🛠️ DIMIA IMPLEMENTATION GUIDE

Phase 5A Technical Implementation

Version: 5A.1.0

Target Platform: Bio-Quantum AI Trading Platform

Implementation Timeline: 2-3 weeks

📋 IMPLEMENTATION OVERVIEW

This guide provides step-by-step instructions for implementing the DNA-Inspired Middleware Integration Architecture (DIMIA) into the Bio-Quantum AI platform. The implementation follows a modular approach that allows for incremental deployment and testing.

Implementation Phases:

1. 🏗️ **Foundation Setup** (Week 1)
2. 🧬 **Core Integration** (Week 2)
3. 🚀 **Production Deployment** (Week 3)

🏗️ PHASE 1: FOUNDATION SETUP

1.1 Environment Preparation

System Requirements:

Bash

```
# Python Environment
Python 3.11+
pip 23.0+
virtualenv or conda

# Database
SQLite 3.35+
(PostgreSQL 13+ for production)

# Dependencies
numpy>=1.21.0
asyncio (built-in)
sqlite3 (built-in)
hashlib (built-in)
json (built-in)
logging (built-in)
```

Development Environment Setup:

Bash

```
# 1. Create virtual environment
python -m venv dimia_env
source dimia_env/bin/activate  # Linux/Mac
# dimia_env\Scripts\activate  # Windows

# 2. Install dependencies
pip install numpy pandas asyncio-mqtt websockets

# 3. Verify installation
python -c "import numpy, asyncio, sqlite3; print('Dependencies OK')"
```

1.2 Directory Structure Setup

Bash

```
# Create Bio-Quantum AI integration directory
mkdir -p bio_quantum_ai/integrations/dimia
cd bio_quantum_ai/integrations/dimia

# Copy DIMIA framework
cp -r PHASE_5A_IMPLEMENTATION_PACKAGE/DIMIA_Discovery_Agent/* .
```

```
# Create additional directories
mkdir -p {config,logs,data,tests,docs}
```

1.3 Configuration Setup

Create `config/dimia_config.json`:

JSON

```
{
  "discovery_agent": {
    "scan_interval": 300,
    "max_concurrent_scans": 5,
    "timeout_seconds": 30,
    "retry_attempts": 3
  },
  "knowledge_codex": {
    "database_path": "data/knowledge_codex.db",
    "backup_interval": 3600,
    "max_cache_size": 1000
  },
  "ai_orchestration": {
    "learning_rate": 0.01,
    "optimization_interval": 600,
    "confidence_threshold": 0.7,
    "max_recommendations": 10
  },
  "codon_management": {
    "max_active_codons": 20,
    "health_check_interval": 60,
    "auto_restart": true,
    "performance_threshold": 0.8
  },
  "logging": {
    "level": "INFO",
    "file_path": "logs/dimia.log",
    "max_file_size": "10MB",
    "backup_count": 5
  }
}
```

Create `config/platform_credentials.json`:

JSON

```
{
  "tradingview": {
    "api_key": "YOUR_TRADINGVIEW_API_KEY",
    "username": "YOUR_USERNAME",
    "rate_limit": 500
  },
  "metatrader": {
    "server": "MetaQuotes-Demo",
    "login": 12345678,
    "password": "YOUR_PASSWORD",
    "path": "/path/to/metatrader"
  },
  "interactive_brokers": {
    "username": "YOUR_IB_USERNAME",
    "password": "YOUR_IB_PASSWORD",
    "trading_mode": "paper"
  }
}
```

🧬 PHASE 2: CORE INTEGRATION

2.1 Knowledge Codex Initialization

Initialize Database:

