HashInsight Platform Operations Manual

Document Version: v2.0
Last Updated: October 3, 2025
Maintained By: HashInsight Platform Operations Team
Classification: INTERNAL - CONFIDENTIAL

Table of Contents

Chapter 1: System Architecture Overview

1.1 Technology Stack

HashInsight is an enterprise-grade Bitcoin mining management platform built with modern technology:

Component	Technology	Version	Purpose
Web Framework	Flask	3.0+	Python web application framework
WSGI Server	Gunicorn	21.0+	Production-grade HTTP server
Database	PostgreSQL	15+	Primary data storage (Neon hosted)
Cache	Redis / Memory Cache	7.0+	Performance optimization layer
Blockchain	Web3.py + Base L2	-	Blockchain integration
Encryption	Cryptography	41.0+	Enterprise-grade encryption

1.2 Module Architecture

The system uses a **fully page-isolated architecture** with independently deployed modules:

```
HashInsight Platform
├── Core Application (main.py + app.py)
│   ├── Authentication & Authorization
│   ├── Session Management
│   └── Security Middleware
│
├── Mining Management Module
│   ├── Miner Dashboard
│   ├── Batch Calculator
│   └── Analytics Engine
│
├── CRM & Client Module
│   ├── Customer Management
│   ├── Billing System
│   └── Subscription Management
│
├── Blockchain Integration Module
│   ├── SLA NFT Management
│   ├── Verifiable Computing
│   └── Trust Reconciliation
│
└── Admin & Analytics Module
    ├── Market Data Analysis
```

```
    ├── Performance Monitoring
    └── Reporting System
```

1.3 Enterprise Upgrade Achievements

Security Enhancements

- **KMS Key Management**: Support for AWS KMS, GCP KMS, Azure Key Vault
- **mTLS Mutual Authentication**: Client certificate validation, CRL/OCSP checking
- **API Key System**: Secure keys based on `hsi_dev_key_*` format
- **WireGuard VPN**: Enterprise-grade network isolation
- **Audit Logging**: SOC 2 / PCI DSS / GDPR compliant

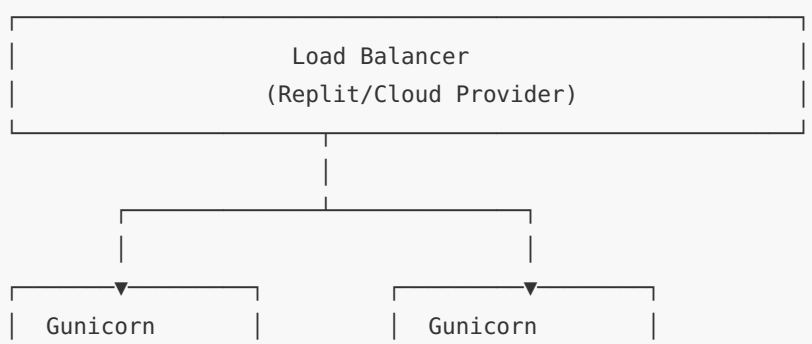
SLO Monitoring

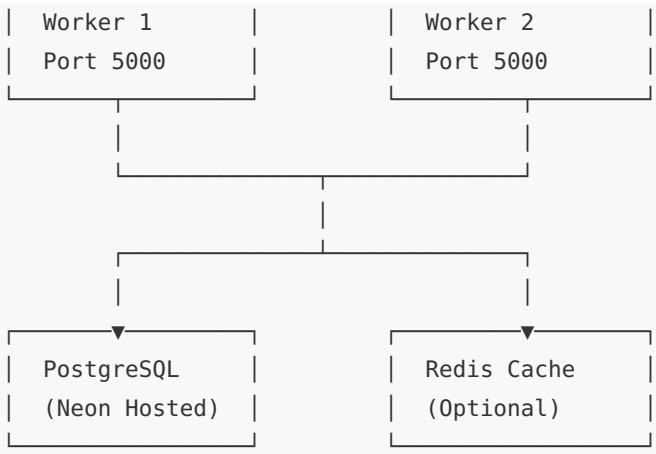
- **Availability**: ≥99.95% (error budget ≤21.6 minutes/month)
- **Latency**: p95 ≤250ms
- **Error Rate**: ≤0.1%
- **Prometheus Metrics**: Comprehensive performance monitoring
- **Circuit Breakers**: Prevent cascading failures

⚡ Performance Optimization

- **Request Coalescing**: 9.8x performance improvement
- **Multi-tier Caching**: Redis + memory cache
- **Connection Pooling**: PostgreSQL connection optimization
- **Batch Processing**: Support for 5000+ concurrent miner imports

1.4 Deployment Topology





Chapter 2: Environment Configuration

2.1 Required Environment Variables

These environment variables **must** be set, or the system will fail to start:

Variable Name	Type	Description	Example
`DATABASE_URL`	String	PostgreSQL connection string	`postgresql://user:pass@host:5432/db`
`SESSION_SECRET`	String	Flask session secret (≥32 chars)	`your-secure-random-secret-key-here`
`ENCRYPTION_PASSWORD`	String	Data encryption master key (≥32 chars)	`encryption-master-key-32-chars-min`

Configuration Example

```
# .env file example
DATABASE_URL=postgresql://hashinsight_user:secure_password@neon-host.us-east-1.aws.neon.tech:5432/hash
SESSION_SECRET=generate_with_python_secrets_token_urlsafe_32
ENCRYPTION_PASSWORD=generate_with_python_secrets_token_urlsafe_32
```

Generate Secure Keys

```
# Generate secure random keys using Python
import secrets
print(f"SESSION_SECRET={secrets.token_urlsafe(32)}")
print(f"ENCRYPTION_PASSWORD={secrets.token_urlsafe(32)}")
```

2.2 Blockchain Integration Configuration

Variable Name	Type	Default	Description
`BLOCKCHAIN_ENABLED`	Boolean	`false`	Enable blockchain features
`BLOCKCHAIN_PRIVATE_KEY`	String	-	Ethereum private key (0x prefix)
`BLOCKCHAIN_NETWORK`	String	`base-sepolia`	Blockchain network
`BASE_RPC_URL`	String	`https://sepolia.base.org`	Base L2 RPC endpoint

```
# Blockchain configuration example
BLOCKCHAIN_ENABLED=true
BLOCKCHAIN_PRIVATE_KEY=0x1234567890abcdef...
BLOCKCHAIN_NETWORK=base-sepolia
```

2.3 Backup System Configuration

Variable Name	Type	Default	Description
`BACKUP_DIR`	String	`/tmp/backups`	Backup storage directory
`BACKUP_ENCRYPTION_KEY`	String	-	Backup encryption key
`BACKUP_RETENTION_DAYS`	Integer	`30`	Backup retention period (days)
`BACKUP_STORAGE_TYPE`	String	`local`	Remote storage type (s3/azure/gcs)
`BACKUP_STORAGE_BUCKET`	String	-	Remote storage bucket name

```
# AWS S3 backup configuration
BACKUP_DIR=/var/backups/hashinsight
BACKUP_ENCRYPTION_KEY=backup-encryption-key-32-chars
BACKUP_RETENTION_DAYS=30
BACKUP_STORAGE_TYPE=s3
BACKUP_STORAGE_BUCKET=hashinsight-backups
BACKUP_STORAGE_REGION=us-east-1
AWS_ACCESS_KEY_ID=AKIAXXXXXXXX
AWS_SECRET_ACCESS_KEY=xxxxxxxxxx
```

2.4 KMS Key Management Configuration

AWS KMS

```
AWS_KMS_KEY_ID=arn:aws:kms:us-east-1:123456789:key/xxxxx
AWS_KMS_REGION=us-east-1
AWS_ACCESS_KEY_ID=AKIAXXXXXXXX
AWS_SECRET_ACCESS_KEY=xxxxxxxxxx
```

GCP KMS

```
GCP_KMS_PROJECT_ID=hashinsight-prod
GCP_KMS_LOCATION=us-east1
GCP_KMS_KEYRING=hashinsight-keyring
```

```
GCP_KMS_KEY_ID=encryption-key
GOOGLE_APPLICATION_CREDENTIALS=/path/to/service-account.json
```

Azure Key Vault

```
AZURE_KEY_VAULT_URL=https://hashinsight-vault.vault.azure.net/
AZURE_TENANT_ID=xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx
AZURE_CLIENT_ID=xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx
AZURE_CLIENT_SECRET=xxxxxxxxxx
```

2.5 Monitoring and Performance Configuration

Variable Name	Type	Default	Description
`ENABLE_BACKGROUND_SERVICES`	Boolean	`false`	Enable background data collection (must explicitly set to '1' to enable)
`PROMETHEUS_PORT`	Integer	`9090`	Prometheus export port
`SLO_MEASUREMENT_WINDOW`	Integer	`30`	SLO measurement window (minutes)

Note: `FAST_STARTUP` mode is hardcoded in `main.py`, not configured through environment variables. See Section 3.4.

