# Development Guide & Best Practices

## 🎯 Tổng quan

Document này cung cấp comprehensive development guide cho MAS-Planning system, bao gồm setup instructions, coding standards, testing strategies, và deployment best practices.

## 🛠️ Development Environment Setup

### **1. System Requirements**

# Minimum Requirements  
- Python 3.8+  
- Node.js 16+ (for MCP server if needed)  
- Redis Server 6+  
- Git 2.30+  
- Docker & Docker Compose (optional)  
  
# Recommended Development Tools  
- VS Code với Python extension  
- PyCharm Professional  
- Postman cho API testing  
- Redis GUI client

### **2. Initial Setup**

# Clone repository  
git clone https://github.com/BaoBao112233/MAS-Planning.git  
cd MAS-Planning  
  
# Create virtual environment  
python -m venv .venv  
  
# Activate virtual environment  
# Linux/Mac:  
source .venv/bin/activate  
# Windows:  
.venv\Scripts\activate  
  
# Install dependencies  
pip install -r requirements-dev.txt  
  
# Copy environment template  
cp .env.template .env  
cp service-account.json.example service-account.json  
  
# Configure environment variables  
# Edit .env file với your settings

### **3. Environment Configuration**

# .env file configuration  
APP\_NAME="MAS Planning System"  
APP\_DESC="Multi-Agent Smart Home Planning"  
API\_VERSION="1.0.0"  
APP\_PORT=9000  
  
# Google Cloud Configuration  
GOOGLE\_CLOUD\_PROJECT="your-project-id"  
GOOGLE\_CLOUD\_LOCATION="us-central1"  
MODEL\_NAME="gemini-2.5-pro"  
GOOGLE\_APPLICATION\_CREDENTIALS="service-account.json"  
  
# Redis Configuration  
REDIS\_HOST="localhost"  
REDIS\_PORT=6379  
REDIS\_DB=0  
TTL\_SECONDS=3600  
  
# MCP Server Configuration  
MCP\_SERVER\_URL="http://localhost:9031"  
  
# External API Configuration  
PLAN\_API\_BASE\_URL="http://localhost:8080"  
PLAN\_API\_KEY="your-api-key"  
OXII\_ROOT\_API\_URL="https://api.oxii.com"  
  
# Debug Settings  
DEBUG\_MODE=true  
MAX\_TURNS=20  
LIMIT\_MINUTES=10  
MAX\_MSG=12

### **4. Google Cloud Setup**

# Install Google Cloud CLI  
curl https://sdk.cloud.google.com | bash  
exec -l $SHELL  
  
# Initialize gcloud  
gcloud init  
  
# Create project (if needed)  
gcloud projects create your-project-id  
  
# Enable required APIs  
gcloud services enable aiplatform.googleapis.com  
gcloud services enable cloudbuild.googleapis.com  
  
# Create service account  
gcloud iam service-accounts create mas-planning-sa \  
 --description="Service account for MAS Planning" \  
 --display-name="MAS Planning Service Account"  
  
# Grant required permissions  
gcloud projects add-iam-policy-binding your-project-id \  
 --member="serviceAccount:mas-planning-sa@your-project-id.iam.gserviceaccount.com" \  
 --role="roles/aiplatform.user"  
  
# Create và download key  
gcloud iam service-accounts keys create service-account.json \  
 --iam-account=mas-planning-sa@your-project-id.iam.gserviceaccount.com

