Lab: Building Custom Subagents

Module: T3 - Claude Code Advanced
Duration: 30 minutes
Difficulty: Intermediate

Overview

In this lab, you will create custom subagents that act as specialized AI assistants within Claude Code. Subagents are separate Claude sessions with defined roles, limited tools, and specific expertise.

Objectives

After completing this lab, you will be able to:

- Understand the difference between main session and subagents
- Create custom subagents with defined roles and tool restrictions
- Use built-in subagents (Explore, Plan) effectively
- Invoke subagents from the main session

Prerequisites

- Claude Code installed and authenticated
- Completed Lab T3: Custom Commands (recommended)
- Understanding of CLAUDE.md configuration

Scenario

Your team needs specialized AI assistants for: - Security auditing before deployments - Documentation verification - Dependency analysis

You'll build subagents for each role, each with appropriate tool access.

Exercise 1: Explore Built-in Subagents

Duration: 8 minutes

Task 1: List Available Agents

1. Start Claude Code in your sample project:

`bash cd labs/sample-project claude`

1. List the built-in agents:

```

> /agents ```

**Expected output:** You'll see agents like: - `explore` - Fast, read-only codebase exploration (uses Haiku) - `plan` - Research during planning mode (uses Sonnet)

## Task 2: Use the Explore Agent

1. Invoke the explore agent with `@`:

```

> @explore what testing framework does this project use? ```

1. Observe the behavior:
2. Notice the speed (Haiku is fast)
3. Notice it only READS files - no modifications

4. Notice the scoped tool access

5. Try a more complex exploration:

```

> @explore trace the request flow from POST /api/users to database ```

## Task 3: Compare Main vs Subagent

1. In the main session, ask:
```

~~~

> What testing framework does this project use? ```

1. Compare:
2. Main session has full context and tools
3. Subagent starts fresh, focused on the question
4. Subagent may use different model (faster/cheaper)

Validation Checkpoint

- [ ] You can list available agents with `/agents`
- [ ] You can invoke agents with `@agentname`
- [ ] You understand the difference between main session and subagent

---

Exercise 2: Create a Security Auditor Subagent

Duration: 10 minutes

Task 1: Set Up Agents Directory

1. Create the agents directory:

`bash mkdir -p .claude/agents`

Task 2: Create the Security Auditor

1. Create `.claude/agents/security-auditor.md`:

```markdown

---

name: security-auditor description: Security-focused code reviewer that checks for vulnerabilities tools: Read, Grep, Glob, Bash(npm audit:*) model: sonnet

---

You are a security auditor examining code for vulnerabilities.

## Your Mission

When invoked, perform a comprehensive security audit:
~~~

### 1. Dependency Vulnerabilities Run `npm audit` (or equivalent) to check for known vulnerabilities in dependencies.

### 2. Hardcoded Secrets Search for patterns that might indicate hardcoded credentials: - API keys, tokens, secrets - Passwords in config files - Private keys

### 3. Injection Vulnerabilities Check for: - SQL injection (string concatenation in queries) - Command injection (unsanitized input in exec/spawn) - XSS (unescaped output in templates)

### 4. Input Validation Verify that all user inputs are validated: - Request body validation - Query parameter sanitization - File upload restrictions

## Report Format

Categorize findings by severity:

□**CRITICAL** - Must fix before deploy - Finding and location - Exploitation risk - Recommended fix

□**WARNING** - Should fix soon - Finding and location - Potential impact - Suggested remediation

□**INFO** - Best practice suggestions - Improvement opportunity - Why it matters

Be thorough but concise. Prioritize actionable findings. ```

### Task 3: Test Your Security Auditor

1. Restart Claude to load the new agent:

```

> /quit ```

`bash claude`

1. Verify it appears in the agent list:

```

> /agents ```

1. Invoke the security auditor:

```

> @security-auditor audit this project ```

1. Review the findings:
2. Did it run `npm audit`?
3. Did it search for hardcoded secrets?
4. Did it find the missing input validation in users route?

Task 4: Understand Tool Restrictions

Notice the `tools` field in the frontmatter:

```
tools: Read, Grep, Glob, Bash(npm audit:*)
~~~

This means: - □Can read files (Read) - □Can search with Grep, Glob - □Can run `npm audit` specifically - × Cannot write or edit files - × Cannot run arbitrary commands

**Why this matters:** Least-privilege security. The auditor can inspect but not modify.

### Validation Checkpoint

- [ ] Security auditor agent is created and loads
- [ ] Agent runs `npm audit` successfully
- [ ] Agent searches for security issues
- [ ] Agent cannot modify files (test by asking it to fix something)

---

## Exercise 3: Create a Documentation Checker Subagent

**Duration:** 8 minutes

### Task 1: Create the Documentation Checker

1. Create `.claude/agents/doc-checker.md`:

```markdown
~~~

---

name: doc-checker description: Verifies documentation quality and completeness tools: Read, Glob, Grep model: haiku

---

You are a documentation quality checker.

