CLAUDE COWORK

Plugin Management Handbook

The Complete Guide to Creating, Customizing, and Managing Claude Cowork Plugins

VERSION	`1.0.0`
PLUGIN	`cowork-plugin-management`

RationalEyes.ai
contact@rationaleyes.ai

February 2026

Table of Contents

The Complete Guide to Creating, Customizing, and Managing Claude Cowork Plugins

---

Plugin Name: cowork-plugin-management **Version:** 1.0.0 **Author:** Anthropic **Description:** Create, customize, and manage plugins tailored to your organization's tools and workflows. Configure MCP servers, adjust plugin behavior, and adapt templates to match how your team works.

Part I: Overview & Quick Start

What Is the Plugin Management Plugin?

The cowork-plugin-management plugin is the meta-plugin in the Claude Cowork ecosystem. While every other plugin gives Claude domain-specific capabilities — marketing expertise, legal analysis, financial modeling — the plugin management plugin gives Claude the ability to build and customize those plugins themselves. It is the tool that builds other tools.

Think of it this way: if the marketing plugin is a specialized appliance in your kitchen, the plugin management plugin is the workshop where you design and assemble new appliances. It encodes the complete plugin architecture, the rules for how every component works, and guided workflows for creating or adapting plugins to your exact needs.

This plugin operates through two core skills that activate automatically during conversation. There are no slash commands to memorize. You simply describe what you want to build or customize, and Claude walks you through the process step by step, asking clarifying questions along the way and delivering a finished, installable plugin file at the end.

Business Problems Solved

For Plugin Administrators and IT Teams:

- **Standardizing workflows across the organization**: Package your company's best practices, terminology, and processes into plugins that every team member uses. No more inconsistent approaches or tribal knowledge.
- **Connecting your specific tool stack**: Generic plugins reference tool categories. You need those categories replaced with your actual tools — Asana instead of "project tracker," Slack instead of "chat." The customizer skill handles this automatically.
- **Version-controlled distribution**: Package plugins as `.plugin` files that can be tested, version-controlled, and distributed to dozens or hundreds of users at once.

For Team Leads and Managers:

- **Codifying institutional knowledge**: Turn your team's processes, checklists, and decision frameworks into skills that Claude applies automatically. When a new team member joins, they get the benefit of years of accumulated knowledge through the plugin.
- **Reducing onboarding time**: Instead of documenting every process in wikis that nobody reads, embed them in plugins where they surface naturally during work.

- **Maintaining consistency**: When every team member uses the same plugin with your workflows embedded, output quality becomes more predictable.

For Enterprise IT Managers:

- **Customizing plugins for your organization**: Adapt generic plugins to reference your specific tools, project names, workspace IDs, and team conventions without building anything from scratch.
- **Managing plugin lifecycles:** Create, version, distribute, and update plugins across multiple departments with a structured process.
- **Enforcing governance:** Build plugins with hooks that automatically enforce policies — compliance checks, naming conventions, security reviews — without relying on people to remember the rules.

For Power Users:

- **Packaging custom workflows:** Turn your personal productivity patterns into formal plugins that colleagues can install with one click.
- **Sharing expertise**: If you have built effective prompts and reference documents for a particular domain, package them as a plugin so others benefit.
- **Prototyping quickly:** Test new plugin ideas through a guided conversation, get a working prototype in minutes, and iterate from there.

Target Users

This plugin is designed for:

- **Plugin Administrators** who manage Claude deployments and maintain plugin libraries for their organization
- **Team Leads** who want to codify their team's processes and best practices into reusable, shareable tools
- **Enterprise IT Managers** who oversee Claude rollouts across multiple departments and need to customize plugins for their tool stack
- **Power Users** who have discovered effective workflows and want to standardize and share them as formal plugins
- **Solutions Architects** who design Claude-based solutions for specific business needs and use cases
- **Operations Managers** who want to embed process compliance and quality checks into automated workflows

Quick Start Walkthrough

Step 1: Install the Plugin

Cowork (recommended): Open **Plugin Settings** in the Cowork desktop app, find **Cowork Plugin Management**, and click **Install**. The plugin activates immediately — no CLI required.

Claude Code CLI (alternative): If you are using Claude Code in the terminal, install via:

```
claude plugins add knowledge-work-plugins/cowork-plugin-management
```

> **Note:** *All standard Cowork plugins, including Plugin Management, are available from Plugin Settings with a single click. The CLI command above is only needed for Claude Code terminal users. Be aware that the plugin creation and customization skills have limited functionality outside the Cowork desktop app, as they require access to mounted plugin directories and the outputs directory for delivering `.plugin` files.*

Step 2: Verify Installation

Ask Claude:

```
What plugins are installed?
```

You should see `cowork-plugin-management` in the list.

Step 3: Create Your First Plugin

Since this plugin has no slash commands, you interact with it through natural conversation. Try saying:

```
I want to create a plugin for our team's weekly reporting process
```

Claude will activate the `create-cowork-plugin` skill and begin the guided discovery phase. It will ask you:

- What the plugin should do and what problem it solves
- Who will use it and in what context
- Whether it needs to connect to any external tools
- Whether there is a similar workflow or plugin to reference

Answer these questions, and Claude will walk you through planning, designing, building, and packaging your plugin.

Step 4: Or Customize an Existing Plugin

If you have an installed plugin that needs to be adapted to your organization's tools, say:

```
I want to customize the marketing plugin for our company
```

Claude will activate the `cowork-plugin-customizer` skill. It will search your connected knowledge sources (Slack, documents, email) to learn what tools your organization uses, then systematically replace generic references with your actual tool names and configurations.

Step 5: Install Your New Plugin

Both skills deliver a `.plugin` file at the end of the workflow. This file appears in the chat as a rich preview where you can browse the plugin's files and accept it with a button click. Once accepted, the plugin is installed and ready to use.

How This Plugin Differs from Domain Plugins

Most Cowork plugins add domain knowledge to Claude. The marketing plugin knows how to plan campaigns. The legal plugin knows how to review contracts. The data plugin knows how to analyze datasets.

The plugin management plugin adds *architectural* knowledge. It knows how plugins themselves work — what components they contain, how to structure them correctly, how to write skill descriptions that trigger reliably, how to configure MCP server connections, and how to package everything into a distributable file.

This difference manifests in several ways:

CHARACTERISTIC	DOMAIN PLUGINS (E.G., MARKETING)	PLUGIN MANAGEMENT
Commands	Multiple slash commands (e.g., `/campaign-plan`)	None
Interaction model	Command-driven: invoke, provide inputs, get output	Conversational: describe intent, answer questions, iterate
Skills	Domain expertise that loads based on topic	Architectural knowledge that loads when building or customizing plugins
MCP servers	Connects to domain tools (HubSpot, Amplitude)	None of its own (but configures them for other plugins)
Output	Content, reports, analysis	Plugin files (`.plugin` packages)

CHARACTERISTIC	DOMAIN PLUGINS (E.G., MARKETING)	PLUGIN MANAGEMENT
Session length	Typically one command, one result	Multi-phase workflow spanning many exchanges

Understanding this difference is important because it sets expectations for how you interact with the plugin. You do not run a command and step back. You engage in a conversation that may span 10-20 exchanges as you and Claude work through discovery, planning, design, and implementation together.

The Two Core Skills

The plugin contains two skills, each supporting a distinct workflow:

create-cowork-plugin guides you through building a new plugin from scratch. It encodes the complete plugin architecture, component schemas, and a five-phase creation process that moves from discovery to implementation to packaging. Use this skill when you are starting from zero — when no existing plugin covers your need, or when you want full control over every component and design decision.

cowork-plugin-customizer helps you adapt an existing plugin to your organization's specific tools and processes. It searches for customization points marked with `~~` placeholders in plugin files, gathers context from your connected knowledge sources, and systematically replaces generic references with your actual tool names and configuration values. Use this skill when you have a generic plugin template that needs to be personalized for your organization.

Both skills require the Cowork desktop app environment. They need access to the outputs directory for delivering `.plugin` files and to mounted plugin directories for reading existing plugins. They will not function in remote containers or CLI-only sessions.

How Plugin Management Fits Into the Ecosystem

Understanding where this plugin sits in the broader lifecycle helps explain why it exists and when to use it:

```
Plugin Development Lifecycle
+------------------------------------------------------------+
|                                                            |
|  Path 1: Build from Scratch                                |
|                                                            |
|  Identified need                                           |
|         |                                                  |
|  [create-cowork-plugin skill]                              |
|         |                                                  |
|  Custom plugin (built from scratch)                        |
|         |                                                  |
|  Testing and iteration                                     |
|         |                                                  |
|  Distribution as .plugin file                              |
|                                                            |
+------------------------------------------------------------+
|                                                            |
|  Path 2: Customize an Existing Template                    |
|                                                            |
|  Generic plugin template (with ~~ placeholders)            |
|         |                                                  |
|  [cowork-plugin-customizer skill]                          |
|         |                                                  |
|  Organization-specific plugin (your tools configured)      |
|         |                                                  |
|  Distribution as .plugin file                              |
|         |                                                  |
|  End users install and use                                 |
|                                                            |
+------------------------------------------------------------+
```

This plugin enables both paths. It sits at the beginning of the plugin lifecycle, enabling everything downstream.

