**Production Deployment Guide**

**🚀 From Prototype to Production**

This guide covers deploying the cognitive architecture in real-world applications.

**Table of Contents**

1. [Quick Deployment](#quick-deployment)
2. [Docker Containerization](#docker-containerization)
3. [Cloud Deployment](#cloud-deployment)
4. [Hardware Integration](#hardware-integration)
5. [Monitoring & Logging](#monitoring--logging)
6. [Security Considerations](#security-considerations)
7. [Performance Tuning](#performance-tuning)
8. [Troubleshooting](#troubleshooting)

**Quick Deployment**

**Minimal Production Setup**

# 1. Install in production mode

pip install -e .

# 2. Run as service

python -m philosophical\_agi.service --config production.yaml

**Configuration File (production.yaml)**

agent:

id: "production\_agent\_1"

workspace:

x\_min: 0.0

x\_max: 10.0

y\_min: 0.0

y\_max: 10.0

safety:

max\_violations: 5

verification\_timeout: 100ms

emergency\_stop\_enabled: true

monitoring:

log\_level: INFO

thought\_log\_enabled: true

metrics\_export: true

performance:

max\_beliefs: 1000

memory\_limit: 500MB

**Docker Containerization**

**Dockerfile**

FROM python:3.11-slim

WORKDIR /app

# Install system dependencies

RUN apt-get update && apt-get install -y \

build-essential \

&& rm -rf /var/lib/apt/lists/\*

# Copy requirements

COPY requirements.txt .

RUN pip install --no-cache-dir -r requirements.txt

# Copy application

COPY src/ ./src/

COPY config/ ./config/

# Create non-root user

RUN useradd -m -u 1000 cogagent && \

chown -R cogagent:cogagent /app

USER cogagent

# Health check

HEALTHCHECK --interval=30s --timeout=3s --start-period=5s --retries=3 \

CMD python -c "from philosophical\_agi.core.agent import CognitiveAgent; CognitiveAgent('health')" || exit 1

# Run application

CMD ["python", "-m", "philosophical\_agi.service"]

**Build and Run**

# Build image

docker build -t cognitive-arch:latest .

# Run container

docker run -d \

--name cognitive-agent \

--memory=1g \

--cpus=2 \

-p 8080:8080 \

-v $(pwd)/logs:/app/logs \

cognitive-arch:latest

# View logs

docker logs -f cognitive-agent

**Docker Compose**

version: '3.8'

services:

agent:

build: .

image: cognitive-arch:latest

container\_name: cognitive-agent

restart: unless-stopped

ports:

- "8080:8080"

volumes:

- ./logs:/app/logs

- ./config:/app/config:ro

environment:

- LOG\_LEVEL=INFO

- ENVIRONMENT=production

healthcheck:

test: ["CMD", "curl", "-f", "http://localhost:8080/health"]

interval: 30s

timeout: 3s

retries: 3

resources:

limits:

cpus: '2'

memory: 1G

reservations:

cpus: '1'

memory: 512M

monitoring:

image: prom/prometheus:latest

ports:

- "9090:9090"

volumes:

- ./prometheus.yml:/etc/prometheus/prometheus.yml

**Cloud Deployment**

**AWS Deployment**

**EC2 Instance**

# Launch EC2 instance (t3.medium or larger)

aws ec2 run-instances \

--image-id ami-xxxxx \

--instance-type t3.medium \

--key-name your-key \

--security-group-ids sg-xxxxx \

--user-data file://deploy-script.sh

# deploy-script.sh

#!/bin/bash

apt-get update

apt-get install -y python3.11 python3-pip git

git clone https://github.com/yourusername/philosophical-agi.git

cd philosophical-agi

pip3 install -r requirements.txt

python3 -m philosophical\_agi.service --config production.yaml

**ECS Deployment**

{

"family": "cognitive-agent",

"networkMode": "awsvpc",

"containerDefinitions": [

{

"name": "agent",

"image": "your-ecr-repo/cognitive-arch:latest",

"memory": 1024,

"cpu": 512,

"essential": true,

"portMappings": [

{

"containerPort": 8080,

"protocol": "tcp"

}

],

"environment": [

{

"name": "LOG\_LEVEL",

"value": "INFO"

