

The Complete Guide to Claude Code Sub-Agents

Table of Contents

1. [What Are Sub-Agents?](#)
2. [Why Use Sub-Agents?](#)
3. [How Sub-Agents Work](#)
4. [Setting Up Your First Sub-Agent](#)
5. [Sub-Agent Configuration Deep Dive](#)
6. [Best Practices for Sub-Agent Design](#)
7. [Advanced Usage Patterns](#)
8. [Real-World Examples](#)
9. [Common Pitfalls and Solutions](#)
10. [Team Collaboration with Sub-Agents](#)

What Are Sub-Agents?

Sub-agents are specialized AI assistants within Claude Code that handle specific types of tasks.

Think of them as expert team members, each with their own expertise, context window, and tool permissions.

Core Characteristics

- **Independent Context:** Each sub-agent operates in its own context window, preventing cross-contamination between different tasks
- **Specialized Expertise:** Configured with detailed instructions for specific domains (e.g., code review, security auditing, performance optimization)
- **Custom Tool Access:** Can be granted specific tools or inherit all tools from the main thread
- **Reusable Across Projects:** Once created, can be used in multiple projects and shared with team members

The Mental Model

Imagine you're leading a development team where you have:

- A **Senior Code Reviewer** who focuses only on code quality and best practices
- A **Security Specialist** who scans for vulnerabilities
- A **Performance Engineer** who optimizes bottlenecks
- A **Documentation Expert** who keeps all docs current

Sub-agents work exactly like this specialized team, but as AI assistants you can invoke instantly.

Why Use Sub-Agents?

1. Context Preservation

The main Claude Code conversation stays focused on high-level objectives while sub-agents handle specific tasks without polluting the primary context.

Without Sub-Agents:

You: "Review this authentication code and also help me plan the next sprint"
Claude: [Gets confused between code review and sprint planning contexts]

With Sub-Agents:

You: "Have code-reviewer analyze the auth code, then help me plan the sprint"
Claude: [Delegates to code-reviewer sub-agent, then continues with sprint planning]

2. Specialized Performance

Sub-agents trained for specific domains consistently outperform generalist approaches.

3. Scalability

As projects grow, you can add specialized sub-agents for new domains without overloading the main conversation.

4. Team Consistency

Shared sub-agents ensure consistent approaches across team members and projects.

5. Workflow Efficiency

Reduces context switching and enables parallel processing of different aspects of your codebase.

How Sub-Agents Work

Delegation Process

1. **Task Analysis:** Claude Code analyzes your request
2. **Sub-Agent Selection:** Matches the task to appropriate sub-agent based on descriptions
3. **Independent Execution:** Sub-agent works in its own context with specified tools
4. **Result Integration:** Sub-agent returns results to the main conversation

Context Isolation

Main Context: Project planning, architecture decisions, feature requirements

↓ delegates to

Sub-Agent Context: Code review guidelines, security patterns, performance metrics

↓ returns results

Main Context: Continues with architectural decisions

Automatic vs Explicit Invocation

Automatic: Claude Code chooses sub-agents based on context

"Please review the security of this authentication flow"

→ Automatically delegates to security-auditor sub-agent

Explicit: You specify which sub-agent to use

"Have security-auditor check this auth code for OWASP compliance"

→ Directly invokes the security-auditor

Setting Up Your First Sub-Agent

Method 1: Using the `/agents` Command (Recommended)

1. Open the interface:

```
bash
```

```
/agents
```

2. Create new agent:

- Select "Create New Agent"
- Choose project-level or user-level scope

3. Generate with Claude (Highly Recommended):

- Describe your sub-agent in detail
- Let Claude generate the initial configuration
- Customize the generated template to your needs

4. Configure tools:

- Select specific tools or inherit all
- The interface shows all available tools including MCP tools

Method 2: Manual File Creation

Create a file in `.claude/agents/` (project) or `~/.claude/agents/` (user):

markdown

```
---
name: code-reviewer
description: Senior code reviewer specializing in best practices, SOLID principles, and maintainability. Use PROACTIV
tools: read, write, bash
model: sonnet
---
```

You are a senior code reviewer with 10+ years of experience. When reviewing code:

- 1. **Structure & Architecture**
 - Check for SOLID principles adherence
 - Evaluate separation of concerns
 - Assess code organization and modularity
- 2. **Code Quality**
 - Look for code smells and anti-patterns
 - Suggest refactoring opportunities
 - Ensure consistent naming conventions
- 3. **Security & Performance**
 - Identify potential security vulnerabilities
 - Point out performance bottlenecks
 - Recommend optimizations
- 4. **Documentation & Testing**
 - Verify adequate commenting
 - Check for missing documentation
 - Assess test coverage gaps

Always provide specific, actionable feedback with examples.

