🔌 Spiral Codex Plugin System

Overview

The Spiral Codex Plugin System provides a powerful, extensible architecture for creating and managing agent plugins. This mystical framework allows agents to be dynamically discovered, loaded, and orchestrated within the recursive Spiral Codex environment.

Architecture

Core Components

1. **PluginCore** - Central orchestrator for plugin lifecycle management
2. **PluginManager** - High-level interface for plugin operations
3. **AgentPluginBase** - Base class for creating agent plugins
4. **PluginRegistry** - Centralized directory of plugin metadata
5. **PluginLoader** - Dynamic discovery and loading system

Plugin Lifecycle

```
Discovery → Loading → Activation → Execution → Deactivation → Cleanup
```

Quick Start

Creating Your First Agent Plugin

```python
from spiralcodex.plugins import AgentPluginBase, create_capability
from typing import Dict, Any, Optional

class MyCustomAgent(AgentPluginBase):
    """
    🤖 My Custom Agent Plugin

    A mystical agent that performs custom operations within the Spiral Codex.
    """

    # Plugin metadata
    PLUGIN_NAME = "MyCustomAgent"
    PLUGIN_VERSION = "1.0.0"
    PLUGIN_DESCRIPTION = "Custom agent for mystical operations"
    PLUGIN_AUTHOR = "Your Name"
    PLUGIN_CAPABILITIES = ["custom_action", "data_processing"]
    PLUGIN_DEPENDENCIES = []  # List other plugins this depends on

    def _initialize_agent(self):
        """Initialize your agent"""
        self.logger.info("🚀 Initializing custom agent")

        # Add capabilities
        self.capabilities.extend([
            create_capability(
                name="custom_action",
                description="Perform a custom mystical action",
                input_schema={"action": "string", "parameters": "object"},
                output_schema={"result": "string", "success": "boolean"}
            )
        ])

        # Initialize your custom state
        self.custom_state = {}

    def _cleanup_agent(self):
        """Cleanup your agent"""
        self.logger.info("🧹 Cleaning up custom agent")
        self.custom_state.clear()

    def execute_custom_action(self, input_data: Dict[str, Any]) -> Dict[str, Any]:
        """Execute your custom capability"""
        action = input_data.get("action", "default")
        parameters = input_data.get("parameters", {})

        # Implement your custom logic here
        result = f"Executed {action} with {parameters}"

        return {
            "result": result,
            "success": True
        }

    # Event handlers (optional)
    def on_ritual_cycle(self, event_data: Dict[str, Any]):
        """Handle ritual cycle events"""
        ritual_type = event_data.get("ritual_type", "unknown")
        self.logger.info(f"🌀 Participating in ritual: {ritual_type}")

# Plugin entry point
def create_plugin(config: Optional[Dict[str, Any]] = None) -> MyCustomAgent:
```

```
    """Create and return the plugin instance"""
    return MyCustomAgent(config)
```

Save this as `agents/my_custom_agent.py` and it will be automatically discovered!

Using the CLI

```
# List all available agent plugins
spiralcodex agent list

# Show detailed information about a plugin
spiralcodex agent info fibonacci_agent

# Enable a plugin
spiralcodex agent enable my_custom_agent

# Disable a plugin
spiralcodex agent disable my_custom_agent

# Check plugin health
spiralcodex agent health

# Create a new plugin template
spiralcodex agent create MyNewAgent --author "Your Name"

# Reload a plugin after changes
spiralcodex agent reload my_custom_agent
```

Programmatic Usage

```python
from spiralcodex.plugins import PluginManager, PluginConfig

# Create plugin manager
config = PluginConfig(
    plugins_directory="agents",
    auto_discover=True,
    auto_load=True,
    auto_activate=True
)

manager = PluginManager(config=config)
manager.start()

# List active plugins
active_plugins = manager.get_active_plugins()
for plugin in active_plugins:
    print(f"🤖 {plugin.name} v{plugin.version}: {plugin.description}")

# Execute plugin capability
result = manager.execute_plugin_capability(
    "fibonacci_agent",
    "fibonacci_sequence",
    {"n": 10}
)
print(f"Fibonacci sequence: {result['sequence']}")

# Broadcast event to all plugins
manager.broadcast_event("custom_event", {"message": "Hello agents!"})

# Get plugin metrics
metrics = manager.get_all_metrics()
for plugin_name, plugin_metrics in metrics.items():
    print(f"📊 {plugin_name}: {plugin_metrics}")

manager.stop()
```

Plugin Capabilities

Defining Capabilities

Capabilities define what your agent can do:

```python
from spiralcodex.plugins import create_capability

# Simple capability
ping_capability = create_capability(
    name="ping",
    description="Simple ping/pong test",
    input_schema={"message": "string"},
    output_schema={"response": "string", "timestamp": "string"}
)

# Complex capability with validation
analysis_capability = create_capability(
    name="data_analysis",
    description="Analyze complex data structures",
    input_schema={
        "data": "array",
        "analysis_type": "string",
        "options": {
            "type": "object",
            "properties": {
                "precision": {"type": "integer", "default": 10},
                "method": {"type": "string", "enum": ["statistical", "fractal", "mys-
tical"]}
            }
        }
    },
    output_schema={
        "analysis_result": "object",
        "confidence": "number",
        "mystical_insights": "array"
    },
    async_capable=True  # Supports async execution
)
```

