OP TRADING PLATFORM - TECHNICAL CONCEPTS EXPLAINED IN LAYMAN'S TERMS

TABLE OF CONTENTS

CONFIGURATION CONCEPTS

Memory Mapping

What it is: Memory mapping lets your computer treat files like they're part of your computer's RAM memory.

Simple Analogy: Think of it like having a book (file) that magically appears on your desk (RAM) whenever you need it, instead of having to walk to the library (hard disk) every time you want to read it.

When to use:

- ✅ **Enable (true)**: When you have large data files (>10MB) and plenty of RAM (8GB+)
- ❌ **Disable (false)**: When you have small files or limited RAM (<4GB)

Performance Impact:

- **Enabled**: 2-5x faster file access, uses more RAM
- **Disabled**: Slower file access, uses less RAM

Configuration:

```
# For systems with 8GB+ RAM and large data files
USE_MEMORY_MAPPING=true
```

```
# For systems with <4GB RAM or small files
USE_MEMORY_MAPPING=false
```

Compression

What it is: Compression reduces file sizes by removing redundant information, like ZIP files.

Simple Analogy: Like vacuum-sealing clothes in your suitcase - they take up less space but you need to unpack them before using.

Trade-offs:

- **Enabled**: Smaller files, less disk space needed, more CPU usage
- **Disabled**: Larger files, more disk space needed, less CPU usage

Compression Levels (1-9):

- **Level 1**: Fast compression, bigger files (use for real-time data)
- **Level 6**: Balanced (recommended for most cases)
- **Level 9**: Maximum compression, slower (use for archival)

When to use:

```
# Development: Disable for faster processing
COMPRESSION_ENABLED=false

# Production with limited disk: Enable with balanced level
COMPRESSION_ENABLED=true
COMPRESSION_LEVEL=6

# Archival storage: Maximum compression
COMPRESSION_ENABLED=true
COMPRESSION_LEVEL=9
```

Buffer Sizes

What it is: Buffers are temporary storage areas that batch data before writing to files.

Simple Analogy: Like using a basket to collect apples instead of making a trip to the storage room for each apple.

CSV Buffer Size:

- **Small (1024)**: More frequent writes, less memory usage
- **Medium (8192)**: Balanced performance (recommended)
- **Large (32768)**: Fewer writes, more memory usage

JSON Buffer Size:

- JSON files are typically larger, so use 2x the CSV buffer size

Optimal Settings:

```
# For systems with limited RAM (<4GB)
CSV_BUFFER_SIZE=4096
JSON_BUFFER_SIZE=8192

# For systems with adequate RAM (8GB+)
CSV_BUFFER_SIZE=8192
JSON_BUFFER_SIZE=16384

# For high-performance systems (16GB+ RAM)
CSV_BUFFER_SIZE=16384
JSON_BUFFER_SIZE=32768
```

Maximum Memory Usage

What it is: A soft limit on how much RAM the application should try to use.

Important Note: This is NOT a hard limit - your system won't crash if exceeded. It's a guideline that helps the application manage memory efficiently.

How it works:

- Application monitors its memory usage
- When approaching the limit, it triggers cleanup operations
- Helps prevent system slowdowns from excessive memory usage

Setting Guidelines:

```
# For 4GB system: Use 25-30% of total RAM
MAX_MEMORY_USAGE_MB=1024

# For 8GB system: Use 30-40% of total RAM
MAX_MEMORY_USAGE_MB=2048

# For 16GB system: Use 40-50% of total RAM
MAX_MEMORY_USAGE_MB=4096

# For 32GB+ system: Use 50%+ of total RAM
MAX_MEMORY_USAGE_MB=8192
```

Why this won't disrupt your system:

- Modern operating systems have virtual memory management
- Other applications can still use remaining RAM
- OS will manage memory swapping automatically
- Application will adapt its behavior as it approaches the limit

Strike Offsets Configuration

What it is: Strike offsets determine which option contracts to collect relative to the current market price (ATM - At The Money).

Simple Explanation:

- **Offset 0**: The strike price closest to current market price
- **Offset +1**: One strike price above market price
- **Offset -1**: One strike price below market price

Visual Example:
If NIFTY is trading at 19,750:

- Offset -2: 19,650 strike
- Offset -1: 19,700 strike
- Offset 0: 19,750 strike (ATM)
- Offset +1: 19,800 strike
- Offset +2: 19,850 strike

How to Switch Between Default and Extended:

Method 1: Environment Variable

```
# Use default range (5 strikes)
ACTIVE_STRIKE_OFFSETS=-2,-1,0,1,2

# Use extended range (11 strikes)
ACTIVE_STRIKE_OFFSETS=-5,-4,-3,-2,-1,0,1,2,3,4,5

# Use custom range
ACTIVE_STRIKE_OFFSETS=-1,0,1
```

Method 2: Application Setting

```
# In your application code
from shared.config.settings import get_settings

settings = get_settings()
if settings.feature_extended_analysis:
    offsets = settings.extended_strike_offsets
else:
    offsets = settings.default_strike_offsets
```

Data Security Configuration

Data Retention:

- **Infinite retention** means data is never automatically deleted
- Suitable for trading data which has long-term analytical value
- Monitor disk space growth and plan storage accordingly

Archival vs Retention:

- **Archival**: Moving old data to compressed, slower storage
- **Retention**: How long data is kept before deletion
- You can archive old data while keeping infinite retention

Security Settings:

```
# Data never expires (recommended for trading data)
DATA_RETENTION_PERIOD=infinite

# Archive old data to save space (optional)
ENABLE_ARCHIVAL=true
ARCHIVAL_AFTER_DAYS=30

# Compress archived data heavily
ARCHIVE_COMPRESSION_ENABLED=true
ARCHIVE_COMPRESSION_LEVEL=9

# Encrypt sensitive data (optional, adds ~5% CPU overhead)
ENABLE_DATA_ENCRYPTION_AT_REST=false
```

Structured Logging

What it is: Instead of plain text log messages, structured logging creates machine-readable logs with consistent fields.

Traditional Logging:

```
2025-08-25 10:30:15 - Processing option chain for NIFTY completed in 1.2 seconds
```

Structured Logging:

```
{
  "timestamp": "2025-08-25T10:30:15.123Z",
  "level": "INFO",
  "message": "Processing completed",
  "service": "analytics",
  "index": "NIFTY",
  "duration_ms": 1200,
  "trace_id": "abc123xyz789",
```

```
  "request_id": "req_456def"
}
```

Benefits:

- Easy to search and filter logs
- Can correlate logs across different services
- Machine-readable for automated analysis

Key Fields:

- **Trace ID**: Follows a request through multiple services
- **Request ID**: Unique identifier for each API request
- **Service**: Which service generated the log
- **Duration**: How long operations took

Configuration:

```
# Enable structured JSON logging
ENABLE_STRUCTURED_LOGGING=true
LOG_FORMAT=json

# Include correlation IDs for tracing
INCLUDE_TRACE_ID=true
INCLUDE_REQUEST_ID=true
INCLUDE_USER_ID=true

# Additional context
LOG_INCLUDE_HOSTNAME=true
LOG_INCLUDE_PROCESS_ID=true
```

□ CONTAINERIZATION & ORCHESTRATION

Production-Ready Containers

What Containers Are: Think of containers like shipping containers - they package your application with everything it needs to run, ensuring it works the same everywhere.

