Template Repository Guide

This guide provides comprehensive information about creating and using template repositories for both AI Module and Task Prompt workflows in the Enhanced Two-Tiered Multi-Agent Orchestration System.

Overview

The orchestration system supports two types of repositories:

1. **AI Module Repositories**: Self-contained with `ai-module.yaml` configuration
2. **Task Prompt Repositories**: Traditional repositories that rely on natural language prompts

AI Module Templates

AI Module repositories include a structured `ai-module.yaml` file that defines the complete deployment configuration.

Template Structure

```
ai-module-repository/
├── ai-module.yaml          # Main configuration file
├── Dockerfile              # Container definition
├── requirements.txt        # Dependencies (Python)
├── package.json            # Dependencies (Node.js)
├── src/                    # Source code
├── tests/                  # Test files
├── docs/                   # Documentation
└── README.md               # Project documentation
```

AI Module Configuration Schema

```yaml
# Basic Information
name: "application-name"
version: "1.0.0"
description: "Application description"
module_type: "web_app|api|ml_model|data_pipeline|microservice|automation|infrastruc-
ture"

# Build Configuration
build_command: "npm install && npm run build"
test_command: "npm test"
start_command: "npm start"

# Dependencies
dependencies:
  - "package@version"
dev_dependencies:
  - "dev-package@version"
system_dependencies:
  - "system-package"

# Environment
environment_variables:
  ENV_VAR: "value"
secrets:
  - "SECRET_NAME"

# Deployment
deployment_target: "docker|kubernetes|cloud_run|lambda|vercel|local"
port: 8000

# Resources
resources:
  cpu: "500m"
  memory: "1Gi"
  storage: "2Gi"
  gpu: false

# Health Check
health_check:
  enabled: true
  endpoint: "/health"
  interval: 30
  timeout: 10
  retries: 3

# Security
security:
  enable_auth: true
  auth_type: "jwt"
  rate_limiting: true
  input_validation: true
  cors_enabled: true
  allowed_origins:
    - "https://example.com"

# Monitoring
monitoring:
  metrics_enabled: true
  logging_level: "INFO"
  tracing_enabled: false
  alerts_enabled: true
```

```yaml
# Advanced Features
auto_scaling: false
backup_enabled: true
rollback_enabled: true

# Custom Configuration
custom_config:
  key: "value"
```

Template Examples

1. Web Application Template

React Web App

File: `ai-module.yaml`

```yaml
name: "crypto-dashboard"
version: "1.0.0"
description: "Cryptocurrency dashboard web application"
module_type: "web_app"

build_command: "npm install && npm run build"
test_command: "npm test"
start_command: "npm start"

dependencies:
  - "react@^18.0.0"
  - "typescript@^4.9.0"
  - "axios@^1.0.0"
  - "chart.js@^4.0.0"
  - "react-router-dom@^6.0.0"

dev_dependencies:
  - "jest@^29.0.0"
  - "eslint@^8.0.0"
  - "@testing-library/react@^13.0.0"

system_dependencies:
  - "nodejs"
  - "npm"

environment_variables:
  NODE_ENV: "production"
  PORT: "3000"
  REACT_APP_API_URL: "https://api.xplaincrypto.ai"

secrets:
  - "REACT_APP_API_KEY"

deployment_target: "docker"
port: 3000

resources:
  cpu: "200m"
  memory: "512Mi"
  storage: "1Gi"
  gpu: false

health_check:
  enabled: true
  endpoint: "/health"
  interval: 30
  timeout: 10
  retries: 3

security:
  enable_auth: false
  rate_limiting: true
  input_validation: true
  cors_enabled: true
  allowed_origins:
    - "https://xplaincrypto.ai"

monitoring:
  metrics_enabled: true
  logging_level: "INFO"
  tracing_enabled: false
  alerts_enabled: true
```

```yaml
auto_scaling: false
backup_enabled: true
rollback_enabled: true

custom_config:
  theme: "dark"
  refresh_interval: 30
  max_chart_points: 100
```

