Unique ID Generation

CUNY Brooklyn College
Mohamed Massoud, Jubael Mamon, Kamali McKenzie, Ahmed Mostafa

Design and Implementation of Large-Scale Applications
Professor Priyanka Samanta
December 17th, 2025

Role	Members
Team Lead	Kamali McKenzie
Research Analyst	MD Jubael Mamon
Diagram & Documentation Lead	Mohammed Massoud
Presentation Lead	Ahmed Mostafa

Weekly Progress Tracker	Phase Completion/Status
Meeting 1: 11/12	Phase 1
Meeting 2: 11/17	Phase 2
Meeting 3: 12/1	Phases 3-5
Meeting 4: 12/7	Phases 6-8

Phase 1: Problem Statement

Overview of Project Objectives

In today's technological revolution, modern software systems are in great need of unique identifiers that can track essential data such as users, devices, transactions, and content, while also being unique and collision-free to prevent conflicts across a network of servers. Existing ID algorithms, such as UUID, ULID, and Snowflake, are the backbone of unique ID generation as we know it, but they lack the readability and customization that users often want. This project aims to design a third-party Unique ID Generation Service that allows users to generate unique IDs in customizable formats while also serving as a pillar of reliability, consistency, and global uniqueness.

Users and Use Cases

The primary users of this service include:

- Software Developers (who utilize APIs to generate IDs so that they can track orders, payments, and analytical events)
- High-Scale Distributed Systems (companies that need thousands or even millions of IDs per second to fulfill the needs of e-commerce platforms, such as user IDs, authentication IDs, order IDs, and shipping IDs).

Users should be able to:

- Generate globally unique, secure, and sortable IDs with low latency.
- Customizability through format is a cornerstone of our service; they can also choose from several ID formats that suit their needs.
- Monitor the volume of ID requests and ensure constant availability to minimize downtime.
- Designed for high scalability and real-world problems.
- Made with millions of users in mind, with thousands to millions of ID requests per second, and also factors in a user base that spans multiple regions.

Overarching Goal

Our goal is to create a service fundamental to distributed software, one in which reliability, uniqueness, and customizability are at the forefront, while also focusing heavily on low latency and high throughput to provide a fantastic user experience so users can trust us with their ID infrastructure.

Phase 2: Requirement Analysis

Requirements Table

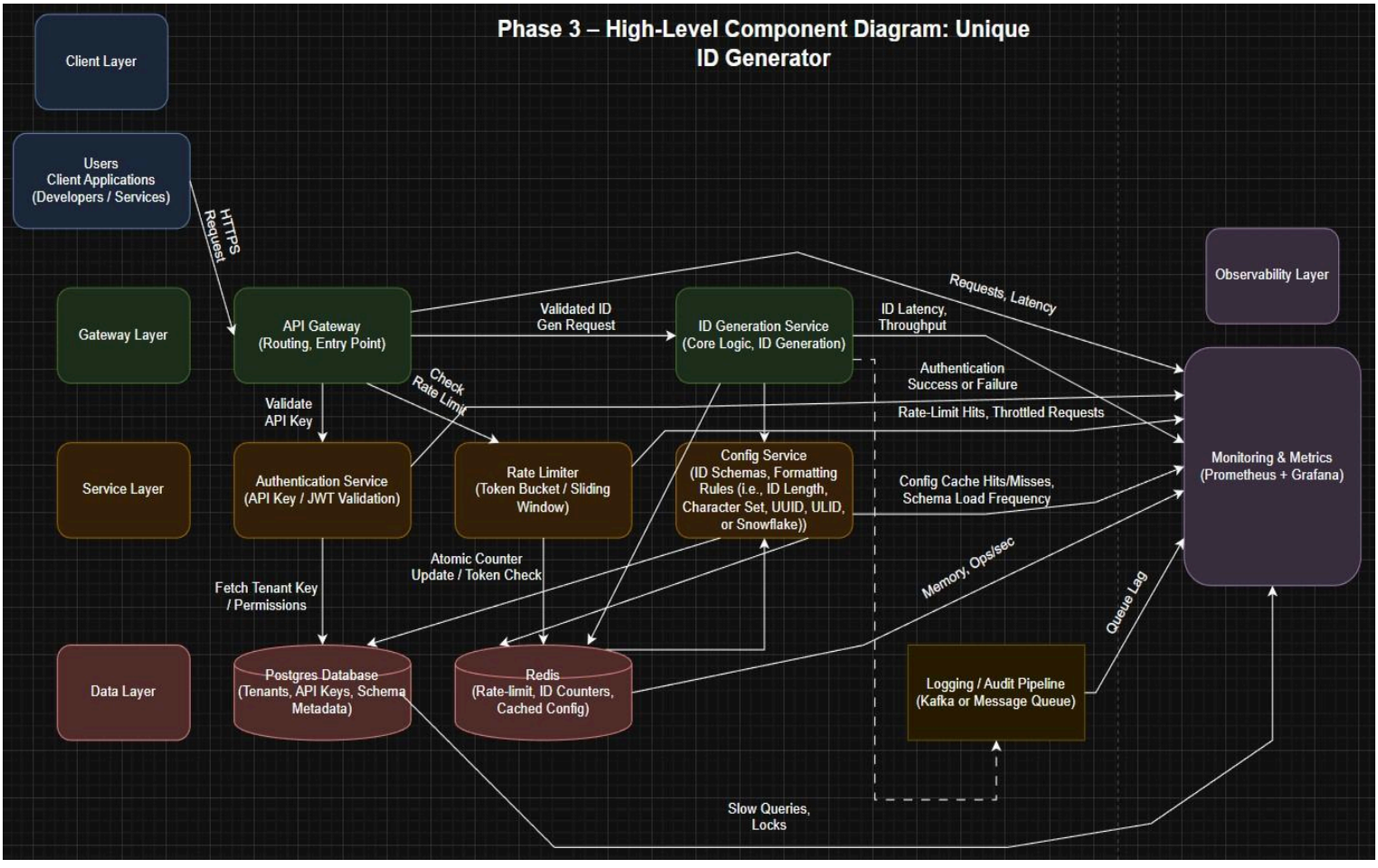
Functional Requirements	Description	Priority
API for ID Generation	Provide REST endpoints for generating unique IDs.	Must-have
Custom ID creation while also supporting standard formats	Allow both predefined formats (UUID, ULID, Snowflake) and	Should-Have