Our Multi-Stage Builds:

```
# Stage 1: Base environment with Python
FROM python:3.11-slim as base
# Install system dependencies

# Stage 2: Development with debugging tools
FROM base as development
# Add development tools, hot reload

# Stage 3: Production optimized
```

```
FROM base as production
# Minimal footprint, security hardened
```

Benefits:

- **Consistency**: Same environment everywhere (dev, test, prod)
- **Isolation**: Applications don't interfere with each other
- **Scalability**: Easy to run multiple copies
- **Security**: Isolated from host system

Development Containers with Hot Reload

What Hot Reload Is: Your application automatically restarts when you change code, so you see changes immediately.

How it Works:

- Container watches your code files for changes
- When you save a file, container automatically restarts the application
- No need to manually stop/start during development

Configuration:

```
# Development docker-compose
version: '3.8'
services:
  api:
    build:
      target: development  # Use development stage
    volumes:
      - .:/app  # Mount code for hot reload
    environment:
      - API_RELOAD=true
```

Health Checks Built Into Containers

What Health Checks Are: Automatic tests that verify your application is working properly.

How They Work:

```
HEALTHCHECK --interval=30s --timeout=10s --start-period=40s \
  CMD curl -f http://localhost:8000/health || exit 1
```

This means:

- Every 30 seconds, check if the application responds
- Wait up to 10 seconds for a response

- Give the application 40 seconds to start up initially
- If the health check fails, container is marked as unhealthy

Resource Limits and Optimization

Why Resource Limits: Prevent any single container from using all system resources.

Setting Limits:

```
services:
  api:
    deploy:
      resources:
        limits:
          cpus: '0.5'      # Use max 50% of one CPU core
          memory: 512M     # Use max 512MB of RAM
        reservations:
          cpus: '0.25'     # Reserve 25% of one CPU core
          memory: 256M     # Reserve 256MB of RAM
```

Optimization Techniques:

- **Multi-stage builds**: Smaller final images
- **Layer caching**: Faster builds by reusing unchanged layers
- **Alpine Linux**: Smaller, more secure base images
- **Non-root users**: Enhanced security

Security Hardening with Non-Root Users

Why Non-Root: Running as root inside containers is a security risk.

Our Implementation:

```
# Create non-root user
RUN addgroup --system --gid 1001 optrading
RUN adduser --system --uid 1001 --ingroup optrading optrading

# Switch to non-root user
USER optrading

# Application runs as optrading user, not root
```

Security Benefits:

- Reduced attack surface
- Limits damage if container is compromised
- Follows security best practices
- Required by many enterprise environments

MESSAGE QUEUE & DECOUPLED DATA FLOW

Event-Driven Architecture with Redis Pub/Sub

What Event-Driven Architecture Is: Services communicate by sending messages about events, rather than directly calling each other.

Traditional Approach (Tightly Coupled):

```
Collection Service → Directly calls → Processing Service
                  → Directly calls → Analytics Service
```

Event-Driven Approach (Loosely Coupled):

```
Collection Service → Publishes "data_collected" event → Redis
                                                          ↓
Processing Service ← Subscribes to events ←──────────────┘
                   ↓
                   Publishes "data_processed" event → Redis
                                                          ↓
Analytics Service ← Subscribes to events ←────────────────┘
```

Benefits:

- Services can be developed independently
- Easy to add new services without changing existing ones
- Better fault tolerance - if one service fails, others continue
- Natural load balancing

How Our Implementation Works:

```
# Publishing an event
await redis_coord.publish_message("data_events", {
    "event_type": "legs_collected",
    "index": "NIFTY",
    "count": 150,
    "timestamp": now(),
    "bucket": "this_week"
})

# Subscribing to events
async def handle_data_event(message):
    if message["event_type"] == "legs_collected":
        # Process the new data
        await process_option_legs(message["index"])
```

Distributed Coordination with Locks and Cursors

Why Coordination is Needed: When multiple services work with the same data, they need to coordinate to avoid conflicts.

Distributed Locks: Ensure only one service processes data at a time.

```
# Only one service can process NIFTY data at a time
with redis_coord.distributed_lock("processing_nifty", timeout=300):
    # Safely process NIFTY data
    process_nifty_options()
    # Lock automatically released
```

File Cursors: Track progress through large files.

```
# Remember where we left off in a large CSV file
cursor = redis_coord.get_file_cursor("nifty_data.csv")
if cursor:
    start_from_line = cursor.position
else:
    start_from_line = 0

# Process file from last position
process_csv_file("nifty_data.csv", start_from_line)

# Update cursor with new position
redis_coord.set_file_cursor("nifty_data.csv", new_position, checksum)
```

Asynchronous Processing Between Services

Synchronous vs Asynchronous:

Synchronous (Blocking):

```
# Service waits for response before continuing
result = process_data()  # Waits here
send_to_analytics(result)  # Then does this
```

Asynchronous (Non-blocking):

```
# Service starts processing and continues immediately
asyncio.create_task(process_data())  # Starts this
await send_to_analytics()  # And does this in parallel
```

Benefits of Async:

- Better performance and responsiveness
- Can handle multiple requests simultaneously
- More efficient use of system resources

- Better user experience

Message Persistence and Reliability

What Message Persistence Means: Messages are saved to disk so they survive system restarts.

How Redis Ensures Reliability:

```
# Redis persistence settings
REDIS_SAVE_ENABLED=true          # Enable saving to disk
REDIS_SAVE_INTERVAL=900          # Save every 15 minutes
REDIS_AOF_ENABLED=true           # Append-only file for durability
```

Message Delivery Guarantees:

- **At-least-once**: Message is delivered but might be delivered multiple times
- **Exactly-once**: Message is delivered exactly once (harder to implement)
- Our system uses at-least-once with idempotent processing

Service Discovery Through Redis

What Service Discovery Is: Services automatically find and connect to each other.

How It Works:

```
# Service registers itself
await redis_coord.register_service("analytics", {
    "host": "localhost",
    "port": 8001,
    "health_endpoint": "/health",
    "status": "healthy"
})

# Other services discover it
analytics_service = await redis_coord.discover_service("analytics")
# Returns: {"host": "localhost", "port": 8001, ...}
```

Benefits:

- Services don't need hardcoded addresses
- Easy to scale services up/down
- Automatic failover to healthy instances
- Better fault tolerance

⚓ KUBERNETES DEPLOYMENT

Auto-Scaling API Service (2-10 Replicas)

What Auto-Scaling Is: Kubernetes automatically creates more copies of your service when it gets busy, and removes them when traffic is low.

How It Works:

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: api-service-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: api-service
  minReplicas: 2      # Always keep at least 2 running
  maxReplicas: 10     # Never exceed 10 copies
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        averageUtilization: 70  # Scale up when CPU > 70%
  - type: Resource
    resource:
      name: memory
      target:
        averageUtilization: 80  # Scale up when memory > 80%
```

What This Means:

- During normal trading hours: 2-3 API services running
- During high volume periods: Scales up to 8-10 services
- During off-hours: Scales back down to 2 services
- Saves money by using only what you need

Load Balancing with Ingress Controller

What Load Balancing Is: Distributes incoming requests evenly across multiple service copies.

Simple Analogy: Like having multiple cashiers at a store - customers are directed to the cashier with the shortest line.

Ingress Controller Setup:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
```

```
metadata:
  name: op-trading-ingress
  annotations:
    kubernetes.io/ingress.class: nginx
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - host: api.your-domain.com
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: api-service
            port:
              number: 8000
```

Benefits:

- Even distribution of load across services
- Automatic failover if one service fails
- SSL termination at the load balancer
- Single entry point for all requests

Service Mesh Ready Architecture

What Service Mesh Is: An infrastructure layer that handles service-to-service communication, security, and monitoring.

