Pastebin System Design - High Level Design (HLD)

System Architecture Overview

This document presents a comprehensive High Level Design for a Pastebin-like system capable of handling millions of paste operations per day with high availability, low latency, and robust scalability.

System Components & Architecture Layers

1. Client Layer

Components:

- Web Clients (Browser-based)
- Mobile Clients (iOS/Android Native Apps)
- Desktop Applications
- API Clients

Protocol: HTTPS (Port 443)

Characteristics:

- Multiple concurrent client connections
- Support for various content types
- Responsive UI/UX across devices

2. Content Delivery Network (CDN)

Implementation: CloudFront / Akamai / Cloudflare

Features:

- **Edge Caching:** Distributes content across global edge locations
- **Cache Hit Rate:** Target 80-90% for frequently accessed pastes
- **TTL Management:** Configurable cache expiration policies
- **DDoS Protection:** Built-in security features
- **SSL/TLS Termination:** Secure content delivery

Protocol: HTTPS

Benefits:

- Reduces latency from ~200ms to ~50ms for cached content
- Offloads 80-90% of read traffic from origin servers
- Improves user experience globally

3. Load Balancer

Implementation: AWS Application Load Balancer (ALB) / Elastic Load Balancer (ELB) / Nginx

Features:

- **Algorithm:** Round Robin or Least Connections
- **Health Checks:** Monitors application server availability
- **SSL Termination:** Decrypts HTTPS traffic
- **Session Persistence:** Sticky sessions if needed
- **Auto-scaling Integration:** Adjusts to traffic patterns

Protocol: HTTPS → HTTP/HTTPS

Configuration:

- Health check interval: 30 seconds
- Unhealthy threshold: 2 consecutive failures
- Timeout: 5 seconds

4. API Gateway

Implementation: Kong / AWS API Gateway / Apigee / Spring Cloud Gateway

Features:

Rate Limiting:

- Algorithm: Token Bucket
- Limit: 100 paste creations per hour per IP address
- Burst capacity: 20 requests
- Response: HTTP 429 (Too Many Requests) when exceeded

Authentication & Authorization:

- OAuth 2.0 / JWT tokens (for authenticated users)
- API key validation
- Role-based access control

Request Routing:

- `/api/v1/paste` → Paste Service

- `/api/v1/analytics` → Analytics Service
- `/api/v1/user` → User Service

Additional Features:

- API versioning
- Request/response transformation
- Logging and monitoring
- Circuit breaker pattern

Protocol: REST/HTTP

5. Application Server Cluster

Implementation: Java Spring Boot / Node.js / Python Flask

Architecture:

- **Stateless Design:** No session state stored locally
- **Horizontal Scaling:** Multiple instances (3+ servers)
- **Auto-scaling:** Based on CPU (>70%), Memory (>80%), or QPS metrics

Instance Configuration:

- Instance Type: t3.large or equivalent (2 vCPU, 8GB RAM)
- JVM Settings: Heap size 4GB, G1GC
- Connection Pool: 50 database connections per instance

Core Services:

Paste Service:

```
@RestController
@RequestMapping("/api/v1/paste")
public class PasteController {

    @PostMapping
    public ResponseEntity&lt;PasteResponse&gt; createPaste(@RequestBody PasteRequest requ
        // 1. Get unique key from KGS
        String pasteId = kgsClient.getKey();

        // 2. Upload content to S3
        String contentKey = s3Service.upload(pasteId, request.getContent());

        // 3. Store metadata in database
        pasteRepository.save(new Paste(pasteId, contentKey, request.getExpiration()));

        // 4. Populate cache
        cacheService.set(pasteId, request.getContent(), TTL);
```

```
        // 5. Return response
        return ResponseEntity.ok(new PasteResponse(pasteId, buildUrl(pasteId)));
    }

    @GetMapping("/{pasteId}")
    public ResponseEntity&lt;String&gt; getPaste(@PathVariable String pasteId) {
        // 1. Check cache
        String content = cacheService.get(pasteId);
        if (content != null) return ResponseEntity.ok(content);

        // 2. Query database
        Paste paste = pasteRepository.findById(pasteId);
        if (paste == null || paste.isExpired()) {
            return ResponseEntity.notFound().build();
        }

        // 3. Fetch from S3
        content = s3Service.download(paste.getContentKey());

        // 4. Update cache
        cacheService.set(pasteId, content, TTL);

        return ResponseEntity.ok(content);
    }
}
```

