## Contents

# RADIANT Performance Guide

## Overview

This guide covers performance optimization, caching strategies, and scalability considerations for the RADIANT platform.

## Architecture Performance

### Request Flow

```
Client → CloudFront → WAF → API Gateway → Lambda → Aurora
                                                ↓
                                           Redis Cache
```

### Latency Targets

| Component | Target | Max Acceptable |
|---|---|---|
| CloudFront edge | $<$ 50ms | 100ms |
| WAF processing | $<$ 5ms | 20ms |
| API Gateway | $<$ 20ms | 50ms |
| Lambda cold start | $<$ 500ms | 1000ms |
| Lambda execution | $<$ 200ms | 500ms |
| Database query | $<$ 50ms | 200ms |
| **Total P95** | **$<$ 500ms** | **2000ms** |

## Caching Strategy

### Multi-Layer Caching

```
                   CloudFront CDN
  TTL: 5m for static, 1m for API (with stale-while-revalidate)

                          ↓

                  API Gateway Cache
         TTL: 60s for GET endpoints

                          ↓

                    Redis Cache
     Session: 24h, Config: 5m, Translations: 1h

                          ↓

                  Aurora Database
         Query cache, Connection pooling
```

### Cache Keys

```
// Session cache
`session:${tenantId}:${userId}` → TTL: 24h

// Configuration cache
```

```
`config:${tenantId}:${key}` → TTL: 5m
`config:global:${key}` → TTL: 5m

// Translation cache
`i18n:${language}:bundle` → TTL: 1h
`i18n:${language}:${key}` → TTL: 1h

// Model cache
`models:${tenantId}:list` → TTL: 5m
`models:${tenantId}:${modelId}` → TTL: 5m

// Rate limit cache
`ratelimit:${tenantId}:${endpoint}` → TTL: 1m
`ratelimit:ip:${ip}` → TTL: 5m
```

### Cache Invalidation

```
// Pattern-based invalidation
await redis.del(`config:${tenantId}:*`);

// Event-driven invalidation
eventBridge.putEvents({
  Entries: [{
    Source: 'radiant.config',
    DetailType: 'ConfigUpdated',
    Detail: JSON.stringify({ tenantId, key }),
  }],
});
```

## Database Optimization

### Connection Pooling

```
// RDS Proxy configuration
const pool = {
  min: 2,
  max: 10,
  idleTimeoutMillis: 30000,
  connectionTimeoutMillis: 5000,
};
```

### Query Optimization

```
-- Always use indexes
CREATE INDEX idx_models_tenant_status ON ai_models(tenant_id, status);
CREATE INDEX idx_transactions_tenant_date ON credit_transactions(tenant_id, created_at DESC);

-- Use covering indexes for common queries
CREATE INDEX idx_models_list ON ai_models(tenant_id, status, is_enabled)
```

```
  INCLUDE (display_name, category, input_cost_per_1k);

-- Partition large tables by date
CREATE TABLE audit_logs_2024_01 PARTITION OF audit_logs
  FOR VALUES FROM ('2024-01-01') TO ('2024-02-01');
```

### RLS Performance

```
-- Set tenant context once per request
SET app.current_tenant_id = 'tenant-123';

-- All subsequent queries automatically filtered
SELECT * FROM models;  -- Implicitly filtered by RLS
```

## Lambda Optimization

### Cold Start Reduction

```
// 1. Minimize dependencies
// 2. Use Lambda layers for shared code
// 3. Enable provisioned concurrency for critical functions

// Provisioned concurrency config
new lambda.Function(this, 'Router', {
  // ... config
  provisionedConcurrentExecutions: 10, // Keep 10 warm
});
```

### Memory Optimization

```
// Memory vs CPU tradeoff
// More memory = more CPU = faster execution

// Recommended settings by function type:
const memoryConfig = {
  router: 1024,      // Main API - balanced
  billing: 512,      // Light compute
  aiProxy: 2048,     // Heavy compute for AI
  migration: 256,    // Infrequent, light
};
```

### Bundling

```
// esbuild configuration for minimal bundle size
{
  bundle: true,
  minify: true,
  treeShaking: true,
  external: ['aws-sdk'], // Use Lambda runtime SDK
  target: 'node20',
}
```

