Data Download Duplication Alert System (DDAS) – HLD Document

Version: 1.1
Date: 02.02.25
Authors: Sayantika Ghosh

Table of Contents

---

1. Executive Summary

The Data Download Duplication Alert System (DDAS) is engineered to efficiently process a high volume of download requests and promptly detect duplicate downloads. With an expected scale of thousands to millions of requests per day, the system emphasizes low latency, high throughput, robust duplicate detection, and strong security measures. The design leverages multi-layer duplicate detection (Bloom filters, in-memory caches, and persistent database

checks), dynamic rate limiting, and comprehensive monitoring and alerting. This document outlines the complete high-level architecture, detailed component specifications, fault tolerance, disaster recovery strategies, and testing/security plans, ensuring that DDAS meets both current needs and future scalability demands.

---

2. Introduction

Background

Organizations offering downloadable content—such as datasets, reports, or software—face challenges when duplicate downloads occur. Duplicate events can indicate abuse, accidental re-downloads, or network issues. DDAS aims to detect and alert administrators when duplicates exceed configurable thresholds, thereby safeguarding system resources and improving security.

Objectives

- **Prevent Abuse:** Detect and flag duplicate download attempts.
- **Ensure Performance:** Handle high volumes with low latency and high throughput.
- **Provide Real-Time Monitoring:** Offer comprehensive dashboards and alerts for immediate administrative action.
- **Guarantee Security and Compliance:** Enforce robust authentication, encryption, and data privacy standards.

---

3. Requirements

3.1 Functional Requirements

Priority	Requirement	Acceptance Criteria
Must have	**User Authentication & Authorization:** Secure login using JWT/OAuth2; role-based access control.	Successful authentication with valid JWT tokens; unauthorized access blocked.
Must have	**Download Processing:** Generate a unique hash per download request; detect duplicates using Bloom filter, in-memory cache, and database verification.	Duplicate requests are flagged when exceeding the threshold (e.g., 3 downloads per 10 minutes).
Must have	**Rate Limiting:** Enforce limits per user/IP (e.g., 100 requests/minute).	API returns HTTP 429 when limits are exceeded.

Should have	**Alerting:** Generate alerts via email, Slack, PagerDuty for duplicate events.	Alerts are dispatched and visible on the admin dashboard.
Should have	**Data Persistence & Retention:** Log all download events with metadata; support a sliding window (30–90 days) with archiving.	Historical data is queryable and archived properly.
Could have	**Dynamic Configuration:** Admins can update duplicate thresholds and rate limits via an API.	Changes reflect in real-time without redeployment.
Could have	**Monitoring & Reporting:** Dashboards for system metrics and duplicate detection analytics.	Metrics such as CPU load, request latency, and duplicate rates are available.

