Skill authoring best practices

Learn how to write effective Skills that Claude can discover and use successfully.

Copy page

Good Skills are concise, well-structured, and tested with real usage. This guide provides practical authoring decisions to help you write Skills that Claude can discover and use effectively.

For conceptual background on how Skills work, see the Skills overview.

Core principles

Concise is key

The context window is a public good. Your Skill shares the context window with everything else Claude needs to know, including:

- The system prompt
- Conversation history
- Other Skills' metadata
- Your actual request

Not every token in your Skill has an immediate cost. At startup, only the metadata (name and description) from all Skills is pre-loaded. Claude reads SKILL.md only when the Skill becomes relevant, and reads additional files only as needed. However, being concise in SKILL.md still matters: once Claude loads it, every token competes with conversation history and other context.

Default assumption: Claude is already very smart

Only add context Claude doesn't already have. Challenge each piece of information:

- "Does Claude really need this explanation?"
- "Can I assume Claude knows this?"

~~Good example: Concise (approximately 50 tokens):~~

```
## Extract PDF text

Use pdfplumber for text extraction:

```python
import pdfplumber

with pdfplumber.open("file.pdf") as pdf:
    text = pdf.pages[0].extract_text()
```
```

Bad example: Too verbose (approximately 150 tokens):

```
## Extract PDF text

PDF (Portable Document Format) files are a common file format that contains
text, images, and other content. To extract text from a PDF, you'll need to
use a library. There are many libraries available for PDF processing, but we
recommend pdfplumber because it's easy to use and handles most cases well.
First, you'll need to install it using pip. Then you can use the code below...
```

The concise version assumes Claude knows what PDFs are and how libraries work.

Set appropriate degrees of freedom

Match the level of specificity to the task's fragility and variability.

High freedom (text-based instructions):

Use when:

- Multiple approaches are valid
- Decisions depend on context
- Heuristics guide the approach

Example:

```
1. Analyze the code structure and organization
2. Check for potential bugs or edge cases
3. Suggest improvements for readability and maintainability
4. Verify adherence to project conventions
```

Medium freedom (pseudocode or scripts with parameters):

Use when:

- A preferred pattern exists
- Some variation is acceptable
- Configuration affects behavior

Example:

````
## Generate report

Use this template and customize as needed:

```python
def generate_report(data, format="markdown", include_charts=True):
    # Process data
    # Generate output in specified format
    # Optionally include visualizations
```
````

Low freedom (specific scripts, few or no parameters):

Use when:

- Operations are fragile and error-prone
- Consistency is critical
- A specific sequence must be followed

Example:

````
Run exactly this script:

```bash
python scripts/migrate.py --verify --backup
```

Do not modify the command or add additional flags.
````

Analogy: Think of Claude as a robot exploring a path:

- **Narrow bridge with cliffs on both sides**: There's only one safe way forward. Provide specific guardrails and exact instructions (low freedom). Example: database migrations that must run in exact sequence.
- **Open field with no hazards**: Many paths lead to success. Give general direction and trust Claude to find the best route (high freedom). Example: code reviews where context determines the best approach.

Test with all models you plan to use

Skills act as additions to models, so effectiveness depends on the underlying model. Test your Skill with all the models you plan to use it with.

Testing considerations by model:

- **Claude Haiku** (fast, economical): Does the Skill provide enough guidance?
- **Claude Sonnet** (balanced): Is the Skill clear and efficient?
- **Claude Opus** (powerful reasoning): Does the Skill avoid over-explaining?

What works perfectly for Opus might need more detail for Haiku. If you plan to use your Skill across multiple models, aim for instructions that work well with all of them.

Skill structure

YAML Frontmatter: The SKILL.md frontmatter requires two fields:

`name :`

- Maximum 64 characters
- Must contain only lowercase letters, numbers, and hyphens
- Cannot contain XML tags

- Must be non-empty
- Maximum 1024 characters
- Cannot contain XML tags
- Should describe what the Skill does and when to use it

For complete Skill structure details, see the Skills overview.

Naming conventions

Use consistent naming patterns to make Skills easier to reference and discuss. We recommend using **gerund form** (verb + -ing) for Skill names, as this clearly describes the activity or capability the Skill provides.

