Application Updates and Enhancements

This document outlines the significant updates and enhancements made to the PowerShell Script Analysis application, with a focus on incorporating agentic capabilities using LangGraph and Py-g frameworks.

Agentic Capabilities Integration

1. LangGraph Agent Implementation

- Created a new `LangGraphAgent` class that leverages LangGraph's explicit state management and multi-actor workflows
- Implemented a state-based graph structure with planning, execution, error handling, and response generation nodes
- Added support for stateful conversations with working memory persistence
- Integrated with existing script analysis pipeline
- Implemented explicit state transitions for more predictable agent behavior
- Added checkpoint capabilities for state persistence and recovery

2. Py-g Declarative Agent Implementation

- Created a new `PyGAgent` class that uses Py-g's declarative approach to agent definitions
- Implemented workflow-based execution with explicit state transitions
- Added support for structured planning and execution
- Integrated with existing script analysis pipeline
- Implemented task-based execution model with explicit task queues
- Added support for dynamic task generation based on context

3. Agent Factory Enhancements

- Updated the `AgentFactory` to support LangGraph and Py-g agent types
- Added dynamic agent selection based on task complexity and requirements
- Implemented graceful fallback mechanisms when specific agent types are unavailable
- Enhanced logging for better debugging and monitoring
- Added support for agent-specific configuration options
- Implemented agent caching for better performance

4. Multi-Turn Conversation Support

- Enhanced the conversation system to maintain context across multiple turns

- Implemented state tracking for complex multi-step tasks
- Added support for conversation history in agent decision-making
- Improved context management for more coherent responses
- Added support for conversation branching and merging
- Implemented conversation summarization for long interactions

5. Advanced Tool Use Framework

- Created a structured tool definition and execution system
- Implemented PowerShell-specific analysis tools for script evaluation
- Added security analysis tools for vulnerability detection
- Integrated documentation tools for command reference lookup
- Added support for tool composition and chaining
- Implemented tool result caching for better performance

6. Planning Capabilities

- Implemented explicit planning nodes in the agent workflow
- Added support for plan creation, execution, and revision
- Enhanced error recovery with plan adaptation
- Improved task decomposition for complex requests
- Added support for hierarchical planning
- Implemented plan visualization for debugging

7. State Management Enhancements

- Added explicit state schemas for all agent types
- Implemented working memory for persistent information across turns
- Created state transition logic for workflow management
- Added state checkpointing for recovery and debugging
- Implemented state visualization tools
- Added support for state rollback in case of errors

8. Error Handling and Recovery

- Implemented comprehensive error detection and classification
- Added recovery strategies for different error types
- Created fallback mechanisms for graceful degradation

- Enhanced logging for error analysis and debugging
- Implemented automatic retry logic for transient errors
- Added support for human intervention in critical errors

9. External API Integration

- Added structured interfaces for OpenAI API interactions
- Implemented rate limiting and retry logic for API stability
- Enhanced error handling for API failures
- Added support for model switching based on task requirements
- Implemented API response caching for better performance
- Added support for streaming responses for better user experience

10. Database Schema Verification

- Verified and enhanced the existing database schema
- Added support for storing agent state and conversation history
- Implemented efficient query patterns for script analysis retrieval
- Enhanced data integrity checks for analysis results
- Added support for vector embeddings for similarity search
- Implemented database migration tools for schema updates

11. Testing Framework

- Created comprehensive test scripts for LangGraph and Py-g agents
- Implemented test cases for different script types and complexity levels
- Added support for automated testing of agent capabilities
- Enhanced test reporting for better debugging
- Implemented performance benchmarking for agent comparison
- Added support for regression testing

12. Documentation and Examples

- Created detailed documentation for agent capabilities and usage
- Added examples for different use cases and scenarios
- Implemented interactive examples for better understanding
- Enhanced API documentation for developers
- Added troubleshooting guides for common issues

- Implemented documentation generation from code comments

Specific Technical Improvements

13. LangGraph Graph Structure Implementation

```
def _build_graph(self) -> StateGraph:
    """Build the agent workflow graph."""
    # Define the nodes
    planner = self._create_planner_node()
    tool_executor_node = ToolNode(self.tool_executor)
    responder = self._create_responder_node()
    error_handler = self._create_error_handler_node()

    # Create the graph
    workflow = StateGraph(AgentState)

    # Add nodes
    workflow.add_node("planner", planner)
    workflow.add_node("tool_executor", tool_executor_node)
    workflow.add_node("responder", responder)
    workflow.add_node("error_handler", error_handler)

    # Add edges
    workflow.add_edge("planner", "tool_executor")
    workflow.add_edge("tool_executor", "planner")
    workflow.add_edge("planner", "responder")
    workflow.add_edge("responder", END)

    # Add error handling
    workflow.add_edge_from_exception("tool_executor", "error_hand
    workflow.add_edge("error_handler", "planner")

    # Set the entry point
    workflow.set_entry_point("planner")

    # Compile the graph
    return workflow.compile()
```

