Mastering Agent Skills

Workshop Handout

Introduction

What You'll Learn

By the end of this workshop, you will be able to:

- Understand what agent skills are and why they matter
- Create simple skills using SKILL.md files
- Build skills with references, scripts, and assets
- Use agentic approaches to create skills collaboratively
- Combine and orchestrate multiple skills
- Apply best practices for skill development

Who This Workshop Is For

This workshop is designed for **power users** of AI agents—people who want to customise and extend their AI tools to work better for their specific needs. You do not need to be a programmer. If you're comfortable with tools like Excel formulas or Zapier, you have enough technical fluency to create effective skills. Skills let you inject your domain knowledge into AI agents.

The Workshop Structure

Part	Topic
1	Introduction and First Skills
2	Assets, Scripts, and Agentic Creation
3	Advanced Techniques
4	Summary and Discussion

Part 1: Introduction and First Skills

The Problem Skills Solve

General-purpose AI agents are remarkably capable, but they lack domain expertise. Consider the analogy of choosing someone to do your taxes: who would you rather have? A 300 IQ mathematical genius, or an experienced tax professional? The genius might be brilliant, but the tax professional knows the specific rules, exceptions, and strategies that matter. The same applies to AI agents—they're intelligent, but they don't know your specific workflows, terminology, or requirements.

Skills bridge this gap. They let you teach agents your domain expertise, transforming a general-purpose assistant into a specialist who understands your work.

What Are Agent Skills?

Agent skills are **portable folders containing instructions, scripts, and resources** that extend an AI agent's capabilities for specific tasks.

At their simplest, a skill is a **folder** containing a `SKILL.md` file:

```
my-skill/
└── SKILL.md
```

The SKILL.md file contains:

- **Metadata** (name, description) in YAML frontmatter
- **Instructions** in markdown format

Here's a minimal example:

```yaml
---
name: greeting
description: Responds to greetings with a friendly message
---

# Greeting Skill

When the user greets you, respond warmly and ask how you can help today.
```

Why Skills Matter

Composable: Combine multiple skills for complex workflows. A research skill can feed into a report-writing skill.

Reusable: Create once, use repeatedly. Your invoice skill works every time without re-explaining.

Shareable: Distribute your domain expertise to colleagues or the community.

Extensible: Skills can use CLI tools, API calls, MCP servers, and other capabilities.

Portable: Skills are just text files—they work across different agents and platforms.

Forward compatible: No API versioning to worry about—skills are just text files. No model in the future will be worse at interpreting text instructions than models today.

What Is an Agent Platform?

An agent platform is the combination of three essential components:

1. A Powerful Agentic Model

Large language models with strong reasoning, instruction-following, and tool-use capabilities. These models can understand complex requests, generate code, and coordinate multi-step tasks.

Examples include Claude Opus 4.5, GPT-5.2, Gemini 3, and open models such as GLM-4.7 and Minimax M-2.1.

2. An Execution Environment

A place where the agent can do more than generate text. It can read and write files, execute code, and use local programs or cloud services.

This might be your local computer (when using Claude Code, GitHub Copilot, or OpenCode), a virtual machine in the cloud (when using Claude web app), or a container.

3. Context and Specialisation

Provided by **skills**. Skills inject domain expertise into the agent, transforming general capability into specific competence for your work.

Without all three components, you don't have a full agent platform. A model alone is just a chatbot. A model with execution but no skills is capable but generic. Skills close the loop—they provide the knowledge and procedures that make agents genuinely useful for your specific tasks.

The Agent Platform Ecosystem

Agent platforms supporting the Agent Skills standard:

Platform	Notes
Claude Code	CLI and desktop app
Claude Web/Cowork	Upload skills as zip files
VS Code + GitHub Copilot	IDE integration
Codex CLI	Command-line interface
OpenCode	CLI and desktop app
Antigravity	Gemini-powered agent
Additional agents	Many others support the standard

The paradigm: don't build agents, build skills. Skills are all you need. They are the universal extension mechanism. They can replace commands, hooks, MCP servers, and even entire applications. One skill encapsulating a workflow is often simpler and more powerful than a dedicated app.

The paradigm shift is significant. Rather than building custom AI agents for each domain, we build skills that any general-purpose agent can use. Think of it like the evolution of computing:

Era	Pattern
Early computing	Custom programs for each task
Modern computing	General OS + composable applications
AI agents	General agent + composable skills

Most agents load skills from `.claude/skills/` for compatibility, though some have their own preferred locations.

