# JIUCOM

## MSA Infrastructure Manual

Docker + Monitoring + API Gateway + Distributed Tracing

9-Service Architecture Guide

**Document** : MSA Infrastructure Manual v1.0

**Date** : 2026-02-27

**Project** : JIUCOM API Platform

**Stack** : Spring Boot + Docker + Prometheus + Grafana

# Table of Contents

# 1. Architecture Overview

## >> Overall Architecture

JIUCOM uses a microservice-oriented Docker Compose architecture with 9 services. The system is designed around the following principles:

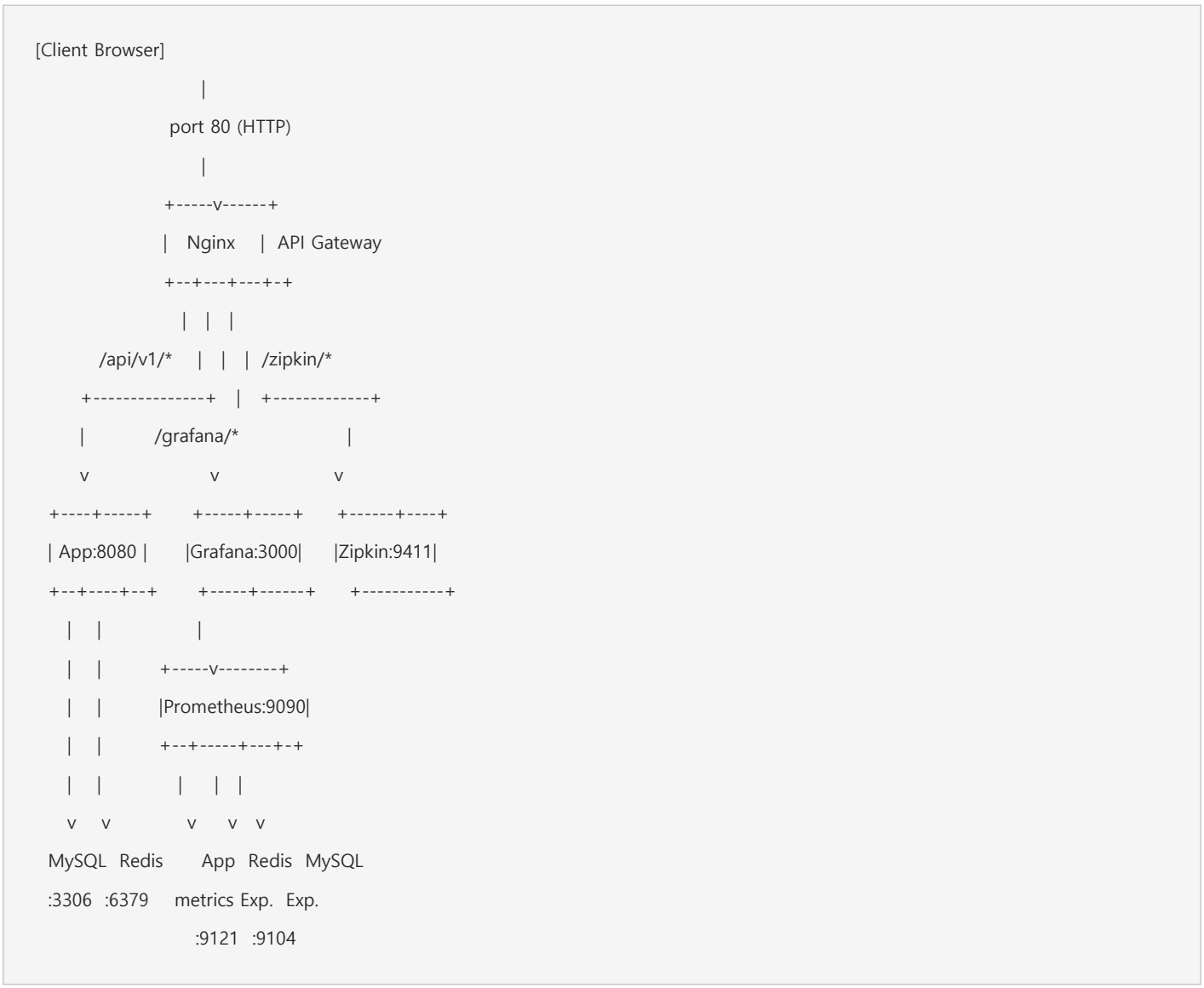
1) API Gateway Pattern - Nginx serves as the single entry point, routing requests to backend services and handling cross-cutting concerns (rate limiting, CORS, security headers).