}

],

"logConfiguration": {

"logDriver": "awslogs",

"options": {

"awslogs-group": "/ecs/cognitive-agent",

"awslogs-region": "us-east-1",

"awslogs-stream-prefix": "agent"

}

}

}

],

"requiresCompatibilities": ["FARGATE"],

"cpu": "512",

"memory": "1024"

}

**Google Cloud Platform**

**Cloud Run Deployment**

# Build and push to GCR

gcloud builds submit --tag gcr.io/PROJECT\_ID/cognitive-arch

# Deploy to Cloud Run

gcloud run deploy cognitive-agent \

--image gcr.io/PROJECT\_ID/cognitive-arch \

--platform managed \

--region us-central1 \

--memory 1Gi \

--cpu 2 \

--max-instances 10 \

--allow-unauthenticated

**Azure Deployment**

**Container Instances**

az container create \

--resource-group cognitive-rg \

--name cognitive-agent \

--image your-registry.azurecr.io/cognitive-arch:latest \

--cpu 2 \

--memory 1 \

--port 8080 \

--environment-variables LOG\_LEVEL=INFO

**Hardware Integration**

**ROS (Robot Operating System)**

#!/usr/bin/env python3

"""

ROS node wrapping the cognitive architecture

"""

import rospy

from geometry\_msgs.msg import Pose, Twist

from sensor\_msgs.msg import LaserScan

from philosophical\_agi.core.agent import CognitiveAgent

from philosophical\_agi.core.types import Action

class CognitiveROSNode:

def \_\_init\_\_(self):

rospy.init\_node('cognitive\_agent')

# Create cognitive agent

self.agent = CognitiveAgent("ros\_agent")

# Publishers

self.cmd\_pub = rospy.Publisher('/cmd\_vel', Twist, queue\_size=10)

# Subscribers

rospy.Subscriber('/scan', LaserScan, self.scan\_callback)

rospy.Subscriber('/pose', Pose, self.pose\_callback)

self.current\_observation = {}

def scan\_callback(self, msg):

# Convert laser scan to observation

min\_distance = min(msg.ranges)

self.current\_observation['threat\_level'] = 1.0 if min\_distance < 1.0 else 0.0

def pose\_callback(self, msg):

# Update position

self.current\_observation['agent\_position'] = (

msg.position.x,

msg.position.y,

msg.position.z

)

# Run cognitive cycle

action, info = self.agent.cognitive\_cycle(self.current\_observation)

# Convert to ROS command

cmd = Twist()

if action.velocity:

cmd.linear.x = action.velocity[0]

cmd.linear.y = action.velocity[1]

self.cmd\_pub.publish(cmd)

def run(self):

rospy.spin()

if \_\_name\_\_ == '\_\_main\_\_':

node = CognitiveROSNode()

node.run()

**Arduino/Microcontroller Integration**

"""

Serial communication with microcontroller

"""

import serial

import json

from philosophical\_agi.core.agent import CognitiveAgent

class MicrocontrollerInterface:

def \_\_init\_\_(self, port='/dev/ttyUSB0', baudrate=115200):

self.serial = serial.Serial(port, baudrate)

self.agent = CognitiveAgent("embedded\_agent")

def read\_sensors(self):

"""Read sensor data from microcontroller"""

line = self.serial.readline().decode('utf-8').strip()

return json.loads(line)

def send\_command(self, action):

"""Send action to microcontroller"""

command = {

'type': action.action\_type,

'target': action.target\_position

}

self.serial.write(json.dumps(command).encode('utf-8') + b'\n')

def run\_loop(self):

"""Main control loop"""

while True:

# Read sensors

observation = self.read\_sensors()

# Run cognitive cycle

action, info = self.agent.cognitive\_cycle(observation)

# Send command if safe

if not info['safety\_monitoring']['modified']:

self.send\_command(action)

**Monitoring & Logging**

**Structured Logging**

"""

