AI Employee Platform - Performance Documentation

Overview

This document provides comprehensive performance guidelines, benchmarks, and monitoring procedures for the AI Employee Platform. It covers performance requirements, testing methodologies, optimization strategies, and operational procedures.

📊 Performance Requirements

Response Time Requirements

Endpoint Type	Target Response Time	Maximum Acceptable
Health Checks	< 100ms	200ms
Authentication	< 500ms	1000ms
User Registration	< 1000ms	2000ms
Profile Access	< 200ms	500ms
AI API Requests	< 5000ms	10000ms
Database Queries	< 100ms	500ms

Throughput Requirements

Service	Target RPS	Peak RPS	Concurrent Users
Auth Service	100	500	100
Health Endpoints	500	2000	200
API Gateway	200	1000	150
Database Operations	150	750	100

Resource Utilization Limits

Resource	Normal Operation	Peak Load	Alert Threshold
CPU Usage	< 50%	< 80%	70%
Memory Usage	< 256MB	< 512MB	400MB
Database Connections	< 20	< 50	40
Redis Memory	< 100MB	< 256MB	200MB

🧪 Performance Testing

Test Types

1. Load Testing

- **Purpose**: Verify system behavior under expected load
- **Duration**: 10-60 minutes
- **Users**: 10-100 concurrent users
- **Success Criteria**: All response time and throughput targets met

2. Stress Testing

- **Purpose**: Determine system breaking point
- **Duration**: 30-120 minutes
- **Users**: 100-500 concurrent users
- **Success Criteria**: System degrades gracefully, recovers after load removal

3. Spike Testing

- **Purpose**: Test system response to sudden load increases
- **Duration**: 5-15 minutes
- **Pattern**: Rapid scaling from baseline to peak load
- **Success Criteria**: System handles spikes without crashes

4. Endurance Testing

- **Purpose**: Verify system stability over extended periods
- **Duration**: 2-24 hours
- **Load**: Sustained moderate load
- **Success Criteria**: No memory leaks, performance degradation, or resource exhaustion

Test Scenarios

Scenario 1: Light Load

```
# Configuration
Duration: 30 seconds
Concurrent Users: 5
Ramp-up Time: 5 seconds

# Expected Results
- Response Time: < 200ms (avg)
- Throughput: > 50 RPS
- Error Rate: < 0.1%
```

Scenario 2: Moderate Load

```
# Configuration
Duration: 60 seconds
Concurrent Users: 25
Ramp-up Time: 10 seconds

# Expected Results
- Response Time: < 500ms (avg)
- Throughput: > 100 RPS
- Error Rate: < 1%
```

Scenario 3: Heavy Load

```
# Configuration
Duration: 120 seconds
Concurrent Users: 50
Ramp-up Time: 20 seconds

# Expected Results
- Response Time: < 1000ms (avg)
- Throughput: > 150 RPS
- Error Rate: < 2%
```

Running Performance Tests

Using Load Test Script

```
# Run predefined scenarios
./scripts/load-test.sh scenario light
./scripts/load-test.sh scenario moderate
./scripts/load-test.sh scenario heavy

# Run all tests
./scripts/load-test.sh all

# Custom Artillery test
./scripts/load-test.sh artillery "duration=120,users=50,ramp=15"

# Apache Bench test
./scripts/load-test.sh ab 1000 10

# Generate performance report
./scripts/load-test.sh report
```

Using Jest Performance Tests

```
# Run performance test suite
yarn test tests/performance/

# Run specific performance tests
yarn test tests/performance/auth-service.performance.test.ts
yarn test tests/performance/system.performance.test.ts

# Run with verbose output
yarn test tests/performance/ --verbose
```

Test Environment Setup

Prerequisites

```
# Install testing tools
npm install -g artillery@latest

# For Ubuntu/Debian
sudo apt-get install apache2-utils

# For macOS
brew install apache-bench

# Verify installations
artillery --version
ab -V
```

Environment Configuration

```
# Set environment variables
export NODE_ENV=test
export DATABASE_URL="postgresql://postgres:testpassword@localhost:5432/
ai_platform_test"
export REDIS_URL="redis://localhost:6379/1"

# Start test services
./scripts/test-setup.sh setup

# Verify services are running
curl http://localhost:3001/health
```

📈 Performance Monitoring

Key Metrics

Application Metrics

- **Response Time**: P50, P95, P99 response times
- **Throughput**: Requests per second
- **Error Rate**: Percentage of failed requests
- **Active Connections**: Current database/Redis connections

System Metrics

- **CPU Usage**: Per-core and overall utilization
- **Memory Usage**: Heap usage, RSS, external memory
- **Disk I/O**: Read/write operations per second
- **Network I/O**: Bytes sent/received per second

Business Metrics

- **User Registrations**: Rate of new user signups
- **Authentication Success Rate**: Percentage of successful logins
- **API Usage**: AI API calls per minute
- **Credit Consumption**: Credits used per time period