Think of it like: A smart postal system that automatically handles mail delivery between buildings, provides tracking, ensures security, and gives you statistics.

Popular Service Mesh Options:

- **Istio**: Full-featured but complex
- **Linkerd**: Lightweight and easy to use
- **Consul Connect**: HashiCorp's solution

What Service Mesh Provides:

- Automatic mutual TLS between services
- Traffic routing and splitting for canary deployments
- Detailed metrics and tracing
- Circuit breakers and retries
- Policy enforcement

Multi-Zone Deployment Capability

What Multi-Zone Deployment Is: Your application runs across multiple data centers or availability zones.

Benefits:

- High availability - if one zone fails, others continue
- Better performance - users connect to nearest zone
- Disaster recovery built-in

Configuration Example:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: api-service
spec:
  replicas: 6
  template:
    spec:
      affinity:
        podAntiAffinity:
          preferredDuringSchedulingIgnoredDuringExecution:
          - weight: 100
            podAffinityTerm:
              labelSelector:
                matchExpressions:
                - key: app
                  operator: In
                  values: ["api-service"]
              topologyKey: topology.kubernetes.io/zone
```

This ensures pods are distributed across different zones.

Rolling Updates with Zero Downtime

What Rolling Updates Are: Updating your application without any service interruption.

How It Works:

1. Start new version of service alongside old version
2. Gradually shift traffic from old to new version
3. Once all traffic is on new version, remove old version
4. If problems occur, quickly rollback to old version

Configuration:

```
apiVersion: apps/v1
kind: Deployment
metadata:
```

```
  name: api-service
spec:
  replicas: 4
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 1    # Only take down 1 instance at a time
      maxSurge: 1          # Can temporarily have 1 extra instance
```

Zero Downtime Process:

1. Current: 4 instances of version 1.0 running
2. Start: 1 instance of version 1.1, now have 5 total
3. Test: Verify version 1.1 is healthy
4. Shift: Stop 1 instance of version 1.0, now have 4 total (3 v1.0, 1 v1.1)
5. Repeat: Continue until all instances are version 1.1

AUTHENTICATION & SECURITY

JWT Tokens - Detailed Setup

What JWT Is: JSON Web Tokens are a secure way to transmit information between parties. Think of them like tamper-proof ID cards.

JWT Structure:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwi

Header.Payload.Signature
```

Step-by-Step JWT Setup:

1. Generate Secret Key:

```
# Generate a secure random key
$key = [System.Web.Security.Membership]::GeneratePassword(64, 10)
Write-Output $key
```

2. Configure Environment:

```
JWT_SECRET_KEY=your_generated_secret_key_here
JWT_ALGORITHM=HS256
JWT_EXPIRATION_HOURS=24
JWT_REFRESH_TOKEN_EXPIRE_DAYS=7
```

3. Implementation Example:

```
import jwt
from datetime import datetime, timedelta
from fastapi import HTTPException, Depends
from fastapi.security import HTTPBearer

security = HTTPBearer()

def create_jwt_token(user_data: dict) -> str:
    """Create JWT token for user"""
    payload = {
        "user_id": user_data["id"],
        "username": user_data["username"],
        "exp": datetime.utcnow() + timedelta(hours=24),
        "iat": datetime.utcnow(),
        "iss": "op-trading-platform"
    }

    token = jwt.encode(
        payload,
        settings.jwt_secret_key,
        algorithm="HS256"
    )
    return token

def verify_jwt_token(token: str) -> dict:
    """Verify and decode JWT token"""
    try:
        payload = jwt.decode(
            token,
            settings.jwt_secret_key,
            algorithms=["HS256"]
        )
        return payload
    except jwt.ExpiredSignatureError:
        raise HTTPException(401, "Token has expired")
    except jwt.InvalidTokenError:
        raise HTTPException(401, "Invalid token")
```

4. Protecting API Endpoints:

```
from fastapi import Depends

async def get_current_user(token: str = Depends(security)):
    """Get current authenticated user"""
    credentials_exception = HTTPException(
        status_code=401,
        detail="Could not validate credentials",
        headers={"WWW-Authenticate": "Bearer"},
    )

    try:
        payload = verify_jwt_token(token.credentials)
        username = payload.get("username")
        if username is None:
```

```
            raise credentials_exception
        return {"username": username, "user_id": payload.get("user_id")}
    except Exception:
        raise credentials_exception

# Protected endpoint
@app.get("/protected")
async def protected_endpoint(current_user: dict = Depends(get_current_user)):
    return {"message": f"Hello {current_user['username']}"}
```

API Keys - Implementation

What API Keys Are: Long-lived tokens for programmatic access, like permanent passwords for applications.

API Key Implementation:

```
import secrets
import hashlib
from datetime import datetime

def generate_api_key() -> tuple[str, str]:
    """Generate API key and hash"""
    # Generate random key
    api_key = f"op_{secrets.token_urlsafe(32)}"

    # Create hash for storage (never store raw keys)
    key_hash = hashlib.sha256(api_key.encode()).hexdigest()

    return api_key, key_hash

def verify_api_key(provided_key: str, stored_hash: str) -> bool:
    """Verify API key against stored hash"""
    provided_hash = hashlib.sha256(provided_key.encode()).hexdigest()
    return secrets.compare_digest(provided_hash, stored_hash)

# API Key middleware
from fastapi import Request, HTTPException

async def api_key_middleware(request: Request, call_next):
    """Middleware to check API keys"""
    # Skip auth for public endpoints
    if request.url.path in ["/health", "/docs", "/openapi.json"]:
        return await call_next(request)

    # Check for API key in header
    api_key = request.headers.get("X-API-Key")
    if not api_key:
        raise HTTPException(401, "API key required")

    # Verify API key (check against database)
    if not verify_api_key_in_db(api_key):
        raise HTTPException(401, "Invalid API key")
```

```
    return await call_next(request)
```

API Key Management:

```
class APIKeyManager:
    def __init__(self, redis_client):
        self.redis = redis_client

    async def create_api_key(self, user_id: str, name: str) -> str:
        """Create new API key for user"""
        api_key, key_hash = generate_api_key()

        # Store in Redis with metadata
        key_data = {
            "user_id": user_id,
            "name": name,
            "created_at": datetime.utcnow().isoformat(),
            "last_used": None,
            "usage_count": 0,
            "is_active": True
        }

        await self.redis.hset(f"api_key:{key_hash}", mapping=key_data)

        return api_key  # Return raw key only once

    async def revoke_api_key(self, api_key: str):
        """Revoke an API key"""
        key_hash = hashlib.sha256(api_key.encode()).hexdigest()
        await self.redis.hset(f"api_key:{key_hash}", "is_active", "false")

    async def list_api_keys(self, user_id: str) -> list:
        """List user's API keys (without showing actual keys)"""
        keys = []
        async for key in self.redis.scan_iter(match="api_key:*"):
            key_data = await self.redis.hgetall(key)
            if key_data.get("user_id") == user_id:
                keys.append({
                    "name": key_data.get("name"),
                    "created_at": key_data.get("created_at"),
                    "last_used": key_data.get("last_used"),
                    "usage_count": key_data.get("usage_count"),
                    "is_active": key_data.get("is_active") == "true"
                })
        return keys
```