Protocol: REST/HTTP (Port 8080)

6. Cache Layer (Redis Cluster)

Architecture:

- **Redis Master:** Handles all write operations
- **Redis Replica 1, 2, ... N:** Handle read operations
- **Replication:** Asynchronous master-slave replication

Configuration:

- Memory: 32GB per instance
- Eviction Policy: LRU (Least Recently Used)
- Persistence: RDB snapshots + AOF logs
- Clustering: Redis Cluster with 3 master nodes, 3 replica nodes

Data Structure:

```
Key: paste:{paste_id}
Value: {
    "content": "actual paste content",
    "created_at": "2025-10-25T00:00:00Z",
    "expiration": "2025-10-26T00:00:00Z"
```

```
}
TTL: 3600 seconds (1 hour)
```

Cache Strategy:

- **Cache-Aside Pattern:** Application manages cache
- **Cache Hit Rate Target:** 80-90%
- **TTL:** 1 hour for pastes, configurable based on access patterns

Protocol: Redis Protocol (RESP) - Port 6379

Benefits:

- Reduces database load by 80-90%
- Latency: <5ms for cache hits
- Supports millions of operations per second

7. Key Generation Service (KGS)

Purpose: Generate unique, short, collision-free keys for paste URLs

Architecture:

- Dedicated microservice
- Pre-generates keys in batches
- Maintains key pool in memory

Key Generation Algorithm:

Base62 Encoding:

- Character set: A-Z, a-z, 0-9 (62 characters)
- Key length: 6 characters
- Total combinations: 62^6 ≈ 56.8 billion unique keys

Key Range Strategy:

```
Server 1: AAA000 - AAA999 (1,000 keys)
Server 2: AAB000 - AAB999 (1,000 keys)
Server 3: AAC000 - AAC999 (1,000 keys)
...
```

Implementation:

```
import random
import string

BASE62 = string.ascii_uppercase + string.ascii_lowercase + string.digits
```

```
def generate_key(length=6):
    return ''.join(random.choice(BASE62) for _ in range(length))

def generate_key_batch(count=1000000):
    keys = set()
    while len(keys) &lt; count:
        keys.add(generate_key())
    return list(keys)
```

Key Allocation Flow:

1. KGS pre-generates 10 million keys at startup
2. Stores keys in `unused_keys` table
3. Loads 100K keys into memory
4. App server requests key via gRPC call
5. KGS returns key and marks as used atomically
6. When memory pool depletes, loads next batch from database

Protocol: gRPC (Port 50051) / HTTP (Port 8081)

Performance:

- Key allocation latency: <10ms
- Throughput: 10,000+ keys/second

8. Key Database

Implementation: PostgreSQL / MySQL

Schema:

```
CREATE TABLE unused_keys (
    key_id VARCHAR(10) PRIMARY KEY,
    created_at TIMESTAMP DEFAULT NOW(),
    INDEX idx_created_at (created_at)
);

CREATE TABLE used_keys (
    key_id VARCHAR(10) PRIMARY KEY,
    used_at TIMESTAMP DEFAULT NOW(),
    paste_id VARCHAR(10),
    INDEX idx_used_at (used_at)
);
```

Operations:

Atomic Key Allocation:

```
BEGIN TRANSACTION;

SELECT key_id FROM unused_keys LIMIT 1 FOR UPDATE;

DELETE FROM unused_keys WHERE key_id = ?;

INSERT INTO used_keys (key_id, paste_id) VALUES (?, ?);

COMMIT;
```

Protocol: SQL (Port 5432 for PostgreSQL)

9. Metadata Database Cluster

Implementation: PostgreSQL with Master-Slave Replication

Architecture:

- **Master Database:** Handles all write operations
- **Read Replica 1, 2, ... N:** Handle read operations (distributed read load)
- **Replication:** Asynchronous streaming replication
- **Replication Lag:** Target <100ms

