The Complete Guide to Building Skills for Claude

Claude

Contents

Introduction

A **skill** is a set of instructions - packaged as a simple folder - that teaches Claude how to handle specific tasks or workflows. Skills are one of the most powerful ways to customize Claude for your specific needs. Instead of re-explaining your preferences, processes, and domain expertise in every conversation, skills let you teach Claude once and benefit every time.

Skills are powerful when you have repeatable workflows: generating frontend designs from specs, conducting research with consistent methodology, creating documents that follow your team's style guide, or orchestrating multi-step processes. They work well with Claude's built-in capabilities like code execution and document creation. For those building MCP integrations, skills add another powerful layer helping turn raw tool access into reliable, optimized workflows.

This guide covers everything you need to know to build effective skills - from planning and structure to testing and distribution. Whether you're building a skill for yourself, your team, or for the community, you'll find practical patterns and real-world examples throughout.

What you'll learn:

- Technical requirements and best practices for skill structure
- Patterns for standalone skills and MCP-enhanced workflows
- Patterns we've seen work well across different use cases
- How to test, iterate, and distribute your skills

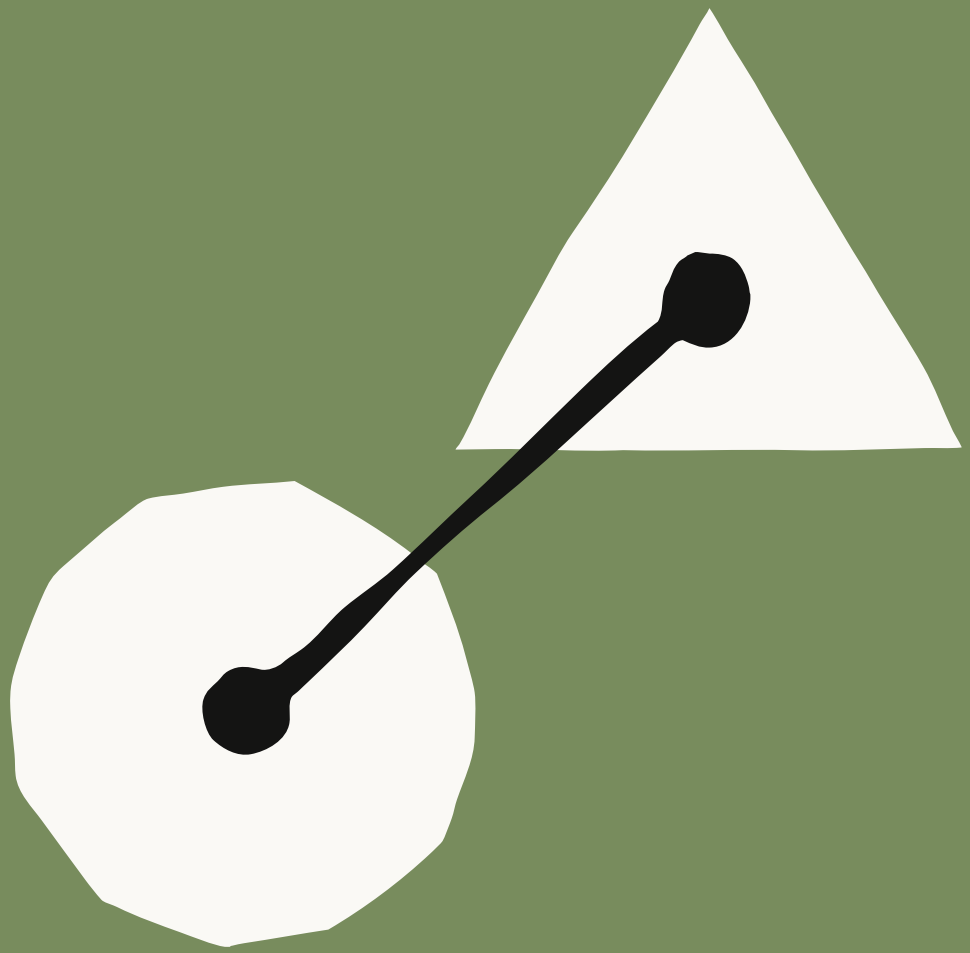
Who this is for:

- Developers who want Claude to follow specific workflows consistently
- Power users who want Claude to follow specific workflows
- Teams looking to standardize how Claude works across their organization

Two Paths Through This Guide
Building standalone skills? Focus on Fundamentals, Planning and Design, and category 1-2. Enhancing an MCP integration? The "Skills + MCP" section and category 3 are for you. Both paths share the same technical requirements, but you choose what's relevant to your use case.

What you'll get out of this guide: By the end, you'll be able to build a functional skill in a single sitting. Expect about 15-30 minutes to build and test your first working skill using the skill-creator.

Let's get started.



Chapter 1

Fundamentals

Chapter 1

Fundamentals

What is a skill?

A skill is a folder containing:

- **SKILL.md** (required): Instructions in Markdown with YAML frontmatter
- **scripts/** (optional): Executable code (Python, Bash, etc.)
- **references/** (optional): Documentation loaded as needed
- **assets/** (optional): Templates, fonts, icons used in output

Core design principles

Progressive Disclosure

Skills use a three-level system:

- **First level (YAML frontmatter):** Always loaded in Claude's system prompt. Provides just enough information for Claude to know when each skill should be used without loading all of it into context.
- **Second level (SKILL.md body):** Loaded when Claude thinks the skill is relevant to the current task. Contains the full instructions and guidance.
- **Third level (Linked files):** Additional files bundled within the skill directory that Claude can choose to navigate and discover only as needed.

This progressive disclosure minimizes token usage while maintaining specialized expertise.

Composability

Claude can load multiple skills simultaneously. Your skill should work well alongside others, not assume it's the only capability available.