2.6 External API Configuration

Variable Name	Description	How to Obtain
`COINWARZ_API_KEY`	CoinWarz mining data API	https://www.coinwarz.com/api
`COINGECKO_API_KEY`	CoinGecko price API	https://www.coingecko.com/api
`SENDGRID_API_KEY`	SendGrid email service	https://sendgrid.com

2.7 Configuration File Locations

```
HashInsight/
├── config.py                    # Main configuration file (single source of truth)
```

```
├── .env                          # Environment variables (not committed to Git)
├── .env.example                  # Environment variable template
└── replit.md                    # System architecture documentation
```

2.8 Configuration Validation

```
# Check required environment variables
python -c "
import os
required = ['DATABASE_URL', 'SESSION_SECRET', 'ENCRYPTION_PASSWORD']
missing = [v for v in required if not os.getenv(v)]
if missing:
    print(f'  Missing: {missing}')
    exit(1)
print('  All required variables set')
"
```

Chapter 3: Deployment Operations Guide

3.1 Startup Commands

Standard Startup (Production Environment)

```
gunicorn --bind 0.0.0.0:5000 --reuse-port --reload main:app
```

Parameter Explanation

Parameter	Description
`--bind 0.0.0.0:5000`	Bind to all network interfaces, port 5000 (Replit required)
`--reuse-port`	Allow multiple workers to bind to same port
`--reload`	Auto-reload on code changes (development)
`--workers 4`	Number of worker processes (CPU cores × 2 + 1)
`--timeout 120`	Worker timeout (seconds)

Complete Production Startup

```
gunicorn \
  --bind 0.0.0.0:5000 \
  --workers 4 \
  --threads 2 \
  --worker-class gthread \
  --timeout 120 \
  --max-requests 1000 \
  --max-requests-jitter 50 \
  --access-logfile - \
  --error-logfile - \
  --log-level info \
  --preload \
  main:app
```

3.2 Port Configuration

⚠ **Important**: Frontend application **must** bind to port 5000

```
# Check port usage
lsof -i :5000

# Force kill occupying process
kill -9 $(lsof -t -i:5000)
```

3.3 Health Checks

The system provides health check endpoints for monitoring:

```
# Basic health check
curl http://localhost:5000/health

# Detailed health check (includes database status)
curl http://localhost:5000/health/detailed

# Expected response
{
  "status": "healthy",
  "timestamp": "2025-10-03T12:00:00Z",
  "database": "connected",
  "cache": "available",
  "version": "2.0.0"
}
```

3.4 Fast Startup Mode

HashInsight's fast startup mode is **hardcoded in** `main.py` as a built-in behavior, not managed through `config.py`.

Implementation Location

```
# Hardcoded implementation in main.py
fast_startup = os.environ.get("FAST_STARTUP", "1").lower() in ("1", "true", "yes")  # Default: enable
skip_db_check = os.environ.get("SKIP_DATABASE_HEALTH_CHECK", "1").lower() in ("1", "true", "yes")  # D
```

Control Method (Optional Environment Variables)

Although these variables are not defined in `config.py`, they can be temporarily adjusted via environment variables:

```
# Enable fast startup (default behavior, no need to set)
export FAST_STARTUP=1
export SKIP_DATABASE_HEALTH_CHECK=1

# Disable fast startup (full initialization)
```

```
export FAST_STARTUP=0
export SKIP_DATABASE_HEALTH_CHECK=0
```

⚠ **Important Note**: - These variables are **not in the `config.py` single source of truth** - They are temporary control switches for the `main.py` startup script - Production configuration should be managed through `config.py`

Fast Startup Behavior

When enabled (default): 1. Main application starts immediately (2-3 seconds) 2. Background services delayed 5 seconds (if `ENABLE_BACKGROUND_SERVICES=1`) 3. Database health checks skipped 4. Suitable for CI/CD rapid deployment

When disabled: 1. Full database health check 2. Synchronous startup of all services 3. Longer startup time (10-15 seconds) 4. Suitable for initial production deployment

3.5 Database Migration

⚠ **Important**: System uses ORM auto-migration, **avoid manual SQL operations**

```
# Database tables created automatically on application startup
# See db.create_all() in app.py

# Manually trigger migration if needed
python -c "from app import app, db; app.app_context().push(); db.create_all()"
```

3.6 Rolling Deployment Strategy

Canary Deployment Process

```
# Step 1: Deploy new version to Canary instance
# Only 1 worker runs new version
gunicorn --bind 0.0.0.0:5001 --workers 1 main:app

# Step 2: Monitor Canary instance (5-10 minutes)
watch -n 5 'curl -s http://localhost:5001/health | jq'

# Step 3: Gradually shift traffic
# Use load balancer to adjust weights: old(90%) -> new(10%)
# Monitor error rate and latency

# Step 4: Full cutover
```

```
# old(0%) -> new(100%)
killall -9 gunicorn  # Stop old version
gunicorn --bind 0.0.0.0:5000 --workers 4 main:app  # Start new version
```

3.7 Graceful Shutdown

```
# Send SIGTERM signal (graceful shutdown)
kill -TERM $(cat /var/run/gunicorn.pid)

# Wait 30 seconds to process existing requests
sleep 30

# Force stop (if still running)
kill -KILL $(cat /var/run/gunicorn.pid)
```

3.8 Log Management

```
# Enable structured logging at startup
export LOG_LEVEL=INFO
export LOG_FORMAT=json

# View real-time logs
tail -f /var/log/hashinsight/app.log

# View error logs
grep ERROR /var/log/hashinsight/app.log | tail -20

# Replit environment logs
# Logs output to stdout, view through Replit Console
```

3.9 Deployment Checklist

- [ ] Environment variables configured (DATABASE_URL, SESSION_SECRET, ENCRYPTION_PASSWORD)
- [ ] Database connection working
- [ ] Port 5000 available
- [ ] Health check endpoint returns 200
- [ ] Audit log directory writable (`logs/audit.jsonl`)
- [ ] Backup directory created

- [ ] SSL certificates valid (if mTLS enabled)
- [ ] KMS keys accessible (if enabled)
- [ ] Prometheus metrics accessible (`:9090/metrics`)

Chapter 4: Monitoring & Alerting

4.1 SLO Definitions

HashInsight follows strict SLO standards:

Availability SLO

Metric	Target	Error Budget	Measurement Period
Availability	≥99.95%	≤21.6 minutes/month	30-day rolling
Success Rate	≥99.9%	≤43.2 minutes/month	30-day rolling

Latency SLO

Percentile	Target	Measurement Window
P50	≤100ms	5 minutes
P95	≤250ms	5 minutes
P99	≤500ms	5 minutes

Error Rate SLO

Type	Target	Threshold
4xx errors	≤1%	Warning
5xx errors	≤0.1%	Critical

4.2 Prometheus Metrics

System Metrics Export

```
# monitoring/prometheus_exporter.py
from prometheus_client import Counter, Histogram, Gauge

# Request count
request_count = Counter(
    'hashinsight_requests_total',
    'Total request count',
    ['method', 'endpoint', 'status']
)

# Request latency
request_latency = Histogram(
    'hashinsight_request_latency_seconds',
    'Request latency',
    ['method', 'endpoint'],
    buckets=[0.01, 0.05, 0.1, 0.25, 0.5, 1.0, 2.5, 5.0]
)

# Cache hit rate
cache_hit_rate = Gauge(
    'hashinsight_cache_hit_rate',
    'Cache hit rate percentage'
)

# Database query duration
db_query_duration = Histogram(
    'hashinsight_db_query_duration_seconds',
    'Database query duration',
    ['query_type'],
    buckets=[0.001, 0.01, 0.05, 0.1, 0.5, 1.0]
)

# SLO compliance
slo_compliance = Gauge(
    'hashinsight_slo_compliance',
    'SLO compliance percentage',
    ['slo_type']
)
```

Metrics Access

```
# View Prometheus metrics
curl http://localhost:9090/metrics

# Example output
# HELP hashinsight_requests_total Total request count
# TYPE hashinsight_requests_total counter
hashinsight_requests_total{method="GET",endpoint="/dashboard",status="200"} 1234

# HELP hashinsight_request_latency_seconds Request latency
# TYPE hashinsight_request_latency_seconds histogram
hashinsight_request_latency_seconds_bucket{method="GET",endpoint="/api/miners",le="0.1"} 450
hashinsight_request_latency_seconds_bucket{method="GET",endpoint="/api/miners",le="0.25"} 480
```

4.3 Grafana Dashboard

Core Monitoring Panels

```
{
  "dashboard": {
    "title": "HashInsight Production Monitoring",
    "panels": [
      {
        "title": "Request Rate",
        "targets": [{
          "expr": "rate(hashinsight_requests_total[5m])"
        }]
      },
      {
        "title": "P95 Latency",
        "targets": [{
          "expr": "histogram_quantile(0.95, rate(hashinsight_request_latency_seconds_bucket[5m]))"
        }]
      },
      {
        "title": "Error Rate",
        "targets": [{
          "expr": "rate(hashinsight_requests_total{status=~\"5..\"}[5m]) / rate(hashinsight_requests_
        }]
      },
      {
        "title": "SLO Compliance",
        "targets": [{
```

```
          "expr": "hashinsight_slo_compliance"
        }]
      }
    ]
  }
}
```

