LangGraph 1.0 Production Implementation

Overview

This document provides comprehensive information about the LangGraph 1.0 production orchestrator implementation for PowerShell script analysis.

Architecture

Components

```
┌─────────────────────────────────────────────────────────────┐
│                     FastAPI Application                      │
│  ┌───────────────────────────────────────────────────────┐  │
│  │         LangGraph Endpoints (/langgraph/*)             │  │
│  └──────────────────────┬────────────────────────────────┘  │
│                         │                                    │
│  ┌──────────────────────▼────────────────────────────────┐  │
│  │      LangGraphProductionOrchestrator                   │  │
│  │  ┌─────────────────────────────────────────────────┐  │  │
│  │  │           StateGraph Workflow                    │  │  │
│  │  │                                                   │  │  │
│  │  │  START                                            │  │  │
│  │  │    ↓                                              │  │  │
│  │  │  [analyze]  ← feedback ─ [human_review]          │  │  │
│  │  │    ↓                                              │  │  │
│  │  │  [tools] → [analyze] → [synthesis] → END         │  │  │
│  │  │                                                   │  │  │
│  │  └─────────────────────────────────────────────────┘  │  │
│  │                                                         │  │
│  │  Tools:                                                 │  │
│  │  • analyze_powershell_script                           │  │
│  │  • security_scan                                        │  │
│  │  • quality_analysis                                     │  │
│  │  • generate_optimizations                              │  │
│  └─────────────────────────────────────────────────────────┘  │
│                         │                                      │
│  ┌──────────────────────▼────────────────────────────────┐  │
│  │              Checkpointer                              │  │
│  │  • MemorySaver (dev)                                   │  │
│  │  • PostgresSaver (prod)                                │  │
│  └───────────────────────────────────────────────────────┘  │
└─────────────────────────────────────────────────────────────┘
```

State Schema

```
class PowerShellAnalysisState(TypedDict):
    # Message history with automatic deduplication
    messages: Annotated[Sequence[BaseMessage], add_messages]

    # Script content being analyzed
    script_content: Optional[str]

    # Analysis results from various stages
    analysis_results: Dict[str, Any]

    # Current workflow stage
    current_stage: str

    # Security findings
    security_findings: List[Dict[str, Any]]

    # Code quality metrics
    quality_metrics: Dict[str, float]

    # Optimization recommendations
    optimizations: List[Dict[str, Any]]

    # Error tracking
    errors: List[Dict[str, Any]]

    # Metadata
    workflow_id: str
    started_at: str
    completed_at: Optional[str]

    # Human-in-the-loop flags
    requires_human_review: bool
    human_feedback: Optional[str]

    # Final output
    final_response: Optional[str]
```

API Reference

Analyze Script

Analyze a PowerShell script using the LangGraph orchestrator.

Endpoint: `POST /langgraph/analyze`

Request Body:

```
{
  "script_content": "Get-Process | Where-Object CPU -gt 100",
  "thread_id": "optional_thread_id",
  "require_human_review": false,
  "stream": false,
  "model": "gpt-4",
  "api_key": "optional_api_key"
}
```

Response:

```
{
  "workflow_id": "analysis_1704649200.123",
  "status": "completed",
  "final_response": "This script retrieves processes with CPU usag
  "analysis_results": {
    "analyze_powershell_script": {
      "purpose": "Process monitoring",
      "complexity": "Low",
      "line_count": 1
    },
    "security_scan": {
      "risk_level": "LOW",
      "risk_score": 0,
      "findings": []
    },
    "quality_analysis": {
      "quality_score": 6.0,
      "metrics": {
        "total_lines": 1,
        "code_lines": 1
      }
    }
  },
  "current_stage": "completed",
  "started_at": "2026-01-07T12:00:00.000Z",
  "completed_at": "2026-01-07T12:00:04.523Z",
  "requires_human_review": false
}
```

Provide Human Feedback

Continue a paused workflow with human feedback.