14. Py-g Declarative Agent Definition

```
def _plan_tasks(self, user_request: str) -> List[Tuple[str, Dict[
    """
    Plan tasks based on the user's request.

    Args:
        user_request: The user's request

    Returns:
        A list of tasks to execute, where each task is a tuple of
        (task_name, task_args)
    """
    # Check if the request contains a PowerShell script
    if "```powershell" in user_request or "```ps1" in user_reques
        # Extract the script content
        script_content = self._extract_script(user_request)

        # Plan tasks for script analysis
        return [
            ("script_analysis", {"content": script_content}),
            ("security_analysis", {"content": script_content}),
            ("categorization", {"content": script_content}),
            ("code_improvement", {"content": script_content})
        ]
    else:
        # For general PowerShell questions, just use the LLM
        return [
            ("documentation_lookup", {"query": user_request})
        ]
```

15. Enhanced Agent Selection Logic

```
def determine_agent_type(self, message: str) -> str:
    """
    Determine the most appropriate agent type for a message.

    Args:
        message: The user's message

    Returns:
        The recommended agent type ('langchain', 'autogpt', 'hybri
    """
    # Simple heuristic for now - could be replaced with a more sop
    langgraph_task_indicators = [
        "multi-actor workflow",
        "state management",
        "explicit state",
        "workflow graph",
        "complex workflow",
        "state machine",
        "error recovery",
        "multi-step planning",
        "tool orchestration",
        "structured workflow"
    ]

    pyg_task_indicators = [
        "declarative agent",
        "declarative definition",
        "agent definition",
        "workflow-based",
        "state-based agent",
        "explicit workflow",
        "agent workflow",
        "declarative approach",
        "structured agent",
        "agent orchestration"
    ]
```

```
    # Additional indicators...

    message_lower = message.lower()

    # Check for LangGraph task indicators first (if available)
    if LANGGRAPH_AVAILABLE:
        for indicator in langgraph_task_indicators:
            if indicator in message_lower:
                logger.info(f"Selected LangGraph agent based on i
                return "langgraph"

    # Check for Py-g task indicators (if available)
    if PYG_AVAILABLE:
        for indicator in pyg_task_indicators:
            if indicator in message_lower:
                logger.info(f"Selected Py-g agent based on indica
                return "pyg"

    # Additional checks...

    # Default to LangChain for simpler queries
    logger.info("Selected LangChain agent (default)")
    return "langchain"
```

16. PowerShell Analysis Tool Implementation

```
class PowerShellAnalysisTool(BaseTool):
    """Tool for analyzing PowerShell scripts."""

    name = "powershell_analysis"
    description = "Analyze a PowerShell script to identify its pu

    def _run(self, script_content: str, run_manager: Optional[Cal
        """
        Analyze a PowerShell script.

        Args:
            script_content: The content of the PowerShell script

        Returns:
            A detailed analysis of the script
        """
        try:
            # Use the ScriptAnalyzer to analyze the script
            script_analyzer = ScriptAnalyzer(use_cache=True)
            analysis = script_analyzer.analyze_script_content(scr

            # Format the analysis as a structured response
            response = {
                "purpose": analysis.get("purpose", "Unknown purpo
                "security_score": analysis.get("security_score", 
                "code_quality_score": analysis.get("code_quality_
                "risk_score": analysis.get("risk_score", 5.0),
                "security_analysis": analysis.get("security_analy
                "optimization": analysis.get("optimization", []),
                "parameters": analysis.get("parameters", {})
            }

            return json.dumps(response, indent=2)
        except Exception as e:
            logger.error(f"Error analyzing PowerShell script: {e}"
            return f"Error analyzing script: {str(e)}"
```