2) Observability Stack - Prometheus collects metrics from all services, Grafana provides visualization dashboards, and Zipkin enables distributed tracing.

3) Sidecar Exporters - Redis Exporter and MySQL Exporter run as sidecar containers, translating native database metrics into Prometheus-compatible format.

4) Resource Isolation - Every container has CPU/memory limits to prevent resource starvation and simulate production-grade infrastructure.

## >> Service Topology


[Client Browser]
port 80 (HTTP)
Nginx
API Gateway
/api/v1/*
/zipkin/*
/grafana/*
App:8080
Grafana:3000
Zipkin:9411
Prometheus:9090
MySQL
Redis
App
Redis
MySQL
:3306
:6379
metrics Exp.
Exp.
:9121
:9104

# 2. Service Details

## >> All 9 Services

| Service | Image | Port | Role | Resources |
|---|---|---|---|---|
| mysql | mysql:8.0 | 3306 | Primary database (user, part, build, post...) | 2 CPU / 2GB |
| redis | redis:7-alpine | 6379 | Cache + session store (price cache, rate limit) | 1 CPU / 512MB |
| app | Spring Boot (custom build) | 8080 | JIUCOM API server (70+ REST endpoints) | 2 CPU / 1GB |
| nginx | nginx:alpine | 80 | API Gateway reverse proxy | 0.5 CPU / 256MB |
| prometheus | prom/prometheus | 9090 | Metrics collection time-series DB | 0.5 CPU / 512MB |
| grafana | grafana/grafana | 3000 | Monitoring dashboard visualization | 0.5 CPU / 512MB |
| redis-exp | oliver006/ redis_exporter | 9121 | Redis metrics exporter | 0.25 CPU / 128MB |
| mysql-exp | prom/ mysqld-exporter | 9104 | MySQL metrics exporter | 0.25 CPU / 128MB |
| zipkin | openzipkin/ zipkin | 9411 | Distributed tracing request tracking | 0.5 CPU / 512MB |

## >> Service Dependency Chain

Services start in dependency order:

1) mysql, redis (independent, start first with health checks)
2) app (depends on mysql + redis being healthy)
3) prometheus, zipkin (depend on app)

4) grafana (depends on prometheus)

5) redis-exporter, mysql-exporter (depend on redis/mysql)

6) nginx (depends on app, grafana, zipkin - starts last)

# 3. Nginx API Gateway

## >> Role & Responsibilities

Nginx acts as the API Gateway - the single entry point for all external traffic. This is a core MSA pattern that provides:

* Reverse Proxy - Routes requests to the correct backend service based on URL path
* Rate Limiting - Protects backend from DDoS/abuse (30 req/sec per IP, burst 50)
* Security Headers - X-Frame-Options, X-Content-Type-Options, XSS protection
* WebSocket Support - HTTP/1.1 Upgrade for real-time notifications (STOMP)
* Gzip Compression - Reduces response payload size for JSON/text

## >> Routing Rules

| Path | Upstream | Description |
|---|---|---|
| /api/v1/* | app:8080 | All REST API endpoints (auth, parts, builds, posts...) |
| /grafana/* | grafana:3000 | Monitoring dashboard (Grafana UI) |
| /zipkin/* | zipkin:9411 | Distributed tracing UI (Zipkin web console) |
| /health | (direct) | Nginx health check returns 200 OK |
| / | (redirect) | Redirects to /api/v1/swagger-ui.html |

## >> How Reverse Proxy Works

When a client sends a request to http://localhost/api/v1/parts:

1) Nginx receives the request on port 80
2) The 'location /api/v1/' block matches the URL
3) Rate limiter checks if the client IP is within the 30 req/sec limit

4) If allowed, Nginx forwards (proxies) the request to http://app:8080/api/v1/parts

5) Nginx adds headers: X-Real-IP, X-Forwarded-For, X-Forwarded-Proto

6) The Spring Boot app processes the request and returns a response

7) Nginx adds security headers and forwards the response to the client

The client never communicates directly with the Spring Boot app. This provides a layer of abstraction and security.