	custom ID rules.	
Collision Prevention	Guarantee that no two generated IDs are identical.	Must-Have
Admin Dashboard	Web UI for managing clients, namespaces, and policies.	Should-Have
Load Balancer	Distribute incoming traffic across multiple ID generation nodes.	Must-Have
Rate Limiting	Enforce per-client request limits to prevent abuse.	Must-Have
Logging of ID Generation Events	Every ID generated is recorded asynchronously.	Must-Have
Authentication through API tokens (JWT Tokens)	Validate dashboard users with JWT and clients with API keys.	Must-Have
Security(Authentication and Authorization)	Ensure only authorized users and admins can access services.	Must-Have
Dashboard for monitoring ID requests	Admin UI displaying analytics, traffic, and logs.	Should-Have
Support for client libraries of programming languages	Provide SDKs (e.g., Python, JS, Go) for easy integration.	Nice-to-Have
Ability to request large batches of IDs at once	Allow requesting large sets of IDs in a single API call.	Should-Have
Length Configuration	Enable configuration of ID length for custom formats.	Nice-to-Have
Monitoring & Metrics	Track performance, errors, and system health.	Should-Have

Non-Functional Requirements	**Description**	**Priority**
Low Latency	IDs should be generated and	Must-Have

	returned extremely fast (<5 ms).	
High Throughput	System must handle millions of ID requests per second.	Must-Have
Horizontal Scalability	Add more nodes to increase capacity with no redesign.	Must-Have
High Availability	System must remain operational with minimal downtime.	Must-Have
Fault Tolerance	System should continue functioning despite component failures.	Must-Have
Usability (friendly interface and documentation)	Dashboard and client docs should be simple and intuitive.	Should-Have
Storage	Efficient, durable storage of configuration and logs.	Must-Have

Phase 3 - Conceptual Design and Component Breakdown



Component Descriptions

Client Applications

- The client application represents external systems consuming the ID generation service. It manages tenant credentials, sends HTTPS REST requests to generate IDs, and receives JSON responses containing the generated identifiers.

API Gateway

- The API Gateway acts as the single entry point for all client traffic. It receives incoming requests, validates authentication and rate limits, routes authorized traffic to backend services, and returns responses to clients. This centralizes security, traffic control, and request routing.

Authentication Service (JWT)

- The authentication service verifies client credentials and issues secure JWT tokens. It ensures that only authorized tenants and administrators can access system resources by validating request identity through the API Gateway and checking tenant permissions stored in PostgreSQL.

Rate Limiter

- The rate limiter protects system stability by enforcing per-tenant usage limits. It intercepts requests at the gateway level and uses Redis-backed counters and sliding windows to prevent abuse, accidental overload, or denial-of-service scenarios.