Production logging configuration

"""

import logging

import json

from datetime import datetime

class StructuredLogger:

def \_\_init\_\_(self, name):

self.logger = logging.getLogger(name)

handler = logging.StreamHandler()

handler.setFormatter(JsonFormatter())

self.logger.addHandler(handler)

self.logger.setLevel(logging.INFO)

def log\_cognitive\_cycle(self, agent\_id, time, action, info):

self.logger.info('cognitive\_cycle', extra={

'agent\_id': agent\_id,

'time': time,

'action\_type': action.action\_type,

'safety\_verified': not info['safety\_monitoring']['modified'],

'beliefs\_count': info['beliefs\_updated']

})

def log\_safety\_violation(self, agent\_id, reason):

self.logger.warning('safety\_violation', extra={

'agent\_id': agent\_id,

'reason': reason,

'timestamp': datetime.utcnow().isoformat()

})

class JsonFormatter(logging.Formatter):

def format(self, record):

log\_data = {

'timestamp': datetime.utcnow().isoformat(),

'level': record.levelname,

'message': record.getMessage(),

'module': record.module

}

if hasattr(record, 'agent\_id'):

log\_data['agent\_id'] = record.agent\_id

return json.dumps(log\_data)

**Prometheus Metrics**

"""

Expose metrics for Prometheus

"""

from prometheus\_client import Counter, Histogram, Gauge, start\_http\_server

# Metrics

cognitive\_cycles = Counter('cognitive\_cycles\_total', 'Total cognitive cycles')

safety\_violations = Counter('safety\_violations\_total', 'Total safety violations')

cycle\_duration = Histogram('cycle\_duration\_seconds', 'Cognitive cycle duration')

active\_beliefs = Gauge('active\_beliefs', 'Number of active beliefs')

def instrumented\_cognitive\_cycle(agent, observation):

"""Cognitive cycle with metrics"""

cognitive\_cycles.inc()

with cycle\_duration.time():

action, info = agent.cognitive\_cycle(observation)

if info['safety\_monitoring']['modified']:

safety\_violations.inc()

active\_beliefs.set(info['beliefs\_updated'])

return action, info

# Start metrics server

start\_http\_server(9090)

**Security Considerations**

**Authentication & Authorization**

"""

Secure API with authentication

"""

from fastapi import FastAPI, Depends, HTTPException, status

from fastapi.security import HTTPBearer, HTTPAuthorizationCredentials

import jwt

app = FastAPI()

security = HTTPBearer()

SECRET\_KEY = "your-secret-key"

def verify\_token(credentials: HTTPAuthorizationCredentials = Depends(security)):

try:

payload = jwt.decode(credentials.credentials, SECRET\_KEY, algorithms=["HS256"])

return payload

except jwt.PyJWTError:

raise HTTPException(

status\_code=status.HTTP\_401\_UNAUTHORIZED,

detail="Invalid authentication credentials"

)

@app.post("/action")

async def execute\_action(request: dict, user=Depends(verify\_token)):

# Only authenticated users can execute actions

agent = get\_agent(user['agent\_id'])

action, info = agent.cognitive\_cycle(request['observation'])

return {"action": action, "info": info}

**Input Validation**

"""

Validate all inputs

"""

from pydantic import BaseModel, validator, Field

from typing import Tuple, List

class ObservationModel(BaseModel):

agent\_position: Tuple[float, float, float] = Field(..., description="Agent position")

target: Tuple[float, float, float] = Field(..., description="Target position")

threat\_level: float = Field(ge=0.0, le=1.0, description="Threat level 0-1")

@validator('agent\_position', 'target')

def validate\_position(cls, v):

if any(abs(x) > 1000 for x in v):

raise ValueError("Position values too large")

return v

# Use in API

@app.post("/cycle")

async def cognitive\_cycle(obs: ObservationModel):

observation = obs.dict()

# Process...

**Performance Tuning**

**Optimization Tips**

1. **Belief Management**
2. # Limit belief set size
3. if len(agent.beliefs.de\_se\_beliefs) > 1000:
4. # Keep only recent beliefs
5. agent.beliefs.de\_se\_beliefs = set(
6. list(agent.beliefs.de\_se\_beliefs)[-1000:]
7. )
8. **Verification Caching**
9. from functools import lru\_cache
10. @lru\_cache(maxsize=1000)
11. def cached\_verification(state\_hash, action\_hash):
12. # Cache verification results
13. return verifier.verify\_action\_safety(state, action)
14. **Async Processing**
15. import asyncio
16. async def async\_cognitive\_cycle(agent, observation):
17. # Run verification in parallel
18. loop = asyncio.get\_event\_loop()
19. action = await loop.run\_in\_executor(None, agent.\_plan\_action, observation)
20. is\_safe = await loop.run\_in\_executor(None, verify\_safety, action)
21. return action if is\_safe else fallback\_action