# 4. Monitoring Stack

## >> Prometheus - Metrics Collection

Prometheus is a time-series database that collects metrics from all services using a pull-based model. Every 15 seconds, Prometheus scrapes (HTTP GET) each target's metrics endpoint.

> Prometheus Pull Model: Prometheus --HTTP GET--> Target /metrics

Scrape targets configured in prometheus.yml:

| Job Name | Target | Metrics Path | What It Collects |
|---|---|---|---|
| jiucom-api | app:8080 | /api/v1/actuator/prometheus | JVM heap, GC, threads HTTP requests, HikariCP |
| redis-exporter | redis-exporter :9121 | /metrics | Connected clients memory usage, commands |
| mysql-exporter | mysql-exporter :9104 | /metrics | Queries/sec, threads slow queries, connections |

## >> How Spring Boot Exposes Metrics

The Spring Boot app uses Micrometer (a metrics facade) to expose metrics:

1) spring-boot-starter-actuator enables the /actuator endpoints
2) micrometer-registry-prometheus formats metrics in Prometheus text format
3) application.yml exposes: health, info, metrics, prometheus, loggers
4) Prometheus scrapes GET /api/v1/actuator/prometheus every 15s

Key metrics exposed:

* jvm_memory_used_bytes - JVM heap/non-heap memory usage
* jvm_threads_live_threads - Number of active threads
* http_server_requests_seconds_count - HTTP request count by status/method/uri
* http_server_requests_seconds_sum - Total response time (for avg calculation)
* hikaricp_connections_active - Active database connections

* jvm_gc_pause_seconds_count - Garbage collection frequency

## >> Grafana - Visualization Dashboard

Grafana reads data from Prometheus and displays it as visual dashboards. The JIUCOM dashboard is auto-provisioned on startup.

Auto-provisioning files (mounted as Docker volumes):

* datasource.yml - Connects Grafana to Prometheus (http://prometheus:9090)
* dashboard.yml - Tells Grafana to load dashboards from /var/lib/grafana/dashboards/
* jiucom-dashboard.json - Pre-built dashboard with 16 panels

## >> Dashboard Panels (16 Total)

| Section | Panels | Metrics Used |
|---|---|---|
| Application Overview | API Status, Heap Usage | up{}, jvm_memory_* |
| | Threads, Request Rate | jvm_threads_* |
| | Error Rate (5xx) | http_server_requests_* |
| HTTP Metrics | Response Time (avg) | http_server_requests_ |
| | Requests by Status | seconds_sum/count |
| JVM Metrics | Heap Memory (used/committed/max) | jvm_memory_used_bytes |
| | GC Pause Count | jvm_gc_pause_* |
| Database & Cache | HikariCP Pool | hikaricp_connections_* |
| | Redis Clients | redis_connected_clients |
| | Redis Memory | redis_memory_used_bytes |
| MySQL Metrics | Queries/sec | mysql_global_status_ |
| | Slow Queries | queries/threads_* |
| | Threads | |

## >> Exporter Pattern

Redis and MySQL don't natively expose Prometheus metrics. Exporters are sidecar containers that:

1) Connect to the database (Redis/MySQL) using native protocol
2) Query internal statistics (INFO, SHOW GLOBAL STATUS)

3) Translate those stats into Prometheus metric format

4) Expose them on /metrics endpoint for Prometheus to scrape

This is the standard Prometheus 'Exporter' pattern for third-party systems.

# 5. Distributed Tracing (Zipkin)

## >> What is Distributed Tracing?

In a microservice architecture, a single user request may pass through multiple services. Distributed tracing tracks the entire journey of a request across all services, showing:

- Which services were called
- How long each service took
- Where bottlenecks or errors occurred

Each request gets a unique Trace ID, and each service call within that request gets a Span ID.