ID Generation Service

- The ID Generation Service is responsible for producing unique, collision-free identifiers using algorithms such as Snowflake, ULID, UUID, or custom formats. It processes validated requests from the API Gateway, retrieves sequence or state data from Redis, publishes audit events to Kafka, and returns generated IDs to the client with minimal latency.

Configuration Service

- The configuration service manages all ID generation rules and tenant-specific settings, such as ID formats, lengths, and namespaces. It persists configuration data in PostgreSQL and provides frequently accessed values to the ID Generation Service via Redis caching to reduce latency.

Logging Service

- The logging service asynchronously processes system events, generated ID records, and error logs. It consumes messages from Kafka and stores them in the database for auditing, analytics, and dashboard visualization, ensuring logging does not impact request latency.

Caching Layer (Redis)

- Redis serves as a high-performance in-memory cache for rate limiting counters, configuration data, and ID generation metadata. It is accessed by the ID Generation Service and Rate Limiter to minimize database load and ensure fast, atomic operations.

Database (PostgreSQL)

- PostgreSQL stores structured system data, including tenants, API keys, namespaces, rate limits, and configuration metadata. It is accessed by the authentication and configuration services, while high-volume log data is written asynchronously through event consumers.

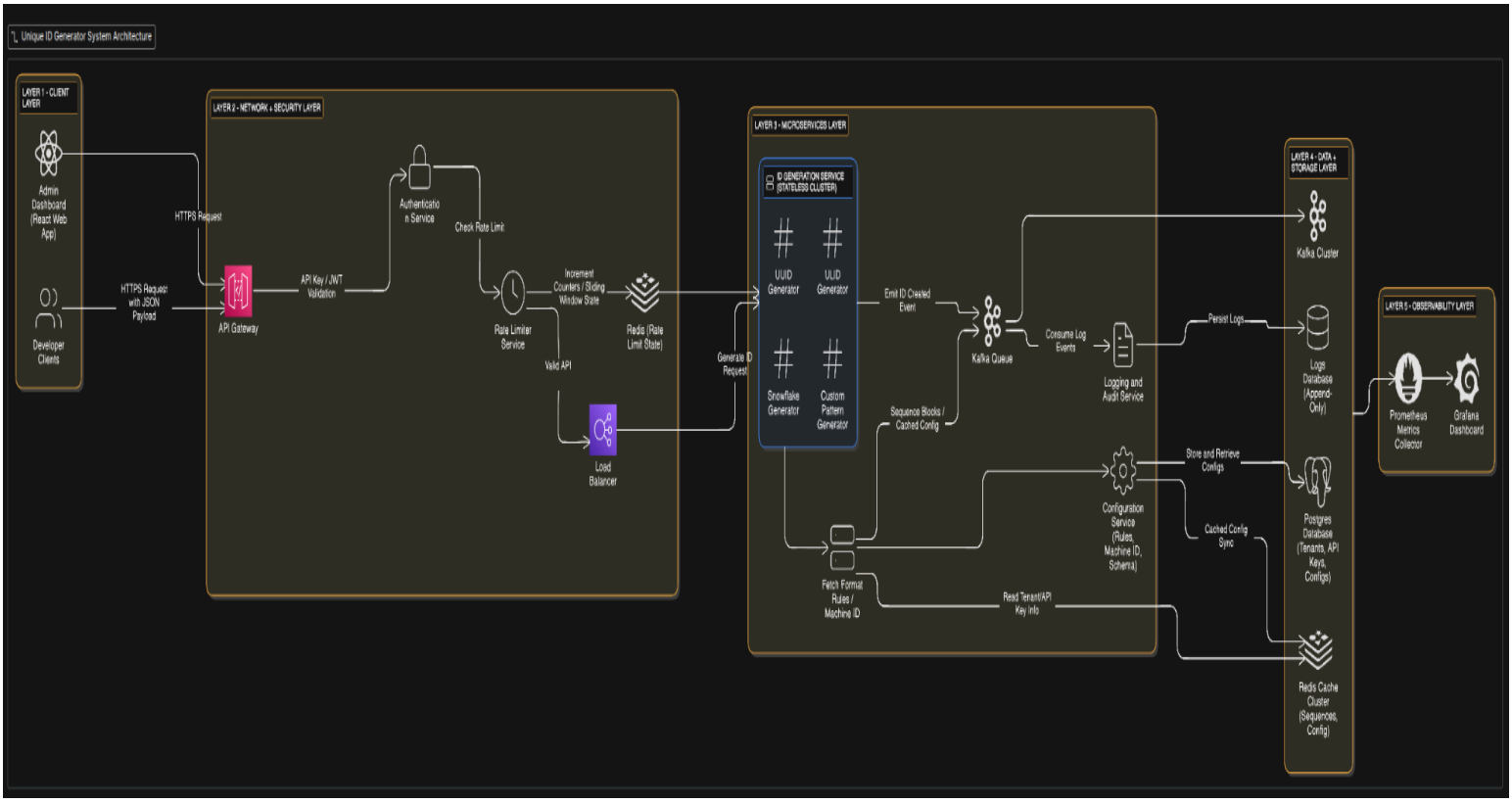
Monitoring & Metrics Layer

- The monitoring layer tracks system health, performance, and failures by collecting metrics such as request latency, error rates, and resource utilization. It integrates with all backend components and generates alerts via dashboards or notifications to ensure operational visibility and reliability.