Portability

Skills work identically across Claude.ai, Claude Code, and API. Create a skill once and it works across all surfaces without modification, provided the environment supports any dependencies the skill requires.

For MCP Builders: Skills + Connectors

💡 *Building standalone skills without MCP? Skip to Planning and Design - you can always return here later.*

If you already have a **working MCP server**, you've done the hard part. Skills are the knowledge layer on top - capturing the workflows and best practices you already know, so Claude can apply them consistently.

The kitchen analogy

MCP provides the professional kitchen: access to tools, ingredients, and equipment.

Skills provide the recipes: step-by-step instructions on how to create something valuable.

Together, they enable users to accomplish complex tasks without needing to figure out every step themselves.

How they work together:

MCP (Connectivity)	Skills (Knowledge)
Connects Claude to your service (Notion, Asana, Linear, etc.)	Teaches Claude how to use your service effectively
Provides real-time data access and tool invocation	Captures workflows and best practices
What Claude can do	How Claude should do it

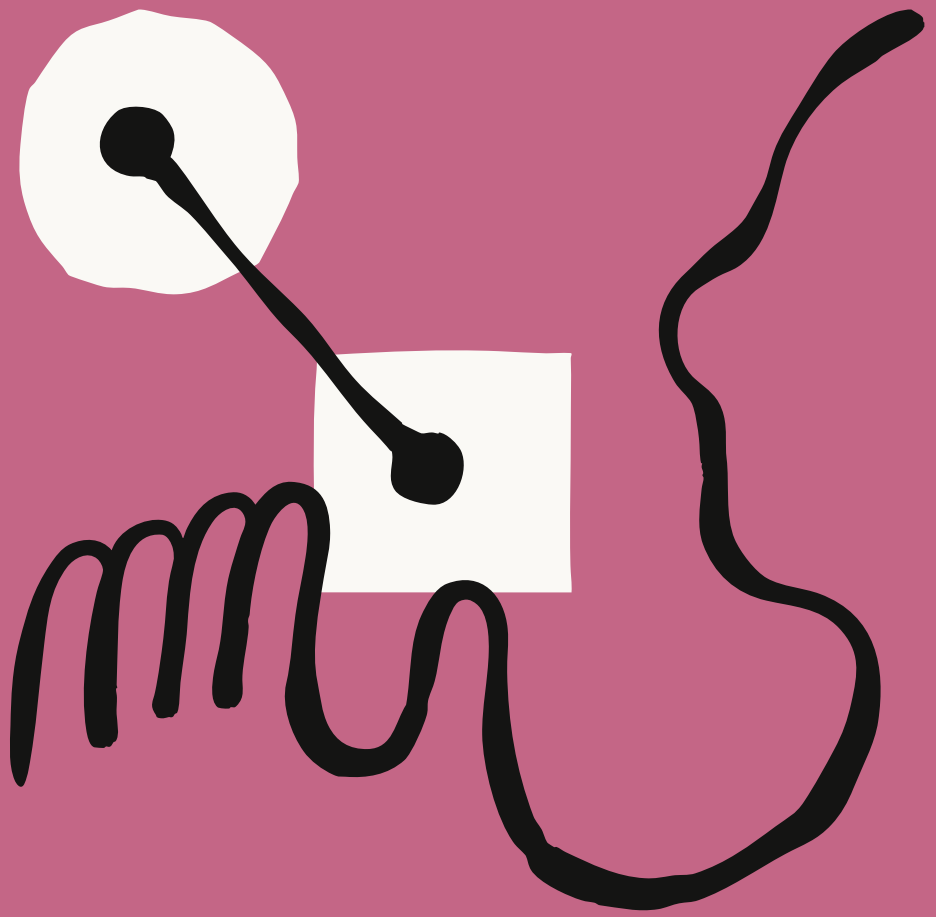
Why this matters for your MCP users

Without skills:

- Users connect your MCP but don't know what to do next
- Support tickets asking "how do I do X with your integration"
- Each conversation starts from scratch
- Inconsistent results because users prompt differently each time
- Users blame your connector when the real issue is workflow guidance

With skills:

- Pre-built workflows activate automatically when needed
- Consistent, reliable tool usage
- Best practices embedded in every interaction
- Lower learning curve for your integration



Chapter 2

Planning and design

Chapter 2

Planning and design

Start with use cases

Before writing any code, identify 2-3 concrete use cases your skill should enable.

Good use case definition:

```
Use Case: Project Sprint Planning
Trigger: User says "help me plan this sprint" or "create
sprint tasks"
Steps:
1. Fetch current project status from Linear (via MCP)
2. Analyze team velocity and capacity
3. Suggest task prioritization
4. Create tasks in Linear with proper labels and estimates
Result: Fully planned sprint with tasks created
```

Ask yourself:

- What does a user want to accomplish?
- What multi-step workflows does this require?
- Which tools are needed (built-in or MCP?)
- What domain knowledge or best practices should be embedded?

Common skill use case categories

At Anthropic, we've observed three common use cases:

Category 1: Document & Asset Creation

Used for: Creating consistent, high-quality output including documents, presentations, apps, designs, code, etc.

Real example: **frontend-design skill** (also see **skills for docx, pptx, xlsx, and ppt**)

"Create distinctive, production-grade frontend interfaces with high design quality. Use when building web components, pages, artifacts, posters, or applications."

Key techniques:

- Embedded style guides and brand standards
- Template structures for consistent output
- Quality checklists before finalizing
- No external tools required - uses Claude's built-in capabilities

Category 2: Workflow Automation

Used for: Multi-step processes that benefit from consistent methodology, including coordination across multiple MCP servers.