**Profiling**

"""

Profile performance bottlenecks

"""

import cProfile

import pstats

def profile\_agent():

profiler = cProfile.Profile()

profiler.enable()

# Run agent

agent = CognitiveAgent("profile\_agent")

for \_ in range(100):

agent.cognitive\_cycle({'agent\_position': (5, 5, 0)})

profiler.disable()

stats = pstats.Stats(profiler)

stats.sort\_stats('cumulative')

stats.print\_stats(20)

**Troubleshooting**

**Common Issues**

**Issue: Z3 verification slow**

**Solution:**

# Reduce verification complexity

verifier = FormalSafetyVerifier()

verifier.solver.set("timeout", 50) # 50ms timeout

**Issue: Memory growth**

**Solution:**

# Periodic cleanup

if agent.time % 1000 == 0:

agent.episode\_memory = agent.episode\_memory[-100:]

agent.beliefs.temporal\_beliefs.clear()

**Issue: High CPU usage**

**Solution:**

# Rate limit cognitive cycles

import time

MIN\_CYCLE\_TIME = 0.01 # 10ms minimum

last\_cycle = time.time()

while True:

now = time.time()

if now - last\_cycle >= MIN\_CYCLE\_TIME:

agent.cognitive\_cycle(observation)

last\_cycle = now

else:

time.sleep(MIN\_CYCLE\_TIME - (now - last\_cycle))

**Debug Mode**

"""

Enable detailed debugging

"""

import logging

# Set debug logging

logging.basicConfig(level=logging.DEBUG)

# Add debug hooks

class DebugAgent(CognitiveAgent):

def cognitive\_cycle(self, observation):

print(f"[DEBUG] Cycle {self.time}")

print(f"[DEBUG] Observation: {observation}")

action, info = super().cognitive\_cycle(observation)

print(f"[DEBUG] Action: {action.action\_type}")

print(f"[DEBUG] Safety: {not info['safety\_monitoring']['modified']}")

print(f"[DEBUG] Beliefs: {len(self.beliefs.get\_all\_beliefs())}")

return action, info

**Advanced Features**

**Load Balancing Multiple Agents**

"""

Distribute work across multiple agent instances

"""

from multiprocessing import Pool, Manager

from philosophical\_agi.core.agent import CognitiveAgent

class AgentPool:

def \_\_init\_\_(self, num\_agents=4):

self.pool = Pool(num\_agents)

self.agents = [CognitiveAgent(f"agent\_{i}") for i in range(num\_agents)]

def process\_batch(self, observations):

"""Process multiple observations in parallel"""

results = self.pool.starmap(

self.\_process\_single,

[(agent, obs) for agent, obs in zip(self.agents, observations)]

)

return results

@staticmethod

def \_process\_single(agent, observation):

return agent.cognitive\_cycle(observation)

def shutdown(self):

self.pool.close()

self.pool.join()

# Usage

pool = AgentPool(num\_agents=4)

observations = [obs1, obs2, obs3, obs4]

results = pool.process\_batch(observations)

**Distributed Deployment (Kubernetes)**

# kubernetes/deployment.yaml

apiVersion: apps/v1

kind: Deployment

metadata:

name: cognitive-agent

labels:

app: cognitive-agent

spec:

replicas: 3

selector:

matchLabels:

app: cognitive-agent

template:

metadata:

labels:

app: cognitive-agent

spec:

containers:

- name: agent

image: your-registry/cognitive-arch:latest

ports:

- containerPort: 8080

resources:

requests:

memory: "512Mi"

cpu: "500m"

limits:

memory: "1Gi"

cpu: "1000m"

env:

- name: AGENT\_ID

valueFrom:

fieldRef:

fieldPath: metadata.name

- name: LOG\_LEVEL

value: "INFO"

livenessProbe:

httpGet:

path: /health

port: 8080

initialDelaySeconds: 30

periodSeconds: 10

readinessProbe:

httpGet:

path: /ready

port: 8080

initialDelaySeconds: 5

periodSeconds: 5

---

apiVersion: v1

kind: Service

metadata:

name: cognitive-agent-service

spec:

selector:

app: cognitive-agent

ports:

- protocol: TCP

port: 80

targetPort: 8080

type: LoadBalancer

---

apiVersion: autoscaling/v2

kind: HorizontalPodAutoscaler

metadata:

name: cognitive-agent-hpa

spec:

scaleTargetRef:

apiVersion: apps/v1

kind: Deployment

name: cognitive-agent

minReplicas: 3

maxReplicas: 10

metrics:

- type: Resource

resource:

name: cpu

target:

type: Utilization

averageUtilization: 70

- type: Resource

resource:

name: memory

target:

type: Utilization

averageUtilization: 80

**CI/CD Pipeline (GitHub Actions)**

# .github/workflows/deploy.yml

name: Deploy Cognitive Architecture

on:

push:

branches: [main]

pull\_request:

branches: [main]

jobs:

test:

runs-on: ubuntu-latest

steps:

- uses: actions/checkout@v3

- name: Set up Python

uses: actions/setup-python@v4

with:

python-version: '3.11'

- name: Install dependencies

run: |

pip install -r requirements.txt

pip install pytest pytest-cov

- name: Run tests

run: |

python tests/test\_suite.py

pytest tests/ --cov=philosophical\_agi --cov-report=xml

- name: Upload coverage

uses: codecov/codecov-action@v3

with:

file: ./coverage.xml

build:

needs: test

runs-on: ubuntu-latest

if: github.event\_name == 'push' && github.ref == 'refs/heads/main'

steps:

- uses: actions/checkout@v3

- name: Set up Docker Buildx

uses: docker/setup-buildx-action@v2

- name: Login to DockerHub

uses: docker/login-action@v2

with:

username: ${{ secrets.DOCKER\_USERNAME }}

password: ${{ secrets.DOCKER\_PASSWORD }}

- name: Build and push

uses: docker/build-push-action@v4

with:

context: .

push: true

tags: your-username/cognitive-arch:latest

cache-from: type=gha

cache-to: type=gha,mode=max

deploy:

needs: build

runs-on: ubuntu-latest

if: github.event\_name == 'push' && github.ref == 'refs/heads/main'

steps:

- uses: actions/checkout@v3

- name: Configure AWS credentials

uses: aws-actions/configure-aws-credentials@v2

with:

aws-access-key-id: ${{ secrets.AWS\_ACCESS\_KEY\_ID }}

aws-secret-access-key: ${{ secrets.AWS\_SECRET\_ACCESS\_KEY }}

aws-region: us-east-1

- name: Deploy to ECS

run: |

aws ecs update-service \

--cluster cognitive-cluster \

--service cognitive-agent \

--force-new-deployment

**Production Checklist**

**Pre-Deployment**

* [ ] All tests passing (unit, integration, performance)
* [ ] Security audit completed
* [ ] Performance benchmarks meet targets
* [ ] Configuration validated
* [ ] Monitoring and alerting configured
* [ ] Backup and recovery plan in place
* [ ] Documentation complete

**Deployment**

* [ ] Blue-green deployment strategy
* [ ] Canary testing (10% traffic initially)
* [ ] Health checks validated
* [ ] Rollback plan ready
* [ ] On-call engineer assigned

**Post-Deployment**

* [ ] Monitor key metrics (latency, error rate, CPU, memory)
* [ ] Check logs for errors/warnings
* [ ] Verify safety verification success rate
* [ ] Test critical paths
* [ ] Gradual traffic ramp-up

**Monitoring Dashboard**

**Grafana Dashboard JSON**

{

"dashboard": {

"title": "Cognitive Architecture Monitoring",

"panels": [

{

"title": "Cognitive Cycles/sec",

"targets": [

{

"expr": "rate(cognitive\_cycles\_total[5m])"

}

]

},

{

"title": "Safety Violations",

"targets": [

{

"expr": "rate(safety\_violations\_total[5m])"