Python

```python
# scripts/init_knowledge_codex.py
import asyncio
import sys
import os

# Add DIMIA to path
sys.path.append(os.path.join(os.path.dirname(__file__), '..', 'src'))

from database.knowledge_codex import KnowledgeCodex

async def initialize_codex():
    """Initialize the Knowledge Codex with platform data"""
    print("🚀 Initializing DIMIA Knowledge Codex...")

    # Create codex instance
```

```
    codex = KnowledgeCodex("data/knowledge_codex.db")

    # Wait for initial data population
    await asyncio.sleep(2)

    # Verify initialization
    stats = await codex.db.get_taxonomy_stats()
    print(f"✅ Codex initialized with {stats['total_platforms']} platforms")

    # Close codex
    await codex.close()
    print("📚 Knowledge Codex ready for use")

if __name__ == "__main__":
    asyncio.run(initialize_codex())
```

Run Initialization:

Bash

```
cd bio_quantum_ai/integrations/dimia
python scripts/init_knowledge_codex.py
```

2.2 Discovery Agent Integration

Create Discovery Service:

Python

```
# services/discovery_service.py
import asyncio
import json
import logging
from typing import Dict, List, Optional

from src.core.discovery_agent import DiscoveryAgent
from src.database.knowledge_codex import KnowledgeCodex

logger = logging.getLogger(__name__)

class DIMIADiscoveryService:
    """DIMIA Discovery Service for Bio-Quantum AI"""

    def __init__(self, config_path: str = "config/dimia_config.json"):
```

```
        with open(config_path, 'r') as f:
            self.config = json.load(f)

        self.discovery_agent = None
        self.knowledge_codex = None
        self.running = False

    async def start(self):
        """Start the discovery service"""
        logger.info("🚀 Starting DIMIA Discovery Service...")

        # Initialize components
        self.discovery_agent = DiscoveryAgent()
        self.knowledge_codex = KnowledgeCodex()

        # Start discovery agent
        await self.discovery_agent.start()

        self.running = True
        logger.info("✅ DIMIA Discovery Service started")

    async def stop(self):
        """Stop the discovery service"""
        self.running = False

        if self.discovery_agent:
            await self.discovery_agent.stop()

        if self.knowledge_codex:
            await self.knowledge_codex.close()

        logger.info("🔴 DIMIA Discovery Service stopped")

    async def discover_platforms(self, user_context: Dict) -> List[Dict]:
        """Discover available platforms for user"""
        if not self.running:
            raise RuntimeError("Discovery service not running")

        # Perform discovery scan
        discovered = await
self.discovery_agent.scan_environment(user_context)

        # Enrich with codex data
        enriched_platforms = []
        for platform in discovered:
            codex_data = await
self.knowledge_codex.get_platform(platform.platform_id)
            if codex_data:
```

```
                enriched_platforms.append({
                    'discovered': platform.to_dict(),
                    'codex_data': codex_data.to_dict()
                })

        return enriched_platforms

    async def get_integration_guidance(self, platform_id: str) -> Dict:
        """Get integration guidance for a platform"""
        return await
self.knowledge_codex.get_integration_guidance(platform_id)

# Integration with Bio-Quantum AI
discovery_service = DIMIADiscoveryService()
```

2.3 Codon Management Integration

Create Codon Manager:

Python

```
# services/codon_manager.py
import asyncio
import json
import logging
from typing import Dict, List, Optional

from src.utils.codon_factory import CodonFactory
from src.codons.tradingview_codon import create_tradingview_codon
from src.codons.metatrader_codon import create_metatrader_codon

logger = logging.getLogger(__name__)

class DIMIACodonManager:
    """DIMIA Codon Manager for Bio-Quantum AI"""

    def __init__(self, config_path: str = "config/dimia_config.json"):
        with open(config_path, 'r') as f:
            self.config = json.load(f)

        self.active_codons = {}
        self.codon_factory = CodonFactory()
        self.running = False

    async def start(self):
        """Start the codon manager"""
```

```
        logger.info("🧬 Starting DIMIA Codon Manager...")
        self.running = True
        logger.info("✅ DIMIA Codon Manager started")

    async def stop(self):
        """Stop the codon manager"""
        self.running = False

        # Cleanup all active codons
        for codon_id in list(self.active_codons.keys()):
            await self.deactivate_codon(codon_id)

        logger.info("⏹️ DIMIA Codon Manager stopped")

    async def activate_codon(self, platform_name: str, credentials: Dict) ->
str:
        """Activate a codon for a specific platform"""
        logger.info(f"🧬 Activating codon for {platform_name}...")

        try:
            # Create codon based on platform
            if platform_name.lower() == "tradingview":
                codon = await create_tradingview_codon(
                    api_key=credentials.get('api_key'),
                    username=credentials.get('username')
                )
            elif platform_name.lower() == "metatrader":
                codon = await create_metatrader_codon(
                    server=credentials.get('server', 'MetaQuotes-Demo'),
                    login=credentials.get('login'),
                    password=credentials.get('password')
                )
            else:
                raise ValueError(f"Unsupported platform: {platform_name}")

            # Store active codon
            codon_id = f"{platform_name.lower()}_{len(self.active_codons) +
1:03d}"
            self.active_codons[codon_id] = codon

            logger.info(f"✅ Codon activated: {codon_id}")
            return codon_id

        except Exception as e:
            logger.error(f"❌ Failed to activate codon for {platform_name}:
{str(e)}")
            raise
```

```
    async def deactivate_codon(self, codon_id: str) -> bool:
        """Deactivate a specific codon"""
        if codon_id not in self.active_codons:
            return False

        try:
            codon = self.active_codons[codon_id]
            await codon.cleanup()
            del self.active_codons[codon_id]

            logger.info(f"🧬 Codon deactivated: {codon_id}")
            return True

        except Exception as e:
            logger.error(f"❌ Failed to deactivate codon {codon_id}:
{str(e)}")
            return False

    async def execute_codon_operation(self, codon_id: str, operation: str,
**kwargs):
        """Execute an operation on a specific codon"""
        if codon_id not in self.active_codons:
            raise ValueError(f"Codon not found: {codon_id}")

        codon = self.active_codons[codon_id]
        return await codon.execute(operation, **kwargs)

    def get_active_codons(self) -> Dict:
        """Get status of all active codons"""
        return {
            codon_id: codon.get_status()
            for codon_id, codon in self.active_codons.items()
        }

# Integration with Bio-Quantum AI
codon_manager = DIMIACodonManager()
```

2.4 AI Orchestration Integration

Create Orchestration Service:

Python

```
# services/orchestration_service.py
import asyncio
import json
```

```python
import logging
from typing import Dict, List, Optional

from src.ai.orchestration_engine import OrchestrationEngine

logger = logging.getLogger(__name__)

class DIMIAOrchestrationService:
    """DIMIA AI Orchestration Service for Bio-Quantum AI"""

    def __init__(self, config_path: str = "config/dimia_config.json"):
        with open(config_path, 'r') as f:
            self.config = json.load(f)

        self.orchestration_engine = OrchestrationEngine()
        self.running = False

    async def start(self):
        """Start the orchestration service"""
        logger.info("🧠 Starting DIMIA AI Orchestration Service...")

        await self.orchestration_engine.start()
        self.running = True

        logger.info("✅ DIMIA AI Orchestration Service started")

    async def stop(self):
        """Stop the orchestration service"""
        self.running = False
        await self.orchestration_engine.stop()
        logger.info("🔲 DIMIA AI Orchestration Service stopped")

    async def record_user_behavior(self, user_id: str, action: str,
                                   context: Dict, outcome: Optional[str] =
None):
        """Record user behavior for AI learning"""
        await self.orchestration_engine.record_user_behavior(
            user_id, action, context, outcome
        )

    async def get_recommendations(self, user_id: str) -> List[Dict]:
        """Get AI-generated optimization recommendations"""
        recommendations = await
self.orchestration_engine.get_recommendations(user_id)
        return [rec.to_dict() for rec in recommendations]

    async def register_codon(self, codon_id: str, codon_info: Dict):
        """Register a codon with the AI orchestration engine"""
```

```
        await self.orchestration_engine.register_codon(codon_id, codon_info)

    async def record_codon_performance(self, codon_id: str, metrics: Dict):
        """Record codon performance metrics for AI analysis"""
        await self.orchestration_engine.record_codon_performance(codon_id,
metrics)

# Integration with Bio-Quantum AI
orchestration_service = DIMIAOrchestrationService()
```