Remember that the `name` field must use lowercase letters, numbers, and hyphens only.

Good naming examples (gerund form):

- `processing-pdfs`
- `analyzing-spreadsheets`
- `managing-databases`
- `testing-code`
- `writing-documentation`

Acceptable alternatives:

- Noun phrases: `pdf-processing` , `spreadsheet-analysis`
- Action-oriented: `process-pdfs` , `analyze-spreadsheets`

Avoid:

- Vague names: `helper` , `utils` , `tools`
- Overly generic: `documents` , `data` , `files`
- Reserved words: `anthropic-helper` , `claude-tools`
- Inconsistent patterns within your skill collection

Consistent naming makes it easier to:

- Reference Skills in documentation and conversations
- Understand what a Skill does at a glance

Writing effective descriptions

The `description` field enables Skill discovery and should include both what the Skill does and when to use it.

> ⚠ **Always write in third person**. The description is injected into the system prompt, and inconsistent point-of-view can cause discovery problems.
>
> - **Good:** "Processes Excel files and generates reports"
> - **Avoid:** "I can help you process Excel files"
> - **Avoid:** "You can use this to process Excel files"

Be specific and include key terms. Include both what the Skill does and specific triggers/contexts for when to use it.

Each Skill has exactly one description field. The description is critical for skill selection: Claude uses it to choose the right Skill from potentially 100+ available Skills. Your description must provide enough detail for Claude to know when to select this Skill, while the rest of SKILL.md provides the implementation details.

Effective examples:

PDF Processing skill:

```
description: Extract text and tables from PDF files, fill forms, merge dc en
```

Excel Analysis skill:

```
description: Analyze Excel spreadsheets, create pivot tables, generate ch s.
```

Git Commit Helper skill:

```
description: Generate descriptive commit messages by analyzing git diffs. e
```


Avoid vague descriptions like these:

```
description: Processes data
```

```
description: Does stuff with files
```

Progressive disclosure patterns

SKILL.md serves as an overview that points Claude to detailed materials as needed, like a table of contents in an onboarding guide. For an explanation of how progressive disclosure works, see How Skills work in the overview.

Practical guidance:

- Keep SKILL.md body under 500 lines for optimal performance
- Split content into separate files when approaching this limit
- Use the patterns below to organize instructions, code, and resources effectively

Visual overview: From simple to complex

A basic Skill starts with just a SKILL.md file containing metadata and instructions:

Claude Docs

The complete Skill directory structure might look like this:

```
pdf/
├── SKILL.md              # Main instructions (loaded when triggered)
├── FORMS.md              # Form-filling guide (loaded as needed)
├── reference.md          # API reference (loaded as needed)
├── examples.md           # Usage examples (loaded as needed)
└── scripts/
    ├── analyze_form.py   # Utility script (executed, not loaded)
    ├── fill_form.py      # Form filling script
    └── validate.py       # Validation script
```

Pattern 1: High-level guide with references

Ask Docs

````
description: Extracts text and tables from PDF files, fills forms, and merges
---

# PDF Processing

## Quick start

Extract text with pdfplumber:
```python
import pdfplumber
with pdfplumber.open("file.pdf") as pdf:
    text = pdf.pages[0].extract_text()
```

## Advanced features

**Form filling**: See [FORMS.md](FORMS.md) for complete guide
**API reference**: See [REFERENCE.md](REFERENCE.md) for all methods
**Examples**: See [EXAMPLES.md](EXAMPLES.md) for common patterns
````

Claude loads FORMS.md, REFERENCE.md, or EXAMPLES.md only when needed.