Real example: **skill-creator skill**

"Interactive guide for creating new skills. Walks the user through use case definition, frontmatter generation, instruction writing, and validation."

Key techniques:

- Step-by-step workflow with validation gates
- Templates for common structures
- Built-in review and improvement suggestions
- Iterative refinement loops

Category 3: MCP Enhancement

Used for: Workflow guidance to enhance the tool access an MCP server provides.

Real example: **sentry-code-review skill (from Sentry)**

"Automatically analyzes and fixes detected bugs in GitHub Pull Requests using Sentry's error monitoring data via their MCP server."

Key techniques:

- Coordinates multiple MCP calls in sequence
- Embeds domain expertise
- Provides context users would otherwise need to specify
- Error handling for common MCP issues

Define success criteria

How will you know your skill is working?

These are aspirational targets - rough benchmarks rather than precise thresholds. Aim for rigor but accept that there will be an element of vibes-based assessment. We are actively developing more robust measurement guidance and tooling.

Quantitative metrics:

- Skill triggers on 90% of relevant queries
  - *How to measure:* Run 10-20 test queries that should trigger your skill. Track how many times it loads automatically vs. requires explicit invocation.
- Completes workflow in X tool calls
  - *How to measure:* Compare the same task with and without the skill enabled. Count tool calls and total tokens consumed.
- 0 failed API calls per workflow
  - *How to measure:* Monitor MCP server logs during test runs. Track retry rates and error codes.

Qualitative metrics:

- Users don't need to prompt Claude about next steps
  - *How to assess:* During testing, note how often you need to redirect or clarify. Ask beta users for feedback.
- Workflows complete without user correction
  - *How to assess:* Run the same request 3-5 times. Compare outputs for structural consistency and quality.
- Consistent results across sessions
  - *How to assess:* Can a new user accomplish the task on first try with minimal guidance?

Technical requirements

File structure

```
your-skill-name/
├── SKILL.md              # Required - main skill file
├── scripts/              # Optional - executable code
│   ├── process_data.py # Example
│   └── validate.sh # Example
├── references/           # Optional - documentation
│   ├── api-guide.md # Example
│   └── examples/ # Example
└── assets/              # Optional - templates, etc.
    └── report-template.md # Example
```

Critical rules

SKILL.md naming:

- Must be exactly `SKILL.md` (case-sensitive)
- No variations accepted (SKILL.MD, skill.md, etc.)

Skill folder naming:

- Use kebab-case: `notion-project-setup` ✅
- No spaces: `Notion Project Setup` ❌
- No underscores: `notion_project_setup` ❌
- No capitals: `NotionProjectSetup` ❌

No README.md:

- Don't include README.md inside your skill folder
- All documentation goes in SKILL.md or references/
- Note: when distributing via GitHub, you'll still want a repo-level README for human users — see Distribution and Sharing.

YAML frontmatter: The most important part

The YAML frontmatter is how Claude decides whether to load your skill. Get this right.

Minimal required format

```
---
name: your-skill-name
description: What it does. Use when user asks to [specific
phrases].
---
```

That's all you need to start.

Field requirements

name (required):

- kebab-case only
- No spaces or capitals
- Should match folder name

description (required):

- **MUST include BOTH:**
  - What the skill does
  - When to use it (trigger conditions)
- Under 1024 characters
- No XML tags (< or >)
- Include specific tasks users might say
- Mention file types if relevant

license (optional):

- Use if making skill open source
- Common: MIT, Apache-2.0

compatibility (optional)

- 1-500 characters
- Indicates environment requirements: e.g. intended product, required system packages, network access needs, etc.

metadata (optional):

- Any custom key-value pairs
- Suggested: author, version, mcp-server
- *Example:*

```yaml
metadata:
     author: ProjectHub
     version: 1.0.0 mcp-server: projecthub
```

Security restrictions

Forbidden in frontmatter:

- XML angle brackets (`< >`)
- Skills with "claude" or "anthropic" in name (reserved)

Why: Frontmatter appears in Claude's system prompt. Malicious content could inject instructions.

Writing effective skills

The description field

According to Anthropic's **engineering blog**: "This metadata...provides just enough information for Claude to know when each skill should be used without loading all of it into context." This is the first level of progressive disclosure.

Structure:

```
[What it does] + [When to use it] + [Key capabilities]
```

Examples of good descriptions:

```
# Good - specific and actionable
description: Analyzes Figma design files and generates
developer handoff documentation. Use when user uploads .fig
files, asks for "design specs", "component documentation", or
"design-to-code handoff".

# Good - includes trigger phrases
description: Manages Linear project workflows including sprint
planning, task creation, and status tracking. Use when user
mentions "sprint", "Linear tasks", "project planning", or asks
to "create tickets".

# Good - clear value proposition
description: End-to-end customer onboarding workflow for
PayFlow. Handles account creation, payment setup, and
subscription management. Use when user says "onboard new
customer", "set up subscription", or "create PayFlow account".
```

Examples of bad descriptions:

```
# Too vague
description: Helps with projects.

# Missing triggers
description: Creates sophisticated multi-page documentation
systems.

# Too technical, no user triggers
description: Implements the Project entity model with
hierarchical relationships.
```

Writing the main instructions

After the frontmatter, write the actual instructions in Markdown.

