ASG Trading System: Testing & CI/CD Suite

This repository organizes all testing, staging deployment, production CI/CD configuration, and fund allocation needed for immediate deployment of the ASG Trading System.

---

1. GitHub Actions CI Workflow (`.github/workflows/ci.yml`)

```
name: CI Pipeline

on:
  pull_request:
    branches: [ main ]
  push:
    branches: [ main ]

jobs:
  build-and-test:
    runs-on: ubuntu-latest
    services:
      postgres:
        image: postgres:13
        env:
          POSTGRES_USER: test
          POSTGRES_PASSWORD: test
          POSTGRES_DB: test_db
        ports:
          - 5432:5432
        options: >-
          --health-cmd "pg_isready -U test" \
          --health-interval 10s \
          --health-timeout 5s \
          --health-retries 5
    steps:
      - uses: actions/checkout@v3
      - name: Set up Python
        uses: actions/setup-python@v4
        with:
          python-version: '3.11'
      - name: Install dependencies
        run: |
          python -m pip install --upgrade pip
          pip install -r requirements.txt
      - name: Lint (flake8)
```

```
        run: flake8 src/ tests/
      - name: Security scan (safety)
        run: pip install safety && safety check
      - name: Run unit & integration tests
        env:
          DATABASE_URL: postgresql://test:test@localhost:5432/test_db
        run: pytest --cov=src --cov-report=xml
      - name: Publish coverage to Coveralls
        uses: coverallsapp/github-action@v2
        with:
          github-token: ${{ secrets.GITHUB_TOKEN }}
      - name: Build Docker image
        run: |
          docker build -t asg-trading:${{ github.sha }} .
      - name: Push to registry
        run: |
          echo ${{ secrets.REGISTRY_PASSWORD }} | docker login $
{{ secrets.REGISTRY_URL }} -u ${{ secrets.REGISTRY_USER }} --password-stdin
          docker tag asg-trading:${{ github.sha }} ${{ secrets.REGISTRY_URL }}/
asg-trading:${{ github.sha }}
          docker push ${{ secrets.REGISTRY_URL }}/asg-trading:${{ github.sha }}
```

2. Staging Deployment (`docker-compose.staging.yml`)

```yaml
version: '3.8'
services:
  app:
    image: ${{ secrets.REGISTRY_URL }}/asg-trading:${{ github.sha }}
    environment:
      - DATABASE_URL=${STAGING_DB_URL}
      - REDIS_URL=${STAGING_REDIS_URL}
    ports:
      - '8000:8000'
    depends_on:
      - db
      - redis
  db:
    image: postgres:13
    environment:
      POSTGRES_USER: asg
      POSTGRES_PASSWORD: securepass
      POSTGRES_DB: asg_staging
    volumes:
      - db_data:/var/lib/postgresql/data
  redis:
    image: redis:6-alpine
    volumes:
      - redis_data:/data
volumes:
  db_data:
  redis_data:
```

---

3. Test Suite Skeleton

```
asg_trading/
├── src/
│   └── ...
└── tests/
    ├── __init__.py
    ├── test_strategy.py      # Unit tests for entry/exit logic
    ├── test_api_endpoints.py # Integration tests for FastAPI endpoints
    └── test_utils.py         # Edge-case and helper function tests
```

Example Unit Test (`tests/test_strategy.py`)

```python
import pytest
from src.strategy import EntryExitStrategy

def test_entry_signal():
    strategy = EntryExitStrategy()
    data = [100, 102, 105, 103]
    assert strategy.compute_entry(data) == 'BUY'

def test_exit_signal():
    strategy = EntryExitStrategy()
    data = [105, 104, 100, 98]
    assert strategy.compute_exit(data) == 'SELL'
```

---

4. Configuration Files

`pytest.ini`

```ini
[pytest]
minversion = 6.0
```

```
addopts = --strict-markers --tb=short
testpaths = tests
```

`locustfile.py` (Performance Load Test)

```python
from locust import HttpUser, task, between

class TradingUser(HttpUser):
    wait_time = between(1, 2)

    @task
    def place_trade(self):
        self.client.post("/trade", json={
            "symbol": "BTCUSD",
            "side": "buy",
            "quantity": 0.001
        })
```

---

5. Chaos Testing

- **Chaos Monkey**: Integrate Gremlin or a custom script to randomly kill the `app` container in staging.
- **Network Latency**: Use `tc qdisc` on staging hosts to inject packet delay/loss for resilience validation.