Categorising Skills

Skills can be understood along two dimensions:

By Scope (where they live):

- **Third-party**: Skills from external sources (companies, open-source)
- **Global/Personal**: Your skills for all projects (`~/.claude/skills/`)
- **Project-specific**: Skills for a particular repository (`.claude/skills/`)

By Purpose (what they do):

- **Information**: Load context when needed (documentation, references)
- **Behaviour**: Modify how the agent works (style guides, conventions)

- **Workflows**: Encode multi-step processes (release procedures, checklists)
- **Tools**: Perform specific tasks (image generation, file conversion)

	Third-party	Personal	Project-specific
Information	Industry guides	Research notes	Team procedures
Behaviour	Writing standards	Communication style	Report formats
Workflows	Approval processes	Personal routines	Publication steps
Tools	Image generation	Personal automations	Batch processing

Progressive Disclosure — The Core Design Principle

Skills use **progressive disclosure** to keep agent context efficient. Not everything loads at once:

Level	What Loads	When	Guidance
1	Name + description	Always (at startup)	~50-100 tokens
2	Full SKILL.md body	When skill activates	Keep under 500 lines / 5000 tokens
3	References, scripts	As needed during execution	Loaded as needed (still costs tokens)

This design means you can have hundreds of skills installed without overwhelming the agent's context window. Only the metadata is always present; full instructions load only when relevant. It works like a well-organised manual: table of contents first, then chapters, then appendix—only when needed.

Think of it like Neo in The Matrix—when he needs kung fu, they plug him in and he instantly knows it. Skills give agents instant access to specialised capabilities without having to load everything upfront. If the agent thinks the skill is relevant, it will read the full SKILL.md into context.

Setting Up Your Environment

Choosing Your Agent Platform

You need an agent that supports the Agent Skills standard—not just a chat interface. Recommended options:

Agent	Notes
Claude Code (CLI or desktop)	Best skills support, active development
VS Code with GitHub Copilot	Familiar IDE environment
Codex	CLI and VS Code extension
OpenCode	CLI and desktop app
Antigravity	Visual agent interface
Claude web/Cowork	Skills uploaded as zip files

Any agent listed at agentskills.io will work. Ensure your agent is connected to a capable model (Claude 4.5, GPT-5.2, Gemini 3, or similar).

Storage Locations

Location	Path	Who Can Use
Personal	`~/.claude/skills/`	You, across all projects
Project	`.claude/skills/`	Anyone working on this repository

The spec defines format; each agent defines discovery paths. GitHub Copilot prefers `.github/skills/` but reads `.claude/skills/` for backward compatibility. Check your agent's documentation for specific paths.

Creating the Skills Directory

```
# For personal skills
mkdir -p ~/.claude/skills

# For project skills
mkdir -p .claude/skills
```

Verifying Skill Discovery

After creating a skill, verify it appears in your agent's interface:

- In Claude Code: Check the slash menu (type `/`)

- In other agents: Invoke the skill from the chat (e.g., `/skill-name` or "use the skill-name skill")

Hands-On: Installing and Testing a Skill

Let's practice by installing a real skill from the Anthropic skills repository:

0. **Create your skills directory** (if needed):

```
mkdir -p ~/.claude/skills
```

1. **Download the skill:** Clone or download the `internal-comms` skill from: https://github.com/anthropics/skills/tree/main/skills/internal-comms
2. **Copy to your skills directory:**

```
# Copy the internal-comms folder to your personal skills
cp -r internal-comms ~/.claude/skills/
```

3. **Verify it appears:** In Claude Code, type `/` and look for `internal-comms` in the list
4. **Test with direct invocation:**

```
/internal-comms
```

   Then ask: "Write a 3P update for the skills workshop project"

5. **Test with natural language:**

```
Use the internal-comms skill to write a weekly status update about
preparing workshop materials
```

6. **Observe the behaviour:** The agent loads the skill, reads the appropriate template from the references folder, and follows the prescribed format for internal communications.

This exercise demonstrates the full skill lifecycle: installation, discovery, invocation, and execution.

For Non-Coders: Runtime Essentials

Agents can write and execute code on your behalf, but they need runtime environments to do so. You don't need to understand the code—just have these tools installed.

Note: Part 1 skills are instruction-only—no runtimes needed yet. Part 2 introduces scripts that require Python or JavaScript.

Python (recommended: install `uv`)

- Download from: https://docs.astral.sh/uv/
- `uv` handles packages automatically when scripts run

JavaScript/TypeScript

- Install Node.js from: https://nodejs.org/
- Or install Bun from: https://bun.sh/

Discovering Capabilities

If you're unsure what's possible, just ask your AI chat:

- "Can an agent do X?"
- "What tools exist for working with PDFs?"
- "Is there a way to generate images?"