Phase 4 - Deep System Architecture

System Architecture Diagram

https://app.eraser.io/workspace/gIfg7kfsmNVHaTV9xN91?origin=share



Comparison between Approaches

Microservices Architecture (Chosen)

- We use a microservices architecture to scale each component independently and prevent failures from cascading across the system. The ID Generation Service can scale horizontally for high throughput, while slower components like logging and configuration remain isolated.

Monolith (Rejected)

- A monolithic design was rejected because it forces all modules to scale and deploy together, creating single points of failure and higher operational risk. A bug in logging or configuration could bring down ID generation, and scaling for peak loads would be unnecessarily expensive and inefficient.

Gateway + Load Balancer

- The API Gateway centralizes authentication, rate limiting, and request validation, protecting backend services from overload or abuse. In front of it, a load balancer distributes traffic across stateless ID generation nodes, enabling simple and elastic horizontal scaling.

Data Layer (PostgreSQL + Redis + Kafka)

- PostgreSQL provides strong consistency and structured schemas essential for tenant configs, namespaces, and API keys. Redis offers extremely low-latency access for cached configs and rate limiting. Kafka decouples logging from ID generation, ensuring the core service stays fast while logs are processed asynchronously at massive scale.

Stateless ID Generation Nodes

- IDGen nodes remain stateless to eliminate coordination overhead and simplify scaling. Each node only needs a machine ID, timestamp, and sequence counter, enabling linear growth with no shared state or leader election, while still guaranteeing globally unique IDs.

Event-Driven Logging

- Logging is handled asynchronously through Kafka to avoid blocking the request path. This ensures ID generation remains sub-millisecond even under heavy logging volumes while still guaranteeing durable, ordered audit trails processed by downstream consumers.

Time-Partitioned Logs

- Log tables are partitioned by time to support high write throughput, fast analytical queries, and instant data retention cleanup. Dropping old partitions avoids expensive deletes and keeps the database consistently performant as the system scales.

Scalability Strategy

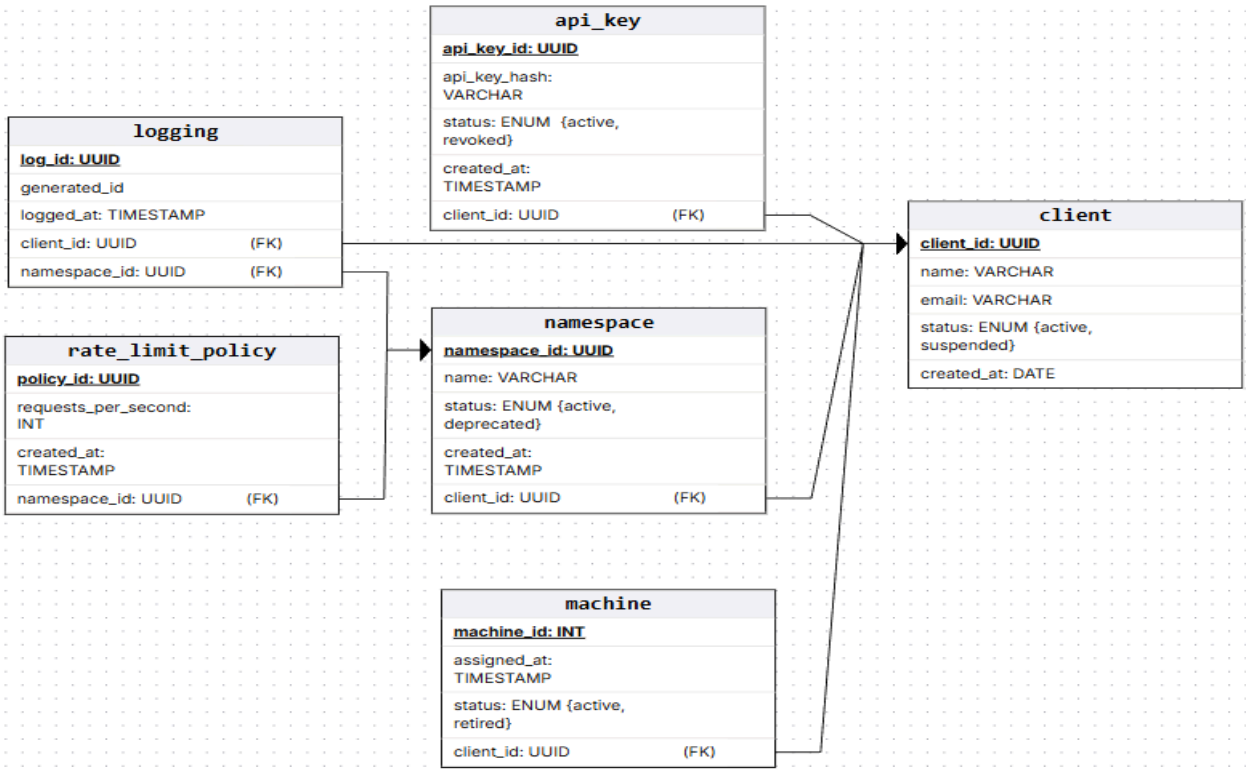
- The architecture scales horizontally at every tier—IDGen nodes, Redis clusters, Kafka partitions, and Postgres replicas. The share-nothing design eliminates bottlenecks, while rate limiting and caching ensure predictable performance even at millions of requests per second.