Endpoint: `POST /langgraph/feedback`

Request Body:

```
{
  "thread_id": "analysis_1704649200.123",
  "feedback": "The security analysis looks good. Please proceed wi
}
```

Response: Same format as analyze response

Health Check

Check the health of the LangGraph service.

Endpoint: `GET /langgraph/health`

Response:

```
{
  "status": "healthy",
  "service": "LangGraph Production Orchestrator",
  "version": "1.0.5",
  "features": {
    "checkpointing": true,
    "human_in_the_loop": true,
    "streaming": true,
    "durable_execution": true
  },
  "model": "gpt-4",
  "checkpointer_type": "MemorySaver"
}
```

Service Info

Get detailed information about the orchestrator.

Endpoint: `GET /langgraph/info`

Response:

```
{
  "orchestrator": "LangGraph Production Orchestrator",
  "version": "1.0.5",
  "description": "Production-grade PowerShell script analysis usi
  "workflow_stages": ["analyze", "tools", "synthesis", "human_revi
  "available_tools": [
    {
      "name": "analyze_powershell_script",
      "description": "Analyze script purpose, functionality, and r
    },
    {
      "name": "security_scan",
      "description": "Comprehensive security analysis and vulnerab
    },
    {
      "name": "quality_analysis",
      "description": "Code quality evaluation and best practices c
    },
    {
      "name": "generate_optimizations",
      "description": "Generate optimization recommendations"
    }
  ],
  "supported_models": ["gpt-4", "gpt-4-turbo", "gpt-3.5-turbo"]
}
```

Batch Analysis

Analyze multiple scripts concurrently.

Endpoint: `POST /langgraph/batch-analyze`

Request Body:

```json
{
  "scripts": [
    "Get-Process",
    "Get-Service | Where-Object Status -eq 'Running'"
  ]
}
```

Response:

```json
{
  "total": 2,
  "successful": 2,
  "failed": 0,
  "results": [
    {
      "index": 0,
      "workflow_id": "analysis_1704649200.123",
      "status": "completed"
    },
    {
      "index": 1,
      "workflow_id": "analysis_1704649200.456",
      "status": "completed"
    }
  ],
  "errors": []
}
```

Test Orchestrator

Test the orchestrator with a sample script.

Endpoint: `POST /langgraph/test`

Response:

```
{
  "test_status": "passed",
  "result": {
    "workflow_id": "...",
    "status": "completed",
    ...
  }
}
```

Tools

analyze_powershell_script

Analyzes script purpose, structure, and basic metrics.

Input: Script content string

Output:

```
{
  "purpose": "File search and filtering",
  "complexity": "Medium",
  "parameters": {"Path": "string"},
  "functions": [],
  "line_count": 7,
  "timestamp": "2026-01-07T12:00:00.000Z"
}
```

security_scan

Performs comprehensive security analysis.

Input: Script content string

Output:

```
{
  "risk_level": "MEDIUM",
  "risk_score": 12,
  "findings": [
    {
      "category": "Network Activity",
      "severity": 5,
      "pattern": "invoke-webrequest",
      "description": "Makes web requests"
    }
  ],
  "findings_count": 1,
  "best_practices": ["Uses error handling", "Uses parameter valida
  "timestamp": "2026-01-07T12:00:00.000Z"
}
```

Security Patterns Detected: - `invoke-expression` - Code injection risk (severity: 10) - `downloadstring` - Remote code execution (severity: 9) - `bypass` - Security control bypass (severity: 8) - `-encodedcommand` - Obfuscation (severity: 8) - `hidden` - Stealth execution (severity: 7) - `downloadfile` - Untrusted download (severity: 7) - `start-process` - Process creation (severity: 6) - `add-type` - Code compilation (severity: 6) - `invoke-webrequest` - Network activity (severity: 5)

Risk Levels: - CRITICAL: risk_score > 30 - HIGH: risk_score > 20 - MEDIUM: risk_score > 10 - LOW: risk_score ≤ 10

quality_analysis

Evaluates code quality and best practices.