4.4 Alert Rules

Prometheus Alert Rules

```
# prometheus/alerts.yml
groups:
  - name: hashinsight_alerts
    interval: 30s
    rules:
      # Availability alert
      - alert: HighErrorRate
        expr: |
          rate(hashinsight_requests_total{status=~"5.."}[5m])
          / rate(hashinsight_requests_total[5m]) > 0.01
        for: 5m
        labels:
          severity: critical
        annotations:
          summary: "High 5xx error rate detected"
          description: "Error rate is {{ $value | humanizePercentage }}"

      # Latency alert
      - alert: HighLatency
        expr: |
          histogram_quantile(0.95,
            rate(hashinsight_request_latency_seconds_bucket[5m])
          ) > 0.25
        for: 10m
        labels:
          severity: warning
        annotations:
          summary: "P95 latency exceeds SLO"
          description: "P95 latency is {{ $value }}s (SLO: 0.25s)"

      # SLO error budget alert
      - alert: ErrorBudgetExhausted
        expr: hashinsight_slo_error_budget_remaining < 0.1
        for: 5m
```

```
        labels:
          severity: critical
        annotations:
          summary: "SLO error budget nearly exhausted"
          description: "Only {{ $value | humanizePercentage }} budget remaining"

      # Database connection alert
      - alert: DatabaseConnectionFailure
        expr: hashinsight_db_connection_status == 0
        for: 2m
        labels:
          severity: critical
        annotations:
          summary: "Database connection failure"
          description: "Unable to connect to PostgreSQL"

      # Cache hit rate alert
      - alert: LowCacheHitRate
        expr: hashinsight_cache_hit_rate < 50
        for: 15m
        labels:
          severity: warning
        annotations:
          summary: "Low cache hit rate"
          description: "Cache hit rate is {{ $value }}%"
```

4.5 Circuit Breaker Configuration

HashInsight uses circuit breaker pattern to prevent cascading failures:

```
# monitoring/circuit_breaker.py
from monitoring.circuit_breaker import CircuitBreaker, circuit_breaker

# Database query circuit breaker
db_breaker = CircuitBreaker(
    failure_threshold=5,      # Trigger after 5 consecutive failures
    recovery_timeout=60,      # Attempt recovery after 60 seconds
    name="database_queries"
)

# API call circuit breaker
@circuit_breaker(
    failure_threshold=3,
    recovery_timeout=30,
    name="external_api"
)
```

```
def call_external_api():
    response = requests.get("https://api.coinwarz.com/...")
    return response.json()
```

Circuit Breaker Status Monitoring

```
# View circuit breaker status
curl http://localhost:5000/api/circuit-breakers

# Response example
{
  "database_queries": {
    "state": "closed",
    "failure_count": 0,
    "total_calls": 1234,
    "success_rate": "99.8%"
  },
  "external_api": {
    "state": "half_open",
    "failure_count": 3,
    "total_calls": 456,
    "success_rate": "95.2%"
  }
}
```

4.6 Alert Notifications

Slack Integration

```
# Environment variable configuration
SLACK_WEBHOOK_URL=https://hooks.slack.com/services/T00/B00/xxxxx
ALERT_SLACK_CHANNEL=#hashinsight-alerts
```

PagerDuty Integration

```
PAGERDUTY_API_KEY=xxxxx
PAGERDUTY_SERVICE_KEY=xxxxx
```

4.7 Monitoring Checklist

- [ ] Prometheus scraping metrics (`:9090/targets`)
- [ ] Grafana dashboard displaying data
- [ ] Alert rules loaded
- [ ] Slack/PagerDuty notifications working
- [ ] SLO monitoring panel shows green
- [ ] Circuit breaker status normal

Chapter 5: Backup & Recovery

5.1 Automated Backup Strategy

HashInsight uses `backup/backup_manager.py` for automated backups:

Backup Features

- **PostgreSQL full backup** (pg_dump)
- **AES-256 encryption** (backup file encryption)
- **gzip compression** (save storage space)
- **Remote storage** (S3/Azure/GCS support)
- **Integrity verification** (SHA256 checksum)

Backup Scheduling

```
# Configure automatic backups via cron
# /etc/cron.d/hashinsight-backup

# Daily full backup at 2 AM
0 2 * * * /usr/bin/python3 /app/backup/backup_manager.py --type full

# Incremental backup every 4 hours
0 */4 * * * /usr/bin/python3 /app/backup/backup_manager.py --type incremental

# Upload to remote storage every Sunday at 3 AM
0 3 * * 0 /usr/bin/python3 /app/backup/backup_manager.py --upload
```

5.2 Manual Backup

```
# Execute full backup
python backup/backup_manager.py

# Backup output example
  Backup created successfully: hashinsight_backup_20251003_140000.sql.gz.enc
  Size: 245.3 MB
  Encrypted: AES-256
  Checksum: e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855
   Duration: 45.2s
```

5.3 Backup Retention Policy

Backup Type	Retention Period	Frequency
Full backup	30 days	Daily
Incremental backup	7 days	Every 4 hours
Weekly backup	12 weeks	Every Sunday
Monthly backup	12 months	1st of month

Automatic Cleanup of Old Backups

```
# Clean up backups older than 30 days
python backup/backup_manager.py --cleanup --days 30

# Output
   Deleting old backups: hashinsight_backup_20250903_*.sql.gz.enc
  Cleanup complete: 2.1 GB freed
```

5.4 RTO/RPO Targets

Metric	Target	Actual
RTO (Recovery Time Objective)	≤4 hours	~2 hours
RPO (Recovery Point Objective)	≤15 minutes	~4 hours

5.5 Backup Recovery Process

Step 1: List Available Backups

```
# List local backups
ls -lh /tmp/backups/

# List remote backups (S3)
aws s3 ls s3://hashinsight-backups/
```

Step 2: Download Backup (if remote)

```
# Download from S3
aws s3 cp s3://hashinsight-backups/hashinsight_backup_20251003_020000.sql.gz.enc /tmp/restore/

# Download from Azure
az storage blob download \
  --account-name hashinsight \
  --container-name backups \
  --name hashinsight_backup_20251003_020000.sql.gz.enc \
  --file /tmp/restore/backup.sql.gz.enc
```

Step 3: Decrypt Backup

```
# Decrypt using backup_manager
python backup/backup_manager.py \
  --decrypt /tmp/restore/hashinsight_backup_20251003_020000.sql.gz.enc \
  --output /tmp/restore/backup.sql.gz

# Manual decryption (if needed)
openssl enc -d -aes-256-cbc \
  -in hashinsight_backup_20251003_020000.sql.gz.enc \
```

```
  -out backup.sql.gz \
  -pass env:BACKUP_ENCRYPTION_KEY
```

Step 4: Decompress Backup

```
# Decompress
gunzip /tmp/restore/backup.sql.gz

# Verify file integrity
sha256sum /tmp/restore/backup.sql
```

Step 5: Restore Database

```
# ⚠ Warning: This will overwrite current database!

# 1. Stop application
systemctl stop hashinsight

# 2. Create emergency backup of current state
python backup/backup_manager.py --type emergency

# 3. Restore from backup
psql $DATABASE_URL < /tmp/restore/backup.sql

# 4. Verify restoration
psql $DATABASE_URL -c "SELECT COUNT(*) FROM miners;"

# 5. Restart application
systemctl start hashinsight
```

5.6 Disaster Recovery Plan

Complete System Recovery Steps

```
# 1. Prepare new environment
export DATABASE_URL="new_database_url"
export SESSION_SECRET="new_session_secret"
export ENCRYPTION_PASSWORD="new_encryption_password"

# 2. Download latest backup
python backup/backup_manager.py --download-latest
```

```
# 3. Restore database
python backup/backup_manager.py --restore-latest

# 4. Deploy application
git clone https://github.com/hashinsight/platform.git
cd platform
pip install -r requirements.txt

# 5. Verify configuration
python scripts/verify_setup.py

# 6. Start application
gunicorn --bind 0.0.0.0:5000 --workers 4 main:app

# 7. Verify service health
curl http://localhost:5000/health
```

5.7 Backup Testing

```
# Regular backup restoration testing (quarterly)
# 1. Create test environment
docker run -d --name test-postgres -e POSTGRES_PASSWORD=test postgres:15

# 2. Restore to test environment
export TEST_DATABASE_URL="postgresql://postgres:test@localhost:5432/test_db"
psql $TEST_DATABASE_URL < /tmp/backups/latest_backup.sql

# 3. Verify data integrity
python scripts/verify_backup_integrity.py --database $TEST_DATABASE_URL

# 4. Cleanup test environment
docker stop test-postgres
docker rm test-postgres
```