The agent can tell you what's available and help you use it.

Creating Your First Skill

SKILL.md Structure

Every skill needs a `SKILL.md` file with YAML frontmatter:

```
---
name: my-skill
description: What this skill does and when to use it
---

# My Skill

Instructions for the agent go here in markdown format.
```

Required Fields

Field	Constraints	Purpose
`name`	1-64 chars, lowercase, letters/numbers/hyphens only, no starting/ending/consecutive hyphens	Unique identifier

Field	Constraints	Purpose
`description`	1-1024 chars	Triggers activation; should describe what AND when

Optional Fields

Field	Purpose
`license`	Licence reference
`compatibility`	Environment requirements
`metadata`	Arbitrary key-value pairs

Important Formatting Notes

- SKILL.md must start with `---` on line 1 (YAML frontmatter)
- Use spaces, not tabs, in YAML
- The skill directory name must match the `name` field

Common mistakes: wrong filename (`SKILLS.md` instead of `SKILL.md`), wrong case, bad YAML indentation, directory name mismatch

Writing Descriptions That Trigger Correctly

The description is crucial—it determines when the agent activates your skill. Good descriptions answer: "When would this activate?"

Good example:

```
Extracts text and tables from PDF files, fills PDF forms, and
merges multiple PDFs. Use when working with PDF documents or
when the user mentions PDFs, forms, or document extraction.
```

Poor example:

```
Helps with PDFs.
```

Tips for effective descriptions:

- Include action verbs (extracts, generates, converts)
- Mention specific file types or formats
- List concrete use cases

- Include keywords the user might say
- Be specific enough to avoid false activations

Adding References

References let you keep SKILL.md lean while providing detailed documentation the agent can access when needed.

Directory Structure with References

```
my-skill/
├── SKILL.md
└── references/
    ├── topic-one.md
    ├── topic-two.md
    └── topic-three.md
```

When to Use References

- Detailed documentation that isn't always needed
- Examples and edge cases
- Technical specifications
- Templates for output formats

The One-Level-Deep Rule

Keep your reference structure flat. Avoid chains where reference A links to reference B, which links to reference C. All references should be directly accessible from SKILL.md. Deep chains cause partial loading and unpredictable behaviour—the agent may not follow the full chain correctly.

Good:

```
SKILL.md → references/topic-one.md
SKILL.md → references/topic-two.md
```

Avoid:

```
SKILL.md → references/topic-one.md → references/details.md
```

Instructing the Agent to Use References

In your SKILL.md, tell the agent when to consult references:

```
# Recipe Skill

When the user asks about pasta, read `references/pasta.md`.
When the user asks about salads, read `references/salad.md`.
When the user asks about soups, read `references/soup.md`.
```

Part 1 Exercises

Hello World

The simplest possible skill. Responds to greetings with a friendly acknowledgement and the current date/time. This verifies your setup works correctly.

Directory structure:

```
hello-world/
└── SKILL.md
```

What to do:

1. Create the `hello-world/` directory in `~/.claude/skills/`
2. Create SKILL.md with minimal frontmatter (name, description)
3. Add simple instructions in the markdown body
4. Test by saying “hello” to the agent

Current Date

Responds with the current date formatted according to user preference (UK, US, ISO). Introduces conditional responses based on simple preferences.

Directory structure:

```
current-date/
└── SKILL.md
```

What to do:

1. Create the skill directory
2. Write SKILL.md with instructions for three date formats
3. Include format examples in the instructions
4. Test with “What’s the date?” and “What’s today’s date in UK format?”

Weekly Status

Create a skill that generates structured weekly status reports from a brief description of what you accomplished. The skill should output a consistent format with sections for completed work, in-progress items, blockers, and next week's priorities.

Directory structure:

```
weekly-status/
└── SKILL.md
```

What to do:

1. Create the skill directory
2. Write SKILL.md with the report template structure
3. Include a sample formatted output showing the sections
4. Test by describing your own week's activities

Recipe Collection

Create a personal recipe collection stored as a skill. Each reference file contains a complete recipe. The main SKILL.md instructs the agent to retrieve one or more recipes on request, suggesting options based on ingredients, cuisine, or meal type.