🚀 PHASE 3: PRODUCTION DEPLOYMENT

3.1 Bio-Quantum AI Integration

Main DIMIA Service:

Python

```python
# dimia_service.py
import asyncio
import logging
from typing import Dict, List, Optional

from services.discovery_service import DIMIADiscoveryService
from services.codon_manager import DIMIACodonManager
from services.orchestration_service import DIMIAOrchestrationService

logger = logging.getLogger(__name__)

class DIMIAService:
    """Main DIMIA service for Bio-Quantum AI integration"""

    def __init__(self):
        self.discovery_service = DIMIADiscoveryService()
        self.codon_manager = DIMIACodonManager()
        self.orchestration_service = DIMIAOrchestrationService()
        self.running = False

    async def start(self):
        """Start all DIMIA services"""
        logger.info("🚀 Starting DIMIA Services...")

        # Start services in order
```

```
        await self.discovery_service.start()
        await self.codon_manager.start()
        await self.orchestration_service.start()

        self.running = True
        logger.info("✅ All DIMIA services started successfully")

    async def stop(self):
        """Stop all DIMIA services"""
        logger.info("⏹️ Stopping DIMIA Services...")

        self.running = False

        # Stop services in reverse order
        await self.orchestration_service.stop()
        await self.codon_manager.stop()
        await self.discovery_service.stop()

        logger.info("✅ All DIMIA services stopped")

    # Public API methods for Bio-Quantum AI

    async def discover_platforms(self, user_id: str, context: Dict) ->
List[Dict]:
        """Discover available platforms for user"""
        return await self.discovery_service.discover_platforms(context)

    async def activate_integration(self, user_id: str, platform_name: str,
                                   credentials: Dict) -> str:
        """Activate platform integration for user"""
        # Activate codon
        codon_id = await self.codon_manager.activate_codon(platform_name,
credentials)

        # Register with AI orchestration
        await self.orchestration_service.register_codon(codon_id, {
            'platform': platform_name,
            'user_id': user_id,
            'activated_at': asyncio.get_event_loop().time()
        })

        # Record user behavior
        await self.orchestration_service.record_user_behavior(
            user_id, 'activate_integration',
            {'platform': platform_name}, 'success'
        )

        return codon_id
```

```
    async def execute_platform_operation(self, codon_id: str, operation: str,
**kwargs):
        """Execute operation on integrated platform"""
        return await self.codon_manager.execute_codon_operation(codon_id,
operation, **kwargs)

    async def get_user_recommendations(self, user_id: str) -> List[Dict]:
        """Get AI-generated recommendations for user"""
        return await self.orchestration_service.get_recommendations(user_id)

    async def get_integration_status(self) -> Dict:
        """Get overall DIMIA integration status"""
        return {
            'running': self.running,
            'active_codons': self.codon_manager.get_active_codons(),
            'orchestration_status':
self.orchestration_service.orchestration_engine.get_engine_status()
        }

# Global DIMIA service instance
dimia_service = DIMIAService()
```

3.2 Bio-Quantum AI API Integration

Add DIMIA endpoints to Bio-Quantum AI API:

Python

```
# In your existing Bio-Quantum AI Flask/FastAPI app

from dimia_service import dimia_service

# Initialize DIMIA on app startup
@app.on_event("startup")
async def startup_dimia():
    await dimia_service.start()

@app.on_event("shutdown")
async def shutdown_dimia():
    await dimia_service.stop()

# DIMIA API endpoints
@app.get("/api/dimia/discover")
async def discover_platforms(user_id: str, context: dict = None):
    """Discover available platforms for integration"""
```

```
    context = context or {}
    platforms = await dimia_service.discover_platforms(user_id, context)
    return {"platforms": platforms}

@app.post("/api/dimia/integrate")
async def activate_integration(user_id: str, platform_name: str, credentials:
dict):
    """Activate platform integration"""
    codon_id = await dimia_service.activate_integration(user_id,
platform_name, credentials)
    return {"codon_id": codon_id, "status": "activated"}

@app.post("/api/dimia/execute")
async def execute_operation(codon_id: str, operation: str, parameters: dict =
None):
    """Execute operation on integrated platform"""
    parameters = parameters or {}
    result = await dimia_service.execute_platform_operation(codon_id,
operation, **parameters)
    return {"result": result}

@app.get("/api/dimia/recommendations")
async def get_recommendations(user_id: str):
    """Get AI-generated optimization recommendations"""
    recommendations = await dimia_service.get_user_recommendations(user_id)
    return {"recommendations": recommendations}

@app.get("/api/dimia/status")
async def get_status():
    """Get DIMIA system status"""
    status = await dimia_service.get_integration_status()
    return status
```

3.3 Frontend Integration

React Component for Codon Management:

JSX

```
// components/DIMIACodonDashboard.jsx
import React, { useState, useEffect } from 'react';
import { Card, Button, Badge, Progress } from 'your-ui-library';

const DIMIACodonDashboard = ({ userId }) => {
  const [codons, setCodons] = useState([]);
  const [recommendations, setRecommendations] = useState([]);
```

```
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    loadDIMIAData();
  }, [userId]);

  const loadDIMIAData = async () => {
    try {
      // Load active codons
      const statusResponse = await fetch('/api/dimia/status');
      const status = await statusResponse.json();
      setCodons(Object.entries(status.active_codons || {}));

      // Load recommendations
      const recResponse = await fetch(`/api/dimia/recommendations?
user_id=${userId}`);
      const recData = await recResponse.json();
      setRecommendations(recData.recommendations || []);

      setLoading(false);
    } catch (error) {
      console.error('Failed to load DIMIA data:', error);
      setLoading(false);
    }
  };

  const activateIntegration = async (platformName, credentials) => {
    try {
      const response = await fetch('/api/dimia/integrate', {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify({
          user_id: userId,
          platform_name: platformName,
          credentials: credentials
        })
      });

      const result = await response.json();
      console.log('Integration activated:', result.codon_id);
      loadDIMIAData(); // Refresh data
    } catch (error) {
      console.error('Failed to activate integration:', error);
    }
  };

  if (loading) {
    return <div className="dimia-loading">🧬 Loading DIMIA...</div>;
```

```
  }

  return (
    <div className="dimia-dashboard">
      <h2>🧬 DIMIA Integration Dashboard</h2>

      {/* Active Codons */}
      <section className="active-codons">
        <h3>Active Integrations</h3>
        {codons.length === 0 ? (
          <p>No active integrations. Discover platforms to get started!</p>
        ) : (
          <div className="codon-grid">
            {codons.map(([codonId, status]) => (
              <Card key={codonId} className="codon-card">
                <div className="codon-header">
                  <h4>{status.platform_name}</h4>
                  <Badge variant={status.active ? 'success' : 'warning'}>
                    {status.active ? 'Active' : 'Inactive'}
                  </Badge>
                </div>
                <div className="codon-metrics">
                  <Progress
                    value={status.success_rate * 100}
                    label="Success Rate"
                  />
                  <div className="metric">
                    <span>Operations: {status.success_count +
status.error_count}</span>
                  </div>
                </div>
              </Card>
            ))}
          </div>
        )}
      </section>

      {/* AI Recommendations */}
      <section className="recommendations">
        <h3>🧠 AI Recommendations</h3>
        {recommendations.length === 0 ? (
          <p>No recommendations available. Use the platform to generate
insights!</p>
        ) : (
          <div className="recommendation-list">
            {recommendations.map((rec) => (
              <Card key={rec.recommendation_id} className="recommendation-
card">
```

```
                <div className="rec-header">
                  <h4>{rec.optimization_type.replace('_', ' ').toUpperCase()}
</h4>
                  <Badge variant="info">
                    {(rec.confidence * 100).toFixed(0)}% confidence
                  </Badge>
                </div>
                <p>{rec.reasoning}</p>
                <div className="rec-impact">
                  Expected improvement: {(rec.expected_improvement *
100).toFixed(0)}%
                </div>
              </Card>
            ))}
          </div>
        )}
      </section>
    </div>
  );
};

export default DIMIACodonDashboard;
```