## >> How It Works in JIUCOM

1) Client sends request to Nginx -> forwarded to Spring Boot app

2) Micrometer Tracing (Brave bridge) automatically:
   - Generates a Trace ID for the request
   - Creates a Span for each operation (HTTP handler, DB query, Redis call)
   - Propagates trace context via HTTP headers (B3 propagation)

3) When the request completes, spans are sent asynchronously to Zipkin
   via HTTP POST to http://zipkin:9411/api/v2/spans

4) Zipkin stores the trace data (in-memory for dev, could use Elasticsearch for prod)

5) Zipkin UI (port 9411) visualizes the trace as a timeline/waterfall diagram

## >> Dependencies Added

```
// build.gradle.kts
implementation("io.micrometer:micrometer-tracing-bridge-brave")
implementation("io.zipkin.reporter2:zipkin-reporter-brave")

// application.yml
management.tracing.sampling.probability: 1.0  (dev: 100%)
management.zipkin.tracing.endpoint: http://zipkin:9411/api/v2/spans
```

## >> Sampling Strategy

| Profile | Sampling Rate | Reason |
|---|---|---|
| dev | 1.0 (100%) | Trace every request for debugging |
| prod | 0.1 (10%) | Reduce overhead, sample enough for analysis |

# 6. Docker Compose Orchestration

## >> Container Resource Limits

Every container has CPU and memory limits to prevent resource starvation. This simulates production-grade Kubernetes resource management.

| Service | CPU Limit | Memory Limit | CPU Reserve | Memory Reserve |
|---|---|---|---|---|
| app | 2.0 | 1 GB | 0.5 | 512 MB |
| mysql | 2.0 | 2 GB | 0.5 | 512 MB |
| redis | 1.0 | 512 MB | 0.25 | 128 MB |
| nginx | 0.5 | 256 MB | 0.1 | 64 MB |
| prometheus | 0.5 | 512 MB | 0.25 | 256 MB |
| grafana | 0.5 | 512 MB | 0.25 | 256 MB |
| zipkin | 0.5 | 512 MB | 0.25 | 256 MB |
| redis-exp | 0.25 | 128 MB | 0.1 | 64 MB |
| mysql-exp | 0.25 | 128 MB | 0.1 | 64 MB |

## >> Network Architecture

All services share a single Docker bridge network: jiucom-network

Within this network, containers communicate using service names as DNS hostnames. For example, the app container connects to 'mysql:3306' and 'redis:6379'. Docker's embedded DNS resolves these names to container IP addresses.

Only specific ports are published to the host machine:
- Port 80 (Nginx) - main entry point
- Port 8080 (App) - direct API access (for development)
- Port 3000 (Grafana), 9090 (Prometheus), 9411 (Zipkin) - monitoring

## >> Volume Persistence

| Volume | Container Path | Purpose |
|---|---|---|

| | | |
|---|---|---|
| mysql-data | /var/lib/mysql | MySQL data files<br>(survives container restart) |
| redis-data | /data | Redis RDB snapshots |
| prometheus-data | /prometheus | Time-series metrics data<br>(15 day retention) |
| grafana-data | /var/lib/grafana | Grafana settings, users<br>(not dashboards - those are provisioned) |

# 7. Data Flow - Request Lifecycle

## >> Example: GET /api/v1/parts (Search Parts)

Step-by-step flow when a client searches for computer parts:

### Phase 1: Request Ingress

* Client sends HTTP GET http://localhost/api/v1/parts?category=CPU
* Nginx receives on port 80, matches 'location /api/v1/'
* Rate limiter checks: client IP within 30 req/sec? If yes, proceed
* Nginx proxies to http://app:8080/api/v1/parts?category=CPU
* Adds headers: X-Real-IP, X-Forwarded-For, X-Forwarded-Proto