## 🏗️ Code Structure & Organization

### **1. Project Structure**

MAS-Planning/  
├── main.py # FastAPI application entry point  
├── requirements-dev.txt # Python dependencies  
├── docker-compose.yml # Docker configuration  
├── Dockerfile # Container configuration  
├── .env.template # Environment template  
├── service-account.json # GCP credentials  
├──   
├── template/ # Main application code  
│ ├── agent/ # Agent implementations  
│ │ ├── \_\_init\_\_.py # Base agent class  
│ │ ├── agent.py # Agent utilities  
│ │ ├── histories.py # Chat history management  
│ │ ├── prompts.py # Shared prompts  
│ │ ├── api\_client.py # External API client  
│ │ │  
│ │ ├── manager/ # Manager Agent  
│ │ │ ├── \_\_init\_\_.py # Manager Agent implementation  
│ │ │ ├── prompt.py # Manager-specific prompts  
│ │ │ ├── state.py # State definitions  
│ │ │ └── utils.py # Utility functions  
│ │ │  
│ │ ├── plan/ # Plan Agent  
│ │ │ ├── \_\_init\_\_.py # Plan Agent implementation  
│ │ │ ├── prompts.py # Planning prompts  
│ │ │ ├── state.py # Plan state management  
│ │ │ └── utils.py # Planning utilities  
│ │ │  
│ │ ├── meta/ # Meta Agent  
│ │ │ ├── \_\_init\_\_.py # Meta Agent implementation  
│ │ │ ├── prompt.py # Analysis prompts  
│ │ │ ├── state.py # Meta state management  
│ │ │ └── utils.py # Analysis utilities  
│ │ │  
│ │ └── tool/ # Tool Agent  
│ │ ├── \_\_init\_\_.py # Tool Agent implementation  
│ │ ├── prompt.py # Tool prompts  
│ │ ├── state.py # Tool state management  
│ │ └── utils.py # Tool utilities  
│ │  
│ ├── configs/ # Configuration management  
│ │ └── environments.py # Environment settings  
│ │  
│ ├── message/ # Message handling  
│ │ ├── message.py # Message classes  
│ │ └── converter.py # Message converters  
│ │  
│ ├── router/ # API routing  
│ │ └── v1/  
│ │ └── ai.py # AI endpoints  
│ │  
│ └── schemas/ # Data models  
│ └── model.py # Pydantic models  
│  
├── docs/ # Documentation  
│ ├── 01\_Kiến\_Trúc\_Tổng\_Quan.md  
│ ├── 02\_Manager\_Agent.md  
│ ├── 03\_Plan\_Agent.md  
│ ├── 04\_Meta\_Agent.md  
│ ├── 05\_Tool\_Agent.md  
│ ├── 06\_MCP\_Integration\_External\_Services.md  
│ └── 07\_Development\_Guide.md  
│  
└── tests/ # Test suite  
 ├── unit/ # Unit tests  
 ├── integration/ # Integration tests  
 └── e2e/ # End-to-end tests

### **2. Coding Standards**

# File header template  
"""  
Module: agent/plan/\_\_init\_\_.py  
Description: Plan Agent implementation for MAS-Planning system  
Author: MAS-Planning Team  
Created: 2025-01-XX  
Modified: 2025-01-XX  
"""  
  
# Import organization  
import os  
import sys  
import time  
import asyncio  
from typing import Dict, List, Any, Optional, TypedDict  
  
# Third-party imports  
import logging  
from termcolor import colored  
from langchain\_google\_vertexai import ChatVertexAI  
from langgraph.graph import StateGraph, END, START  
  
# Local imports  
from template.agent import BaseAgent  
from template.configs.environments import env  
from template.message.message import SystemMessage, HumanMessage  
  
# Logging configuration  
logger = logging.getLogger(\_\_name\_\_)  
  
# Class definition với comprehensive docstring  
class PlanAgent(BaseAgent):  
 """  
 Plan Agent - Strategic Planning Engine  
   
 The Plan Agent is responsible for creating intelligent smart home automation plans  
 và orchestrating their execution through other specialized agents.  
   
 Attributes:  
 name (str): Agent identifier  
 model (str): LLM model name  
 temperature (float): LLM temperature setting  
 verbose (bool): Verbose logging flag  
 tools (List): Available MCP tools  
 llm: Language model instance  
 graph: LangGraph execution graph  
   
 Methods:  
 router: Route input to appropriate planning workflow  
 priority\_plan: Generate 3 priority-based plans  
 execute\_selected\_plan: Execute user-selected plan  
 init\_sub\_agents: Initialize Meta và Tool agents  
 """  
   
 def \_\_init\_\_(self,   
 model: str = "gemini-2.5-pro",   
 temperature: float = 0.2,   
 max\_iteration: int = 10,  
 verbose: bool = True):  
 """  
 Initialize Plan Agent  
   
