CLAUDE COWORK

Plugin Development Guide

The Complete Reference for Understanding, Extending, and Creating Claude Cowork Plugins

---

VERSION 1.0.0

PLUGIN cowork-plugin-system

RationalEyes.ai

contact@rationaleyes.ai

February 2026

Table of Contents

The Complete Reference for Understanding, Extending, and Creating Claude Cowork Plugins

1. What Is a Plugin?

A plugin is a self-contained package 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

2. Plugin Anatomy and Architecture

Every plugin follows the same directory structure. Some plugins use all of these components; many use only two or three. The only truly required file is `plugin.json` inside the `.claude-plugin/` folder.

```
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: name, description, knowledge
│       ├── references/         # Optional: detailed reference documents
│       ├── scripts/            # Optional: utility scripts (Python, shell)
│       ├── assets/             # Optional: templates, configuration files
│       └── examples/           # Optional: sample data, example configurations
├── agents/                     # Optional: autonomous specialist definitions
│   └── agent-name.md
├── hooks/                      # Optional: event-driven automation
│   └── hooks.json
├── .mcp.json                   # Optional: external service connections
├── CONNECTORS.md               # Optional: tool-category documentation
├── README.md                   # Plugin documentation
└── LICENSE                     # License information
```

Key structural rules:

- The `.claude-plugin/` folder and its `plugin.json` file are always required. This is how the system recognizes a directory as a plugin.
- Component directories (`commands/`, `skills/`, `agents/`) sit at the plugin root, not inside `.claude-plugin/`.
- Only create directories for components the plugin actually uses. An empty `agents/` folder is unnecessary clutter.
- All directory and file names use **kebab-case** (lowercase words separated by hyphens): `campaign-planning`, not `CampaignPlanning` or `campaign_planning`.

Two layout patterns exist across the Cowork plugin library:

PATTERN	EXAMPLE	STRUCTURE
Versioned	`marketing/1.0.0/`	Plugin contents live inside a version subfolder

PATTERN	EXAMPLE	STRUCTURE
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.

---

3. Component Deep Dive

3.1 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.

Minimal example:

```
{
  "name": "marketing",
  "version": "1.0.0",
  "description": "Create content, plan campaigns, and analyze performance across marke
  "author": {
    "name": "Anthropic"
  }
}
```

Field reference:

FIELD	REQUIRED	FORMAT	DESCRIPTION
`name`	Yes	kebab-case string	Unique plugin identifier. Lowercase, hyphens, no spaces or special characters.
`version`	No	Semver (e.g., `1.0.0`)	Version number following MAJOR.MINOR.PATCH convention. Start new plugins at `0.1.0`.
`description`	No	String	Brief explanation of what the plugin does. One to two sentences.
`author`	No	Object with `name`	Who created the plugin.
`homepage`	No	URL string	Link to the plugin's website or documentation.

FIELD	REQUIRED	FORMAT	DESCRIPTION
`repository`	No	URL string	Link to the source code repository.
`license`	No	String	License identifier (e.g., `MIT`, `Apache-2.0`).
`keywords`	No	Array of strings	Tags for discovery (e.g., `["marketing", "campaigns", "content"]`).

Custom component paths (advanced): By default, the system discovers components by looking in the standard directories (`commands/`, `skills/`, `agents/`). You can override or supplement these locations:

```
{
  "name": "my-plugin",
  "commands": "./custom-commands",
  "agents": ["./agents", "./specialized-agents"],
  "hooks": "./config/hooks.json",
  "mcpServers": "./.mcp.json"
}
```

This supplements automatic discovery — it does not replace it. If you specify a custom path, the system checks both the custom path and the default location.

3.2 Commands

Commands are actions that users trigger by typing a slash followed by the command name (e.g., `/campaign-plan`, `/review-contract`, `/forecast`). Each command is a single Markdown file in the `commands/` directory.

Critical concept: Command files are instructions *for Claude*, not documentation for the user. When you write a command file, you are telling Claude what to do when someone invokes that command. Write in the imperative: "Generate a report," "Analyze the data," "Ask the user for their budget."

File location: `commands/command-name.md`

Example command file (`commands/campaign-plan.md`):

```
---
description: Generate a full campaign brief with objectives, channels, and metrics
argument-hint: "<campaign objective or product>"
---

Generate a comprehensive marketing campaign brief.

## Inputs

Gather the following from the user. If not provided, ask before proceeding:

1. **Campaign goal** — the primary objective
2. **Target audience** — who the campaign is aimed at
3. **Timeline** — campaign duration and key dates
4. **Budget range** — approximate budget (optional)

## Output

Present the full campaign brief with these sections:
1. Campaign Overview
2. Target Audience
3. Key Messages
4. Channel Strategy
5. Content Calendar
6. Success Metrics
```

Frontmatter fields (the section between the `---` markers at the top):

FIELD	PURPOSE	EXAMPLE
`description`	Brief text shown when users type `/help`. Keep under 60 characters.	`"Generate a full campaign brief"`
`argument-hint`	Autocomplete hint showing what input the command expects.	`"<campaign objective or product>"`
`allowed-tools`	Restricts which tools Claude can use during this command.	`Read, Grep, Bash(git:*)`
`model`	Overrides the AI model for this command.	`sonnet`, `opus`, or `haiku`

All frontmatter fields are optional. A command file with no frontmatter at all is valid.