Pattern 2: Domain-specific organization

For Skills with multiple domains, organize content by domain to avoid loading irrelevant context. When a user asks about sales metrics, Claude only needs to read sales-related schemas, not finance or marketing data. This keeps token usage low and context focused.

```
bigquery-skill/
├── SKILL.md (overview and navigation)
└── reference/
    ├── finance.md (revenue, billing metrics)
    ├── sales.md (opportunities, pipeline)
    ├── product.md (API usage, features)
    └── marketing.md (campaigns, attribution)
```

SKILL.md

```
## Available datasets

**Finance**: Revenue, ARR, billing → See [reference/finance.md](reference/fina
**Sales**: Opportunities, pipeline, accounts → See [reference/sales.md](refere
**Product**: API usage, features, adoption → See [reference/product.md](refere
**Marketing**: Campaigns, attribution, email → See [reference/marketing.md](re

## Quick search

Find specific metrics using grep:

```bash
grep -i "revenue" reference/finance.md
grep -i "pipeline" reference/sales.md
grep -i "api usage" reference/product.md
```
```

Pattern 3: Conditional details

Show basic content, link to advanced content:

```
# DOCX Processing

## Creating documents

Use docx-js for new documents. See [DOCX-JS.md](DOCX-JS.md).

## Editing documents

For simple edits, modify the XML directly.

**For tracked changes**: See [REDLINING.md](REDLINING.md)
**For OOXML details**: See [OOXML.md](OOXML.md)
```

Claude reads REDLINING.md or OOXML.md only when the user needs those features.

Avoid deeply nested references

Claude may partially read files when they're referenced from other referenced files. When encountering nested references, Claude might use commands like `head -100` to preview content rather than reading entire files, resulting in incomplete information.

Bad example: Too deep:

```
# SKILL.md
See [advanced.md](advanced.md)...

# advanced.md
See [details.md](details.md)...

# details.md
Here's the actual information...
```

Good example: One level deep:

```
# SKILL.md

**Basic usage**: [instructions in SKILL.md]
**Advanced features**: See [advanced.md](advanced.md)
**API reference**: See [reference.md](reference.md)
**Examples**: See [examples.md](examples.md)
```

Structure longer reference files with table of contents

For reference files longer than 100 lines, include a table of contents at the top. This ensures Claude can see the full scope of available information even when previewing with partial reads.

Example:

```
## Contents
- Authentication and setup
- Core methods (create, read, update, delete)
- Advanced features (batch operations, webhooks)
- Error handling patterns
- Code examples

## Authentication and setup
...

## Core methods
...
```

Claude can then read the complete file or jump to specific sections as needed.

For details on how this filesystem-based architecture enables progressive disclosure, see the Runtime environment section in the Advanced section below.

Workflows and feedback loops

Use workflows for complex tasks

Break complex operations into clear, sequential steps. For particularly complex workflows, provide a checklist that Claude can copy into its response and check off as it progresses.

Example 1: Research synthesis workflow (for Skills without code):

````
Copy this checklist and track your progress:

```
Research Progress:
- [ ] Step 1: Read all source documents
- [ ] Step 2: Identify key themes
- [ ] Step 3: Cross-reference claims
- [ ] Step 4: Create structured summary
- [ ] Step 5: Verify citations
```

**Step 1: Read all source documents**

Review each document in the `sources/` directory. Note the main arguments and

**Step 2: Identify key themes**

Look for patterns across sources. What themes appear repeatedly? Where do sour

**Step 3: Cross-reference claims**

For each major claim, verify it appears in the source material. Note which sou

**Step 4: Create structured summary**

Organize findings by theme. Include:
- Main claim
- Supporting evidence from sources
- Conflicting viewpoints (if any)

**Step 5: Verify citations**

Check that every claim references the correct source document. If citations ar
````

This example shows how workflows apply to analysis tasks that don't require code. The checklist pattern works for any complex, multi-step process.

Example 2: PDF form filling workflow (for Skills with code):

````
Copy this checklist and check off items as you complete them:

```
Task Progress:
- [ ] Step 1: Analyze the form (run analyze_form.py)
- [ ] Step 2: Create field mapping (edit fields.json)
- [ ] Step 3: Validate mapping (run validate_fields.py)
- [ ] Step 4: Fill the form (run fill_form.py)
- [ ] Step 5: Verify output (run verify_output.py)
```

**Step 1: Analyze the form**

Run: `python scripts/analyze_form.py input.pdf`

This extracts form fields and their locations, saving to `fields.json`.

**Step 2: Create field mapping**

Edit `fields.json` to add values for each field.

**Step 3: Validate mapping**

Run: `python scripts/validate_fields.py fields.json`

Fix any validation errors before continuing.

**Step 4: Fill the form**

Run: `python scripts/fill_form.py input.pdf fields.json output.pdf`

**Step 5: Verify output**

Run: `python scripts/verify_output.py output.pdf`

If verification fails, return to Step 2.
````

Clear steps prevent Claude from skipping critical validation. The checklist helps both Claude and you track progress through multi-step workflows.