 Args:  
 model: LLM model name  
 temperature: LLM temperature (0.0-1.0)  
 max\_iteration: Maximum planning iterations  
 verbose: Enable detailed logging  
 """  
 super().\_\_init\_\_()  
   
 self.name = "Plan Agent"  
 self.model = model  
 self.temperature = temperature  
 self.verbose = verbose  
   
 # Initialize components  
 self.tools = []  
 self.llm = None  
 self.meta\_agent = None  
 self.tool\_agent = None  
   
 # Create execution graph  
 self.graph = self.create\_graph()  
   
 if self.verbose:  
 logger.info(f"✅ {self.name} initialized successfully")

### **3. Error Handling Patterns**

# Comprehensive error handling  
class PlanningError(Exception):  
 """Base exception for planning operations"""  
 pass  
  
class MCPConnectionError(PlanningError):  
 """Raised when MCP server connection fails"""  
 pass  
  
class AuthenticationError(PlanningError):  
 """Raised when authentication fails"""  
 pass  
  
def robust\_planning\_operation(func):  
 """Decorator for robust planning operations"""  
 def wrapper(\*args, \*\*kwargs):  
 try:  
 return func(\*args, \*\*kwargs)  
 except MCPConnectionError as e:  
 logger.error(f"MCP connection failed: {e}")  
 # Fallback to cached tools  
 return fallback\_planning\_operation(\*args, \*\*kwargs)  
 except AuthenticationError as e:  
 logger.error(f"Authentication failed: {e}")  
 raise # Re-raise auth errors  
 except Exception as e:  
 logger.error(f"Unexpected error trong {func.\_\_name\_\_}: {e}")  
 # Return safe fallback response  
 return safe\_fallback\_response()  
 return wrapper  
  
@robust\_planning\_operation  
def generate\_priority\_plans(self, input\_text: str) -> Dict:  
 """Generate plans với comprehensive error handling"""  
 # Implementation với try-catch blocks  
 pass

## 🧪 Testing Strategy

### **1. Testing Framework Setup**

# tests/conftest.py  
import pytest  
import asyncio  
import os  
from unittest.mock import Mock, AsyncMock  
from template.configs.environments import env  
  
@pytest.fixture  
def mock\_llm():  
 """Mock LLM for testing"""  
 mock = Mock()  
 mock.invoke = Mock(return\_value=Mock(content="Test response"))  
 return mock  
  
@pytest.fixture  
def mock\_mcp\_tools():  
 """Mock MCP tools"""  
 tools = [  
 Mock(name="get\_device\_list", description="Get device list"),  
 Mock(name="switch\_device\_control", description="Control switches"),  
 Mock(name="control\_air\_conditioner", description="Control AC")  
 ]  
 return tools  
  
@pytest.fixture  
async def plan\_agent(mock\_llm, mock\_mcp\_tools):  
 """Plan Agent fixture for testing"""  
 from template.agent.plan import PlanAgent  
   
 agent = PlanAgent(verbose=False)  
 agent.llm = mock\_llm  
 agent.tools = mock\_mcp\_tools  
   
 return agent  
  
@pytest.fixture  
def sample\_plan\_data():  
 """Sample plan data for testing"""  
 return {  
 'security\_plan': [  
 'Install motion sensors',  
 'Set up security cameras',  
 'Configure alarm system'  
 ],  
 'convenience\_plan': [  
 'Automate lighting',  
 'Set up voice control',  
 'Create routines'  
 ],  
 'energy\_plan': [  
 'Install smart thermostat',  
 'Use LED bulbs',  
 'Monitor energy usage'  
 ]  
 }

### **2. Unit Tests**

# tests/unit/test\_plan\_agent.py  
import pytest  
from unittest.mock import Mock, patch  
from template.agent.plan import PlanAgent  
  
class TestPlanAgent:  
 """Test suite for Plan Agent"""  
   
 def test\_plan\_agent\_initialization(self):  
 """Test Plan Agent initialization"""  
 agent = PlanAgent(verbose=False)  
   
 assert agent.name == "Plan Agent"  
 assert agent.model == "gemini-2.5-pro"  
 assert agent.temperature == 0.2  
 assert agent.tools == []  
 assert agent.llm is None  
   
 def test\_router\_logic(self, plan\_agent, sample\_plan\_data):  
 """Test routing logic"""  
 # Test new planning request  
 state = {'input': 'Create a smart home plan'}  
 result = plan\_agent.router(state)  
 assert result['plan\_type'] == 'priority'  
   