5.8 Backup Checklist

- [ ] Daily automated backups running
- [ ] Backup encryption enabled
- [ ] Remote storage configured
- [ ] Backup integrity verification passing
- [ ] Restoration tested within last 90 days
- [ ] Old backups automatically cleaned up

- [ ] Backup monitoring alerts configured
- [ ] Documentation up to date

Chapter 6: Security Operations

6.1 KMS Key Management

HashInsight supports enterprise-grade Key Management Systems (KMS):

Supported KMS Providers

- **AWS KMS**: Amazon Web Services Key Management Service
- **GCP KMS**: Google Cloud Platform Key Management Service
- **Azure Key Vault**: Microsoft Azure Key Vault

AWS KMS Configuration

```
# Configure AWS KMS
from security.kms_client import KMSClient, KMSProvider, EncryptionContext

kms_config = {
    'key_id': 'arn:aws:kms:us-east-1:123456789:key/xxxxx',
    'region': 'us-east-1'
}

client = KMSClient(KMSProvider.AWS_KMS, kms_config)

# Encrypt data
context = EncryptionContext(
    purpose="user_data_encryption",
    tenant_id="tenant_123"
)

ciphertext = client.encrypt_secret(
    plaintext="sensitive data",
    key_id=kms_config['key_id'],
    context=context
)
```

GCP KMS Configuration

```
# Configure GCP KMS
kms_config = {
    'project_id': 'hashinsight-prod',
    'location': 'us-east1',
    'keyring': 'hashinsight-keyring',
    'key_id': 'encryption-key'
}

client = KMSClient(KMSProvider.GCP_KMS, kms_config)
```

Azure Key Vault Configuration

```
# Configure Azure Key Vault
kms_config = {
    'vault_url': 'https://hashinsight-vault.vault.azure.net/',
    'key_name': 'encryption-key'
}

client = KMSClient(KMSProvider.AZURE_KEY_VAULT, kms_config)
```

Key Rotation Process

```
# 1. Create new key version in KMS
aws kms create-key --description "HashInsight Master Key v2"

# 2. Update application configuration
export AWS_KMS_KEY_ID=arn:aws:kms:us-east-1:123456789:key/new-key-id

# 3. Re-encrypt existing data (background task)
python scripts/rotate_encryption_keys.py --old-key OLD_KEY_ID --new-key NEW_KEY_ID

# 4. Verify new key
python scripts/verify_encryption.py --key-id NEW_KEY_ID

# 5. Disable old key (retain for 90 days)
aws kms disable-key --key-id OLD_KEY_ID
```

6.2 mTLS Mutual Authentication

Certificate Generation

```
# 1. Generate CA certificate
openssl genrsa -out ca.key 4096
openssl req -new -x509 -days 3650 -key ca.key -out ca.crt \
  -subj "/C=US/O=HashInsight/CN=HashInsight Root CA"

# 2. Generate server certificate
openssl genrsa -out server.key 4096
openssl req -new -key server.key -out server.csr \
  -subj "/C=US/O=HashInsight/CN=*.hashinsight.net"
openssl x509 -req -days 365 -in server.csr -CA ca.crt -CAkey ca.key \
  -CAcreateserial -out server.crt

# 3. Generate client certificate
openssl genrsa -out client.key 4096
openssl req -new -key client.key -out client.csr \
  -subj "/C=US/O=HashInsight/CN=client.hashinsight.net"
openssl x509 -req -days 365 -in client.csr -CA ca.crt -CAkey ca.key \
  -CAcreateserial -out client.crt
```

mTLS Configuration

```
# Environment variable configuration
export MTLS_ENABLED=true
export MTLS_CA_CERT_PATH=/app/certs/ca.crt
export MTLS_SERVER_CERT_PATH=/app/certs/server.crt
export MTLS_SERVER_KEY_PATH=/app/certs/server.key
export MTLS_VERIFY_CLIENT=true
export MTLS_ALLOWED_DN_PATTERNS="CN=*.hashinsight.net,O=HashInsight,C=US"
```

Using mTLS Authentication

```
from common.mtls_auth import require_mtls

@app.route('/api/admin/sensitive')
@require_mtls()
def sensitive_endpoint():
    # Only allow requests with valid client certificates
```

```
    client_dn = g.client_cert_subject
    return jsonify({"message": f"Authenticated as {client_dn}"})
```

6.3 API Key Management

Key Format

HashInsight API key format: `hsi_{env}_key_{random}`

Examples: - Production: `hsi_prod_key_a1b2c3d4e5f6g7h8` - Development: `hsi_dev_key_x9y8z7w6v5u4t3s2`

Create API Key

```
# Create using management tool
python scripts/create_api_key.py \
  --user-id 123 \
  --permissions "miners:read,miners:write" \
  --expires-in 90

# Output
  API key created
Key: hsi_prod_key_a1b2c3d4e5f6g7h8
User ID: 123
Permissions: miners:read,miners:write
Expires: 2025-12-31T23:59:59Z
⚠  Please store this key securely, it will not be shown again
```

API Key Rotation

```
# 1. Create new key
new_key=$(python scripts/create_api_key.py --user-id 123 --copy-from old_key_id)

# 2. Update client configuration (7-day dual-key period)
# Both old and new keys valid simultaneously

# 3. Verify new key
curl -H "Authorization: Bearer $new_key" http://localhost:5000/api/miners

# 4. Revoke old key
python scripts/revoke_api_key.py --key-id old_key_id
```

6.4 WireGuard Enterprise Network

Hub Server Deployment

```
# 1. Run Hub installation script
sudo bash wireguard/hub_setup.sh

# 2. Configure firewall
sudo ufw allow 51820/udp
sudo ufw enable

# 3. Start WireGuard
sudo systemctl enable wg-quick@wg0
sudo systemctl start wg-quick@wg0

# 4. Check status
sudo wg show
```

Site Gateway Configuration

```
# 1. Generate site keys
cd wireguard/site-gateway
python key_manager.py --generate-keys --site beijing-dc1

# 2. Add to Hub configuration
sudo nano /etc/wireguard/wg0.conf

# Add Peer configuration
[Peer]
PublicKey = site_public_key
AllowedIPs = 10.8.1.0/24
Endpoint = beijing-gateway.hashinsight.net:51820

# 3. Reload configuration
sudo wg-quick down wg0
sudo wg-quick up wg0
```

6.5 Audit Logging

HashInsight logs all critical operations to audit logs (`audit/audit_logger.py`):

Audit Event Types

- Authentication (login/logout/failure)
- Data access (CRUD operations)
- Configuration changes
- Permission changes
- Encryption operations
- API key management
- Suspicious activities

Audit Log Format

```
{
  "timestamp": "2025-10-03T12:00:00.000Z",
  "event_id": "a1b2c3d4e5f6g7h8",
  "level": "INFO",
  "category": "authentication",
  "action": "login",
  "user_id": "123",
  "user_email": "user@example.com",
  "user_role": "admin",
  "ip_address": "192.168.1.100",
  "status": "success",
  "details": {
    "login_method": "password",
    "two_factor": true
  }
}
```

Query Audit Logs

```
# View last 100 audit logs
tail -100 logs/audit.jsonl

# Query specific user operations
jq 'select(.user_email == "admin@hashinsight.net")' logs/audit.jsonl

# Query failed login attempts
jq 'select(.action == "login_failed")' logs/audit.jsonl

# Query security events from last 24 hours
jq --arg date "$(date -d '24 hours ago' -Iseconds)" \
```

```
  'select(.timestamp > $date and .level == "SECURITY")' \
  logs/audit.jsonl
```

6.6 Compliance Requirements

HashInsight follows these compliance standards:

SOC 2 Type II

- Access control
- Encrypted transmission (TLS 1.3)
- Encrypted storage (AES-256)
- Audit logging (immutable)
- Change management
- Disaster recovery

PCI DSS (if processing payments)

- Sensitive data masking
- Key management (KMS)
- Network isolation (WireGuard)
- Regular penetration testing

GDPR

- Data minimization
- User data export
- Data deletion (right to be forgotten)
- Data processing records

6.7 Security Checklist

- [ ] All keys stored in KMS (not in code/config files)
- [ ] mTLS enabled (production environment)
- [ ] API keys rotated regularly (90 days)
- [ ] SSL/TLS certificates valid and not expired
- [ ] Audit logs writing normally
- [ ] No sensitive information leaked to logs

- [ ] Database connections encrypted (SSL)
- [ ] Session key strength ≥256 bits
- [ ] Regular security scans (quarterly)
- [ ] Penetration testing report (annually)

Chapter 7: Troubleshooting Guide

7.1 Application Won't Start

Symptoms

```
$ gunicorn main:app
[ERROR] Application failed to start
```

Diagnostic Steps

1. Check Port Usage

```
# Check port 5000 usage
lsof -i :5000

# If occupied, kill process
kill -9 $(lsof -t -i:5000)
```