System Architecture Flow

- The system uses a microservices architecture optimized for low-latency, high-throughput ID generation. Client requests enter through the API Gateway, where authentication, JWT validation, and rate limiting are enforced before traffic is distributed across stateless ID Generation nodes. Each node retrieves cached configuration data from Redis and generates collision-free IDs using UUID, ULID, Snowflake, or custom formats, returning results immediately to preserve sub-millisecond latency. Logging is handled asynchronously by publishing events to Kafka, which are later consumed by a separate Logging Service and persisted to storage. Configuration and tenant metadata are stored in PostgreSQL and cached in Redis to minimize database load. Monitoring tools collect metrics across all components to ensure visibility, scalability, and fault isolation under heavy load.

Phase 5: Data Design

ER Diagram



Database Choice & Rationale

Chosen Database: PostgreSQL (SQL)

- PostgreSQL is selected because the system requires strict consistency, relational integrity, and powerful analytical querying for the admin dashboard. SQL schemas prevent malformed configurations and enforce referential integrity across tenants, namespaces, and format rules. PostgreSQL also provides native table partitioning, B-tree indexing, transactional safety (ACID), and mature replication. These features allow us to ingest millions of log entries while still supporting complex analytical queries.

Why Not NoSQL?

- NoSQL excels at raw write throughput but sacrifices strong consistency and complex querying. Our system requires relational joins, strong referential guarantees, and structured configurations—making SQL the more appropriate choice.

Trade-offs

- **Consistency vs. Write Latency:** By selecting PostgreSQL, we prioritized **Data Integrity (Strong Consistency)** over raw write speed.
    - The Trade-off: Unlike NoSQL systems that accept writes immediately without validation (Eventual Consistency), our SQL database must enforce schema constraints and write to the Write-Ahead Log (WAL) before confirming success. This introduces a slight increase in write latency per request to guarantee that no audit logs are ever lost or corrupted.
- **Analytical Power vs. Scaling Complexity:** We chose SQL to enable complex queries for the Admin Dashboard (e.g., JOINs and Aggregations).
    - The Trade-off: This makes **Horizontal Scaling** significantly more complex. While reading data is easily scaled via Read Replicas, scaling writes beyond a single node requires complex manual sharding, whereas NoSQL would handle this automatically. We accept this operational complexity to enable running deep analytical queries natively.

- **Hybrid Consistency Model:** Strong consistency in the control plane, eventual consistency in the data plane, optimizing both safety and performance.

Database Schema Strategy

Our schema design addresses the critical requirement of balancing high-velocity write ingestion with the need for analytical querying via the admin dashboard. To sustain insertion rates of millions of records per

second into a relational database, we utilize native declarative partitioning and targeted indexing rather than relying on a single monolithic table structure.

- **Time-Based Partitioning:**
  To manage the massive accumulation of log data, the log_entry table is partitioned by range based on the creation timestamp. We instantiate a new partition for each day (e.g., logs_2025_11_01). This strategy ensures that write operations interact primarily with a smaller, active dataset (the "hot" partition), significantly reducing index maintenance overhead and keeping insertion latency low compared to writing to a table containing billions of historical rows.
- **Strategic Indexing:**
  To facilitate rapid data retrieval for the administrative dashboard, we implement B-Tree indexes on the client_id and created_at columns within each partition. This allows the query planner to execute efficient index scans when filtering by specific clients or time ranges, avoiding full table scans which would be computationally prohibitive at this scale.
- **Strong Consistency (ACID):**
  By utilizing PostgreSQL, the system adheres to a Strong Consistency model. Unlike eventually consistent NoSQL systems where replication lag can cause temporary data invisibility, our architecture guarantees that once a transaction commits a log entry, it is immediately durable and visible to the logging service. This ensures strict accuracy for audit trails without the risk of data loss or temporary discrepancies.