 # Test plan selection  
 state = {'input': '2', 'plan\_options': sample\_plan\_data}  
 result = plan\_agent.router(state)  
 assert result['plan\_type'] == 'execute'  
 assert result['selected\_plan\_id'] == 2  
   
 @patch('template.agent.plan.extract\_priority\_plans')  
 def test\_priority\_plan\_generation(self, mock\_extract, plan\_agent, sample\_plan\_data):  
 """Test priority plan generation"""  
 # Mock extraction function  
 mock\_extract.return\_value = {  
 'Security\_Plan': sample\_plan\_data['security\_plan'],  
 'Convenience\_Plan': sample\_plan\_data['convenience\_plan'],  
 'Energy\_Plan': sample\_plan\_data['energy\_plan']  
 }  
   
 state = {'input': 'Create automation plan for living room'}  
 result = plan\_agent.priority\_plan(state)  
   
 assert 'plan\_options' in result  
 assert result['needs\_user\_selection'] is True  
 assert len(result['plan\_options']) == 3  
   
 def test\_error\_handling(self, plan\_agent):  
 """Test error handling trong plan generation"""  
 # Mock LLM to raise exception  
 plan\_agent.llm.invoke.side\_effect = Exception("LLM error")  
   
 state = {'input': 'Create plan'}  
 result = plan\_agent.priority\_plan(state)  
   
 # Should return fallback data  
 assert 'plan\_options' in result  
 assert all(key in result['plan\_options'] for key in ['security\_plan', 'convenience\_plan', 'energy\_plan'])

### **3. Integration Tests**

# tests/integration/test\_agent\_integration.py  
import pytest  
import asyncio  
from unittest.mock import Mock, AsyncMock  
  
class TestAgentIntegration:  
 """Test integration between agents"""  
   
 @pytest.mark.asyncio  
 async def test\_plan\_to\_meta\_integration(self):  
 """Test Plan Agent to Meta Agent integration"""  
 from template.agent.plan import PlanAgent  
 from template.agent.meta import MetaAgent  
   
 # Setup  
 plan\_agent = PlanAgent(verbose=False)  
 plan\_agent.llm = Mock()  
 plan\_agent.meta\_agent = MetaAgent(verbose=False)  
 plan\_agent.meta\_agent.llm = Mock()  
   
 # Mock Meta Agent response  
 plan\_agent.meta\_agent.invoke = Mock(return\_value={  
 'output': 'Task analysis complete',  
 'agent\_data': {'Agent Name': 'Tool Agent'},  
 'success': True  
 })  
   
 # Test integration  
 task = "Turn on living room lights"  
 meta\_input = {  
 "input": task,  
 "context": "Test context",  
 "previous\_results": []  
 }  
   
 result = plan\_agent.meta\_agent.invoke(meta\_input)  
   
 assert result['success'] is True  
 assert 'output' in result  
   
 @pytest.mark.asyncio  
 async def test\_end\_to\_end\_workflow(self):  
 """Test end-to-end workflow"""  
 from template.agent.manager import ManagerAgent  
   
 # Setup Manager Agent  
 manager = ManagerAgent(  
 session\_id="test\_session",  
 conversation\_id="test\_conv",  
 verbose=False  
 )  
   
 # Mock external dependencies  
 manager.llm = Mock()  
 manager.llm.invoke = Mock(return\_value=Mock(content="Mock LLM response"))  
   