17. Security Analysis Tool Implementation

```
class SecurityAnalysisTool(BaseTool):
    """Tool for analyzing security aspects of PowerShell scripts."

    name = "security_analysis"
    description = "Analyze a PowerShell script for security vulne

    def _run(self, script_content: str, run_manager: Optional[Cal
        """
        Analyze a PowerShell script for security issues.

        Args:
            script_content: The content of the PowerShell script

        Returns:
            A detailed security analysis of the script
        """
        script_lower = script_content.lower()
        security_issues = []

        # Check for common security issues
        if "invoke-expression" in script_lower or "iex " in scrip
            security_issues.append("Uses Invoke-Expression which

        if "convertto-securestring" in script_lower and "key" in
            security_issues.append("Uses ConvertTo-SecureString w

        # Additional checks...

        # Generate security score
        base_score = 5.0
        score_adjustment = min(len(security_issues) * -0.5, -4.0)
        security_score = max(1.0, min(10.0, base_score + score_ad

        # Format the response
        response = {
            "security_score": round(security_score, 1),
            "security_issues": security_issues,
```

```
            "security_best_practices": best_practices,
            "recommendations": [
                "Use parameter validation attributes to restrict i
                "Implement proper error handling with try/catch bl
                "Avoid using Invoke-Expression with user input",
                "Use ShouldProcess for functions that make changes
                "Implement logging for security-relevant actions"
            ]
        }

        return json.dumps(response, indent=2)
```

18. Script Categorization Tool Implementation

```
class ScriptCategorizationTool(BaseTool):
    """Tool for categorizing PowerShell scripts."""

    name = "script_categorization"
    description = "Categorize a PowerShell script into predefined

    def _run(self, script_content: str, run_manager: Optional[Cal
        """
        Categorize a PowerShell script.

        Args:
            script_content: The content of the PowerShell script

        Returns:
            The category of the script with explanation
        """
        # Define categories
        categories = {
            "System Administration": "Scripts for managing Windows
            "Security & Compliance": "Scripts for security auditi
            # Additional categories...
        }

        script_lower = script_content.lower()

        # Categorization logic
        if "get-process" in script_lower or "cpu" in script_lower
            category = "Monitoring & Diagnostics"
        elif "new-aduser" in script_lower or "get-aduser" in scri
            category = "Active Directory"
        # Additional categorization rules...

        return f"""
        Category: {category}

        Explanation: This script appears to be a {category.lower(
```

```
        {categories[category]}
        """
```

19. Asynchronous Processing Support

```
async def process_message(self, messages: List[Dict[str, str]]) -
    """
    Process a message using the LangGraph agent.

    Args:
        messages: List of message dictionaries with 'role' and 'co

    Returns:
        The agent's response as a string
    """
    try:
        # Initialize the state
        initial_state: AgentState = {
            "messages": messages,
            "tools": [{"name": tool.name, "description": tool.desc
            "tool_results": [],
            "current_plan": None,
            "working_memory": {},
            "errors": [],
            "final_response": None
        }

        # Run the graph
        for event in self.graph.stream(initial_state, checkpointe
            if "final_response" in event:
                return event["final_response"]

        # If we get here, something went wrong
        return "I apologize, but I couldn't process your request.

    except Exception as e:
        logger.error(f"Error processing message with LangGraph age
        return f"I encountered an error while processing your requ
```

20. State Persistence and Recovery

```
 # In LangGraphAgent.__init__
self.memory_saver = MemorySaver()

# In process_message method
for event in self.graph.stream(initial_state, checkpointer=self.me
    if "final_response" in event:
        return event["final_response"]
```

21. Error Handling Implementation

```
def _create_error_handler_node(self) -> Callable:
    """Create the error handler node for the graph."""
    def error_handler(state: AgentState, error: Exception) -> Dic
        """Handle errors that occur during tool execution."""
        # Log the error
        logger.error(f"Error during tool execution: {error}")

        # Add the error to the state
        errors = state.get("errors", [])
        errors.append({"error": str(error), "timestamp": time.time

        # Add a message about the error
        messages = state.get("messages", [])
        messages.append({"role": "system", "content": f"Error: {s

        # Return the updated state
        return {"errors": errors, "messages": messages}

    return error_handler
```

22. Agent Factory Update for New Agent Types

```
 # Import new agent implementations
try:
    from .langgraph_agent import LangGraphAgent
    LANGGRAPH_AVAILABLE = True
except ImportError:
    LANGGRAPH_AVAILABLE = False
    logging.warning("LangGraph agent not available. Install with

try:
    from .py_g_agent import PyGAgent
    PYG_AVAILABLE = True
except ImportError:
    PYG_AVAILABLE = False
    logging.warning("Py-g agent not available. Install with 'pip
```