Directory structure:

```
recipe-collection/
├── SKILL.md
└── references/
    ├── spaghetti-carbonara.md
    ├── chicken-stir-fry.md
    └── chocolate-brownies.md
```

What to do:

1. Create skill directory with references/
2. Write SKILL.md explaining the collection and how to request recipes
3. Add 3-5 favourite recipes as individual reference files
4. Test by asking for recipes by name, ingredient, or cuisine

Persona Writer

Create a skill that writes content in different personas (CEO, engineer, marketer). Each persona has its own reference file with voice characteristics. The SKILL.md explains how to select a persona and instructs the agent to load the appropriate reference.

Directory structure:

```
persona-writer/
├── SKILL.md
└── references/
    ├── ceo-voice.md
    ├── engineer-voice.md
    └── marketer-voice.md
```

What to do:

1. Create skill directory with references/
2. Write SKILL.md explaining persona selection
3. Create detailed persona files
4. Test by requesting content from different perspectives

Additional Ideas

If you finish early or want more practice, try these:

- **British Spelling**: Enforces British English spelling conventions when reviewing or writing text
- **Email Subject**: Crafts effective email subject lines following best practices
- **File Naming**: Suggests file names following a consistent convention (lowercase, hyphens, date prefixes)
- **Meeting Agenda**: Creates structured meeting agendas with categorised items and time allocations
- **Tone Guide**: Enforces a specific writing tone using reference files with detailed guidelines

---

Part 2: Assets, Scripts, and Agentic Creation

The Full Skill Directory Structure

Skills can contain more than just SKILL.md and references:

```
skill-name/
├── SKILL.md          (required - instructions)
├── references/       (documentation loaded as needed)
├── scripts/          (executable code)
└── assets/           (templates, images — NOT loaded into context)
```

Understanding the Components

Component	Purpose	Loaded Into Context?
SKILL.md	Core instructions	Yes, when activated
references/	Detailed documentation	Yes, when requested
scripts/	Executable code	No — only results returned
assets/	Templates, images, data files	No — used for output

The distinction between references and assets is important:

- **References** are read by the agent for information
- **Assets** are used by the agent to produce output (never read into context)

Scripts: Deterministic Execution

Scripts provide precision and repeatability for tasks where you need consistent results. Code is self-documenting, modifiable, and can live in the file system until needed.

Why Use Scripts?

- **Deterministic**: Same input always produces same output
- **Token efficient**: Code doesn't load into context; only results return
- **Testable**: Scripts can be validated independently
- **Reusable**: Write once, execute many times

When to Use Scripts vs. Instructions

Use Scripts When	Use Instructions When
Exact calculations required	Flexibility is valuable
Consistent formatting needed	Creative interpretation wanted
External API calls	Natural language processing
File format conversions	Subjective judgements

Example: A Simple Script Skill

```
youtube-transcript/
├── SKILL.md
└── scripts/
    └── fetch_transcript.py
```

SKILL.md:

```
---
name: youtube-transcript
description: Extracts transcripts from YouTube videos. Use when the user
  wants a transcript, subtitles, or captions from a YouTube URL.
---

# YouTube Transcript

To extract a transcript, run the script:

\`\`\`bash
python scripts/fetch_transcript.py VIDEO_URL
\`\`\`

The script accepts YouTube URLs or video IDs.
```

Assets: Files Used "As Is"

Assets are files the skill can use directly—they're never loaded into context. The agent doesn't read assets for information; it uses them to produce output.

Examples of assets:

- Template documents (Word, text, HTML)
- Images and logos
- Fonts
- Data files (CSV, JSON)
- Boilerplate documents

Example directory structure:

```
report-generator/
├── SKILL.md
├── assets/
```

```
│   ├── report-template.docx
│   └── company-logo.png
└── scripts/
    └── generate_report.py
```

The distinction from references: references are read by the agent for information. Assets are used by scripts or copied directly to output.

Installing the Skill-Creator Skill

Before creating skills agentically, install the **skill-creator** skill from Anthropic's repository. This skill teaches the agent best practices for skill design.

What it provides:

- Structured creation process: understand examples → plan contents → implement → package
- Core principles: conciseness, appropriate degrees of freedom, progressive disclosure
- Helper scripts for initialisation (`init_skill.py`) and packaging (`package_skill.py`)
- Guidance on when to use scripts, references, and assets

Installation:

Download from: https://github.com/anthropics/skills/tree/main/skills/skill-creator

Place in your global skills directory:

```
~/.claude/skills/skill-creator/
```

With this skill installed, when you ask the agent to create a skill, it will follow a structured process and apply best practices automatically.

Creating Skills Agentically

You don't have to write skills by hand. There are two powerful approaches to having the agent create skills for you.