Input: Script content string

Output:

```
{
  "quality_score": 7.5,
  "metrics": {
    "total_lines": 50,
    "comment_lines": 10,
    "empty_lines": 5,
    "code_lines": 35,
    "comment_ratio": 0.286
  },
  "issues": ["Script is very long - consider breaking into modules
  "recommendations": [
    "Add comment-based help",
    "Implement try/catch error handling"
  ],
  "timestamp": "2026-01-07T12:00:00.000Z"
}
```

Quality Scoring: - Base score: 5.0 - +1.0 for CmdletBinding - +0.5 for param block - +0.5 for comments (>10% ratio) - +1.0 for try/catch blocks - -0.5 for very long scripts (>500 lines) - -0.5 for many long lines (>5 lines over 120 chars)

generate_optimizations

Generates optimization recommendations.

Input: Script content and quality metrics

Output:

```
{
  "total_optimizations": 3,
  "optimizations": [
    {
      "category": "Performance",
      "priority": "Medium",
      "recommendation": "Use .ForEach() method instead of ForEach-
      "impact": "Can improve loop performance by 2-3x"
    },
    {
      "category": "Reliability",
      "priority": "High",
      "recommendation": "Add try/catch blocks for error handling",
      "impact": "Prevents script failures and improves debugging"
    },
    {
      "category": "Documentation",
      "priority": "Medium",
      "recommendation": "Add comment-based help",
      "impact": "Improves code understanding and maintenance"
    }
  ],
  "timestamp": "2026-01-07T12:00:00.000Z"
}
```

Workflow

Standard Analysis Flow

1. **START** → Initial state created
2. **analyze** → LLM determines required analysis
3. **tools** → Execute requested tools (security_scan, quality_analysis, etc.)
4. **analyze** → LLM processes tool results
5. **synthesis** → Generate final comprehensive response
6. **END** → Workflow complete

Human-in-the-Loop Flow

1. **START** → Initial state created with `requires_human_review=true`
2. **analyze** → LLM determines required analysis
3. **tools** → Execute requested tools
4. **human_review** → Workflow pauses, waiting for feedback
5. *[Human provides feedback via `/feedback` endpoint]*
6. **analyze** → Re-analyze with human input
7. **synthesis** → Generate final response
8. **END** → Workflow complete

Checkpointing

Development (MemorySaver)

```
orchestrator = LangGraphProductionOrchestrator(
    api_key="...",
    use_postgres_checkpointing=False
)
```

- Stores state in memory
- Fast for development/testing
- No persistence across restarts
- Suitable for development environments

Production (PostgresSaver)

```
orchestrator = LangGraphProductionOrchestrator(
    api_key="...",
    use_postgres_checkpointing=True,
    postgres_connection_string="postgresql://user:pass@host/db"
)
```

- Stores state in PostgreSQL
- Durable across restarts
- Supports state recovery
- Required for production

Database Schema:

```
CREATE TABLE checkpoints (
    thread_id TEXT PRIMARY KEY,
    checkpoint_id TEXT NOT NULL,
    parent_checkpoint_id TEXT,
    checkpoint JSONB NOT NULL,
    metadata JSONB,
    created_at TIMESTAMP DEFAULT NOW()
);

CREATE INDEX idx_checkpoints_thread ON checkpoints(thread_id);
CREATE INDEX idx_checkpoints_parent ON checkpoints(parent_checkpo
```

Configuration

Environment Variables

```
# OpenAI API Key (required)
OPENAI_API_KEY=sk-...

# Model selection
DEFAULT_MODEL=gpt-4

# Checkpointing (production)
USE_POSTGRES_CHECKPOINTING=true
DATABASE_URL=postgresql://user:pass@host:5432/psscript

# Logging
LOG_LEVEL=INFO

# Feature flags
ENABLE_LANGGRAPH=true
LANGGRAPH_TRAFFIC_PERCENTAGE=100
```