 # Test input  
 input\_data = {  
 "message": "Create a security plan for my home",  
 "token": "test\_token"  
 }  
   
 # Execute workflow  
 result = manager.invoke(input\_data)  
   
 assert 'output' in result  
 assert result['success'] is True

### **4. End-to-End Tests**

# tests/e2e/test\_api\_endpoints.py  
import pytest  
import httpx  
from fastapi.testclient import TestClient  
from main import app  
  
class TestAPIEndpoints:  
 """End-to-end API tests"""  
   
 def setup\_method(self):  
 """Setup test client"""  
 self.client = TestClient(app)  
   
 def test\_health\_endpoint(self):  
 """Test health check endpoint"""  
 response = self.client.get("/health")  
   
 assert response.status\_code == 200  
 assert response.json() == {"status": "healthy"}  
   
 def test\_chat\_endpoint\_plan\_creation(self):  
 """Test chat endpoint for plan creation"""  
 payload = {  
 "conversationId": "test\_conv",  
 "sessionId": "test\_session",  
 "message": "Create a smart home plan for my bedroom",  
 "channelId": "test\_channel",  
 "socialNetworkId": "test\_network",  
 "pageName": "test\_page",  
 "token": "test\_token"  
 }  
   
 response = self.client.post("/ai/chat", json=payload)  
   
 assert response.status\_code == 200  
 data = response.json()  
 assert "sessionId" in data  
 assert "response" in data  
 assert data["error\_status"] == "success"  
   
 def test\_chat\_endpoint\_plan\_selection(self):  
 """Test plan selection workflow"""  
 # First, create plans  
 create\_payload = {  
 "conversationId": "test\_conv\_2",  
 "sessionId": "test\_session\_2",  
 "message": "Create automation plan",  
 "channelId": "test\_channel",  
 "socialNetworkId": "test\_network",  
 "pageName": "test\_page"  
 }  
   
 create\_response = self.client.post("/ai/chat", json=create\_payload)  
 assert create\_response.status\_code == 200  
   
 # Then, select a plan  
 select\_payload = {  
 "conversationId": "test\_conv\_2",  
 "sessionId": "test\_session\_2",  
 "message": "2", # Select plan 2  
 "channelId": "test\_channel",  
 "socialNetworkId": "test\_network",  
 "pageName": "test\_page",  
 "token": "test\_token"  
 }  
   
 select\_response = self.client.post("/ai/chat", json=select\_payload)  
 assert select\_response.status\_code == 200  
   
 def test\_token\_endpoint(self):  
 """Test token generation endpoint"""  
 response = self.client.get(  
 "/token",  
 params={  
 "user\_phone": "1234567890",  
 "user\_password": "test\_password",  
 "user\_country": "VI"  
 }  
 )  
   
 # Note: This will fail trong test environment without real OXII API  
 # But we can test the endpoint structure  
 assert response.status\_code in [200, 500] # Allow both success và expected failure

## 🚀 Deployment Guide

### **1. Local Development**

# Start development server  
python main.py  
  
# With auto-reload  
uvicorn main:app --reload --host 0.0.0.0 --port 9000  
  
# With verbose logging  
DEBUG\_MODE=true python main.py

### **2. Docker Deployment**

# Dockerfile  
FROM python:3.11-slim  
  
WORKDIR /app  
  
# Install system dependencies  
RUN apt-get update && apt-get install -y \  
 gcc \  
 && rm -rf /var/lib/apt/lists/\*  
  
# Copy requirements first for better caching  
COPY requirements-dev.txt .  
RUN pip install --no-cache-dir -r requirements-dev.txt  
  
# Copy application code  
COPY . .  
  
# Create non-root user  
RUN useradd --create-home --shell /bin/bash mas-user  
RUN chown -R mas-user:mas-user /app  
USER mas-user  
  
# Expose port  
EXPOSE 9000  
  
# Health check  
HEALTHCHECK --interval=30s --timeout=10s --start-period=5s --retries=3 \  
 CMD curl -f http://localhost:9000/health || exit 1  
  