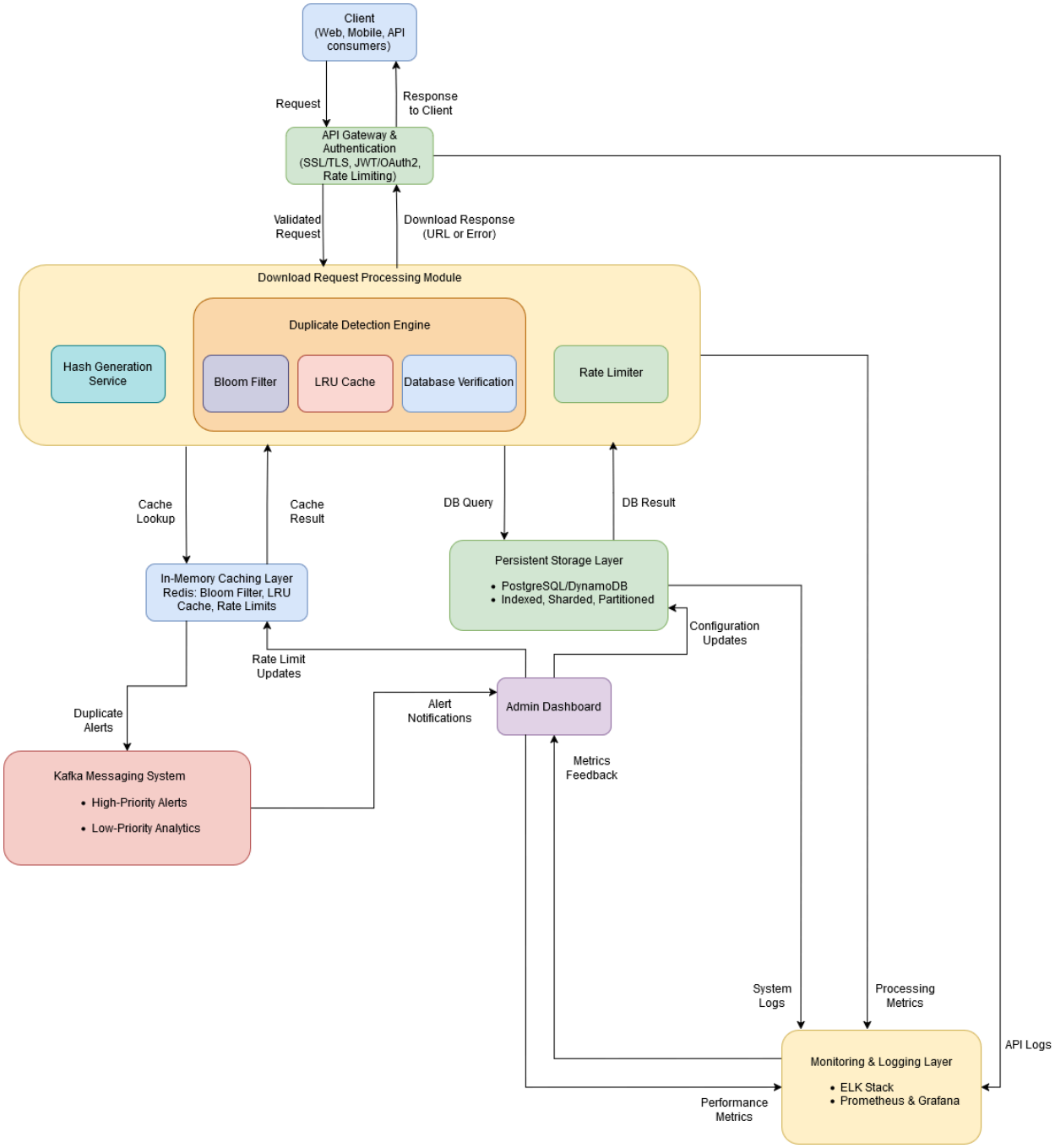
3.2 Non-Functional Requirements

Priority	Requirement	Acceptance Criteria
Must have	**Performance:** Handle ~1,157 req/s average, up to ~5,800 req/s peak with sub-100ms latency.	Stress tests meet target throughput and latency under load.
Must have	**Scalability:** Horizontal and vertical scaling capabilities.	System scales without degradation; additional nodes can be added seamlessly.
Must have	**Reliability & Fault Tolerance:** Ensure high availability (99.9% uptime) with replication and failover.	Failover tests simulate node loss without service interruption.
Must have	**Security:** Encrypted communications (TLS), robust input validation, regular security audits.	No critical vulnerabilities; regular penetration tests pass.
Should have	**Maintainability:** Modular design with clear documentation and dynamic configuration.	Code and configuration changes are isolated; documentation is complete and current.
Could have	**Data Privacy:** Data masking and access auditing to comply with regulations like GDPR.	Sensitive data is masked; audit logs are maintained and reviewed.

---

4. High-Level Architecture Overview

Overview Diagram



(Figure 1: High-Level Architecture Diagram)

- **Client Layer:** Endpoints for web, mobile, and external API consumers.

- **API Gateway & Authentication:** Routes requests; handles SSL/TLS termination; integrates with a dedicated documentation server (Swagger/OpenAPI).
- **Download Request Processing Module:**
    - **Hash Generation:** Uses MurmurHash/xxHash for performance; SHA-256 for security-sensitive cases.
    - **Duplicate Detection Engine:** Combines Bloom filter, LRU cache, and database verification.
    - **Rate Limiter:** Applies token bucket/leaky bucket algorithms.
- **Persistent Storage & Data Retention:**
    - Operational database (PostgreSQL or DynamoDB) with a detailed schema, indexing, and sharding.
    - Time-windowed sliding logs with TTL and data compression.
- **In-Memory Caching & Messaging:**
    - Redis Cluster (standalone/clustered with AOF [appendfsync everysec] and daily RDB snapshots).
    - Kafka for asynchronous task processing with multiple topics.
- **Monitoring, Logging & Alerting:**
    - Prometheus, Grafana, and ELK for system metrics and logs.
- **Administration & Configuration:**
    - Admin Dashboard and dynamic Configuration API.