Implement feedback loops

Common pattern: Run validator → fix errors → repeat

This pattern greatly improves output quality.

```
## Content review process

1. Draft your content following the guidelines in STYLE_GUIDE.md
2. Review against the checklist:
   - Check terminology consistency
   - Verify examples follow the standard format
   - Confirm all required sections are present
3. If issues found:
   - Note each issue with specific section reference
   - Revise the content
   - Review the checklist again
4. Only proceed when all requirements are met
5. Finalize and save the document
```

This shows the validation loop pattern using reference documents instead of scripts. The "validator" is STYLE_GUIDE.md, and Claude performs the check by reading and comparing.

Example 2: Document editing process (for Skills with code):

```
## Document editing process

1. Make your edits to `word/document.xml`
2. **Validate immediately**: `python ooxml/scripts/validate.py unpacked_dir/`
3. If validation fails:
   - Review the error message carefully
   - Fix the issues in the XML
   - Run validation again
4. **Only proceed when validation passes**
5. Rebuild: `python ooxml/scripts/pack.py unpacked_dir/ output.docx`
6. Test the output document
```

The validation loop catches errors early.

Content guidelines

Avoid time-sensitive information

Don't include information that will become outdated:

Bad example: Time-sensitive (will become wrong):

Good example (use "old patterns" section):

```
## Current method

Use the v2 API endpoint: `api.example.com/v2/messages`

## Old patterns

<details>
<summary>Legacy v1 API (deprecated 2025-08)</summary>

The v1 API used: `api.example.com/v1/messages`

This endpoint is no longer supported.
</details>
```

The old patterns section provides historical context without cluttering the main content.

Use consistent terminology

Choose one term and use it throughout the Skill:

Good - Consistent:

- Always "API endpoint"
- Always "field"
- Always "extract"

Bad - Inconsistent:

- Mix "API endpoint", "URL", "API route", "path"
- Mix "field", "box", "element", "control"
- Mix "extract", "pull", "get", "retrieve"

Consistency helps Claude understand and follow instructions.

Common patterns

Provide templates for output format. Match the level of strictness to your needs.

For strict requirements (like API responses or data formats):

````
## Report structure

ALWAYS use this exact template structure:

```markdown
# [Analysis Title]

## Executive summary
[One-paragraph overview of key findings]

## Key findings
- Finding 1 with supporting data
- Finding 2 with supporting data
- Finding 3 with supporting data

## Recommendations
1. Specific actionable recommendation
2. Specific actionable recommendation
```
````

For flexible guidance (when adaptation is useful):

````
Here is a sensible default format, but use your best judgment based on the ana

```markdown
# [Analysis Title]

## Executive summary
[Overview]

## Key findings
[Adapt sections based on what you discover]

## Recommendations
[Tailor to the specific context]
```

Adjust sections as needed for the specific analysis type.
````

Examples pattern

For Skills where output quality depends on seeing examples, provide input/output pairs just like in regular prompting:

````
Generate commit messages following these examples:

**Example 1:**
Input: Added user authentication with JWT tokens
Output:
```
feat(auth): implement JWT-based authentication

Add login endpoint and token validation middleware
```

**Example 2:**
Input: Fixed bug where dates displayed incorrectly in reports
Output:
```
fix(reports): correct date formatting in timezone conversion

Use UTC timestamps consistently across report generation
```

**Example 3:**
Input: Updated dependencies and refactored error handling
Output:
```
chore: update dependencies and refactor error handling

- Upgrade lodash to 4.17.21
- Standardize error response format across endpoints
```

Follow this style: type(scope): brief description, then detailed explanation.
````

Examples help Claude understand the desired style and level of detail more clearly than descriptions alone.