Why Build Plugins Instead of Writing Good Prompts?

Investing time in creating or customizing plugins offers several advantages over ad-hoc prompting:

Reusability. A well-crafted plugin packages workflows, knowledge, and tool integrations so they can be used repeatedly across projects and team members without reinventing the wheel each time.

Standardization. Plugins encode best practices and institutional knowledge. When your entire marketing team uses the same marketing plugin with your brand voice embedded, content quality becomes more consistent.

Discoverability. Skills trigger automatically when relevant topics come up. Users do not need to remember complex prompts or specialized terminology — they just work naturally, and Claude brings in the right knowledge.

Tool integration. MCP server configurations in plugins mean users do not manually set up connections to Asana, HubSpot, Figma, and other services. The plugin carries those configurations, and connections happen seamlessly.

Version control and distribution. Plugins package everything — knowledge, commands, integrations — into a single `.plugin` file that can be version-controlled, tested, and distributed at scale.

Part II: Commands Reference

Why This Plugin Has No Commands

The cowork-plugin-management plugin is unusual among Cowork plugins: it contains **zero slash commands**. There is no `/create-plugin` or `/customize` command. This is a deliberate architectural decision, not an oversight.

Understanding why requires understanding the difference between commands and skills in the Claude plugin architecture.

Commands are user-initiated, explicit actions. You type `/campaign-plan` and Claude executes the workflow defined in that command file. Commands are best for operations with clear start and end points, predictable inputs, and structured outputs. They work well when you know exactly what you want to do.

Skills are context-activated knowledge packages. Claude automatically loads a skill when the conversation touches on relevant topics. Skills are best for providing reference knowledge, frameworks, and domain expertise that may be needed at various points in a conversation.

Plugin creation and customization are inherently conversational, multi-phase processes. They involve:

- Discovery conversations to understand what you want to build
- Clarifying questions that depend on your previous answers
- Decision points where you may change direction
- Iterative refinement of component designs
- Delivery of a final artifact after all decisions are made

Forcing this into a slash command structure would be awkward. You would either need to provide all inputs upfront in a single command invocation (impractical for complex plugin designs) or the command would need to engage in extended back-and-forth dialogue (which blurs the line between command and conversation).

Instead, the plugin management plugin uses **conversational skills** that activate when you express intent:

- "I want to create a plugin for pharmaceutical marketing" triggers the `create-cowork-plugin` skill

- "I need to customize this plugin for my team's tools" triggers the `cowork-plugin-customizer` skill

This design leverages Claude's natural language understanding. You do not need to remember command syntax or argument order. You describe what you want in plain language, and the appropriate skill engages.

How to Invoke Plugin Management Capabilities

Since there are no commands, you interact with this plugin through natural language expressions of intent.

To create a new plugin:

- "I want to create a plugin"
- "Help me build a plugin for compliance workflows"
- "Let's make a plugin from scratch"
- "I need to design a plugin for my team"
- "Can we scaffold a new plugin?"
- "Start a plugin from scratch for onboarding"

To customize an existing plugin:

- "Customize this plugin for my organization"
- "I need to personalize the marketing plugin for our tools"
- "Let's adapt this plugin to our team's workflow"
- "Replace the tool placeholders in this plugin"
- "Configure the engineering plugin for our company"

The skills' trigger descriptions encode these phrases, ensuring reliable activation when you use similar language.

Comparison to Command-Driven Plugins

To illustrate the difference, consider the marketing plugin, which has both commands and skills:

Marketing Plugin Commands (explicit invocation):

COMMAND	PURPOSE
`/campaign-plan`	Generate a campaign brief
`/draft-content`	Create marketing content

COMMAND	PURPOSE
`/brand-review`	Review content for brand alignment
`/competitive-brief`	Create competitive analysis
`/performance-report`	Generate performance report
`/seo-audit`	Run an SEO audit
`/email-sequence`	Design email sequences

Marketing Plugin Skills (automatic activation):

SKILL	ACTIVATES WHEN DISCUSSING
`brand-voice`	Brand guidelines or tone
`campaign-planning`	Campaign strategy
`competitive-analysis`	Competitors
`content-creation`	Content templates or best practices
`performance-analytics`	Metrics or results

The plugin management plugin is **all skills, no commands**, because its entire function is knowledge-intensive and conversational rather than action-oriented. You do not invoke a quick action and get structured output. You engage in a guided process that unfolds over multiple exchanges.

The Implication for Users

If you are accustomed to slash commands as the primary way to interact with plugins, this plugin requires a mindset shift. You do not invoke a command and step back; you engage in a conversation. The plugin asks questions, you provide answers, it makes recommendations, you confirm or adjust, and the process iterates until a plugin is complete.

This conversational model is more flexible than command-based workflows but requires more active participation. You cannot type `/create-plugin marketing` and expect a finished plugin to appear. You must engage in the discovery, planning, and design phases — and that engagement is what makes the resulting plugin genuinely tailored to your needs rather than a generic template.

The benefit is significant: the plugin you get at the end reflects your actual requirements, tool stack, and workflows, not a one-size-fits-all approximation.

Part III: Skills Deep Dive

This section is the heart of the handbook, as skills are the entire interface of the plugin management plugin. We examine both skills in detail, including their trigger mechanisms, internal workflows, design principles, and practical guidance for using them effectively.

Skill 1: create-cowork-plugin

What It Does

The create-cowork-plugin skill guides you through building a new plugin from scratch. It encodes:

- The complete plugin architecture (directory structure, component types, file formats)
- Component schemas for every plugin element (commands, skills, agents, hooks, MCP servers)
- A five-phase creation workflow (discovery, planning, design, implementation, packaging)
- Best practices for each component type
- Reference examples at three complexity levels (minimal, standard, full-featured)

When this skill activates, you are not just getting documentation about how plugins work. You are getting a structured process that walks you through creating one, asking the right questions at each stage, and delivering an installable plugin file at the end.

When to Use It

Use this skill when:

- You have a clear need that no existing plugin addresses
- You want to package institutional knowledge or team workflows into a reusable plugin
- You are building a domain-specific plugin for a narrow use case (e.g., regulatory compliance for medical devices, customer onboarding for your SaaS product)
- You want full control over every component and design decision

Do not use this skill when:

- An existing plugin already does 80% of what you need — customize or extend it instead (see Part V)
- You are unsure what you want to build — start with a general conversation to clarify your needs, then engage the skill

- You only need documentation about how plugins work — this handbook and the file reference in Part VI serve that purpose

How It Triggers

The skill triggers when you express intent to create a plugin. The skill description encodes these activation phrases:

> *"Use when users want to create a plugin, build a plugin, make a new plugin, develop a plugin, scaffold a plugin, start a plugin from scratch, or design a plugin."*

Any of these expressions in conversation will activate the skill. The skill also detects incompatible environments — if you try to trigger it in a remote container or CLI-only session, it will explain that plugin creation requires the Cowork desktop app.

The Five-Phase Workflow

Phase 1: Discovery

The skill's goal in this phase is to understand what you want to build and why. It asks clarifying questions only when your initial request leaves gaps:

- What should this plugin do? What problem does it solve?
- Who will use it and in what context?
- Does it integrate with any external tools or services?
- Is there a similar plugin or workflow to reference?

The skill summarizes its understanding and confirms before proceeding. This prevents building the wrong thing and ensures alignment on scope.

Example discovery exchange:

You say: "I want to build a plugin that helps our QA team manage test plans and track test results."

Claude asks: "A few questions to make sure I understand the scope: (1) Should this plugin connect to any tools your QA team already uses, like TestRail, Zephyr, or a project tracker? (2) Are test plans standardized at your company, or does each team have their own format? (3) Should the plugin generate test plans from requirements, track execution results, or both?"

You answer, and the skill confirms: "So we are building a QA plugin that generates test plans from user stories in Linear, tracks execution results, and produces summary reports. The team uses a standardized test plan format with sections for prerequisites, test steps, expected results, and pass/fail criteria. Does that capture it correctly?"

Output: A clear statement of plugin purpose and scope.

Phase 2: Component Planning

Based on what was learned in discovery, the skill determines which component types the plugin needs. It evaluates each type against specific decision criteria:

COMPONENT	DECISION CRITERIA
Skills	Does the plugin require specialized knowledge that Claude should load on demand? (domain expertise, reference schemas, workflow guides)
Commands	Are there user-initiated actions with clear inputs and outputs? (generate, analyze, review, configure)
MCP Servers	Does the plugin need to read from or write to external services? (databases, APIs, SaaS tools)
Agents	Are there autonomous multi-step tasks that benefit from dedicated focus? (validation, analysis, generation)
Hooks	Should certain behaviors happen automatically on specific events? (enforce policies, load context, validate operations)

The skill presents a component plan and asks for your confirmation:

```
│ Component │ Count │ Purpose                                   │
├───────────┼───────┼───────────────────────────────────────────┤
│ Skills    │ 1     │ QA methodology and test plan frameworks    │
│ Commands  │ 3     │ /generate-test-plan, /test-summary, /coverage-map │
│ Agents    │ 0     │ Not needed for this use case               │
│ Hooks     │ 0     │ Not needed for this use case               │
│ MCP       │ 1     │ Connect to Linear for user stories and tracking │
```

You confirm or adjust before proceeding. If you think a command should be split into two, or an agent would be useful after all, this is the time to say so.