}

]

},

{

"title": "Cycle Duration",

"targets": [

{

"expr": "histogram\_quantile(0.95, rate(cycle\_duration\_seconds\_bucket[5m]))"

}

]

},

{

"title": "Active Beliefs",

"targets": [

{

"expr": "active\_beliefs"

}

]

}

]

}

}

**Alert Rules**

# prometheus/alerts.yml

groups:

- name: cognitive\_agent\_alerts

rules:

- alert: HighSafetyViolationRate

expr: rate(safety\_violations\_total[5m]) > 0.1

for: 5m

labels:

severity: critical

annotations:

summary: "High safety violation rate detected"

description: "Safety violations > 0.1/sec for 5 minutes"

- alert: SlowCognitiveCycles

expr: histogram\_quantile(0.95, rate(cycle\_duration\_seconds\_bucket[5m])) > 0.05

for: 10m

labels:

severity: warning

annotations:

summary: "Cognitive cycles are slow"

description: "95th percentile cycle time > 50ms"

- alert: MemoryUsageHigh

expr: process\_resident\_memory\_bytes > 1073741824 # 1GB

for: 5m

labels:

severity: warning

annotations:

summary: "High memory usage"

description: "Memory usage > 1GB for 5 minutes"

**Disaster Recovery**

**Backup Strategy**

"""

Backup agent state periodically

"""

import pickle

import gzip

from datetime import datetime

class StateBackup:

def \_\_init\_\_(self, backup\_dir='./backups'):

self.backup\_dir = backup\_dir

os.makedirs(backup\_dir, exist\_ok=True)

def backup\_agent(self, agent):

"""Backup agent state"""

timestamp = datetime.utcnow().strftime('%Y%m%d\_%H%M%S')

filename = f"{self.backup\_dir}/agent\_{agent.id}\_{timestamp}.pkl.gz"

state = {

'agent\_id': agent.id,

'time': agent.time,

'beliefs': {

'de\_se': list(agent.beliefs.de\_se\_beliefs),

'de\_dicto': list(agent.beliefs.de\_dicto\_beliefs)

},

'episode\_memory': agent.episode\_memory[-100:], # Last 100 episodes

'safety\_stats': agent.safety\_monitor.get\_statistics()

}

with gzip.open(filename, 'wb') as f:

pickle.dump(state, f)

return filename

def restore\_agent(self, agent, filename):

"""Restore agent from backup"""

with gzip.open(filename, 'rb') as f:

state = pickle.load(f)

agent.id = state['agent\_id']

agent.time = state['time']

agent.beliefs.de\_se\_beliefs = set(state['beliefs']['de\_se'])

agent.beliefs.de\_dicto\_beliefs = set(state['beliefs']['de\_dicto'])

agent.episode\_memory = state['episode\_memory']

return agent

# Usage

backup = StateBackup()

# Periodic backup (every hour)

import schedule

def backup\_job():

backup.backup\_agent(agent)

print("Backup completed")

schedule.every().hour.do(backup\_job)

**Recovery Procedures**

#!/bin/bash

# recovery.sh - Disaster recovery script

echo "=== Cognitive Architecture Disaster Recovery ==="

# 1. Stop current instance

echo "Stopping current instance..."

docker stop cognitive-agent || true

# 2. Restore from backup

echo "Restoring from latest backup..."

LATEST\_BACKUP=$(ls -t backups/agent\_\*.pkl.gz | head -1)

echo "Using backup: $LATEST\_BACKUP"

# 3. Start new instance with restored state

echo "Starting new instance..."

docker run -d \

--name cognitive-agent-recovered \

-v $(pwd)/backups:/app/backups:ro \

-e RESTORE\_FROM=$LATEST\_BACKUP \

cognitive-arch:latest

# 4. Health check

echo "Waiting for health check..."

for i in {1..30}; do

if curl -f http://localhost:8080/health; then

echo "✅ Recovery successful!"