Special syntax available in commands:

SYNTAX	WHAT IT DOES	EXAMPLE
`$ARGUMENTS`	Captures everything the user typed after the command name as a single string.	`/review $ARGUMENTS` captures `"the Q4 report"` from `/review the Q4 report`
`$1`, `$2`, `$3`	Captures individual positional arguments.	`$1` captures `report.md` from `/review report.md`
`@path`	Includes the contents of a file directly into the command context.	`@$1` reads whatever file the user specified
`!`command``	Executes a shell command inline and includes the output.	`!`git diff --name-only`` includes the list of changed files
`${CLAUDE_PLUGIN_ROOT}`	Resolves to the plugin's directory path. Keeps references portable.	`@${CLAUDE_PLUGIN_ROOT}/templates/brief.md`

The `allowed-tools` field in detail:

This field controls what Claude is permitted to do when running the command. It accepts a comma-separated list of tool names with optional patterns:

```
# Allow specific tools
allowed-tools: Read, Write, Edit, Bash(git:*)

# Allow Bash only for specific programs
allowed-tools: Bash(npm:*), Read

# Allow specific MCP server tools
allowed-tools: ["mcp__hubspot__search_contacts"]
```

If you omit `allowed-tools`, the command can use any available tool.

3.3 Skills

Skills are packages of domain knowledge that Claude loads automatically when the conversation touches on relevant topics. Unlike commands (which the user explicitly invokes), skills activate in the background based on what the user is talking about.

Each skill lives in its own subdirectory under `skills/`, and must contain a file named `SKILL.md`.

File location: `skills/skill-name/SKILL.md`

Example skill file (`skills/brand-voice/SKILL.md`):

```
---
name: brand-voice
description: >
  Apply and enforce brand voice, style guide, and messaging pillars
  across content. Use when reviewing content for brand consistency,
  documenting a brand voice, adapting tone for different audiences,
  or checking terminology and style guide compliance.
---

# Brand Voice Skill

Frameworks for documenting, applying, and enforcing brand voice
and style guidelines across marketing content.

## Brand Voice Documentation Framework

A complete brand voice document should cover these areas ...
```

Frontmatter fields:

FIELD	REQUIRED	DESCRIPTION
`name`	Yes	Skill identifier, matching the directory name.
`description`	Yes	Third-person description explaining when this skill should activate. Must include specific trigger phrases.
`version`	No	Semver version number.

The description field is crucial. It determines when Claude loads this skill into its working memory. Write it in the third person and include the exact phrases users would say that should trigger this skill:

```
description: >
  This skill should be used when the user asks to "design an API",
  "create API endpoints", "review API structure", or needs guidance
  on REST API best practices, endpoint naming, or request/response design.
```

Good trigger phrases are specific and varied. Include both formal requests ("conduct a competitive analysis") and casual ones ("what are our competitors doing?").

The skill body (everything after the frontmatter) is the actual knowledge. Write it in imperative form ("Parse the config file," not "You should parse the config file"). This is where you put frameworks, checklists, guidelines, reference tables, and domain expertise.

Progressive disclosure — the three-tier loading system:

Skills use a layered approach to manage how much information Claude holds at any given time:

TIER	WHAT LOADS	WHEN	SIZE GUIDELINE
Metadata	`name` + `description` from frontmatter	Always in context	~100 words
SKILL.md body	The full Markdown body of SKILL.md	When the skill triggers based on the conversation	Under 3,000 words (ideally 1,500-2,000)
References and resources	Files in `references/`, `scripts/`, `assets/`, `examples/`	On demand, when Claude needs specific details	Unlimited

This design is intentional. The metadata is lightweight enough to always be present, so Claude can recognize when a skill is relevant. The body provides working knowledge. And the reference files hold deep detail that only loads when actually needed, keeping Claude's context focused.

Skill directory structure:

```
skill-name/
├── SKILL.md              # Core knowledge (required)
├── references/           # Detailed documents loaded on demand
│   ├── detailed-guide.md
│   └── advanced-topics.md
├── examples/             # Sample configurations, templates
│   └── sample-config.json
├── scripts/              # Utility scripts (Python, shell)
│   └── validate.sh
└── assets/               # Templates, images, resources
    └── report-template.md
```

Not every skill needs all these subdirectories. Simple skills may only have `SKILL.md`. Complex skills (like the bio-research plugin's `scientific-problem-selection` skill, which has nine reference documents) use the full structure.

3.4 Agents

Agents are autonomous specialists that Claude can delegate multi-step tasks to. Think of them as dedicated team members with specific expertise — a code reviewer, a ticket analyzer, a research assistant. They work through tasks methodically and report back.

Agents are less commonly used than commands and skills. Most plugins for business use rely on commands and skills. Agents are best when a task requires sustained, methodical work across multiple steps.

File location: `agents/agent-name.md`

Example agent file (`agents/ticket-analyzer.md`):

```
---
name: ticket-analyzer
description: >
  Use this agent when the user needs to analyze tickets,
  triage incoming issues, or prioritize a backlog.

  <example>
  Context: User is preparing for sprint planning
  user: "Help me triage these new tickets"
  assistant: "I'll use the ticket-analyzer agent to review
  and categorize the tickets."
  <commentary>
  Ticket triage requires systematic analysis across multiple
  dimensions, making the agent appropriate.
  </commentary>
  </example>

model: inherit
color: cyan
tools: ["Read", "Grep"]
---

You are a ticket analysis specialist. Analyze tickets for
priority, effort, and dependencies.

**Analysis Process:**
1. Read all ticket descriptions
2. Categorize each by type
3. Estimate effort based on scope
4. Map dependencies
5. Rank by impact-to-effort ratio
```

Frontmatter fields:

FIELD	REQUIRED	FORMAT	DESCRIPTION
`name`	Yes	3-50 chars, kebab-case	Agent identifier. Lowercase letters, numbers, hyphens only. Must start and end with a letter or number.
`description`	Yes	String with `<example>` blocks	When to use this agent. Must include concrete example scenarios.
`model`	Yes	`inherit`, `sonnet`, `opus`, `haiku`	Which AI model the agent uses. `inherit` uses the current session's model.

FIELD	REQUIRED	FORMAT	DESCRIPTION
`color`	Yes	Color name	Visual identifier. Options: `blue`, `cyan`, `green`, `yellow`, `magenta`, `red`.
`tools`	No	Array of strings	Restricts which tools the agent can use. If omitted, the agent can use all tools.