## Rate Limiting

### Tier Limits

| Tier | RPS | Burst | Daily |
|---|---|---|---|
| Free | 10 | 20 | 1,000 |
| Starter | 50 | 100 | 10,000 |
| Professional | 100 | 200 | 50,000 |
| Business | 500 | 1,000 | 250,000 |
| Enterprise | 2,000 | 5,000 | Unlimited |

### Implementation

```
// Token bucket algorithm in Redis
const rateLimiter = {
  async checkLimit(tenantId: string, limit: number): Promise<boolean> {
    const key = `ratelimit:${tenantId}`;
    const current = await redis.incr(key);

    if (current === 1) {
      await redis.expire(key, 1); // 1 second window
    }

    return current <= limit;
  }
};
```

## Load Testing

### Running Tests

```
# Install k6
brew install k6

# Run smoke test
k6 run --env BASE_URL=https://api-dev.radiant.example.com tests/load/k6-config.js

# Run with specific scenario
k6 run --env BASE_URL=https://api-dev.radiant.example.com \
  -e SCENARIO=load tests/load/k6-config.js
```

### Performance Baselines

| Metric | Baseline | Target |
|---|---|---|
| Throughput | 500 RPS | 2000 RPS |
| P50 Latency | 100ms | 50ms |
| P95 Latency | 500ms | 200ms |
| P99 Latency | 1000ms | 500ms |
| Error Rate | < 1% | < 0.1% |

## Scaling

### Horizontal Scaling

| Component | Scaling Method |
|---|---|
| Lambda | Automatic (up to account limit) |
| Aurora | Read replicas, Serverless v2 |
| Redis | ElastiCache cluster mode |
| API Gateway | Automatic |

### Vertical Scaling

```
// Aurora Serverless v2 scaling
const database = new rds.DatabaseCluster(this, 'Database', {
  serverlessV2MinCapacity: 0.5,  // Minimum ACUs
  serverlessV2MaxCapacity: 16,   // Maximum ACUs
});

// Lambda memory scaling
const lambda = new lambda.Function(this, 'Function', {
  memorySize: 2048,  // More memory = more CPU
});
```

## Monitoring

### Key Metrics

```
// CloudWatch metrics to monitor
const metrics = {
  // Latency
  'AWS/ApiGateway/Latency': 'p95 < 500ms',
  'AWS/Lambda/Duration': 'p95 < 200ms',
  'AWS/RDS/ReadLatency': 'avg < 50ms',

  // Throughput
  'AWS/ApiGateway/Count': 'track trends',
  'AWS/Lambda/Invocations': 'track trends',

  // Errors
  'AWS/ApiGateway/5XXError': 'rate < 1%',
  'AWS/Lambda/Errors': 'rate < 1%',

  // Resources
  'AWS/Lambda/ConcurrentExecutions': '< 80% of limit',
  'AWS/RDS/CPUUtilization': '< 80%',
  'AWS/RDS/DatabaseConnections': '< 80% of max',
};
```

### Alerting Thresholds

| Metric | Warning | Critical |
|---|---|---|
| API P95 Latency | > 1s | > 3s |
| Error Rate | > 1% | > 5% |
| Lambda Concurrent | > 500 | > 800 |
| DB CPU | > 70% | > 85% |
| DB Connections | > 60% | > 80% |

## Best Practices

### Do's

- Cache aggressively with proper invalidation
- Use connection pooling
- Minimize cold starts with provisioned concurrency
- Use read replicas for read-heavy workloads
- Implement circuit breakers for external services
- Use async processing for non-critical paths

### Don'ts

- Don't make synchronous calls to external APIs in hot paths
- Don't use Lambda for long-running tasks (> 15 min)
- Don't store large objects in Redis
- Don't rely on API Gateway caching for dynamic data
- Don't skip database indexes

## Troubleshooting

### High Latency

1. Check Lambda cold starts (enable provisioned concurrency)
2. Check database query times (add indexes)
3. Check external API latency (add caching/circuit breaker)
4. Check connection pool exhaustion

### High Error Rate

1. Check Lambda errors in CloudWatch Logs
2. Check database connection errors

3. Check rate limiting (429 errors)
4. Check WAF blocked requests

**Scaling Issues**

1. Check Lambda concurrent execution limit
2. Check database connection limit
3. Check API Gateway throttling
4. Check Redis memory usage