Agent Patterns

Effective patterns for creating specialized agents.

Table of Contents

Design Patterns

Single Responsibility Pattern

Each agent has one clear purpose.

Good:

```yaml
---
name: security-auditor
specialization: security
capabilities:
  - security-audit
  - vulnerability-scan
---
```

Bad:

```yaml
---
name: do-everything
capabilities:
  - security-audit
  - implement-features
  - write-tests
  - deploy-code
---
```

Why: Focused agents are easier to maintain, test, and delegate to.

Capability-Driven Pattern

Define capabilities clearly for proper routing.

Pattern:

```markdown
---
name: database-engineer
capabilities:
  - database-schema-design
  - database-migration
  - query-optimization
specialization: database
---

# Database Engineer

I specialize in database design and optimization.

## Capabilities

- Design database schemas
- Create and manage migrations
- Optimize slow queries
- Analyze database performance
```

Benefits: PM can route database tasks correctly.

Context-Aware Pattern

Agents understand project context.

Pattern:

```markdown
# My Agent

## Project Context

I will reference project memories for:
- Architecture patterns
- Coding standards
- Current implementation

## Workflow

1. Query project memories
2. Apply context to task
3. Store new learnings
```

Implementation:

```json
{
  "memory-update": {
    "Implementation Guidelines": ["Always use async/await for I/
        O"]
  }
}
```

Specialization Patterns

Domain Expert Pattern

Deep expertise in specific domain.

Example: Python Engineer

```yaml
---
name: python-engineer
specialization: python
capabilities:
  - python-implementation
  - python-refactoring
  - async-programming
---
```

```
# Python Engineer

I'm a senior Python engineer specializing in modern Python 3.11+
        development.

## Expertise

- Asynchronous programming (asyncio, aiohttp)
- Type hints and mypy
- Performance optimization
- Python best practices

## Standards

- Use type hints for all functions
- Prefer async for I/O operations
- Follow PEP 8 with Black formatting
- Use Pydantic for data validation
```

Language-Specific Pattern

Specialized for programming language.

Example: TypeScript Engineer

```
---
name: typescript-engineer
specialization: typescript
capabilities:
  - typescript-implementation
  - type-system-design
---

# TypeScript Engineer

Modern TypeScript development with latest features.

## Standards

- Strict mode enabled
- Prefer `const` over `let`
- Use branded types for type safety
- Leverage discriminated unions
```

Framework-Specific Pattern

Expertise in specific framework.

Example: Next.js Engineer

```
---
name: nextjs-engineer
specialization: nextjs
capabilities:
  - nextjs-implementation
  - app-router-patterns
---

# Next.js Engineer

Specialized in Next.js 15+ with App Router.

## Patterns

- Server Components by default
- Client Components only when needed
- Streaming and Suspense for loading
- Route handlers for API endpoints
```

Instruction Patterns

Clear Responsibility Pattern

State responsibilities explicitly.

Pattern:

```
# Agent Name

## Core Responsibilities

- Responsibility 1: Clear description
- Responsibility 2: Clear description
- Responsibility 3: Clear description

## Not Responsible For

- Task X (delegate to Agent Y)
- Task Z (delegate to Agent W)
```

Workflow Pattern

Define clear workflow steps.

Pattern:

```
## Workflow

1. **Analyze**: Understand requirements
2. **Plan**: Design approach
3. **Implement**: Execute task
```

```
4. **Validate**: Verify results
5. **Document**: Record learnings

For each step:
- [Specific guidance]
```

Example-Driven Pattern

Include examples for clarity.

Pattern:

```
## Examples

### Example 1: Feature Implementation

**Input**: "Add user authentication"

**Approach**:
1. Research existing auth patterns
2. Design JWT-based solution
3. Implement endpoints
4. Add tests

### Example 2: Bug Fix

**Input**: "Fix login timeout issue"

**Approach**:
1. Reproduce issue
2. Identify root cause
3. Implement fix
4. Add regression test
```

Memory Patterns

Learning Pattern

Store learnings systematically.

Pattern:

```
## Memory Storage

After completing tasks, store:

- **Architecture learnings** in Project Architecture
- **Code patterns** in Implementation Guidelines
- **Technical details** in Current Technical Context
```

Usage:

```json
{
  "memory-update": {
    "Project Architecture": ["API uses FastAPI with async
        endpoints"],
    "Implementation Guidelines": ["Use Pydantic models for
        validation"]
  }
}
```

Query-First Pattern

Check memories before acting.

Pattern:

```markdown
## Before Starting

1. Query project memories for relevant context
2. Apply learned patterns
3. Follow established guidelines
4. Store new learnings when done
```

Delegation Patterns

Delegation-Aware Pattern

Know when to delegate.

Pattern:

```markdown
## Delegation

I delegate to:

- **QA Agent**: For test creation
  - When: After implementation
  - Why: Testing expertise

- **Documentation Agent**: For docs
  - When: After feature completion
  - Why: Documentation expertise

- **Research Agent**: For analysis
  - When: Need codebase understanding
  - Why: Research capabilities
```

Return-to-PM Pattern

Return to PM when done or stuck.

Pattern:

```
## Completion

When I complete my task:
1. Summarize what was done
2. Return to PM for next step
3. Store relevant learnings

When I'm blocked:
1. Explain the issue
2. Return to PM for guidance
3. Don't try to work outside my scope
```

Anti-Patterns

God Agent Anti-Pattern

Bad:

```
---
name: super-agent
capabilities:
  - everything
---

# Super Agent

I can do anything and everything!
```

Why bad: Defeats purpose of specialization, makes routing unclear.

Fix: Create focused specialist agents.

No Capabilities Anti-Pattern

Bad:

```
---
name: my-agent
---

# My Agent

I do various coding tasks.
```

Why bad: PM can’t route tasks properly.

Fix: Define explicit capabilities.

Scope Creep Anti-Pattern

Bad:

```
# Engineer Agent

I implement features. I also do testing, documentation,
deployment, monitoring, and whatever else is needed.
```

Why bad: Unclear responsibilities, poor delegation.

Fix: Define clear boundaries and delegate.

No Memory Anti-Pattern

Bad:

```
# Agent

I complete tasks but never store learnings.
```

Why bad: No continuity across sessions.

Fix: Store relevant learnings in memory.

Duplicate Specialization Anti-Pattern

Bad:

```
.claude-mpm/agents/
├── python-expert.md
├── python-engineer.md
└── python-dev.md
```

Why bad: Confusing for PM routing.

Fix: One agent per specialization.

Next Steps: - Creating Agents: See creating-agents.md - PM Workflow: See pm-workflow.md - Extending: See ../developer/extending.md