Creating Agents

Step-by-step guide to creating custom agents.

Table of Contents

- Examples

Quick Start

Create your first agent in 5 minutes.

1. Create Agent File

```bash
# Create in PROJECT tier (highest priority)
claude-mpm agents create my-agent

# This creates: .claude-mpm/agents/my-agent.md
```

2. Edit Agent

```markdown
---
name: my-agent
model: claude-sonnet-4
capabilities:
  - my-capability
specialization: my-domain
delegation: true
version: "1.0.0"
---

# My Agent

Brief description of agent purpose.

## Responsibilities

- Responsibility 1
- Responsibility 2

## Workflow

1. Step 1
2. Step 2
3. Step 3
```

3. Validate

```bash
# Validate syntax
claude-mpm agents validate --agent my-agent

# Test agent
claude-mpm run -i "Task for my-agent" --agent my-agent
```

4. Use Agent

```
# Agent is automatically available
claude-mpm run

# In session:
"Ask my-agent to [task]"
```

Agent Structure

File Format

Agents use Markdown with YAML frontmatter:

```
---
[YAML frontmatter]
---

# Agent Name

[Markdown instructions]
```

Frontmatter Block

Configuration in YAML:

```yaml
---
name: agent-identifier
model: claude-model-name
capabilities:
  - capability1
  - capability2
specialization: domain
delegation: true
version: "1.0.0"
temperature: 0.7
max_tokens: 4096
---
```

Instructions Block

Agent behavior in Markdown:

```
# Agent Name

Brief description.

## Responsibilities

What this agent does.
```

```markdown
## Workflow

How this agent works.

## Examples

Example tasks.
```

Frontmatter Configuration

Required Fields

name (string): - Unique identifier - Lowercase with hyphens - Example: `python-engineer`

model (string): - Claude model to use - Options: `claude-sonnet-4`, `claude-opus-4` - Example: `claude-sonnet-4`

Optional Fields

capabilities (list): - What agent can do - Used for task routing - Example: `[python-implementation, async-programming]`

specialization (string): - Domain expertise - Used for categorization - Example: `python`

delegation (boolean): - Can delegate to other agents - Usually `true` for PM, `false` for specialists - Default: `false`

version (string): - Semantic version - Example: `"1.0.0"`

temperature (float): - Model temperature (0.0-1.0) - Lower = more focused, higher = more creative - Default: `0.7`

max_tokens (integer): - Maximum response tokens - Default: `4096`

timeout (integer): - Request timeout in seconds - Default: `300`

context_window (integer): - Context window size - Default: model default

Example Frontmatter

```yaml
---
name: security-auditor
model: claude-sonnet-4
capabilities:
  - security-audit
  - vulnerability-scan
  - penetration-testing
specialization: security
```

```yaml
delegation: false
version: "1.0.0"
temperature: 0.5
max_tokens: 8192
---
```

Writing Instructions

Structure

```markdown
# Agent Name

One-line description of agent's purpose.

## Core Responsibilities

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

## Workflow

1. **Step 1**: Description
2. **Step 2**: Description
3. **Step 3**: Description

## Standards & Best Practices

- Standard 1
- Standard 2
- Standard 3

## Delegation (if applicable)

When to delegate to:
- **Agent X**: For task type Y
- **Agent Z**: For task type W

## Examples

### Example 1: [Task Type]

**Input**: "[Example input]"

**Approach**:
1. [Step 1]
2. [Step 2]
3. [Step 3]

### Example 2: [Task Type]
```

```
[Another example]
```

Best Practices

Be Specific: - Clear, actionable instructions - Concrete examples - Specific guidelines

Be Focused: - Single, clear purpose - Well-defined scope - Know when to delegate

Be Contextual: - Reference project memories - Apply learned patterns - Store new learnings

Be Practical: - Real-world examples - Common scenarios - Error handling

Memory Integration

Include memory usage:

```
## Memory Usage

I store learnings in project memories:

- **Architecture decisions**: In "Project Architecture"
- **Code patterns**: In "Implementation Guidelines"
- **Technical details**: In "Current Technical Context"

I query memories before starting tasks to apply learned context.
```

Example JSON response:

```
{
  "memory-update": {
    "Project Architecture": ["Key architectural decision"],
    "Implementation Guidelines": ["Important coding pattern"]
  }
}
```

Testing Agents

Validation

```
# Validate syntax
claude-mpm agents validate --agent my-agent

# Common errors:
# - Missing required fields (name, model)
# - Invalid YAML syntax
# - Unescaped special characters
# - Wrong file extension
```

Manual Testing

```
# Test with specific input
claude-mpm run -i "Task for agent" --agent my-agent

# Test with monitoring
claude-mpm run --monitor --agent my-agent
```

Integration Testing

```
# Test in normal workflow (PM delegates)
claude-mpm run

# In session:
"Create a task that my-agent should handle"

# Watch delegation in dashboard
```