Schema:

```
CREATE TABLE paste (
    paste_id VARCHAR(10) PRIMARY KEY,
    content_key VARCHAR(255) NOT NULL,
    user_id VARCHAR(36),
    created_at TIMESTAMP DEFAULT NOW(),
    expiration_date TIMESTAMP,
    size_bytes INT,
    access_count INT DEFAULT 0,
    INDEX idx_expiration (expiration_date),
    INDEX idx_user_id (user_id),
    INDEX idx_created_at (created_at)
);

CREATE TABLE user (
    user_id VARCHAR(36) PRIMARY KEY,
    username VARCHAR(50) UNIQUE,
    email VARCHAR(100),
    password_hash VARCHAR(255),
    created_at TIMESTAMP DEFAULT NOW()
);
```

Connection Pooling:

- Pool size: 50 connections per app server
- Max idle time: 10 minutes

- Connection timeout: 30 seconds

Read/Write Splitting:

```
@Transactional(readOnly = false)
public void writePaste(Paste paste) {
    // Routes to master database
    pasteRepository.save(paste);
}

@Transactional(readOnly = true)
public Paste readPaste(String pasteId) {
    // Routes to read replica
    return pasteRepository.findById(pasteId);
}
```

Protocol: SQL/JDBC (Port 5432)

Database Configuration:

- Instance Type: db.r5.2xlarge (8 vCPU, 64GB RAM)
- Storage: 1TB SSD with auto-scaling
- Backup: Daily automated backups, 7-day retention
- Multi-AZ deployment for high availability

10. Object Storage (Amazon S3)

Architecture:

- **Multiple S3 Buckets:** For redundancy and partitioning
- **Storage Class:** S3 Standard (hot data) → S3 Infrequent Access (after 30 days)
- **Durability:** 99.999999999% (11 nines)
- **Availability:** 99.99%

Bucket Structure:

```
pastebin-content-us-east-1/
├── a/
│   ├── ab/
│   │   ├── abc123
│   │   └── abc456
│   └── ac/
├── b/
└── ...
```

Storage Strategy:

- **Key Pattern:** `{first_char}/{first_two_chars}/{paste_id}`

- **Content Type:** text/plain, application/json, etc.
- **Encryption:** AES-256 server-side encryption
- **Versioning:** Disabled (pastes are immutable)

Lifecycle Policies:

```
{
  "Rules": [
    {
      "Id": "TransitionToIA",
      "Status": "Enabled",
      "Transitions": [
        {
          "Days": 30,
          "StorageClass": "STANDARD_IA"
        }
      ]
    },
    {
      "Id": "DeleteExpired",
      "Status": "Enabled",
      "Expiration": {
        "Days": 365
      }
    }
  ]
}
```

Operations:

Upload:

```
public String uploadContent(String pasteId, String content) {
    String key = buildS3Key(pasteId);
    PutObjectRequest request = PutObjectRequest.builder()
        .bucket(BUCKET_NAME)
        .key(key)
        .contentType("text/plain")
        .build();

    s3Client.putObject(request, RequestBody.fromString(content));
    return key;
}
```

Download:

```
public String downloadContent(String contentKey) {
    GetObjectRequest request = GetObjectRequest.builder()
        .bucket(BUCKET_NAME)
        .key(contentKey)
        .build();
```

```
    return s3Client.getObjectAsBytes(request).asUtf8String();
}
```

Protocol: S3 API / HTTPS (Port 443)

Performance:

- Upload latency: 50-100ms
- Download latency: 50-150ms
- Throughput: 5,500 GET requests/second per prefix

11. Cleanup Service

Purpose: Remove expired pastes to free storage and maintain system hygiene

Architecture:

- **Cron Job:** Scheduled task running hourly/daily
- **Batch Processing:** Processes 10,000 pastes per batch
- **Distributed:** Multiple workers for parallel processing

