

CLAUDE COWORK

# The Complete Cowork Plugin Management Handbook

Version 0.2.1

---

VERSION 1.0.0

PLUGIN cowork-plugin-management

# Table of Contents

---

**Part I: Introduction and Overview**

---

**Part II: Commands Reference**

---

**Part III: Skills Deep Dive**

---

**Part IV: Connectors and MCP Integration**

---

**Part V: Workflows and Use Cases**

---

**Part VI: Best Practices and Guidelines**

---

**Part VII: Troubleshooting and Advanced Topics**

---

Conclusion

## Version 0.2.1

A comprehensive guide to creating, customizing, and managing Claude Cowork plugins using the cowork-plugin-management plugin.

---

# Part I: Introduction and Overview

---

## What Is the Plugin Management Plugin?

The cowork-plugin-management plugin is the meta-plugin in the Claude Cowork ecosystem — it is the tool you use to create and manage all other plugins. Think of it as the master key that unlocks the ability to extend Claude's capabilities for your specific organization, industry, or use case.

Unlike most plugins that add domain-specific capabilities (marketing, engineering, data analysis), this plugin operates at an architectural level. It provides two core capabilities:

1. **Plugin Creation:** A guided, conversational workflow for building new plugins from scratch
2. **Plugin Customization:** A systematic process for adapting generic plugin templates to your organization's specific tools and workflows

**Critical architectural note:** This plugin has no slash commands and no MCP server connections. It operates entirely through skills that trigger automatically based on conversational context. This design choice reflects its nature as a development tool — you do not invoke plugin management operations with slash commands; you have conversations about what you want to build or customize, and the skills activate as needed.

## Who Should Use This Plugin?

This plugin is designed for several key personas:

**Plugin Administrators** are IT professionals or system administrators who manage Claude deployments across an organization. They use this plugin to create company-specific plugins that standardize workflows, enforce policies, and connect internal tools.

**Team Leads** are managers who want to codify their team's processes and best practices into reusable skills. They use this plugin to create plugins that embed institutional knowledge, ensuring consistency as team members come and go.

**Enterprise IT Managers** oversee Claude rollouts across multiple departments. They use this plugin to customize existing plugins for their organization's tool stack (replacing generic "project tracker" references with "Jira" or "Asana," for example) and to manage plugin versions across teams.

**Power Users** are individuals who have discovered workflows they want to standardize and share. They use this plugin to package their custom prompts, reference documents, and tool integrations into formal plugins.

## Installation

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

**Note:** All standard Cowork plugins, including **Plugin Management**, are available from **Plugin Settings** with a single click. This plugin requires the Cowork desktop app — its skills depend on access to mounted plugin directories and the outputs directory for delivering `.plugin` files. It cannot function in CLI-only or remote container environments.

**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 that the plugin creation and customization skills have limited functionality outside the Cowork desktop app, as they require access to mounted plugin directories (`mnt/.plugins/`, `mnt/.local-plugins/`) and the outputs directory for delivering `.plugin` files.

## What Makes This Plugin Different?

Most plugins extend Claude with domain knowledge — how to write marketing copy, how to analyze code, how to process scientific papers. The plugin management plugin extends Claude with *architectural* knowledge — how plugins themselves work, what components they contain, how to structure them correctly, and how to adapt them to different contexts.

This plugin is also unique in its delivery mechanism. While most plugins deliver value through commands (actions you invoke) or persistent skills (knowledge that is always available), this plugin's skills are **contextual and conversational**. They activate when you express intent to create or customize a plugin, guide you through a multi-phase workflow, and then deliver a concrete artifact (a `.plugin` file) at the end.

## The Two Core Skills

The plugin contains two major 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.

**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, gathers context from your 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.

Both skills require the Cowork desktop app environment with access to the outputs directory. They cannot function in remote containers or CLI-only sessions, as they need to deliver plugin files and access mounted plugin directories.

## How Plugin Management Fits Into the Ecosystem

Understanding this plugin requires understanding where it sits in the broader Claude plugin ecosystem:

### Plugin Development Lifecycle

```
Generic Plugin Template (with ~ placeholders)
    ↓
[cowork-plugin-customizer skill]
    ↓
Organization-Specific Plugin (Acme Corp tools)
    ↓
Distribution as .plugin file
    ↓
End Users Install and Use

OR

Identified Need
    ↓
[create-cowork-plugin skill]
    ↓
Custom Plugin (built from scratch)
    ↓
Testing and Iteration
    ↓
Distribution as .plugin file
```

This plugin enables both paths: creating net-new plugins and customizing existing ones. It sits at the beginning of the plugin lifecycle, enabling everything downstream.

## The Value Proposition

Why invest time in creating or customizing plugins rather than just writing good prompts?

**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 know 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 to dozens or hundreds of users at once.

## What You Will Learn in This Handbook

This handbook provides comprehensive coverage of the plugin management plugin:

- **Part II** explains that this plugin has no commands and why that architectural decision was made
- **Part III** deep-dives into both skills, covering the five-phase creation workflow and the four-phase customization workflow in detail
- **Part IV** clarifies that this plugin has no MCP connectors of its own but extensively manages MCP configurations for other plugins
- **Part V** provides real-world workflows: creating a plugin from scratch, customizing a template, forking and extending an existing plugin, and troubleshooting common issues
- **Part VI** covers best practices: when to create versus customize, how to structure knowledge for progressive disclosure, and how to write skills that trigger reliably
- **Part VII** addresses troubleshooting, advanced customization scenarios, and how to extend the plugin management plugin itself

By the end of this handbook, you will understand not only how to use the plugin management plugin but also the deeper architectural principles that make the entire plugin system work.

---

# Part II: Commands Reference

---

## Why This Plugin Has No Commands

The cowork-plugin-management plugin is unusual: it contains **no slash commands at all**. There is no `/create-plugin` or `/customize` command. This is an intentional architectural choice, not an oversight.

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

**Commands** are user-initiated, explicit actions. The user types `/campaign-plan` and Claude executes the workflow defined in that command file. Commands are best for operations with clear start and end points, where the user knows exactly what they 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 workflow.

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

- Discovery conversations to understand what the user wants to build
- Clarifying questions that depend on previous answers
- Decision points where the user 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. The user 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 back-and-forth dialogue (which blurs the line between command and conversation).

Instead, the plugin management plugin uses **conversational skills** that activate when the user expresses intent:

- "I want to create a plugin for pharmaceutical marketing" → `create-cowork-plugin` skill triggers
- "I need to customize this plugin for my team's tools" → `cowork-plugin-customizer` skill triggers

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

## How to Invoke Plugin Management Capabilities

Since there are no commands, how do you actually use 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"

To customize an existing plugin:

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

The skills' trigger descriptions encode these phrases, ensuring reliable activation.

## Comparison to Command-Driven Plugins

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

**Marketing Plugin Commands (explicit invocation):**

- `/campaign-plan` — Generate a campaign brief
- `/brand-review` — Review content for brand alignment
- `/competitive-brief` — Create competitive analysis
- `/performance-report` — Generate campaign performance report

**Marketing Plugin Skills (automatic activation):**

- `brand-voice` — Loads when conversation involves brand guidelines or tone
- `campaign-planning` — Loads when discussing campaign strategy
- `competitive-intelligence` — Loads when analyzing competitors
- `performance-analysis` — Loads when reviewing metrics or results

The plugin management plugin is **all skills, no commands**, because its entire function is knowledge-intensive and conversational, not action-oriented.

## The Implication for Users

If you are accustomed to slash commands as the primary plugin interface, 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 just run `/create-plugin marketing` and expect a finished plugin to appear. You must engage in the discovery, planning, and design phases.

The benefit: the resulting plugin is tailored to your exact needs, not a generic template with placeholders you have to fill in later.

---

# Part III: Skills Deep Dive

---

This section is the heart of the handbook, as skills are the entire substance of the plugin management plugin. We will examine both skills in detail, including their trigger mechanisms, internal workflows, design principles, and how to use 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 you trigger this skill, you are not just getting documentation about how plugins work — you are getting a structured process that walks you through creating one.