Debugging

Check logs:

```
# View agent logs
tail -f .claude-mpm/logs/agents/my-agent.log

# View system logs
tail -f .claude-mpm/logs/claude-mpm.log
```

Enable debug mode:

```
# Run with debug logging
claude-mpm run --debug
```

Check routing:

```
# List agents with capabilities
claude-mpm agents list --capabilities

# Verify your agent appears with correct capabilities
```

Deploying Agents

Tier Selection

PROJECT Tier (`.claude-mpm/agents/`): - Use for: Project-specific agents - Priority: Highest (overrides everything) - Scope: Single project

USER Tier (`~/.claude-agents/`): - Use for: Personal agents across projects - Priority: Medium (overrides SYSTEM) - Scope: All projects

SYSTEM Tier (bundled): - Use for: Built-in agents - Priority: Lowest - Scope: All installations

Deployment Commands

```
# Deploy to PROJECT tier (automatic on creation)
claude-mpm agents create my-agent

# Deploy to USER tier
claude-mpm agents create my-agent --tier user

# Redeploy agents
claude-mpm agents deploy

# Redeploy with force (rebuild all)
claude-mpm agents deploy --force
```

Agent Updates

```
# Edit agent file
vim .claude-mpm/agents/my-agent.md

# Validate changes
claude-mpm agents validate --agent my-agent

# Changes take effect immediately (no redeploy needed)
```

Version Management

Update version in frontmatter:

```
---
name: my-agent
version: "1.1.0"  # Increment version
---
```

Follow semantic versioning: - **Major (1.0.0)**: Breaking changes - **Minor (1.1.0)**: New features - **Patch (1.1.1)**: Bug fixes

Examples

Example 1: Security Auditor

```
---
name: security-auditor
model: claude-sonnet-4
capabilities:
  - security-audit
  - vulnerability-scan
```

```markdown
specialization: security
version: "1.0.0"
temperature: 0.5
---

# Security Auditor

I perform comprehensive security audits and vulnerability
        assessments.

## Core Responsibilities

- Identify security vulnerabilities
- Analyze authentication and authorization
- Review input validation and sanitization
- Check for common security issues (OWASP Top 10)

## Workflow

1. **Scan**: Review code for security issues
2. **Analyze**: Assess severity and impact
3. **Report**: Document findings with examples
4. **Recommend**: Suggest fixes and improvements

## Standards

- Follow OWASP guidelines
- Check for: SQL injection, XSS, CSRF, auth issues
- Verify input validation everywhere
- Review sensitive data handling

## Examples

### Example: API Security Audit

**Input**: "Audit the authentication system"

**Approach**:
1. Review auth endpoints
2. Check token handling
3. Verify password security
4. Test for common vulnerabilities
5. Document findings and recommendations
```

Example 2: Database Engineer

```markdown
---
name: database-engineer
model: claude-sonnet-4
capabilities:
  - database-schema-design
  - database-migration
  - query-optimization
specialization: database
version: "1.0.0"
---

# Database Engineer

I specialize in database design, migrations, and query
        optimization.

## Core Responsibilities

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

## Workflow

1. **Analyze**: Understand requirements
2. **Design**: Create schema or query design
3. **Implement**: Write SQL or migration code
4. **Test**: Verify performance and correctness
5. **Document**: Explain design decisions

## Standards

- Normalize to 3NF (unless performance requires denormalization)
- Use appropriate indexes
- Follow naming conventions: snake_case, descriptive names
- Always use migrations (never manual schema changes)

## Delegation

I delegate to:
- **QA Agent**: For integration tests
- **Documentation Agent**: For schema documentation

## Memory

I store in memories:
- Schema design patterns
- Performance optimization techniques
- Migration best practices
```

Example 3: Python Engineer

```
---
name: python-engineer
model: claude-sonnet-4
capabilities:
  - python-implementation
```

```markdown
  - python-refactoring
  - async-programming
specialization: python
version: "2.0.0"
---

# Python Engineer

Senior Python engineer specializing in Python 3.11+ with modern best practices.

## Core Responsibilities

- Implement features in Python
- Refactor code for maintainability
- Debug and fix issues
- Apply Python best practices

## Workflow

1. **Understand**: Analyze requirements and context
2. **Design**: Plan implementation approach
3. **Implement**: Write clean, typed, tested code
4. **Validate**: Test and verify
5. **Document**: Add docstrings and comments

## Standards

- **Type Hints**: Use for all functions
- **Async**: Prefer async/await for I/O
- **Formatting**: Black with line length 100
- **Validation**: Pydantic for data models
- **Testing**: pytest with fixtures

## Python 3.11+ Features

- Use structural pattern matching
- Leverage exception groups
- Apply performance improvements

## Delegation

- **QA Agent**: For comprehensive test suites
- **Documentation Agent**: For API documentation

## Memory

Store in Implementation Guidelines:
- Python patterns used
- Performance optimizations
- Testing approaches
```

Next Steps: - PM Workflow: See pm-workflow.md - Agent Patterns: See agent-patterns.md - Extending: See ../developer/extending.md