Output: A confirmed list of components to create.

Phase 3: Design and Clarifying Questions

This is the most interactive phase. The skill asks targeted questions for each component type in the plan, presented in groups by component type.

For skills:

- What user queries should trigger this skill?
- What knowledge domains does it cover?
- Should it include reference files for detailed content?

For commands:

- What arguments does each command accept?
- What tools does each command need? (Read, Write, Bash, Grep, etc.)
- Is each command interactive (asks follow-up questions) or automated (runs to completion)?

For agents:

- Should each agent trigger proactively or only when explicitly requested?
- What tools does it need?
- What should the output format be?

For hooks:

- Which events should trigger the hook? (PreToolUse, PostToolUse, Stop, SessionStart, etc.)
- What behavior should it enforce? (validate, block, modify, add context)
- Should it be prompt-based (Claude decides) or command-based (deterministic script)?

For MCP servers:

- What server type? (stdio for local processes, SSE for cloud services with OAuth, HTTP for REST APIs)
- What authentication method?
- What tools should be exposed?

If you say "whatever you think is best," the skill provides specific recommendations based on common patterns and asks for explicit confirmation. It does not assume defaults.

Output: Detailed specifications for every component.

Phase 4: Implementation

The skill creates all plugin files following the specifications from Phase 3. It works through a defined order of operations:

1. Create the plugin directory structure
2. Create `plugin.json` manifest with the correct metadata
3. Create each component following the appropriate schema

4. Create `README.md` documenting the plugin

During implementation, the skill follows important principles:

- **Commands are instructions for Claude**, not messages to the user. Command files are written as directives ("Generate a test plan based on the user stories," "Analyze the test results and produce a summary").
- **Skills use progressive disclosure.** The SKILL.md body stays lean (under 3,000 words), with detailed content placed in `references/` subdirectories that load on demand. The frontmatter description includes specific trigger phrases for reliable activation.
- **Agents include example blocks** in their descriptions, showing concrete scenarios where the agent should be activated.
- **Hooks use portable paths.** All script paths reference `${CLAUDE_PLUGIN_ROOT}` instead of hardcoded absolute paths, ensuring the plugin works regardless of where it is installed.
- **MCP configurations use environment variables** for credentials (`${API_TOKEN}`), keeping secrets out of plugin files.

Output: All plugin files, created and ready for review.

Phase 5: Review and Package

The skill wraps up by:

1. **Summarizing** what was created — listing each component and its purpose
2. **Asking** if you want any adjustments (add a section to a skill, change a command's behavior, add another MCP connection)
3. **Validating** the plugin by running `claude plugin validate` to check for structural errors
4. **Packaging** the plugin as a `.plugin` file

The packaging process creates a zip archive and copies it to the outputs directory:

```
cd /path/to/plugin-dir && zip -r /tmp/plugin-name.plugin . -x "*.DS_Store" && \
  cp /tmp/plugin-name.plugin /path/to/outputs/plugin-name.plugin
```

The `.plugin` file appears in the chat as a rich preview where you can browse the plugin's files and accept it with a button click. Once accepted, the plugin is installed and immediately available.

Output: A finished `.plugin` file delivered to the outputs directory.

Practical Example: Building a Customer Onboarding Plugin

Here is a realistic walkthrough of the entire five-phase process:

Discovery: "I want to build a plugin that helps our customer success team onboard new enterprise clients. We have a 90-day onboarding process with milestones, and each client gets a customized implementation plan."

Claude asks about tools (Salesforce for client data, Slack for communication), team size (5 CSMs), and whether the onboarding process is documented (yes, in Notion).

Component Planning:

```
│ Component │ Count │ Purpose                                          │
├───────────┼───────┼──────────────────────────────────────────────────┤
│ Skills    │ 1     │ Onboarding methodology and milestone frameworks  │
│ Commands  │ 2     │ /onboarding-plan, /milestone-check               │
│ MCP       │ 2     │ Salesforce for client data, Notion for process docs │
```

Design: Claude asks about plan structure, milestone definitions, what data to pull from Salesforce, and what the milestone check should report on.

Implementation: Claude creates all files — the skill with onboarding frameworks, two commands with clear workflows, MCP connections, and a README.

Review and Package: Claude shows a summary, you request one tweak (add a "risk flags" section to the onboarding plan), Claude makes the change, validates, and packages the `.plugin` file.

Total time: approximately 15-25 minutes of conversation.

The Placeholder System for Distributable Plugins

By default, the skill does not use or ask about the `~~` placeholder pattern. It only introduces placeholders if you explicitly say you want people outside your organization to use the plugin.

When placeholders are needed, the skill marks tool references with `~~` (e.g., `~~project tracker`, `~~chat`) and creates a `CONNECTORS.md` file at the plugin root explaining the system. This makes the plugin tool-agnostic and ready for customization by other organizations using the `cowork-plugin-customizer` skill.

Example Plugin Structures

The skill includes three complete example plugin structures as reference implementations:

Minimal Plugin (meeting-notes): A single command, no other components. Shows the simplest possible plugin — just a `plugin.json` and one command file that generates structured meeting notes from transcripts.

Standard Plugin (code-quality): A skill for coding standards, two commands (`/review` and `/fix-lint`), and GitHub MCP integration. Demonstrates how domain knowledge, user actions, and tool connections work together.

Full-Featured Plugin (engineering-workflow): Skills, commands, an agent (ticket-analyzer), hooks (load team context on session start), MCP integration (Linear, GitHub, Slack), and tool-agnostic connectors with `CONNECTORS.md`. Shows every component type in use.

These examples serve as templates and reference implementations during the creation workflow.

Best Practices for Using This Skill

Start with a clear problem statement. The more specific you are in the discovery phase, the better the final plugin will be. "I need a plugin for marketing" is too vague. "I need a plugin that helps pharmaceutical marketing teams create compliant promotional materials and get them through MLR review" gives the skill enough context to make smart design decisions.

Do not over-build on the first iteration. Begin with the minimum viable set of components. A plugin with one well-crafted skill and two focused commands is more useful than one with five half-baked components. You can always add more in a future iteration.

Think about trigger phrases early. For skills, be explicit about what phrases users would naturally say that should activate the skill. "When should someone use this?" leads to better trigger descriptions than generic statements.

Identify tool integrations upfront. If your plugin needs to connect to external services, mention them in the discovery phase. MCP server setup requires knowing authentication methods, API endpoints, and required environment variables.

Leverage reference files for depth. If a skill requires detailed tables, code examples, regulatory checklists, or long reference guides, those belong in `references/` subdirectories, not in the main SKILL.md body. This keeps the core skill lean and context-efficient.

Common Mistakes and How the Skill Prevents Them

Mistake: Creating commands that read like documentation for the user rather than instructions for Claude. **Prevention:** The skill explicitly writes command content in imperative form, framing everything as directives Claude should follow.

Mistake: Writing skill descriptions that are too generic to trigger reliably. **Prevention:** The skill asks for specific trigger phrases and ensures they are included in the description frontmatter, using quoted examples.

Mistake: Hardcoding absolute paths in hooks or MCP configurations. **Prevention:** The skill always uses `${CLAUDE_PLUGIN_ROOT}` for intra-plugin references, ensuring portability.

Mistake: Omitting required frontmatter fields or using incorrect formats. **Prevention:** The skill validates against the component schemas before writing files, ensuring structural correctness.

Skill 2: cowork-plugin-customizer

What It Does

The cowork-plugin-customizer skill adapts generic plugin templates to your specific organization. It does this by:

- Finding all customization points marked with `~~` placeholders in plugin files
- Searching your connected knowledge sources (Slack, documents, email) to learn what tools and processes your organization uses
- Systematically replacing placeholders with actual tool names and configuration values
- Connecting MCP servers for the tools identified during customization
- Delivering a customized `.plugin` file ready for installation

This skill is the bridge between generic, tool-agnostic plugin templates (designed for broad distribution) and the specific reality of your organization's tool stack.