Recommended structure:

Adapt this template for your skill. Replace bracketed sections with your specific content.

```
---
name: your-skill
description: [...]
---

# Your Skill Name

## Instructions

### Step 1: [First Major Step]
Clear explanation of what happens.
```

````
Example:
```bash
python scripts/fetch_data.py --project-id PROJECT_ID
Expected output: [describe what success looks like]
````

(Add more steps as needed)

Examples

Example 1: [common scenario]

User says: "Set up a new marketing campaign"

Actions:

1. Fetch existing campaigns via MCP
2. Create new campaign with provided parameters

Result: Campaign created with confirmation link

(Add more examples as needed)

Troubleshooting

Error: [Common error message]

Cause: [Why it happens]

Solution: [How to fix]

(Add more error cases as needed)
````

Best Practices for Instructions

Be Specific and Actionable

✅ **Good:**

```
Run `python scripts/validate.py --input {filename}` to check
data format.
If validation fails, common issues include:
- Missing required fields (add them to the CSV)
- Invalid date formats (use YYYY-MM-DD)
```

❌ **Bad:**

```
Validate the data before proceeding.
```

Include error handling

```
## Common Issues

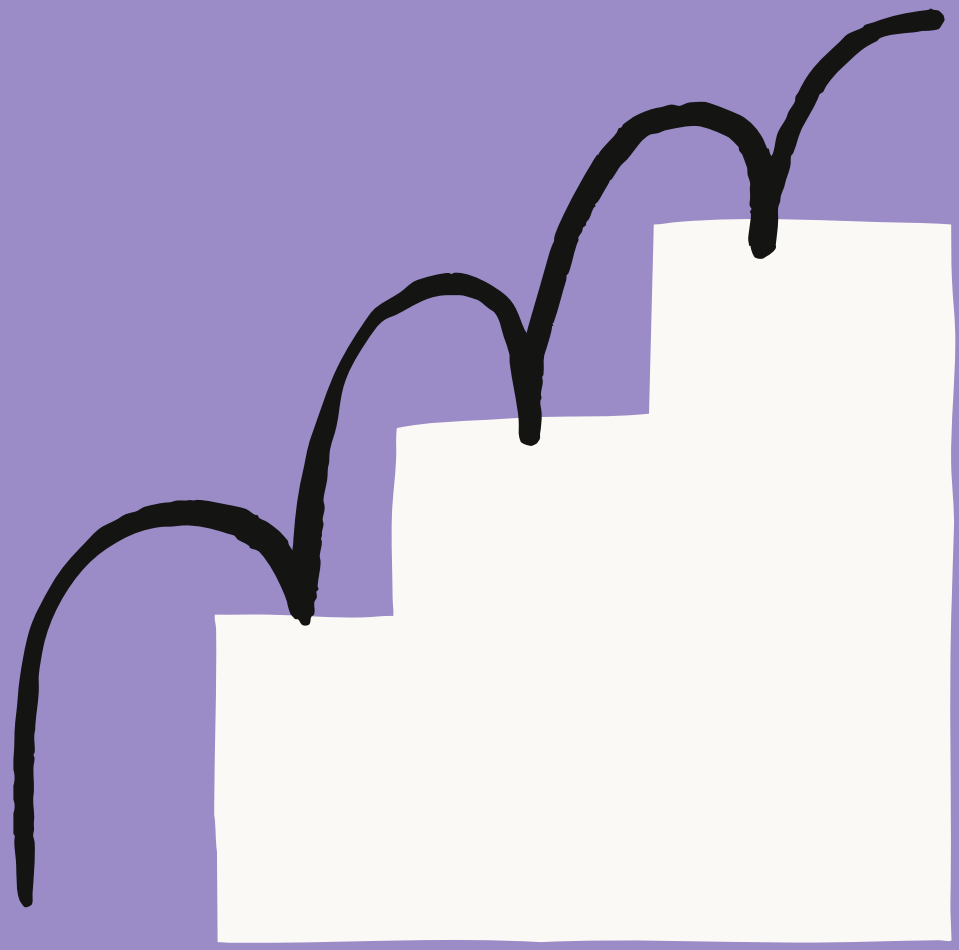
### MCP Connection Failed
If you see "Connection refused":
1. Verify MCP server is running: Check Settings > Extensions
2. Confirm API key is valid
3. Try reconnecting: Settings > Extensions > [Your Service] >
Reconnect
```

Reference bundled resources clearly

```
Before writing queries, consult `references/api-patterns.md`
for:
- Rate limiting guidance
- Pagination patterns
- Error codes and handling
```

Use progressive disclosure

Keep SKILL.md focused on core instructions. Move detailed documentation to `references/` and link to it. (See **Core Design Principles** for how the three-level system works.)



Chapter 3

Testing and iteration

Chapter 3

Testing and iteration

Skills can be tested at varying levels of rigor depending on your needs:

- **Manual testing in Claude.ai** - Run queries directly and observe behavior. Fast iteration, no setup required.
- **Scripted testing in Claude Code** - Automate test cases for repeatable validation across changes.
- **Programmatic testing via skills API** - Build evaluation suites that run systematically against defined test sets.

Choose the approach that matches your quality requirements and the visibility of your skill. A skill used internally by a small team has different testing needs than one deployed to thousands of enterprise users.

Pro Tip: Iterate on a single task before expanding

We've found that the most effective skill creators iterate on a single challenging task until Claude succeeds, then extract the winning approach into a skill. This leverages Claude's in-context learning and provides faster signal than broad testing. Once you have a working foundation, expand to multiple test cases for coverage.

Recommended Testing Approach

Based on early experience, effective skills testing typically covers three areas:

1. Triggering tests

Goal: Ensure your skill loads at the right times.

Test cases:

- ✅ Triggers on obvious tasks
- ✅ Triggers on paraphrased requests
- ❌ Doesn't trigger on unrelated topics

Example test suite:

```
Should trigger:
- "Help me set up a new ProjectHub workspace"
- "I need to create a project in ProjectHub"
- "Initialize a ProjectHub project for Q4 planning"

Should NOT trigger:
- "What's the weather in San Francisco?"
- "Help me write Python code"
- "Create a spreadsheet" (unless ProjectHub skill handles
sheets)
```

2. Functional tests

Goal: Verify the skill produces correct outputs.