Approach 1: Direct Instruction

Tell the agent what you want and provide source material:

Describing what you want:

```
Create a skill that generates weekly status reports for the ACME team.
It should ask me about accomplishments, blockers, and next week's priorities.
Format the output as an email with a consistent structure every time.
```

Providing a template:

```
Here's a Word template for invoices [attach file].
Create a skill that generates invoices using this template.
```

Providing format requirements:

```
Create a skill that formats my commit messages following
the Conventional Commits specification.
```

The key insight: **you don't need to write the skill yourself**. Describe what you want, provide the format or structure, and the agent creates the SKILL.md with instructions to gather information, ask clarifying questions, and produce consistent output.

Approach 2: Show Don't Tell

Work through a task collaboratively, then ask the agent to create a skill from the process:

The Workflow:

1. Work through a task together with the agent
2. Iterate until you're satisfied with the result
3. Ask: "Create a skill from what we just did"
4. Test the skill on a similar task
5. Refine based on real usage

If it works once, capture it. If it works twice, skill it.

Why this works:

- No upfront specification writing
- The agent captures what actually works
- Results in practical, tested skills
- Skills emerge from work, not from abstract design

You bring the knowledge. The agent handles the rest.

Alternative approach:

```
Review all my conversations with you. Build me a skill
for tasks that I do repeatedly.
```

The agent can identify patterns in your work and suggest skills.

A Real Example: Energy Report (Show Don't Tell)

This skill was created through collaborative iteration, not upfront specification:

1. **Work together**: "Help me create a report from this energy data CSV"
2. **Iterate**: "Actually, I want a pie chart instead of a bar chart"
3. **Refine**: "Can you add country flags and emissions per capita?"
4. **Capture**: "Create a skill that does what we just did"

The resulting skill includes:

- `SKILL.md` with workflow instructions
- `scripts/generate_report.py` for data analysis and charts
- `scripts/convert_to_pdf.py` for PDF output

The skill now works consistently on any year's data, producing markdown reports with embedded charts. It emerged from work, not from abstract design.

Part 2 Exercises

Password Generator

Generates secure passwords with customisable length, character sets, and requirements. Demonstrates a skill with a script for deterministic execution.

Directory structure:

```
password-generator/
├── SKILL.md
└── scripts/
    └── generate_password.py
```

What to do:

1. Create skill directory with scripts/
2. Write a Python script with argparse for options (length, character set, requirements)
3. Write SKILL.md explaining the options
4. Test with various password requirements

Simple Template

Fills a simple text template with provided values. Introduction to assets folder for templates.

Directory structure:

```
simple-template/
├── SKILL.md
├── scripts/
│   └── fill_template.py
└── assets/
    └── template.txt
```

What to do:

1. Create skill directory with scripts/ and assets/
2. Create a simple text template with `{placeholders}`
3. Write a script to read the template and fill values
4. Write SKILL.md explaining the workflow
5. Test by providing placeholder values

Readability Scorer

Create a skill that analyses text and calculates readability metrics: Flesch-Kincaid grade level, reading ease score, average sentence length, and estimated reading time. The skill uses a script to perform the analysis and return structured results.

Directory structure:

```
readability-scorer/
├── SKILL.md
└── scripts/
    └── calculate_readability.py
```

What to do:

1. Create skill directory with scripts/
2. Write a script using text analysis algorithms
3. Write SKILL.md with usage examples
4. Test with different text samples

QR Code Generator

Create a skill that generates QR codes from text or URLs. This skill demonstrates using a script with external dependencies. The script should accept text or a URL and produce a QR code image.

Directory structure:

```
qr-code-generator/
├── SKILL.md
└── scripts/
    └── generate_qr
```

What to do:

1. Create skill directory with scripts/
2. Write a script that generates QR codes (let the agent choose the language and libraries)
3. Write SKILL.md explaining usage and output options
4. Test by generating QR codes for different inputs

Spreadsheet Summariser

Create a skill that reads a spreadsheet (CSV or Excel) and provides a plain-language summary: row count, column names, what kind of data each column contains, and example values. This is useful for quickly understanding unfamiliar data files.

Directory structure:

```
spreadsheet-summariser/
├── SKILL.md
└── scripts/
    └── summarise_spreadsheet.py
```

Test data: For testing, download OWID Energy Data — a comprehensive CSV dataset with global energy statistics.