The `<example>` blocks in the description are important. They show Claude concrete scenarios where this agent should be activated, making delegation decisions more reliable.

Color guidelines (for visual identification in the interface):

COLOR	SUGGESTED USE
Blue, Cyan	Analysis, review, research
Green	Success-oriented tasks, completion
Yellow	Validation, caution, quality checks
Red	Critical operations, security
Magenta	Creative work, generation

3.5 Hooks

Hooks are automatic behaviors that fire when specific events occur — like a standing instruction that applies without anyone having to remember to invoke it. For example, a hook could enforce coding standards every time a file is written, or load project context every time a new session starts.

Hooks are configured in a single JSON file at `hooks/hooks.json`.

Two types of hooks:

TYPE	HOW IT WORKS	BEST FOR
Prompt-based	Sends the event to Claude for evaluation. Claude decides what to do.	Complex logic, nuanced decisions, context-dependent rules
Command-based	Runs a shell script or command. Deterministic, no AI involved.	Simple checks, loading files, running validators

Available events:

EVENT	WHEN IT FIRES	COMMON USES
`PreToolUse`	Before a tool executes	Validate operations, enforce policies
`PostToolUse`	After a tool completes	Log actions, verify results
`Stop`	When Claude finishes a response	Quality checks, format enforcement
`SubagentStop`	When a subagent finishes	Review subagent output
`SessionStart`	When a session begins	Load project context, set up environment
`SessionEnd`	When a session ends	Save state, clean up
`UserPromptSubmit`	When the user sends a message	Input validation, context injection
`PreCompact`	Before context is compacted	Preserve important information
`Notification`	When a notification fires	React to system events

Example hooks.json:

```
{
  "SessionStart": [
    {
      "matcher": "",
      "hooks": [
        {
          "type": "command",
          "command": "cat ${CLAUDE_PLUGIN_ROOT}/context/project-context.md",
          "timeout": 10
        }
      ]
    }
  ],
  "PreToolUse": [
    {
      "matcher": "Write|Edit",
      "hooks": [
        {
          "type": "prompt",
          "prompt": "Check that this file write follows project standards. If it viola
          "timeout": 30
        }
      ]
    }
  ]
}
```

The `matcher` field filters which specific triggers activate the hook. For `PreToolUse` and `PostToolUse`, it matches tool names (e.g., `"Write|Edit"` fires for Write or Edit operations). An empty matcher (`""`) means the hook fires for all events of that type.

Command hook output format:

Command-based hooks communicate their decision by printing JSON to standard output:

```
{
  "decision": "approve",
  "reason": "File write follows naming conventions"
}
```

DECISION	EFFECT
`approve`	Allow the operation to proceed
`block`	Prevent the operation and show the reason

DECISION	EFFECT
`ask_user`	Pause and ask the user for confirmation

3.6 MCP Servers

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

MCP server connections are defined in a JSON file, typically `.mcp.json` at the plugin root.

Three types of MCP servers:

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}"
      }
    }
  }
}
```

FIELD	PURPOSE
`command`	The program to run (e.g., `node`, `python`)
`args`	Arguments to pass to the program
`env`	Environment variables the program needs

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 Claude encounters an SSE server for the first time, it prompts you to authenticate through the service's standard sign-in process. After that, the connection is maintained automatically.

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"
    }
  }
}
```

Real-world example — the marketing plugin's `.mcp.json` connects to nine services:

```
{
  "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" },
    "ahrefs":      { "type": "http", "url": "https://api.ahrefs.com/mcp/mcp" },
    "similarweb":  { "type": "http", "url": "https://mcp.similarweb.com" },
    "klaviyo":     { "type": "http", "url": "https://mcp.klaviyo.com/mcp" }
  }
}
```

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 OAuth sign-in	SSE	Handles authentication automatically
Custom internal tool	stdio	Runs locally, full control
REST API with token auth	HTTP	Simple header-based authentication
Local database or file system tool	stdio	Needs local machine access

Portable paths and environment variables:

Always use `${CLAUDE_PLUGIN_ROOT}` when referencing files within the plugin. This variable resolves to the plugin's actual directory path, so the plugin works regardless of where it is installed.

For credentials, use `${ENV_VAR_NAME}` syntax. This pulls values from the user's environment, keeping secrets out of plugin files:

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

---

4. The Placeholder System and CONNECTORS.md

When a plugin is designed to be shared across different organizations, it faces a challenge: Company A uses Slack while Company B uses Microsoft Teams; Company A uses Jira while Company B uses Asana. The placeholder system solves this by letting plugin authors write tool-agnostic instructions that get customized for each organization.

How it works:

In plugin files (commands, skills, agents), 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. During customization (covered in Section 9), the placeholders are replaced with the specific tools your organization uses.

The CONNECTORS.md file documents all the placeholder categories a plugin uses, what tools are pre-configured, and what alternatives are available. It sits at the plugin root.

Example CONNECTORS.md (from the marketing plugin):

```
# Connectors

## How tool references work

Plugin files use `~~category` as a placeholder for whatever tool the user
connects in that category. For example, `~~marketing automation` might mean
HubSpot, Marketo, or any other marketing platform with an MCP server.

Plugins are tool-agnostic — they describe workflows in terms of categories
rather than specific products.

## Connectors for this plugin

| Category | Placeholder | Included servers | Other options |
├──────────┼─────────────┼──────────────────┼────────────────┤
| Chat | `~~chat` | Slack | Microsoft Teams |
| Design | `~~design` | Canva, Figma | Adobe Creative Cloud |
| Marketing automation | `~~marketing automation` | HubSpot | Marketo, Pardot, Mailchi
| Product analytics | `~~product analytics` | Amplitude | Mixpanel, Google Analytics |
| Knowledge base | `~~knowledge base` | Notion | Confluence, Guru |
| SEO | `~~SEO` | Ahrefs, Similarweb | Semrush, Moz |
| Email marketing | `~~email marketing` | Klaviyo | Mailchimp, Brevo, Customer.io |
```

"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 you are 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.