Test cases:

- Valid outputs generated
- API calls succeed
- Error handling works
- Edge cases covered

Example:

```
Test: Create project with 5 tasks
Given: Project name "Q4 Planning", 5 task descriptions
When: Skill executes workflow
Then:
  - Project created in ProjectHub
  - 5 tasks created with correct properties
  - All tasks linked to project
  - No API errors
```

3. Performance comparison

Goal: Prove the skill improves results vs. baseline.

Use the metrics from **Define Success Criteria**. Here's what a comparison might look like.

Baseline comparison:

```
Without skill:
- User provides instructions each time
- 15 back-and-forth messages
- 3 failed API calls requiring retry
- 12,000 tokens consumed

With skill:
- Automatic workflow execution
- 2 clarifying questions only
- 0 failed API calls
- 6,000 tokens consumed
```

Using the skill-creator skill

The `skill-creator` skill - available in Claude.ai via plugin directory or download for Claude Code - can help you build and iterate on skills. If you have an MCP server and know your top 2–3 workflows, you can build and test a functional skill in a single sitting - often in 15–30 minutes.

Creating skills:

- Generate skills from natural language descriptions
- Produce properly formatted SKILL.md with frontmatter
- Suggest trigger phrases and structure

Reviewing skills:

- Flag common issues (vague descriptions, missing triggers, structural problems)
- Identify potential over/under-triggering risks
- Suggest test cases based on the skill's stated purpose

Iterative improvement:

- After using your skill and encountering edge cases or failures, bring those examples back to skill-creator
- Example: "Use the issues & solution identified in this chat to improve how the skill handles [specific edge case]"

To use:

```
"Use the skill-creator skill to help me build a skill for
[your use case]"
```

Note: skill-creator helps you design and refine skills but does not execute automated test suites or produce quantitative evaluation results.

Iteration based on feedback

Skills are living documents. Plan to iterate based on:

Undertriggering signals:

- Skill doesn't load when it should
- Users manually enabling it
- Support questions about when to use it

Solution: Add more detail and nuance to the description - this may include keywords particularly for technical terms

Overtriggering signals:

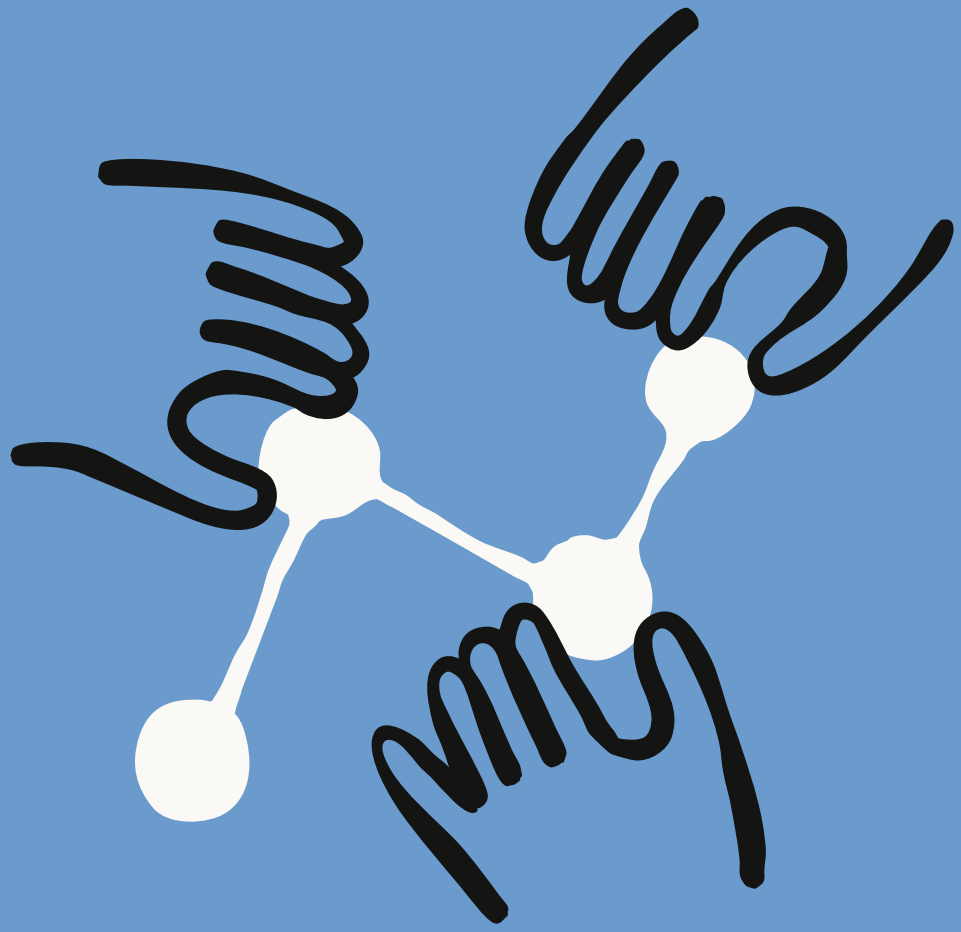
- Skill loads for irrelevant queries
- Users disabling it
- Confusion about purpose

Solution: Add negative triggers, be more specific

Execution issues:

- Inconsistent results
- API call failures
- User corrections needed

Solution: Improve instructions, add error handling



Chapter 4

Distribution and sharing

Chapter 4

Distribution and sharing

Skills make your MCP integration more complete. As users compare connectors, those with skills offer a faster path to value, giving you an edge over MCP-only alternatives.