### *When to Use It*

Use this skill when:

- You have a clear need that is not met by existing plugins
- You want to package institutional knowledge or workflows into a reusable form
- You are building a domain-specific plugin for a narrow use case (e.g., regulatory compliance for medical devices)
- 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 (extend or customize instead)
- You are unsure what you want to build (start with a conversation to clarify, then engage the skill)
- You are looking for documentation only (the plugin development guide in the handbook repository is better for reference)

## How It Triggers

The skill's description encodes these trigger 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 description also notes:

*"This skill requires Cowork mode with access to the outputs directory for delivering the final .plugin file."*

If the user tries to trigger this skill in a remote container or CLI session, the skill will detect the incompatible environment and abort with a helpful error message.

## 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.

**Output:** A clear statement of plugin purpose and scope.

### Phase 2: Component Planning

Based on the discovery phase, the skill determines which component types are needed. It considers:

COMPONENT	DECISION CRITERIA
Skills	Does the plugin require specialized knowledge that Claude should load on demand? (domain expertise, reference schemas, workflow guides)

COMPONENT	DECISION CRITERIA
Commands	Are there user-initiated actions with clear inputs and outputs? (deploy, configure, analyze, review)
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, generation, analysis)
Hooks	Should certain behaviors happen automatically on specific events? (enforce policies, load context, validate operations)

The skill presents a component plan table:

Component	Count	Purpose
Skills	1	Domain knowledge for regulatory compliance
Commands	2	/mlr-review, /fair-balance-check
Agents	0	Not needed
Hooks	1	Validate all content writes for compliance
MCP	1	Connect to regulatory database

You confirm or adjust before proceeding.

**Output:** A confirmed list of components to create.

### Phase 3: Design & Clarifying Questions

This is the most interactive phase. The skill asks targeted questions for each component type in the plan. Questions are grouped by component and presented in batches.

#### 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 or automated?

#### For agents:

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

#### For hooks:

- Which events? (PreToolUse, PostToolUse, Stop, SessionStart, etc.)
- What behavior – validate, block, modify, add context?
- Prompt-based (LLM-driven) or command-based (deterministic script)?

#### For MCP servers:

- What server type? (stdio for local processes, SSE for hosted OAuth services, 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.

**Output:** Detailed specifications for every component.

### Phase 4: Implementation

The skill creates all plugin files following the specifications from Phase 3. Order of operations:

1. Create the plugin directory structure
2. Create `plugin.json` manifest
3. Create each component (see component schemas for exact formats)
4. Create `README.md` documenting the plugin

Implementation principles the skill follows:

- **Commands are instructions for Claude**, not messages to the user. The skill writes command files as directives ("Generate a report," "Analyze the data").
- **Skills use progressive disclosure**: Lean SKILL.md body (under 3,000 words), detailed content in `references/`. The skill frontmatter description includes specific trigger phrases.
- **Agents** need descriptions with `<example>` blocks showing triggering conditions, plus system prompts in the markdown body.
- **Hooks** configuration goes in `hooks/hooks.json`. The skill uses `#{CLAUDE_PLUGIN_ROOT}` for all script paths and prefers prompt-based hooks for complex logic.
- **MCP configs** go in `.mcp.json` at plugin root. The skill uses `#{CLAUDE_PLUGIN_ROOT}` for local server paths and documents required environment variables in README.

## Phase 5: Review & Package

The skill:

1. Summarizes what was created — lists each component and its purpose
2. Asks if you want any adjustments
3. Runs `claude plugin validate <path-to-plugin-json>` to check for errors and warnings; fixes any issues found
4. Packages the plugin as a `.plugin` file:

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

The skill always creates the zip in `/tmp/` first, then copies to the outputs folder. Writing directly to the outputs folder may fail due to permissions.

The skill uses the plugin name from `plugin.json` for the `.plugin` filename (e.g., if name is `code-reviewer`, output is `code-reviewer.plugin`).

**Output:** A finished `.plugin` file delivered to the outputs directory, where it appears in the chat as a rich preview. You can browse the plugin's files and accept it with a button click.

### *Special Features and Patterns*

#### 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.

#### Progressive Disclosure in Skill Design

When creating skills as components within a new plugin, the skill applies a three-tier loading strategy:

TIER	CONTENT	SIZE GUIDELINE
Metadata	name + description from frontmatter	~100 words
SKILL.md body	Core knowledge	1,500-2,000 words (max 3,000)

TIER	CONTENT	SIZE GUIDELINE
References	Detailed docs in <code>references/</code> , <code>examples/</code> , <code>scripts/</code>	Unlimited

This ensures skills are lightweight enough to load quickly but comprehensive enough to be useful.

### Component Schemas as Reference Material

The skill contains a reference file (`references/component-schemas.md`) with exhaustive specifications for every component type. This is the source of truth for file formats, frontmatter fields, syntax rules, and structural requirements.

### Example Plugins as Templates

The skill includes three complete example plugins in `references/example-plugins.md`:

1. **Minimal Plugin:** Single command, no other components
2. **Standard Plugin:** Skill + commands + MCP integration
3. **Full-Featured Plugin:** Skills, commands, agents, hooks, MCP integration, tool-agnostic connectors

These serve as starting points and reference implementations.

### *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" is specific enough to guide 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 is more useful than one with five half-baked components. You can add more in version 0.2.0.

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

**Think about tool integration upfront.** If your plugin needs to connect to external services, identify 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, or long reference guides, put those in `references/` rather than bloating the SKILL.md body.

### ***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 by:

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

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" or "Jira")
- 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 are making structural changes (adding new skills or commands; fork the plugin instead)

## How It Triggers

The skill's description encodes:

*"Customize or personalize a Claude Code plugin for a specific organization's tools and workflows. Use when users want to customize a plugin, replace tool placeholders, or configure MCP servers for a plugin."*

Trigger phrases include:

- "Customize this plugin"
- "Replace tool placeholders"
- "Personalize the plugin for my team"
- "Configure this plugin for our tools"

The skill also requires the Cowork desktop app environment. If triggered in an incompatible environment, it aborts with an error message explaining that customization is only available in Cowork mode.

## The Four-Phase Workflow

### Phase 1: Gather Context from Knowledge MCPs

Before asking you any questions, the skill searches your connected knowledge sources to learn about your organization. It uses strategies detailed in [references/search-strategies.md](#).

### What it gathers:

- Tool names for each `~` placeholder category (project tracker, chat, design, analytics, etc.)
- Organizational processes and workflows
- Team conventions (naming patterns, status values, estimation scales)
- Configuration values (workspace IDs, project names, team identifiers)

### Sources it searches:

1. **Chat/Slack MCPs:** Tool mentions, integration discussions, workflow conversations
2. **Document MCPs:** Onboarding docs, tool setup guides, internal handbooks
3. **Email MCPs:** License notifications, admin setup emails, service invitations

The skill uses category-to-keyword mappings to search efficiently:

CATEGORY	SEARCH KEYWORDS
Project management	<code>["asana", "jira", "linear", "monday", "tasks"]</code>
Software coding	<code>["github", "gitlab", "bitbucket", "code"]</code>
Chat	<code>["slack", "teams", "discord"]</code>
Documents	<code>["google docs", "notion", "confluence"]</code>
Calendar	<code>["google calendar", "calendar"]</code>
Design/graphics	<code>["figma", "sketch", "design"]</code>
Analytics/BI	<code>["datadog", "grafana", "analytics"]</code>
CRM	<code>["salesforce", "hubspot", "crm"]</code>

The skill records all findings for use in Phase 3.

### Phase 2: Create Todo List from Customization Points

The skill runs `grep -rn '~~\w' /path/to/plugin --include='*.md' --include='*.json'` to find all lines containing `~~` placeholders.

It groups them by theme and creates a todo list with user-friendly descriptions:

- **Good:** "Learn how standup prep works at your company"

- **Bad:** "Replace placeholders in commands/standup-prep.md"

The todo list is written in plain language, focusing on learning about the organization rather than technical file operations. Users never see mentions of `~` prefixes or customization points — those are implementation details.

### Phase 3: Complete Todo Items

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

**If knowledge MCPs 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 `~project tracker` with "Linear" automatically.

**If knowledge MCPs did not provide an answer:** The skill uses `AskUserQuestion` to ask directly. It does not assume "industry standard" defaults. If the answer is unclear, it asks.