Authentication Flow - Complete Process

1. User Registration/Login:

```
@app.post("/auth/login")
async def login(credentials: UserCredentials):
```

```
    # Verify username/password
    user = await verify_user_credentials(credentials)
    if not user:
        raise HTTPException(401, "Invalid credentials")

    # Generate JWT token
    access_token = create_jwt_token(user)
    refresh_token = create_refresh_token(user)

    return {
        "access_token": access_token,
        "refresh_token": refresh_token,
        "token_type": "bearer",
        "expires_in": 3600
    }
```

2. Token Refresh:

```
@app.post("/auth/refresh")
async def refresh_token(refresh_token: str):
    try:
        payload = verify_refresh_token(refresh_token)
        user = await get_user_by_id(payload["user_id"])

        # Generate new access token
        new_access_token = create_jwt_token(user)

        return {
            "access_token": new_access_token,
            "token_type": "bearer",
            "expires_in": 3600
        }
    except Exception:
        raise HTTPException(401, "Invalid refresh token")
```

3. Authorization Levels:

```
from enum import Enum

class Permission(Enum):
    READ_DATA = "read_data"
    WRITE_DATA = "write_data"
    ADMIN_ACCESS = "admin_access"

def require_permission(permission: Permission):
    def decorator(func):
        async def wrapper(*args, **kwargs):
            # Get current user from JWT
            user = kwargs.get("current_user")
            if not user:
                raise HTTPException(401, "Authentication required")

            # Check if user has required permission
            user_permissions = await get_user_permissions(user["user_id"])
```

```
            if permission.value not in user_permissions:
                raise HTTPException(403, "Insufficient permissions")

            return await func(*args, **kwargs)
        return wrapper
    return decorator

# Usage
@app.get("/admin/users")
@require_permission(Permission.ADMIN_ACCESS)
async def list_users(current_user: dict = Depends(get_current_user)):
    return await get_all_users()
```

MONITORING & TESTING

Prometheus - Detailed Setup Instructions

What Prometheus Is: A monitoring system that collects metrics from your applications and stores them in a time-series database.

Step-by-Step Setup:

1. Install Prometheus (Windows):

```
# Download Prometheus
Invoke-WebRequest -Uri "https://github.com/prometheus/prometheus/releases/download/v2.45.

# Extract
Expand-Archive -Path "prometheus.zip" -DestinationPath "C:\prometheus"

# Add to PATH
$env:PATH += ";C:\prometheus\prometheus-2.45.0.windows-amd64"
```

2. Create Prometheus Configuration:

```
# prometheus.yml
global:
  scrape_interval: 15s
  evaluation_interval: 15s

rule_files:
  - "alert_rules.yml"

scrape_configs:
  # OP Trading Platform API Service
  - job_name: 'op-api'
    static_configs:
      - targets: ['localhost:8080']  # Metrics endpoint
    metrics_path: '/metrics'
    scrape_interval: 30s
```

```
  # OP Trading Platform Analytics Service
  - job_name: 'op-analytics'
    static_configs:
      - targets: ['localhost:8081']

  # System metrics (optional)
  - job_name: 'node-exporter'
    static_configs:
      - targets: ['localhost:9100']

alerting:
  alertmanagers:
    - static_configs:
        - targets:
          - alertmanager:9093
```

3. Create Alert Rules:

```
# alert_rules.yml
groups:
  - name: op-trading-alerts
    rules:
      - alert: HighAPILatency
        expr: api_request_duration_seconds{quantile="0.95"} > 2
        for: 2m
        labels:
          severity: warning
        annotations:
          summary: "High API latency detected"
          description: "95th percentile latency is {{ $value }} seconds"

      - alert: DataCollectionStopped
        expr: increase(options_legs_collected_total[5m]) == 0
        for: 2m
        labels:
          severity: critical
        annotations:
          summary: "Data collection has stopped"
          description: "No option legs collected in the last 5 minutes"

      - alert: HighErrorRate
        expr: rate(api_requests_total{status="error"}[5m]) > 0.1
        for: 1m
        labels:
          severity: critical
        annotations:
          summary: "High error rate detected"
          description: "Error rate is {{ $value }} requests/second"
```

4. Implement Metrics in Your Application:

```
# metrics.py
from prometheus_client import Counter, Histogram, Gauge, start_http_server
import time
```

```
from functools import wraps

# Define metrics
api_requests_total = Counter(
    'api_requests_total',
    'Total API requests',
    ['method', 'endpoint', 'status']
)

api_request_duration_seconds = Histogram(
    'api_request_duration_seconds',
    'API request duration',
    ['method', 'endpoint']
)

options_legs_collected_total = Counter(
    'options_legs_collected_total',
    'Total option legs collected',
    ['index', 'bucket']
)

system_memory_usage_bytes = Gauge(
    'system_memory_usage_bytes',
    'Current memory usage in bytes'
)

# Decorator for automatic metrics collection
def track_metrics(endpoint: str):
    def decorator(func):
        @wraps(func)
        async def wrapper(*args, **kwargs):
            start_time = time.time()
            status = "success"

            try:
                result = await func(*args, **kwargs)
                return result
            except Exception as e:
                status = "error"
                raise
            finally:
                duration = time.time() - start_time
                api_request_duration_seconds.labels(
                    method="GET",
                    endpoint=endpoint
                ).observe(duration)
                api_requests_total.labels(
                    method="GET",
                    endpoint=endpoint,
                    status=status
                ).inc()

        return wrapper
    return decorator

# Usage in FastAPI
```

```
@app.get("/option-chain/{index}")
@track_metrics("option-chain")
async def get_option_chain(index: str):
    # Your API logic here
    pass

# Start metrics server
def start_metrics_server():
    start_http_server(8080)
    print("Metrics server started on port 8080")
```

5. Start Prometheus:

```
# Navigate to Prometheus directory
cd C:\prometheus\prometheus-2.45.0.windows-amd64

# Start Prometheus
.\prometheus.exe --config.file=prometheus.yml --storage.tsdb.path=data --web.console.libr
```

6. Access Prometheus:

- Web UI: http://localhost:9090
- Metrics: http://localhost:8080/metrics (your app)
- Query examples:
  - `api_requests_total` - Total API requests
  - `rate(api_requests_total[5m])` - Request rate over 5 minutes
  - `api_request_duration_seconds{quantile="0.95"}` - 95th percentile latency

Grafana API Key vs Datasource UID

They Are Different Things:

Grafana API Key:

- Used for programmatic access to Grafana's REST API
- Allows you to create/modify dashboards, datasources, etc. via code
- Format: `eyJrIjoiVjF...` (long random string)

Datasource UID:

- Unique identifier for a specific data source within Grafana
- Used internally by Grafana to reference data sources in dashboards
- Format: `influxdb-uid` or `prometheus-uid` (short descriptive string)

How to Get Grafana API Key:

1. Via Grafana Web UI:

```
1. Login to Grafana (http://localhost:3000)
2. Go to Configuration → API Keys
3. Click "New API Key"
4. Name: "OP-Trading-Automation"
5. Role: "Editor" (or "Admin" if needed)
6. Time to live: 1y (or never expire)
7. Click "Add"
8. Copy the generated key immediately (you won't see it again)
```

2. Via REST API:

```
# Create API key via curl
curl -X POST http://admin:admin@localhost:3000/api/auth/keys \
  -H "Content-Type: application/json" \
  -d '{
    "name": "op-trading-automation",
    "role": "Editor",
    "secondsToLive": 31536000
  }'
```