What to do:

1. Create skill directory with scripts/
2. Write a script with openpyxl for Excel support
3. Write SKILL.md for file input handling
4. Test with various spreadsheet files

Additional Ideas

If you finish early or want more practice, try these:

- **Random Code Generator**: Generates random codes for vouchers, tickets, or reference numbers
- **Unit Converter**: Converts between common units (temperature, weight, distance, volume)
- **Business Card**: Generates a business card layout from provided details using an HTML template asset
- **Invoice Simple**: Generates invoices with agentic templating (asset template + script)
- **Show Don't Tell Practice**: Work through a task together, then ask the agent to create a skill from it

---

Part 3: Advanced Techniques

Combining and Orchestrating Skills

Skills become more powerful when combined. You can build complex workflows from simple, focused components.

Nesting Skills

Call one skill from within another. There are two invocation approaches:

Slash command format (Claude recognises this directly, but not universally supported):

```
# Research Report Skill

1. Use /web-research to gather information on the topic
2. Use /summarise to create key points
3. Use /report-generator to format the final document
```

Natural language format (works broadly across agents):

```
# Research Report Skill

1. Use the web-research skill to gather information on the topic
2. Use the summarise skill to create key points
3. Use the report-generator skill to format the final document
```

Some agents may not recognise the `/skill-name` slash command syntax—in these cases, natural language invocation is the more portable option.

Dynamic Loading

Skills enable dynamic loading of capabilities. Consider a game system where fighting gets fight rules only when it needs to fight—you save context and can build many capabilities that load dynamically. Only the relevant skills load when needed, keeping context efficient.

Gathering User Input

Skills can instruct the agent to collect information before proceeding. This is useful for gathering choices, preferences, and configuration options from users.

The `AskUserQuestion` Tool

In Claude Code and OpenCode, the agent has access to the `AskUserQuestion` tool, which displays a UI with selectable options for the user. This provides a better experience than typing answers in chat—users can click to select from predefined choices.

In other agents where `AskUserQuestion` is not yet available, the agent can still ask questions in the chat and the user answers in text. The skill instructions work the same way—only the presentation differs.

The Pattern

In your SKILL.md, specify what information to gather:

```
# Invoice Generator

Before generating an invoice, collect the following from the user:

- Client name and address
- Invoice items with quantities and prices
- Payment terms
- Due date

If any required information is missing, ask the user before proceeding.
```

Multi-Step Interactions

For complex workflows, gather information in stages:

```
# Project Setup

Phase 1 - Basic Information:

- Ask for project name and type
- Ask for primary programming language

Phase 2 - Configuration:

- Based on project type, ask about specific frameworks
- Ask about testing preferences

Phase 3 - Review:

- Show summary of choices
- Ask if the user wants to change anything
```

Conditional Logic

Handle different paths based on user responses:

```
If the user wants to export:

- Ask which format (PDF, Word, or Markdown)
- For PDF, ask about page size and orientation
```

Delegating Complex Tasks

For large tasks, instruct the skill to break work into sub-tasks:

```
# Large Document Analysis

For documents over 50 pages:

1. First, scan the table of contents and summarise the structure
2. Then, analyse each major section separately
3. Finally, synthesise findings into a comprehensive report

Each analysis phase should focus deeply on its section before moving on.
```

This approach works across all agent platforms—the agent handles the delegation appropriately.

Security Considerations

Skills can be powerful, which means security matters. The good news: **skills are plain text files—fully auditable before installation.**

Installing Skills Safely

- **Only install from trusted sources**
- **Read the contents** before installing third-party skills
- **Check scripts** for unexpected network calls or file operations
- **Review bundled resources** (images, data files)

Watch for: unexpected network calls (e.g., `curl` to unknown URLs), file operations outside expected paths, data exfiltration patterns.

Malicious skills may introduce vulnerabilities or direct agents to exfiltrate data and take unintended actions.

Script Execution Practices

Practice	Description
Sandboxing	Run scripts in isolated environments when possible
Allowlisting	Only execute scripts from trusted skills
Logging	Record script executions for auditing

Cross-Platform vs Agent-Specific Features

Most skill features work across all platforms. Some are specific to Claude Code:

Feature	Cross-Platform	Claude Code Only
`name`, `description`	✓	
`scripts/`, `references/`, `assets/`	✓	
Progressive disclosure	✓	
Skill nesting	✓	
User input gathering	✓*	
`allowed-tools`	experimental	robust support
`context: fork` (subagent)		✓
`user-invocable` (show in slash menu)		✓

* Via chat in all agents; `AskUserQuestion` tool UI in Claude Code, OpenCode (Copilot coming soon)

Designing for Portability

When building skills, prefer cross-platform features over agent-specific options. Almost all the advantages of skills—reusability, composability, shareability—work perfectly with the standard feature set.