**If the user skips or says they 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:

1. **Placeholder replacements:** `~Jira` → `Asana` , `~your-org-channel` → `#engineering`
2. **URL pattern updates:** `tickets.example.com/your-team/123` → `app.asana.com/0/PROJECT_ID/TASK_ID`
3. **Organization-specific values:** Workspace IDs, project names, team identifiers

The skill applies changes using the Edit tool, ensuring exact string replacements.

### Phase 4: Search for Useful MCPs

After all customization points are resolved, the skill connects MCP servers for the tools that were identified. It follows the workflow in `references/mcp-servers.md` :

For each tool identified during customization:

1. **Search the MCP registry:** `search_mcp_registry(keywords=[ ... ])` using category keywords from the mapping table
2. **Present results:** Show all matching MCPs with descriptions and tool lists
3. **If not connected:** Use `suggest_connectors(directoryUuids=["uuid"])` to prompt OAuth or connection setup
4. **Update plugin MCP config:** Add the connection to the plugin's `.mcp.json` file

The skill checks `plugin.json` for a custom `mcpServers` path; if none is specified, it updates `.mcp.json` at the plugin root.

**Note:** First-party integrations (Gmail, Google Calendar, Google Drive) are connected at the user level and do not need plugin `.mcp.json` entries.

The skill collects all MCP results and presents them together in the summary output at the end — it does not present MCPs one at a time during the phase.

## Packaging the Plugin

After all customizations are applied, the skill packages the plugin:

```
cd /path/to/plugin && zip -r /tmp/plugin-name.plugin . -x "setup/*" && cp /tmp/plugin-
```

The skill excludes the `setup/` directory (if present) since it is no longer needed after customization. The `.plugin` file is delivered to the outputs directory as a rich preview.

**Important naming rule:** The skill uses the original plugin directory name for the `.plugin` filename. It does not rename the plugin or its files during customization — only placeholder values and content are replaced.

## Summary Output

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

```
## From searching Slack
- You use Asana for project management
- Sprint cycles are 2 weeks
- Team channel is #product-engineering

## From searching documents
- Story points use Fibonacci scale (1, 2, 3, 5, 8, 13)
- Code review requires two approvals before merge

## From your answers
- Ticket statuses are: Backlog, In Progress, In Review, Done
- Design files are managed in Figma workspace "Product 2025"
```

The skill then shows:

**MCPs connected during setup:** Services that were connected during the customization process

**MCPs the user should still connect:** Additional services mentioned in the plugin that are not yet connected, with instructions

If no knowledge MCPs were available in Phase 1 and the user had to answer questions manually, the skill includes a note:

*By the way, connecting sources like Slack or Microsoft Teams would let me find answers automatically next time you customize a plugin.*

This educates users about the value of connecting knowledge sources.

### ***Special Features and Patterns***

#### **Finding the Plugin Directory**

To locate the plugin files, the skill runs:

```
find mnt/.local-plugins mnt/.plugins -type d -name "*<plugin-name>*" 
```

If the plugin directory cannot be found, the skill recognizes the user is likely in a remote container and aborts with a helpful message:

*"Customizing plugins is currently only available in the desktop app's Cowork mode."*

#### **Non-Technical Output Requirement**

All user-facing output (todo list items, questions, summaries) must be in plain, non-technical language. The skill never mentions `~` prefixes, placeholders, or customization points to the user. Everything is framed in terms of learning about the organization and its tools.

#### **Preserving Plugin Identity**

The skill never changes the plugin's `name` field in `plugin.json`, the plugin directory name, skill names, or file names. It only replaces placeholder values within file content. This ensures the plugin retains its identity across customization.

#### **MCP Category Mapping**

The skill contains a comprehensive category-to-keywords mapping in `references/mcp-servers.md`, covering:

- Project management
- Software coding
- Chat
- Documents

- Calendar
- Email
- Design/graphics
- Analytics/BI
- CRM
- Wiki/knowledge base
- Data warehouse
- Conversation intelligence

This mapping enables efficient MCP discovery for the tools identified during customization.

### ***Best Practices for Using This Skill***

**Connect knowledge sources before customizing.** The more knowledge MCPs you have connected (Slack, email, internal documents), the fewer questions you will need to answer manually. The skill will find most answers automatically.

**Have organization context ready.** If you know you will need to provide workspace IDs, project names, or channel identifiers, gather that information before starting. The skill will ask for it.

**Customize in multiple passes if needed.** If you do not know all the answers during the first customization, skip those items. Run the customization again later when you have more information.

**Review the summary carefully.** The skill shows what it learned and what changes it made. Verify the tool replacements and configuration values are correct before accepting the customized plugin.

**Test the customized plugin immediately.** After customization, install the plugin and test its commands and skills to ensure everything works with your specific tool configuration.

### ***Common Mistakes and How the Skill Prevents Them***

**Mistake:** Providing generic answers when the skill needs specific values.

**Prevention:** The skill asks follow-up questions when answers are too vague. For example, if you say "we use Slack," it asks for the specific channel name or workspace ID.

**Mistake:** Assuming defaults that do not match the organization's reality.

**Prevention:** The skill searches knowledge MCPs first and only falls back to asking when data is not found. It does not assume any defaults without confirmation.

**Mistake:** Breaking the plugin by changing structural elements during customization.

**Prevention:** The skill only replaces content within files; it never renames files, directories, or changes `name` fields in manifests.

**Mistake:** Forgetting to connect MCP servers for identified tools.

**Prevention:** The skill automatically searches for and suggests MCP connections in Phase 4, ensuring tools are not just named but actually connected.

### *Comparing the Two Skills*

ASPECT	CREATE-COWORK-PLUGIN	COWORK-PLUGIN-CUSTOMIZER
Starting point	Nothing (build from scratch)	Existing plugin template
Primary activity	Designing components and structure	Replacing placeholders and configuring
Phases	5 (Discovery, Planning, Design, Implementation, Packaging)	4 (Gather Context, Create Todos, Complete Todos, Connect MCPs)
User interaction	High (design decisions at every phase)	Medium (mostly automated if knowledge MCPs connected)
Output	Net-new plugin	Customized version of existing plugin
When to use	Unique need, no existing plugin fits	Generic plugin exists, needs organization-specific adaptation
Skill complexity	Higher (encodes entire architecture)	Lower (focused on find-and-replace with context)

Both skills deliver the same final artifact type (a `.plugin` file), but the path to get there is very different.

# Part IV: Connectors and MCP Integration

---

## A Critical Distinction: This Plugin Has No MCP Servers

The cowork-plugin-management plugin does not connect to any external services itself. It has no `.mcp.json` file and no MCP server configurations. This is because the plugin management plugin operates on plugin files and directory structures, not on external tools.

However, this plugin extensively manages MCP configurations for other plugins. The customization workflow (Phase 4 of the cowork-plugin-customizer skill) includes MCP discovery, connection setup, and `.mcp.json` file generation.

## Understanding MCP Integration in the Plugin Management Context

When you customize a plugin using the cowork-plugin-customizer skill, one of the key outputs is a properly configured `.mcp.json` file that connects the plugin to the external tools your organization uses.

### *The MCP Discovery Workflow*

The skill follows this process in Phase 4:

#### Step 1: Identify Tools

Based on the customization points resolved in Phase 3, the skill has a list of tools your organization uses:

- Project tracker: Linear
- Chat: Slack
- Design: Figma
- Analytics: Amplitude
- CRM: HubSpot

#### Step 2: Search the MCP Registry

For each tool, the skill calls `search_mcp_registry(keywords=[ ... ])` using the category-to-keywords mapping:

```
{
  "keywords": ["linear", "tasks", "project management"]
}
```

The search returns up to 10 results, each with:

- `name` : Display name (e.g., "Linear")
- `description` : One-line description
- `tools` : List of tool names the MCP provides
- `url` : MCP endpoint URL
- `directoryUuid` : UUID for connection setup
- `connected` : Boolean indicating if already connected

### Step 3: Present Results and Gather Choices

The skill shows all matching MCPs with descriptions and asks which one you want to use. For well-known tools, the choice is usually obvious.

### Step 4: Connect if Needed

If the MCP is not already connected, the skill calls:

```
{
  "directoryUuids": ["uuid-from-search"]
}
```

This displays a Connect button in the UI. You complete OAuth or authentication, and the connection is established.