Conditional workflow pattern

Guide Claude through decision points:

```
1. Determine the modification type:

   **Creating new content?** → Follow "Creation workflow" below
   **Editing existing content?** → Follow "Editing workflow" below

2. Creation workflow:
   - Use docx-js library
   - Build document from scratch
   - Export to .docx format

3. Editing workflow:
   - Unpack existing document
   - Modify XML directly
   - Validate after each change
   - Repack when complete
```

If workflows become large or complicated with many steps, consider pushing them into separate files and tell Claude to read the appropriate file based on the task at hand.

Evaluation and iteration

Build evaluations first

Create evaluations BEFORE writing extensive documentation. This ensures your Skill solves real problems rather than documenting imagined ones.

Evaluation-driven development:

1. **Identify gaps**: Run Claude on representative tasks without a Skill. Document specific failures or missing context
2. **Create evaluations**: Build three scenarios that test these gaps
3. **Establish baseline**: Measure Claude's performance without the Skill
4. **Write minimal instructions**: Create just enough content to address the gaps and pass evaluations
5. **Iterate**: Execute evaluations, compare against baseline, and refine

This approach ensures you're solving actual problems rather than anticipating requirements that may never materialize.

Evaluation structure:

```
  "query": "Extract all text from this PDF file and save it to output.txt",
  "files": ["test-files/document.pdf"],
  "expected_behavior": [
    "Successfully reads the PDF file using an appropriate PDF processing libra
    "Extracts text content from all pages in the document without missing any
    "Saves the extracted text to a file named output.txt in a clear, readable
  ]
}
```

This example demonstrates a data-driven evaluation with a simple testing rubric. We do not currently provide a built-in way to run these evaluations. Users can create their own evaluation system. Evaluations are your source of truth for measuring Skill effectiveness.

Develop Skills iteratively with Claude

The most effective Skill development process involves Claude itself. Work with one instance of Claude ("Claude A") to create a Skill that will be used by other instances ("Claude B"). Claude A helps you design and refine instructions, while Claude B tests them in real tasks. This works because Claude models understand both how to write effective agent instructions and what information agents need.

Creating a new Skill:

1. **Complete a task without a Skill**: Work through a problem with Claude A using normal prompting. As you work, you'll naturally provide context, explain preferences, and share procedural knowledge. Notice what information you repeatedly provide.
2. **Identify the reusable pattern**: After completing the task, identify what context you provided that would be useful for similar future tasks.

   Example: If you worked through a BigQuery analysis, you might have provided table names, field definitions, filtering rules (like "always exclude test accounts"), and common query patterns.
3. **Ask Claude A to create a Skill**: "Create a Skill that captures this BigQuery analysis pattern we just used. Include the table schemas, naming conventions, and the rule about filtering test accounts."

Claude models understand the Skill format and structure natively. You don't need special system prompts or a "writing skills" skill to get Claude to help create Skills.

4. **Review for conciseness**: Check that Claude A hasn't added unnecessary explanations. Ask: "Remove the explanation about what win rate means - Claude already knows that."
5. **Improve information architecture**: Ask Claude A to organize the content more effectively. For example: "Organize this so the table schema is in a separate reference file. We might add more tables later."
6. **Test on similar tasks**: Use the Skill with Claude B (a fresh instance with the Skill loaded) on related use cases. Observe whether Claude B finds the right information, applies rules correctly, and handles the task successfully.
7. **Iterate based on observation**: If Claude B struggles or misses something, return to Claude A with specifics: "When Claude used this Skill, it forgot to filter by date for Q4. Should we add a section about date filtering patterns?"

Iterating on existing Skills:

The same hierarchical pattern continues when improving Skills. You alternate between:

- **Working with Claude A** (the expert who helps refine the Skill)
- **Testing with Claude B** (the agent using the Skill to perform real work)
- **Observing Claude B's behavior** and bringing insights back to Claude A

1. **Use the Skill in real workflows**: Give Claude B (with the Skill loaded) actual tasks, not test scenarios
2. **Observe Claude B's behavior**: Note where it struggles, succeeds, or makes unexpected choices