### Phase 2: Application Processing

* Spring Boot receives request, Micrometer creates Trace ID + Span
* JwtAuthenticationFilter checks Authorization header (if present)
* RequestLoggingFilter logs: method, URI, client IP, start time
* PartController.searchParts() invoked by DispatcherServlet
* PartService calls PartRepository (QueryDSL dynamic query)
* HikariCP provides a connection from the pool to MySQL
* Query executes, results mapped to PartSearchResponse DTOs

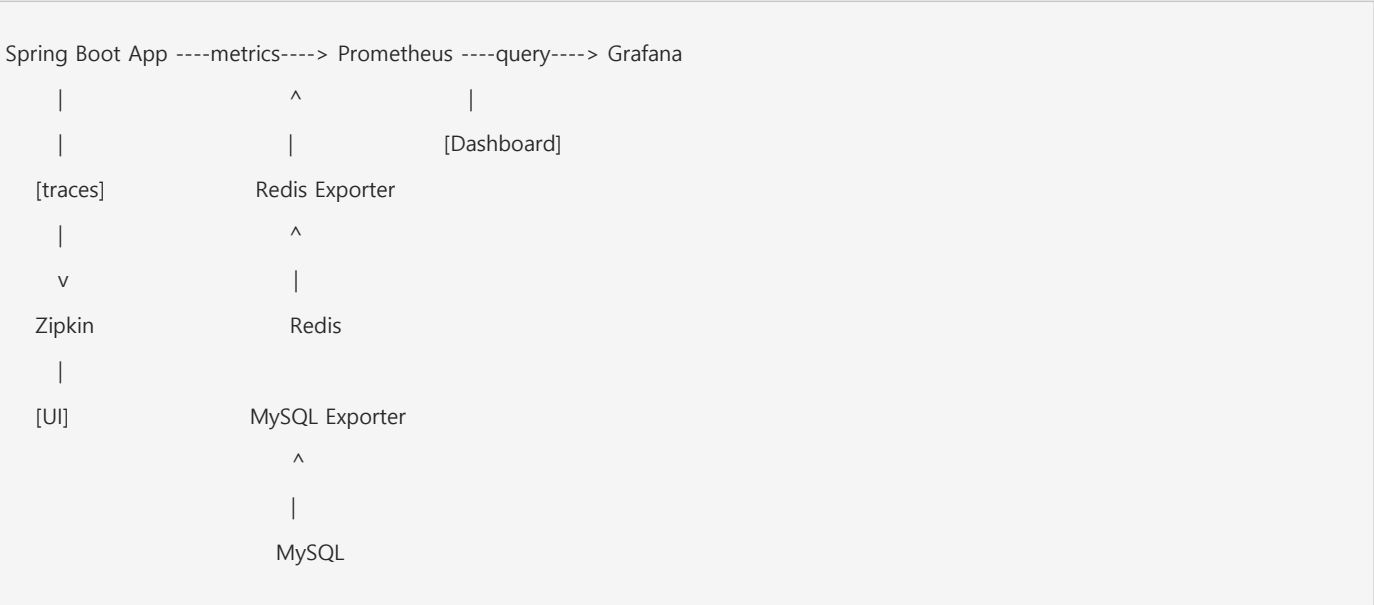
### Phase 3: Response Egress

* ApiResponse<T> wraps the result: {success: true, data: [...], message: ...}
* Micrometer records: http_server_requests_seconds (duration, status, uri)
* Trace span sent async to Zipkin (http://zipkin:9411/api/v2/spans)
* Response returns through Nginx, security headers added
* Client receives JSON response with parts data

Phase 4: Metrics Collection (Background)

* Every 15 seconds, Prometheus scrapes GET /api/v1/actuator/prometheus
* Metrics collected: request count, response time, JVM heap, thread count
* Prometheus stores in TSDB (time-series database), 15-day retention
* Grafana queries Prometheus via PromQL for dashboard visualization
* Redis Exporter scrapes Redis INFO, MySQL Exporter scrapes SHOW STATUS

## >> Monitoring Data Flow



# 8. Dashboard & Alert Design

## >> Grafana Dashboard Layout

The pre-provisioned JIUCOM Dashboard (UID: jiucom-main) has 4 row sections with 16 panels total. Auto-refresh is set to every 10 seconds.