File: `package.json`

```json
{
  "name": "crypto-dashboard",
  "version": "1.0.0",
  "description": "Cryptocurrency dashboard web application",
  "main": "src/index.js",
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test",
    "eject": "react-scripts eject"
  },
  "dependencies": {
    "react": "^18.0.0",
    "react-dom": "^18.0.0",
    "react-scripts": "5.0.1",
    "typescript": "^4.9.0",
    "axios": "^1.0.0",
    "chart.js": "^4.0.0",
    "react-chartjs-2": "^5.0.0",
    "react-router-dom": "^6.0.0"
  },
  "devDependencies": {
    "jest": "^29.0.0",
    "eslint": "^8.0.0",
    "@testing-library/react": "^13.0.0"
  }
}
```

File: `Dockerfile`

```
FROM node:18-alpine

WORKDIR /app

# Copy package files
COPY package*.json ./

# Install dependencies
RUN npm ci --only=production

# Copy source code
COPY . .

# Build the application
RUN npm run build

# Expose port
EXPOSE 3000

# Health check
HEALTHCHECK --interval=30s --timeout=10s --start-period=5s --retries=3 \
  CMD curl -f http://localhost:3000/health || exit 1

# Start the application
CMD ["npm", "start"]
```

2. API Service Template

FastAPI Service

File: `ai-module.yaml`

```yaml
name: "crypto-api"
version: "1.0.0"
description: "Cryptocurrency data API service"
module_type: "api"

build_command: "pip install -r requirements.txt"
test_command: "pytest tests/ -v"
start_command: "uvicorn main:app --host 0.0.0.0 --port 8000"

dependencies:
  - "fastapi==0.104.0"
  - "uvicorn==0.24.0"
  - "pydantic==2.5.0"
  - "httpx==0.25.0"
  - "redis==5.0.0"
  - "sqlalchemy==2.0.0"
  - "psycopg2-binary==2.9.0"

dev_dependencies:
  - "pytest==7.4.0"
  - "pytest-asyncio==0.21.0"
  - "black==23.11.0"
  - "flake8==6.1.0"

system_dependencies:
  - "python3"
  - "pip"
  - "postgresql-client"

environment_variables:
  ENVIRONMENT: "production"
  LOG_LEVEL: "INFO"
  DATABASE_URL: "postgresql://user:pass@localhost/crypto_db"
  REDIS_URL: "redis://localhost:6379"

secrets:
  - "DATABASE_PASSWORD"
  - "REDIS_PASSWORD"
  - "API_SECRET_KEY"
  - "COINMARKETCAP_API_KEY"

deployment_target: "docker"
port: 8000

resources:
  cpu: "500m"
  memory: "1Gi"
  storage: "2Gi"
  gpu: false

health_check:
  enabled: true
  endpoint: "/health"
  interval: 30
  timeout: 10
  retries: 3

security:
  enable_auth: true
  auth_type: "jwt"
  rate_limiting: true
  input_validation: true
```

```yaml
  cors_enabled: true
  allowed_origins:
    - "https://xplaincrypto.ai"

monitoring:
  metrics_enabled: true
  logging_level: "INFO"
  tracing_enabled: true
  alerts_enabled: true

auto_scaling: true
backup_enabled: true
rollback_enabled: true

custom_config:
  cache_ttl: 300
  max_requests_per_minute: 1000
  database_pool_size: 10
```

File: `requirements.txt`

```
fastapi==0.104.0
uvicorn==0.24.0
pydantic==2.5.0
httpx==0.25.0
redis==5.0.0
sqlalchemy==2.0.0
psycopg2-binary==2.9.0
python-jose[cryptography]==3.3.0
passlib[bcrypt]==1.7.4
python-multipart==0.0.6
prometheus-client==0.19.0

# Development dependencies
pytest==7.4.0
pytest-asyncio==0.21.0
black==23.11.0
flake8==6.1.0
mypy==1.7.0
```