Sub-Agent Configuration Deep Dive

File Structure Breakdown

```
yaml
---
name: unique-identifier      # Required: lowercase-with-hyphens
description: Activation criteria # Required: When to use this agent
tools: tool1, tool2, tool3   # Optional: Specific tools
model: sonnet                # Optional: Model selection
---
```

Configuration Fields

Field	Required	Purpose	Example
<code>name</code>	Yes	Unique identifier	<code>database-optimizer</code>
<code>description</code>	Yes	When to activate	<code>Database performance analysis and optimization. Use PROACTIVELY for query tuning.</code>
<code>tools</code>	No	Tool permissions	<code>read, write, bash</code>
<code>model</code>	No	Model selection	<code>sonnet</code> , <code>opus</code> , <code>haiku</code>

Model Selection Strategy

```
yaml

# For simple, deterministic tasks
model: haiku

# For standard development tasks (recommended default)
model: sonnet

# For complex analysis and critical operations
model: opus
```

Tool Configuration Patterns

```
yaml

# Inherit all tools (default)
tools:


# Read-only access (planning, analysis)
tools: read, search

# Development access (coding tasks)
tools: read, write, bash

# Full access (complex operations)
tools: read, write, bash, git, npm, pip
```

Best Practices for Sub-Agent Design

1. Single Responsibility Principle

 **Bad:** Generic "helper" sub-agent

```
yaml
```

```
---
```

```
name: helper
```

```
description: Helps with various tasks
```

```
---
```

```
You help with coding, reviews, documentation, and planning.
```

✓ **Good:** Focused sub-agent

```
yaml
```

```
---
```

```
name: api-documenter
```

```
description: Generate OpenAPI specifications and API documentation. Use PROACTIVELY for API documentation tasks
```

```
---
```

```
You specialize in creating comprehensive API documentation...
```

2. Detailed System Prompts

Include specific instructions, examples, and constraints:

markdown

name: security-auditor

description: OWASP security compliance checker. Use PROACTIVELY for security analysis.

tools: read, write

You are a security specialist focused on OWASP Top 10 vulnerabilities:

Security Analysis Framework

1. **Authentication & Authorization**

- Check for proper JWT handling
- Verify role-based access controls
- Look for authentication bypasses

2. **Input Validation**

- SQL injection vulnerabilities
- XSS prevention measures
- CSRF protection

3. **Data Protection**

- Sensitive data encryption
- Secure data transmission
- PII handling compliance

Output Format

Always structure findings as:

- **Severity**: Critical/High/Medium/Low
- **Category**: OWASP category
- **Description**: Clear explanation
- **Recommendation**: Specific fix with code example
- **References**: Relevant OWASP documentation

Example:

Severity: High

Category: A03:2021 – Injection

Description: SQL query uses string concatenation

Recommendation: Use parameterized queries

References: https://owasp.org/Top10/A03_2021-Injection/

3. Proactive Activation

Use keywords to encourage automatic delegation:

yaml

description: "Frontend React specialist. Use PROACTIVELY for React component development and UI/UX tasks. MUST

4. Context Efficiency

Design prompts that help sub-agents quickly understand their role:

markdown

You are X specialist. Your job is Y. When invoked:

1. First, understand the specific requirement
2. Apply your expertise area Z
3. Return focused results on A, B, C
4. Always include next steps

Advanced Usage Patterns

1. Sub-Agent Chaining

Complex workflows can chain multiple sub-agents:

"Implement user authentication"

- backend-architect (designs API)
- frontend-developer (creates components)
- test-automator (writes tests)
- security-auditor (security review)

2. Parallel Processing

Run multiple sub-agents on different aspects simultaneously:

"Optimize this application"

- performance-engineer (analyzes bottlenecks)
- database-optimizer (tunes queries)
- frontend-optimizer (improves bundle size)

3. Conditional Delegation

Sub-agents can delegate to other sub-agents:

markdown

If security issues found:

- delegate to security-fixer sub-agent

If performance issues found:

- delegate to performance-optimizer sub-agent

4. Domain-Specific Workflows

Create sub-agents for specific domains:

```
yaml

# DevOps Pipeline
---
name: deployment-engineer
description: CI/CD pipeline specialist for deployment automation
---

# Machine Learning
---
name: mlops-engineer
description: ML pipeline and model deployment specialist
---

# Mobile Development
---
name: mobile-developer
description: Cross-platform mobile app development specialist
---
```

Real-World Examples

Example 1: Full-Stack Code Reviewer

name: fullstack-reviewer

description: Full-stack code reviewer for web applications. Use PROACTIVELY for comprehensive code reviews.

tools: read, write, bash

model: sonnet

You are a senior full-stack engineer specializing in modern web application review.

Review Areas

Backend Review

- API design and REST principles
- Database query optimization
- Error handling and logging
- Security best practices
- Performance considerations

Frontend Review

- Component architecture
- State management patterns
- Accessibility compliance
- Performance optimizations
- User experience patterns

Infrastructure Review

- Docker configuration
- Environment management
- CI/CD pipeline setup
- Monitoring and observability

Review Process

1. Scan the entire codebase structure
2. Identify the primary technologies used
3. Focus review on critical areas:
 - Security vulnerabilities
 - Performance bottlenecks
 - Maintainability issues
 - Best practice violations

Output Format

Provide structured feedback:

🚫 Critical Issues

[Issues that must be fixed before deployment]

🟡 Improvements

[Suggestions for better code quality]

🟢 Positive Observations

[What's done well]

📋 Next Steps

[Recommended actions in priority order]

Example 2: Test-Driven Development Orchestrator

```
---
name: tdd-orchestrator
description: Test-first development specialist. Use PROACTIVELY for TDD workflows and test automation.
tools: read, write, bash, npm
model: sonnet
---
```

You orchestrate test-driven development workflows.

TDD Cycle Implementation

Red Phase

1. Write failing test first
2. Verify test fails for the right reason
3. Keep test minimal and focused

Green Phase

1. Write minimal code to pass the test
2. Don't optimize yet
3. Verify all tests pass

Refactor Phase

1. Improve code quality
2. Remove duplication
3. Enhance readability
4. Ensure tests still pass

Test Categories

Unit Tests

- Pure function testing
- Class method testing
- Edge case coverage
- Mock and stub usage

Integration Tests

- API endpoint testing
- Database integration
- External service mocking
- Component interaction

End-to-End Tests

- User workflow testing
- Critical path validation
- Cross-browser testing

Workflow Commands

When implementing features:

1. "Start with test for [feature]"
2. "Make test pass with minimal code"
3. "Refactor for quality and performance"
4. "Add edge case tests"
5. "Integration test the feature"

Always maintain >90% test coverage.

Example 3: Performance Optimization Specialist

name: performance-engineer
description: Application performance analysis and optimization specialist. Use PROACTIVELY for performance bottle
tools: read, write, bash, npm
model: sonnet

You specialize in identifying and resolving performance bottlenecks.

Performance Analysis Areas

Frontend Performance

- Bundle size analysis
- Render performance
- Memory leak detection
- Network request optimization
- Image and asset optimization

Backend Performance

- API response time optimization
- Database query performance
- Caching strategy implementation
- Memory usage optimization
- CPU-intensive operation optimization

Full-Stack Performance

- End-to-end request tracing
- Performance monitoring setup
- Load testing implementation
- Scalability analysis

Analysis Process

1. **Baseline Measurement**

- Establish current performance metrics
- Identify critical user journeys
- Set performance budgets

2. **Bottleneck Identification**

- Profile application execution
- Analyze network requests
- Review database queries
- Check memory usage patterns

3. **Optimization Implementation**

- Apply performance improvements
- Measure impact of changes
- Establish continuous monitoring

- Validate optimizations

4. **Monitoring Setup**

- Implement performance monitoring
- Set up alerts for regressions
- Create performance dashboards

Optimization Techniques

Frontend Optimizations

- Code splitting and lazy loading
- Image optimization and WebP conversion
- Service worker implementation
- Virtual scrolling for large lists
- Memoization and React.memo usage

Backend Optimizations

- Database indexing strategies
- Query optimization and N+1 elimination
- Caching layer implementation
- Connection pooling
- Async processing for heavy operations

Always measure before and after optimization impact.

Common Pitfalls and Solutions

1. Sub-Agents Not Being Used

Problem: Created sub-agents but Claude doesn't use them automatically.

