**# Development Rules for Selfy**

This document outlines the strict development rules that must be followed throughout the Selfy project to ensure an organized, maintainable codebase and prevent the issues that plagued the previous implementation.

**## 1. Directory Structure and Organization**

**### Rules:**

```

selfy\_project/

├── selfy\_core/           # Production code only

│   ├── docs/             # Production documentation

│   ├── global\_modules/   # Shared modules

│   ├── user\_pipeline/    # User interaction pipeline

│   └── self\_dev\_pipeline/# Self-development pipeline

├── selfy\_dev/            # Development code

│   ├── docs/             # Development documentation

│   ├── experiments/      # Clearly marked experimental code

│   └── prototypes/       # Prototype implementations

├── tests/                # Test suite

│   ├── unit/             # Unit tests

│   ├── integration/      # Integration tests

│   └── mocks/            # Mock data

└── tools/                # Development tools

    ├── validators/       # Structure and naming validators

    ├── templates/        # Code and doc templates

    └── hooks/            # Git hooks for enforcement

```

- Maintain a strict, predefined directory structure

- Development code must stay in `selfy\_dev/` directory

- Production code must stay in `selfy\_core/` directory

- Test code must be in `tests/` directories adjacent to the code they test

- Mock data must be stored in `tests/mocks/` directories

- Documentation must be in `docs/` directories at appropriate levels

**### Enforcement:**

- Include directory structure diagrams in all development documents

- Implement pre-commit hooks that validate file locations

- Create a "structure validator" tool that runs periodically

**## 2. Naming Conventions**

**### Rules:**

- All file names must follow `snake\_case` convention

- Module names must be descriptive and unique across the codebase

- Class names must follow `PascalCase` convention

- No generic names like "utils.py" or "helper.py" without specific prefixes

- Test files must be named `test\_[module\_name].py`

- Mock data files must be named `mock\_[data\_type].json`

**### Enforcement:**

- Create a naming convention checker tool

- Include naming rules in code templates

- Reject commits that violate naming conventions

**## 3. Documentation Requirements**

**### Rules:**

- Every module must have a docstring explaining its purpose and relationships

- Every class and function must have a docstring

- README.md files must exist at each directory level

- Architecture documentation must be updated when architecture changes

- Code examples in documentation must be tested

**### Enforcement:**

- Automated docstring coverage checker

- Documentation validation in CI pipeline

- Regular documentation review sessions

**## 4. Code Quality and Duplication**

**### Rules:**

- No duplicate code across modules

- Maximum cyclomatic complexity limits for functions

- Maximum function and class sizes

- Consistent code style following PEP 8

- No commented-out code in production

**### Enforcement:**

- Run linters and code quality tools automatically

- Implement duplicate code detection

- Set hard limits in CI pipeline

**## 5. Testing Discipline**

**### Rules:**

- All production code must have unit tests

- Minimum test coverage requirements (e.g., 80%)

- Mock data must be versioned and validated

- Tests must be independent and not rely on global state

- Test scaffolding must be cleaned up after tests

**### Enforcement:**

- Automated test coverage reports

- Test isolation validation

- Mock data validation tools

**## 6. Development Workflow**

**### Rules:**

- Development must follow the defined architecture

- New capabilities must be registered in the capability manifest

- Changes must be traceable to requirements or gaps

- Experimental code must be clearly marked and isolated

- Regular cleanup of development artifacts

**### Enforcement:**

- Architecture compliance checking

- Capability manifest validation

- Traceability matrix maintenance

- Regular cleanup sprints

**## Implementation Strategy**

**### 1. Development Environment Setup**

Create a development environment that enforces these rules from the start:

- Configure linters and formatters

- Set up pre-commit hooks

- Establish CI/CD pipeline with validation

**### 2. Automated Enforcement Tools**

**\*\*Structure Validator\*\***

- Checks file locations against the defined structure

- Runs on commit and in CI pipeline

**\*\*Naming Convention Checker\*\***

- Validates file, class, and function names

- Integrated with IDE and CI pipeline

**\*\*Documentation Validator\*\***

- Checks docstring coverage and README presence

- Validates links and examples in documentation

**\*\*Code Quality Suite\*\***

- Runs linters, complexity checkers, and duplicate detectors

- Enforces style guidelines

**### 3. Development Guidance Documents**

Create clear, concise guidance documents that are always accessible:

**\*\*Development Rules Summary\*\***

- One-page summary of all rules

- Included at the top of every development document

**\*\*Architecture Reference\*\***

- Quick reference to the agreed architecture

- Visual diagrams for easy understanding

**\*\*Workflow Guides\*\***

- Step-by-step guides for common development tasks

- Templates for new modules, tests, etc.

**### 4. Continuous Reinforcement**

**\*\*Start-of-Session Reminder\*\***

- When the agent starts a development session, remind it of the rules

- Include specific rules relevant to the current task

**\*\*In-Process Validation\*\***

- Periodically check compliance during development

- Provide immediate feedback on violations

**\*\*End-of-Session Review\*\***

- Validate all changes against the rules

- Clean up any temporary artifacts

**## Balancing Rules and Productivity**

To ensure these rules don't become a burden that distracts from actual development:

**### Automate Everything Possible**

- Rules should be enforced by tools, not manual checks

- Provide immediate, clear feedback on violations

**### Prioritize Rules**

- Categorize rules as "critical," "important," and "preferred"

- Focus enforcement on critical and important rules

**### Provide Clear Remediation Paths**

- When a rule is violated, provide clear instructions for fixing it

- Include examples of compliant alternatives

**### Regular Rule Review**

- Periodically review the rules for effectiveness

- Remove or modify rules that cause more problems than they solve

**## Implementation in Agent Instructions**

To ensure the agent follows these rules without being overwhelmed:

- Include a condensed rule summary in the agent's system prompt

- Create a "development rules" tool the agent can reference

- Implement automated validation that provides feedback to the agent

- Use a "development mode" that enforces stricter validation

**## Rule Summary for Quick Reference**

1. **\*\*Structure\*\***: Follow the directory structure strictly

2. **\*\*Naming\*\***: Use consistent naming conventions

3. **\*\*Documentation\*\***: Document everything thoroughly

4. **\*\*Quality\*\***: Maintain high code quality and avoid duplication

5. **\*\*Testing\*\***: Test everything and maintain high coverage

6. **\*\*Workflow\*\***: Follow the architecture and development process

These rules are designed to ensure that the codebase remain organized, maintainable, and free from the issues that plagued the previous implementation. By following these rules consistently, we can create a high-quality, robust agent that can continue to evolve over time.