### Step 5: Update the Plugin's MCP Configuration

The skill reads the plugin's `plugin.json` to check for a custom `mcpServers` path:

```
{
  "mcpServers": "./config/servers.json"
}
```

If present, it updates that file. If not, it updates `.mcp.json` at the plugin root. If the `mcpServers` field points only to `.mcpb` files (bundled servers), the skill creates a new `.mcp.json` at the root.

The skill adds the MCP using the `url` from the search results:

```
{  
  "mcpServers": {  
    "linear": {  
      "type": "http",  
      "url": "https://mcp.linear.app/mcp"  
    },  
    "slack": {  
      "type": "http",  
      "url": "https://slack.mcp.claude.com/mcp"  
    }  
  }  
}
```

## ***MCP Server Types and Configuration Formats***

The skill understands three server types:

### **stdio — Local Process Servers**

Used for custom servers running on your local machine:

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

### **SSE — Hosted Servers with OAuth**

Used for cloud services that authenticate via OAuth:

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

## HTTP — REST API Servers

Used for REST APIs with token-based auth:

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

The skill chooses the appropriate type based on the MCP registry results and applies it correctly in the configuration file.

### *First-Party Integrations Exception*

Gmail, Google Calendar, and Google Drive are connected at the user level, not the plugin level. The skill does not add these to plugin `.mcp.json` files even if they are mentioned during customization.

### *Environment Variable Expansion*

The skill uses  `${CLAUDE_PLUGIN_ROOT}`  for all intra-plugin path references and  `${VAR_NAME}`  for user environment variables:

```
{  
  "mcpServers": {  
    "custom-api": {  
      "type": "http",  
      "url": "https://api.example.com/mcp",  
      "headers": {  
        "Authorization": "Bearer ${MY_API_TOKEN}"  
      }  
    }  
  }  
}
```

When the skill adds environment variable references, it documents them in the plugin's README.

## The Category-to-Keywords Mapping

The skill maintains a comprehensive mapping in `references/mcp-servers.md` :

CATEGORY	PLACEHOLDER	KEYWORDS
Project management	<code>~project tracker</code>	asana, jira, linear, monday, tasks
Software coding	<code>~source control</code>	github, gitlab, bitbucket, code
Chat	<code>~chat</code>	slack, teams, discord
Documents	<code>~documents</code>	google docs, notion, confluence
Calendar	<code>~calendar</code>	google calendar, calendar
Email	<code>~email</code>	gmail, outlook, email
Design/graphics	<code>~design</code>	figma, sketch, design
Analytics/BI	<code>~analytics</code>	datadog, grafana, analytics
CRM	<code>~~CRM</code>	salesforce, hubspot, crm
Wiki/knowledge base	<code>~~wiki</code>	notion, confluence, outline, wiki
Data warehouse	<code>~~data warehouse</code>	bigquery, snowflake, redshift
Conversation intelligence	<code>~~call recording</code>	gong, chorus, call recording

This mapping ensures efficient MCP discovery regardless of how placeholders are named in the plugin template.

## How MCP Configuration Differs from Typical Plugins

Most plugins include an `.mcp.json` file with pre-configured connections to specific services. For example, the marketing plugin comes with connections to Slack, Canva, Figma, HubSpot, Amplitude, Notion, Ahrefs, Similarweb, and Klaviyo.

The plugin management plugin does not have such a file because:

1. It does not need external service integrations to perform its function (creating and customizing plugins)
2. It operates on local files and directory structures
3. Its purpose is to **create** `.mcp.json` files for other plugins, not to use them itself

This is an important architectural distinction. The plugin management plugin is a development tool, not a domain-specific workflow plugin.

---

# Part V: Workflows and Use Cases

This section provides detailed, step-by-step workflows for common plugin management tasks. Each workflow includes expected inputs, process steps, decision points, and outputs.

## Workflow 1: Creating a Plugin from Scratch

**Scenario:** You are a compliance manager at a pharmaceutical company. You want to create a plugin that helps the marketing team create promotional materials that comply with FDA regulations and get them through Medical-Legal-Regulatory (MLR) review.

### Step 1: Initiate the Conversation

Start a conversation in Claude Cowork and express your intent:

*"I want to create a plugin for pharmaceutical marketing compliance. It should help our team create promotional content that follows FDA rules and passes MLR review."*

The create-cowork-plugin skill triggers and enters Discovery phase.

### Step 2: Discovery Phase

The skill asks clarifying questions:

- **What should this plugin do?** → Help create compliant promotional materials for pharma products and guide them through MLR review
- **Who will use it?** → Marketing team members creating content for prescription products
- **What tools does it integrate with?** → Our MLR review system and regulatory database
- **Similar workflows?** → We have a manual checklist we currently use

You provide answers, and the skill summarizes:

*"This plugin will provide regulatory compliance guidance for pharmaceutical promotional content, automate fair balance checks, generate MLR review packages, and connect to your regulatory database. Is that correct?"*

You confirm.

## Step 3: Component Planning

The skill proposes a component plan:

COMPONENT	COUNT	PURPOSE
Skills	2	FDA promotional rules, MLR review process
Commands	3	/mlr-package, /fair-balance-check, /compliance-review
Agents	0	Not needed
Hooks	1	Validate all content writes for compliance
MCP	1	Connect to regulatory database

You confirm the plan.

## Step 4: Design Phase

The skill asks detailed questions for each component:

For the regulatory compliance skill:

- **What topics trigger this?** → Creating promotional content, reviewing materials for compliance, checking fair balance
- **What knowledge does it cover?** → FDA Prescription Drug Advertising rules, fair balance requirements, off-label restrictions
- **Reference files?** → Yes, detailed FDA guidance and our internal checklist

For the /mlr-package command:

- **What inputs?** → Path to the content file
- **What tools?** → Read (to read the file), Write (to generate the package), and the regulatory database MCP
- **Output?** → A formatted MLR review package with all required sections

For the PreToolUse hook:

- **Which tools should it monitor?** → Write and Edit
- **What should it check?** → Verify no off-label claims, ensure fair balance is present if efficacy claims are made
- **Prompt-based or command-based?** → Prompt-based for nuanced evaluation

For the regulatory database MCP:

- **Server type?** → HTTP (REST API with token authentication)

- **Authentication?** → Bearer token in header
- **Required environment variables?** → REGULATORY\_DB\_API\_TOKEN

You provide answers, and the skill confirms the specification.

### Step 5: Implementation

The skill creates all files:

```

pharma-compliance/
├── .claude-plugin/
│   └── plugin.json
└── commands/
    ├── mlr-package.md
    ├── fair-balance-check.md
    └── compliance-review.md
└── skills/
    ├── regulatory-compliance/
    │   ├── SKILL.md
    │   └── references/
    │       ├── fda-prescription-drug-advertising.md
    │       └── mlr-checklist.md
    └── mlr-review-process/
        ├── SKILL.md
        └── references/
            └── review-workflow.md
└── hooks/
    └── hooks.json
└── .mcp.json
└── README.md

```

The skill shows you a preview of key files and asks if you want any changes.

### Step 6: Packaging

You confirm the plugin looks good. The skill:

1. Runs `claude plugin validate pharma-compliance/.claude-plugin/plugin.json`
2. Fixes a warning about a missing description field
3. Packages the plugin: `cd pharma-compliance && zip -r /tmp/pharma-compliance.plugin . && cp /tmp/pharma-compliance.plugin /path/to/outputs/`
4. Delivers `pharma-compliance.plugin` to the outputs directory

You see a rich preview in the chat with the plugin structure. You click Accept, and the plugin is installed.

## Step 7: Testing

You test the plugin:

- Type `/mlr-package sample-email.md` — it generates a properly formatted MLR review package
- Try writing promotional content — the PreToolUse hook validates it and flags an off-label claim before allowing the write
- Ask "What are the fair balance requirements?" — the regulatory-compliance skill activates and provides detailed guidance

The plugin works as expected.

**Outcome:** A fully functional compliance plugin tailored to your organization's needs, created in a single session without writing any code manually.