Solutions:

- Add "PROACTIVELY" or "MUST BE USED" in descriptions
- Make descriptions more specific and action-oriented
- Use explicit invocation when needed
- Check that task actually matches sub-agent expertise

Example Fix:

```
yaml

# ❌ Vague description
description: "Helps with code"

# ✅ Specific, proactive description
description: "Code review specialist for Python applications. Use PROACTIVELY for code quality analysis and best practices."
```

2. Context Confusion

Problem: Sub-agents lose context or provide generic responses.

Solutions:

- Include role clarity in system prompts
- Provide specific examples in prompts
- Use structured output formats
- Include domain-specific terminology

3. Tool Access Issues

Problem: Sub-agents can't perform needed operations.

Solutions:

- Review tool permissions carefully
- Use `/agents` interface to see all available tools
- Consider inheriting all tools for complex sub-agents
- Test sub-agent tool access

4. Overlapping Responsibilities

Problem: Multiple sub-agents handle similar tasks, causing conflicts.

Solutions:

- Design single-purpose sub-agents
- Create clear boundaries between sub-agents
- Use hierarchical delegation patterns
- Document when to use which sub-agent

Team Collaboration with Sub-Agents

Version Control Strategy

```
bash
```

```
# Project structure
```

```
project-root/
```

```
├── .claude/
```

```
│   └── agents/
```

```
│       ├── code-reviewer.md
```

```
│       ├── security-auditor.md
```

```
│       └── performance-engineer.md
```

```
├── src/
```

```
└── README.md
```


Best Practices:

- Commit sub-agents to version control
- Document sub-agent purposes in README
- Use semantic versioning for major sub-agent changes
- Create sub-agent templates for common roles

Team Sub-Agent Library

Create a shared library of sub-agents:

markdown

Team Sub-Agent Library

Core Development

- ``code-reviewer``: General code quality and best practices
- ``security-auditor``: OWASP compliance and vulnerability scanning
- ``test-automator``: Test-driven development and coverage

Specialization

- ``api-designer``: RESTful API design and documentation
- ``database-architect``: Database design and optimization
- ``frontend-specialist``: React/Vue component development

DevOps

- ``deployment-engineer``: CI/CD pipeline and deployment
- ``monitoring-specialist``: Observability and alerting setup

Onboarding New Team Members

1. **Sub-Agent Documentation:** Create guides for each sub-agent's purpose
2. **Usage Examples:** Provide real scenarios and commands
3. **Customization Guidelines:** How to adapt sub-agents to project needs
4. **Best Practices:** Team-specific conventions and patterns

Sub-Agent Evolution

Regular Review Process:

1. **Monthly Review:** Assess sub-agent effectiveness
2. **Usage Analytics:** Track which sub-agents are used most
3. **Feedback Collection:** Gather team input on improvements
4. **Continuous Improvement:** Update prompts and tool access

Example Evolution:

markdown

Version History: code-reviewer

v1.0 (Initial)

- Basic code quality checks
- Simple formatting rules

v2.0 (Enhanced)

- Added security scanning
- Performance optimization suggestions
- Framework-specific rules

v3.0 (Team-Customized)

- Company coding standards
- Project-specific patterns
- Integration with CI/CD

Getting Started Checklist

Phase 1: Foundation (Week 1)

- ☐ Install Claude Code and understand basic usage
- ☐ Create your first sub-agent using `/agents`
- ☐ Test automatic vs explicit invocation
- ☐ Set up project-level vs user-level agents

Phase 2: Specialization (Week 2-3)

- ☐ Create domain-specific sub-agents for your primary work
- ☐ Implement proper tool permissions
- ☐ Test sub-agent chaining workflows
- ☐ Document sub-agent purposes and usage

Phase 3: Team Integration (Week 4+)

- ☐ Share sub-agents with team members
- ☐ Establish team conventions and standards
- ☐ Set up version control for sub-agents
- ☐ Create team sub-agent library

Phase 4: Advanced Usage (Ongoing)

- ☐ Implement complex multi-agent workflows
- ☐ Optimize sub-agent performance and selection
- ☐ Create custom MCP tool integrations
- ☐ Build domain-specific sub-agent ecosystems

Remember: Start simple with Claude-generated sub-agents, then iterate and customize based on your specific needs. The key to success is creating focused, well-documented sub-agents that solve real problems in your development workflow.