How to Get Datasource UID:

1. Via Grafana Web UI:

```
1. Go to Configuration → Data Sources
2. Click on your data source (e.g., InfluxDB)
3. Look at the URL: /datasources/edit/UID_HERE
4. Or check the "Basic settings" section for UID field
```

2. Via API:

```
# List all datasources and their UIDs
curl -H "Authorization: Bearer YOUR_API_KEY" \
  http://localhost:3000/api/datasources
```

Usage Example:

```
import requests

# Grafana configuration
GRAFANA_URL = "http://localhost:3000"
GRAFANA_API_KEY = "eyJrIjoiVjF..."  # Your API key
DATASOURCE_UID = "influxdb-main"    # Your datasource UID

headers = {
    "Authorization": f"Bearer {GRAFANA_API_KEY}",
    "Content-Type": "application/json"
}

# Create a dashboard that uses the datasource
```

```
dashboard_config = {
    "dashboard": {
        "title": "OP Trading Dashboard",
        "panels": [
            {
                "title": "API Requests",
                "type": "graph",
                "datasource": {
                    "uid": DATASOURCE_UID  # Reference to datasource
                },
                "targets": [
                    {
                        "query": "SELECT * FROM api_requests"
                    }
                ]
            }
        ]
    }
}

response = requests.post(
    f"{GRAFANA_URL}/api/dashboards/db",
    headers=headers,
    json=dashboard_config
)
```

Comprehensive Testing Framework

Testing Strategy Overview:

1. Unit Tests:

```
# Test individual components in isolation
import pytest
from unittest.mock import Mock, patch

class TestOptionsAnalytics:
    def test_black_scholes_calculation(self):
        # Test Black-Scholes pricing
        price = BlackScholesModel.call_price(
            S=100, K=105, T=0.25, r=0.05, sigma=0.2
        )
        assert 2.0 < price < 4.0  # Reasonable range

    @patch('redis.Redis')
    def test_analytics_service_initialization(self, mock_redis):
        # Test service starts properly
        service = OptionsAnalyticsService()
        assert service.is_running == False
        assert service.redis_client is not None
```

2. Integration Tests:

```
# Test service interactions
import pytest
import asyncio

class TestServiceIntegration:
    @pytest.mark.asyncio
    async def test_collection_to_processing_flow(self):
        # Start collection service
        collector = ATMOptionCollector()
        await collector.initialize()

        # Start processing service
        processor = ConsolidatedCSVWriter()

        # Simulate data collection
        legs = await collector.collect_option_legs("NIFTY")

        # Process collected data
        result = await processor.process_and_write(legs)

        assert result["success"] == True
        assert result["legs_written"] > 0
```

3. Chaos Testing:

```
# Test system resilience
import random
import time

class TestChaosEngineering:
    def test_random_service_failures(self):
        """Randomly kill services and verify recovery"""
        services = ["api", "collection", "analytics"]

        for _ in range(10):  # 10 chaos rounds
            # Randomly kill a service
            service = random.choice(services)
            self.kill_service(service)

            # Wait for auto-recovery
            time.sleep(30)

            # Verify service is back up
            assert self.check_service_health(service) == True

    def test_network_partition(self):
        """Simulate network failures between services"""
        # Block Redis connection
        with self.block_redis_connection():
            # Services should gracefully degrade
            response = self.make_api_request("/health")
            assert response.status_code == 503  # Service Unavailable
```

```
            # Should include degraded status
            assert "redis_unavailable" in response.json()
```

4. Performance Tests:

```
import pytest
import asyncio
import time
from concurrent.futures import ThreadPoolExecutor

class TestPerformance:
    @pytest.mark.performance
    async def test_api_throughput(self):
        """Test API can handle expected load"""

        async def make_request():
            async with httpx.AsyncClient() as client:
                response = await client.get("http://localhost:8000/option-chain/NIFTY")
                return response.status_code == 200

        # Simulate 100 concurrent requests
        start_time = time.time()
        tasks = [make_request() for _ in range(100)]
        results = await asyncio.gather(*tasks)
        end_time = time.time()

        # Verify performance metrics
        success_rate = sum(results) / len(results)
        total_time = end_time - start_time
        requests_per_second = len(results) / total_time

        assert success_rate > 0.95  # 95% success rate
        assert requests_per_second > 10  # At least 10 RPS
        assert total_time < 30  # Complete within 30 seconds

    def test_memory_usage_under_load(self):
        """Verify memory doesn't grow indefinitely under load"""
        import psutil
        import os

        process = psutil.Process(os.getpid())
        initial_memory = process.memory_info().rss / 1024 / 1024  # MB

        # Generate sustained load
        for i in range(1000):
            # Simulate processing option legs
            self.process_large_dataset()

            if i % 100 == 0:  # Check every 100 iterations
                current_memory = process.memory_info().rss / 1024 / 1024
                memory_growth = current_memory - initial_memory

                # Memory growth should be reasonable
                assert memory_growth < 500  # Less than 500MB growth
```

5. Test Automation:

```
# github-actions.yml (CI/CD pipeline)
name: OP Trading Platform Tests

on: [push, pull_request]

jobs:
  test:
    runs-on: ubuntu-latest
    services:
      redis:
        image: redis:7-alpine
        ports:
          - 6379:6379
      influxdb:
        image: influxdb:2.7-alpine
        ports:
          - 8086:8086

    steps:
    - uses: actions/checkout@v3

    - name: Set up Python
      uses: actions/setup-python@v3
      with:
        python-version: '3.11'

    - name: Install dependencies
      run: |
        python -m pip install --upgrade pip
        pip install -r requirements.txt

    - name: Run unit tests
      run: pytest tests/unit/ -v

    - name: Run integration tests
      run: pytest tests/integration/ -v

    - name: Run performance tests
      run: pytest tests/performance/ -v -m performance

    - name: Generate coverage report
      run: pytest --cov=shared --cov=services --cov-report=xml

    - name: Upload coverage to Codecov
      uses: codecov/codecov-action@v3
```

DETAILED SETUP INSTRUCTIONS

Complete Kubernetes Setup

Prerequisites:

- Docker Desktop with Kubernetes enabled
- kubectl command line tool
- Helm package manager (optional but recommended)

Step 1: Enable Kubernetes in Docker Desktop

```
1. Open Docker Desktop
2. Go to Settings → Kubernetes
3. Check "Enable Kubernetes"
4. Click "Apply & Restart"
5. Wait for Kubernetes to start (green indicator)
```

Step 2: Verify Kubernetes Installation

```
kubectl version --client
kubectl cluster-info
kubectl get nodes
```

Step 3: Create Namespace

```
kubectl create namespace op-trading
kubectl config set-context --current --namespace=op-trading
```

Step 4: Create Secrets

```
# Create secret for sensitive data
kubectl create secret generic op-secrets \
  --from-literal=kite-api-key="your_api_key" \
  --from-literal=kite-api-secret="your_api_secret" \
  --from-literal=influxdb-token="your_token" \
  --from-literal=jwt-secret="your_jwt_secret"
```

Step 5: Create ConfigMap

```
# Create ConfigMap from your .env file
kubectl create configmap op-config --from-env-file=.env
```

Step 6: Deploy Application

```
# Apply Kubernetes manifests
kubectl apply -f infrastructure/kubernetes/
```

```
# Check deployment status
kubectl get deployments
kubectl get pods
kubectl get services
```