When to Use It

Use this skill when:

- You have a plugin template that uses `~~` placeholders for tool references
- You want to adapt a generic workflow plugin to your company's actual tools (replacing "project tracker" with Linear, "chat" with Slack)
- You need to configure organization-specific values (workspace IDs, channel names, project identifiers)
- You have knowledge sources connected (Slack, email, documents) that can provide context about your organization

Do not use this skill when:

- You are creating a plugin from scratch (use `create-cowork-plugin` instead)
- The plugin has no `~~` placeholders to customize
- You need to make structural changes like adding new skills or commands (fork the plugin instead — see Part V)

How It Triggers

The skill triggers when you express intent to customize or personalize a plugin:

- "Customize this plugin for my organization"
- "Replace tool placeholders in this plugin"
- "Personalize the marketing plugin for my team"
- "Configure this plugin for our tools"
- "Adapt this plugin to our workflow"

Like the creation skill, the customizer requires the Cowork desktop app environment. If triggered in an incompatible environment, it explains that customization requires Cowork mode with access to mounted plugin directories.

Understanding the Placeholder System

Before diving into the workflow, it helps to understand what the customizer is working with. When a plugin is designed to be shared across different organizations, its authors face a challenge: Company A uses Slack while Company B uses Microsoft Teams; Company A uses Jira while Company B uses Asana. How do you write one plugin that works for both?

The answer is the `~~` placeholder system. In plugin files, references to external tools use a `~~` prefix followed by a category name:

```
Check ~~project tracker for open tickets assigned to the user.
Post a summary to ~~chat in the team channel.
Pull conversion data from ~~product analytics.
```

These placeholders are generic. `~~project tracker` could mean Jira, Asana, Linear, Monday, or any other project management tool. The customizer skill's job is to replace these generic references with the specific tools your organization actually uses.

You never see these placeholders during the customization process. The skill presents everything in plain language: "Which project management tool does your team use?" rather than "What should replace the `~~project tracker` placeholder?"

The Four-Phase Workflow

Phase 1: Gather Context from Knowledge Sources

Before asking you any questions, the skill searches your connected knowledge sources to learn about your organization. It looks through:

1. **Chat and Slack history** — tool mentions, integration discussions, workflow conversations
2. **Documents** — onboarding guides, tool setup instructions, internal handbooks
3. **Email** — license notifications, admin setup emails, service invitations

The skill uses intelligent search strategies to find the information it needs. For example, when looking for your project management tool, it searches for terms like "Asana," "Jira," "Linear," "Monday," "sprint," and "tickets" across your connected sources. When looking for your chat platform, it searches for "Slack," "Teams," and "Discord."

Here is the category-to-keyword mapping the skill uses:

CATEGORY	SEARCH KEYWORDS
Project management	asana, jira, linear, monday, tasks
Source control	github, gitlab, bitbucket, code
Chat	slack, teams, discord
Documents	google docs, notion, confluence
Calendar	google calendar, calendar
Email	gmail, outlook, email
Design and graphics	figma, sketch, design
Analytics and BI	datadog, grafana, analytics
CRM	salesforce, hubspot, crm
Wiki and knowledge base	notion, confluence, outline, wiki
Data warehouse	bigquery, snowflake, redshift
Conversation intelligence	gong, chorus, call recording

The skill records all findings for use in Phase 3.

Phase 2: Create a Task List from Customization Points

The skill scans the plugin files to find all lines containing `~~` placeholders. It groups them by theme and creates a task list with user-friendly descriptions that focus on learning about your organization:

- "Learn how standup prep works at your company" (not "Replace placeholders in commands/standup-prep.md")

- "Find out which project management tool your team uses" (not "Determine value for ~~project tracker")
- "Understand your team's sprint process and status conventions" (not "Replace ~~sprint-status values")

The task list is presented in plain language. You never see mentions of `~~` prefixes or customization points — those are implementation details the skill handles behind the scenes.

Phase 3: Complete Task Items

For each item in the task list, the skill determines how to resolve it:

If knowledge sources provided a clear answer: The skill applies the change directly without asking for confirmation. For example, if Slack messages consistently reference "Linear" when discussing tickets and projects, the skill replaces the project tracker placeholder with "Linear" automatically.

If knowledge sources did not provide an answer: The skill asks you directly using plain language. It does not assume "industry standard" defaults. If the answer is unclear, it asks.

If you skip or say you do not know: The skill leaves the placeholder unchanged. The plugin can be customized again in a future session when more information is available.

Types of changes made during this phase:

1. **Tool name replacements**: generic "project tracker" becomes "Asana," generic "chat" becomes "Slack"
2. **URL pattern updates**: `tickets.example.com/your-team/123` becomes `app.asana.com/0/PROJECT_ID/TASK_ID`
3. **Organization-specific values**: workspace IDs, project names, team identifiers, channel names
4. **Process-specific content**: sprint duration, estimation scales, status workflow stages, naming conventions

Phase 4: Search for and Connect MCP Servers

After all customization points have been resolved, the skill connects MCP servers for the tools that were identified. For each tool:

1. **Search the MCP registry** for available connectors using category keywords
2. **Present options** if multiple connectors exist for a category
3. **Offer connection** — if the tool has an MCP server and you have not connected it yet, the skill presents a Connect button for you to authenticate
4. **Update the plugin's MCP configuration** with the correct server URLs and settings

The skill collects all MCP results and presents them together in a summary rather than one at a time. This gives you a clear picture of what was connected and what still needs attention.

> **Note:** *First-party integrations (Gmail, Google Calendar, Google Drive) are connected at the user level and do not need entries in the plugin's MCP configuration file.*

Summary Output

After customization is complete, the skill presents a summary of what was learned, grouped by source:

```
## From searching Slack
- Your team uses Linear for project management
- Sprint cycles are 2 weeks
- The engineering channel is #eng-team

## From searching documents
- Story points use T-shirt sizes (S, M, L, XL)
- Ticket statuses are: Backlog, In Progress, In Review, Done

## From your answers
- Your design tool is Figma
- Your monitoring platform is Datadog

## MCP Connections
- Linear: Connected
- Slack: Connected
- Figma: Connected
- Datadog: Not yet connected (requires API key setup)
```

If no knowledge sources were available and you had to answer questions manually, the skill adds a helpful note: "By the way, connecting sources like Slack or Microsoft Teams would let me find answers automatically next time you customize a plugin."

The skill then packages the customized plugin as a `.plugin` file and delivers it to the outputs directory for installation.

Practical Example: Customizing the Engineering Workflow Plugin

Here is a realistic walkthrough:

You say: "I want to customize the engineering-workflow plugin for our company."

Phase 1: The skill searches your Slack history and finds mentions of Linear (project management), GitHub (source control), and references to "#eng-standup" channel. It searches your documents and finds an onboarding guide mentioning Datadog for monitoring and Figma for design.

Phase 2: The skill creates a task list:

- Learn about your project management setup
- Understand your standup process
- Find out about your code review workflow
- Identify your monitoring tools

Phase 3: Based on Slack findings, the skill automatically replaces "project tracker" with "Linear" and "chat" with "Slack." It asks you: "I found references to Datadog for monitoring. Is that your primary observability platform?" You confirm. It asks about estimation — your documents did not mention this. You tell it you use story points on a Fibonacci scale.

Phase 4: The skill connects Linear, GitHub, and Slack MCP servers, and notes that Datadog requires API keys you will need to configure separately.

Delivery: A customized `.plugin` file with all generic references replaced with your actual tools, processes, and conventions.

Best Practices for Using This Skill

Connect knowledge sources first. The more sources the skill can search, the fewer questions it needs to ask you. Connect Slack, email, and document sources before running the customizer.

Run it in the Cowork desktop app. The customizer needs access to mounted plugin directories and the outputs folder. It cannot function in CLI-only or remote container environments.

Do not worry about getting everything right. If you skip a question or are not sure about an answer, the skill leaves the placeholder unchanged. You can run the customizer again later when you have the information.

Review the summary carefully. The summary shows what the skill learned from your sources. If something is wrong (it found a mention of Jira from two years ago but your team switched to Linear), let it know and it will correct the change.

Part IV: Connectors & MCP Servers

This Plugin Has No MCP Servers

The plugin management plugin does not connect to any external tools or services. It has no `.mcp.json` file and no MCP server configurations. This makes sense when you consider what the plugin does: it builds and customizes other plugins, which is an entirely local operation involving file creation and manipulation.

However, MCP is a central concept in the plugin management plugin because the plugins you *create* and *customize* often include MCP server connections. Understanding how MCP works is essential for using both skills effectively.

What Are MCP Servers?

MCP (Model Context Protocol) servers are bridges between Claude and external tools and services. They allow Claude to read data from and take actions in your existing software stack — your CRM, project tracker, design tools, analytics platform, and so on.

When a plugin includes MCP server configurations, Claude can:

- Pull real data from your tools instead of asking you to copy-paste it
- Take actions in external services (create tickets, post messages, update records)
- Authenticate once and maintain persistent connections

Without MCP connections, plugins still work. Claude will ask you for any information it would have pulled from tools automatically. MCP makes the experience smoother but is never strictly required.