# Start application  
CMD ["python", "main.py"]

# docker-compose.yml  
version: '3.8'  
  
services:  
 mas-planning:  
 build: .  
 ports:  
 - "9000:9000"  
 environment:  
 - REDIS\_HOST=redis  
 - MCP\_SERVER\_URL=http://mcp-server:9031  
 volumes:  
 - ./service-account.json:/app/service-account.json:ro  
 - ./logs:/app/logs  
 depends\_on:  
 - redis  
 - mcp-server  
 restart: unless-stopped  
 healthcheck:  
 test: ["CMD", "curl", "-f", "http://localhost:9000/health"]  
 interval: 30s  
 timeout: 10s  
 retries: 3  
  
 redis:  
 image: redis:7-alpine  
 ports:  
 - "6379:6379"  
 volumes:  
 - redis\_data:/data  
 restart: unless-stopped  
 healthcheck:  
 test: ["CMD", "redis-cli", "ping"]  
 interval: 10s  
 timeout: 5s  
 retries: 3  
  
 mcp-server:  
 image: your-mcp-server:latest  
 ports:  
 - "9031:9031"  
 environment:  
 - NODE\_ENV=production  
 restart: unless-stopped  
 healthcheck:  
 test: ["CMD", "curl", "-f", "http://localhost:9031/health"]  
 interval: 30s  
 timeout: 10s  
 retries: 3  
  
volumes:  
 redis\_data:  
  
networks:  
 default:  
 driver: bridge

### **3. Production Deployment**

# Production docker-compose  
version: '3.8'  
  
services:  
 mas-planning:  
 image: mas-planning:latest  
 ports:  
 - "9000:9000"  
 environment:  
 - ENV=production  
 - REDIS\_HOST=redis  
 - MCP\_SERVER\_URL=http://mcp-server:9031  
 volumes:  
 - ./service-account.json:/app/service-account.json:ro  
 - ./logs:/app/logs  
 - ./data:/app/data  
 deploy:  
 replicas: 3  
 restart\_policy:  
 condition: on-failure  
 delay: 5s  
 max\_attempts: 3  
 resources:  
 limits:  
 memory: 2G  
 cpus: '1'  
 reservations:  
 memory: 1G  
 cpus: '0.5'  
 depends\_on:  
 - redis  
 - mcp-server  
 networks:  
 - mas-network  
  
 nginx:  
 image: nginx:alpine  
 ports:  
 - "80:80"  
 - "443:443"  
 volumes:  
 - ./nginx.conf:/etc/nginx/nginx.conf:ro  
 - ./ssl:/etc/ssl:ro  
 depends\_on:  
 - mas-planning  
 networks:  
 - mas-network  
  
 redis:  
 image: redis:7-alpine  
 volumes:  
 - redis\_data:/data  
 - ./redis.conf:/etc/redis/redis.conf:ro  
 command: redis-server /etc/redis/redis.conf  
 deploy:  
 replicas: 1  
 restart\_policy:  
 condition: on-failure  
 networks:  
 - mas-network  
  
networks:  
 mas-network:  
 driver: overlay  
 attachable: true  
  
volumes:  
 redis\_data:  
 driver: local

### **4. Kubernetes Deployment**

# k8s/deployment.yaml  
apiVersion: apps/v1  
kind: Deployment  
metadata:  
 name: mas-planning  
 namespace: default  
spec:  
 replicas: 3  
 selector:  
 matchLabels:  
 app: mas-planning  
 template:  
 metadata:  
 labels:  
 app: mas-planning  
 spec:  
 containers:  
 - name: mas-planning  
 image: mas-planning:latest  
 ports:  
 - containerPort: 9000  
 env:  
 - name: REDIS\_HOST  
 value: "redis-service"  
 - name: MCP\_SERVER\_URL  
 value: "http://mcp-server-service:9031"  
 volumeMounts:  
 - name: service-account  
 mountPath: /app/service-account.json  
 subPath: service-account.json  
 readOnly: true  
 - name: logs  
 mountPath: /app/logs  
 livenessProbe:  
 httpGet:  
 path: /health  
 port: 9000  
 initialDelaySeconds: 30  
 periodSeconds: 30  
 readinessProbe:  
 httpGet:  
 path: /health  
 port: 9000  
 initialDelaySeconds: 5  
 periodSeconds: 10  
 resources:  
 requests:  
 memory: "1Gi"  
 cpu: "500m"  
 limits:  
 memory: "2Gi"  
 cpu: "1"  
 volumes:  
 - name: service-account  
 secret:  
 secretName: gcp-service-account  
 - name: logs  
 emptyDir: {}  
---  
apiVersion: v1  
kind: Service  
metadata:  
 name: mas-planning-service  
spec:  
 selector:  
 app: mas-planning  
 ports:  
 - protocol: TCP  
 port: 80  
 targetPort: 9000  
 type: LoadBalancer