Implementation:

```
import schedule
import time
from datetime import datetime

class CleanupService:
    def __init__(self, db, s3_client, kgs_client):
        self.db = db
        self.s3 = s3_client
        self.kgs = kgs_client

    def cleanup_expired_pastes(self):
        print(f"[{datetime.now()}] Starting cleanup job...")

        # Step 1: Query expired pastes
        expired_pastes = self.db.execute("""
            SELECT paste_id, content_key
            FROM paste
            WHERE expiration_date &lt; NOW()
            LIMIT 10000
        """)

        print(f"Found {len(expired_pastes)} expired pastes")

        for paste in expired_pastes:
            try:
                # Step 2: Delete from S3
                self.s3.delete_object(
                    Bucket='pastebin-content',
```

```
                    Key=paste['content_key']
                )

                # Step 3: Delete from database
                self.db.execute("""
                    DELETE FROM paste WHERE paste_id = ?
                """, paste['paste_id'])

                # Step 4: Return key to KGS
                self.kgs.return_key(paste['paste_id'])

            except Exception as e:
                print(f"Error cleaning paste {paste['paste_id']}: {e}")

        print(f"[{datetime.now()}] Cleanup job completed")

# Schedule cleanup job
schedule.every().hour.do(cleanup_service.cleanup_expired_pastes)

while True:
    schedule.run_pending()
    time.sleep(60)
```

Alternative Strategy: Lazy Deletion

```
@GetMapping("/{pasteId}")
public ResponseEntity&lt;String&gt; getPaste(@PathVariable String pasteId) {
    Paste paste = pasteRepository.findById(pasteId);

    if (paste == null) {
        return ResponseEntity.notFound().build();
    }

    // Check if expired
    if (paste.getExpirationDate().isBefore(LocalDateTime.now())) {
        // Trigger async deletion
        cleanupService.deleteAsync(pasteId);
        return ResponseEntity.notFound().build();
    }

    return ResponseEntity.ok(fetchContent(paste));
}
```

Cleanup Metrics:

- Pastes deleted per day: ~1 million (assuming 1-day expiration average)
- Storage freed per day: ~10GB
- Execution time: ~10-15 minutes per batch

12. Monitoring & Observability

Stack:

- **Prometheus:** Metrics collection and storage
- **Grafana:** Visualization and dashboards
- **ELK Stack:** Log aggregation (Elasticsearch, Logstash, Kibana)
- **Jaeger:** Distributed tracing

Key Metrics:

Application Metrics:

- **Request Rate (QPS):** Total requests/second, per endpoint
- **Latency:** p50, p95, p99 percentiles for all endpoints
- **Error Rate:** 4xx, 5xx response rates
- **Throughput:** Requests processed per second

Cache Metrics:

- **Hit Rate:** (Cache hits / Total requests) * 100
- **Miss Rate:** (Cache misses / Total requests) * 100
- **Eviction Rate:** Keys evicted per second
- **Memory Usage:** Current vs maximum

Database Metrics:

- **Query Latency:** Average, p95, p99
- **Connection Pool:** Active connections, idle connections
- **Replication Lag:** Master to replica delay
- **Slow Queries:** Queries taking >1 second

Storage Metrics:

- **S3 Usage:** Total storage consumed
- **Growth Rate:** GB added per day
- **Request Count:** GET, PUT, DELETE operations
- **Error Rate:** Failed S3 operations

System Metrics:

- **CPU Utilization:** Per instance
- **Memory Usage:** Per instance
- **Network Bandwidth:** Inbound/outbound traffic
- **Disk I/O:** Read/write operations

Dashboards:

1. **System Overview Dashboard:**
    - Total QPS, Latency, Error Rate
    - Active users, Paste creation rate
    - Cache hit rate, Database connections
    - S3 storage usage
2. **Performance Dashboard:**
    - Latency heatmaps per endpoint
    - Request distribution across instances
    - Database query performance
    - Cache performance trends
3. **Alerts:**
    - Error rate >1% for 5 minutes
    - p99 latency >500ms for 5 minutes
    - Cache hit rate <70% for 10 minutes
    - Database replication lag >5 seconds
    - CPU utilization >85% for 10 minutes