| Row | Panel Type | What to Watch | Warning Sign |
|---|---|---|---|
| Overview (Row 1) | Stat panels x5 | API UP status | DOWN status |
| | | Heap usage % | Heap > 90% |
| | | HTTP request rate | Rate spike/drop |
| HTTP (Row 2) | Time series x2 | Avg response time | Latency > 1s |
| | | Requests by status code | 5xx errors > 1% |
| JVM (Row 3) | Time series x2 | Heap memory trend | Memory climbing |
| | | GC pause frequency | Frequent full GC |
| DB/Cache (Row 4) | Time series x5 | HikariCP pool usage | Pool exhaustion |
| | | Redis clients/memory | Redis memory full |
| | | MySQL queries/threads | Slow query spike |

## >> Key PromQL Queries

```
# API Server Up/Down
up{job="jiucom-api"}

# JVM Heap Usage Percentage
jvm_memory_used_bytes{area="heap"} / jvm_memory_max_bytes{area="heap"} * 100

# HTTP Request Rate (per second, 5m average)
rate(http_server_requests_seconds_count{job="jiucom-api"}[5m])

# HTTP 5xx Error Rate (%)
rate(http_server_requests_seconds_count{status=~"5.."}[5m])
  / rate(http_server_requests_seconds_count[5m]) * 100

# Average Response Time (seconds)
rate(http_server_requests_seconds_sum[5m])
  / rate(http_server_requests_seconds_count[5m])
```

# 9. Operations Guide

## >> Start / Stop Commands

```
# Start all 9 services (build app image first)
docker compose up -d --build

# Start without rebuilding
docker compose up -d

# Stop all services (keep data volumes)
docker compose down

# Stop and delete all data (DESTRUCTIVE)
docker compose down -v

# View logs for specific service
docker compose logs -f app
docker compose logs -f nginx

# Restart single service
docker compose restart app
```

## >> Health Check URLs

| Service | URL | Expected |
|---|---|---|
| API (via Nginx) | http://localhost/api/v1/actuator/health | 200 OK {status: UP} |
| API (direct) | http://localhost:8080/api/v1/actuator/health | 200 OK {status: UP} |
| Prometheus | http://localhost:9090/targets | 3 targets UP |
| Grafana | http://localhost:3000/grafana/login | Login page (admin/admin) |

| | | |
|---|---|---|
| Zipkin | http://localhost:9411/<br>zipkin/ | Zipkin UI |
| Swagger | http://localhost/api/v1/<br>swagger-ui.html | Swagger UI |

## >> Troubleshooting

| Problem | Diagnosis | Solution |
|---|---|---|
| App won't start | docker compose logs app<br>Check DB connection | Ensure mysql is healthy<br>Check DB_HOST env var |
| Prometheus<br>no targets | http://localhost:9090<br>/targets | Check prometheus.yml<br>Ensure app is running |
| Grafana<br>no data | Check datasource<br>settings in Grafana | Verify Prometheus URL<br>http://prometheus:9090 |
| Nginx<br>502 Bad Gateway | docker compose logs nginx<br>Check upstream | Ensure app container<br>is running and healthy |
| Zipkin<br>no traces | Check app logs for<br>Zipkin reporter errors | Verify ZIPKIN_ENDPOINT<br>env var in app |
| MySQL exporter<br>connection refused | docker compose logs<br>mysql-exporter | Check DATA_SOURCE_NAME<br>MySQL must be healthy |

## >> Scaling Considerations

For production deployment, consider these upgrades:

1) Nginx: Add SSL/TLS termination with Let's Encrypt certificates
2) App: Scale horizontally with 'docker compose up --scale app=3'
3) Zipkin: Switch from in-memory to Elasticsearch storage
4) Prometheus: Add alertmanager for Slack/email notifications
5) Grafana: Configure SMTP for scheduled report delivery
6) MySQL: Add read replicas for query distribution
7) Redis: Enable Redis Cluster for high availability

---

**JIUCOM - Production-Ready MSA Infrastructure**

Document generated: 2026-02-27 | JIUCOM Team