Step 7: Set up Ingress (for external access)

```
# Install NGINX Ingress Controller
kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/controller-v1

# Wait for ingress controller to be ready
kubectl wait --namespace ingress-nginx \
  --for=condition=ready pod \
  --selector=app.kubernetes.io/component=controller \
  --timeout=300s
```

Step 8: Configure Horizontal Pod Autoscaler

```
# Enable metrics server (if not already enabled)
kubectl apply -f https://github.com/kubernetes-sigs/metrics-server/releases/latest/downlo

# Verify HPA is working
kubectl get hpa
kubectl describe hpa api-service-hpa
```

Step 9: Set up Persistent Volumes

```
# persistent-volumes.yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: op-data-pv
spec:
  capacity:
    storage: 50Gi
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Retain
  storageClassName: local-storage
  hostPath:
    path: /data/op-trading
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: op-data-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
```

```
      storage: 50Gi
  storageClassName: local-storage
```

Step 10: Monitor and Troubleshoot

```
# Check pod logs
kubectl logs -f deployment/api-service

# Check events
kubectl get events --sort-by=.metadata.creationTimestamp

# Shell into a pod for debugging
kubectl exec -it deployment/api-service -- /bin/bash

# Port forward for local testing
kubectl port-forward service/api-service 8000:8000
```

Production Deployment Checklist

Pre-Production Checklist:

1. Environment Configuration ✓

```
# Verify all required environment variables
python -c "
from shared.config.settings import get_settings
settings = get_settings()
print('Environment:', settings.environment)
print('Debug mode:', settings.debug)
print('Database configured:', bool(settings.influxdb.url))
print('Redis configured:', bool(settings.redis.host))
"
```

2. Kite Connect API Credentials ✓

```
# Test API credentials
python services/collection/kite_auth_manager.py --status

# Verify authentication
python -c "
from services.collection.kite_auth_manager import get_kite_client
client = get_kite_client(interactive=False)
if client:
    profile = client.get_profile()
    print(f'Authenticated as: {profile.get(\"user_name\", \"Unknown\")}')
else:
    print('Authentication failed')
"
```

3. Redis Cluster Configuration ✓

```
# Test Redis connectivity
redis-cli -h localhost -p 6379 ping

# Test Redis cluster (if using cluster)
redis-cli -c -h localhost -p 6379 cluster info
```

4. InfluxDB Authentication Setup ✓

```
# Test InfluxDB connection
curl -H "Authorization: Token YOUR_TOKEN" \
  "http://localhost:8086/api/v2/buckets"

# Verify bucket exists
python -c "
from influxdb_client import InfluxDBClient
client = InfluxDBClient(
    url='http://localhost:8086',
    token='YOUR_TOKEN',
    org='your-org'
)
buckets = client.buckets_api().find_buckets()
print([bucket.name for bucket in buckets])
"
```

5. SSL Certificates Installation ✓

```
# Generate self-signed certificate for testing
openssl req -x509 -newkey rsa:4096 -keyout key.pem -out cert.pem -days 365 -nodes

# Or install Let's Encrypt certificate
sudo certbot certonly --standalone -d your-domain.com
```

6. Firewall Rules Configuration ✓

```
# Windows Firewall
netsh advfirewall firewall add rule name="OP Trading API" dir=in action=allow protocol=TC
netsh advfirewall firewall add rule name="OP Trading Redis" dir=in action=allow protocol=
netsh advfirewall firewall add rule name="OP Trading InfluxDB" dir=in action=allow protoc

# Check current firewall rules
netsh advfirewall firewall show rule name=all | findstr "OP Trading"
```

7. Backup Strategy Implementation ✓

```
# Test backup creation
python -c "
import subprocess
import datetime
import os
```

```
# Create backup
backup_name = f'backup_{datetime.datetime.now().strftime(\"%Y%m%d_%H%M%S\")}.tar.gz'
subprocess.run(['tar', '-czf', backup_name, 'data/'])

# Verify backup
if os.path.exists(backup_name):
    size = os.path.getsize(backup_name) / 1024 / 1024  # MB
    print(f'Backup created: {backup_name} ({size:.2f} MB)')
else:
    print('Backup creation failed')
"
```

8. Monitoring Dashboards Configuration ✓

```
# Test Grafana API
curl -H "Authorization: Bearer YOUR_API_KEY" \
  http://localhost:3000/api/health

# Import dashboards
curl -X POST \
  -H "Authorization: Bearer YOUR_API_KEY" \
  -H "Content-Type: application/json" \
  -d @infrastructure/grafana/op-options-analytics-dashboard.json \
  http://localhost:3000/api/dashboards/db
```

9. Alert Channels Configuration ✓

```
# Test email alerts
python -c "
import smtplib
from email.mime.text import MIMEText

try:
    server = smtplib.SMTP('smtp.gmail.com', 587)
    server.starttls()
    server.login('your_email@gmail.com', 'your_app_password')

    msg = MIMEText('Test alert from OP Trading Platform')
    msg['Subject'] = 'Test Alert'
    msg['From'] = 'your_email@gmail.com'
    msg['To'] = 'admin@company.com'

    server.send_message(msg)
    server.quit()
    print('Email alert test successful')
except Exception as e:
    print(f'Email alert test failed: {e}')
"
```

10. Load Testing Completion ✓

```
# Install load testing tool
pip install locust

# Create load test file
cat > load_test.py << 'EOF'
from locust import HttpUser, task, between

class TradingPlatformUser(HttpUser):
    wait_time = between(1, 3)

    @task(3)
    def get_health(self):
        self.client.get("/health")

    @task(2)
    def get_option_chain(self):
        self.client.get("/option-chain/NIFTY?bucket=this_week")

    @task(1)
    def get_analytics(self):
        self.client.get("/analytics/NIFTY/greeks?bucket=this_week")

EOF

# Run load test
locust -f load_test.py --host=http://localhost:8000 --users=50 --spawn-rate=5 --run-time=
```

Go-Live Checklist:

1. All Services Deployed and Healthy ✓

```
# Check all services
kubectl get pods -o wide
kubectl get deployments
kubectl get services

# Verify health endpoints
for service in api collection analytics monitoring; do
  echo "Checking $service health..."
  curl -f http://localhost:8000/health || echo "$service health check failed"
done
```

2. Health Checks Passing ✓

```
# Comprehensive health check script
python -c "
import requests
import time

services = {
    'API': 'http://localhost:8000/health',
    'Grafana': 'http://localhost:3000/api/health',
    'Prometheus': 'http://localhost:9090/-/ready',
```

```
    'InfluxDB': 'http://localhost:8086/ping'
}

for service, url in services.items():
    try:
        response = requests.get(url, timeout=5)
        if response.status_code == 200:
            print(f'✅ {service}: Healthy')
        else:
            print(f'⚠️  {service}: Unhealthy (Status: {response.status_code})')
    except Exception as e:
        print(f'❌ {service}: Error - {e}')
"
```

3. Market Data Flowing Correctly ✓

```
# Verify data collection
python -c "
from services.collection.atm_option_collector import ATMOptionCollector
import asyncio

async def test_data_flow():
    collector = ATMOptionCollector()
    await collector.initialize()

    # Test data collection
    legs = await collector.collect_option_legs('NIFTY')

    if legs:
        print(f'✅ Data collection working: {len(legs)} legs collected')
        print(f'Sample leg: {legs[0].to_dict()}')
    else:
        print('❌ No data collected')

asyncio.run(test_data_flow())
"
```