---

5. Three Approaches to Extending a Plugin

There are three ways to add capabilities to an existing plugin. Each has different trade-offs around effort, maintainability, and risk. 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`.

When to use:

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

Risks:

- 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 new campaign type to the marketing plugin:

Open `skills/campaign-planning/SKILL.md` and add a new section:

```
## Event Marketing Campaigns

When planning an event-based campaign (trade show, conference, webinar series):

### Pre-Event Phase (6-8 weeks out)
- Speaker recruitment and content planning
- Registration page and email nurture sequence
- Social media countdown campaign
- Partner and sponsor coordination

### Event Phase
- Live social coverage and engagement
- On-site lead capture and qualification
- Real-time content creation (quotes, photos, highlights)

### Post-Event Phase (1-2 weeks after)
- Follow-up email sequences by engagement level
- Content repurposing (recordings, blog posts, infographics)
- ROI analysis and attendee feedback
```

The change takes effect immediately. No reinstallation or restart is needed.

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 adapted for pharmaceutical and life sciences tea
}
```

3. Make your modifications to the copy — add skills, modify commands, update MCP servers.
4. Install the forked plugin (place it in your plugins directory or install the `.plugin` file).

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

Risks:

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

Example — creating a pharmaceutical marketing plugin:

```
marketing-pharma/
├── .claude-plugin/
│   └── plugin.json             # name: "marketing-pharma"
├── commands/
│   ├── campaign-plan.md        # Original, unmodified
│   ├── brand-review.md         # Original, unmodified
│   ├── mlr-review.md           # NEW: medical-legal-regulatory review
│   └── congress-plan.md        # NEW: conference marketing planning
├── skills/
│   ├── brand-voice/
│   │   └── SKILL.md            # Modified: added pharma voice rules
│   ├── campaign-planning/
│   │   └── SKILL.md            # Modified: added pharma campaign types
│   ├── regulatory-compliance/  # NEW SKILL
│   │   ├── SKILL.md
│   │   └── references/
│   │       ├── fda-rules.md
│   │       └── ema-requirements.md
│   └── medical-affairs/        # NEW SKILL
│       ├── SKILL.md
│       └── references/
│           └── kol-framework.md
├── .mcp.json                   # Modified: added PubMed, ClinicalTrials.gov
├── CONNECTORS.md               # Modified: added life sciences categories
└── README.md                   # Updated documentation
```

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

Keep the original plugin completely untouched. Instead, 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.

When planning campaigns for HCP audiences, ensure all content
follows fair balance guidelines from the pharma-compliance skill.
```

3. The original marketing plugin continues to work as before. Your project-level skill adds pharmaceutical knowledge on top, and the `CLAUDE.md` rules tell Claude when to combine them.

When to use:

- Adding project-specific knowledge without touching any plugin
- Layering industry or company knowledge on top of a general-purpose plugin
- When you want to receive plugin updates without any merge conflicts
- Quick additions that only apply to a specific project

Risks:

- The routing rules in `CLAUDE.md` may not always trigger perfectly
- No structural link between your additional skills and the plugin — they coexist by convention
- If the plugin changes significantly, your routing rules may need updating

Example — adding pharma compliance to the marketing plugin via CLAUDE.md:

File: `.claude/skills/pharma-compliance/SKILL.md`

```
---
name: pharma-compliance
description: >
  Pharmaceutical marketing compliance rules and regulatory requirements.
  Use when creating content for pharmaceutical products, reviewing
  promotional materials for fair balance, preparing for MLR review,
  or working with HCP-targeted marketing materials.
---

# Pharmaceutical Marketing Compliance

## Fair Balance Requirements

All promotional materials must present a fair balance between
efficacy claims and risk information ...

## MLR Review Checklist

Before any promotional content is finalized:
1. All efficacy claims have cited references
2. Risk information is presented with equal prominence
3. Indication statement matches approved labeling
4. No off-label claims or implications
...
```

File: `CLAUDE.md` **(additions)**

```
## Pharmaceutical Marketing Rules

When any marketing plugin command is used for pharmaceutical products:
- Always apply the pharma-compliance skill
- Ensure all content includes required safety information
- Flag any claims that would need MLR review
- Use INN (generic) names alongside brand names
```

---

6. Use Case 1: Adding Industry-Specific Features and Knowledge

This section walks through a detailed example of adding life sciences / pharmaceutical capabilities to the marketing plugin. The same pattern applies to any industry — financial services, legal, healthcare, manufacturing, education, and so on.

What Industry-Specific Customization Typically Involves

AREA	GENERAL PLUGIN	INDUSTRY-SPECIFIC ADDITION
Terminology	General marketing vocabulary	INN vs. brand names, clinical endpoints (PFS, OS, ORR), safety profiles, REMS
Compliance	Basic brand guidelines	FDA promotional rules, fair balance, MLR review, off-label restrictions
Audiences	Demographics and personas	HCPs by specialty, patients, caregivers, payers, KOLs, MSLs, formulary committees
Campaign types	Product launch, awareness, lead gen	Disease awareness (branded/unbranded), congress marketing, peer-to-peer, patient support programs
Channels	Social, email, web, paid	Medical journals, conference booths, MSL field visits, medical portals (Medscape, DocCheck)
KPIs	Leads, conversions, engagement	Share of voice, prescriber adoption, formulary wins, congress engagement scores