## Your Mission

Analyze the project's documentation and report on:

### 1. README Quality - Does README.md exist? - Does it explain what the project does? - Are installation instructions provided? - Is there usage documentation?

### 2. Code Documentation - Are functions documented with JSDoc/docstrings? - Are complex algorithms explained? - Are configuration options documented?

### 3. API Documentation - Are endpoints documented? - Are request/response formats shown? - Are error codes explained?

### 4. Broken References - Do linked files exist? - Are code examples accurate?

## Report Format

□**Well Documented** - What's good about the docs

□**Needs Improvement** - What's missing or outdated

□**Missing** - Critical documentation gaps

Provide specific, actionable recommendations. ```

### Task 2: Test the Documentation Checker

1. Restart Claude and invoke:

```

> @doc-checker review project documentation ```

1. Review the analysis:
2. Did it find the README.md?
3. Did it check for JSDoc comments?
```

4. What improvements did it suggest?

Why Use Haiku?

Note `model: haiku` in the config. Haiku is: - **10x cheaper** than Opus - **3x faster** response time - **Sufficient** for documentation scanning tasks

Match the model to the task complexity!

Exercise 4: Create a Dependency Analyzer Subagent

Duration: 5 minutes

Task 1: Create the Dependency Analyzer

1. Create `.claude/agents/dep-analyzer.md`:

```markdown

---

name: dep-analyzer description: Analyzes project dependencies for health and optimization tools: Read, Bash(npm outdated:*, npm ls:*, npm view:*) model: sonnet

---

You are a dependency health analyst.

## Your Mission

When invoked, analyze the project's dependencies:

### 1. Outdated Packages Run `npm outdated` and categorize: - Patch updates (safe to update) - Minor updates (likely safe, review changelog) - Major updates (breaking changes possible)

### 2. Unused Dependencies Check if all declared dependencies are actually imported in the code.

### 3. Bundle Size Impact For web projects, identify large dependencies that might affect performance.

### 4. Security Status Cross-reference with `npm audit` results if security issues exist.
```

Report Format

□**Package Health Summary** - Total dependencies: X - Up to date: X - Outdated: X - Security issues: X

□**Recommended Updates** Table of packages to update with risk level.

□**Potentially Unused** Dependencies that may be removable.

⚠ **Large Dependencies** Packages contributing significantly to bundle size. ```

Task 2: Quick Test

1. Invoke the analyzer:

```

> @dep-analyzer check dependencies ```

1. Did it identify outdated packages?

---

## Exercise 5: Subagent Composition

**Duration:** 4 minutes

### Using Subagents in Sequence

Subagents report back to your main session. You can chain them:

```
 > @doc-checker review this project

[reviews, reports findings]

> Now @security-auditor check for the issues mentioned above

[audits with context from previous discussion]
```
```

Subagent vs Main Session Decision Guide

Use Subagent When...	Use Main Session When...
Task is focused and isolated	Task requires full context
You want cheaper/faster model	You need maximum capability
Read-only inspection	Modifications needed
Parallel conceptual work	Sequential reasoning required

Challenge: Create Your Own Specialist

Duration: Bonus

Design a subagent for your actual work. Ideas:

1. **Test Coverage Auditor** - Checks which functions lack tests
2. **Performance Analyzer** - Looks for inefficient patterns
3. **Accessibility Checker** - Reviews UI code for a11y issues
4. **API Contract Validator** - Ensures API matches OpenAPI spec
5. **Migration Planner -** Analyzes code for framework upgrades

Template:

```
 ---
name: your-agent-name
description: What it does (shown in /agents)
tools: Read, Grep, Glob  # Add only what's needed
model: haiku|sonnet|opus  # Match complexity
---

[Agent instructions - be specific about what to do and how to report]
```

Troubleshooting Guide

Issue	Solution
Agent doesn't appear in `/agents`	Check file location: `.claude/agents/name.md`
Agent can't run expected command	Verify the `tools` field includes the command
Agent seems slow	Consider using `model: haiku` for simpler tasks
Agent hallucinates capabilities	Be explicit in instructions about what it can/cannot do
Agent won't modify files	By design - add Write/Edit to tools if needed (carefully!)

Summary

In this lab, you learned to:

1. **Use built-in subagents** like `explore` and `plan`
2. **Create specialized subagents** with focused roles
3. **Restrict tools** for least-privilege security
4. **Choose appropriate models** for cost/performance balance
5. **Compose subagent outputs** in your workflow

Key Takeaways

- Subagents are **separate sessions** with defined roles
- Use **tool restrictions** to limit what agents can do
- Choose **cheaper models** (Haiku) for simple tasks
- Subagents **report back** to main session for coordination

The Power of Specialization

Instead of one AI that does everything, you now have: - A security expert that only audits - A documentation expert that only reviews - A dependency expert that only analyzes

Each is better at their job because of focus.

Additional Resources

- Subagent Examples: github.com/VoltAgent/awesome-claude-code-subagents
- Claude Code Agents Docs: docs.anthropic.com/claude-code/agents