exit 0

fi

sleep 2

done

echo "❌ Recovery failed - manual intervention required"

exit 1

**Cost Optimization**

**Cloud Cost Estimates**

| **Platform** | **Configuration** | **Monthly Cost** |
| --- | --- | --- |
| AWS EC2 | t3.medium (2 vCPU, 4GB) | ~$30 |
| AWS ECS Fargate | 0.5 vCPU, 1GB | ~$15 |
| Google Cloud Run | Auto-scale 0-10 instances | ~$5-50 |
| Azure Container Instances | 1 vCPU, 1GB | ~$30 |
| Self-hosted | Raspberry Pi 4 | ~$5 (electricity) |

**Cost Reduction Tips**

1. **Use spot instances** (AWS/GCP): 70% cost savings
2. **Auto-scaling**: Scale to zero during low usage
3. **Reserved instances**: 40% savings for steady workloads
4. **Right-sizing**: Monitor and adjust resources
5. **Multi-tenant**: Run multiple agents per container

**Success Metrics**

**Key Performance Indicators (KPIs)**

"""

Track production KPIs

"""

class ProductionMetrics:

def \_\_init\_\_(self):

self.metrics = {

'uptime': 0.0,

'total\_cycles': 0,

'safety\_success\_rate': 0.0,

'avg\_cycle\_time': 0.0,

'p95\_cycle\_time': 0.0,

'memory\_usage': 0.0,

'error\_rate': 0.0

}

def calculate\_kpis(self, agent):

"""Calculate KPIs from agent"""

stats = agent.safety\_monitor.get\_statistics()

return {

'uptime': self.\_calculate\_uptime(),

'total\_cycles': agent.time,

'safety\_success\_rate': stats['verifier\_stats']['success\_rate'],

'avg\_cycle\_time': self.\_get\_avg\_cycle\_time(),

'error\_rate': stats['total\_violations'] / max(1, agent.time),

'beliefs\_growth': len(agent.beliefs.get\_all\_beliefs()) / max(1, agent.time)

}

**Target Metrics**

| **Metric** | **Target** | **Critical Threshold** |
| --- | --- | --- |
| Uptime | > 99.9% | < 99% |
| Safety Success Rate | > 99% | < 95% |
| Avg Cycle Time | < 10ms | > 50ms |
| P95 Cycle Time | < 20ms | > 100ms |
| Error Rate | < 0.1% | > 1% |
| Memory Usage | < 500MB | > 1GB |

**Support & Maintenance**

**Update Procedures**

#!/bin/bash

# Rolling update with zero downtime

echo "=== Rolling Update ==="

# 1. Deploy new version alongside old

docker run -d --name cognitive-agent-new cognitive-arch:v2

# 2. Health check

if ! curl -f http://localhost:8081/health; then

echo "❌ New version failed health check"

docker stop cognitive-agent-new

exit 1

fi

# 3. Switch traffic (using load balancer)

# Configure load balancer to route to new instance

# 4. Monitor for issues

sleep 60

# 5. Decommission old version

docker stop cognitive-agent-old

docker rm cognitive-agent-old

echo "✅ Update complete"

**Maintenance Windows**

"""

Schedule maintenance windows

"""

from datetime import datetime, time

class MaintenanceScheduler:

def \_\_init\_\_(self):

self.maintenance\_windows = [

(time(2, 0), time(4, 0)) # 2 AM - 4 AM

]

def is\_maintenance\_window(self):

"""Check if current time is in maintenance window"""

now = datetime.now().time()

for start, end in self.maintenance\_windows:

if start <= now <= end:

return True

return False

def safe\_to\_update(self, agent):

"""Check if safe to perform update"""

return (

self.is\_maintenance\_window() and

len(agent.episode\_memory) < 10 and # Low activity

agent.safety\_monitor.violations == 0 # No recent violations

)

**Conclusion**

This deployment guide provides a comprehensive roadmap for taking the cognitive architecture from prototype to production. Key takeaways:

1. **Start simple**: Docker + single instance
2. **Add monitoring**: Prometheus + Grafana
3. **Scale gradually**: Kubernetes when needed
4. **Prioritize safety**: Multiple verification layers
5. **Plan for failure**: Backups + recovery procedures

For questions or issues:

* GitHub Issues: https://github.com/yourusername/philosophical-agi/issues
* Documentation: https://docs.example.com
* Email: support@example.com

**Remember**: This is a safety-critical system. Always test thoroughly before deploying to production.