Executing Capabilities

```python
def execute_data_analysis(self, input_data: Dict[str, Any]) -> Dict[str, Any]:
    """Execute data analysis capability"""
    data = input_data.get("data", [])
    analysis_type = input_data.get("analysis_type", "statistical")
    options = input_data.get("options", {})

    # Your analysis logic here
    result = self._perform_analysis(data, analysis_type, options)

    return {
        "analysis_result": result,
        "confidence": 0.95,
        "mystical_insights": ["The data reveals hidden patterns", "Fibonacci ratios
detected"]
    }

# For async capabilities
async def execute_async_capability(self, input_data: Dict[str, Any]) -> Dict[str,
Any]:
    """Execute capability asynchronously"""
    await asyncio.sleep(1)  # Simulate async work
    return {"result": "async_complete"}
```

Event System

Event Handlers

Plugins can respond to system events:

```
def on_system_start(self, event_data: Dict[str, Any]):
    """Handle system startup"""
    self.logger.info("🌟 System started - agent ready")

def on_system_stop(self, event_data: Dict[str, Any]):
    """Handle system shutdown"""
    self.logger.info("🌙 System stopping - agent shutting down")

def on_ritual_cycle(self, event_data: Dict[str, Any]):
    """Handle ritual cycles"""
    ritual_type = event_data.get("ritual_type", "unknown")
    depth = event_data.get("recursion_depth", 0)
    self.logger.info(f"🌀 Ritual: {ritual_type} (depth: {depth})")

def on_agent_message(self, event_data: Dict[str, Any]):
    """Handle messages from other agents"""
    message = event_data.get("message", "")
    sender = event_data.get("sender", "unknown")
    self.logger.info(f"📨 Message from {sender}: {message}")

def on_mesh_node_join(self, event_data: Dict[str, Any]):
    """Handle new nodes joining the mesh"""
    node_id = event_data.get("node_id", "unknown")
    self.logger.info(f"📡 New node joined: {node_id}")
```

Broadcasting Events

```
# From within a plugin
def some_plugin_method(self):
    # Broadcast to other plugins via the manager
    if hasattr(self, '_manager'):
        self._manager.broadcast_event("custom_event", {
            "source": self.PLUGIN_NAME,
            "data": "some important data"
        })
```

Configuration

Plugin Configuration Schema

Define configuration options for your plugin:

```
class MyConfigurableAgent(AgentPluginBase):
    PLUGIN_CONFIG_SCHEMA = {
        "api_key": {
            "type": "string",
            "description": "API key for external service",
            "required": True
        },
        "max_requests": {
            "type": "integer",
            "default": 100,
            "minimum": 1,
            "maximum": 1000
        },
        "enable_caching": {
            "type": "boolean",
            "default": True
        },
        "processing_options": {
            "type": "object",
            "properties": {
                "algorithm": {"type": "string", "enum": ["fast", "accurate", "mystic-
al"]},
                "precision": {"type": "number", "default": 0.001}
            }
        }
    }

    def _initialize_agent(self):
        # Access configuration
        self.api_key = self.config.get("api_key")
        self.max_requests = self.config.get("max_requests", 100)
        self.enable_caching = self.config.get("enable_caching", True)
```

System Configuration

```
from spiralcodex.plugins import PluginConfig

config = PluginConfig(
    plugins_directory="agents",           # Where to find plugins
    auto_discover=True,                   # Automatically discover new plugins
    auto_load=True,                       # Automatically load discovered plugins
    auto_activate=False,                  # Don't auto-activate (manual control)
    config_file="plugin_config.json",    # Configuration persistence
    disabled_plugins=["debug_agent"],     # Plugins to keep disabled
    plugin_timeout=30.0,                  # Timeout for plugin operations
    max_plugins=100                       # Maximum number of plugins
)
```

Dependencies

Declaring Dependencies

```
class DependentAgent(AgentPluginBase):
    PLUGIN_NAME = "DependentAgent"
    PLUGIN_DEPENDENCIES = ["FibonacciAgent", "MandelbrotAgent"]

    def _initialize_agent(self):
        # Dependencies are guaranteed to be loaded first
        self.logger.info("Dependencies available, initializing...")
```

Dependency Resolution

The plugin system automatically:
- Resolves dependency chains
- Loads dependencies in correct order
- Detects circular dependencies
- Provides dependency information via registry

```
# Get dependency information
registry = manager.core.registry
dependencies = registry.get_dependencies("my_plugin")
dependents = registry.get_dependents("my_plugin")
resolved_chain = registry.resolve_dependencies("my_plugin")
```