The Three Types of MCP Servers

When building or customizing plugins, you will encounter three server types:

stdio: Local Process Servers

These run a program on your local machine. Used for custom servers, local tools, or when you need to run code alongside Claude.

```
{
  "mcpServers": {
    "my-local-server": {
      "command": "node",
      "args": ["${CLAUDE_PLUGIN_ROOT}/servers/server.js"],
      "env": {
        "API_KEY": "${API_KEY}"
      }
    }
  }
}
```

When to use: Custom internal tools, local databases, programs that need to run on your machine.

SSE: Hosted Servers with OAuth

These connect to cloud-hosted services using OAuth authentication (the "Sign in with..." flow). Most major SaaS integrations use this type.

```
{
  "mcpServers": {
    "asana": {
      "type": "sse",
      "url": "https://mcp.asana.com/sse"
    }
  }
}
```

When to use: Cloud services that offer OAuth sign-in (Asana, Linear, and similar platforms).

HTTP: REST API Servers

These connect to HTTP endpoints, often with token-based authentication. Used for REST APIs and services that provide MCP endpoints.

```
{
  "mcpServers": {
    "hubspot": {
      "type": "http",
      "url": "https://mcp.hubspot.com/anthropic"
    }
  }
}
```

When to use: SaaS platforms with MCP endpoints (Slack, HubSpot, Notion, Figma, and many others).

Choosing the Right Server Type

SCENARIO	RECOMMENDED TYPE	WHY
Major SaaS product (Slack, HubSpot, Notion)	HTTP or SSE	Cloud-hosted, managed by the vendor
Service with "Sign in with..." OAuth	SSE	Handles authentication automatically
Custom internal tool	stdio	Runs locally, full control over the process
REST API with token authentication	HTTP	Simple header-based auth
Local database or file system tool	stdio	Needs direct machine access

The .mcp.json Configuration File

MCP server connections are defined in a JSON file, typically `.mcp.json` at the plugin root. The create-cowork-plugin skill creates this file for you when your plugin needs external connections. The customizer skill updates it when connecting your organization's specific tools.

Here is an example of a fully configured `.mcp.json` from a customized engineering plugin:

```json
{
  "mcpServers": {
    "github": {
      "type": "http",
      "url": "https://api.githubcopilot.com/mcp/"
    },
    "asana": {
      "type": "sse",
      "url": "https://mcp.asana.com/sse"
    },
    "slack": {
      "type": "http",
      "url": "https://slack.mcp.claude.com/mcp"
    },
    "figma": {
      "type": "http",
      "url": "https://mcp.figma.com/mcp"
    },
    "datadog": {
      "type": "http",
      "url": "https://api.datadoghq.com/mcp",
      "headers": {
        "DD-API-KEY": "${DATADOG_API_KEY}",
        "DD-APPLICATION-KEY": "${DATADOG_APP_KEY}"
      }
    }
  }
}
```

Portable Paths and Environment Variables

Two important patterns keep MCP configurations portable and secure:

`${CLAUDE_PLUGIN_ROOT}` resolves to the plugin's actual directory path at runtime. Use this for all references to files within the plugin. This ensures the plugin works regardless of where it is installed.

`${ENV_VAR_NAME}` pulls values from the user's environment. Use this for credentials, API keys, and any values that differ between users or environments:

```json
{
  "headers": {
    "Authorization": "Bearer ${MY_API_TOKEN}"
  }
}
```

Never hardcode credentials or absolute paths in plugin files. Environment variables keep secrets out of plugin files, and `${CLAUDE_PLUGIN_ROOT}` keeps paths portable.

The Category-to-Keywords Mapping

The customizer skill uses a category-to-keywords mapping to search the MCP registry for appropriate connectors. This is how it knows to search for "asana," "jira," and "linear" when looking for a project management tool, or "figma," "sketch," and "design" when looking for a design tool.

The complete mapping:

CATEGORY	KEYWORDS USED FOR SEARCH
Project management	asana, jira, linear, monday, tasks
Software coding	github, gitlab, bitbucket, code
Chat	slack, teams, discord
Documents	google docs, notion, confluence
Calendar	google calendar, calendar
Email	gmail, outlook, email
Design and graphics	figma, sketch, design
Analytics and BI	datadog, grafana, analytics
CRM	salesforce, hubspot, crm
Wiki and knowledge base	notion, confluence, outline, wiki
Data warehouse	bigquery, snowflake, redshift
Conversation intelligence	gong, chorus, call recording

When you build a new plugin with the creation skill and include MCP servers, the skill uses this same mapping to help you identify which server type and endpoint to configure.

How This Differs from Domain Plugins

Most domain plugins (marketing, sales, data) include pre-configured MCP connections to tools relevant to their domain. The marketing plugin connects to HubSpot, Amplitude, Slack, and others. The sales plugin connects to Salesforce, Gong, and LinkedIn.

The plugin management plugin has no MCP connections because it does not interact with external services during its own operation. However, it is deeply knowledgeable about MCP. Both skills understand server types, authentication methods, configuration formats, and the MCP registry. They use this knowledge to:

- Include correct MCP configurations when creating new plugins
- Connect appropriate MCP servers when customizing existing plugins
- Configure environment variables and authentication for different server types

In short: this plugin does not *use* MCP, but it *creates and configures* MCP connections for the plugins it builds.

Part V: Customization & Extension

This section covers plugin architecture in depth and explains how to extend existing plugins. While the plugin management plugin's skills handle creation and customization through guided workflows, understanding the underlying architecture gives you the knowledge to make informed decisions during those workflows and to extend plugins on your own.

Plugin Architecture: What a Plugin Actually Is

A plugin is a self-contained directory that extends what Claude can do inside the Cowork desktop application. Think of it like an expansion pack: it gives Claude specialized knowledge, new commands you can invoke, connections to external services, and even automated behaviors that trigger on certain events.

Plugins are designed for business users. You do not need to be a software developer to use, customize, or even create one. Every plugin is made up of plain-text files (mostly Markdown and JSON) organized in a specific directory structure. If you can edit a document and follow a template, you can work with plugins.

What plugins can contain:

COMPONENT	WHAT IT DOES	ANALOGY
Commands	Actions you trigger with a slash (e.g., `/campaign-plan`)	Keyboard shortcuts for complex tasks
Skills	Domain knowledge Claude loads when relevant topics come up	Reference books Claude can consult
Agents	Autonomous specialists for multi-step tasks	Dedicated team members with specific expertise
Hooks	Automatic behaviors triggered by events	Standing instructions that always apply
MCP Servers	Connections to external tools and services	Integrations with your existing software stack

Every plugin follows the same directory structure:

```
plugin-name/
├── .claude-plugin/
│   └── plugin.json          # Required: plugin identity and metadata
├── commands/                # Slash commands (Markdown files)
│   ├── do-something.md
│   └── check-something.md
├── skills/                  # Domain knowledge (subdirectories)
│   └── skill-name/
│       ├── SKILL.md         # Required per skill
│       ├── references/      # Detailed reference documents
│       ├── scripts/         # Utility scripts
│       ├── assets/          # Templates, configuration files
│       └── examples/        # Sample data, example configurations
├── agents/                  # Autonomous specialist definitions
│   └── agent-name.md
├── hooks/                   # Event-driven automation
│   └── hooks.json
├── .mcp.json                # External service connections
├── CONNECTORS.md            # Tool-category documentation
├── README.md                # Plugin documentation
└── LICENSE                  # License information
```

The only truly required file is `.claude-plugin/plugin.json`. Everything else is optional — only create directories for components the plugin actually uses.

Key structural rules:

- The `.claude-plugin/` folder and its `plugin.json` file are always required
- Component directories (`commands/`, `skills/`, `agents/`) sit at the plugin root, not inside `.claude-plugin/`
- Only create directories for components the plugin actually uses
- All directory and file names use kebab-case (lowercase words separated by hyphens)

Two layout patterns exist across the Cowork plugin library:

PATTERN	EXAMPLE	STRUCTURE
Versioned	`marketing/1.0.0/`	Plugin contents live inside a version subfolder
Direct	`customer-support/`	Plugin contents live at the plugin root

Both patterns work identically. The versioned pattern is useful when multiple versions of a plugin coexist on the same system.

The plugin.json Manifest

Located at `.claude-plugin/plugin.json`, this file tells the system who the plugin is. It is the only file that every plugin must have:

```
{
  "name": "my-plugin",
  "version": "1.0.0",
  "description": "Brief explanation of what the plugin does.",
  "author": {
    "name": "Your Name or Organization"
  }
}
```

FIELD	REQUIRED	DESCRIPTION
`name`	Yes	Unique identifier. Lowercase, hyphens, no spaces (kebab-case).
`version`	No	Semver format: MAJOR.MINOR.PATCH (e.g., `1.0.0`).
`description`	No	One to two sentence explanation of the plugin's purpose.
`author`	No	Object with `name` field identifying the creator.
`homepage`	No	URL to the plugin's website or documentation.
`repository`	No	URL to the source code repository.
`license`	No	License identifier (e.g., `MIT`, `Apache-2.0`).
`keywords`	No	Array of strings for discovery (e.g., `["marketing", "campaigns"]`).