## 🔍 Debugging & Troubleshooting

### **1. Logging Configuration**

# Enhanced logging setup  
import logging  
import sys  
from logging.handlers import RotatingFileHandler  
  
def setup\_logging(level=logging.INFO, log\_file='mas-planning.log'):  
 """Setup comprehensive logging"""  
   
 # Create formatter  
 formatter = logging.Formatter(  
 '%(asctime)s - %(name)s - %(levelname)s - %(filename)s:%(lineno)d - %(message)s',  
 datefmt='%Y-%m-%d %H:%M:%S'  
 )  
   
 # Console handler  
 console\_handler = logging.StreamHandler(sys.stdout)  
 console\_handler.setFormatter(formatter)  
 console\_handler.setLevel(level)  
   
 # File handler với rotation  
 file\_handler = RotatingFileHandler(  
 log\_file,   
 maxBytes=10\*1024\*1024, # 10MB  
 backupCount=5  
 )  
 file\_handler.setFormatter(formatter)  
 file\_handler.setLevel(level)  
   
 # Root logger  
 root\_logger = logging.getLogger()  
 root\_logger.setLevel(level)  
 root\_logger.addHandler(console\_handler)  
 root\_logger.addHandler(file\_handler)  
   
 # Specific loggers  
 loggers = [  
 'template.agent.manager',  
 'template.agent.plan',  
 'template.agent.meta',  
 'template.agent.tool',  
 'template.router.v1.ai'  
 ]  
   
 for logger\_name in loggers:  
 logger = logging.getLogger(logger\_name)  
 logger.setLevel(level)  
  
# Usage  
if env.DEBUG\_MODE:  
 setup\_logging(logging.DEBUG, 'debug.log')  
else:  
 setup\_logging(logging.INFO, 'mas-planning.log')

### **2. Debug Tools**

# Debug utilities  
class DebugManager:  
 """Debug utilities for development"""  
   
 def \_\_init\_\_(self):  
 self.debug\_enabled = env.DEBUG\_MODE  
 self.trace\_calls = False  
 self.performance\_tracking = True  
   
 def trace\_function\_calls(self, func):  
 """Decorator to trace function calls"""  
 if not self.debug\_enabled:  
 return func  
   
 def wrapper(\*args, \*\*kwargs):  
 logger.debug(f"TRACE: Calling {func.\_\_name\_\_} với args={args}, kwargs={kwargs}")  
 start\_time = time.time()  
   
 try:  
 result = func(\*args, \*\*kwargs)  
 duration = time.time() - start\_time  
 logger.debug(f"TRACE: {func.\_\_name\_\_} completed trong {duration:.3f}s")  
 return result  
 except Exception as e:  
 duration = time.time() - start\_time  
 logger.debug(f"TRACE: {func.\_\_name\_\_} failed trong {duration:.3f}s với error: {e}")  
 raise  
   
 return wrapper  
   
 def dump\_state(self, state: Dict, context: str = ""):  
 """Dump state for debugging"""  
 if not self.debug\_enabled:  
 return  
   
 logger.debug(f"STATE DUMP ({context}):")  
 for key, value in state.items():  
 if isinstance(value, (str, int, float, bool)):  
 logger.debug(f" {key}: {value}")  
 elif isinstance(value, (list, dict)):  
 logger.debug(f" {key}: {type(value).\_\_name\_\_} length={len(value)}")  
 else:  
 logger.debug(f" {key}: {type(value).\_\_name\_\_}")  
   
 def performance\_monitor(self, operation\_name: str):  
 """Context manager for performance monitoring"""  
 class PerformanceMonitor:  
 def \_\_init\_\_(self, name):  
 self.name = name  
 self.start\_time = None  
   
 def \_\_enter\_\_(self):  
 self.start\_time = time.time()  
 logger.debug(f"PERF: Starting {self.name}")  
 return self  
   
 def \_\_exit\_\_(self, exc\_type, exc\_val, exc\_tb):  
 duration = time.time() - self.start\_time  
 if exc\_type:  
 logger.debug(f"PERF: {self.name} failed trong {duration:.3f}s")  
 else:  
 logger.debug(f"PERF: {self.name} completed trong {duration:.3f}s")  
   
 return PerformanceMonitor(operation\_name)  
  