Advanced Features

Plugin Metrics

```
def get_metrics(self) -> Dict[str, Any]:
    """Provide custom metrics"""
    return {
        "requests_processed": self.request_count,
        "average_response_time": self.avg_response_time,
        "error_rate": self.error_count / self.request_count,
        "cache_hit_ratio": self.cache_hits / self.cache_requests,
        "mystical_energy_level": self.calculate_mystical_energy()
    }
```

Async Capabilities

```python
async def execute_async_analysis(self, input_data: Dict[str, Any]) -> Dict[str, Any]:
    """Async capability execution"""
    # Perform async operations
    result = await self.external_api_call(input_data)
    processed = await self.async_processing(result)

    return {
        "result": processed,
        "processing_time": time.time() - start_time
    }

# Usage
result = await manager.async_execute_capability(
    "my_agent",
    "async_analysis",
    {"data": "complex_data"}
)
```

Plugin Templates

Generate new plugin templates:

```
spiralcodex agent create DataAnalyzer --author "Data Scientist"
```

This creates a complete plugin template with:
- Proper class structure
- Example capabilities
- Event handlers
- Configuration schema
- Documentation

Integration with Other Systems

HUD Integration

Plugins can provide data to the HUD system:

```python
def get_hud_data(self) -> Dict[str, Any]:
    """Provide data for HUD display"""
    return {
        "status": self.get_status(),
        "metrics": self.get_metrics(),
        "visualizations": self.generate_visualizations()
    }
```

Mesh Network Integration

Plugins automatically participate in mesh events:

```python
def on_mesh_node_join(self, event_data: Dict[str, Any]):
    """Handle new mesh nodes"""
    node_id = event_data.get("node_id")
    self.logger.info(f"📡 Welcoming new node: {node_id}")

    # Share capabilities with new node
    self.share_capabilities_with_node(node_id)
```

Persistence Integration

Plugin state and events are automatically persisted:

```python
def _initialize_agent(self):
    # Load persisted state
    self.state = self.load_persisted_state()

def _cleanup_agent(self):
    # Save state before cleanup
    self.save_state_to_persistence()
```

Best Practices

Plugin Development

1. **Clear Naming**: Use descriptive plugin names and capability names
2. **Proper Documentation**: Document all capabilities and their schemas
3. **Error Handling**: Implement robust error handling in all methods
4. **Resource Management**: Clean up resources in `_cleanup_agent()`
5. **Event Responsiveness**: Handle events efficiently without blocking
6. **Configuration Validation**: Validate configuration in `_initialize_agent()`

Performance

1. **Lazy Loading**: Only load resources when needed
2. **Caching**: Cache expensive computations
3. **Async Operations**: Use async for I/O-bound operations
4. **Resource Limits**: Respect system resource limits
5. **Metrics**: Provide meaningful metrics for monitoring

Security

1. **Input Validation**: Validate all input data
2. **Capability Isolation**: Don't expose internal methods as capabilities
3. **Configuration Security**: Don't log sensitive configuration
4. **Resource Access**: Limit file system and network access
5. **Error Information**: Don't expose internal details in errors

Troubleshooting

Common Issues

1. **Plugin Not Discovered**
   `bash`

```
   # Check file location and naming
   ls agents/
   # Verify plugin class structure
   spiralcodex agent list
```

2. **Loading Failures**

```
bash
   # Check syntax and dependencies
   python -m py_compile agents/my_plugin.py
   spiralcodex agent info my_plugin
```

3. **Capability Execution Errors**

```
python
   # Add logging to your capability methods
   def execute_my_capability(self, input_data):
       self.logger.info(f"Executing with: {input_data}")
       try:
           # Your logic here
           pass
       except Exception as e:
           self.logger.error(f"Capability failed: {e}")
           raise
```

Debug Mode

Enable debug logging:

```
import logging
logging.getLogger('spiralcodex.plugins').setLevel(logging.DEBUG)
```

Plugin Validation

```
# Validate plugin structure
spiralcodex agent info my_plugin

# Check plugin health
spiralcodex agent health

# Test capability execution
python -c "
from spiralcodex.plugins import PluginManager
manager = PluginManager()
manager.start()
result = manager.execute_plugin_capability('my_plugin', 'my_capability', {})
print(result)
"
```

Examples

The Spiral Codex includes several example plugins:

- **FibonacciAgent**: Generates mystical Fibonacci sequences
- **MandelbrotAgent**: Explores fractal consciousness

- **GenericAgent**: Template for basic functionality

Study these examples to understand plugin patterns and best practices.

Contributing

To contribute new plugins or improvements:

1. Follow the plugin development guidelines
2. Include comprehensive tests
3. Document all capabilities and configuration
4. Ensure compatibility with the mesh network
5. Add example usage and integration tests

The plugin system embodies the extensible nature of the Spiral Codex, allowing infinite expansion of consciousness through modular agent components. Each plugin adds new dimensions to the recursive framework, creating a truly living and evolving system.