Current distribution model (January 2026)

How individual users get skills:

1. Download the skill folder
2. Zip the folder (if needed)
3. Upload to Claude.ai via Settings > Capabilities > Skills
4. Or place in Claude Code skills directory

Organization-level skills:

- Admins can deploy skills workspace-wide (shipped December 18, 2025)
- Automatic updates
- Centralized management

An open standard

We've published **Agent Skills** as an open standard. Like MCP, we believe skills should be portable across tools and platforms - the same skill should work whether you're using Claude or other AI platforms. That said, some skills are designed to take full advantage of a specific platform's capabilities; authors can note this in the skill's `compatibility` field. We've been collaborating with members of the ecosystem on the standard, and we're excited by early adoption.

Using skills via API

For programmatic use cases - such as building applications, agents, or automated workflows that leverage skills - the API provides direct control over skill management and execution.

Key capabilities:

- `/v1/skills` endpoint for listing and managing skills
- Add skills to Messages API requests via the `container.skills` parameter
- Version control and management through the Claude Console
- Works with the Claude Agent SDK for building custom agents

When to use skills via the API vs. Claude.ai:

Use Case	Best Surface
End users interacting with skills directly	Claude.ai / Claude Code
Manual testing and iteration during development	Claude.ai / Claude Code
Individual, ad-hoc workflows	Claude.ai / Claude Code
Applications using skills programmatically	API
Production deployments at scale	API
Automated pipelines and agent systems	API

Note: Skills in the API require the Code Execution Tool beta, which provides the secure environment skills need to run.

For implementation details, see:

- Skills API Quickstart
- Create Custom skills
- Skills in the Agent SDK

Recommended approach today

Start by hosting your skill on GitHub with a public repo, clear README (for human visitors — this is separate from your skill folder, which should not contain a README.md), and example usage with screenshots. Then add a section to your MCP documentation that links to the skill, explains why using both together is valuable, and provides a quick-start guide.

1. Host on GitHub

- Public repo for open-source skills
- Clear README with installation instructions
- Example usage and screenshots

2. Document in Your MCP Repo

- Link to skills from MCP documentation
- Explain the value of using both together
- Provide quick-start guide

3. Create an Installation Guide

```
## Installing the [Your Service] skill

1. Download the skill:
   - Clone repo: `git clone https://github.com/yourcompany/
     skills`
   - Or download ZIP from Releases

2. Install in Claude:
   - Open Claude.ai > Settings > skills
   - Click "Upload skill"
   - Select the skill folder (zipped)

3. Enable the skill:
   - Toggle on the [Your Service] skill
   - Ensure your MCP server is connected

4. Test:
   - Ask Claude: "Set up a new project in [Your Service]"
```

Positioning your skill

How you describe your skill determines whether users understand its value and actually try it. When writing about your skill—in your README, documentation, or marketing - keep these principles in mind.

Focus on outcomes, not features:

✅ **Good:**

```
"The ProjectHub skill enables teams to set up complete project
workspaces in seconds — including pages, databases, and
templates — instead of spending 30 minutes on manual setup."
```

❌ **Bad:**

```
"The ProjectHub skill is a folder containing YAML frontmatter
and Markdown instructions that calls our MCP server tools."
```

Highlight the MCP + skills story:

```
"Our MCP server gives Claude access to your Linear projects.
Our skills teach Claude your team's sprint planning workflow.
Together, they enable AI-powered project management."
```

Chapter 5

Patterns and troubleshooting

Chapter 5

Patterns and troubleshooting

These patterns emerged from skills created by early adopters and internal teams. They represent common approaches we've seen work well, not prescriptive templates.

Choosing your approach: Problem-first vs. tool-first

Think of it like Home Depot. You might walk in with a problem - "I need to fix a kitchen cabinet" - and an employee points you to the right tools. Or you might pick out a new drill and ask how to use it for your specific job.

Skills work the same way:

- **Problem-first:** "I need to set up a project workspace" → Your skill orchestrates the right MCP calls in the right sequence. Users describe outcomes; the skill handles the tools.
- **Tool-first:** "I have Notion MCP connected" → Your skill teaches Claude the optimal workflows and best practices. Users have access; the skill provides expertise.

Most skills lean one direction. Knowing which framing fits your use case helps you choose the right pattern below.

Pattern 1: Sequential workflow orchestration

Use when: Your users need multi-step processes in a specific order.

Example structure:

```
## Workflow: Onboard New Customer

### Step 1: Create Account
Call MCP tool: `create_customer`
Parameters: name, email, company

### Step 2: Setup Payment
Call MCP tool: `setup_payment_method`
Wait for: payment method verification

### Step 3: Create Subscription
Call MCP tool: `create_subscription`
Parameters: plan_id, customer_id (from Step 1)

### Step 4: Send Welcome Email
Call MCP tool: `send_email`
Template: welcome_email_template
```

Key techniques:

- Explicit step ordering
- Dependencies between steps
- Validation at each stage
- Rollback instructions for failures

Pattern 2: Multi-MCP coordination

Use when: Workflows span multiple services.

Example: Design-to-development handoff

```
### Phase 1: Design Export (Figma MCP)
1. Export design assets from Figma
2. Generate design specifications
3. Create asset manifest

### Phase 2: Asset Storage (Drive MCP)
1. Create project folder in Drive
2. Upload all assets
3. Generate shareable links

### Phase 3: Task Creation (Linear MCP)
1. Create development tasks
2. Attach asset links to tasks
3. Assign to engineering team

### Phase 4: Notification (Slack MCP)
1. Post handoff summary to #engineering
2. Include asset links and task references
```

Key techniques:

- Clear phase separation
- Data passing between MCPs
- Validation before moving to next phase
- Centralized error handling

Pattern 3: Iterative refinement

Use when: Output quality improves with iteration.