4. Analytics Updating in Real-time ✓

```
# Check analytics computation
curl -s http://localhost:8000/analytics/NIFTY/greeks?bucket=this_week | python -m json.to

# Verify analytics timestamps are recent
python -c "
from datetime import datetime, timedelta
import requests

response = requests.get('http://localhost:8000/analytics/NIFTY/greeks?bucket=this_week')
if response.status_code == 200:
    data = response.json()
    timestamp = datetime.fromisoformat(data.get('timestamp', ''))
    age = datetime.now() - timestamp

    if age < timedelta(minutes=5):
```

```
        print(f'✅ Analytics are fresh (age: {age})')
    else:
        print(f'⚠️  Analytics are stale (age: {age})')
else:
    print('❌ Analytics endpoint not responding')
"
```

5. Dashboards Showing Live Data ✓

```
# Test Grafana dashboards
curl -H "Authorization: Bearer YOUR_API_KEY" \
  "http://localhost:3000/api/dashboards/db/op-options-analytics-dashboard" | \
  python -c "
import sys, json
data = json.load(sys.stdin)
if 'dashboard' in data:
    print('✅ Dashboard accessible')
    panels = len(data['dashboard']['panels'])
    print(f'Dashboard has {panels} panels')
else:
    print('❌ Dashboard not found')
"
```

6. Alerts Working Correctly ✓

```
# Test alert firing
python -c "
import requests
import json

# Trigger a test alert by simulating high error rate
for _ in range(10):
    try:
        requests.get('http://localhost:8000/nonexistent-endpoint')
    except:
        pass

print('Generated test errors - check if alerts fired')
"

# Check Prometheus alerts
curl -s http://localhost:9090/api/v1/alerts | python -c "
import sys, json
data = json.load(sys.stdin)
alerts = data['data']['alerts']
print(f'Active alerts: {len(alerts)}')
for alert in alerts:
    print(f'- {alert[\"labels\"][\"alertname\"]}: {alert[\"state\"]}')
"
```

7. Performance Metrics Within Acceptable Ranges ✓

```
# Check key performance metrics
curl -s http://localhost:8000/metrics | grep -E "(api_request_duration|memory_usage|cpu_u

# Verify response times
python -c "
import requests
import time

response_times = []
for _ in range(10):
    start = time.time()
    requests.get('http://localhost:8000/health')
    duration = time.time() - start
    response_times.append(duration * 1000)  # Convert to ms

avg_response = sum(response_times) / len(response_times)
max_response = max(response_times)

print(f'Average response time: {avg_response:.2f}ms')
print(f'Maximum response time: {max_response:.2f}ms')

if avg_response < 500 and max_response < 1000:
    print('✅ Performance within acceptable range')
else:
    print('⚠️  Performance may be degraded')
"
```

8. Backup Systems Verified ✓

```
# Test backup and restore process
python -c "
import subprocess
import os
import shutil
import tempfile

# Create test data
test_file = 'data/test_backup.csv'
os.makedirs('data', exist_ok=True)
with open(test_file, 'w') as f:
    f.write('timestamp,value\\n2025-08-25 10:00:00,100\\n')

# Create backup
backup_file = 'test_backup.tar.gz'
subprocess.run(['tar', '-czf', backup_file, 'data/'])

# Simulate data loss
os.remove(test_file)

# Restore from backup
with tempfile.TemporaryDirectory() as tmpdir:
    subprocess.run(['tar', '-xzf', backup_file, '-C', tmpdir])
    restored_file = os.path.join(tmpdir, test_file)
```

```
    if os.path.exists(restored_file):
        print('✅ Backup and restore process working')
        # Restore the file
        shutil.copy2(restored_file, test_file)
    else:
        print('❌ Backup and restore process failed')

# Cleanup
os.remove(backup_file)
"
```

9. Rollback Plan Prepared ✓

```
# Create rollback script
cat > rollback.sh << 'EOF'
#!/bin/bash
echo "Rolling back OP Trading Platform..."

# Stop current services
kubectl scale deployment --all --replicas=0

# Deploy previous version
kubectl set image deployment/api-service api-service=op-trading/api:previous
kubectl set image deployment/collection-service collection-service=op-trading/collection:
kubectl set image deployment/analytics-service analytics-service=op-trading/analytics:pre

# Scale back up
kubectl scale deployment --all --replicas=2

echo "Rollback completed. Verifying services..."
kubectl get pods
EOF

chmod +x rollback.sh
echo "✅ Rollback script created: rollback.sh"
```

🔄 DATA RECOVERY PROCEDURES

Understanding Recovery Settings

Environment Variables Related to Recovery:

```
# Point-in-time recovery capability
ENABLE_POINT_IN_TIME_RECOVERY=true

# How long to keep recovery logs
RECOVERY_LOG_RETENTION_DAYS=7

# Automatic recovery from corruption
AUTO_RECOVERY_ENABLED=false
```

```
# Verify recovered data integrity
RECOVERY_VERIFICATION_ENABLED=true

# Backup settings
ENABLE_AUTOMATED_BACKUP=true
BACKUP_INTERVAL_HOURS=24
BACKUP_RETENTION_DAYS=30
BACKUP_COMPRESSION=true
```

Recovery Scenarios and Solutions

Scenario 1: Database Corruption

```
# recovery_procedures.py
import asyncio
from influxdb_client import InfluxDBClient
from datetime import datetime, timedelta

async def recover_from_database_corruption():
    """Recover from InfluxDB corruption using backups"""

    # Step 1: Stop data ingestion
    await stop_data_ingestion_services()

    # Step 2: Assess corruption extent
    corruption_report = await assess_database_corruption()

    if corruption_report['severity'] == 'complete':
        # Complete restore from backup
        await restore_from_latest_backup()
    elif corruption_report['severity'] == 'partial':
        # Selective restore of affected data
        await selective_restore(corruption_report['affected_measurements'])

    # Step 3: Verify data integrity
    integrity_check = await verify_data_integrity()

    # Step 4: Resume services
    if integrity_check['passed']:
        await restart_data_ingestion_services()
        print("✅ Recovery completed successfully")
    else:
        print("❌ Recovery failed, manual intervention required")

async def restore_from_latest_backup():
    """Restore database from latest backup"""

    # Find latest backup
    latest_backup = find_latest_backup()
    print(f"Restoring from backup: {latest_backup}")

    # Stop InfluxDB
    subprocess.run(['docker', 'stop', 'op-influxdb'])

    # Restore data directory
```

```
    subprocess.run([
        'tar', '-xzf', latest_backup,
        '-C', '/var/lib/influxdb2'
    ])

    # Start InfluxDB
    subprocess.run(['docker', 'start', 'op-influxdb'])

    # Wait for InfluxDB to be ready
    await wait_for_influxdb_ready()
```