   Example observation: "When I asked Claude B for a regional sales report, it wrote the query but forgot to filter out test accounts, even though the Skill mentions this rule."
3. **Return to Claude A for improvements**: Share the current SKILL.md and describe what you observed. Ask: "I noticed Claude B forgot to filter test accounts when I asked for a regional report. The Skill mentions filtering, but maybe it's not prominent enough?"
4. **Review Claude A's suggestions**: Claude A might suggest reorganizing to make rules more prominent, using stronger language like "MUST filter" instead of "always filter", or restructuring the workflow section.
5. **Apply and test changes**: Update the Skill with Claude A's refinements, then test again with Claude B on similar requests
6. **Repeat based on usage**: Continue this observe-refine-test cycle as you encounter new scenarios. Each iteration improves the Skill based on real agent behavior, not

1. Share Skills with teammates and observe their usage
2. Ask: Does the Skill activate when expected? Are instructions clear? What's missing?
3. Incorporate feedback to address blind spots in your own usage patterns

Why this approach works: Claude A understands agent needs, you provide domain expertise, Claude B reveals gaps through real usage, and iterative refinement improves Skills based on observed behavior rather than assumptions.

Observe how Claude navigates Skills

As you iterate on Skills, pay attention to how Claude actually uses them in practice. Watch for:

- **Unexpected exploration paths**: Does Claude read files in an order you didn't anticipate? This might indicate your structure isn't as intuitive as you thought
- **Missed connections**: Does Claude fail to follow references to important files? Your links might need to be more explicit or prominent
- **Overreliance on certain sections**: If Claude repeatedly reads the same file, consider whether that content should be in the main SKILL.md instead
- **Ignored content**: If Claude never accesses a bundled file, it might be unnecessary or poorly signaled in the main instructions

Iterate based on these observations rather than assumptions. The 'name' and 'description' in your Skill's metadata are particularly critical. Claude uses these when deciding whether to trigger the Skill in response to the current task. Make sure they clearly describe what the Skill does and when it should be used.

Anti-patterns to avoid

Avoid Windows-style paths

Always use forward slashes in file paths, even on Windows:

- ✓ **Good**: `scripts/helper.py , reference/guide.md`
- ✗ **Avoid**: `scripts\helper.py , reference\guide.md`

Unix-style paths work across all platforms, while Windows-style paths cause errors on Unix systems.

Don't present multiple approaches unless necessary:

````
**Bad example: Too many choices** (confusing):
"You can use pypdf, or pdfplumber, or PyMuPDF, or pdf2image, or..."

**Good example: Provide a default** (with escape hatch):
"Use pdfplumber for text extraction:
```python
import pdfplumber
```

For scanned PDFs requiring OCR, use pdf2image with pytesseract instead."
````

Advanced: Skills with executable code

The sections below focus on Skills that include executable scripts. If your Skill uses only markdown instructions, skip to Checklist for effective Skills.

Solve, don't punt

When writing scripts for Skills, handle error conditions rather than punting to Claude.

Good example: Handle errors explicitly:

```python
def process_file(path):
    """Process a file, creating it if it doesn't exist."""
    try:
        with open(path) as f:
            return f.read()
    except FileNotFoundError:
        # Create file with default content instead of failing
        print(f"File {path} not found, creating default")
        with open(path, 'w') as f:
            f.write('')
        return ''
    except PermissionError:
        # Provide alternative instead of failing
        print(f"Cannot access {path}, using default")
        return ''
```


Bad example: Punt to Claude:

```
    return open(path).read()
```

Configuration parameters should also be justified and documented to avoid "voodoo constants" (Ousterhout's law). If you don't know the right value, how will Claude determine it?

Good example: Self-documenting:

```
# HTTP requests typically complete within 30 seconds
# Longer timeout accounts for slow connections
REQUEST_TIMEOUT = 30

# Three retries balances reliability vs speed
# Most intermittent failures resolve by the second retry
MAX_RETRIES = 3
```

Bad example: Magic numbers:

```
TIMEOUT = 47  # Why 47?
RETRIES = 5   # Why 5?
```

Provide utility scripts

Even if Claude could write a script, pre-made scripts offer advantages:

Benefits of utility scripts:

- More reliable than generated code
- Save tokens (no need to include code in context)
- Save time (no code generation required)
- Ensure consistency across uses

Claude Docs

The diagram above shows how executable scripts work alongside instruction files. The instruction file (forms.md) references the script, and Claude can execute it without loading its contents into context.