Example: Report generation

```
## Iterative Report Creation

### Initial Draft
1. Fetch data via MCP
2. Generate first draft report
3. Save to temporary file

### Quality Check
1. Run validation script: `scripts/check_report.py`
2. Identify issues:
   - Missing sections
   - Inconsistent formatting
   - Data validation errors

### Refinement Loop
1. Address each identified issue
2. Regenerate affected sections
3. Re-validate
4. Repeat until quality threshold met

### Finalization
1. Apply final formatting
2. Generate summary
3. Save final version
```

Key techniques:

- Explicit quality criteria
- Iterative improvement
- Validation scripts
- Know when to stop iterating

Pattern 4: Context-aware tool selection

Use when: Same outcome, different tools depending on context.

Example: File storage

```
## Smart File Storage

### Decision Tree
1. Check file type and size
2. Determine best storage location:
   - Large files (>10MB): Use cloud storage MCP
   - Collaborative docs: Use Notion/Docs MCP
   - Code files: Use GitHub MCP
   - Temporary files: Use local storage

### Execute Storage
Based on decision:
- Call appropriate MCP tool
- Apply service-specific metadata
- Generate access link

### Provide Context to User
Explain why that storage was chosen
```

Key techniques:

- Clear decision criteria
- Fallback options
- Transparency about choices

Pattern 5: Domain-specific intelligence

Use when: Your skill adds specialized knowledge beyond tool access.

Example: Financial compliance

```
## Payment Processing with Compliance

### Before Processing (Compliance Check)
1. Fetch transaction details via MCP
2. Apply compliance rules:
   - Check sanctions lists
   - Verify jurisdiction allowances
   - Assess risk level
3. Document compliance decision

### Processing
IF compliance passed:
  - Call payment processing MCP tool
  - Apply appropriate fraud checks
  - Process transaction
ELSE:
  - Flag for review
  - Create compliance case

### Audit Trail
- Log all compliance checks
- Record processing decisions
- Generate audit report
```

Key techniques:

- Domain expertise embedded in logic
- Compliance before action
- Comprehensive documentation
- Clear governance

Troubleshooting

Skill won't upload

Error: "Could not find SKILL.md in uploaded folder"

Cause: File not named exactly SKILL.md

Solution:

- Rename to SKILL.md (case-sensitive)
- Verify with: ls -la should show SKILL.md

Error: "Invalid frontmatter"

Cause: YAML formatting issue

Common mistakes:

```
# Wrong - missing delimiters
name: my-skill
description: Does things

# Wrong - unclosed quotes
name: my-skill
description: "Does things

# Correct
---
name: my-skill
description: Does things
---
```

Error: "Invalid skill name"

Cause: Name has spaces or capitals

```
# Wrong
name: My Cool Skill

# Correct
name: my-cool-skill
```

Skill doesn't trigger

Symptom: Skill never loads automatically

Fix:

Revise your description field. See The Description Field for good/bad examples.

Quick checklist:

- Is it too generic? ("Helps with projects" won't work)
- Does it include trigger phrases users would actually say?
- Does it mention relevant file types if applicable?

Debugging approach:

Ask Claude: "When would you use the [skill name] skill?" Claude will quote the description back. Adjust based on what's missing.

Skill triggers too often

Symptom: Skill loads for unrelated queries

Solutions:

1. **Add negative triggers**

```
description: Advanced data analysis for CSV files. Use for
statistical modeling, regression, clustering. Do NOT use for
simple data exploration (use data-viz skill instead).
```

2. **Be more specific**

```
# Too broad
description: Processes documents

# More specific
description: Processes PDF legal documents for contract review
```

3. **Clarify scope**

```
description: PayFlow payment processing for e-commerce. Use
specifically for online payment workflows, not for general
financial queries.
```

MCP connection issues

Symptom: Skill loads but MCP calls fail

Checklist:

1. **Verify MCP server is connected**
   - Claude.ai: Settings > Extensions > [Your Service]
   - Should show "Connected" status
2. **Check authentication**
   - API keys valid and not expired
   - Proper permissions/scopes granted
   - OAuth tokens refreshed
3. **Test MCP independently**
   - Ask Claude to call MCP directly (without skill)
   - "Use [Service] MCP to fetch my projects"
   - If this fails, issue is MCP not skill
4. **Verify tool names**
   - Skill references correct MCP tool names
   - Check MCP server documentation
   - Tool names are case-sensitive

Instructions not followed

Symptom: Skill loads but Claude doesn't follow instructions

Common causes:

1. **Instructions too verbose**
   - Keep instructions concise
   - Use bullet points and numbered lists
   - Move detailed reference to separate files
2. **Instructions buried**
   - Put critical instructions at the top
   - Use ## Important or ## Critical headers
   - Repeat key points if needed
3. **Ambiguous language**

```
# Bad
Make sure to validate things properly

# Good
CRITICAL: Before calling create_project, verify:
- Project name is non-empty
- At least one team member assigned
- Start date is not in the past
```

Advanced technique: For critical validations, consider bundling a script that performs the checks programmatically rather than relying on language instructions. Code is deterministic; language interpretation isn't. See the **Office skills** for examples of this pattern.