---

5. Detailed Component Breakdown

5.1 API Gateway & Authentication

- **Technology:** Consider using Kong, Nginx, or AWS API Gateway.
- **Function:**
    - Routes requests, enforces SSL/TLS, applies global rate limiting.
    - Manages authentication via JWT/OAuth2.
    - Hosts interactive API documentation on a dedicated server.
- **Acceptance Criteria:**
    - Secure access with valid tokens; documentation is accessible and updated.

5.2 Download Request Processing Module

- **Hash Generation Service:**
    - **Algorithm Choice:** Use MurmurHash or xxHash for general use; SHA-256 for security-sensitive datasets.
    - **Rationale:** Performance trade-off—fast, low-collision non-cryptographic hash versus cryptographic security.
- **Duplicate Detection Engine:**
    - **Bloom Filter:**

        - Configured in Redis for ~1% false-positive rate.
        - **Sizing Formula:** m≈−n*lnp/((ln2)^2) where n is the expected number of elements in the time window and p=0.01
        - **Update Frequency:** Refreshed every hour or when significant new datasets are added.
        - **Metrics:** Tracks false positive count and rate.
    - **LRU Cache:**
        - Fixed size based on the sliding window with TTL-based eviction.
        - Invalidation mechanisms for when user download limits reset.
    - **Database Verification:**
        - Exact duplicate check enforcing thresholds (e.g., max 3 downloads/10 minutes per user).
- **Rate Limiter:**
    - Uses token bucket or leaky bucket algorithms.
    - **Configuration:** Bucket size (e.g., 100 requests) and refill rate (e.g., 100 tokens/minute), configurable via an API.

5.3 Persistent Storage & Data Retention

- **Technology:**
    - **Workload Analysis:**
        - PostgreSQL if strong relational integrity and complex queries are required.
        - DynamoDB for horizontal scalability and high write throughput.
- **Schema:**
    - Fields: user_id, dataset_id, timestamp, status, file size, file version, download source (IP, user agent), created_at, updated_at.
- **Indexing & Sharding:**
    - Composite indexes on user_id, dataset_id, and timestamp.
    - Sharding strategy based on user_id or dataset_id with documented re-sharding.
- **Data Retention:**
    - Active data for 30–90 days; older data archived (stored in cloud cold storage) with compression (gzip/Snappy).

5.4 In-Memory Caching & Messaging

- **Redis Cluster:**
    - Deployment Model: Clustered Redis.
    - **Persistence:**
        - AOF enabled (appendfsync set to everysec for near-real-time recovery) and daily RDB snapshots.
- **Kafka Messaging System:**
    - **Topics:**
        - High-priority topics for alerts and critical logs.

        - Lower-priority topics for analytics.
    - **Consumer Groups:**
        - Enable parallel processing.
    - **Compression:**
        - Use Snappy to reduce network overhead.

5.5 Monitoring, Logging & Alerting

- **Monitoring Tools:**
    - Prometheus and Grafana to track metrics such as CPU utilization, memory usage, disk I/O, request latency, error rates, duplicate download rate, and false positive rate.
    - **Alerting Thresholds:**
        - Example: Alert if the duplicate download rate exceeds 5%.
- **Logging:**
    - ELK stack collects detailed logs for all events.
- **Alerting Module:**
    - Aggregates and escalates alerts via email, Slack, PagerDuty.

5.6 Administration & Configuration

- **Admin Dashboard:**
    - Real-time views of system health, duplicate alerts, download history.
- **Configuration API:**
    - Allows dynamic updates to thresholds, rate limits, and retention policies.
- **Access Auditing & Data Masking:**
    - Logs administrative actions; sensitive data is masked in logs and dashboards.