Scenario 2: File System Corruption

```
async def recover_from_filesystem_corruption():
    """Recover from CSV/JSON file corruption"""

    # Step 1: Identify corrupted files
    corrupted_files = await scan_for_corrupted_files()

    for file_path in corrupted_files:
        print(f"Recovering file: {file_path}")

        # Try to recover from InfluxDB
        if await recover_csv_from_influxdb(file_path):
            print(f"✅ Recovered {file_path} from InfluxDB")
            continue

        # Try to recover from backup
        if await recover_file_from_backup(file_path):
            print(f"✅ Recovered {file_path} from backup")
            continue

        # Try to reconstruct from related data
        if await reconstruct_file_from_related_data(file_path):
            print(f"✅ Reconstructed {file_path} from related data")
            continue

        print(f"❌ Could not recover {file_path}")

async def recover_csv_from_influxdb(csv_file_path):
    """Reconstruct CSV file from InfluxDB data"""

    # Parse file path to determine data type and time range
    index, bucket, date = parse_csv_filename(csv_file_path)

    # Query InfluxDB for the data
    client = InfluxDBClient(url=influxdb_url, token=influxdb_token)

    query = f'''
    from(bucket: "{influxdb_bucket}")
    |> range(start: {date}T00:00:00Z, stop: {date}T23:59:59Z)
    |> filter(fn: (r) => r._measurement == "option_legs")
    |> filter(fn: (r) => r.index == "{index}")
    |> filter(fn: (r) => r.bucket == "{bucket}")
    '''
```

```
    result = client.query_api().query_data_frame(query)

    if not result.empty:
        # Convert back to CSV format
        csv_data = convert_influxdb_to_csv_format(result)

        # Write to file
        with open(csv_file_path, 'w') as f:
            csv_data.to_csv(f, index=False)

        return True

    return False
```

Scenario 3: Service Configuration Loss

```
async def recover_service_configuration():
    """Recover service configuration from various sources"""

    recovery_sources = [
        recover_config_from_backup,
        recover_config_from_git,
        recover_config_from_kubernetes_configmap,
        recover_config_from_defaults
    ]

    for recovery_method in recovery_sources:
        try:
            config = await recovery_method()
            if validate_configuration(config):
                await apply_configuration(config)
                print(f"✅ Configuration recovered using {recovery_method.__name__}")
                return True
        except Exception as e:
            print(f"⚠️  {recovery_method.__name__} failed: {e}")

    print("❌ Could not recover configuration")
    return False

async def recover_config_from_kubernetes_configmap():
    """Recover configuration from Kubernetes ConfigMap"""

    # Get ConfigMap data
    result = subprocess.run([
        'kubectl', 'get', 'configmap', 'op-config',
        '-o', 'jsonpath={.data}'
    ], capture_output=True, text=True)

    if result.returncode == 0:
        config_data = json.loads(result.stdout)

        # Convert to .env format
        env_content = []
        for key, value in config_data.items():
```

```
            env_content.append(f"{key}={value}")

        # Write to .env file
        with open('.env', 'w') as f:
            f.write('\n'.join(env_content))

        return config_data

    return None
```

Scenario 4: Historical Data Recovery

```
async def recover_historical_data(start_date, end_date, indices):
    """Recover historical data for specific time period"""

    print(f"Recovering data from {start_date} to {end_date} for {indices}")

    recovery_stats = {
        'total_days': 0,
        'recovered_days': 0,
        'failed_days': [],
        'data_sources_used': []
    }

    current_date = start_date
    while current_date <= end_date:
        for index in indices:
            day_recovery_success = False

            # Try multiple recovery methods
            if await recover_day_from_backup(current_date, index):
                day_recovery_success = True
                recovery_stats['data_sources_used'].append('backup')

            elif await recover_day_from_influxdb(current_date, index):
                day_recovery_success = True
                recovery_stats['data_sources_used'].append('influxdb')

            elif await recover_day_from_external_api(current_date, index):
                day_recovery_success = True
                recovery_stats['data_sources_used'].append('external_api')

            if day_recovery_success:
                recovery_stats['recovered_days'] += 1
                print(f"✅ Recovered data for {index} on {current_date}")
            else:
                recovery_stats['failed_days'].append(f"{index}_{current_date}")
                print(f"❌ Failed to recover data for {index} on {current_date}")

        recovery_stats['total_days'] += 1
        current_date += timedelta(days=1)

    # Generate recovery report
    success_rate = recovery_stats['recovered_days'] / recovery_stats['total_days'] * 100
    print(f"\nRecovery Summary:")
```

```
    print(f"Success Rate: {success_rate:.1f}%")
    print(f"Data Sources Used: {set(recovery_stats['data_sources_used'])}")

    if recovery_stats['failed_days']:
        print(f"Failed Recoveries: {recovery_stats['failed_days']}")

    return recovery_stats
```

Automated Recovery Scripts

Daily Health Check and Auto-Recovery:

```
# scheduled_recovery_check.py
import schedule
import time
from datetime import datetime

def daily_health_check():
    """Comprehensive daily health check with auto-recovery"""

    print(f"Starting daily health check: {datetime.now()}")

    issues_found = []

    # Check database health
    db_health = check_database_health()
    if not db_health['healthy']:
        issues_found.append('database')
        if AUTO_RECOVERY_ENABLED:
            asyncio.run(auto_recover_database(db_health))

    # Check file system integrity
    fs_health = check_filesystem_integrity()
    if not fs_health['healthy']:
        issues_found.append('filesystem')
        if AUTO_RECOVERY_ENABLED:
            asyncio.run(auto_recover_filesystem(fs_health))

    # Check service configuration
    config_health = check_configuration_integrity()
    if not config_health['healthy']:
        issues_found.append('configuration')
        if AUTO_RECOVERY_ENABLED:
            asyncio.run(auto_recover_configuration(config_health))

    # Generate health report
    if issues_found:
        send_health_alert(f"Issues found: {', '.join(issues_found)}")
    else:
        print("✅ All health checks passed")

# Schedule daily health check
schedule.every().day.at("02:00").do(daily_health_check)  # 2 AM daily

# Keep the script running
```

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

Recovery Testing

Test Recovery Procedures:

```
async def test_recovery_procedures():
    """Test all recovery procedures in safe environment"""

    # Create test environment
    test_env = await create_isolated_test_environment()

    recovery_tests = [
        ('Database Corruption', test_database_recovery),
        ('File Corruption', test_file_recovery),
        ('Configuration Loss', test_config_recovery),
        ('Service Failure', test_service_recovery),
        ('Network Partition', test_network_partition_recovery)
    ]

    test_results = {}

    for test_name, test_function in recovery_tests:
        print(f"Testing: {test_name}")

        try:
            # Introduce controlled failure
            await introduce_controlled_failure(test_env, test_name)

            # Test recovery
            recovery_time = await time_recovery_process(test_function, test_env)

            # Verify recovery success
            recovery_success = await verify_recovery_success(test_env)

            test_results[test_name] = {
                'success': recovery_success,
                'recovery_time_seconds': recovery_time,
                'status': 'PASSED' if recovery_success else 'FAILED'
            }

        except Exception as e:
            test_results[test_name] = {
                'success': False,
                'error': str(e),
                'status': 'ERROR'
            }

        # Clean up for next test
        await reset_test_environment(test_env)

    # Generate test report
    print("\nRecovery Test Results:")
```

```
    for test_name, result in test_results.items():
        status_icon = "✅" if result['status'] == 'PASSED' else "❌"
        print(f"{status_icon} {test_name}: {result['status']}")

        if result.get('recovery_time_seconds'):
            print(f"   Recovery Time: {result['recovery_time_seconds']:.1f}s")

    # Clean up test environment
    await destroy_test_environment(test_env)

    return test_results
```

This comprehensive technical guide covers all the concepts and procedures needed to understand, deploy, and maintain your OP Trading Platform. Each section includes practical examples and step-by-step instructions that you can follow for your Windows environment.