23. Script Analysis with LangGraph

```
async def analyze_script(self, script_id: str, content: str,
                         include_command_details: bool = False,
                         fetch_ms_docs: bool = False) -> Dict[str,
    """
    Analyze a PowerShell script using the LangGraph agent.

    Args:
        script_id: The ID of the script to analyze
        content: The content of the script
        include_command_details: Whether to include detailed comma
        fetch_ms_docs: Whether to fetch Microsoft documentation re

    Returns:
        A dictionary containing the analysis results
    """
    logger.info(f"Starting LangGraph analysis for script {script_

    # Create a message for script analysis
    messages = [
        {"role": "system", "content": "You are an expert PowerShel
        {"role": "user", "content": f"""
        Analyze the following PowerShell script and provide a deta

        1. PURPOSE: Summarize what this script is designed to do
        2. SECURITY_ANALYSIS: Identify potential security vulnera
        3. CODE_QUALITY: Evaluate code quality and best practices
        4. PARAMETERS: Identify and document all parameters, inclu
        5. CATEGORY: Classify this script into an appropriate cate
        6. OPTIMIZATION: Provide specific suggestions for improvi
        7. RISK_ASSESSMENT: Evaluate the potential risk of executi

        Script content:
        ```powershell
        {content}
        ```
        """}
    ]
```

```
    # Initialize the state and run the graph
    # ...

    # Parse the final response to extract structured information
    # ...

    return analysis_results
```

24. Microsoft Documentation Reference Tool

```
def _fetch_ms_docs_references(self, content: str, state: Dict[str
    """Fetch Microsoft documentation references for PowerShell co
    # Check if MS Docs references are in working memory
    if "working_memory" in state and "ms_docs_references" in state
        return state["working_memory"]["ms_docs_references"]

    # Extract commands from the script
    import re
    command_pattern = r'((?:Get|Set|New|Remove|Add|Start|Stop|Impo

    commands_found = re.findall(command_pattern, content)
    unique_commands = list(set(commands_found))

    # Create MS Docs references for each command
    ms_docs_references = []
    for cmd in unique_commands:
        # Create a reference with a search URL
        search_url = f"https://learn.microsoft.com/en-us/search/?t
        ms_docs_references.append({
            "title": f"{cmd} Documentation",
            "url": search_url,
            "description": f"Microsoft documentation for {cmd} Pow
        })

    return ms_docs_references
```

25. Py-g Task Planning Implementation

```
async def _analyze_script(self, content: str) -> Dict[str, Any]:
    """
    Analyze a PowerShell script.

    Args:
        content: The content of the PowerShell script

    Returns:
        Analysis results
    """
    try:
        # Use the script analyzer to analyze the script
        analysis = self.script_analyzer.analyze_script_content(co

        return {
            "section": "Script Analysis",
            "content": f"Purpose: {analysis.get('purpose', 'Unkno
                       f"Code Quality Score: {analysis.get('code_q
        }
    except Exception as e:
        logger.error(f"Error analyzing script: {e}")
        return {
            "section": "Script Analysis",
            "content": f"Error analyzing script: {str(e)}"
        }
```

26. Py-g Response Generation

```
def _generate_response(self, results: List[Dict[str, Any]]) -> st
    """
    Generate a response based on task results.

    Args:
        results: List of task results

    Returns:
        The generated response
    """
    # Combine results into a coherent response
    if not results:
        return "I couldn't analyze your request. Please provide a

    # Start with a header
    response_parts = ["Here's my analysis:"]

    # Add each result section
    for result in results:
        if "section" in result and "content" in result:
            response_parts.append(f"\n## {result['section']}")

            if isinstance(result["content"], list):
                # For lists (like suggestions or issues)
                for item in result["content"]:
                    response_parts.append(f"- {item}")
            else:
                # For text content
                response_parts.append(result["content"])

    # Join all parts
    return "\n".join(response_parts)
```

27. Testing Framework Implementation

```
async def test_langgraph_agent():
    """Test the LangGraph agent."""
    print("Testing LangGraph agent...")

    # Get the API key from the environment
    api_key = os.environ.get("OPENAI_API_KEY")
    if not api_key:
        print("Error: OPENAI_API_KEY environment variable not set"
        return

    # Get the LangGraph agent
    agent = agent_factory.get_agent("langgraph", api_key)

    # Analyze the sample script
    print("Analyzing sample PowerShell script...")
    analysis = await agent.analyze_script(
        "test-script",
        SAMPLE_SCRIPT,
        include_command_details=True,
        fetch_ms_docs=True
    )

    # Print the analysis results
    print("\nAnalysis Results:")
    print(f"Purpose: {analysis.get('purpose', 'Unknown')}")
    print(f"Security Score: {analysis.get('security_score', 0)}/1
    print(f"Code Quality Score: {analysis.get('code_quality_score
    print(f"Risk Score: {analysis.get('risk_score', 0)}/10")
    print(f"Category: {analysis.get('category', 'Unknown')}")

    # Additional tests...
```