# Usage  
debug\_manager = DebugManager()  
  
@debug\_manager.trace\_function\_calls  
def some\_function(param1, param2):  
 with debug\_manager.performance\_monitor("some\_operation"):  
 # Your code here  
 debug\_manager.dump\_state({"param1": param1, "param2": param2}, "function\_start")  
 return result

### **3. Common Issues & Solutions**

# Common troubleshooting scenarios  
class TroubleshootingGuide:  
 """Common issues và their solutions"""  
   
 @staticmethod  
 def diagnose\_mcp\_connection():  
 """Diagnose MCP connection issues"""  
 try:  
 import asyncio  
 import aiohttp  
   
 async def test\_mcp():  
 timeout = aiohttp.ClientTimeout(total=5)  
 async with aiohttp.ClientSession(timeout=timeout) as session:  
 async with session.get(f"{env.MCP\_SERVER\_URL}/health") as response:  
 if response.status == 200:  
 logger.info("✅ MCP server is reachable")  
 return True  
 else:  
 logger.error(f"❌ MCP server returned status {response.status}")  
 return False  
   
 return asyncio.run(test\_mcp())  
   
 except Exception as e:  
 logger.error(f"❌ MCP connection test failed: {e}")  
 return False  
   
 @staticmethod  
 def diagnose\_vertex\_ai():  
 """Diagnose Vertex AI connection"""  
 try:  
 from langchain\_google\_vertexai import ChatVertexAI  
   
 # Check credentials file  
 if not os.path.exists(env.GOOGLE\_APPLICATION\_CREDENTIALS):  
 logger.error(f"❌ Service account file not found: {env.GOOGLE\_APPLICATION\_CREDENTIALS}")  
 return False  
   
 # Test LLM initialization  
 llm = ChatVertexAI(  
 model\_name="gemini-2.5-pro",  
 project=env.GOOGLE\_CLOUD\_PROJECT,  
 location=env.GOOGLE\_CLOUD\_LOCATION  
 )  
   
 # Test simple call  
 response = llm.invoke("Hello")  
 logger.info("✅ Vertex AI connection successful")  
 return True  
   
 except Exception as e:  
 logger.error(f"❌ Vertex AI connection failed: {e}")  
 return False  
   
 @staticmethod  
 def diagnose\_redis():  
 """Diagnose Redis connection"""  
 try:  
 import redis  
   
 r = redis.Redis(  
 host=env.REDIS\_HOST,  
 port=env.REDIS\_PORT,  
 db=env.REDIS\_DB,  
 socket\_timeout=5  
 )  
   
 # Test connection  
 r.ping()  
 logger.info("✅ Redis connection successful")  
 return True  
   
 except Exception as e:  
 logger.error(f"❌ Redis connection failed: {e}")  
 return False  
   
 @staticmethod  
 def run\_full\_diagnosis():  
 """Run complete system diagnosis"""  
 logger.info("🔍 Running system diagnosis...")  
   
 results = {  
 'mcp\_server': TroubleshootingGuide.diagnose\_mcp\_connection(),  
 'vertex\_ai': TroubleshootingGuide.diagnose\_vertex\_ai(),  
 'redis': TroubleshootingGuide.diagnose\_redis()  
 }  
   
 all\_healthy = all(results.values())  
   
 if all\_healthy:  
 logger.info("✅ All systems healthy")  
 else:  
 failed\_systems = [k for k, v in results.items() if not v]  
 logger.error(f"❌ Failed systems: {', '.join(failed\_systems)}")  
   
 return results  
  
# CLI tool for diagnosis  
if \_\_name\_\_ == "\_\_main\_\_":  
 import sys  
   
 if len(sys.argv) > 1 and sys.argv[1] == "diagnose":  
 TroubleshootingGuide.run\_full\_diagnosis()

*Development Guide này cung cấp comprehensive framework để develop, test, và deploy MAS-Planning system effectively. Follow các best practices này để ensure code quality, maintainability, và reliable deployment.*