Approach A Example: Direct Modification

Add pharmaceutical audience categories directly to `skills/campaign-planning/SKILL.md`:

```
## Life Sciences Audience Categories

### Healthcare Professionals (HCPs)
- **Physicians** — segment by specialty (oncology, cardiology, neurology, etc.),
  prescribing volume, and influence tier
- **Pharmacists** — hospital vs. retail, formulary committee involvement
- **Nurses and PAs** — often key influencers in treatment decisions

### Patients and Caregivers
- **Diagnosed patients** — segment by disease stage, treatment history,
  and information-seeking behavior
- **Caregivers** — primary decision influencers for pediatric and elderly patients

### Key Opinion Leaders (KOLs)
- **Tier 1 (National)** — published researchers, guideline authors,
  conference speakers
- **Tier 2 (Regional)** — department heads, local conference speakers
- **Tier 3 (Community)** — High-prescribing community physicians

### Market Access
- **Payers** — insurance medical directors, pharmacy benefit managers
- **Formulary committees** — hospital and health system decision-makers
- **HEOR specialists** — health economics and outcomes research contacts
```

Approach B Example: Fork with New Skills

Create a new skill in the forked plugin at `skills/regulatory-compliance/SKILL.md`:

```
---
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,
  classifying promotional materials, or ensuring FDA/EMA compliance
  in marketing communications.
---

# Regulatory Compliance for Life Sciences Marketing

## Material Classification

Classify every piece of marketing content before production begins:

| Type | Definition | Review Requirements |
|---|---|---|
| Branded promotional | Names the product, makes claims | Full MLR review required |
```

```
| Unbranded disease awareness | Discusses the condition, no product mention | Medical/
| Reminder advertising | Product name only, no claims | Abbreviated review |
| Scientific exchange | Peer-to-peer, balanced data presentation | Medical review |

## Fair Balance Checklist

For every branded promotional piece:
- [ ] Efficacy claims are supported by cited clinical data
- [ ] Risk information (side effects, contraindications, warnings)
      is presented with equal prominence to efficacy claims
- [ ] Indication statement matches the approved prescribing information
- [ ] No off-label claims, whether explicit or implied
- [ ] Black box warnings (if applicable) are prominently displayed
- [ ] Reference list is complete and uses approved citations only

## MLR 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 before production
```

Create a corresponding reference file at `skills/regulatory-compliance/references/fda-promotional-rules.md` with detailed FDA guidance, and another at `references/ema-requirements.md` for European regulations.

Approach C Example: CLAUDE.md with Complementary Skill

Create `.claude/skills/pharma-kpis/SKILL.md` in your project:

```
---
name: pharma-kpis
description: >
  Pharmaceutical and life sciences marketing KPIs and performance
  measurement. Use when defining success metrics for pharma campaigns,
  measuring HCP engagement, tracking prescriber adoption, analyzing
  share of voice, or reporting on medical affairs activities.
---

# Pharmaceutical Marketing KPIs

## Commercial KPIs

| KPI | What It Measures | Typical Targets |
|-----|-----------------|-----------------|
| New prescriptions (NRx) | First-time prescriptions written | Varies by launch phase
| Total prescriptions (TRx) | All prescriptions including refills | Growth rate vs. ma
| Market share | Your product vs. competitors in the category | Depends on market posi
| Share of voice (SOV) | Your brand mentions vs. competitors | SOV > market share = gr
| Prescriber adoption curve | % of target physicians who have prescribed | 30-50% in f
| Formulary wins | Health plans that added your product | Track by lives covered |

## HCP Engagement KPIs

| KPI | What It Measures | Typical Targets |
|-----|-----------------|-----------------|
| HCP reach | Unique physicians engaged across all channels | 70-80% of target list |
| Frequency | Average number of touchpoints per HCP | 6-12 per year across channels |
| Rep-triggered email open rate | Email engagement from field force | 25-40% |
| Congress booth visits | Traffic at conference exhibits | Depends on event size |
| KOL engagement score | Depth of relationship with key opinion leaders | Composite: p
```

Add to your project's `CLAUDE.md`:

```
## Pharma Marketing Measurement

When the marketing plugin generates performance reports or success
metrics for pharmaceutical products, always incorporate pharma-specific
KPIs from the pharma-kpis skill. Replace generic marketing KPIs
(leads, MQLs, website conversions) with pharmaceutical equivalents
(NRx, TRx, market share, prescriber adoption) where appropriate.
```

7. Use Case 2: Adding Company-Specific Features

This section walks through adding "Acme Corp" specifics to a plugin. Every company has its own brand voice, approval processes, team structure, terminology, and tools. Here is how to encode those into the plugin system.

What Company-Specific Customization Typically Involves

AREA	WHAT TO CAPTURE	WHERE IT GOES
Brand voice	Tone, vocabulary, style rules, do/don't examples	Modify `brand-voice` skill or create complementary skill
Internal processes	Approval workflows, review cycles, sign-off chains	New skill or additions to existing command instructions
Team structure	Who owns what, routing rules, escalation paths	CLAUDE.md rules or skill reference files
Proprietary methods	Frameworks, scoring models, naming conventions	New skill with reference documents
Tool configuration	Specific workspace IDs, channel names, project names	CONNECTORS.md replacements and `.mcp.json`

Approach A Example: Direct Modification

Edit `skills/brand-voice/SKILL.md` to add Acme Corp's voice rules:

```
## Acme Corp 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're," "you'll").
  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 |
| team members | employees, workers, staff | Company culture preference |
| clients | customers, users | Reflects relationship, not transaction |
| challenges | problems, issues | Positive framing |

### Approval Workflow
All external content must follow this review chain:
1. Content creator drafts
2. Marketing manager reviews for brand alignment
3. Legal reviews for compliance (for claims, comparisons, testimonials only)
4. VP Marketing final approval (for campaigns over $10K budget)
```