Custom component paths can be specified in `plugin.json` to supplement (not replace) automatic component discovery. For example, you can point to a non-standard directory for commands or hooks.

The Placeholder System and CONNECTORS.md

When a plugin is designed to be shared across different organizations, it uses `~~` placeholders for tool references:

```
Check ~~project tracker for open tickets.
Post a summary to ~~chat in the team channel.
```

These placeholders are documented in a `CONNECTORS.md` file at the plugin root:

```
# Connectors

## How tool references work

Plugin files use `~~category` as a placeholder for whatever tool the user
connects in that category. Plugins are tool-agnostic.

## Connectors for this plugin

| Category | Placeholder | Included servers | Other options |
|----------|-------------|------------------|---------------|
| Chat | `~~chat` | Slack | Microsoft Teams, Discord |
| Project tracker | `~~project tracker` | Linear | Asana, Jira, Monday |
```

"Included servers" are pre-configured in the plugin's `.mcp.json` file. **"Other options"** are alternatives that work but require the user to configure the connection manually.

When to use placeholders: Only when building a plugin intended for distribution to people outside your organization. If the plugin is for internal use and everyone uses the same tools, skip placeholders and reference your tools directly.

Three Approaches to Extending a Plugin

There are three ways to add capabilities to an existing plugin. Each has different trade-offs. Choose the approach that fits your situation.

Approach A: Direct Plugin Modification

Edit the plugin's files directly — add content to existing skills, create new command files, modify the MCP configuration.

How it works: Open the plugin directory and change its files. Add a section to a skill's `SKILL.md`, drop a new command file into `commands/`, or add a new MCP server to `.mcp.json`. Changes take effect immediately.

When to use:

- Quick personal experiments
- Testing an idea before committing to a bigger change
- Small additions that do not fundamentally alter the plugin

Pros:

- Fast (minutes to implement)
- No installation or setup required
- Changes take effect immediately

Cons:

- Plugin updates may overwrite your changes
- No clean separation between original and customized content
- Hard to track what you changed versus what came with the plugin

Example: Adding a pharmaceutical compliance section to the marketing plugin's brand review command. Open the command file, add a new section on fair balance requirements and off-label risk checks, save. Done in five minutes.

Approach B: Fork the Plugin (Recommended for Teams)

Copy the entire plugin directory, give it a new name, and modify the copy. Install the fork as a separate plugin alongside the original (or instead of it).

How it works:

1. Copy the plugin directory:

```
marketing/1.0.0/  ⟶  marketing-pharma/1.0.0/
```

2. Update `plugin.json` in the copy:

```
{
  "name": "marketing-pharma",
  "version": "1.0.0",
  "description": "Marketing plugin extended for pharmaceutical companies."
}
```

3. Make your modifications — add skills, modify commands, update MCP servers.

4. Install the forked plugin.

When to use:

- Team-wide customizations that everyone should share
- Significant modifications that change the plugin's behavior
- When you need to maintain your changes across plugin updates
- When you want a clean, auditable separation between original and custom content

Pros:

- Full control over all content
- Changes survive plugin updates (your fork is separate)
- Clean separation between original and customized content
- Can be distributed to your entire team

Cons:

- You must manually track and merge upstream changes when the original plugin updates
- Maintaining a full copy requires more storage and oversight

Example: Creating a pharmaceutical marketing plugin by forking the marketing plugin, adding regulatory compliance and medical affairs skills, adding new commands for MLR review and congress planning, and updating MCP connections to include PubMed and ClinicalTrials.gov.

Approach C: Complementary Skills via CLAUDE.md (Lightest Touch)

Keep the original plugin completely untouched. Add routing rules in your project's `CLAUDE.md` file and create additional skills in your project's `.claude/skills/` directory.

How it works:

1. Create a new skill directory in your project (not inside the plugin):

```
your-project/
├── .claude/
│   └── skills/
│       └── pharma-compliance/
│           ├── SKILL.md
│           └── references/
│               └── mlr-checklist.md
│
└── CLAUDE.md
```

2. Add a routing rule in your project's `CLAUDE.md`:

```
## Plugin Extensions

When creating pharmaceutical marketing content, also apply the
pharma-compliance skill for regulatory review requirements.
```

3. The original marketing plugin continues to work unchanged. Your project-level skill adds knowledge on top.

When to use:

- Adding project-specific knowledge without touching any plugin
- Supplementing a plugin for a particular context without affecting other projects
- When you want zero risk of conflicting with plugin updates

Pros:

- Zero risk to the original plugin
- Changes are project-scoped (do not affect other projects)
- Coexists cleanly with any plugin version
- Easy to maintain (your additions are fully separate)

Cons:

- Limited to adding knowledge — cannot modify existing plugin behavior
- Requires maintaining both the plugin and your project-level additions
- Only works within the project that has the `CLAUDE.md` file

Example: You use the standard marketing plugin across all projects. For one client in pharmaceuticals, you create a project-level `pharma-compliance` skill with MLR review checklists and add a `CLAUDE.md` rule that activates it when creating content for that client. Other projects using the same marketing plugin are unaffected.

Choosing the Right Approach

SITUATION	RECOMMENDED APPROACH
Quick personal experiment	A: Direct Modification
Team-wide customization	B: Fork the Plugin
Project-specific additions	C: CLAUDE.md Skills
Industry adaptation for distribution	B: Fork the Plugin
Adding company-specific knowledge	B or C depending on scope
Temporary or experimental changes	A: Direct Modification

Adding Industry-Specific Features

Many organizations need plugins adapted for their industry. Here are common patterns:

Start with a domain plugin. If a marketing, sales, finance, or legal plugin covers your base workflows, fork it rather than building from scratch.

Add industry-specific skills. Create new skills for domain knowledge unique to your industry:

- Pharmaceutical: regulatory compliance, medical affairs, KOL management
- Financial services: compliance review, risk assessment, regulatory reporting
- Healthcare: HIPAA compliance, clinical documentation, care coordination
- Manufacturing: quality management, supply chain optimization, safety compliance

Modify existing commands. Extend existing commands with industry-specific sections. Add compliance checks to content review commands. Add regulatory requirements to planning commands.

Update MCP connections. Connect industry-specific tools. Pharmaceutical companies may need PubMed and ClinicalTrials.gov. Financial firms may need Bloomberg terminal access.

Adding Company-Specific Features

Beyond industry, every company has unique processes and conventions:

Embed your brand voice. If you have brand guidelines, create a skill that Claude references when generating content. Include tone attributes, terminology preferences, and examples of good and bad writing.

Encode your workflows. If your team follows specific processes (sprint ceremonies, review stages, approval flows), encode them in skills so Claude follows the same steps.

Reference your tools by name. Use the customizer skill to replace generic tool references with your actual tool names and URLs.

Include your templates. If your team uses standard formats for reports, plans, or documents, put them in a skill's `assets/` or `references/` directory so Claude can apply them.

Set up hooks for governance. If your organization has policies that should always be enforced (security reviews on code changes, compliance checks on content, naming conventions on files), implement them as hooks that fire automatically.

Part VI: File Reference

Complete File Tree

```
cowork-plugin-management/0.2.1/
├── .claude-plugin/
│   └── plugin.json                                     # Plugin metadata
├── skills/
│   ├── create-cowork-plugin/
│   │   ├── SKILL.md                                    # Plugin creation workf
│   │   └── references/
│   │       ├── component-schemas.md                    # Component format spec
│   │       └── example-plugins.md                      # Three example plugins
│   └── cowork-plugin-customizer/
│       ├── SKILL.md                                    # Plugin customization
│       ├── LICENSE.txt                                 # Skill license
│       ├── examples/
│       │   └── customized-mcp.json                     # Example MCP configura
│       └── references/
│           ├── mcp-servers.md                          # MCP discovery guide
│           └── search-strategies.md                    # Knowledge search patt
└── LICENSE                                             # Plugin license
```

File Count: 10 files total

Detailed File Descriptions

`.claude-plugin/plugin.json`

The plugin manifest. Contains the plugin name (`cowork-plugin-management`), version, description, and author information. This is the only truly required file — it is how the system recognizes the directory as a plugin.

```
{
  "name": "cowork-plugin-management",
  "version": "0.2.1",
  "description": "Create, customize, and manage plugins tailored to your organization'
  "author": {
    "name": "Anthropic"
  }
}
```

`skills/create-cowork-plugin/SKILL.md`

The core file for the plugin creation skill. Contains the five-phase guided workflow (Discovery, Component Planning, Design, Implementation, Review and Package), the complete plugin directory structure specification, the plugin.json schema, a summary of component types, the `~~` placeholder system documentation, the `${CLAUDE_PLUGIN_ROOT}` variable explanation, and best practices for plugin design. This is the primary knowledge source that Claude loads when you express intent to create a plugin.