---

## Workflow 2: Customizing a Generic Plugin Template

**Scenario:** You downloaded the generic "engineering" plugin, which uses `~` placeholders for tools. You need to customize it for your team, which uses Linear for project management, GitHub for code, Slack for chat, and Datadog for monitoring.

### Step 1: Initiate Customization

*"I need to customize the engineering plugin for my team's tools."*

The cowork-plugin-customizer skill triggers.

### Step 2: Gather Context (Automated)

The skill searches your connected knowledge sources:

**From Slack:**

- Finds mentions of "Linear" when discussing tickets
- Finds "GitHub" when discussing pull requests
- Identifies "#engineering" as the team channel
- Sees references to "Datadog dashboards"

**From documents:**

- Finds an onboarding doc mentioning "Linear workspace: Acme Engineering"

- Finds a runbook referencing "Datadog API key setup"

The skill records all findings.

### Step 3: Create Todo List

The skill scans the plugin:

```
grep -rn '~~\w' /path/to/engineering --include='*.md' --include='*.json'
```

Finds placeholders:

- `~project tracker` (in 5 files)
- `~source control` (in 3 files)
- `~chat` (in 2 files)
- `~monitoring` (in 1 file)
- `~your-workspace-id` (in 1 file)

Creates todo list:

- Learn what project tracking tool your team uses
- Learn what source control system your team uses
- Learn what chat platform your team uses
- Learn what monitoring tool your team uses
- Identify your Linear workspace ID

### Step 4: Complete Todo Items

For each item, the skill checks Phase 2 findings:

- **Project tracker** → Found "Linear" in Slack and docs → Replaces `~project tracker` with "Linear" automatically
- **Source control** → Found "GitHub" in Slack → Replaces `~source control` with "GitHub"
- **Chat** → Found "Slack" and "#engineering" → Replaces `~chat` with "Slack"
- **Monitoring** → Found "Datadog" → Replaces `~monitoring` with "Datadog"
- **Workspace ID** → Found "Acme Engineering" in docs but no UUID → Uses AskUserQuestion:

*"What is your Linear workspace ID? I found references to 'Acme Engineering' but need the workspace UUID for API calls."*

You provide: `acme-engineering-a1b2c3`

The skill replaces `~your-workspace-id` with `acme-engineering-a1b2c3`.

## Step 5: Connect MCPs

The skill searches for MCP servers:

### Linear:

- Searches: `search_mcp_registry(keywords=["linear", "tasks", "project management"])`
- Finds Linear MCP (not connected)
- Calls `suggest_connectors(directoryUuids=["linear-uuid"])`
- You authenticate via OAuth
- Adds to `.mcp.json`:

```
{  
  "mcpServers": {  
    "linear": {  
      "type": "sse",  
      "url": "https://mcp.linear.app/sse"  
    }  
  }  
}
```

### GitHub:

- Searches and finds GitHub MCP (already connected at user level)
- Adds to `.mcp.json`:

```
{  
  "github": {  
    "type": "http",  
    "url": "https://api.githubcopilot.com/mcp/"  
  }  
}
```

### Slack:

- Searches and finds Slack MCP (already connected)
- Adds to `.mcp.json`

### Datadog:

- Searches and finds Datadog MCP (not connected)
- You connect it

- Adds to `.mcp.json` with environment variables:

```
{  
  "datadog": {  
    "type": "http",  
    "url": "https://api.datadoghq.com/mcp",  
    "headers": {  
      "DD-API-KEY": "${DATADOG_API_KEY}",  
      "DD-APPLICATION-KEY": "${DATADOG_APP_KEY}"  
    }  
  }  
}
```

## Step 6: Summary and Packaging

The skill presents a summary:

```
## From searching Slack  
- You use Linear for project management  
- You use GitHub for source control  
- You use Slack for team chat (#engineering channel)  
- You use Datadog for monitoring  
  
## From searching documents  
- Linear workspace is "Acme Engineering"  
  
## From your answers  
- Linear workspace ID is acme-engineering-a1b2c3  
  
## MCPs connected during setup  
- Linear (authenticated via OAuth)  
- Datadog (authenticated via OAuth)  
  
## MCPs already connected  
- GitHub (already configured at user level)  
- Slack (already configured at user level)  
  
## Required environment variables  
Add these to your environment:  
- DATADOG_API_KEY  
- DATADOG_APP_KEY
```

The skill packages the plugin:

```
cd engineering && zip -r /tmp/engineering.plugin . -x "setup/*" && cp /tmp/engineering
```

You receive `engineering.plugin` as a rich preview and accept it.

**Outcome:** A fully customized engineering plugin with all tool placeholders replaced, MCP servers connected, and organization-specific values configured, delivered in minutes with minimal manual input.

---

## Workflow 3: Forking and Extending an Existing Plugin

**Scenario:** The marketing plugin is great, but you are in the pharmaceutical industry and need to add regulatory compliance features. You decide to fork it, add pharma-specific skills, and distribute the forked version to your team.

### Step 1: Copy the Plugin

You copy the marketing plugin directory:

```
cp -r marketing/ marketing-pharma/
```

### Step 2: Update Plugin Identity

Edit `marketing-pharma/.claude-plugin/plugin.json`:

```
{  
  "name": "marketing-pharma",  
  "version": "1.0.0",  
  "description": "Marketing plugin adapted for pharmaceutical and life sciences teams,  
  "author": {  
    "name": "Your Organization"  
  }  
}
```

### Step 3: Add Pharma-Specific Skills

Initiate a conversation with Claude:

*"I forked the marketing plugin to create marketing-pharma. I want to add two new skills: regulatory-compliance and mlr-review-process. Can you help me create them?"*

The create-cowork-plugin skill partially triggers (you are not creating a full plugin, just components). You guide the conversation to create the skills:

### regulatory-compliance skill:

```
--  
name: regulatory-compliance  
description: >  
    Pharmaceutical and medical device regulatory compliance for marketing  
    materials. Use when reviewing promotional content for compliance,  
    preparing materials for MLR review, checking fair balance requirements,  
    or ensuring FDA/EMA compliance in marketing communications.  
--  
  
# Regulatory Compliance for Life Sciences Marketing  
  
## Fair Balance Requirements  
  
All promotional materials must present a fair balance between efficacy  
claims and risk information ...  
  
[ ... detailed content ... ]
```

### mlr-review-process skill:

```
--  
name: mlr-review-process  
description: >  
    Medical-Legal-Regulatory (MLR) review workflow and requirements. Use  
    when preparing materials for MLR review, understanding review cycles,  
    or coordinating with Medical Affairs, Legal, and Regulatory teams.  
--  
  
# MLR Review Process  
  
## Review Workflow  
  
1. **Draft creation** – Marketing creates content using approved claims  
2. **Medical review** – Medical affairs verifies clinical accuracy  
3. **Legal review** – Legal checks regulatory compliance and risk  
4. **Regulatory review** – Regulatory confirms alignment with approved labeling  
5. **Final approval** – All three functions sign off  
  
[ ... detailed content ... ]
```

You create these skills in `marketing-pharma/skills/`.

### Step 4: Add Pharma-Specific Commands

You add `/mlr-review` and `/congress-plan` commands:

```
---  
description: Generate an MLR review package for promotional content  
argument-hint: <content-file-path>  
---
```

Read the content at `@$1` and generate a complete MLR review package.