Approach B Example: Fork with Company Processes

In a forked plugin, create `skills/acme-processes/SKILL.md`:

```
---
name: acme-processes
description: >
  Acme Corp internal marketing processes, approval workflows, and
  team structures. Use when planning campaigns at Acme, routing
  content for review, identifying stakeholders, understanding
  budget approval chains, or following Acme's go-to-market playbook.
---

# Acme Corp Marketing Processes

## Campaign Approval Chain

| Budget Tier | Approver | Turnaround |
|---|---|---|
| Under $5K | Marketing Manager | 2 business days |
| $5K-$25K | Director of Marketing | 3 business days |
| $25K-$100K | VP Marketing | 5 business days |
```

```
| Over $100K | CMO + CFO | 10 business days |

## Content Review Matrix

| Content Type | Brand Review | Legal Review | Exec Review |
|---|---|---|---|
| Blog post | Required | Not needed | Not needed |
| Social post | Required | Not needed | Not needed |
| Email campaign | Required | If claims made | Over 10K recipients |
| Press release | Required | Required | Required |
| Case study | Required | Required | Required |
| Paid ad creative | Required | If claims made | Over $10K spend |

## Team Structure

- **Content Team** (reports to Director of Content)
  - Blog, SEO, thought leadership, email newsletters
  - Slack channel: #content-team
- **Demand Gen** (reports to Director of Demand Gen)
  - Paid campaigns, webinars, lead nurture
  - Slack channel: #demand-gen
- **Brand and Creative** (reports to Director of Brand)
  - Design, brand guidelines, event materials
  - Slack channel: #brand-creative
- **Product Marketing** (reports to VP Marketing)
  - Launches, positioning, competitive intelligence
  - Slack channel: #product-marketing
```

Approach C Example: CLAUDE.md Rules

Add to your project's `CLAUDE.md`:

```
## Acme Corp Marketing Rules

### Brand Voice
- Always use "Acme Platform" (capitalized) when referring to our product
- Use "clients" not "customers" or "users"
- Use "team members" not "employees"
- Tone: confident, clear, warm. No jargon. No ALL CAPS.
- Always use Oxford comma

### Routing
- Content over $10K budget: tag VP Marketing for approval
- Press releases and case studies: always require legal review
- Product claims in any content: require legal review

### Channels and Tools
- Project management: Linear (workspace: acme-marketing)
- Design requests: Figma (project: Marketing Assets 2025)
- Analytics: Amplitude (project: Acme Production)
- Email: Klaviyo (list: "Acme Newsletter Subscribers")
```

---

8. Creating a New Plugin from Scratch

Building a new plugin follows a five-phase workflow. The `cowork-plugin-management` plugin provides a guided, conversational experience for this process. Below is a summary of each phase so you understand what is involved.

Phase 1: Discovery

Define what the plugin should do and why. Answer these questions:

- What problem does this plugin solve?
- Who will use it and in what context?
- Does it need to connect to external tools or services?
- Is there a similar plugin or workflow to use as a starting point?

The output of this phase is a clear statement of purpose and scope.

Phase 2: Component Planning

Based on the discovery, determine which component types you need:

COMPONENT	WHEN YOU NEED IT
Skills	You have domain knowledge Claude should draw on automatically
Commands	Users need specific actions they invoke by name
MCP Servers	The plugin needs to read from or write to external services
Agents	Complex multi-step tasks benefit from a dedicated specialist
Hooks	Something should happen automatically on certain events

Most plugins start with just commands and skills. Add other components as needs become clear.

Phase 3: Design and Clarifying Questions

Specify each component in detail before creating any files:

- **Skills**: What topics should trigger each skill? What knowledge domains does it cover?
- **Commands**: What inputs does each command accept? What output should it produce?
- **MCP Servers**: What services need to be connected? What authentication method?
- **Agents**: What is the agent's process? What tools does it need?
- **Hooks**: What events should trigger behavior? Should the logic be AI-driven or deterministic?

Phase 4: Implementation

Create all the files following the directory structure and component schemas described in Sections 2 and 3. Order of operations:

1. Create the directory structure
2. Create `plugin.json`
3. Create each component (skills, commands, agents, hooks, MCP config)
4. Create `README.md`
5. Create `CONNECTORS.md` if using the placeholder system

Phase 5: Review and Package

1. Review every component — does each skill have clear trigger phrases? Are commands written as instructions for Claude? Are MCP servers correctly configured?
2. Test by installing the plugin and exercising each command and skill.
3. Package as a `.plugin` file (see Section 10).

Start small. A plugin with one well-crafted skill and two commands is far more useful than one with five incomplete skills and ten poorly written commands. You can always add more components in later versions.

---

9. Customizing an Existing Plugin

The `cowork-plugin-management` plugin includes a dedicated customizer skill that walks you through adapting a generic plugin for your specific organization. Here is the workflow it follows.

The Customizer Workflow

Step 1 — Gather Context

The customizer searches your connected knowledge sources (Slack messages, internal documents, email) to learn what tools and processes your organization uses. It looks for:

- Tool names for each `~~` placeholder category
- Organizational processes and workflows
- Team naming conventions
- Configuration values (workspace IDs, project names)

Step 2 — Identify Customization Points

The customizer scans the plugin for all `~~` placeholders and creates a task list. Each task is presented in plain language:

- "Learn how standup prep works at your company"
- "Identify which design tool your team uses"
- "Determine your project management workflow"

Step 3 — Apply Changes

For each customization point:

- If the answer was found in your knowledge sources, it applies the change automatically
- If not, it asks you directly

Changes include:

- Replacing `~~project tracker` with "Linear" throughout all plugin files
- Updating URL patterns to match your specific tools
- Adding organization-specific values (workspace IDs, channel names)

Step 4 — Configure MCP Servers

After resolving all placeholders, the customizer searches for MCP servers that match your identified tools and helps you connect them. For example, if you said you use Asana for project management, it finds the Asana MCP server and prompts you to authenticate.

Step 5 — Package and Deliver

The customized plugin is packaged as a `.plugin` file for installation.

---

10. Plugin Packaging and Distribution

The .plugin File Format

A `.plugin` file is a ZIP archive containing the entire plugin directory structure. It is the standard distribution format for sharing plugins.