File: `main.py`

```python
from fastapi import FastAPI, HTTPException, Depends
from fastapi.middleware.cors import CORSMiddleware
from pydantic import BaseModel
import httpx
import redis
import os
from datetime import datetime

app = FastAPI(
    title="Crypto API",
    description="Cryptocurrency data API service",
    version="1.0.0"
)

# CORS middleware
app.add_middleware(
    CORSMiddleware,
    allow_origins=["https://xplaincrypto.ai"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

# Redis connection
redis_client = redis.from_url(os.getenv("REDIS_URL", "redis://localhost:6379"))

class CryptoPrice(BaseModel):
    symbol: str
    price: float
    change_24h: float
    timestamp: datetime

@app.get("/health")
async def health_check():
    """Health check endpoint"""
    return {
        "status": "healthy",
        "timestamp": datetime.now(),
        "service": "crypto-api"
    }

@app.get("/api/v1/price/{symbol}", response_model=CryptoPrice)
async def get_crypto_price(symbol: str):
    """Get cryptocurrency price"""
    try:
        # Check cache first
        cached_price = redis_client.get(f"price:{symbol}")
        if cached_price:
            return CryptoPrice.parse_raw(cached_price)

        # Fetch from external API
        async with httpx.AsyncClient() as client:
            response = await client.get(
                f"https://api.coinmarketcap.com/v1/ticker/{symbol}/"
            )
            data = response.json()

        price_data = CryptoPrice(
            symbol=symbol,
            price=float(data[0]["price_usd"]),
            change_24h=float(data[0]["percent_change_24h"]),
            timestamp=datetime.now()
```

```
        )

        # Cache for 5 minutes
        redis_client.setex(
            f"price:{symbol}",
            300,
            price_data.json()
        )

        return price_data

    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))

if __name__ == "__main__":
    import uvicorn
    uvicorn.run(app, host="0.0.0.0", port=8000)
```

3. Machine Learning Model Template

File: `ai-module.yaml`

```yaml
name: "crypto-price-predictor"
version: "1.0.0"
description: "Machine learning model for cryptocurrency price prediction"
module_type: "ml_model"

build_command: "pip install -r requirements.txt && python train_model.py"
test_command: "pytest tests/ -v"
start_command: "python serve.py"

dependencies:
  - "scikit-learn==1.3.0"
  - "pandas==2.0.0"
  - "numpy==1.24.0"
  - "fastapi==0.104.0"
  - "uvicorn==0.24.0"
  - "joblib==1.3.0"

dev_dependencies:
  - "pytest==7.4.0"
  - "jupyter==1.0.0"
  - "matplotlib==3.7.0"

system_dependencies:
  - "python3"
  - "pip"

environment_variables:
  MODEL_PATH: "/app/models"
  PREDICTION_INTERVAL: "3600"

secrets:
  - "DATA_API_KEY"

deployment_target: "docker"
port: 8000

resources:
  cpu: "1000m"
  memory: "2Gi"
  storage: "5Gi"
  gpu: false

health_check:
  enabled: true
  endpoint: "/health"
  interval: 60
  timeout: 30
  retries: 3

security:
  enable_auth: true
  auth_type: "jwt"
  rate_limiting: true
  input_validation: true

monitoring:
  metrics_enabled: true
  logging_level: "INFO"
  tracing_enabled: true
  alerts_enabled: true

auto_scaling: false
```

```
backup_enabled: true
rollback_enabled: true

custom_config:
  model_retrain_interval: 86400  # 24 hours
  prediction_confidence_threshold: 0.8
  max_prediction_horizon: 168  # 7 days
```

Task Prompt Templates

Task Prompt repositories don't require `ai-module.yaml` files. Instead, they rely on repository structure analysis and natural language prompts to determine deployment strategy.

Task Prompt Repository Structure

```
task-prompt-repository/
├── README.md              # Project description and task prompt
├── requirements.txt       # Python dependencies
├── package.json           # Node.js dependencies
├── Dockerfile             # Optional container definition
├── src/                   # Source code
├── tests/                 # Test files
└── docs/                  # Documentation
```

Task Prompt Examples

1. Simple Web Server

Repository Structure:

```
simple-web-server/
├── app.py
├── requirements.txt
├── templates/
│   └── dashboard.html
└── README.md
```

Task Prompt (in README.md):

```
# Simple Web Server

**Task Prompt:** "Create a simple web server that serves static files and provides a
basic API for cryptocurrency prices. The server should be built with Python Flask and
include basic logging and error handling."

## Expected Behavior

The orchestrator should:
1. Detect this as a Python Flask application
2. Install dependencies from requirements.txt
3. Start the Flask server on port 5000
4. Setup basic health monitoring
```

File: `requirements.txt`

```
Flask==2.3.3
requests==2.31.0
python-dotenv==1.0.0
gunicorn==21.2.0
```

