Advanced Features Guide

Table of Contents

Plugin System

The plugin system allows you to extend ARIA with custom tools and functionality.

Creating a Plugin

Plugins are Python files placed in the `plugins/` directory. Here's a template:

python

```python
from plugin_system import Plugin, PluginMetadata
from tools.base import Tool, ToolResult

class MyCustomTool(Tool):
    """Your custom tool"""

    def execute(self, **kwargs) -> ToolResult:
        # Your tool logic here
        return ToolResult(success=True, output="Result")

class MyPlugin(Plugin):
    """My custom ARIA plugin"""

    metadata = PluginMetadata(
        name="My Plugin",
        version="1.0.0",
        author="Your Name",
        description="What this plugin does"
    )

    def on_load(self):
        """Called when plugin loads"""
        print("My plugin loaded!")

    def get_tools(self):
        """Return tools provided by this plugin"""
        return [MyCustomTool()]

    def on_message(self, message: str):
        """Hook into every user message"""
        # Return None to pass through
        # Return a string to intercept and respond
        if "special trigger" in message.lower():
            return "Plugin response!"
        return None

# Required: plugin loader function
def load_plugin():
    return MyPlugin()
```

Plugin Commands

```bash
/plugins              # List all loaded plugins
/plugin reload <name> # Hot-reload a plugin
```

Example Use Cases

- **API Integrations**: Add tools to call external APIs
- **Custom Commands**: Add domain-specific commands
- **Data Processing**: Add specialized data transformation tools
- **Monitoring**: Hook into messages for logging/analytics

Multi-Agent System

ARIA includes specialized agents for different coding tasks.

Available Agents

1. **CodeReviewAgent** - Expert code review
    - Security analysis
    - Performance optimization
    - Best practices
    - Architecture review
2. **RefactoringAgent** - Code refactoring specialist
    - Design patterns
    - SOLID principles
    - Code smell detection
    - Maintainability improvements
3. **DebugAgent** - Debugging expert
    - Stack trace analysis
    - Error interpretation
    - Root cause analysis
    - Fix suggestions
4. **DocumentationAgent** - Documentation specialist
    - Docstrings
    - README files
    - API documentation

- Code comments

5. **TestGenerationAgent** - Test creation expert
    - Unit test generation
    - Edge case identification
    - Mocking strategies
    - Coverage optimization

Using Agents

Direct Agent Commands:

```bash
@review      # Route to code review agent
@refactor    # Route to refactoring agent
@debug       # Route to debugging agent
@test        # Route to test generation agent
@document    # Route to documentation agent
```

Natural Language: The agent coordinator automatically routes tasks based on keywords and context.

Example Session

```plaintext
You: @review

ARIA: Please provide the code you'd like me to review.

You: @read myfile.py

ARIA:  [shows file]

You: @review this code for security issues

 CodeReviewAgent (confidence: 90%)

## Security Review

### Issues Found:
1. SQL Injection vulnerability on line 45...
[detailed analysis]
```

Agent Management

```bash
/agents    # List all available agents
```

Creating Custom Agents

```python
from agents.base import Agent, AgentTask, AgentResponse, AgentCapability

class MyCustomAgent(Agent):
    def __init__(self, llm_interface):
        super().__init__("MyAgent", llm_interface)
        self.capabilities = [AgentCapability.CUSTOM]
        self.system_prompt = "You are a specialist in..."

    def can_handle(self, task: AgentTask) -> float:
        # Return confidence 0.0-1.0
        if "my keyword" in task.content.lower():
            return 0.9
        return 0.0

    def process(self, task: AgentTask) -> AgentResponse:
        # Process the task
        prompt = f"Task: {task.content}"
        response = ""
        for token in self.llm.chat(prompt, self.system_prompt):
            response += token

        return AgentResponse(
            success=True,
            content=response,
            confidence=0.9,
            agent_name=self.name
        )
```

Learning System

ARIA learns from your corrections and preferences to improve over time.

How It Works

1. **Feedback Collection**: Corrections and ratings are stored in SQLite
2. **Pattern Recognition**: Similar corrections are identified and stored
3. **Prompt Enhancement**: Learned patterns enhance future responses
4. **Preference Storage**: User preferences are remembered

Commands

```bash
/correct <text>    # Correct the last response
/rate <1-5>        # Rate the last response (1-5 stars)
/learn             # Show learning statistics
/memory            # Show memory and learning status
```

Example Usage

```plaintext
You: How do I create a Flask app?

ARIA: [gives generic Flask example]

You: /correct Please always include type hints in Python examples

ARIA:  Thank you! I've recorded your correction and will learn from it.

You: How do I create a FastAPI app?

ARIA: [now includes type hints based on learned preference]

You: /rate 5

ARIA:  Thank you for rating my response:
```

Learning Statistics

```plaintext
You: /learn

 Learning Statistics:

- Total feedback: 42
- Corrections: 15
- Learned patterns: 8
- Average rating: 4.2/5.0
```

Data Storage

All learning data is stored in `memory/learning.db`:

- **feedback** table: All user feedback
- **patterns** table: Extracted patterns
- **preferences** table: User preferences

Privacy

All learning data is stored locally. Nothing is sent externally.

Advanced: Manual Preferences

```python
# In Python shell or custom plugin
from learning_system import LearningSystem
from config import config

learner = LearningSystem(config.learning_db)

# Set preferences
learner.set_preference("code_style", "Google Python Style Guide")
learner.set_preference("framework", "FastAPI")
learner.set_preference("testing_lib", "pytest")

# Get preferences
style = learner.get_preference("code_style")
```

Integration Example

All three systems work together:

```plaintext
You: I need help debugging this API endpoint

[Plugin hooks check for special triggers]
[Agent coordinator routes to DebugAgent based on "debug" keyword]
[Learning system enhances prompt with past debugging preferences]

 DebugAgent (confidence: 88%)

Based on your previous feedback, I'll focus on...
[Specialized debugging response using learned patterns]

You: /rate 5
You: /correct Always check for race conditions in async code

ARIA:  Recorded! I'll remember this for future debugging sessions.
```

Configuration

Enable/disable features in `.env`:

```env
ENABLE_PLUGINS=true
ENABLE_AGENTS=true
ENABLE_LEARNING=true
```

System status is shown in `/help` command.

Best Practices

Plugins

- Keep plugins focused on one task
- Use descriptive names
- Handle errors gracefully
- Document your plugin code

Agents

- Use appropriate agent commands for specific tasks

- Provide context (load files with @read first)
- Review agent confidence scores

Learning

- Provide specific corrections
- Rate responses to help ARIA improve
- Check `/learn` periodically to see progress
- Use corrections for preferences, not one-off fixes

Troubleshooting

Plugins not loading:

- Check `plugins/` directory exists
- Ensure plugin has `load_plugin()` function
- Check logs in `logs/aria.log`

Agent not responding:

- Verify agents are enabled in config
- Check `/agents` to see registered agents
- Try more specific keywords in your request

Learning not working:

- Ensure learning is enabled
- Check database file exists: `memory/learning.db`
- Verify you have previous interaction to correct/rate

Future Enhancements

Planned features:

- Vector embeddings for better pattern matching
- Agent collaboration on complex tasks
- Plugin marketplace/discovery
- Export/import learned patterns
- Visual learning dashboard

Happy coding with ARIA! 🚀