Creating a .plugin file:

```
cd /path/to/your-plugin-directory
zip -r /tmp/my-plugin.plugin . -x "*.DS_Store"
```

Naming convention: Use the plugin name from `plugin.json`. If the name is `marketing-pharma`, the file should be `marketing-pharma.plugin`.

What to include: Everything in the plugin directory — `plugin.json`, commands, skills, agents, hooks, MCP configuration, README, LICENSE, and all reference files.

What to exclude: Operating system files (`.DS_Store`), temporary files, and any files containing credentials or secrets.

Installing a Plugin

There are two installation paths depending on the plugin source:

Standard plugins (from the Cowork library): All standard Cowork plugins (Marketing, Data, Finance, Legal, Sales, etc.) are available directly from **Plugin Settings** in the Cowork desktop app. Open Plugin Settings, find the plugin you want, and click **Install**. One click, no CLI required.

Custom or modified plugins (`.plugin` files): When you open a `.plugin` file in Cowork, it presents a rich preview showing the plugin's contents — commands, skills, references, and MCP configurations. You can browse the files and accept the plugin by pressing the install button. The plugin activates immediately after installation. This is the path for plugins you build yourself, receive from your IT team, or create by customizing a standard plugin.

Claude Code CLI (alternative): For users running Claude Code in the terminal rather than the Cowork desktop app, install plugins via:

```
claude plugins add knowledge-work-plugins/<plugin-name>   # standard plugins
claude plugins add /path/to/custom-plugin                  # custom plugins
```

After installation (via any method), the plugin's commands appear in `/help`, skills activate automatically based on conversational context, and MCP servers defined in the plugin's `.mcp.json` become available. No restart is required.

Version Management

Use semantic versioning (MAJOR.MINOR.PATCH) in `plugin.json`:

VERSION COMPONENT	WHEN TO INCREMENT	EXAMPLE
MAJOR	Breaking changes that require users to update their workflows	`1.0.0` to `2.0.0`
MINOR	New commands, skills, or features that are backward-compatible	`1.0.0` to `1.1.0`
PATCH	Bug fixes, typo corrections, small improvements	`1.0.0` to `1.0.1`

Start new plugins at `0.1.0`. Increment to `1.0.0` when the plugin is stable and ready for broad use.

---

11. Best Practices and Common Mistakes

Do This

PRACTICE	WHY IT MATTERS
Start small — one well-crafted skill is better than five half-baked ones	Quality over quantity. A focused plugin that does one thing well gets used. A sprawling plugin with gaps gets abandoned.
Use progressive disclosure — lean SKILL.md, detailed references	Keeps Claude's context focused. Loads detail only when needed, not all at once.
Write clear trigger phrases in skill descriptions	If Claude cannot tell when a skill is relevant, it will never load it. Be specific: "Use when the user asks to plan a campaign" beats "Marketing assistance."
Write commands as instructions for Claude	Commands tell Claude what to do, not the user what to expect. "Generate a report" not "This command generates a report."
Use imperative writing style in skills	"Parse the data. Identify patterns. Present findings." Not "You should parse the data."
Use `${CLAUDE_PLUGIN_ROOT}` for all intra-plugin paths	Makes the plugin work regardless of where it is installed. Never hardcode `/Users/someone/plugins/my-plugin/`.
Use environment variables for credentials	Never put API keys, tokens, or passwords in plugin files. Use `${MY_API_KEY}` in `.mcp.json`.
Keep SKILL.md under 3,000 words	Move detailed content to `references/`. Claude's context is valuable — do not waste it on material that is only occasionally relevant.
Test each command and skill after creation	Invoke every command, check that every skill triggers on the right topics, verify every MCP connection.
Document your changes when modifying a plugin	Add comments or update README to track what you changed and why. Future you will thank present you.

Avoid This

MISTAKE	WHY IT IS A PROBLEM
Creating empty component directories	Clutter. Only create `agents/`, `hooks/`, etc. if you actually have components of that type.

MISTAKE	WHY IT IS A PROBLEM
Writing skill descriptions without trigger phrases	Claude will not know when to load the skill. "Provides marketing knowledge" tells Claude nothing useful.
Putting user documentation in command files	Commands are Claude's instructions, not user manuals. User docs belong in README.md.
Hardcoding file paths	Breaks the plugin when installed in a different location. Always use `${CLAUDE_PLUGIN_ROOT}`.
Storing credentials in plugin files	Security risk. Anyone with the plugin file sees your secrets. Use environment variables.
Making skills too long	A 10,000-word SKILL.md wastes Claude's context window. Split into a concise SKILL.md and `references/` files.
Skipping the description frontmatter on skills	Without a description, the skill metadata is useless and triggering becomes unreliable.
Forgetting CONNECTORS.md when using placeholders	Users will not know what `~project tracker` means or how to configure it without this documentation.
Modifying original plugins without a backup	Plugin updates will overwrite your changes. Fork the plugin (Approach B) or use complementary skills (Approach C) instead.

---

12. Troubleshooting Guide

Plugin Not Loading

Symptoms: Plugin commands do not appear in `/help`. Skills do not trigger. The plugin seems invisible.

Checklist:

1. **Verify plugin.json exists and is valid JSON.**
   - Must be at `.claude-plugin/plugin.json` (note the dot prefix on the directory).
   - Must contain at least a `name` field.
   - Common error: trailing commas in JSON (not allowed).
2. **Check the directory structure.**
   - `.claude-plugin/` must be at the plugin root, not nested inside another folder.

- If using the versioned pattern (`plugin-name/1.0.0/`), make sure `plugin.json` is inside the version folder: `plugin-name/1.0.0/.claude-plugin/plugin.json` .

3. **Verify the plugin is installed.**
   - Confirm the plugin appears in your installed plugins list.
   - If you moved or renamed the directory, reinstall the `.plugin` file.
4. **Check the plugin name.**
   - Must be kebab-case (lowercase, hyphens only).
   - No spaces, underscores, or special characters.

Skills Not Triggering

Symptoms: You discuss a topic the skill should handle, but Claude does not seem to have that knowledge.