2. Check Environment Variables

```
# Verify required variables
python3 << 'EOF'
import os
required = ['DATABASE_URL', 'SESSION_SECRET', 'ENCRYPTION_PASSWORD']
for var in required:
    value = os.getenv(var)
    if not value:
        print(f"  Missing: {var}")
    else:
        print(f"  {var}: {'*' * 8} (set)")
EOF
```

3. Check Database Connection

```
# Test PostgreSQL connection
psql $DATABASE_URL -c "SELECT version();"

# If fails, check Neon endpoint status
# Visit https://console.neon.tech
```

4. Check Python Dependencies

```
# Verify key libraries
python3 -c "
import flask
import gunicorn
import psycopg2
import sqlalchemy
print('  All dependencies OK')
"
```

5. View Detailed Errors

```
# Enable debug mode
export FLASK_DEBUG=1
python3 main.py

# View full stack trace
```

Common Errors and Solutions

Error	Cause	Solution
`ModuleNotFoundError: No module named 'flask'`	Dependencies not installed	`pip install -r requirements.txt`
`OperationalError: could not connect to server`	Database connection failed	Check DATABASE_URL, verify Neon endpoint
`ValueError: SECRET_KEY must be set`	SESSION_SECRET not set	`export SESSION_SECRET=$(python -c 'import secrets; print(secrets.token_urlsafe(32))')`
`Address already in use`	Port occupied	`kill -9 $(lsof -t -i:5000)`

7.2 Database Connection Failure

Symptoms

```
sqlalchemy.exc.OperationalError: (psycopg2.OperationalError)
could not connect to server: Connection timed out
```

Diagnostic Steps

1. Verify Connection String

```
# Parse DATABASE_URL
python3 << 'EOF'
import os
from urllib.parse import urlparse
url = os.getenv('DATABASE_URL')
parsed = urlparse(url)
print(f"Host: {parsed.hostname}")
print(f"Port: {parsed.port}")
print(f"Database: {parsed.path[1:]}")
print(f"User: {parsed.username}")
EOF
```

2. Test Network Connectivity

```
# Extract host and port
export DB_HOST=$(python3 -c "from urllib.parse import urlparse; import os; print(urlparse(os.getenv('D
export DB_PORT=$(python3 -c "from urllib.parse import urlparse; import os; print(urlparse(os.getenv('D

# Test TCP connection
nc -zv $DB_HOST $DB_PORT
```

3. Check Connection Pool

```
# Connection pool configuration (config.py)
SQLALCHEMY_ENGINE_OPTIONS = {
    'pool_size': 10,           # Default 10 connections
    'pool_recycle': 300,       # Recycle after 5 minutes
    'pool_pre_ping': True,     # Test connection before use
    'pool_timeout': 30,        # 30 second timeout
    'max_overflow': 20,        # Max 20 overflow connections
    'connect_args': {
        'connect_timeout': 15  # 15 second connect timeout
    }
}
```

4. Check Neon Endpoint Status

```
# Visit Neon Console
# https://console.neon.tech/app/projects

# Check endpoint status:
# - Active (green) - Normal
# - Idle (yellow) - Needs wake-up
# - Suspended (gray) - Suspended
```

5. Enable Connection Retry

```
# Add retry logic in app.py
from sqlalchemy import event, exc
from sqlalchemy.pool import Pool

@event.listens_for(Pool, "connect")
def receive_connect(dbapi_conn, connection_record):
    connection_record.info['pid'] = os.getpid()

@event.listens_for(Pool, "checkout")
def receive_checkout(dbapi_conn, connection_record, connection_proxy):
```

```
    pid = os.getpid()
    if connection_record.info['pid'] != pid:
        connection_record.connection = connection_proxy.connection = None
        raise exc.DisconnectionError(
            "Connection record belongs to pid %s, "
            "attempting to check out in pid %s" %
            (connection_record.info['pid'], pid)
        )
```

7.3 Cache Issues

Symptoms

- Response time significantly increased
- Cache hit rate below 50%
- Redis connection errors

Diagnostic Steps

1. Check Redis Connection

```
# Test Redis
redis-cli ping
# Expected: PONG

# Check Redis memory
redis-cli info memory

# Check key count
redis-cli dbsize
```

2. View Cache Hit Rate

```
# Query via API
curl http://localhost:5000/api/cache/stats

# Expected response
{
  "cache_type": "redis",
  "hit_rate": 75.5,
  "total_requests": 10000,
  "hits": 7550,
```

```
  "misses": 2450
}
```

3. Cache Fallback Mechanism

```
# cache_manager.py automatic fallback
if redis_available:
    cache_backend = RedisCache()
else:
    logger.warning("Redis unavailable, falling back to memory cache")
    cache_backend = MemoryCache()
```

4. Clear Cache

```
# Clear all cache
redis-cli FLUSHDB

# Clear specific prefix
redis-cli --scan --pattern 'hashinsight:*' | xargs redis-cli DEL
```

7.4 Performance Degradation

Symptoms

- API response time p95 > 250ms
- Slow database queries
- High CPU/memory usage

Diagnostic Steps

1. Check Request Coalescing Status

```
# View request merging statistics
curl http://localhost:5000/api/performance/coalescing-stats

# Expected response
{
  "enabled": true,
  "performance_improvement": "9.8x",
  "deduplicated_requests": 5432,
```

```
  "total_requests": 53210
}
```

2. Analyze Slow Queries

```
-- Enable slow query log (PostgreSQL)
ALTER DATABASE hashinsight_db SET log_min_duration_statement = 1000;

-- Query slow queries
SELECT query, calls, total_time, mean_time
FROM pg_stat_statements
WHERE mean_time > 1000
ORDER BY mean_time DESC
LIMIT 10;
```

3. Check Database Indexes

```
# Run index optimization
python database/optimize_indexes.py --analyze

# View missing indexes
python database/optimize_indexes.py --suggest
```

4. Monitor System Resources

```
# CPU usage
top -b -n 1 | grep gunicorn

# Memory usage
ps aux | grep gunicorn | awk '{sum+=$6} END {print sum/1024 " MB"}'

# Database connections
psql $DATABASE_URL -c "SELECT count(*) FROM pg_stat_activity WHERE datname = 'hashinsight_db';"
```

5. Enable Performance Profiling

```
# Add to routes requiring profiling
from werkzeug.middleware.profiler import ProfilerMiddleware

app.wsgi_app = ProfilerMiddleware(
    app.wsgi_app,
    restrictions=[10],
```

```
    profile_dir='./profiles'
)
```

7.5 Blockchain Integration Errors

Symptoms

```
ERROR:blockchain_integration: Encryption configuration error: ENCRYPTION_PASSWORD environment variable
ERROR:sla_nft_routes: Failed to get SLA status
```

Diagnostic Steps

1. Check ENCRYPTION_PASSWORD

```
# Verify environment variable
echo $ENCRYPTION_PASSWORD

# If not set
export ENCRYPTION_PASSWORD=$(python3 -c 'import secrets; print(secrets.token_urlsafe(32))')
```

2. Check Blockchain Configuration

```
# Verify blockchain environment variables
python3 << 'EOF'
import os
config = {
    'BLOCKCHAIN_ENABLED': os.getenv('BLOCKCHAIN_ENABLED'),
    'BLOCKCHAIN_PRIVATE_KEY': '***' if os.getenv('BLOCKCHAIN_PRIVATE_KEY') else None,
    'BLOCKCHAIN_NETWORK': os.getenv('BLOCKCHAIN_NETWORK', 'base-sepolia'),
    'BASE_RPC_URL': os.getenv('BASE_RPC_URL', 'https://sepolia.base.org')
}
for k, v in config.items():
    status = ' ' if v else ' '
    print(f"{status} {k}: {v}")
EOF
```

3. Test Web3 Connection

```
python3 << 'EOF'
from web3 import Web3
import os
```

```
rpc_url = os.getenv('BASE_RPC_URL', 'https://sepolia.base.org')
w3 = Web3(Web3.HTTPProvider(rpc_url))

if w3.is_connected():
    print(f"  Web3 connected to {rpc_url}")
    print(f"Block number: {w3.eth.block_number}")
else:
    print(f"  Web3 connection failed")
EOF
```

4. Verify Private Key Format

```
# Private key should start with 0x, followed by 64 hex characters
python3 << 'EOF'
import os
import re

private_key = os.getenv('BLOCKCHAIN_PRIVATE_KEY', '')
if re.match(r'^0x[0-9a-fA-F]{64}$', private_key):
    print("  Private key format valid")
else:
    print("  Invalid private key format")
    print("Expected: 0x + 64 hex characters")
EOF
```

7.6 Log Locations

Log Type	Path	Format
Application logs	stdout	Text
Audit logs	`logs/audit.jsonl`	JSON Lines
Error logs	stderr	Text
Gunicorn logs	`/var/log/gunicorn/`	Text
PostgreSQL logs	Neon Console	Text
Workflow logs	`/tmp/logs/`	Text

