower access times compared to a local Redis instance.
- **Potential bottlenecks:** A global Redis can become a single point of failure.

Consistency and ease of management were prioritized over minor performance trade-offs. With proper Redis clustering and replication, failures can be mitigated.

Choosing Kafka for Event-Driven Communication

Pros:

- **Asynchronous Processing:** Reduces API response time by queuing tasks.
- **Fault Tolerance:** Ensures email events are not lost even if a service goes down.
- **Scalability:** Kafka allows handling spikes in email traffic smoothly.

Cons:

- **Complexity:** Managing message failures, retries, and consumer lag adds engineering overhead.
- **Infrastructure cost:** Running Kafka requires additional computing resources.

A message queue is essential for an email service. Without Kafka, spikes in traffic (e.g., promotional emails) could overwhelm the system.

Why Not Use S3/MinIO for Attachments?

Pros of Using Kubernetes Volumes Instead:

- **Better Integration** with containerized workloads.
- **Lower latency** as data is stored within the same cluster.

Cons:

- **Scalability issues:** Object storage like S3 is better for large-scale storage.
- **Less redundancy** compared to cloud storage options.

Since the project is deployed using Kubernetes and Docker, using K8s Persistent Volumes (PV) made more sense for an initial deployment.

Rate Limiting Strategy

Token Bucket (Best for Gmail Service)

How it Works

- Each user gets a "bucket" filled with tokens.
- Each request consumes a token.
- Tokens refill at a fixed rate (e.g., 100 requests per minute).
- If the bucket is empty, the request is rejected or delayed.

Pros:

- **Smooth User Experience** – Allows bursty requests (e.g., sending multiple emails at once) while enforcing an average rate.
- **Efficient for APIs** – Works well for email-sending APIs, login requests, and search queries.
- **Predictable Usage** – Ensures consistent rate limits for different user types (free vs. paid accounts).
- **Easy to Scale** – Works well in distributed systems with Redis.

Cons:

- **More Memory Usage** – Each user requires a separate token bucket in Redis.

- **Not Instantaneously Strict** – Users can burst requests until their bucket is empty.

Gmail needs to handle burst requests (e.g., sending multiple emails) while enforcing long-term limits. The Token Bucket algorithm is ideal as it balances user experience and API protection.

Conclusion

Trade-offs in system design are always a balance between performance, scalability, consistency, and maintainability. The choices made in this project prioritize scalability and consistency, even if they introduce slight overhead. With proper monitoring, logging, and failover mechanisms, the system ensures high availability and reliability.