Why portability matters:

- **Wider reach**: A portable skill works across Claude Code, GitHub Copilot, OpenCode, Codex, Antigravity, and future agents
- **Future-proofing**: Today's agent-specific features often become tomorrow's cross-platform standards
- **Simpler maintenance**: One skill definition serves all platforms, rather than maintaining variants
- **Community sharing**: Portable skills can be shared and adopted more broadly

Practical guidance:

- Start with only cross-platform features
- Add agent-specific options (like `allowed-tools` or `context: fork`) only when they provide clear, necessary value
- When you need advanced behaviour, consider whether natural language instructions can achieve the same result portably

The ecosystem is converging. Features that start as Claude-specific—like the `AskUserQuestion` tool UI—are being adopted by other agents. Build for portability now, and your skills will benefit from platform improvements automatically.

Claude-Specific: Allowed Tools

Pre-authorise specific tools to skip permission prompts:

```yaml
---
name: code-searcher
description: Searches codebase for patterns
allowed-tools: Read, Grep, Glob
---
```

Claude-Specific: Forked Context

Run a skill in a subagent with its own separate context:

```yaml
---
name: deep-research
description: Thorough research on a topic
context: fork
---
```

This creates a subagent for running the skill that has its own conversation history.

- **When to use:** Complex multi-step operations that would clutter the main conversation
- **When not to use:** When the skill needs shared context with the caller

For cross-platform compatibility, use natural language instructions instead:

```
For the research phase, focus deeply on gathering information
before returning your findings.
```

Part 3 Exercises

Travel Packing List

Creates personalised packing lists based on destination, duration, and trip purpose. Asks essential questions before generating a contextual list with climate-appropriate clothing, toiletries, and travel essentials. Demonstrates user input gathering.

Directory structure:

```
travel-packing-list/
├── SKILL.md
└── references/
    └── packing-categories.md
```

What to do:

1. Create skill directory with references/
2. Write SKILL.md instructing: “Ask the user for destination, duration (days), and purpose (business/leisure/mixed)”
3. Create reference file with packing categories and climate considerations
4. Include example interaction flow showing question → answer → list
5. Test with different trip types

Travel Brief (Skill Nesting)

Creates comprehensive travel preparation documents by calling another skill. This demonstrates skill nesting—one skill calling another using the `/skill-name` syntax. You need to create two skills:

Skill 1: travel-packing-list/ (the skill being called)

```
travel-packing-list/
├── SKILL.md
└── references/
    └── packing-categories.md
```

This is the same skill from the previous demo. It asks for destination, duration, and trip purpose, then generates contextual packing suggestions.

Skill 2: travel-brief/ (the skill that calls it)

```
travel-brief/
├── SKILL.md
└── references/
    └── brief-sections.md
```

What to do:

1. Ensure travel-packing-list is installed from the previous demo
2. Create the travel-brief skill directory with references/
3. Write SKILL.md with explicit instruction: "First, invoke `/travel-packing-list` to generate the packing section"
4. Add instructions for weather research, local tips, and logistics
5. Create brief-sections.md describing how to combine skill output with other sections
6. Test by invoking `/travel-brief` and observe it calling `/travel-packing-list`

Expense Report

Create a skill that generates expense reports with conditional follow-up questions based on expense type. Travel expenses prompt for dates and locations; meals prompt for attendees and business purpose; supplies prompt for itemisation. Different paths lead to appropriate report sections.

Directory structure:

```
expense-report/
├── SKILL.md
└── references/
    └── expense-categories.md
```

What to do:

1. Create skill directory with references/
2. Write SKILL.md with conditional logic: "First ask expense type, then ask relevant follow-ups"
3. Create reference file with expense categories and their required fields
4. Document the decision tree in SKILL.md
5. Test each expense type path

Content Pipeline (Multi-Skill Chaining)

Create a content publication pipeline that chains multiple skills together using explicit `/skill-name` calls. This exercise requires creating three skills that work together:

Skill 1: brand-voice/

```
brand-voice/
├── SKILL.md
└── references/
    └── voice-guidelines.md
```

Applies consistent organisational tone to content. Write SKILL.md that transforms text to match voice guidelines. Create voice-guidelines.md with tone, vocabulary, and style rules.

Skill 2: content-refiner/

```
content-refiner/
├── SKILL.md
└── references/
    └── refinement-checklist.md
```

Clarifies, condenses, and polishes text. Write SKILL.md that improves clarity and removes fluff. Create refinement-checklist.md with clarity and conciseness rules.