Code Configuration

```python
# Initialize orchestrator
orchestrator = LangGraphProductionOrchestrator(
    api_key=os.getenv("OPENAI_API_KEY"),
    model="gpt-4",
    use_postgres_checkpointing=True,
    postgres_connection_string=os.getenv("DATABASE_URL")
)

# Analyze with options
result = await orchestrator.analyze_script(
    script_content=script,
    thread_id="my-session-id",
    require_human_review=False,
    stream=False
)
```

Monitoring

Metrics to Track

1. **Performance Metrics**
2. Workflow duration (p50, p95, p99)
3. Tool execution time
4. LLM response time
5. Total analysis time
6. **Reliability Metrics**
7. Success rate
8. Error rate by type
9. Retry count
10. Checkpoint recovery success
11. **Resource Metrics**
12. Memory usage
13. State size
14. Database query count
15. API calls per analysis
16. **Business Metrics**
17. Analyses per day
18. Average tools used per analysis
19. Human review request rate
20. User satisfaction

Logging

The orchestrator uses structured logging:

```
 logger.info(f"Entering analyze_node for workflow {workflow_id}")
logger.warning("Human review required - workflow paused")
logger.error(f"Error in analyze_script: {e}", exc_info=True)
```

Log Levels: - `DEBUG`: Detailed state transitions - `INFO`: Workflow milestones - `WARNING`: Recoverable issues - `ERROR`: Failures requiring attention

Error Handling

Automatic Recovery

The orchestrator handles errors gracefully:

```
try:
    result = await orchestrator.analyze_script(script)
except Exception as e:
    # Error logged and returned in response
    return {
        "error": str(e),
        "workflow_id": workflow_id,
        "status": "failed"
    }
```

Retry Logic

Tool execution includes automatic retries:

```
@tool
def security_scan(script_content: str) -> str:
    try:
        # Analysis logic
        return json.dumps(result)
    except Exception as e:
        logger.error(f"Error in security_scan: {e}")
        return json.dumps({"error": str(e)})
```

State Recovery

With checkpointing enabled, workflows can resume after failures:

```
 # Original execution fails
result = await orchestrator.analyze_script(
    script_content=script,
    thread_id="session-123"
)

# Resume from checkpoint after restart
orchestrator = LangGraphProductionOrchestrator()
result = await orchestrator.analyze_script(
    script_content=script,
    thread_id="session-123"  # Same thread ID
)
# Continues from last checkpoint
```

Best Practices

1. Always Use Thread IDs for Sessions

```
# Good
result = await orchestrator.analyze_script(
    script_content=script,
    thread_id=user_session_id
)

# Bad - generates random ID
result = await orchestrator.analyze_script(
    script_content=script
)
```

2. Enable Checkpointing in Production

```
# Production
orchestrator = LangGraphProductionOrchestrator(
    use_postgres_checkpointing=True,
    postgres_connection_string=DATABASE_URL
)

# Development only
orchestrator = LangGraphProductionOrchestrator()
```

3. Handle Streaming Properly

```
if stream:
    async for event in orchestrator.graph.astream(state, config):
        # Process each event
        yield event
```

4. Use Human Review for Sensitive Scripts

```
 # For scripts with high risk scores
result = await orchestrator.analyze_script(
    script_content=script,
    require_human_review=True if risk_score > 50 else False
)
```

5. Monitor and Alert

```
 # Track metrics
duration = result["completed_at"] - result["started_at"]
if duration > 10:  # seconds
    alert("Slow analysis detected")

# Track errors
if result["status"] == "failed":
    alert(f"Analysis failed: {result['error']}")
```

Troubleshooting

Issue: Workflow hangs in tool execution

Symptoms: Analysis doesn't complete, stuck in "tool_execution" stage

Diagnosis:

```
# Check state
logger.info(f"Current stage: {state['current_stage']}")
logger.info(f"Last message: {state['messages'][-1]}")
```