🧪 TESTING & VALIDATION

4.1 Unit Testing

Create test suite:

Python

```
# tests/test_dimia_integration.py
import pytest
import asyncio
from unittest.mock import Mock, patch

from services.discovery_service import DIMIADiscoveryService
from services.codon_manager import DIMIACodonManager
from services.orchestration_service import DIMIAOrchestrationService

class TestDIMIAIntegration:

    @pytest.fixture
```

```python
    async def discovery_service(self):
        service = DIMIADiscoveryService()
        await service.start()
        yield service
        await service.stop()

    @pytest.fixture
    async def codon_manager(self):
        manager = DIMIACodonManager()
        await manager.start()
        yield manager
        await manager.stop()

    async def test_discovery_service_startup(self, discovery_service):
        """Test discovery service starts correctly"""
        assert discovery_service.running
        assert discovery_service.discovery_agent is not None
        assert discovery_service.knowledge_codex is not None

    async def test_codon_activation(self, codon_manager):
        """Test codon activation and deactivation"""
        # Mock credentials
        credentials = {
            'api_key': 'test_key',
            'username': 'test_user'
        }

        # Activate TradingView codon
        codon_id = await codon_manager.activate_codon('tradingview',
credentials)
        assert codon_id in codon_manager.active_codons

        # Test codon operation
        result = await codon_manager.execute_codon_operation(
            codon_id, 'get_quote', symbol='AAPL'
        )
        assert 'last' in result

        # Deactivate codon
        success = await codon_manager.deactivate_codon(codon_id)
        assert success
        assert codon_id not in codon_manager.active_codons

    async def test_ai_recommendations(self):
        """Test AI recommendation generation"""
        orchestration = DIMIAOrchestrationService()
        await orchestration.start()
```

```
        # Record some user behavior
        await orchestration.record_user_behavior(
            'test_user', 'view_chart',
            {'platform': 'TradingView', 'symbol': 'AAPL'},
            'success'
        )

        # Get recommendations
        recommendations = await
orchestration.get_recommendations('test_user')
        assert isinstance(recommendations, list)

        await orchestration.stop()

# Run tests
if __name__ == "__main__":
    pytest.main([__file__])
```

4.2 Integration Testing

Create integration test script:

Bash

```
#!/bin/bash
# scripts/test_integration.sh

echo "🧪 Running DIMIA Integration Tests..."

# Test 1: Knowledge Codex initialization
echo "📚 Testing Knowledge Codex..."
python -c "
import asyncio
from src.database.knowledge_codex import KnowledgeCodex

async def test():
    codex = KnowledgeCodex(':memory:')
    await asyncio.sleep(1)
    stats = await codex.db.get_taxonomy_stats()
    assert stats['total_platforms'] > 0
    await codex.close()
    print('✅ Knowledge Codex test passed')

asyncio.run(test())
"
```

```
# Test 2: Discovery Agent
echo "🔍 Testing Discovery Agent..."
python -c "
import asyncio
from src.core.discovery_agent import DiscoveryAgent

async def test():
    agent = DiscoveryAgent()
    await agent.start()

    # Test platform detection
    context = {'user_agent': 'test', 'url': 'https://tradingview.com'}
    platforms = await agent.scan_environment(context)

    await agent.stop()
    print('✅ Discovery Agent test passed')

asyncio.run(test())
"

# Test 3: Codon functionality
echo "🧬 Testing Codon Operations..."
python -c "
import asyncio
from src.codons.tradingview_codon import create_tradingview_codon

async def test():
    codon = await create_tradingview_codon()

    # Test basic operations
    symbols = await codon.execute('search_symbols', query='AAPL')
    assert len(symbols) > 0

    quote = await codon.execute('get_quote', symbol='AAPL')
    assert 'last' in quote

    await codon.cleanup()
    print('✅ Codon operations test passed')

asyncio.run(test())
"

echo "✅ All integration tests passed!"
```