Logging Strategy:

```
@Slf4j
@RestController
public class PasteController {

    @PostMapping("/api/v1/paste")
    public ResponseEntity&lt;PasteResponse&gt; createPaste(@RequestBody PasteRequest requ
        String traceId = UUID.randomUUID().toString();
        MDC.put("traceId", traceId);

        log.info("Creating paste - size: {} bytes, expiration: {}",
                 request.getContent().length(), request.getExpiration());

        try {
            PasteResponse response = pasteService.create(request);
            log.info("Paste created successfully - pasteId: {}", response.getPasteId());
            return ResponseEntity.ok(response);
        } catch (Exception e) {
            log.error("Error creating paste", e);
            return ResponseEntity.status(500).build();
        } finally {
            MDC.clear();
        }
    }
}
```

Detailed Data Flows

Write Flow (Creating a Paste)

Step-by-Step Process:

Step 1: Client Request

- User enters paste content in browser/mobile app
- Selects expiration time (1h, 1d, 1w, 1m, never)
- Clicks "Create Paste"
- Client sends `POST /api/v1/paste` request

Step 2: CDN → Load Balancer

- Request passes through CDN (no caching for POST requests)
- CDN forwards to Load Balancer
- **Protocol:** HTTPS
- **Latency:** ~10-20ms

Step 3: Load Balancer → API Gateway

- Load Balancer selects healthy app server using Round Robin
- Routes request to API Gateway
- **Protocol:** HTTPS
- **Latency:** ~5-10ms

Step 4: API Gateway - Rate Limiting

- Checks rate limit for client IP
- Token Bucket algorithm:
  - Bucket capacity: 100 tokens
  - Refill rate: 100 tokens per hour
  - Request consumes 1 token
- If limit exceeded, returns HTTP 429
- **Latency:** ~2-5ms

Step 5: API Gateway → App Server

- Routes to available application server
- Request: `POST /api/v1/paste`
- Body: `{ "content": "...", "expiration": "1d" }`
- **Protocol:** REST/HTTP
- **Latency:** ~5ms

Step 6: App Server → KGS (Key Generation)

- App server calls KGS via gRPC: `GenerateKey()`
- KGS returns unique 6-character Base62 key (e.g., "aB3xY9")
- Key marked as used in Key Database
- **Protocol:** gRPC
- **Latency:** ~5-10ms

Step 7: App Server → S3 (Content Upload)

- Upload paste content to S3
- Key: `a/aB/aB3xY9`
- Metadata: content-type, size, timestamp
- **Protocol:** S3 API / HTTPS
- **Latency:** ~50-100ms

Step 8: App Server → Database Master (Metadata Storage)

- Insert metadata record:

```
INSERT INTO paste (paste_id, content_key, expiration_date, created_at, size_bytes)
VALUES ('aB3xY9', 'a/aB/aB3xY9', '2025-10-26 00:00:00', NOW(), 1024);
```

- **Protocol:** SQL/JDBC
- **Latency:** ~10-20ms

Step 9: App Server → Redis Cache (Cache Warmup)

- Store paste in cache for fast reads

```
SET paste:aB3xY9 "{content}" EX 3600
```

- **Protocol:** Redis Protocol
- **Latency:** ~2-5ms

Step 10: App Server → Client (Response)

- Return success response:

```
{
  "paste_id": "aB3xY9",
  "url": "https://pastebin.com/aB3xY9",
  "expiration_date": "2025-10-26T00:00:00Z",
  "created_at": "2025-10-25T00:00:00Z"
}
```

- **Total End-to-End Latency:** ~150-200ms

Read Flow (Retrieving a Paste)

Step-by-Step Process:

Step 1: Client Request

- User accesses `https://pastebin.com/aB3xY9`
- Browser sends `GET` request

Step 2: CDN Cache Check

- CDN checks edge cache for paste
- **Cache Hit:** Returns content directly (Latency: ~50ms) ✓
- **Cache Miss:** Forwards to Load Balancer (Continue to Step 3)

Step 3: Load Balancer → API Gateway

- Load Balancer routes to API Gateway
- **Protocol:** HTTPS
- **Latency:** ~10ms

Step 4: API Gateway → App Server

- Routes `GET /api/v1/paste/aB3xY9`
- **Protocol:** REST/HTTP
- **Latency:** ~5ms