---

6. Next Steps / Secrets Management

1. **GitHub Secrets**: Configure the following in repo settings:
2. `STAGING_DB_URL`
3. `STAGING_REDIS_URL`
4. `PRODUCTION_DB_URL`
5. `PRODUCTION_REDIS_URL`
6. `EXCHANGE_API_KEY`
7. `REGISTRY_URL`, `REGISTRY_USER`, `REGISTRY_PASSWORD`
8. **Canary Deployment**: Set up feature-flag based routing or Kubernetes rollout strategy for 5% traffic.
9. **Monitoring & Alerting**: Add Prometheus & Grafana configurations (see Section 8).

---

7. Production Deployment Workflow (`.github/workflows/deploy.yml`)

```
name: Prod Deploy

on:
  push:
    tags: [ 'v*.*.*' ]

jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Login to registry
        run: |
          echo ${{ secrets.REGISTRY_PASSWORD }} | docker login $
{{ secrets.REGISTRY_URL }} -u ${{ secrets.REGISTRY_USER }} --password-stdin
      - name: Pull image
        run: docker pull ${{ secrets.REGISTRY_URL }}/asg-trading:$
{{ github.ref_name }}
      - name: Deploy to Production
        run: |
          ssh -o StrictHostKeyChecking=no ${{ secrets.PROD_SSH_USER }}@$
{{ secrets.PROD_HOST }} 'docker pull ${{ secrets.REGISTRY_URL }}/asg-trading:$
{{ github.ref_name }} && docker-compose -f docker-compose.prod.yml up -d'
```

---

8. Monitoring & Alerting

`monitoring/prometheus.yml`

```
global:
  scrape_interval: 15s
scrape_configs:
  - job_name: 'asg-app'
    static_configs:
      - targets: ['app:8000']
    metrics_path: /metrics
```

Grafana Provisioning

- `monitoring/grafana/dashboards/asg-dashboard.json` – JSON dashboard definition with key panels:
- Trades/sec
- P&L vs. simulation
- Error rates and latency

Alertmanager Rules (`monitoring/alerts.yml`)

```
groups:
  - name: asg-alerts
    rules:
      - alert: HighErrorRate
        expr:
increase(request_errors_total[5m]) / increase(request_total[5m]) > 0.05
        for: 2m
        labels:
          severity: critical
        annotations:
          summary: ">5% error rate on ASG app"
      - alert: LatencySpike
        expr: histogram_quantile(0.95,
sum(rate(request_duration_seconds_bucket[5m])) by (le)) > 0.5
        for: 5m
        labels:
          severity: warning
        annotations:
          summary: "95th percentile latency >500ms"
```

9. Production Compose (`docker-compose.prod.yml`)

```
version: '3.8'
services:
  app:
    image: ${{ secrets.REGISTRY_URL }}/asg-trading:${VERSION}
    environment:
      - DATABASE_URL=${PRODUCTION_DB_URL}
      - REDIS_URL=${PRODUCTION_REDIS_URL}
      - EXCHANGE_API_KEY=${EXCHANGE_API_KEY}
    ports:
      - '8000:8000'
    depends_on:
      - db
      - redis
  db:
    image: postgres:13
    environment:
      POSTGRES_USER: asg
      POSTGRES_PASSWORD: secureprodpass
      POSTGRES_DB: asg_prod
    volumes:
      - db_data:/var/lib/postgresql/data
  redis:
    image: redis:6-alpine
    volumes:
      - redis_data:/data
volumes:
  db_data:
  redis_data:
```

10. Canary Strategy

- **Traffic Split**: Leverage a load balancer or feature-flag service (e.g., LaunchDarkly) to route 5% of requests to a canary instance. Use `docker-compose.canary.yml` mirroring `prod` but with `VERSION=canary` tag.
- **Health Checks**: Canary must pass smoke tests for 10 minutes before full rollout. Automate via GitHub Action or script.