View Logs

```
# Real-time application logs
tail -f /var/log/hashinsight/app.log

# View error logs
grep ERROR /var/log/hashinsight/app.log | tail -50

# View audit logs
tail -f logs/audit.jsonl | jq '.'

# View Gunicorn access logs
tail -f /var/log/gunicorn/access.log
```

Chapter 8: Daily Operations

8.1 Database Maintenance

8.1.1 Index Optimization

```
# Automatically optimize indexes
python database/optimize_indexes.py --auto

# Analyze and suggest indexes
python database/optimize_indexes.py --analyze --suggest

# Output example:
#   Existing indexes: 45
#   Scanning slow queries...
#   Suggested indexes:
#   - CREATE INDEX idx_miners_user_id ON miners(user_id)
#   - CREATE INDEX idx_market_data_timestamp ON market_analytics(created_at DESC)
```

8.1.2 Connection Pool Monitoring

```
-- View current connections
SELECT
    pid,
    usename,
```

```
    application_name,
    client_addr,
    state,
    query_start
FROM pg_stat_activity
WHERE datname = 'hashinsight_db'
ORDER BY query_start;

-- Kill idle connections
SELECT pg_terminate_backend(pid)
FROM pg_stat_activity
WHERE datname = 'hashinsight_db'
  AND state = 'idle'
  AND state_change < NOW() - INTERVAL '30 minutes';
```

8.1.3 Slow Query Analysis

```
-- Enable pg_stat_statements
CREATE EXTENSION IF NOT EXISTS pg_stat_statements;

-- View Top 10 slow queries
SELECT
    substring(query, 1, 100) AS short_query,
    calls,
    total_time,
    mean_time,
    max_time
FROM pg_stat_statements
WHERE query NOT LIKE '%pg_stat_statements%'
ORDER BY mean_time DESC
LIMIT 10;

-- Reset statistics
SELECT pg_stat_statements_reset();
```

8.1.4 Database Cleanup

```
# Clean old data (retain 90 days)
python scripts/cleanup_old_data.py --days 90

# Cleanup example:
#   Deleted market_analytics older than 90 days: 12,543 records
```

```
#   Deleted audit logs older than 90 days: 45,123 records
#   Cleanup complete, freed space: 2.3 GB
```

8.2 Cache Management

8.2.1 Redis Cache Cleanup

```
# Clear all HashInsight cache
redis-cli --scan --pattern 'hashinsight:*' | xargs redis-cli DEL

# Clear specific module cache
redis-cli --scan --pattern 'hashinsight:miners:*' | xargs redis-cli DEL

# Clear expired keys (Redis automatic, manual trigger)
redis-cli --scan --pattern 'hashinsight:*' | while read key; do
    redis-cli TTL "$key"
done
```

8.2.2 Memory Cache Monitoring

```
# Query memory cache status via API
curl http://localhost:5000/api/cache/memory-stats

# Response
{
  "cache_type": "memory",
  "size_mb": 124.5,
  "entries": 5432,
  "hit_rate": 82.3,
  "evictions": 234
}
```

8.2.3 Request Coalescer Status

```
# View request merging statistics
curl http://localhost:5000/api/performance/coalescing-stats | jq

# Manually clear coalescing cache
curl -X POST http://localhost:5000/api/performance/clear-coalescing-cache
```

8.3 Batch Tasks

8.3.1 Bulk Import Miners

```
# Prepare CSV file (max 5000 miners)
# Format: name,model,hashrate_th,power_w,efficiency

# Execute bulk import
python batch/batch_import_manager.py \
  --file miners_upload_5000.csv \
  --user-id 123 \
  --validate

# Output:
#   File validation passed
#   Record count: 5000
# ⚙  Processing...
# [████████████████████████████████] 100%
#   Import complete: 5000 miners (Duration: 45.2s)
```

8.3.2 Data Collection Tasks

HashInsight automatically collects market data every 15 minutes:

```
# Manually trigger data collection
python modules/analytics/engines/analytics_engine.py --collect-now

# View collection status
curl http://localhost:5000/api/analytics/collection-status

# Response
{
  "last_collection": "2025-10-03T12:15:00Z",
  "next_collection": "2025-10-03T12:30:00Z",
  "status": "healthy",
  "data_points_today": 8
}
```

8.3.3 Data Collection Scheduling

```
# Configure data collection frequency (config.py)
ANALYTICS_COLLECTION_INTERVAL = 15  # minutes
ANALYTICS_MAX_DATA_POINTS_PER_DAY = 10  # Daily limit
```

```
# Enable/disable background services (default: disabled, i.e., 0)
export ENABLE_BACKGROUND_SERVICES=1  # Enable background data collection
export ENABLE_BACKGROUND_SERVICES=0  # Disable background data collection (default)
```

8.4 User Management

8.4.1 Create User

```
# Create via admin interface
# Visit: http://localhost:5000/admin/users/create

# Create via command line
python scripts/create_user.py \
  --email admin@hashinsight.net \
  --username admin \
  --role owner \
  --password-prompt

# Output:
# Please enter password: ********
#   User created
# ID: 123
# Email: admin@hashinsight.net
# Role: owner
```

8.4.2 Role Assignment

Role	Permissions	Description
`owner`	All permissions	System owner
`admin`	Management permissions	System administrator
`broker`	Broker permissions	Customer management, orders
`client`	Client permissions	View own data

```
# Change user role
python scripts/change_user_role.py \
  --user-id 123 \
```

```
  --new-role admin

# Bulk import users
python scripts/bulk_import_users.py --file users.csv
```

8.4.3 Permission Management

```
# Check user permissions
from decorators import has_permission

@app.route('/api/miners/delete/<int:miner_id>', methods=['DELETE'])
@login_required
@requires_permission('miners:delete')
def delete_miner(miner_id):
    # Only allow users with miners:delete permission
    pass
```

8.5 Data Cleanup

8.5.1 Daily Data Point Limits

To control storage costs, HashInsight limits daily data points:

```
# config.py
ANALYTICS_MAX_DATA_POINTS_PER_DAY = 10  # Max 10 data points per day
```

8.5.2 Historical Data Archival

```
# Archive data older than 90 days to cold storage
python scripts/archive_historical_data.py --days 90 --storage s3

# Output:
#   Scanning data for archival...
#   Archived 45,234 records to S3
#   Deleted archived data from primary database
#    Duration: 12m 34s
```

8.6 Monitoring Tasks

8.6.1 Daily Health Check

```
# Run daily health check script
python scripts/daily_health_check.py

# Output:
#   Application: Healthy
#   Database: Connected, 15 active connections
#   Cache: Redis available, 78.5% hit rate
#   Disk space: 234 GB available (85% free)
#   Backups: Latest backup 2 hours ago
#   SSL certificates: Valid, expires in 45 days
# ⚠  Warning: API key 'hsi_prod_key_abc123' expires in 7 days
```

8.6.2 Weekly Reports

```
# Generate weekly operations report
python scripts/weekly_report.py --email ops@hashinsight.net

# Report includes:
# - System uptime and availability
# - Performance metrics (latency, error rate)
# - Resource usage trends
# - Security events
# - Backup status
# - Action items
```

Chapter 9: Performance Tuning Guide

9.1 Request Coalescing Optimization

HashInsight implements Request Coalescing pattern to achieve **9.8x performance improvement**:

How It Works

```
# Multiple concurrent identical requests merged into single request
# Example: 10 simultaneous "get BTC price" requests → 1 API call

from performance.request_coalescer import RequestCoalescer

coalescer = RequestCoalescer()

@coalescer.coalesce(key='btc_price')
def get_btc_price():
    # Expensive API call
    response = requests.get('https://api.coingecko.com/...')
    return response.json()

# 10 concurrent calls to get_btc_price() → Only 1 API call executed
# 9 requests wait for the first one to complete and share result
```

Performance Metrics

```
# View coalescing statistics
curl http://localhost:5000/api/performance/coalescing-stats

# Response
{
  "enabled": true,
  "total_requests": 98234,
  "deduplicated_requests": 85432,
  "api_calls_saved": 85432,
  "performance_improvement": "9.8x",
  "average_wait_time_ms": 45
}
```

9.2 Database Optimization

9.2.1 Index Strategy

```
-- High-frequency query field indexes
CREATE INDEX idx_miners_user_id ON miners(user_id);
CREATE INDEX idx_market_data_timestamp ON market_analytics(created_at DESC);
CREATE INDEX idx_calculations_user_created ON calculations(user_id, created_at);
```

```
-- Composite indexes (commonly queried together)
CREATE INDEX idx_miners_user_model ON miners(user_id, model);

-- Partial indexes (index only active data)
CREATE INDEX idx_active_miners ON miners(user_id) WHERE status = 'active';
```

9.2.2 Query Optimization

Use EXPLAIN to Analyze Queries

```
EXPLAIN ANALYZE
SELECT m.*, u.username
FROM miners m
JOIN users u ON m.user_id = u.id
WHERE m.status = 'active'
ORDER BY m.created_at DESC
LIMIT 100;

-- Review execution plan, optimize slow queries
```