File: `app.py`

```python
from flask import Flask, render_template, jsonify
import requests
import logging
import os
from datetime import datetime

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

app = Flask(__name__)

@app.route('/')
def index():
    """Serve the main dashboard"""
    return render_template('dashboard.html')

@app.route('/health')
def health():
    """Health check endpoint"""
    return jsonify({
        'status': 'healthy',
        'timestamp': datetime.now().isoformat(),
        'service': 'crypto-web-server'
    })

@app.route('/api/prices')
def get_prices():
    """Get cryptocurrency prices"""
    try:
        # Sample data - in production, fetch from real API
        crypto_data = {
            'bitcoin': {'price': 45000, 'change': 2.5},
            'ethereum': {'price': 3200, 'change': -1.2},
            'cardano': {'price': 0.85, 'change': 5.8}
        }

        return jsonify({
            'success': True,
            'data': crypto_data,
            'timestamp': datetime.now().isoformat()
        })
    except Exception as e:
        logger.error(f"Error fetching prices: {e}")
        return jsonify({
            'success': False,
            'error': str(e)
        }), 500

if __name__ == '__main__':
    port = int(os.environ.get('PORT', 5000))
    app.run(host='0.0.0.0', port=port, debug=False)
```

2. Node.js Express API

Repository Structure:

```
express-api/
├── server.js
├── package.json
├── routes/
│   └── api.js
└── README.md
```

Task Prompt:

```
# Express API Server

**Task Prompt:** "Deploy a Node.js Express API server with authentication middleware, rate limiting, and CORS support. The API should provide endpoints for user management and data retrieval."
```

File: `package.json`

```
{
  "name": "express-api",
  "version": "1.0.0",
  "description": "Express API server with authentication",
  "main": "server.js",
  "scripts": {
    "start": "node server.js",
    "dev": "nodemon server.js",
    "test": "jest"
  },
  "dependencies": {
    "express": "^4.18.0",
    "cors": "^2.8.5",
    "helmet": "^7.0.0",
    "express-rate-limit": "^6.8.0",
    "jsonwebtoken": "^9.0.0",
    "bcryptjs": "^2.4.3",
    "dotenv": "^16.3.0"
  },
  "devDependencies": {
    "nodemon": "^3.0.0",
    "jest": "^29.0.0"
  }
}
```

3. Docker-based Application

Repository Structure:

```
docker-app/
├── Dockerfile
├── docker-compose.yml
├── app/
│   ├── main.py
│   └── requirements.txt
└── README.md
```

Task Prompt:

```
# Dockerized Python Application

**Task Prompt:** "Deploy a containerized Python application using Docker. The applica-
tion should include a web interface, database connectivity, and proper logging. Use
docker-compose for orchestration."
```

File: `Dockerfile`

```
FROM python:3.11-slim

WORKDIR /app

COPY app/requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

COPY app/ .

EXPOSE 8000

HEALTHCHECK --interval=30s --timeout=10s --start-period=5s --retries=3 \
  CMD curl -f http://localhost:8000/health || exit 1

CMD ["python", "main.py"]
```

Template Creation Guidelines

AI Module Templates

1. **Complete Configuration**: Include all necessary fields in `ai-module.yaml`
2. **Proper Dependencies**: List all required dependencies with versions
3. **Health Checks**: Implement proper health check endpoints
4. **Security**: Configure appropriate security settings
5. **Documentation**: Provide clear README with usage instructions

Task Prompt Templates

1. **Clear Prompts**: Write specific, actionable task descriptions
2. **Standard Structure**: Follow common project structure conventions
3. **Dependency Files**: Include proper dependency management files
4. **Health Endpoints**: Implement basic health check endpoints
5. **Error Handling**: Include proper error handling and logging

Template Validation

AI Module Validation

```python
from ai_module_parser import AIModuleParser

def validate_ai_module_template(template_path):
    """Validate AI module template"""
    parser = AIModuleParser()

    try:
        config = parser.parse_file(f"{template_path}/ai-module.yaml")
        print(f"✅ Valid AI module: {config.name}")
        return True
    except Exception as e:
        print(f"❌ Invalid AI module: {e}")
        return False

# Usage
validate_ai_module_template("./templates/web-app")
```