28. Py-g Agent Testing

```
async def test_pyg_agent():
    """Test the Py-g agent."""
    print("\nTesting Py-g agent...")

    # Get the API key from the environment
    api_key = os.environ.get("OPENAI_API_KEY")
    if not api_key:
        print("Error: OPENAI_API_KEY environment variable not set"
        return

    # Get the Py-g agent
    agent = agent_factory.get_agent("pyg", api_key)

    # Analyze the sample script
    print("Analyzing sample PowerShell script...")
    analysis = await agent.analyze_script(
        "test-script",
        SAMPLE_SCRIPT,
        include_command_details=True,
        fetch_ms_docs=True
    )

    # Print the analysis results
    print("\nAnalysis Results:")
    print(f"Purpose: {analysis.get('purpose', 'Unknown')}")
    print(f"Security Score: {analysis.get('security_score', 0)}/10
    print(f"Code Quality Score: {analysis.get('code_quality_score
    print(f"Risk Score: {analysis.get('risk_score', 0)}/10")
    print(f"Category: {analysis.get('category', 'Unknown')}")

    # Test declarative workflow capabilities
    print("\nTesting declarative workflow capabilities...")
    messages = [
        {"role": "user", "content": "Can you analyze this deployme
    ]
```

```
    response = await agent.process_message(messages)
    print(f"\nWorkflow Response:\n{response}")
```

29. Dependency Management

```
# requirements.txt
fastapi==0.98.0
uvicorn==0.22.0
openai==0.27.8
numpy>=1.26.0
pandas>=2.0.2
scikit-learn>=1.2.2
pgvector==0.2.3
psycopg2-binary==2.9.6
pydantic==1.10.9
python-dotenv==1.0.0
httpx==0.24.1
tenacity==8.2.2
tiktoken==0.4.0
redis==5.0.1
diskcache==5.6.3

# LangChain and related dependencies
langchain==0.0.267
langchain-openai==0.0.2
langchain-community==0.0.10
langchain-core==0.1.1

# LangGraph and Py-g for agentic capabilities
langgraph>=0.0.20
pyg>=0.3.1
networkx>=3.1

# Additional dependencies...
```

30. Database Schema Updates

```
 -- Add support for storing agent state
CREATE TABLE IF NOT EXISTS agent_state (
    id SERIAL PRIMARY KEY,
    agent_id VARCHAR(255) NOT NULL,
    state JSONB NOT NULL,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP
);

-- Add support for storing conversation history
CREATE TABLE IF NOT EXISTS conversation_history (
    id SERIAL PRIMARY KEY,
    conversation_id VARCHAR(255) NOT NULL,
    messages JSONB NOT NULL,
    agent_type VARCHAR(50) NOT NULL,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP
);

-- Add support for storing tool execution results
CREATE TABLE IF NOT EXISTS tool_execution_results (
    id SERIAL PRIMARY KEY,
    tool_name VARCHAR(255) NOT NULL,
    input JSONB NOT NULL,
    output JSONB NOT NULL,
    execution_time FLOAT NOT NULL,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP
);
```

Summary of Improvements

The application has been significantly enhanced with advanced agentic capabilities through the integration of LangGraph and Py-g frameworks. These enhancements provide:

1. More sophisticated script analysis with explicit state management
2. Better handling of complex, multi-step tasks
3. Improved error recovery and resilience
4. Enhanced planning and reasoning capabilities
5. More structured tool use for specialized tasks
6. Better context management for multi-turn conversations
7. More detailed security analysis of PowerShell scripts
8. Improved categorization and documentation of scripts
9. More efficient state persistence and recovery
10. Better integration with external APIs and services
11. Comprehensive testing framework for agent capabilities
12. Detailed documentation and examples for developers
13. Enhanced database schema for storing agent state and conversation history
14. Improved performance through caching and optimization
15. Better user experience through streaming responses and interactive examples
16. Enhanced security through better error handling and validation
17. More flexible agent selection based on task requirements
18. Better debugging through state visualization and logging
19. Improved deployment and scaling capabilities
20. Enhanced monitoring and analytics for agent performance

These improvements make the application more capable, reliable, and user-friendly, while also providing a foundation for future enhancements and extensions.

Generated 2026-01-16 23:34 UTC