Solution: Increase timeout or check tool implementation

Issue: Checkpoint not found

Symptoms: `KeyError: 'checkpoint_id'`

Diagnosis: Thread ID doesn't exist or checkpointing not enabled

Solution:

```
# Ensure checkpointing is enabled
orchestrator = LangGraphProductionOrchestrator(
    use_postgres_checkpointing=True,
    postgres_connection_string=DATABASE_URL
)
```

Issue: High memory usage

Symptoms: Memory grows during execution

Diagnosis: State accumulation in long-running workflows

Solution: Limit message history or use PostgreSQL checkpointing

```
# Trim old messages from state
state["messages"] = state["messages"][-10:]  # Keep last 10
```

Issue: Tool execution errors

Symptoms: Tools return error responses

Diagnosis: Check tool logs and input validation

Solution:

```
# Add input validation
@tool
def analyze_powershell_script(script_content: str) -> str:
    if not script_content or len(script_content) < 10:
        return json.dumps({"error": "Invalid script content"})
    # Continue with analysis
```

Performance Optimization

1. Parallel Tool Execution

Tools are executed sequentially by default. For independent tools:

```
# Future enhancement: parallel tool execution
results = await asyncio.gather(
    security_scan(script),
    quality_analysis(script)
)
```

2. Caching

Cache frequent analysis results:

```
from functools import lru_cache

@lru_cache(maxsize=1000)
def get_cached_analysis(script_hash: str) -> Dict:
    # Return cached result if available
    pass
```

3. Model Selection

Use faster models for simple scripts:

```
# Simple script - use faster model
if len(script.split('\n')) < 20:
    model = "gpt-3.5-turbo"
else:
    model = "gpt-4"
```

4. Batch Processing

Process multiple scripts together:

```
results = await asyncio.gather(*[
    orchestrator.analyze_script(script)
    for script in scripts
])
```

Testing

Unit Tests

```python
import pytest
from agents.langgraph_production import (
    analyze_powershell_script,
    security_scan,
    quality_analysis
)

def test_security_scan_dangerous_script():
    script = "Invoke-Expression $userInput"
    result = json.loads(security_scan(script))
    assert result["risk_level"] == "CRITICAL"
    assert len(result["findings"]) > 0

def test_quality_analysis():
    script = "[CmdletBinding()]\nparam($Path)\nGet-Item $Path"
    result = json.loads(quality_analysis(script))
    assert result["quality_score"] >= 5.0
```

Integration Tests

```python
@pytest.mark.asyncio
async def test_full_workflow():
    orchestrator = LangGraphProductionOrchestrator()
    script = "Get-Process | Select-Object Name, CPU"

    result = await orchestrator.analyze_script(script)

    assert result["status"] == "completed"
    assert result["final_response"] is not None
    assert "analysis_results" in result
```

Migration from Legacy Agents

See LANGGRAPH-MIGRATION-PLAN.md for comprehensive migration guide.

Quick Migration:

```
# Old (legacy agent coordinator)
from agents.agent_coordinator import AgentCoordinator
coordinator = AgentCoordinator(api_key=api_key)
result = await coordinator.analyze_script(script)

# New (LangGraph orchestrator)
from agents.langgraph_production import LangGraphProductionOrches
orchestrator = LangGraphProductionOrchestrator(api_key=api_key)
result = await orchestrator.analyze_script(script_content=script)
```

Resources

Documentation

- LangGraph Official Docs
- LangChain Documentation
- Migration Plan

Code Examples

- Production orchestrator: `src/ai/agents/langgraph_production.py`
- API endpoints: `src/ai/langgraph_endpoints.py`
- Tests: `tests/test_langgraph_*.py`

Support

- GitHub Issues: Repository Issues
- Team Chat: #ai-platform
- Documentation: `/docs`

Document Version: 1.0 **Last Updated**: 2026-01-07 **Maintainer**: AI Platform Team

Generated 2026-01-16 21:23 UTC