Task Prompt Validation

```python
import os
from pathlib import Path

def validate_task_prompt_template(template_path):
    """Validate task prompt template"""
    path = Path(template_path)

    # Check for README with task prompt
    readme_path = path / "README.md"
    if not readme_path.exists():
        print("❌ Missing README.md file")
        return False

    readme_content = readme_path.read_text()
    if "task prompt" not in readme_content.lower():
        print("❌ README.md missing task prompt description")
        return False

    # Check for dependency files
    has_deps = any([
        (path / "requirements.txt").exists(),
        (path / "package.json").exists(),
        (path / "Dockerfile").exists()
    ])

    if not has_deps:
        print("❌ Missing dependency management files")
        return False

    print("✅ Valid task prompt template")
    return True

# Usage
validate_task_prompt_template("./templates/simple-web-server")
```

Template Repository Management

Repository Organization

```
templates/
├── ai-module-examples/
│   ├── web-app/
│   ├── api/
│   ├── ml-model/
│   └── microservice/
├── task-prompt-examples/
│   ├── simple-web-server/
│   ├── express-api/
│   └── docker-app/
└── README.md
```

Template Metadata

Create a `template-metadata.json` file for each template:

```json
{
  "name": "crypto-dashboard",
  "type": "ai_module",
  "category": "web_app",
  "description": "React-based cryptocurrency dashboard",
  "tags": ["react", "typescript", "crypto", "dashboard"],
  "difficulty": "intermediate",
  "estimated_deployment_time": "5-10 minutes",
  "requirements": {
    "cpu": "200m",
    "memory": "512Mi",
    "storage": "1Gi"
  },
  "author": "XplainCrypto Team",
  "version": "1.0.0",
  "last_updated": "2024-01-15",
  "documentation_url": "https://docs.xplaincrypto.ai/templates/crypto-dashboard"
}
```

Template Testing

```
#!/bin/bash
# Template testing script

TEMPLATE_DIR="$1"
TEMPLATE_TYPE="$2"

if [ -z "$TEMPLATE_DIR" ] || [ -z "$TEMPLATE_TYPE" ]; then
    echo "Usage: $0 <template_dir> <ai_module|task_prompt>"
    exit 1
fi

echo "Testing template: $TEMPLATE_DIR"

# Create test deployment
if [ "$TEMPLATE_TYPE" = "ai_module" ]; then
    # Test AI module deployment
    python -c "
from orchestrator import EnhancedTwoTierOrchestrator
import asyncio

async def test():
    orchestrator = EnhancedTwoTierOrchestrator()
    result = await orchestrator.run_unified_workflow(
        repository_url='file://$TEMPLATE_DIR'
    )
    print('Test result:', result)

asyncio.run(test())
"
else
    # Test task prompt deployment
    python -c "
from orchestrator import EnhancedTwoTierOrchestrator
import asyncio

async def test():
    orchestrator = EnhancedTwoTierOrchestrator()
    result = await orchestrator.run_unified_workflow(
        repository_url='file://$TEMPLATE_DIR',
        task_prompt='Deploy this application with proper configuration'
    )
    print('Test result:', result)

asyncio.run(test())
"
fi

echo "Template test completed"
```

Best Practices

AI Module Templates

1. **Version Pinning**: Pin dependency versions for reproducibility
2. **Resource Limits**: Set appropriate resource limits
3. **Security First**: Enable security features by default
4. **Health Monitoring**: Include comprehensive health checks

5. **Documentation**: Provide detailed configuration documentation

Task Prompt Templates

1. **Clear Instructions**: Write clear, specific task prompts
2. **Standard Conventions**: Follow language-specific conventions
3. **Minimal Dependencies**: Keep dependencies minimal and focused
4. **Error Handling**: Include proper error handling
5. **Logging**: Implement structured logging

General Guidelines

1. **Testing**: Test templates thoroughly before publishing
2. **Documentation**: Provide comprehensive documentation
3. **Versioning**: Use semantic versioning for templates
4. **Security**: Follow security best practices
5. **Maintenance**: Keep templates updated with latest practices

This template guide provides a comprehensive foundation for creating and managing both AI Module and Task Prompt repositories in the Enhanced Two-Tiered Multi-Agent Orchestration System.