Step 5: App Server → Redis Cache

- Check Redis for cached content

```
GET paste:aB3xY9
```

- **Cache Hit Rate:** 80-90%
- **Protocol:** Redis Protocol
- **Latency:** ~2-5ms

Cache Hit Path (Fast Path):

- Return content immediately
- **Total Latency:** ~20-30ms ✓

Cache Miss Path (Slow Path):

Step 6a: App Server → Database Read Replica

- Query metadata:

```
SELECT paste_id, content_key, expiration_date, created_at
FROM paste
```

```
WHERE paste_id = 'aB3xY9';
```

- Check if paste exists and not expired
- **Protocol:** SQL/JDBC
- **Latency:** ~10-20ms

Step 6b: App Server → S3 (Content Fetch)

- Download paste content from S3

```
GetObjectRequest request = GetObjectRequest.builder()
    .bucket("pastebin-content")
    .key("a/aB/aB3xY9")
    .build();
```

- **Protocol:** S3 API / HTTPS
- **Latency:** ~50-100ms

Step 7: App Server → Redis Cache (Cache Update)

- Store fetched content in cache

```
SET paste:aB3xY9 "{content}" EX 3600
```

- **Latency:** ~2-5ms

Step 8: App Server → CDN (Cache Update)

- Set cache-control headers

```
Cache-Control: public, max-age=3600
ETag: "abc123xyz"
```

- CDN caches response

Step 9: App Server → Client (Response)

- Return paste content
- **Total Latency (Cache Miss):** ~100-150ms

Comparison:

- **CDN Cache Hit:** ~50ms
- **Redis Cache Hit:** ~20-30ms
- **Full Miss (DB + S3):** ~100-150ms

Protocol Summary

Complete Protocol Stack

Layer	Source	Destination	Protocol	Port	Purpose
Client	User	CDN	HTTPS	443	Secure client communication
CDN	CDN	Load Balancer	HTTPS	443	Edge to origin
Load Balancer	LB	API Gateway	HTTP/HTTPS	80/443	Traffic distribution
API Gateway	API GW	App Server	REST/HTTP	8080	API routing
Application	App	Redis	Redis Protocol (RESP)	6379	Cache operations
Application	App	KGS	gRPC / HTTP	50051/8081	Key generation
Application	App	Database	SQL/JDBC	5432	Metadata storage
Application	App	S3	S3 API / HTTPS	443	Content storage
Database	Master	Replica	PostgreSQL Replication	5432	Data replication
Cache	Redis Master	Redis Replica	Redis Replication	6379	Cache replication

Scalability Strategies

Horizontal Scaling

Application Servers:

- Add more instances behind load balancer
- Auto-scaling policy:
  - Scale out: CPU >70% or QPS >500/instance
  - Scale in: CPU <30% or QPS <100/instance
- Minimum instances: 3
- Maximum instances: 50

Database Read Replicas:

- Add read replicas to distribute read load
- Each replica can handle ~10,000 QPS
- Automatic failover with promotion

Redis Cluster:

- Redis Cluster with 6 nodes (3 master + 3 replica)
- Each master handles ~50,000 ops/sec
- Hash slot distribution for data partitioning

S3 Buckets:

- Partition by prefix for high throughput
- Use multiple buckets for different regions

Vertical Scaling

- Upgrade instance types for database and cache
- Increase memory for better caching
- Faster storage (NVMe SSDs) for database

Database Sharding

Sharding Strategy:

- Shard by `paste_id` using consistent hashing
- Number of shards: 10 (initially)
- Shard key: `hash(paste_id) % 10`

Shard Distribution:

```
Shard 0: paste_ids with hash % 10 = 0
Shard 1: paste_ids with hash % 10 = 1
...
Shard 9: paste_ids with hash % 10 = 9
```

Benefits:

- Distributes load across multiple database instances
- Each shard handles 10% of total data
- Linear scalability with additional shards

Caching Strategy

Multi-Layer Caching:

1. **CDN Cache:** 80-90% hit rate for popular pastes
2. **Redis Cache:** 80% hit rate for remaining 10-20%
3. **Database:** Only 2-4% of requests hit database

Cache Invalidation:

- TTL-based expiration

- Active invalidation on delete operations
- Lazy invalidation on expired pastes

Reliability & High Availability

Fault Tolerance

Load Balancer:

- Health checks every 30 seconds
- Remove unhealthy instances automatically
- Multi-AZ deployment

Application Servers:

- Stateless design allows instant replacement
- Auto-scaling replaces failed instances
- Circuit breaker prevents cascading failures

Database:

- Master-slave replication with automatic failover
- Multi-AZ deployment (master and replica in different AZs)
- Point-in-time recovery (PITR)
- Daily automated backups

Redis:

- Master-slave replication
- Redis Sentinel for automatic failover
- Persistence: RDB snapshots + AOF logs

S3:

- 99.999999999% durability (11 nines)
- Automatic replication across multiple AZs
- Versioning for accidental deletions

Disaster Recovery

Backup Strategy:

- Database: Daily full backups, 7-day retention
- S3: Cross-region replication to disaster recovery region
- Redis: RDB snapshots every hour

Recovery Time Objective (RTO): <1 hour
Recovery Point Objective (RPO): <15 minutes

Security Considerations

Data Protection

Encryption at Rest:

- S3: AES-256 server-side encryption
- Database: Transparent Data Encryption (TDE)
- Redis: Encryption at rest (if supported)

Encryption in Transit:

- HTTPS/TLS 1.3 for all client communication
- TLS for inter-service communication
- VPN for database connections

Access Control

Network Security:

- VPC with private subnets for databases
- Security groups with least privilege
- WAF (Web Application Firewall) for DDoS protection

Authentication:

- API key authentication for programmatic access
- OAuth 2.0 / JWT for user authentication
- Role-based access control (RBAC)

Rate Limiting

Token Bucket Algorithm:

```
class RateLimiter:
    def __init__(self, capacity, refill_rate):
        self.capacity = capacity
        self.tokens = capacity
        self.refill_rate = refill_rate  # tokens per second
        self.last_refill = time.time()

    def consume(self, tokens=1):
        self.refill()
        if self.tokens &gt;= tokens:
            self.tokens -= tokens
```

```
            return True
        return False

    def refill(self):
        now = time.time()
        elapsed = now - self.last_refill
        self.tokens = min(self.capacity, self.tokens + elapsed * self.refill_rate)
        self.last_refill = now
```

Performance Optimization

Latency Targets

- **Write Operations:** <200ms (p99)
- **Read Operations (Cache Hit):** <30ms (p99)
- **Read Operations (Cache Miss):** <150ms (p99)

Optimization Techniques

1. **Connection Pooling:** Reuse database connections
2. **Batch Processing:** Cleanup operations in batches
3. **Async Operations:** Non-blocking I/O for S3 uploads
4. **Compression:** Gzip compression for large pastes
5. **Indexing:** Database indexes on frequently queried columns

Capacity Planning

Traffic Estimates

- **Daily Pastes:** 1 million
- **Daily Reads:** 10 million (10:1 read-write ratio)
- **Peak QPS:** 500-600 (5x average)

Storage Estimates

- **Average Paste Size:** 10KB
- **Daily Storage:** 10GB
- **Annual Storage:** 3.65TB
- **5-Year Storage:** ~18TB

Cost Estimates (Monthly)

- **EC2 Instances (App Servers):** $500
- **RDS (Database):** $800
- **ElastiCache (Redis):** $400
- **S3 Storage:** $400
- **Data Transfer:** $300
- **CloudFront (CDN):** $200
- **Total:** ~$2,600/month

Conclusion

This High Level Design provides a comprehensive, scalable, and reliable architecture for a Pastebin-like system. The design emphasizes:

- **High Availability:** Multi-AZ deployment, replication, automatic failover
- **Low Latency:** Multi-layer caching, CDN, optimized data flow
- **Scalability:** Horizontal scaling, database sharding, auto-scaling
- **Reliability:** Durability guarantees, backup strategies, monitoring
- **Security:** Encryption, access control, rate limiting

The system can handle millions of requests per day with latency under 200ms and provides a solid foundation for future enhancements such as syntax highlighting, collaborative editing, analytics, and more.