Avoid N+1 Queries

```
#   Bad - N+1 queries
miners = Miner.query.filter_by(user_id=user_id).all()
for miner in miners:
    print(miner.user.username)  # One query per loop iteration

#   Good - Use JOIN
miners = Miner.query.join(User).filter(Miner.user_id == user_id).all()
for miner in miners:
    print(miner.user.username)  # Single query
```

9.2.3 Connection Pool Tuning

```
# config.py
SQLALCHEMY_ENGINE_OPTIONS = {
    'pool_size': 10,           # Adjust based on concurrency (workers × 2)
    'pool_recycle': 300,       # Recycle connections after 5 minutes
    'pool_pre_ping': True,     # Test connection before use (Neon required)
    'pool_timeout': 30,        # 30 second timeout
    'max_overflow': 20,        # Max 20 overflow connections
    'connect_args': {
        'connect_timeout': 15,         # Connection timeout
```

```
        'application_name': 'hashinsight',  # For monitoring
        'options': '-c statement_timeout=30000'  # Query timeout 30 seconds
    }
}
```

9.3 Caching Strategy

9.3.1 Multi-tier Caching

```
# Three-tier cache architecture
L1: Memory cache (fastest, smallest capacity)
    ↓ miss
L2: Redis cache (fast, medium capacity)
    ↓ miss
L3: Database (slowest, largest capacity)

# Implementation
@cache_manager.cached(ttl=300, level='L1')  # 5 minutes
def get_user_profile(user_id):
    return db.session.query(User).get(user_id)

@cache_manager.cached(ttl=3600, level='L2')  # 1 hour
def get_market_data():
    return fetch_from_api()
```

9.3.2 TTL Configuration

Data Type	TTL	Reason
User info	5 minutes	May change
Market data	5 minutes	Real-time requirement
Miner list	1 hour	Low change frequency
Statistics	10 minutes	Compute-intensive
Static content	24 hours	Rarely changes

9.3.3 Cache Warming

```
# Warm critical caches at startup
python scripts/warmup_cache.py

# Script contents:
# 1. Load popular user data
# 2. Load current market data
# 3. Pre-calculate statistics
# 4. Load configuration data

# Output:
#   Warmed user cache: 1,234 entries
#   Warmed market data: Current price + hashrate
#   Warmed statistics: 10 dashboards
#    Duration: 12.3s
```

9.4 Batch Processing Optimization

9.4.1 Vectorized Computation

```
#   Bad - Individual calculation
for miner in miners:
    daily_revenue = calculate_revenue(
        miner.hashrate,
        btc_price,
        network_difficulty
    )
# Duration: 5000 miners × 10ms = 50 seconds

#   Good - Vectorized calculation
import numpy as np

hashrates = np.array([m.hashrate for m in miners])
revenues = calculate_revenue_vectorized(
    hashrates,
    btc_price,
    network_difficulty
)
# Duration: ~500ms (100x improvement)
```

9.4.2 Concurrent Processing

```
from concurrent.futures import ThreadPoolExecutor

# Concurrent batch processing
def process_miner_batch(miners, batch_size=100):
    with ThreadPoolExecutor(max_workers=8) as executor:
        futures = []
        for i in range(0, len(miners), batch_size):
            batch = miners[i:i+batch_size]
            future = executor.submit(process_batch, batch)
            futures.append(future)

        results = [f.result() for f in futures]
    return results
```

9.4.3 Memory Optimization

```
# Batch insert (avoid ORM overhead)
from sqlalchemy import insert

#   Bad - Individual inserts
for data in large_dataset:
    db.session.add(Model(**data))
    db.session.commit()  # Commit each time

#   Good - Batch insert
db.session.bulk_insert_mappings(Model, large_dataset)
db.session.commit()  # Single commit

#   Better - Use bulk insert
stmt = insert(Model).values(large_dataset)
db.session.execute(stmt)
db.session.commit()
```

9.5 Performance Monitoring

```
# Regularly run performance benchmarks
python scripts/performance_benchmark.py

# Output:
#   API response times:
#   - /api/miners: p50=45ms, p95=120ms, p99=250ms
```

```
#   - /api/dashboard: p50=80ms, p95=200ms, p99=400ms
#   Database queries:
#   - Average query time: 15ms
#   - Slow queries (>1s): 0
#   Cache performance:
#   - Hit rate: 78.5%
#   - Request Coalescing: 9.8x improvement
```

Chapter 10: Emergency Response

10.1 On-Call Rotation

Rotation Schedule

Time Period	Primary On-Call	Backup On-Call	Escalation Contact
Weekdays 09:00-18:00	DevOps Engineer	Backend Engineer	Technical Director
Weekdays 18:00-09:00	Rotation Engineer	Backup Engineer	On-Call Manager
Weekends/Holidays	Rotation Engineer	Backup Engineer	On-Call Manager

On-Call Tools

- **PagerDuty**: Alert notifications
- **Slack**: #incident-response channel
- **Grafana**: Real-time monitoring
- **Incident.io**: Incident management

10.2 Incident Severity Levels

Level	Impact	Response Time	Escalation Time	Examples
P0	Complete service outage	15 minutes	30 minutes	Database down, application inaccessible
P1	Core functionality degraded	30 minutes	1 hour	API latency >5s, error rate >5%
P2	Partial functionality failure	2 hours	4 hours	Single feature unavailable
P3	Performance degradation	4 hours	8 hours	Slower response but functional
P4	Non-urgent issues	1 business day	2 business days	UI display issues, minor bugs

10.3 Response Process

P0/P1 Critical Incident Process

```
1. [0-5 minutes] Alert triggered
   ├→ PagerDuty notifies primary on-call
   ├→ Automatically create Slack incident channel
   └→ Automatically notify backup on-call

2. [5-15 minutes] Initial response
   ├→ Acknowledge incident (ACK alert)
   ├→ Post initial status update
   ├→ Begin troubleshooting
   └→ Record timeline

3. [15-30 minutes] Emergency mitigation
   ├→ Implement temporary mitigation
   ├→ Assess impact scope
   ├→ Decide whether to escalate
   └→ Update incident status

4. [30 minutes+] Full resolution
   ├→ Implement permanent fix
```

```
   ├→ Verify service recovery
   ├→ Post recovery announcement
   └→ Begin post-incident review
```

10.4 Common Emergency Scenarios

Scenario 1: Database Connection Pool Exhausted

Symptoms

```
sqlalchemy.exc.TimeoutError: QueuePool limit of size 10 overflow 20 reached
```

Emergency Response

```
# 1. Immediately kill long idle connections
psql $DATABASE_URL -c "
SELECT pg_terminate_backend(pid)
FROM pg_stat_activity
WHERE state = 'idle'
  AND state_change < NOW() - INTERVAL '10 minutes';
"

# 2. Increase connection pool limits (temporary)
export SQLALCHEMY_POOL_SIZE=20
export SQLALCHEMY_MAX_OVERFLOW=40

# 3. Restart application
systemctl restart hashinsight

# 4. Monitor connection count
watch -n 5 "psql $DATABASE_URL -c 'SELECT count(*) FROM pg_stat_activity;'"
```

Scenario 2: Memory Leak Causing OOM

Symptoms

```
MemoryError: Unable to allocate array
Killed (OOM)
```

Emergency Response

```
# 1. Immediately restart affected worker
kill -HUP $(cat /var/run/gunicorn.pid)

# 2. Clear cache to free memory
redis-cli FLUSHDB

# 3. Limit worker count (temporary)
gunicorn --workers 2 --max-requests 500 main:app

# 4. Monitor memory usage
watch -n 5 'free -h'
```

Scenario 3: Malicious Traffic Attack

Symptoms - Abnormally high request volume - Many 401/403 errors - Suspicious requests from specific IP ranges

Emergency Response

```
# 1. Enable rate limiting
export RATE_LIMIT_ENABLED=true
export RATE_LIMIT_REQUESTS_PER_MINUTE=10

# 2. Block malicious IPs (via firewall)
ufw deny from 192.168.1.0/24

# 3. Enable Cloudflare protection (if using)
# Visit Cloudflare Dashboard -> Security -> DDoS

# 4. Analyze attack patterns
tail -1000 /var/log/gunicorn/access.log | \
  awk '{print $1}' | sort | uniq -c | sort -rn | head -20
```

10.5 Rollback Procedures

Code Rollback

```
# 1. Determine rollback version
git log --oneline -10

# 2. Rollback to last stable version
git revert HEAD --no-edit
# Or
```

```
git checkout v1.9.5

# 3. Redeploy
git push origin main

# 4. Verify rollback success
curl http://localhost:5000/health | jq '.version'

# 5. Notify team
# Slack: "Rolled back to v1.9.5, service restored"
```