4. **Model "laziness"** Add explicit encouragement:

```
## Performance Notes
- Take your time to do this thoroughly
- Quality is more important than speed
- Do not skip validation steps
```

Note: Adding this to user prompts is more effective than in SKILL.md

Large context issues

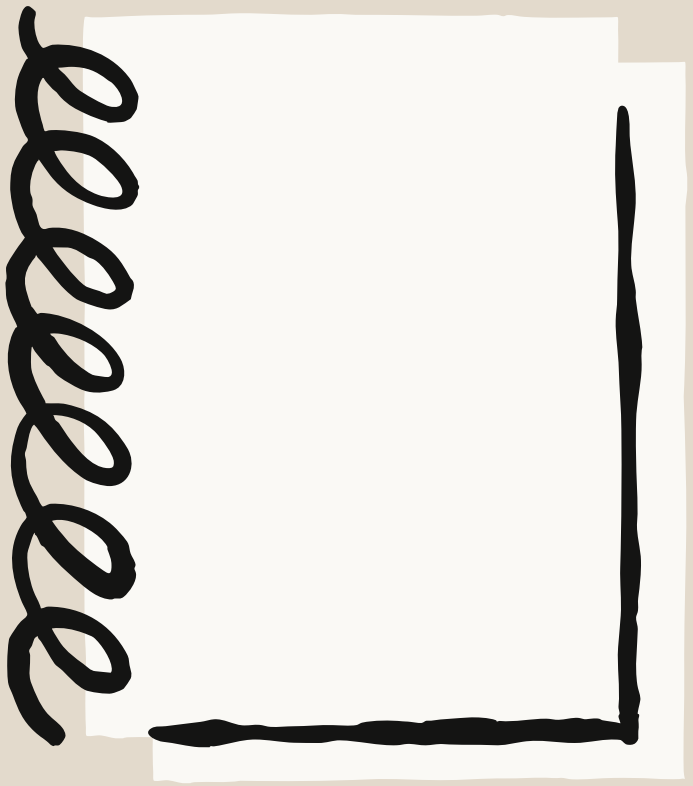
Symptom: Skill seems slow or responses degraded

Causes:

- Skill content too large
- Too many skills enabled simultaneously
- All content loaded instead of progressive disclosure

Solutions:

1. **Optimize SKILL.md size**
   - Move detailed docs to references/
   - Link to references instead of inline
   - Keep SKILL.md under 5,000 words
2. **Reduce enabled skills**
   - Evaluate if you have more than 20 - 50 skills enabled simultaneously
   - Recommend selective enablement
   - Consider skill "packs" for related capabilities



Chapter 6

Resources and references

Chapter 6

Resources and references

If you're building your first skill, start with the Best Practices Guide, then reference the API docs as needed.

Official Documentation

Anthropic Resources:

- Best Practices Guide
- Skills Documentation
- API Reference
- MCP Documentation

Blog Posts:

- Introducing Agent Skills
- Engineering Blog: Equipping Agents for the Real World
- Skills Explained
- How to Create Skills for Claude
- Building Skills for Claude Code
- Improving Frontend Design through Skills

Example skills

Public skills repository:

- GitHub: **anthropics/skills**
- Contains Anthropic-created skills you can customize

Tools and Utilities

skill-creator skill:

- Built into Claude.ai and available for Claude Code
- Can generate skills from descriptions
- Reviews and provides recommendations
- Use: "Help me build a skill using skill-creator"

Validation:

- skill-creator can assess your skills
- Ask: "Review this skill and suggest improvements"

Getting Support

For Technical Questions:

- General questions: Community forums at the **Claude Developers Discord**

For Bug Reports:

- GitHub Issues: **anthropics/skills/issues**
- Include: Skill name, error message, steps to reproduce

Reference A: Quick checklist

Use this checklist to validate your skill before and after upload. If you want a faster start, use the skill-creator skill to generate your first draft, then run through this list to make sure you haven't missed anything.

Before you start

- [ ] Identified 2-3 concrete use cases
- [ ] Tools identified (built-in or MCP)
- [ ] Reviewed this guide and example skills
- [ ] Planned folder structure

During development

- [ ] Folder named in kebab-case
- [ ] SKILL.md file exists (exact spelling)
- [ ] YAML frontmatter has --- delimiters
- [ ] name field: kebab-case, no spaces, no capitals
- [ ] description includes WHAT and WHEN
- [ ] No XML tags (< >) anywhere
- [ ] Instructions are clear and actionable
- [ ] Error handling included
- [ ] Examples provided
- [ ] References clearly linked

Before upload

- [ ] Tested triggering on obvious tasks
- [ ] Tested triggering on paraphrased requests
- [ ] Verified doesn't trigger on unrelated topics
- [ ] Functional tests pass
- [ ] Tool integration works (if applicable)
- [ ] Compressed as .zip file

After upload

- [ ] Test in real conversations
- [ ] Monitor for under/over-triggering
- [ ] Collect user feedback
- [ ] Iterate on description and instructions
- [ ] Update version in metadata

Reference B: YAML frontmatter

Required fields

```
---
name: skill-name-in-kebab-case
description: What it does and when to use it. Include specific
trigger phrases.
---
```

All optional fields

```
name: skill-name
description: [required description]
license: MIT # Optional: License for open-source
allowed-tools: "Bash(python:*) Bash(npm:*) WebFetch" # Optional:
Restrict tool access
metadata: # Optional: Custom fields
  author: Company Name
  version: 1.0.0
  mcp-server: server-name
  category: productivity
  tags: [project-management, automation]
  documentation: https://example.com/docs
  support: support@example.com
```

Security notes

Allowed:

- Any standard YAML types (strings, numbers, booleans, lists, objects)
- Custom metadata fields
- Long descriptions (up to 1024 characters)

Forbidden:

- XML angle brackets (< >) - security restriction
- Code execution in YAML (uses safe YAML parsing)
- Skills named with "claude" or "anthropic" prefix (reserved)

Reference C: Complete skill examples

For full, production-ready skills demonstrating the patterns in this guide:

- Document Skills - **PDF**, **DOCX**, **PPTX**, **XLSX** creation
- **Example Skills** - Various workflow patterns
- **Partner Skills Directory** - View skills from various partners such as Asana, Atlassian, Canva, Figma, Sentry, Zapier, and more

These repositories stay up-to-date and include additional examples beyond what's covered here. Clone them, modify them for your use case, and use them as templates.

Claude

claude.ai