Include these sections:

1. **Content Classification** – branded promotional, unbranded, reminder
2. **Claims Analysis** – list all efficacy and safety claims with references
3. **Fair Balance Check** – verify risk info is appropriately prominent
4. **Indication Accuracy** – compare to approved prescribing information
5. **Reviewer Checklist** – Medical, Legal, and Regulatory sign-off sections

[ ... detailed instructions ... ]

## Step 5: Modify Existing Skills

Edit `skills/campaign-planning/SKILL.md` to add pharmaceutical campaign types:

```
## Life Sciences Campaign Types  
  
### Disease Awareness Campaigns  
- Unbranded – educate about the disease without mentioning the product  
- Branded – connect the disease to your product's indication  
  
### Congress and Conference Marketing  
- Pre-congress – abstract submission support, booth planning  
- At-congress – booth strategy, KOL meetings, symposium support  
- Post-congress – follow-up campaigns, content repurposing  
  
[ ... more pharma-specific content ... ]
```

## Step 6: Update MCP Configuration

Add pharmaceutical-specific MCPs to `.mcp.json`:

```
{  
  "mcpServers": {  
    "slack": { "type": "http", "url": "https://mcp.slack.com/mcp" },  
    "canva": { "type": "http", "url": "https://mcp.canva.com/mcp" },  
    "figma": { "type": "http", "url": "https://mcp.figma.com/mcp" },  
    "hubspot": { "type": "http", "url": "https://mcp.hubspot.com/anthropic" },  
    "amplitude": { "type": "http", "url": "https://mcp.amplitude.com/mcp" },  
    "notion": { "type": "http", "url": "https://mcp.notion.com/mcp" },  
    "pubmed": { "type": "http", "url": "https://api.ncbi.nlm.nih.gov/mcp" },  
    "clinicaltrials": { "type": "http", "url": "https://clinicaltrials.gov/api/mcp" }  
  }  
}
```

## Step 7: Update Documentation

Edit `README.md` to reflect the pharma-specific additions and update `CONNECTORS.md` if needed.

## Step 8: Package and Distribute

Package the forked plugin:

```
cd marketing-pharma && zip -r /tmp/marketing-pharma.plugin . && cp /tmp/marketing-phar...
```

Distribute `marketing-pharma.plugin` to your team via your internal plugin repository or file sharing.

**Outcome:** A pharmaceutical-specific marketing plugin that retains all the value of the original marketing plugin while adding industry-specific compliance, review processes, and campaign types.

---

## Workflow 4: Adding Company-Specific Customizations via CLAUDE.md

**Scenario:** You want to layer your company's brand voice rules on top of the marketing plugin without modifying the plugin itself. This approach lets you receive marketing plugin updates without merge conflicts.

### Step 1: Create a Company Brand Voice Skill

In your project directory, create:

```

your-project/
└── .claude/
    └── skills/
        └── acme-brand-voice/
            ├── SKILL.md
            └── references/
                └── voice-examples.md
CLAUDE.md

```

### SKILL.md content:

```

---
name: acme-brand-voice
description: >
  Acme Corporation brand voice, tone, and messaging guidelines. Use when
  creating any marketing content for Acme, reviewing content for brand
  alignment, adapting tone for different audiences, or checking terminology
  and style compliance.
---

# Acme Brand Voice

## Voice Attributes

- **Confident but not arrogant**: We know our product is excellent. We let
  results speak. We never disparage competitors.
- **Clear and direct**: Short sentences. Active voice. No jargon without
  explanation.
- **Warm but professional**: We use contractions. We avoid slang. We never
  use ALL CAPS for emphasis.

## Terminology Rules

| Use This | Not This | Reason |
| --- | --- | --- |
| Acme Platform | the platform, our tool | Always capitalize product name |
| clients | customers, users | Reflects relationship |
| team members | employees, workers | Company culture preference |

[ ... more brand rules ... ]

```

### Step 2: Add Routing Rules to CLAUDE.md

Edit your project's `CLAUDE.md`:

### ## Brand Voice Rules

When using any marketing plugin command or skill, always apply the acme-brand-voice skill in addition to the marketing plugin's brand-voice skill. Acme-specific rules override generic rules when they conflict.

When the marketing plugin asks about brand voice, tone, or messaging, incorporate Acme's voice attributes and terminology rules from the acme-brand-voice skill.

## Step 3: Test the Integration

Run a marketing plugin command:

*/campaign-plan for new product launch*

Claude generates a campaign brief. You notice it:

- Uses "clients" instead of "customers" (Acme rule)
- Uses "Acme Platform" with capital letters (Acme rule)
- Applies confident, clear tone (Acme rule)
- Also incorporates campaign structure from marketing plugin's campaign-planning skill

Both skills are working together.

## Step 4: Receive Marketing Plugin Updates

When the marketing plugin releases version 2.0 with new features, you:

1. Install the updated marketing plugin
2. Your acme-brand-voice skill remains unchanged (it is in your project, not the plugin)
3. Your CLAUDE.md routing rules remain unchanged
4. The new marketing plugin features automatically work with Acme brand voice layered on top

**Outcome:** Company-specific customizations that persist across plugin updates, with no merge conflicts or manual file editing.

# Part VI: Best Practices and Guidelines

---

This section distills the accumulated wisdom from plugin creation and customization into actionable guidelines for administrators, team leads, and power users.

## When to Create vs. Customize vs. Extend

Create a new plugin from scratch when:

- No existing plugin covers your use case
- The need is highly specific to your organization or industry
- You have deep domain expertise to encode
- You want complete control over structure and components

Customize an existing plugin when:

- A generic plugin template exists with ~ placeholders
- The plugin's structure and workflows fit your needs
- You just need to swap tool references and configure values
- You want to deploy quickly with minimal custom development

Extend an existing plugin when:

- A plugin does 70-80% of what you need
- You want to add features without altering the original
- You may want to receive upstream updates
- You are adding company-specific or industry-specific layers

Decision matrix:

SITUATION	RECOMMENDED APPROACH	RATIONALE
Marketing team needs pharma compliance features	Extend (fork marketing plugin)	Core marketing workflows stay intact; add industry layer
Need a plugin for internal procurement workflows	Create from scratch	No generic template exists for this domain
Engineering plugin exists with ~project tracker	Customize	Perfect fit; just needs tool-specific configuration

SITUATION	RECOMMENDED APPROACH	RATIONALE
Want to add company brand voice to any marketing plugin	Extend (CLAUDE.md + complementary skill)	Lightest touch; survives upstream updates

## Structuring Skills for Progressive Disclosure

Skills are the most knowledge-dense component type. They can easily become bloated and hard to maintain. Follow this three-tier structure:

### Tier 1: Metadata (always in context)

- `name` field: Short, descriptive, kebab-case
- `description` field: Third-person, with specific trigger phrases in quotes
- Target size: ~100 words

### Tier 2: SKILL.md Body (loads when triggered)

- Core frameworks, checklists, and processes
- High-level guidance and decision trees
- References to files in `references/` for deeper detail
- Target size: 1,500-2,000 words (max 3,000)

### Tier 3: Bundled Resources (loads on demand)

- `references/`: Detailed guides, long tables, specifications
- `examples/`: Sample configurations, templates
- `scripts/`: Utility scripts for validation or generation
- `assets/`: Templates, images, supplementary materials
- Size: Unlimited

Example of good progressive disclosure (from the create-cowork-plugin skill):

```

create-cowork-plugin/
└── SKILL.md                               # 2,500 words: workflow overview, best practice
    └── references/
        └── component-schemas.md           # 4,000 words: exhaustive field specifications
            └── example-plugins.md          # 2,000 words: three complete examples
                └── [no examples in this skill]

```

The SKILL.md body provides enough context to guide the workflow. The component schemas provide deep reference detail when implementing. Users are never overwhelmed with information they do not need yet.

## Writing Skill Descriptions That Trigger Reliably

The `description` field in skill frontmatter determines when Claude loads the skill. Poor descriptions lead to skills failing to trigger when needed or triggering at inappropriate times.

Good description anatomy:

1. **Third-person framing:** "This skill should be used when..." or "Use this skill when..."
2. **Specific trigger phrases in quotes:** Actual words users would say
3. **Multiple variations:** Formal and informal phrasings
4. **Context indicators:** Situations or tasks that require the skill

Example (good):

```
description: >
  This skill should be used when the user asks 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". Also
  use when the user needs to understand plugin architecture, component
  types, or the plugin creation workflow.
```

This description:

- Lists seven different phrasings users might use
- Includes both action-oriented ("create") and understanding-oriented ("understand architecture") triggers
- Covers formal ("develop a plugin") and informal ("make a plugin") language

Example (bad):

```
description: Plugin creation and management
```

This description:

- Too generic; could trigger on any mention of plugins
- No specific user phrases
- No context about when it is appropriate

Trigger phrase discovery process:

1. Imagine you are a user with the problem this skill solves. What would you say?
2. Write down 5-10 variations (short and long, formal and casual)
3. Include those exact phrases in quotes in the description
4. Add situational context ("when reviewing content," "when preparing for deployment")

## Command Writing: Instructions for Claude, Not the User

Commands are Markdown files that Claude reads and follows. The most common mistake is writing them like documentation for end users.