Checklist:

1. **Check the skill description for trigger phrases.**
   - The `description` field in SKILL.md frontmatter must include specific phrases that match how users actually talk about the topic.
   - Bad: `"Provides marketing assistance"` (too vague)
   - Good: `"Use when planning a campaign, building a content calendar, defining campaign KPIs, or allocating budget across channels"` (specific trigger phrases)
2. **Verify SKILL.md is named correctly.**
   - Must be `SKILL.md` (all caps), not `skill.md` or `Skill.md` .
   - Must be directly inside the skill's directory: `skills/my-skill/SKILL.md` .
3. **Check YAML frontmatter format.**
   - Must start and end with `---` on their own lines.
   - `name` and `description` fields are required.
   - Indentation must be consistent (use spaces, not tabs).
4. **Check skill body length.**
   - If SKILL.md exceeds 3,000 words, consider moving content to `references/` files. An excessively long skill may cause context-management issues.

MCP Servers Not Connecting

Symptoms: Claude cannot access external tools. Commands that rely on external data fail.

Checklist:

1. **Verify `.mcp.json` is valid JSON.**
   - Use a JSON validator to check for syntax errors.
   - Common errors: missing commas between entries, trailing commas, unescaped characters.
2. **Check the server URL.**
   - Make sure the URL is correct and the service is accessible.
   - SSE servers typically have `/sse` in the path.
   - HTTP servers typically use `/mcp` in the path.
3. **Verify authentication.**
   - For SSE servers: have you completed the OAuth sign-in flow?
   - For HTTP servers with tokens: is the environment variable set?
   - For stdio servers: is the required program (node, python) installed?
4. **Check environment variables.**
   - If the config uses `${MY_API_KEY}`, confirm that environment variable is set in your session.
   - Environment variables are case-sensitive.
5. **Test the connection independently.**
   - Try accessing the MCP server URL in a browser (for HTTP/SSE types) to confirm it is reachable.

Commands Not Appearing

Symptoms: Typing `/command-name` does not work or does not autocomplete.

Checklist:

1. **Verify the file is in `commands/`.**
   - Command files must be directly in the `commands/` directory, not in subdirectories.
   - File extension must be `.md`.
2. **Check frontmatter syntax.**

- If using frontmatter, ensure the `---` delimiters are correct.
- The `description` field (if present) must be a valid string.

3. **Verify file naming.**
   - Use kebab-case: `campaign-plan.md`, not `campaignPlan.md`.
   - The command name comes from the filename (without `.md`): `campaign-plan.md` becomes `/campaign-plan`.
4. **Check for duplicate names.**
   - If two plugins define a command with the same name, only one will work.

Hooks Not Firing

Symptoms: Automated behaviors you configured do not activate.

Checklist:

1. **Verify `hooks/hooks.json` is valid JSON.**
2. **Check the event name** — must match exactly (e.g., `PreToolUse`, not `pre-tool-use`).
3. **Check the matcher** — for tool-based events, the matcher must match the tool name pattern.
4. **Check timeout values** — if the hook's script or prompt takes longer than the timeout, it silently fails.
5. **For command hooks** — verify the script exists at the specified path and is executable.
6. **For prompt hooks** — keep the prompt concise and specific. Vague prompts may produce unpredictable results.

General Debugging Steps

1. **Check file permissions.** Plugin files should be readable. If you copied files from another system, permissions may have changed.
2. **Look for invisible characters.** Files edited in certain word processors may contain invisible characters that break YAML or JSON parsing. Use a plain-text editor.
3. **Validate JSON files.** Both `plugin.json` and `.mcp.json` must be valid JSON. Even a single misplaced comma breaks parsing.
4. **Validate YAML frontmatter.** Ensure the `---` delimiters are present and the indentation is consistent.
5. **Restart the session.** Some changes (particularly to hooks and MCP servers) may require starting a new session to take effect.

Quick Reference: File Formats at a Glance

FILE	LOCATION	FORMAT	REQUIRED FIELDS
`plugin.json`	`.claude-plugin/plugin.json`	JSON	`name`
Command	`commands/*.md`	Markdown + optional YAML frontmatter	None (frontmatter is optional)
SKILL.md	`skills/*/SKILL.md`	Markdown + YAML frontmatter	`name`, `description`
Agent	`agents/*.md`	Markdown + YAML frontmatter	`name`, `description`, `model`, `color`
hooks.json	`hooks/hooks.json`	JSON	Event keys with hook arrays
.mcp.json	`.mcp.json` (plugin root)	JSON	Server entries with type and connection info
CONNECTORS.md	Plugin root	Markdown	Category table

Quick Reference: Extension Approach Decision Guide

SITUATION	RECOMMENDED APPROACH	REASON
Quick personal experiment	A (Direct Modification)	Fastest, lowest effort
Small addition for one project	C (Complementary via CLAUDE.md)	No risk to the plugin, project-scoped
Team-wide customization	B (Fork the Plugin)	Everyone gets the same changes, clean separation
Significant behavior changes	B (Fork the Plugin)	Full control, auditable
Adding company knowledge	C (Complementary via CLAUDE.md)	Lightest touch, survives plugin updates
Adding industry knowledge	B (Fork) or C (CLAUDE.md)	Fork for deep changes, CLAUDE.md for overlays

SITUATION	RECOMMENDED APPROACH	REASON
Distributing to other organizations	B (Fork) with new name	Clean separation, independent versioning
Testing before committing	A (Direct Modification)	Easy to revert by reinstalling the original

---

This guide documents the Claude Cowork plugin system as of version 0.2.1 of the cowork-plugin-management plugin. For the latest schemas and specifications, consult the `create-cowork-plugin` *skill and its reference documents within that plugin.*

RationalEyes.ai

contact@rationaleyes.ai

Intelligent Automation for Knowledge Work

Plugin Development Guide — Version 1.0.0