11. Fund Allocation Strategy

After successful deployment and activation, transfer capital as follows:

- **Questrade Deposit (80%)**: Move 80% of available capital into the Questrade brokerage account via the Questrade API integration.
- **Reserve (20%)**: Retain 20% of capital in the system reserve wallet for contingencies.

Implementation Steps

1. **Retrieve Total Balance**: Query the system ledger for the current total available capital.
2. **Calculate Amounts**:
3. `TransferAmount = TotalBalance * 0.8`
4. `ReserveAmount = TotalBalance * 0.2`
5. **Execute Questrade Transfer**:
6. Use environment variables `QUESTRADE_API_KEY` and `QUESTRADE_ACCOUNT_ID` to authenticate.
7. Call the Questrade API endpoint to deposit `TransferAmount`.
8. Verify transaction success and log details.
9. **Update Ledger**:

10. Record `ReserveAmount` as retained in the reserve wallet.
11. Log the deposit transaction and balances for audit.

Configuration

- `QUESTRADE_API_KEY`: Stored in GitHub Secrets.
- `QUESTRADE_ACCOUNT_ID`: Stored in GitHub Secrets.
- API Timeouts and retry logic should be configured in `src/config.py`.

This suite now covers CI, staging, chaos, production, canary, monitoring, secrets, and capital allocation—fully ready for live deployment of the ASG Trading System.

---

12. Questrade Allocation Script (`scripts/allocate_to_questrade.py`)

```
import os
import logging
import requests
from decimal import Decimal

from src.ledger import get_total_balance, record_reserve_amount, record_deposit_transaction
from src.config import Config

# Configure logging
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

# Load config from environment
API_KEY = os.getenv('QUESTRADE_API_KEY')
ACCOUNT_ID = os.getenv('QUESTRADE_ACCOUNT_ID')
BASE_URL = 'https://api.questrade.com/v1'

if not API_KEY or not ACCOUNT_ID:
    logger.error('QUESTRADE_API_KEY and QUESTRADE_ACCOUNT_ID must be set in env')
    exit(1)

headers = {
    'Authorization': f'Bearer {API_KEY}',
    'Content-Type': 'application/json'
}
```

```
def allocate_funds():
    # 1. Retrieve total balance
    total_balance = Decimal(get_total_balance())
    logger.info(f'Total balance: {total_balance}')

    # 2. Calculate amounts
    transfer_amount = (total_balance * Decimal('0.8')).quantize(Decimal('0.01'))
    reserve_amount = (total_balance * Decimal('0.2')).quantize(Decimal('0.01'))
    logger.info(f'Allocating {transfer_amount} to Questrade; reserving {reserve_amount}')

    # 3. Execute Questrade transfer via deposit endpoint
    endpoint = f"{BASE_URL}/accounts/{ACCOUNT_ID}/funds"
    payload = {
        'amount': float(transfer_amount),
        'currency': 'CAD'
    }

    resp = requests.post(endpoint, headers=headers, json=payload, timeout=30)
    if resp.status_code != 200:
        logger.error(f'Questrade deposit failed: {resp.status_code} {resp.text}')
        exit(1)

    data = resp.json()
    transaction_id = data.get('id')
    logger.info(f'Deposit succeeded, transaction ID: {transaction_id}')

    # 4. Update ledger
    record_deposit_transaction(transaction_id, transfer_amount)
    record_reserve_amount(reserve_amount)
    logger.info('Ledger updated successfully')


if __name__ == '__main__':
    allocate_funds()
```

Make sure this script is executable and that your environment has the correct `QUESTRADE_API_KEY` and `QUESTRADE_ACCOUNT_ID` set. Then you can run:

```
python scripts/allocate_to_questrade.py
```

And that completes all steps for immediate deployment, including fund allocation to Questrade.