Important distinction: Make clear in your instructions whether Claude should:

- **Execute the script** (most common): "Run `analyze_form.py` to extract fields"
- **Read it as reference** (for complex logic): "See `analyze_form.py` for the field extraction algorithm"

For most utility scripts, execution is preferred because it's more reliable and efficient. See the Runtime environment section below for details on how script execution works.

Example:

Ask Docs

````
**analyze_form.py**: Extract all form fields from PDF

```bash
python scripts/analyze_form.py input.pdf > fields.json
```

Output format:
```json
{
  "field_name": {"type": "text", "x": 100, "y": 200},
  "signature": {"type": "sig", "x": 150, "y": 500}
}
```

**validate_boxes.py**: Check for overlapping bounding boxes

```bash
python scripts/validate_boxes.py fields.json
# Returns: "OK" or lists conflicts
```

**fill_form.py**: Apply field values to PDF

```bash
python scripts/fill_form.py input.pdf fields.json output.pdf
```
````

Use visual analysis

When inputs can be rendered as images, have Claude analyze them:

````
## Form layout analysis

1. Convert PDF to images:
   ```bash
   python scripts/pdf_to_images.py form.pdf
   ```

2. Analyze each page image to identify form fields
3. Claude can see field locations and types visually
````


In this example, you'd need to write the `pdf_to_images.py` script.

Create verifiable intermediate outputs

When Claude performs complex, open-ended tasks, it can make mistakes. The "plan-validate-execute" pattern catches errors early by having Claude first create a plan in a structured format, then validate that plan with a script before executing it.

Example: Imagine asking Claude to update 50 form fields in a PDF based on a spreadsheet. Without validation, Claude might reference non-existent fields, create conflicting values, miss required fields, or apply updates incorrectly.

Solution: Use the workflow pattern shown above (PDF form filling), but add an intermediate `changes.json` file that gets validated before applying changes. The workflow becomes: analyze → **create plan file** → **validate plan** → execute → verify.

Why this pattern works:

- **Catches errors early**: Validation finds problems before changes are applied
- **Machine-verifiable**: Scripts provide objective verification
- **Reversible planning**: Claude can iterate on the plan without touching originals
- **Clear debugging**: Error messages point to specific problems

When to use: Batch operations, destructive changes, complex validation rules, high-stakes operations.

Implementation tip: Make validation scripts verbose with specific error messages like "Field 'signature_date' not found. Available fields: customer_name, order_total, signature_date_signed" to help Claude fix issues.

Package dependencies

Skills run in the code execution environment with platform-specific limitations:

- **claude.ai**: Can install packages from npm and PyPI and pull from GitHub repositories
- **Anthropic API**: Has no network access and no runtime package installation

List required packages in your SKILL.md and verify they're available in the code execution tool documentation.

Runtime environment

Skills run in a code execution environment with filesystem access, bash commands, and code execution capabilities. For the conceptual explanation of this architecture, see The

How this affects your authoring:

How Claude accesses Skills:

1. **Metadata pre-loaded**: At startup, the name and description from all Skills' YAML frontmatter are loaded into the system prompt
2. **Files read on-demand**: Claude uses bash Read tools to access SKILL.md and other files from the filesystem when needed
3. **Scripts executed efficiently**: Utility scripts can be executed via bash without loading their full contents into context. Only the script's output consumes tokens
4. **No context penalty for large files**: Reference files, data, or documentation don't consume context tokens until actually read

- **File paths matter**: Claude navigates your skill directory like a filesystem. Use forward slashes (`reference/guide.md`), not backslashes
- **Name files descriptively**: Use names that indicate content: `form_validation_rules.md` , not `doc2.md`
- **Organize for discovery**: Structure directories by domain or feature
  - Good: `reference/finance.md` , `reference/sales.md`
  - Bad: `docs/file1.md` , `docs/file2.md`
- **Bundle comprehensive resources**: Include complete API docs, extensive examples, large datasets; no context penalty until accessed
- **Prefer scripts for deterministic operations**: Write `validate_form.py` rather than asking Claude to generate validation code
- **Make execution intent clear**:
  - "Run `analyze_form.py` to extract fields" (execute)
  - "See `analyze_form.py` for the extraction algorithm" (read as reference)
- **Test file access patterns**: Verify Claude can navigate your directory structure by testing with real requests