Monitoring Tools

Application Performance Monitoring (APM)

```
# Health check endpoints
GET /health
GET /metrics

# Response format
{
  "status": "healthy",
  "timestamp": "2025-08-08T10:00:00Z",
  "uptime": 3600,
  "checks": {
    "database": { "status": "healthy", "responseTime": "45ms" },
    "redis": { "status": "healthy", "responseTime": "8ms" },
    "memory": { "status": "healthy", "usage": "45%" }
  },
  "metrics": {
    "http_requests_total": 15420,
    "http_request_duration_ms": {
      "p50": 125,
      "p95": 450,
      "p99": 850
    },
    "active_connections": 12,
    "memory_usage_mb": 234
  }
}
```

Log-based Monitoring

```
# Access ELK stack
http://localhost:5601  # Kibana dashboard

# Key log patterns
- Response time > 1000ms
- Error rate > 1%
- Memory usage > 80%
- Database connection pool exhaustion
```

Real-time Monitoring

```
# Process monitoring
top -p $(pgrep -f "auth-service")
htop

# Memory monitoring
free -m
cat /proc/meminfo

# Network monitoring
netstat -tuln
ss -tuln
```

Alert Thresholds

Critical Alerts (Immediate Response)

- Response time > 5000ms (sustained 2 minutes)
- Error rate > 5% (sustained 1 minute)
- Memory usage > 90%
- CPU usage > 95% (sustained 5 minutes)
- Database connections > 90% of pool

Warning Alerts (Monitor Closely)

- Response time > 2000ms (sustained 5 minutes)
- Error rate > 2% (sustained 3 minutes)
- Memory usage > 75%
- CPU usage > 80% (sustained 10 minutes)
- Database connections > 70% of pool

🚀 Performance Optimization

Database Optimization

Query Optimization

```sql
-- Index critical columns
CREATE INDEX idx_users_email ON users(email);
CREATE INDEX idx_transactions_user_id ON transactions(user_id);
CREATE INDEX idx_ai_requests_created_at ON ai_requests(created_at);

-- Analyze query performance
EXPLAIN ANALYZE SELECT * FROM users WHERE email = ?;
```

Connection Pooling

```javascript
// Prisma connection pool configuration
const prisma = new PrismaClient({
  datasources: {
    db: {
      url: process.env.DATABASE_URL,
    },
  },
  // Connection pool settings
  pool: {
    min: 5,
    max: 20,
    acquireTimeoutMillis: 60000,
    createTimeoutMillis: 30000,
    destroyTimeoutMillis: 5000,
    idleTimeoutMillis: 600000,
    reapIntervalMillis: 1000,
  },
});
```

Application-Level Optimization

Caching Strategies

```
// Redis caching for frequently accessed data
import Redis from 'ioredis';

const redis = new Redis({
  host: process.env.REDIS_HOST,
  port: process.env.REDIS_PORT,
  // Connection pooling
  lazyConnect: true,
  maxRetriesPerRequest: 3,
  retryDelayOnFailover: 100,
});

// Cache user profiles
const cacheKey = `user:profile:${userId}`;
const cached = await redis.get(cacheKey);

if (cached) {
  return JSON.parse(cached);
}

const profile = await fetchUserProfile(userId);
await redis.setex(cacheKey, 300, JSON.stringify(profile)); // 5 minutes TTL
```

Response Optimization

```
// Compression middleware
import compression from 'compression';

app.use(compression({
  level: 6,
  threshold: 1024,
  filter: (req, res) => {
    if (req.headers['x-no-compression']) {
      return false;
    }
    return compression.filter(req, res);
  }
}));

// Response headers for caching
app.use((req, res, next) => {
  if (req.path.startsWith('/static/')) {
    res.setHeader('Cache-Control', 'public, max-age=31536000'); // 1 year
  } else if (req.path === '/health') {
    res.setHeader('Cache-Control', 'no-cache');
  }
  next();
});
```

Infrastructure Optimization

Load Balancing

```
# Nginx upstream configuration
upstream auth_service {
    least_conn;
    server auth-service-1:3001 weight=1 max_fails=3 fail_timeout=30s;
    server auth-service-2:3001 weight=1 max_fails=3 fail_timeout=30s;
    server auth-service-3:3001 weight=1 max_fails=3 fail_timeout=30s;
}

server {
    location /api/auth/ {
        proxy_pass http://auth_service;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_connect_timeout 30s;
        proxy_send_timeout 30s;
        proxy_read_timeout 30s;
    }
}
```

Resource Limits

```
# Docker resource limits
services:
  auth-service:
    deploy:
      resources:
        limits:
          cpus: '1.0'
          memory: 512M
        reservations:
          cpus: '0.5'
          memory: 256M
    environment:
      - NODE_OPTIONS="--max-old-space-size=384"
```

📋 Performance Baselines

Baseline Measurements

These baselines were established using standard hardware (4 CPU cores, 8GB RAM) and represent minimum acceptable performance levels:

Authentication Service

Metric	Light Load	Moderate Load	Heavy Load
Login Response Time (avg)	150ms	300ms	500ms
Registration Response Time (avg)	400ms	700ms	1200ms
Throughput (RPS)	75	125	175
Memory Usage	180MB	220MB	280MB
CPU Usage	25%	45%	65%

Database Operations

Operation	Response Time (avg)	P95 Response Time	Throughput (ops/sec)
User Lookup	15ms	45ms	200
Transaction Insert	25ms	75ms	150
Credit Balance Update	20ms	60ms	175
AI Request Log	30ms	90ms	125

Performance Regression Detection

Automated Performance Tests

```
# Run performance regression suite
yarn test:performance

# Compare with baselines
yarn test:performance --compare-baseline

# Update baselines (after verification)
yarn test:performance --update-baseline
```

Continuous Integration

```yaml
# GitHub Actions workflow
name: Performance Tests
on:
  pull_request:
    branches: [main]

jobs:
  performance:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Setup Node.js
        uses: actions/setup-node@v3
        with:
          node-version: '20'
      - name: Install dependencies
        run: yarn install
      - name: Setup test environment
        run: ./scripts/test-setup.sh setup
      - name: Run performance tests
        run: yarn test:performance
      - name: Check performance regression
        run: |
          if [ -f "performance-regression.txt" ]; then
            echo "Performance regression detected!"
            cat performance-regression.txt
            exit 1
          fi
```

🔧 Troubleshooting Performance Issues

Common Performance Problems

High Response Times

Symptoms: Average response times > 1000ms
Possible Causes:
- Database query inefficiency
- Unoptimized algorithms
- Network latency
- Resource contention

Diagnosis Steps:

```
# Check database performance
EXPLAIN ANALYZE [slow query];

# Monitor system resources
top -p [process_id]
iostat -x 1

# Check network connectivity
ping [database_host]
telnet [database_host] [port]
```

Solutions:
- Add database indexes
- Optimize query patterns
- Implement caching
- Scale vertically or horizontally

Memory Leaks

Symptoms: Gradually increasing memory usage
Possible Causes:
- Unreleased event listeners
- Circular references
- Large object accumulation
- Inefficient garbage collection

Diagnosis Steps:

```
# Monitor memory usage over time
node --expose-gc --inspect app.js

# Generate heap dump
kill -USR2 [process_id]

# Analyze heap dump
node --inspect-brk=0.0.0.0:9229 app.js
```

Solutions:
- Remove unused event listeners
- Break circular references
- Implement object pooling
- Tune garbage collection parameters

Database Connection Exhaustion

Symptoms: “Too many connections” errors
Possible Causes:
- Connection leaks
- Inadequate connection pooling
- Long-running transactions
- Inefficient connection management

Diagnosis Steps:

```
-- Check active connections
SELECT * FROM pg_stat_activity WHERE state = 'active';

-- Monitor connection pool
SELECT count(*) FROM pg_stat_activity;
```

Solutions:
- Fix connection leaks
- Optimize connection pool size
- Implement connection timeouts
- Use connection pooling middleware

Performance Tuning Checklist

Application Level

- [ ] Enable response compression
- [ ] Implement proper caching strategy
- [ ] Optimize database queries
- [ ] Use connection pooling
- [ ] Minimize payload sizes
- [ ] Implement pagination for large datasets

Database Level

- [ ] Add appropriate indexes
- [ ] Optimize query execution plans
- [ ] Configure connection pooling
- [ ] Monitor slow query log
- [ ] Implement read replicas for scaling

Infrastructure Level

- [ ] Configure load balancing
- [ ] Set appropriate resource limits
- [ ] Enable monitoring and alerting
- [ ] Implement auto-scaling policies
- [ ] Optimize network configuration

Monitoring Level

- [ ] Set up performance monitoring
- [ ] Configure alert thresholds
- [ ] Implement log aggregation
- [ ] Create performance dashboards
- [ ] Establish baseline measurements

📚 Additional Resources

Tools and Libraries

- Artillery (https://artillery.io/) - Load testing toolkit
- Apache Bench (https://httpd.apache.org/docs/2.4/programs/ab.html) - HTTP server benchmarking
- Clinic.js (https://clinicjs.org/) - Node.js performance profiling
- 0x (https://github.com/davidmarkclements/0x) - Flamegraph profiling

Best Practices References

- Node.js Performance Best Practices (https://nodejs.org/en/docs/guides/simple-profiling/)
- PostgreSQL Performance Tuning (https://wiki.postgresql.org/wiki/Performance_Optimization)
- Redis Performance Best Practices (https://redis.io/topics/latency)
- Nginx Performance Tuning (https://nginx.org/en/docs/http/ngx_http_core_module.html)

Monitoring Solutions

- Prometheus + Grafana (https://prometheus.io/) - Metrics collection and visualization
- ELK Stack (https://www.elastic.co/elk-stack) - Log aggregation and analysis

- New Relic (https://newrelic.com/) - APM solution
- DataDog (https://www.datadoghq.com/) - Infrastructure monitoring

---

This performance documentation is maintained by the AI Employee Platform team. Last updated: August 2025