Phase 6: Technology & Communication Decisions

6.1 Communication Choices: API vs WebSocket vs Webhooks

The system must support fast, secure, and reliable communication between client services, admin interfaces, and internal components. Three major communication strategies were evaluated: REST APIs, WebSockets, and Webhooks:

WebSockets

WebSockets allow a persistent and bidirectional connection between client and server. However, this system does not need such a connection. ID generation is not like a streaming process, requiring a continuous feed of IDs. WebSockets would add unnecessary complexity and be harder to scale. They are more ideal for chat and live feeds, not simple API calls.

Webhooks

Used for asynchronous notifications, such as rate-limit threshold alerts and high-latency warnings, Webhooks wouldn’t normally be used for ID generation, only for operations that benefit from an asynchronous event delivery. It can benefit the system by decoupling the alerting mechanism from the core service, just so that ID generation remains fast and uneffected, but it is not necessary.

REST APIs (Chosen)

ID Generation is a request-response operation. A client asks for an ID, and the service returns it immediately. And that is why REST is ideal. REST APIs can be used for client services requesting IDs, admin dashboard and configuration service interaction, and internal service-to-service communication. It is simple, widely supported, and there is low overhead which is ideal for generating a single ID. It integrates nicely with rate limiting, authentication, and load balancing, and it keeps latency extremely low. Furthermore, it is also stateless, all things needed for ID generation. It is the best option for request-response workflows.

6.2 Data Exchange Format: JSON

JSON (JavaScript Object Notation) will be the primary data exchange format across all external and internal APIs of this system.

JSON was chosen after comparing alternatives such as XML, Protocol Buffers, and MessagePack. And it was chosen due to its support in almost every major programming language, such as Java, Python, Go, etc. All of these languages can easily parse and generate JSON, making it ideal for client services, which may be implemented in different tech stacks.

It was also chosen because it is perfect for REST APIs. REST APIs being the main communication method of the system, JSON will fit naturally. Not to mention it is also lightweight for small payloads and human-readable.