---

6. Fault Tolerance and Disaster Recovery

- **Database Replication & Failover:**
    - Implement replication and automatic failover (RTO: 1 hour, RPO: 15 minutes, for example).
- **Redis High Availability:**
    - Use a clustered Redis deployment with AOF/RDB persistence.
- **Multi-Region Deployment:**
    - Optionally deploy across multiple regions for geographic redundancy.
- **Backup & Restore:**
    - Daily automated backups stored in secure cloud storage; documented restore procedures ensure recovery within defined RTO.

---

7. Testing and Security Strategy

- **Performance and Load Testing:**
  - Tools: JMeter or Gatling.
  - Verify throughput (targeting thousands to millions of downloads per day) and latency targets.
- **Automated Testing Suite:**
  - Unit Tests: JUnit.
  - End-to-End Tests: Cypress.
  - Integration Tests.
- **Security Testing:**
  - Penetration Testing, Vulnerability Scanning, Static Code Analysis.
- **Data Encryption:**
  - Use TLS/SSL for data in transit and AES-256 for data at rest.
- **Input Validation:**
  - Robust validation across all endpoints.

---

8. Additional Considerations

- **Documentation Hosting:**
  - API documentation is hosted on a dedicated server integrated with the API gateway.
- **Hash Algorithm Rationale:**
  - Use MurmurHash/xxHash for performance; fallback to SHA-256 when required.
- **Bloom Filter Sizing:**
  - Detailed calculations using the formula provided; adjustments made based on empirical false positive rates.
- **Alert Aggregation:**
  - Group alerts based on time windows and severity; define escalation policies to reduce alert fatigue.
- **Rate Limiting Granularity:**
  - Apply limits per user, per IP, and per dataset with dynamic adjustments based on system load.
- **Data Privacy:**
  - Implement data masking and access auditing to meet regulatory requirements.

---

9. Conclusion and Next Steps

Conclusion

This document presents a comprehensive, high-level design for the Data Download Duplication Alert System (DDAS). The design addresses key functional and non-functional requirements, with detailed considerations for duplicate detection, rate limiting, caching, persistent storage, and fault tolerance. Security, testing, and dynamic configuration strategies are also integrated, ensuring that the system is robust, scalable, and maintainable.

Next Steps

- **Develop a Prototype:**
  Build a prototype of the core components to validate design assumptions.
- **Detailed Design & Implementation:**
  Create detailed design documents for each module.
- **Performance & Security Testing:**
  Conduct performance, load, and security tests.
- **Stakeholder Review:**
  Review the design with stakeholders and incorporate feedback.
- **Documentation Finalization:**
  Complete and publish the interactive API documentation.

---

10. Appendices

- **Appendix A:** Detailed Calculations (e.g., Bloom filter sizing, rate limiting parameters)
- **Appendix B:** Glossary of Terms
- **Appendix C:** References (e.g., research papers, design pattern documentation)
- **Appendix D:** Deployment and Backup Procedures

Data Download Duplication Alert System (DDAS) - LLD Document

---

1. Introduction and Overview

1.1 Purpose

This document provides a detailed implementation blueprint for the Data Download Duplication Alert System (DDAS). It builds on the High-Level Design (HLD) to define all core components, their interactions, and code-level details.

1.2 Scope

This LLD covers:

- Core components: Request Handling, Download Processing, Caching, Database, Rate Limiting, Kafka Integration, and Monitoring.
- Detailed class structures, database schema, and API endpoints.
- Algorithms, design patterns, and deployment considerations.
- Sequence Diagrams, Deployment Diagrams, and Class Diagrams for better visualization.

1.3 References

- **HLD Document**: DDAS High-Level Architecture.
- **Requirements Document**: Functional and Non-Functional Requirements.

---

2. Component Details

2.1 RequestHandler

- **Description**: Entry point for all client requests via the API Gateway.
- **Responsibilities**:
  - Validate and route requests to `DownloadProcessor`.
  - Enforce authentication (JWT/OAuth2) and rate limiting.
- **Interfaces**:
  - **Input**: HTTP requests from the API Gateway.
  - **Output**: Responses with status codes (200, 401, 429, 409).
- **Internal Details**:
  - **Validation**: Verify request parameters (user_id, dataset_id).
  - **Rate Limiting**: Apply global and per-user rate limits using Redis token bucket implementation.
- **Error Handling**:
  - Invalid JWT: Respond with `401 Unauthorized`.
  - Rate Limit Exceeded: Respond with `429 Too Many Requests`.
  - Missing Parameters: Respond with `400 Bad Request`.