Frontmatter fields:

- `name` : create-cowork-plugin
- `description` : Trigger phrases including "create a plugin," "build a plugin," "make a new plugin," "develop a plugin," "scaffold a plugin," "start a plugin from scratch," and "design a plugin"
- `compatibility` : Requires Cowork desktop app environment

`skills/create-cowork-plugin/references/component-schemas.md`

Detailed format specifications for every plugin component type. This is the source of truth for file formats, frontmatter fields, syntax rules, and structural requirements. Claude loads this reference on demand during Phase 4 (Implementation) of the creation workflow. Covers:

- Commands: frontmatter fields (`description` , `allowed-tools` , `model` , `argument-hint`), `$ARGUMENTS` syntax, `@path` file inclusion, inline bash execution, `${CLAUDE_PLUGIN_ROOT}` variable
- Skills: frontmatter fields (`name` , `description` , `version`), writing style rules, progressive disclosure levels, directory structure conventions
- Agents: frontmatter fields (`name` , `description` , `model` , `color` , `tools`), `<example>` blocks for triggering conditions, naming rules, color guidelines
- Hooks: event types (PreToolUse, PostToolUse, Stop, SessionStart, etc.), prompt-based vs. command-based hooks, `hooks.json` format, decision output format (`approve` , `block` , `ask_user`)
- MCP Servers: stdio, SSE, and HTTP types with configuration examples, environment variable expansion, server type selection guide
- CONNECTORS.md: format specification and `~~` placeholder usage patterns

---

`skills/create-cowork-plugin/references/example-plugins.md`

Three complete example plugin structures at different complexity levels:

1. **Minimal Plugin (meeting-notes):** Single command, no other components. Demonstrates the simplest possible plugin — just a `plugin.json` and one command file.
2. **Standard Plugin (code-quality):** One skill (coding-standards), two commands (`/review` , `/fix-lint`), and GitHub MCP integration. Shows how domain knowledge, user actions, and tool connections work together.
3. **Full-Featured Plugin (engineering-workflow):** One skill (team-processes), two commands (`/standup-prep` , `/create-ticket`), one agent (ticket-analyzer), session-start hooks, MCP integration (Linear, GitHub, Slack), and `CONNECTORS.md` with tool-agnostic placeholders. Demonstrates every component type.

`skills/cowork-plugin-customizer/SKILL.md`

The core file for the plugin customization skill. Contains the four-phase workflow (Gather Context, Create Todo List, Complete Todo Items, Search for MCPs), the `~~` placeholder scanning process, guidelines for presenting changes in non-technical language, the summary output format, and plugin packaging instructions. This is the primary knowledge source that Claude loads when you express intent to customize a plugin.

Frontmatter fields:

- `name` : cowork-plugin-customizer
- `description` : Trigger phrases including "customize a plugin," "replace tool placeholders," and "configure MCP servers for a plugin"
- `compatibility` : Requires Cowork desktop app environment with mounted plugin directories

---

`skills/cowork-plugin-customizer/references/mcp-servers.md`

MCP discovery and connection guide used during Phase 4 of the customization workflow. Contains:

- Available tools: `search_mcp_registry` (search by keywords, returns up to 10 results with name, description, tools, URL, UUID, and connection status) and `suggest_connectors` (display Connect buttons for users to authenticate)
- Category-to-keywords mapping for 12 categories with associated search terms
- The MCP connection workflow: find customization point, check earlier findings, search registry, present choices, connect, update config
- Plugin MCP config file location rules (check `plugin.json` for custom `mcpServers` path first, fall back to `.mcp.json` at root)

- Config file format examples for all three server types (stdio, SSE, HTTP)
- Note about first-party integrations not needing plugin config entries

---

`skills/cowork-plugin-customizer/references/search-strategies.md`

Knowledge MCP search patterns used during Phase 1 of the customization workflow. Contains detailed query strategies organized by category:

- Finding tool names: source control (GitHub, GitLab, Bitbucket), project management (Asana, Jira, Linear), chat (Slack, Teams), analytics (Datadog, Grafana), design (Figma, Sketch), CRM (Salesforce, HubSpot)
- Finding organization values: workspace/project IDs, team conventions (story points, estimation scales), channel/team names
- Fallback behavior when no knowledge MCPs are available: skip automatic discovery and proceed directly to asking the user

---

`skills/cowork-plugin-customizer/examples/customized-mcp.json`

A complete example of a fully configured `.mcp.json` file after customization. Shows five server configurations:

- GitHub (HTTP with Bearer token auth)
- Asana (SSE with OAuth)
- Slack (HTTP)
- Figma (HTTP)
- Datadog (HTTP with API key and application key headers)

Also includes a `recommendedCategories` array listing tool categories the plugin benefits from: source-control, project-management, chat, documents, wiki-knowledge-base, design-graphics, and analytics-bi.

`skills/cowork-plugin-customizer/LICENSE.txt`

License file for the customizer skill.

`LICENSE`

License file for the plugin as a whole.

Cross-Reference: When Claude Reads Each File

FILE	READ WHEN
`plugin.json`	Always (plugin identification and loading)
`create-cowork-plugin/SKILL.md`	User expresses intent to create a plugin
`component-schemas.md`	During Phase 4 (Implementation) when creating component files
`example-plugins.md`	When Claude needs a reference structure or the user asks for examples
`cowork-plugin-customizer/SKILL.md`	User expresses intent to customize a plugin
`mcp-servers.md`	During Phase 4 (Connect MCPs) of customization
`search-strategies.md`	During Phase 1 (Gather Context) of customization
`customized-mcp.json`	When Claude needs a reference for MCP configuration format

Part VII: Troubleshooting & FAQ

Common Issues and Solutions

Issue: Plugin Not Loading After Installation

Symptoms:

- You installed the plugin but Claude does not recognize it
- Asking "What plugins are installed?" does not list cowork-plugin-management

Diagnosis:

- Plugin not properly installed or not recognized by the system
- Plugin directory structure is incorrect

Solution:

1. Restart Claude (close and reopen the Cowork desktop app)
2. Verify installation: Ask "What plugins are installed?"
3. If installing via CLI, check that the command completed without errors
4. Ensure the `.claude-plugin/plugin.json` file exists and contains valid JSON with at least a `name` field
5. Reinstall through Plugin Settings if the issue persists

Issue: Skills Not Triggering When Expected

Symptoms:

- You say "I want to create a plugin" but the creation skill does not activate
- You say "Customize this plugin" but the customizer does not engage
- Claude responds generically instead of entering the guided workflow

Diagnosis:

- Plugin not loaded into the current session
- Skill description does not match the exact phrasing you used
- Session started before plugin was installed

Solution:

1. Start a new session (the plugin loads at session start)
2. Use explicit trigger phrases: "I want to create a plugin from scratch" or "Help me customize the marketing plugin for my organization"
3. Verify the plugin is installed: Ask "What plugins are installed?"
4. Check that SKILL.md exists in the correct location within each skill directory
5. Verify frontmatter has correct `---` delimiters and the `name` field matches the directory name

---

Issue: "Customization Only Available in Desktop App" Error

Symptoms:

- You try to create or customize a plugin and Claude says it requires the Cowork desktop app
- The skill aborts with an incompatibility message

Diagnosis:

- You are running Claude in a remote container, CLI-only session, or environment without access to mounted plugin directories

Solution:

- Switch to the Cowork desktop app. Both skills require access to mounted plugin directories (`mnt/.plugins/`, `mnt/.local-plugins/`) and the outputs directory for delivering `.plugin` files
- If you must use the CLI, the skills will have limited functionality. You may be able to discuss plugin architecture, but the skills cannot deliver `.plugin` files

---

Issue: Plugin Validation Fails

Symptoms:

- The creation skill's Phase 5 (Review and Package) reports validation errors
- Running `claude plugin validate` manually produces errors or warnings

Common Errors and Solutions:

ERROR MESSAGE	CAUSE	SOLUTION
`name field is required`	`plugin.json` missing `name`	Add `"name": "plugin-name"` to plugin.json

ERROR MESSAGE	CAUSE	SOLUTION
`Invalid version format`	Version not in semver format	Use MAJOR.MINOR.PATCH (e.g., `0.1.0`)
`Skill missing frontmatter`	SKILL.md has no `---` delimiters	Add frontmatter with `name` and `description` fields
`Command file not found`	plugin.json references non-existent command	Remove the reference or create the missing file
`Hooks config is not valid JSON`	hooks.json has a syntax error	Validate and fix the JSON syntax

The creation skill catches most of these issues during implementation (Phase 4) by validating against component schemas. If errors slip through, Phase 5 catches them.

Issue: Skill Does Not Trigger When Expected (in a Plugin You Created)

Symptoms: You mention the topic the skill covers, but Claude does not load the skill in a plugin you built.

Causes:

1. Skill description is too generic or missing specific trigger phrases
2. Skill name does not match directory name
3. SKILL.md is not in the correct location (`skills/skill-name/SKILL.md`)
4. Frontmatter is malformed

Solution:

Check the skill's frontmatter. Verify:

- Frontmatter has correct `---` delimiters
- `name` field matches directory name exactly
- `description` includes specific trigger phrases in quotes

Edit the `description` field to include explicit trigger phrases:

```
description: >
  Use this skill when the user asks to "create a campaign", "plan a
  campaign", "design a marketing campaign", or needs guidance on campaign
  strategy, audience targeting, or channel selection.
```

Reinstall the plugin and test again.