Example:

```
bigquery-skill/
├── SKILL.md (overview, points to reference files)
└── reference/
    ├── finance.md (revenue metrics)
    ├── sales.md (pipeline data)
    └── product.md (usage analytics)
```

product.md files remain on the filesystem, consuming zero context tokens until needed. This filesystem-based model is what enables progressive disclosure. Claude can navigate and selectively load exactly what each task requires.

For complete details on the technical architecture, see How Skills work in the Skills overview.

MCP tool references

If your Skill uses MCP (Model Context Protocol) tools, always use fully qualified tool names to avoid "tool not found" errors.

Format: `ServerName:tool_name`

Example:

```
Use the BigQuery:bigquery_schema tool to retrieve table schemas.
Use the GitHub:create_issue tool to create issues.
```

Where:

- `BigQuery` and `GitHub` are MCP server names
- `bigquery_schema` and `create_issue` are the tool names within those servers

Without the server prefix, Claude may fail to locate the tool, especially when multiple MCP servers are available.

Avoid assuming tools are installed

Don't assume packages are available:

````
**Good example: Explicit about dependencies**:
"Install required package: `pip install pypdf`

Then use it:
```python
from pypdf import PdfReader
reader = PdfReader("file.pdf")
```"
````

Technical notes

YAML frontmatter requirements

The SKILL.md frontmatter requires `name` and `description` fields with specific validation rules:

- `name`: Maximum 64 characters, lowercase letters/numbers/hyphens only, no XML tags, no reserved words
- `description`: Maximum 1024 characters, non-empty, no XML tags

See the Skills overview for complete structure details.

Token budgets

Keep SKILL.md body under 500 lines for optimal performance. If your content exceeds this, split it into separate files using the progressive disclosure patterns described earlier. For architectural details, see the Skills overview.

Checklist for effective Skills

Before sharing a Skill, verify:

Core quality

- ☐ Description is specific and includes key terms
- ☐ Description includes both what the Skill does and when to use it
- ☐ SKILL.md body is under 500 lines
- ☐ Additional details are in separate files (if needed)

- [ ] Examples are concrete, not abstract
- [ ] File references are one level deep
- [ ] Progressive disclosure used appropriately
- [ ] Workflows have clear steps

Code and scripts

- [ ] Scripts solve problems rather than punt to Claude
- [ ] Error handling is explicit and helpful
- [ ] No "voodoo constants" (all values justified)
- [ ] Required packages listed in instructions and verified as available
- [ ] Scripts have clear documentation
- [ ] No Windows-style paths (all forward slashes)
- [ ] Validation/verification steps for critical operations
- [ ] Feedback loops included for quality-critical tasks

Testing

- [ ] At least three evaluations created
- [ ] Tested with Haiku, Sonnet, and Opus
- [ ] Tested with real usage scenarios
- [ ] Team feedback incorporated (if applicable)

Next steps

Get started with Agent Skills

Create your first Skill

Use Skills in Claude Code ↗

Ask Docs

Claude Docs

Use Skills in the Agent SDK

Use Skills programmatically in TypeScript and Python

Use Skills with the API

Upload and use Skills programmatically

Claude Docs

Solutions

AI agents

Code modernization

Coding

Customer support

Education

Financial services

Government

Life sciences

Partners

Amazon Bedrock

Google Cloud's Vertex AI

Learn

Blog

Catalog

Courses

Use cases

Connectors

Customer stories

Engineering at Anthropic

Events

Powered by Claude

Service partners

Startups program

Company

Anthropic

Careers

Economic Futures

Ask Docs

Claude Docs

Responsible Scaling Policy

Security and compliance

Transparency

Help and security

Availability

Status

Support

Discord

Terms and policies

Privacy policy

Responsible disclosure policy

Terms of service: Commercial

Terms of service: Consumer

Usage policy

Ask Docs