📊 MONITORING & MAINTENANCE

5.1 Performance Monitoring

Create monitoring dashboard:

Python

```python
# monitoring/dimia_monitor.py
import asyncio
import json
import time
from typing import Dict, List
from dataclasses import dataclass
from datetime import datetime

@dataclass
class PerformanceMetrics:
    timestamp: datetime
    active_codons: int
    discovery_scans: int
    ai_recommendations: int
    average_response_time: float
    error_rate: float
    memory_usage: float
    cpu_usage: float

class DIMIAMonitor:
    """Performance monitoring for DIMIA services"""

    def __init__(self, dimia_service):
        self.dimia_service = dimia_service
        self.metrics_history = []
        self.monitoring = False

    async def start_monitoring(self, interval: int = 60):
        """Start performance monitoring"""
        self.monitoring = True

        while self.monitoring:
            metrics = await self.collect_metrics()
            self.metrics_history.append(metrics)

            # Keep only last 24 hours of metrics
            cutoff = datetime.now().timestamp() - (24 * 3600)
            self.metrics_history = [
```

```
                m for m in self.metrics_history
                if m.timestamp.timestamp() > cutoff
            ]

            # Log metrics
            self.log_metrics(metrics)

            await asyncio.sleep(interval)

    async def collect_metrics(self) -> PerformanceMetrics:
        """Collect current performance metrics"""
        status = await self.dimia_service.get_integration_status()

        return PerformanceMetrics(
            timestamp=datetime.now(),
            active_codons=len(status.get('active_codons', {})),
            discovery_scans=0,  # Would track from discovery service
            ai_recommendations=0,  # Would track from orchestration service
            average_response_time=0.5,  # Would calculate from actual metrics
            error_rate=0.02,  # Would calculate from error logs
            memory_usage=50.0,  # Would get from system monitoring
            cpu_usage=25.0  # Would get from system monitoring
        )

    def log_metrics(self, metrics: PerformanceMetrics):
        """Log performance metrics"""
        print(f"📊 DIMIA Metrics
[{metrics.timestamp.strftime('%H:%M:%S')}]:")
        print(f"  Active Codons: {metrics.active_codons}")
        print(f"  Response Time: {metrics.average_response_time:.2f}s")
        print(f"  Error Rate: {metrics.error_rate:.2%}")
        print(f"  Memory: {metrics.memory_usage:.1f}%")
        print(f"  CPU: {metrics.cpu_usage:.1f}%")

    def get_health_status(self) -> Dict:
        """Get overall health status"""
        if not self.metrics_history:
            return {"status": "unknown", "message": "No metrics available"}

        latest = self.metrics_history[-1]

        # Health checks
        issues = []
        if latest.error_rate > 0.1:
            issues.append("High error rate")
        if latest.average_response_time > 5.0:
            issues.append("Slow response times")
        if latest.memory_usage > 80.0:
```

```
            issues.append("High memory usage")
        if latest.cpu_usage > 80.0:
            issues.append("High CPU usage")

        if not issues:
            return {"status": "healthy", "message": "All systems
operational"}
        else:
            return {"status": "warning", "issues": issues}
```

5.2 Automated Maintenance

Create maintenance script:

Python

```
# scripts/maintenance.py
import asyncio
import logging
import os
from datetime import datetime, timedelta

async def perform_maintenance():
    """Perform automated maintenance tasks"""
    print("🔧 Starting DIMIA maintenance...")

    # 1. Clean up old log files
    log_dir = "logs"
    if os.path.exists(log_dir):
        cutoff_date = datetime.now() - timedelta(days=30)

        for filename in os.listdir(log_dir):
            filepath = os.path.join(log_dir, filename)
            if os.path.isfile(filepath):
                file_time =
datetime.fromtimestamp(os.path.getmtime(filepath))
                if file_time < cutoff_date:
                    os.remove(filepath)
                    print(f"🗑️ Removed old log file: {filename}")

    # 2. Backup Knowledge Codex
    import shutil
    db_path = "data/knowledge_codex.db"
    if os.path.exists(db_path):
        backup_path =
f"data/knowledge_codex_backup_{datetime.now().strftime('%Y%m%d_%H%M%S')}.db"
```

```
            shutil.copy2(db_path, backup_path)
            print(f"💾 Created database backup: {backup_path}")

        # 3. Update platform data
        from src.database.knowledge_codex import KnowledgeCodex
        codex = KnowledgeCodex()

        # Refresh platform metadata (would implement actual updates)
        print("🔄 Refreshing platform metadata...")

        await codex.close()

        print("✅ Maintenance completed successfully")

    if __name__ == "__main__":
        asyncio.run(perform_maintenance())
```

🚨 TROUBLESHOOTING

Common Issues and Solutions:

Issue 1: Codon Activation Fails

```
Plain Text

Error: Failed to activate codon for TradingView
```

Solution:

1. Check API credentials in `config/platform_credentials.json`
2. Verify network connectivity to platform
3. Check rate limits and quotas
4. Review platform-specific requirements

Issue 2: Discovery Agent Not Finding Platforms

```
Plain Text
```

```
Warning: No platforms discovered in scan
```

Solution:

1. Verify browser scanner configuration
2. Check network access to target platforms
3. Update platform signatures in Knowledge Codex
4. Review scan timeout settings

Issue 3: AI Recommendations Not Generated

```
Plain Text
Info: No recommendations available
```

Solution:

1. Ensure sufficient user behavior data
2. Check AI orchestration engine status
3. Verify confidence thresholds
4. Review learning algorithm parameters

Issue 4: Database Connection Errors

```
Plain Text
Error: Failed to connect to Knowledge Codex
```

Solution:

1. Check database file permissions
2. Verify SQLite installation
3. Review database path configuration

4. Check disk space availability

📈 PERFORMANCE OPTIMIZATION

Optimization Guidelines:

1. **Database Performance:**
   - Use connection pooling for high-traffic scenarios
   - Implement query caching for frequently accessed data
   - Regular database maintenance and optimization
2. **Codon Efficiency:**
   - Implement connection reuse for platform APIs
   - Use async/await patterns consistently
   - Monitor and optimize rate limiting
3. **AI Processing:**
   - Batch behavior analysis for efficiency
   - Implement model caching
   - Use background processing for recommendations
4. **Memory Management:**
   - Regular cleanup of cached data
   - Monitor memory usage patterns
   - Implement data retention policies

✅ DEPLOYMENT CHECKLIST

Pre-Deployment:

- [ ] All dependencies installed and verified
- [ ] Configuration files properly set up
- [ ] Database initialized with platform data
- [ ] Unit tests passing
- [ ] Integration tests passing
- [ ] Performance benchmarks met

Deployment:

- [ ] DIMIA services integrated into Bio-Quantum AI
- [ ] API endpoints added and tested
- [ ] Frontend components deployed
- [ ] Monitoring systems activated
- [ ] Backup procedures in place

Post-Deployment:

- [ ] Health checks passing
- [ ] User acceptance testing completed
- [ ] Performance monitoring active
- [ ] Documentation updated
- [ ] Team training completed

📞 SUPPORT CONTACTS

Technical Support:

- **Lead Developer:** Manus AI Agent
- **Documentation:** Complete guides in `/docs` directory
- **Issue Tracking:** GitHub repository issues
- **Emergency Contact:** Bio-Quantum AI development team

Resources:

- **API Documentation:** `/docs/api_reference.md`
- **Architecture Guide:** `/docs/architecture.md`
- **Best Practices:** `/docs/best_practices.md`
- **FAQ:** `/docs/faq.md`

Implementation Guide prepared by Manus AI Agent

Bio-Quantum AI Trading Platform

January 7, 2025