6.3 Load Balancing, Caching & Message Queues

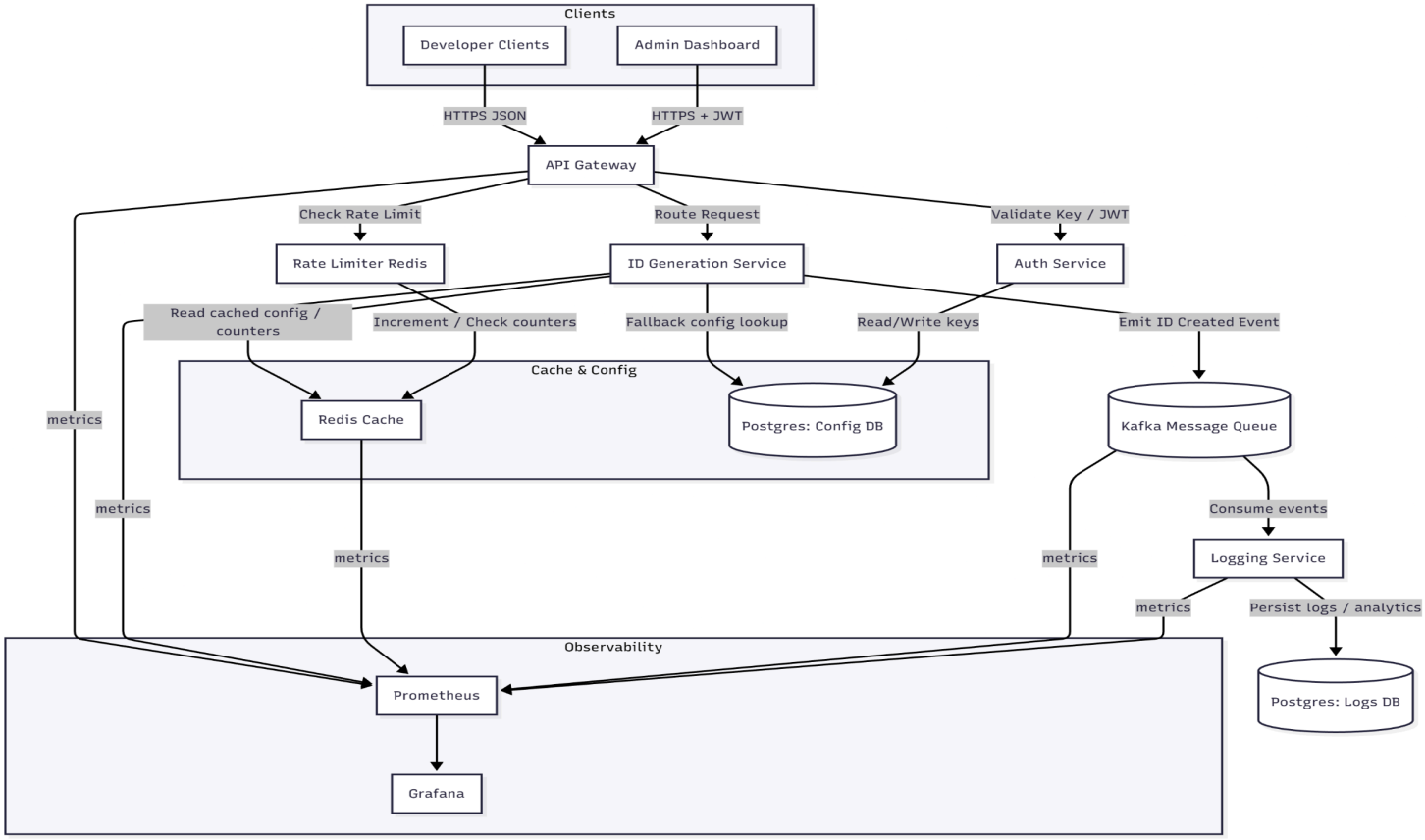
Load balancing is handled by the API Gateway, which routes traffic across multiple ID Generation Service instances to ensure high throughput and horizontal scalability. **Redis** serves as the caching layer for rate-limiting counters, configuration values, and temporary sequence metadata, chosen for its speed and atomic operations; Memcached was excluded due to limited capabilities. For asynchronous logging, the system uses **Apache Kafka**, which offers massive throughput, durability, and fault tolerance—far better suited than RabbitMQ for high-volume event streams.

6.4 Security: Authentication, Encryption, & Role Access

Security relies on **API Keys** for tenant identification and **JWT** for admin authentication. All communication is **encrypted** over TLS 1.3, with **role-based access restricting** (RBAC) clients to ID generation and admins to configuration management. Internal services authenticate through mTLS or private networking. **Rate limiting** is enforced via **Redis token buckets**, and sensitive credentials are protected using hashed storage and external secret managers such as Vault or

AWS Secrets Manager.

Communication Diagram



The communication diagram illustrates how requests move through the system from clients to internal services. Developer Clients and the Admin Dashboard interact with the API Gateway via secure HTTPS using JSON payloads (JWT for admins). The gateway handles routing, authentication through the Auth Service, and rate limiting via Redis before forwarding valid requests to the ID Generation Service. The ID Generator uses cached counters and configs from Redis and performs fallback lookups in Postgres when needed. After producing an ID, it asynchronously emits an event to Kafka, which is consumed by the Logging Service and stored in the Logs Database. Meanwhile, each component exports operational metrics to Prometheus for real-time monitoring, which Grafana visualizes for dashboard users. This design ensures low latency for ID generation, reliable logging through asynchronous queues, strong security via TLS + JWT + API Keys, and scalable observability for millions of requests per minute.

Phase 7: Design Evaluation

Microservices vs Monolith

- We use a microservices architecture to scale each component independently and prevent failures from cascading across the system. The ID Generation Service can scale horizontally for high throughput, while slower components like logging and configuration remain isolated. This structure also allows the use of optimal languages per service and enables safe, independent deployments.
- A monolithic design was rejected because it forces all modules to scale and deploy together, creating single points of failure and higher operational risk. A bug in logging or configuration could bring down ID generation, and scaling for peak loads would be unnecessarily expensive and inefficient.

REST API vs WebSockets

- REST fits the ID generation workflow because it is a simple, stateless request/response action. It integrates cleanly with rate-limiting and authentication layers and ensures minimal latency. WebSockets are unnecessary because ID generation is not a continuous stream; maintaining long-lived connections would add operational overhead without improving performance.