2.2 DownloadProcessor

- **Description**: Handles the core logic for download requests.

- **Responsibilities**:
  - Generate hash for requested data.
  - Perform duplicate detection.
  - Enforce rate limits.
  - Forward valid requests for download.
- **Internal Structure**:
  - **HashService**:
    - Default: `MurmurHash` for performance.
    - Secure: `SHA-256` for compliance-critical operations.
    - Handles collisions by combining hash with a unique user_id.
  - **DuplicateChecker**:
    - **Bloom Filter Check (`checkBloomFilter(hash)`)**:
      - Queries the Redis Bloom filter named "download_bloom_filter."
      - Redis command: `BF.EXISTS download_bloom_filter hash`.
      - Configured for a 1% false positive rate using `BF.RESERVE` during initialization.

        Example Update Logic:
        MULTI
        BF.ADD download_bloom_filter hash1
        BF.ADD download_bloom_filter hash2
        EXEC

      -
    - **LRU Cache Check (`checkLRUCache(hash)`)**:
      - Key Format: `lru_cache:{dataset_id}:{hash}`.
      - Queries the Redis Hash named "recent_downloads."

      Commands:
      HSET recent_downloads {key} {timestamp} -- Add
      HGET recent_downloads {key} -- Retrieve
      HDEL recent_downloads {key} -- Evict

      -
    - **Database Check (`checkDatabase(hash, user_id, dataset_id)`)**:

      SQL Query:
      SELECT download_id, download_time, status
      FROM DatasetDownloads
      WHERE hash = ? AND user_id = ? AND dataset_id = ?;

      -

        - Uses prepared statements to prevent SQL injection.
    - **False Positive Handling**:
        - Log false positives using the monitoring system.
        - Adjust the Bloom filter false positive rate dynamically based on system feedback.

2.3 In-Memory Caching Layer (Redis)

- **Details**:
    - **Bloom Filter**:
        - Configured for ~1% false positive rate.
        - Stored as a single key per dataset or sharded across multiple keys based on dataset ID.

      Commands:
      BF.ADD download_bloom_filter new_hash

        -
    - **LRU Cache**:
        - Implemented using Redis Hash with LRU eviction.

---

3. Diagrams for Better Understanding

- **Sequence Diagram**: Show request validation, duplicate detection, and response generation.
- **Deployment Diagram**: Illustrate infrastructure components and their interactions.
- **Class Diagram**: Represent key classes and their relationships.

---

4. Glossary

- **Bloom Filter**: A probabilistic data structure that allows for efficient duplicate detection.
- **LRU Cache**: Least Recently Used cache mechanism for efficient duplicate verification.
- **Kafka**: A distributed messaging system for logging alerts and analytics.

---

5. Additional Code Snippets

```
private final ReentrantLock lock = new ReentrantLock();
public void processDownloadRequest() {
```

```
    lock.lock();
    try {
        // Access and modify shared resources here
    } finally {
        lock.unlock();
    }
}
```

---

6. Configuration Management Example

```
rate_limiter:
  user_limit: 100
  dataset_limit: 50
redis:
  bloom_filter_size: 10000000
  bloom_false_positive_rate: 0.01
```

---

7. Final Testing Considerations

1. **Unit Tests**: Validate Redis and DB interactions.
2. **Integration Tests**: Simulate end-to-end workflows.
3. **Performance Tests**: Ensure system handles 5,800 req/s peak load with sub-100ms latency.
4. **Security Tests**: Penetration testing for API endpoints.

---

8. Deployment YAML Example

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: ddas-download-processor
spec:
  replicas: 3
  template:
    spec:
      containers:
      - name: download-processor
        image: ddas/processor:latest
        resources:
          limits:
```

```
    memory: "512Mi"
    cpu: "500m"
  requests:
    memory: "256Mi"
    cpu: "250m"
```