**Bad command structure (reads like documentation):**

```
---  
description: Generate a campaign plan  
---  
  
# Campaign Planning Command  
  
This command helps you create a comprehensive campaign plan. It will ask  
you questions about your goals, audience, and budget, then generate a  
structured brief.  
  
To use this command, provide the campaign objective as an argument.
```

This reads like a user guide. Claude does not need to know how the command helps users; Claude is the one executing it.

**Good command structure (reads like instructions):**

```
---  
description: Generate a campaign plan  
argument-hint: <campaign objective>  
---  
  
Generate a comprehensive campaign plan.  
  
## Gather Inputs  
  
Ask the user for:  
1. Campaign goal (if not provided in $ARGUMENTS)  
2. Target audience  
3. Timeline  
4. Budget range (optional)  
  
## Output Structure  
  
Present a campaign brief with these sections:  
1. Campaign Overview  
2. Target Audience  
3. Key Messages  
4. Channel Strategy  
5. Content Calendar  
6. Success Metrics
```

This tells Claude exactly what to do: gather inputs, structure output, include specific sections. It is imperative and action-oriented.

#### Writing style rules for commands:

- Use imperative verbs: "Generate," "Analyze," "Check," "Ask," "Present"
- Structure with headings: `## Inputs`, `## Process`, `## Output`
- Specify exact output formats when precision matters
- Use bulleted or numbered lists for sequences of actions
- Reference files and variables directly: `@$1`,  `${CLAUDE_PLUGIN_ROOT}/templates/brief.md`

## Managing MCP Server Configurations

When creating or customizing plugins, MCP server setup is a common point of failure. Follow these guidelines:

#### Use environment variables for credentials:

```
{
  "mcpServers": {
    "custom-api": {
      "type": "http",
      "url": "https://api.example.com/mcp",
      "headers": {
        "Authorization": "Bearer ${API_TOKEN}"
      }
    }
  }
}
```

Never hardcode tokens or passwords in plugin files.

Always use ``${CLAUDE_PLUGIN_ROOT}`` for local paths:

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

This ensures portability across systems.

**Document required environment variables in README:**

```
## Setup

This plugin requires the following environment variables:

- `HUBSPOT_API_KEY` - Your HubSpot private app access token
- `DATADOG_API_KEY` - Datadog API key
- `DATADOG_APP_KEY` - Datadog application key

Add these to your shell profile or `*.env` file before using the plugin.
```

**Choose the right server type:**

USE CASE	SERVER TYPE	REASON
Major SaaS with OAuth	SSE	Handles auth automatically

USE CASE	SERVER TYPE	REASON
REST API with token	HTTP	Simple header-based auth
Custom local script	stdio	Runs locally, full control
File system operations	stdio	Needs local machine access

### Test MCP connections immediately after configuration:

After adding an MCP server to `.mcp.json`, test it:

1. Install the plugin
2. Try a command or skill that uses the MCP server
3. Verify the connection works and tools are accessible
4. Check logs if authentication fails

## Version Management and Distribution

### Semantic versioning for plugins:

Use MAJOR.MINOR.PATCH:

- **MAJOR**: Incompatible changes (remove commands, change skill names, break existing workflows)
- **MINOR**: Add features (new commands, new skills, new MCP servers)
- **PATCH**: Bug fixes, documentation updates, small improvements

### Example progression:

- `0.1.0` — Initial version
- `0.2.0` — Added two new skills
- `0.2.1` — Fixed typo in command description
- `1.0.0` — Stable release, ready for broad distribution
- `1.1.0` — Added MCP server integration for new tool
- `2.0.0` — Renamed skills for clarity (breaking change)

### Changelog maintenance:

Maintain a `CHANGELOG.md` in the plugin root:

```
# Changelog

## [1.1.0] - 2025-03-15

### Added
- New `/congress-plan` command for conference marketing
- Integrated PubMed MCP server for literature search

### Changed
- Updated fair balance checklist with 2025 FDA guidance

## [1.0.0] - 2025-01-10

### Added
- Initial stable release
- Regulatory compliance skill
- MLR review process skill
- Three commands: /mlr-package, /fair-balance-check, /compliance-review
```

## Distribution channels:

- **Internal plugin repository:** Host `.plugin` files in a shared directory or internal web server
- **Version control:** Store plugin source in Git; tag releases
- **Direct distribution:** Email `.plugin` files to users (works for small teams)
- **Public registry:** If building for external use, consider publishing to a plugin marketplace (if one exists)

## Update strategy:

When releasing updates:

1. Increment version number in `plugin.json`
2. Update `CHANGELOG.md`
3. Package as `.plugin` file
4. Announce to users with migration notes if breaking changes exist
5. Maintain previous major version for 3-6 months to allow gradual migration

## Testing and Validation

Before distributing a plugin, test thoroughly:

### Validation checks:

1. **Structural validation:** Run `claude plugin validate /path/to/plugin.json` — fix all errors, address warnings

2. **Skill triggering:** Test that skills activate on expected phrases and do not activate on unrelated topics
3. **Command execution:** Run every command with typical inputs; verify outputs are correct
4. **MCP connections:** Ensure all MCP servers connect and tools are accessible
5. **Hook behavior:** Trigger hooks (if present) and verify they behave as expected

Test scenarios by persona:

PERSONA	TEST SCENARIO
New user	Install plugin, try basic commands, verify onboarding experience
Power user	Use advanced features, combine commands, test edge cases
Administrator	Install across multiple users, verify MCP auth works for all, check permissions

Common issues to test for:

- Do skills fail to trigger when they should?
- Do commands assume context that may not exist?
- Are MCP server authentication errors clear and actionable?
- Do hooks block operations they should allow, or allow operations they should block?
- Are README instructions sufficient for a new user to set up?

# Part VII: Troubleshooting and Advanced Topics

## Troubleshooting Common Issues

### *Issue: Skill Does Not Trigger When Expected*

**Symptoms:** You mention the topic the skill covers, but Claude does not load the skill.

#### 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

#### Diagnostics:

Check skill metadata:

```
cat skills/my-skill/SKILL.md
```

#### Verify:

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

#### Solution:

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***

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

**Causes:**

1. `allowed-tools` frontmatter field restricts tools
2. Tool name is misspelled in `allowed-tools`
3. MCP tool is not correctly named

**Diagnostics:**

Check command frontmatter:

```
cat commands/my-command.md
```

Look for:

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

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

**Solution:**

Add the required tool to `allowed-tools`:

```
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***

**Symptoms:** Commands or skills that use an MCP server produce authentication errors or connection failures.

**Causes:**

1. MCP server URL is incorrect
2. Required environment variables are not set
3. OAuth authentication not completed
4. MCP server is down or unreachable

### Diagnostics:

Check `.mcp.json`:

```
cat .mcp.json
```

Verify the URL and server type are correct.

Check environment variables:

```
echo $API_TOKEN
```

If empty, the variable is not set.

### Solution:

For environment variable issues, set the required variable:

```
export API_TOKEN="your-token-here"
```

Add it to your shell profile (`.bashrc`, `.zshrc`) to persist.

For OAuth issues, reinstall the plugin — the OAuth flow should trigger during installation.

For URL issues, verify the URL against the MCP server's documentation.

---

### ***Issue: Hook Blocks Operations Incorrectly***

**Symptoms:** A PreToolUse or PostToolUse hook blocks operations that should be allowed.

### Causes:

1. Hook prompt is too strict or poorly worded
2. Matcher pattern is too broad
3. Command-based hook script has a bug

## Diagnostics:

Check `hooks/hooks.json`:

```
{  
  "PreToolUse": [  
    {  
      "matcher": "Write|Edit",  
      "hooks": [  
        {  
          "type": "prompt",  
          "prompt": "Block all file writes that do not follow naming conventions.",  
          "timeout": 30  
        }  
      ]  
    }  
  ]  
}
```

Test the hook logic manually: Would the prompt block this specific operation?

## Solution:

Refine the hook prompt to be more specific:

```
{  
  "prompt": "Check if this file write follows the naming convention (kebab-case, .md etc)",  
  "timeout": 30  
}
```

Or narrow the matcher to specific files:

```
{  
  "matcher": "Write.*\\.md"  
}
```

---

## *Issue: Plugin Validation Fails*

**Symptoms:** Running `claude plugin validate /path/to/plugin.json` produces errors or warnings.

## Common Errors:

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

### Solution:

Fix the specific issue identified in the error message, then rerun validation.

---

## Advanced Customization Scenarios

### *Scenario 1: Multi-Tenant Plugin with Organization-Specific Variants*

**Challenge:** You have multiple clients, each needing the same plugin with different tool configurations.

#### Approach:

1. Create a master template plugin with ~ placeholders
2. For each client, run the customizer skill to generate a client-specific variant
3. Name each variant distinctly (e.g., marketing-acme, marketing-globex )
4. Distribute the appropriate variant to each client

This is essentially the same workflow as regular customization, repeated for each organization.

---

### *Scenario 2: Conditional Skill Loading Based on Project Type*

**Challenge:** You want a skill to load only when working on certain types of projects (e.g., pharma projects vs. consumer projects).

## Approach:

Use project-level `CLAUDE.md` files to create routing rules:

### In pharma project's CLAUDE.md:

```
## Context

This is a pharmaceutical product. Always apply regulatory-compliance
and mlr-review-process skills when creating marketing content.
```

### In consumer project's CLAUDE.md:

```
## Context

This is a consumer product. Use standard marketing workflows without
regulatory compliance checks.
```

The `CLAUDE.md` context influences which skills Claude activates.

---

## ***Scenario 3: Dynamic MCP Configuration Based on Environment***

**Challenge:** Development, staging, and production environments use different MCP server endpoints.

## Approach:

Use environment variables for URLs:

```
{
  "mcpServers": {
    "api-service": {
      "type": "http",
      "url": "${API_MCP_URL}",
      "headers": {
        "Authorization": "Bearer ${API_TOKEN}"
      }
    }
  }
}
```

Set the variable in each environment:

```
# Development
export API_MCP_URL="https://dev-api.example.com/mcp"

# Production
export API_MCP_URL="https://api.example.com/mcp"
```

The plugin adapts automatically based on the environment.

---

## Extending the Plugin Management Plugin Itself

The plugin management plugin is itself a plugin. Can you extend it?

Yes, but with caveats.

### Approach A: Add a complementary skill in CLAUDE.md

Create a project-level skill that supplements the plugin creation workflow:

```
your-project/
└── .claude/
    └── skills/
        └── custom-plugin-templates/
            ├── SKILL.md
            └── references/
                └── org-specific-templates.md
CLAUDE.md
```

**SKILL.md:**

```
---
```

```
name: custom-plugin-templates
description: >
    Organization-specific plugin templates and component patterns. Use
    when creating plugins for internal use, referencing company-standard
    structures, or applying organizational conventions to new plugins.
---

# Custom Plugin Templates for Acme Corp

## Standard Acme Plugin Structure

All Acme plugins should include:
1. A `LICENSE` file (MIT license)
2. A `SECURITY.md` file with contact information
3. A `/.github/` directory with issue templates

[ ... more organization-specific conventions ... ]
```

#### CLAUDE.md:

```
## Plugin Creation Rules

When using the create-cowork-plugin skill, also apply the
custom-plugin-templates skill to ensure all Acme-specific conventions
are followed.
```

#### Approach B: Fork the plugin management plugin

Copy the plugin, rename it (e.g., `cowork-plugin-management-acme`), and add organization-specific templates, examples, or workflow variations.

This is more invasive but gives complete control.

**Recommendation:** Use Approach A unless you have significant divergence from the standard workflow.

---

## Plugin Lifecycle Management

As plugins evolve, you need strategies for managing updates, deprecations, and migrations.

#### Update Strategy:

1. **Minor updates** (new features, no breaking changes): Distribute and encourage adoption; old version continues to work
2. **Major updates** (breaking changes): Announce 3-6 months in advance; provide migration guide; maintain old version during transition period
3. **Deprecation:** Mark plugin as deprecated in `plugin.json` and README; provide replacement recommendations

### Migration Guide Template:

```
# Migration Guide: v1.x to v2.0

## Breaking Changes

### Renamed Skills
- `old-skill-name` → `new-skill-name`

### Removed Commands
- `/old-command` – Use `/new-command` instead

### Changed MCP Configuration
- `mcp.json` now requires environment variable `NEW_VAR`

## Migration Steps

1. Uninstall version 1.x
2. Set environment variable: `export NEW_VAR="value"`
3. Install version 2.0
4. Update any CLAUDE.md references from old skill names to new names
5. Test all critical workflows

## Support

Old version (1.x) will be supported until 2025-12-31. After that,
only 2.x will receive updates.
```

## Performance Considerations

### Skill Size and Load Time:

Large skills slow down context loading. Keep SKILL.md bodies under 3,000 words. Move detailed content to `references/`.

### Number of Components:

Plugins with dozens of skills or commands can become unwieldy. Consider splitting into multiple focused plugins:

- `marketing-core` — foundational skills and commands
- `marketing-pharma` — pharmaceutical-specific extensions
- `marketing-analytics` — analytics and reporting focus

#### MCP Connection Overhead:

Each MCP server connection adds latency. Only include MCP servers the plugin actually uses.

---

## Security and Access Control

#### Credentials Management:

Never hardcode credentials in plugin files. Always use environment variables:

```
{  
  "headers": {  
    "Authorization": "Bearer ${API_TOKEN}"  
  }  
}
```

#### Tool Restrictions:

Use `allowed-tools` in commands to limit what Claude can do:

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

This prevents accidental file modifications during read-only operations.

#### Hook-Based Policy Enforcement:

Use PreToolUse hooks to enforce security policies:

```
{  
  "PreToolUse": [  
    {  
      "matcher": "Write|Edit|Bash",  
      "hooks": [  
        {  
          "type": "prompt",  
          "prompt": "Verify this operation does not write to protected directories (/etc, /var)",  
          "timeout": 30  
        }  
      ]  
    }  
  ]  
}
```

### Access Control for Sensitive MCPs:

For MCP servers that access sensitive data (CRM, financial systems), restrict plugin distribution to authorized users only.

---

## Future-Proofing Your Plugins

### Use Semantic Versioning:

Clear version numbers communicate compatibility and help users plan upgrades.

### Document Extensively:

Detailed README, CHANGELOG, and inline comments make plugins maintainable over time.

### Avoid Hard-Coded Assumptions:

Use environment variables and configuration files rather than hardcoding values.

### Design for Extension:

Structure skills and commands so new capabilities can be added without rewriting existing content.

### Maintain Backward Compatibility:

When possible, deprecate rather than remove features. Provide migration paths.

---

# Conclusion

---

The cowork-plugin-management plugin is the foundational tool for extending Claude Cowork. It encodes the complete plugin architecture, provides guided workflows for creation and customization, and delivers production-ready `.plugin` files that can be distributed across teams and organizations.

Whether you are building a net-new plugin from scratch or adapting a generic template to your organization's specific tools, the plugin management plugin provides the structure, guidance, and automation to make the process efficient and reliable.

Key takeaways:

1. **Two core skills drive everything:** `create-cowork-plugin` for building from scratch, `cowork-plugin-customizer` for adapting templates
2. **No commands by design:** Plugin management is conversational, not command-driven
3. **MCP integration is central:** The customizer skill discovers, connects, and configures MCP servers automatically
4. **Progressive disclosure:** Structure skills with lean core content and detailed reference files to manage context efficiently
5. **Start small, iterate:** Begin with minimal viable components and expand based on real usage

As the plugin ecosystem grows, the plugin management plugin will be the tool administrators, team leads, and power users rely on to create the custom capabilities that make Claude Cowork truly valuable for their specific contexts.

---

**AGENT\_REPORT\_START**

Plugin: cowork-plugin-management

Version: 0.2.1

Files analyzed: 8

Handbook word count: Approximately 16,500 words

Start time: 1771068420

End time: [will be calculated]

Estimated token usage: Approximately 48,000 input + 18,000 output = 66,000 total

**AGENT\_REPORT\_END**

# RationalEyes.ai

contact@rationaleyes.ai

Intelligent Automation for Knowledge Work

The Complete Cowork Plugin Management Handbook — Version 1.0.0