PostgreSQL vs NoSQL

- PostgreSQL is selected because the system requires strict consistency, relational integrity, and powerful analytical querying for the admin dashboard. SQL schemas prevent malformed configurations and enforce referential integrity across tenants, namespaces, and format rules. PostgreSQL also provides native table partitioning, B-tree indexing, transactional safety (ACID), and mature replication. These features allow us to ingest millions of log entries while still supporting complex analytical queries.
- NoSQL excels at raw write throughput but sacrifices strong consistency and complex querying. Our system requires relational joins, strong referential guarantees, and structured configurations—making SQL the more appropriate choice.

Redis vs Memcached

- Redis is ideal because it supports atomic counters, in-memory caching, and persistence—critical for both rate limiting and sequence management in ID generation. Memcached's lack of persistence and limited operations make it unsuitable for maintaining state-critical counters or cached configuration data.

Kafka vs RabbitMQ

- Kafka is optimized for high-throughput log ingestion and event streaming. Since every ID creation event must be logged without slowing down the generator, Kafka's append-only, distributed log architecture is ideal. RabbitMQ is better for task queues and smaller messaging workloads but cannot match Kafka's horizontal scalability for log streams.

Validation and Testing Strategy

- The system is evaluated through scenarios and failure-mode reasoning to ensure each component behaves correctly under expected and extreme conditions. We verify that the API Gateway properly enforces authentication and rate limits under high concurrency, that ID Generation nodes maintain uniqueness guarantees even when scaled horizontally, and that Kafka preserves log durability when downstream services fail. We would also test the reliability of our ID configuration system and schema integrity by implementing mock requests to ensure proper functionality.

Phase 8: System Reliability, Security & Scalability

8.1 - Reliability Strategy

- Each **microservice** is independently deployable and recoverable, ensuring that failures in logging, configuration, or metrics never interrupt ID generation. **Kafka** adds durable buffering so that logs are never lost, even during database outages. **Redis** replication and **Postgres** high-availability setups ensure continuous operation despite node failures, while health checks and rolling restarts in the gateway maintain uninterrupted traffic flow.
- Our use of **stateless microservices** enables **horizontal redundancy**: if a service node crashes, a new instance can immediately replace it. To further reinforce this, we use **partitioned logs** to protect the database from overload and deploy an API Gateway to eliminate single points of failure.

8.2 - Scalability Strategy

- **Scalability** is driven by horizontal expansion across stateless ID Generation nodes and by partitioned log storage. The **gateway** and **load balancer distribute** incoming requests across as many ID workers as needed. Redis handles rapid rate-limit checks and configuration caching, preventing the database from becoming a bottleneck. **Kafka** splits up write-heavy logging workloads, allowing independent scaling of ingestion and storage. **Time-based partitioning** in **Postgres** ensures consistent performance even as log volumes grow into billions of records.

8.3 - Security Design

- **Security** is enforced through layered protections, including **API keys** for tenant identification, **JWT** for administrator access, and **TLS 1.3** for encrypted communication. The gateway verifies identity and enforces access control, ensuring clients can generate IDs while administrators

manage configurations. **Rate limiting** mitigates abuse and DDoS-like traffic, while secure hashing and vault-backed secret storage protect credentials.

8.4 - Observability & Monitoring

- System health is maintained by tracking metrics and logs. **Prometheus** collects service-level metrics such as latency, throughput, cache hit rates, queue lag, and error percentages, while **Grafana** visualizes historical trends for capacity planning. **Structured logs** from **Kafka** provide an audit trail of **ID operations**, and alerting rules signal unusual conditions such as **rate-limit spikes**, degraded node performance, or database partition stalls. This monitoring system allows us to stay on top of operational issues through rapid detection and diagnosis.

8.5 - System Estimations

Traffic & Throughput

- We assume 10,000 active clients, each generating up to 100 IDs per second, resulting in a peak load of ~1 million requests per second (1M QPS). Each request-response averages ~350 bytes, producing approximately 350 MB/s of network traffic, which is easily handled by modern API gateways and cloud load balancers.

Service Capacity

- Each stateless ID Generation instance is estimated to handle ~50,000 QPS. Supporting peak traffic requires ~20 nodes, with additional instances added for redundancy, bringing the total to ~30 nodes to ensure availability during failures or traffic spikes.

Concurrency

- With sub-10 ms response times, the system supports ~10,000 concurrent requests, enabled by stateless services and horizontal scaling behind the load balancer.

Storage & Data Growth

- With sub-10 ms response times, the system supports ~10,000 concurrent requests, enabled by stateless services and horizontal scaling behind the load balancer.

Cache & Database Capacity

- Redis stores rate limits, configs, and sequence metadata, requiring <100 MB for 10,000 tenants; a small Redis cluster (2–3 GB RAM) is sufficient. PostgreSQL stores structured metadata (tenants, keys, namespaces) and remains <10 GB, while high-volume logs are isolated and partitioned for performance.