Database Rollback

```
# ⚠ Warning: Database rollback is high-risk, must backup first!

# 1. Create current state backup
python backup/backup_manager.py --type emergency

# 2. Restore to rollback point
psql $DATABASE_URL < /tmp/backups/hashinsight_backup_20251003_020000.sql

# 3. Verify data integrity
python scripts/verify_database_integrity.py

# 4. Restart application
systemctl restart hashinsight
```

10.6 Communication Templates

Initial Incident Announcement

```
  [P0 Incident] HashInsight Service Outage

Time: 2025-10-03 14:23 UTC
Impact: All users unable to access main application
Status: Under investigation

We have confirmed a service outage, team is urgently investigating.
Will provide update within 30 minutes.

Incident channel: #incident-2025-10-03-db-outage
On-call engineer: @john.doe
```

Progress Update

```
  [Update] HashInsight Incident Progress

Time: 2025-10-03 14:45 UTC
Root cause: Database connection pool exhausted
Mitigation: Restarted database connection pool
Current status: Service partially restored, monitoring

Next update: 15:00 UTC or when major progress made
```

Recovery Announcement

```
  [Resolved] HashInsight Service Restored

Time: 2025-10-03 15:12 UTC
Duration: 49 minutes
Root cause: Improper PostgreSQL connection pool configuration
Solution: Increased connection pool limits and optimized slow queries

All services now fully restored.
Post-incident review will be published within 24 hours.

Thank you for your patience.
```

10.7 Post-Incident Review (Postmortem)

Template

```
# HashInsight Incident Report - 2025-10-03 Database Outage

## Summary
- **Incident ID**: INC-2025-1003-001
- **Severity**: P0
- **Start Time**: 2025-10-03 14:23 UTC
- **Recovery Time**: 2025-10-03 15:12 UTC
- **Duration**: 49 minutes
- **Affected Users**: 100% (all users)

## Timeline
- 14:23 - Prometheus alert: Database connection failure
- 14:25 - On-call engineer confirms incident
```

```
- 14:30 - Identified connection pool exhaustion
- 14:35 - Implemented temporary mitigation (killed idle connections)
- 14:45 - Service partially restored
- 15:00 - Implemented permanent fix (increased connection pool)
- 15:12 - Service fully restored

## Root Cause
PostgreSQL connection pool size configured at 10, but actual concurrent
demand reached 30+, causing connection wait timeouts.

## Resolution
1. Temporary: Killed idle connections, freed connection pool
2. Permanent: Increased connection pool size to 20, overflow to 40
3. Optimization: Identified and optimized 3 slow queries

## Prevention Measures
- [ ] Set connection pool alerts (usage >80%)
- [ ] Regular slow query review
- [ ] Add capacity planning process
- [ ] Add connection pool monitoring dashboard

## Lessons Learned
  Quick response and good communication
  Lacked connection pool monitoring
  Insufficient capacity planning
```

Appendices

Appendix A: Quick Command Reference

Application Management

```
# Start application
gunicorn --bind 0.0.0.0:5000 --workers 4 main:app

# Graceful restart
kill -HUP $(cat /var/run/gunicorn.pid)

# Stop application
kill -TERM $(cat /var/run/gunicorn.pid)

# View processes
```

```
ps aux | grep gunicorn

# View logs
tail -f /var/log/hashinsight/app.log
```

Database

```
# Connect to database
psql $DATABASE_URL

# View tables
\dt

# View connection count
SELECT count(*) FROM pg_stat_activity;

# Backup database
pg_dump $DATABASE_URL > backup.sql

# Restore database
psql $DATABASE_URL < backup.sql
```

Cache

```
# Redis connection
redis-cli

# View all keys
redis-cli KEYS 'hashinsight:*'

# Clear cache
redis-cli FLUSHDB

# View memory usage
redis-cli INFO memory
```

Monitoring

```
# View health status
curl http://localhost:5000/health | jq

# View Prometheus metrics
```

```
curl http://localhost:9090/metrics

# View SLO status
curl http://localhost:5000/api/slo/status | jq
```

Appendix B: Configuration File Examples

.env Example

```
# Required configuration
DATABASE_URL=postgresql://user:pass@host:5432/hashinsight_db
SESSION_SECRET=your-secret-key-min-32-chars
ENCRYPTION_PASSWORD=encryption-key-min-32-chars

# Blockchain configuration
BLOCKCHAIN_ENABLED=true
BLOCKCHAIN_PRIVATE_KEY=0x1234567890abcdef...
BLOCKCHAIN_NETWORK=base-sepolia
BASE_RPC_URL=https://sepolia.base.org

# Backup configuration
BACKUP_DIR=/var/backups/hashinsight
BACKUP_ENCRYPTION_KEY=backup-encryption-key
BACKUP_RETENTION_DAYS=30
BACKUP_STORAGE_TYPE=s3
BACKUP_STORAGE_BUCKET=hashinsight-backups

# KMS configuration (AWS)
AWS_KMS_KEY_ID=arn:aws:kms:us-east-1:123456789:key/xxxxx
AWS_KMS_REGION=us-east-1
AWS_ACCESS_KEY_ID=AKIAXXXXXXXX
AWS_SECRET_ACCESS_KEY=xxxxxxxxxx

# Monitoring configuration
ENABLE_BACKGROUND_SERVICES=0  # Default: disabled; set to 1 to enable background data collection
PROMETHEUS_PORT=9090
SLO_MEASUREMENT_WINDOW=30

# API configuration
COINWARZ_API_KEY=your-coinwarz-api-key
COINGECKO_API_KEY=your-coingecko-api-key
```

systemd Service File Example

```
# /etc/systemd/system/hashinsight.service

[Unit]
Description=HashInsight Platform
After=network.target postgresql.service

[Service]
Type=notify
User=hashinsight
Group=hashinsight
WorkingDirectory=/opt/hashinsight
EnvironmentFile=/opt/hashinsight/.env

ExecStart=/opt/hashinsight/venv/bin/gunicorn \
  --bind 0.0.0.0:5000 \
  --workers 4 \
  --threads 2 \
  --worker-class gthread \
  --timeout 120 \
  --max-requests 1000 \
  --access-logfile /var/log/hashinsight/access.log \
  --error-logfile /var/log/hashinsight/error.log \
  --pid /var/run/gunicorn.pid \
  main:app

ExecReload=/bin/kill -s HUP $MAINPID
KillMode=mixed
KillSignal=SIGTERM
TimeoutStopSec=30

Restart=on-failure
RestartSec=10

[Install]
WantedBy=multi-user.target
```

Appendix C: Monitoring Metrics Reference

Prometheus Metrics List

Metric Name	Type	Description
`hashinsight_requests_total`	Counter	Total request count
`hashinsight_request_latency_seconds`	Histogram	Request latency
`hashinsight_error_rate`	Gauge	Error rate
`hashinsight_cache_hit_rate`	Gauge	Cache hit rate
`hashinsight_db_query_duration_seconds`	Histogram	Database query duration
`hashinsight_db_connection_pool_size`	Gauge	Database connection pool size
`hashinsight_db_connection_pool_active`	Gauge	Active connection count
`hashinsight_slo_compliance`	Gauge	SLO compliance
`hashinsight_slo_error_budget_remaining`	Gauge	Error budget remaining
`hashinsight_circuit_breaker_state`	Gauge	Circuit breaker state

SLO Thresholds

SLO Type	Target	Warning Threshold	Critical Threshold
Availability	99.95%	<99.9%	<99.5%
P95 latency	≤250ms	>200ms	>300ms
Error rate	≤0.1%	>0.5%	>1%
Error budget	21.6min/month	<20%	<10%

Appendix D: Error Code Reference

Error Code	HTTP Status	Description	Solution
`AUTH_001`	401	Not logged in	Re-login
`AUTH_002`	403	Insufficient permissions	Contact administrator
`AUTH_003`	401	Session expired	Re-login
`DB_001`	500	Database connection failed	Check DATABASE_URL
`DB_002`	500	Query timeout	Optimize query or increase timeout
`CACHE_001`	503	Redis unavailable	Check Redis service
`API_001`	429	Too many requests	Reduce request frequency
`API_002`	500	External API failed	Retry later
`BLOCKCHAIN_001`	500	Private key not configured	Set BLOCKCHAIN_PRIVATE_KEY
`ENCRYPTION_001`	500	Encryption key not configured	Set ENCRYPTION_PASSWORD

Appendix E: Version History

Version	Date	Author	Changes
v2.0	2025-10-03	HashInsight Ops Team	Complete operations manual first release
v1.9	2025-09-15	DevOps Team	Added Request Coalescing optimization
v1.8	2025-08-20	Security Team	Enhanced KMS and mTLS documentation
v1.7	2025-07-10	Platform Team	Added SLO monitoring chapter

Document Maintenance

Maintenance Responsibility: HashInsight Platform Operations Team
Review Cycle: Quarterly
Feedback Channel: ops@hashinsight.net
Documentation Repository: https://github.com/hashinsight/operations-manual

Last Updated: October 3, 2025
Next Review: January 3, 2026