Issue: Command Fails with "Tool Not Allowed" Error (in a Plugin You Created)

Symptoms: Running a command produces an error that a tool cannot be used.

Cause: The `allowed-tools` frontmatter field in the command restricts which tools Claude can use, and the command is trying to use a tool not on the list.

Solution:

Check the command's frontmatter for `allowed-tools`:

```
allowed-tools: Read, Write, Grep
```

If the command tries to use `Edit` but `allowed-tools` does not include it, the error occurs. Add the required tool:

```
allowed-tools: Read, Write, Edit, Grep
```

Or remove the `allowed-tools` field entirely to allow all tools.

For MCP tools, ensure the exact tool name is specified:

```
allowed-tools: ["mcp__hubspot__search_contacts", "Read", "Write"]
```

Issue: MCP Server Fails to Connect During Customization

Symptoms:

- The customizer identifies your tools but cannot connect the MCP servers
- Authentication completes but Claude says the tool is not connected
- Connection attempts time out

Diagnosis:

- OAuth permissions not fully granted
- Tool's API is down or rate-limited
- Network or firewall blocking the connection

- MCP server URL is incorrect

Solution:

1. Disconnect and reconnect: try the authentication flow again
2. Check permissions during OAuth: ensure you approve all requested scopes
3. Verify the tool's status: check the service's status page for API issues
4. Try a simpler connection first: if connecting five tools fails, try connecting just one to isolate the issue
5. For services requiring API keys (like Datadog), verify the environment variables are set correctly

---

Issue: Hook Blocks Operations Incorrectly (in a Plugin You Created)

Symptoms: A hook you created blocks operations that should be allowed.

Cause: Hook prompt is too strict, matcher pattern is too broad, or command-based hook script has a logic error.

Solution:

1. Check `hooks/hooks.json` and review the matcher pattern. Narrow it if too broad:
   - `"Write|Edit"` fires for Write or Edit tools
   - `"Write.*\\.md"` fires only for Markdown file writes
   - Empty matcher `""` fires for all events of that type
2. For prompt-based hooks, refine the prompt to be more specific about what to allow and what to block
3. For command-based hooks, test the script manually to ensure it returns the correct JSON decision (`approve`, `block`, or `ask_user`)
4. Temporarily remove the hook to confirm it is the source of the problem

---

Issue: Created Plugin Has Wrong Directory Structure

Symptoms: Commands or skills are not recognized because they are placed in the wrong directories.

Solution:

The correct structure places component directories at the plugin root, not inside `.claude-plugin/`:

```
plugin-name/
├── .claude-plugin/
│   └── plugin.json           # Only plugin.json goes here
├── commands/                 # At root, NOT inside .claude-plugin/
├── skills/                   # At root, NOT inside .claude-plugin/
├── agents/                   # At root, NOT inside .claude-plugin/
└── hooks/                    # At root, NOT inside .claude-plugin/
```

If the creation skill produced the wrong structure, move the component directories to the correct locations and reinstall.

Frequently Asked Questions

Q: Can I use this plugin to modify itself?

A: Yes, the plugin management plugin is itself a plugin. You can fork it and add organization-specific templates, examples, or conventions. However, the lighter approach is usually to create complementary skills in your project's `CLAUDE.md` that supplement the creation workflow with your organization's standards (see Approach C in Part V).

Q: How long does it take to create a plugin from scratch?

A: A simple plugin with one skill and one or two commands typically takes 10-15 minutes of conversation. A complex plugin with multiple skills, commands, agents, hooks, and MCP integrations can take 30-45 minutes. The skill handles all the file creation — your time is spent in the discovery and design phases answering questions about what you want.

Q: Can I create plugins that work outside of Cowork?

A: The `.plugin` file format and the underlying directory structure are compatible with Claude Code (the CLI). Plugins created through this skill can be installed in both Cowork and Claude Code. However, the creation and customization *process* requires the Cowork desktop app because of its access to mounted plugin directories and the outputs folder.

Q: What happens if I customize a plugin but skip some questions?

A: The customizer leaves `~~` placeholders unchanged for any questions you skip. You can run the customizer again later when you have the information. This is by design — partial customization is better than forced guesses.

Q: Should I create a plugin or write a CLAUDE.md file?

A: Use `CLAUDE.md` for project-specific instructions that apply only to one project. Create a plugin when you want reusable, distributable capabilities that work across projects and team members. If you find yourself copying the same `CLAUDE.md` content across multiple projects, that is a strong signal to create a plugin instead.

Q: Can multiple people on my team create plugins simultaneously?

A: Yes. Each person runs the creation skill in their own Cowork session. The resulting `.plugin` files can be shared through your normal file distribution methods (shared drives, version control, internal package registries).

Q: How do I update a plugin I previously created?

A: Open the plugin directory and edit the files directly, or run the creation skill again with updated requirements. For version tracking, increment the version number in `plugin.json` following semver conventions: patch (0.1.1) for bug fixes, minor (0.2.0) for new features, major (1.0.0) for breaking changes.

Q: What is the difference between "customize" and "extend"?

A: **Customize** means adapting a plugin's existing content to your organization — replacing tool placeholders, configuring MCP connections, updating organization-specific values. The plugin's structure and capabilities stay the same. **Extend** means adding new capabilities — new commands, new skills, new integrations. Customization uses the customizer skill. Extension uses one of the three approaches in Part V (direct modification, forking, or complementary skills).

Q: How do I share a plugin with my team?

A: After creating or customizing a plugin, the skill delivers a `.plugin` file. Share this file with your team through any method you use for file distribution. Recipients install it through Cowork's Plugin Settings by opening the file, or through the CLI with `claude plugins add <path-to-file>`.

Q: Can I use this plugin to create plugins for industries I know nothing about?

A: The creation skill guides you through the process, but the quality of the resulting plugin depends on the domain knowledge you bring to the conversation. If you are building a pharmaceutical compliance plugin, you need to understand pharmaceutical compliance. The skill structures and packages that knowledge — it does not generate domain expertise from nothing.

Q: What if my organization uses a tool that does not have an MCP server?

A: The customizer will note that no MCP connector was found for that tool. The plugin still works — Claude just cannot connect to that tool directly. You can still reference the tool by name in your plugin's content, and Claude will ask users to provide data from that tool manually. If a tool later releases an MCP server, you can update the plugin's `.mcp.json` file.

Q: Is there a limit to how many components a plugin can have?

A: There is no hard limit, but practical guidelines apply. Plugins with dozens of skills or commands can become unwieldy and slow down context loading. If a plugin grows too large, consider splitting it into multiple focused plugins (e.g., `marketing-core`, `marketing-pharma`, `marketing-analytics`). Start small and add components based on real usage.

Q: How do I debug a plugin that is not working correctly?

A: Start with these steps:

1. Ask Claude "What plugins are installed?" to verify the plugin is loaded
2. Check `plugin.json` for valid JSON and correct `name` field
3. For skills: verify frontmatter has `---` delimiters, `name` matches directory name, `description` includes trigger phrases
4. For commands: verify file is in `commands/` at plugin root (not inside `.claude-plugin/`), filename is kebab-case with `.md` extension
5. For MCP: verify `.mcp.json` is valid JSON, URLs are correct, environment variables are set
6. For hooks: verify `hooks.json` is valid JSON, matcher patterns are correct, scripts are executable
7. Run `claude plugin validate <path-to-plugin-json>` for automated validation

Q: How do hooks differ from skills in terms of when they activate?

A: Skills activate based on conversation topic — when you discuss something relevant to the skill's domain, Claude loads it. Hooks activate based on system events — when a specific action occurs (a file is written, a session starts, a tool is used). Skills provide knowledge; hooks enforce behavior. For example, a skill might teach Claude about coding standards, while a hook might automatically check every file write against those standards.

Q: What version should I use when creating my first plugin?

A: Start at `0.1.0`. The `0.x.x` range signals that the plugin is in development and may change. Move to `1.0.0` when the plugin is stable and in production use. Follow semver conventions: increment the patch number (0.1.1) for bug fixes, the minor number (0.2.0) for new features, and the major number (1.0.0) for breaking changes.

Q: Can I combine both skills in a single session — create a plugin and then customize it?

A: Yes. You can create a plugin using the creation skill, then immediately customize it using the customizer skill in the same session. This is useful when you want to build a template plugin with `~~` placeholders and then immediately personalize it for your organization. The creation skill delivers the plugin, and you can then say "Now customize that plugin for our tools" to engage the customizer.

This handbook covers the cowork-plugin-management plugin version 1.0.0. For the latest documentation, check the plugin's repository or the Cowork Plugin Settings page.

RationalEyes.ai

contact@rationaleyes.ai

Intelligent Automation for Knowledge Work

Plugin Management Handbook — Version 1.0.0