Skill 3: content-pipeline/ (the orchestrator)

```
content-pipeline/
├── SKILL.md
└── references/
    └── platform-formats.md
```

What to do:

1. Create all three skill directories
2. Write the brand-voice and content-refiner skills first (these are callees)
3. Write content-pipeline SKILL.md with explicit skill chain:
   - "Step 1: Call `/brand-voice` to apply organisation tone to the raw content"
   - "Step 2: Call `/content-refiner` to clarify and polish the message"
   - "Step 3: Format the result for the target platform"
4. Create platform-formats.md with formatting for email, social, newsletter
5. Install all three skills
6. Test the full chain from raw input to polished, formatted output

Image Resizer

Create a skill that resizes and converts images by wrapping a command-line tool. The skill asks for target dimensions, format, and quality settings, then creates web-optimised, social media, or print-ready versions.

Directory structure:

```
image-resizer/
├── SKILL.md
├── scripts/
│   └── resize_image.sh
└── references/
    └── preset-sizes.md
```

What to do:

1. Create skill directory with scripts/ and references/
2. Write a shell script that wraps a command-line image tool with common operations
3. Script should accept: input file, output format, dimensions, quality
4. Write SKILL.md with clear usage instructions
5. Create preset-sizes.md with common sizes (social: 1200x630, thumbnail: 150x150, print: 300dpi)

6. Include instruction: “First ask what the image will be used for, then suggest appropriate settings”
7. Test with sample images

Additional Ideas

If you finish early or want more practice, try these:

- **Daily Standup**: Formats daily standup updates, remembering team preferences within the session
- **File Organiser**: Proposes folder organisation with preview, asks user which option they prefer
- **Weather Briefing**: Creates weather briefings by wrapping a free weather API (wttr.in)
- **Security Audit**: Audits other skills for dangerous patterns and security concerns
- **Document Converter**: Converts documents between formats by wrapping a command-line conversion tool

---

Part 4: Summary and Discussion

Key Concepts Recap

Agent Skills are portable folders of instructions, scripts, and resources that extend AI agent capabilities.

Progressive Disclosure means loading only what’s needed, when needed—keeping context efficient while enabling complex functionality.

Open Standard (agentskills.io) ensures skills work across multiple platforms—write once, use anywhere.

Composability lets you combine simple skills into complex workflows, building sophisticated capabilities from focused components.

Best Practices Checklist

Structure and Size

- ☐ Keep SKILL.md under 500 lines (<5000 tokens)
- ☐ SKILL.md starts with `---` on line 1
- ☐ Use spaces, not tabs, in YAML
- ☐ Directory name matches the `name` field

Descriptions

- ☐ Descriptions answer “when would this activate?”
- ☐ Include action verbs and specific use cases
- ☐ Stay under 1024 characters
- ☐ Keep descriptions distinct to avoid collisions between skills
- ☐ Test with various phrasings

Instructions

- ☐ Include examples of expected input/output in SKILL.md
- ☐ Write instructions that tell the agent when to read each reference

References

- ☐ Keep references one level deep
- ☐ All references link directly from SKILL.md
- ☐ Keep reference files focused and small
- ☐ Use for detailed documentation, not core instructions

Scripts

- ☐ Provide clear error messages with solutions

Testing

- ☐ Test skills with various trigger phrasings
- ☐ Verify references load correctly
- ☐ Check scripts execute without errors
- ☐ Iterate based on real usage

The Skill Development Mindset

Start simple. Your first version doesn’t need every feature. Add complexity as you discover what’s needed.

Show don’t tell. Work through tasks collaboratively, then ask the agent to create skills from successful approaches.

Think from the agent’s perspective. How will the agent know when to use this skill? What information will it need?

Monitor and iterate. Watch how skills perform in real scenarios. Refine descriptions and instructions based on what you observe.

Skills and Continuous Learning

Skills represent a solution to AI continuous learning. When you create a skill, you're encoding knowledge that persists across sessions:

- An agent on day 30 is better than on day 1—because of accumulated skills
- Anything an agent writes down can be used efficiently by a future version of itself
- Skills make learning tangible: procedural knowledge for specific tasks

Future Directions

The skills ecosystem is evolving rapidly:

Skills complement MCP: MCP provides access to external services; skills teach how to use them effectively. They work together.

Skill marketplaces: Discover and share skills like app stores—browse, install, and contribute.

Testing: Verify skills work correctly by testing with 5 prompts that should trigger and 5 that should not.

Versioning: Track changes over time; evolve skills based on observed failures and user feedback.

Enterprise adoption: Organisations using skills to encode best practices across teams —building a collective and evolving knowledge base of capabilities curated by people and agents.

Skills replacing applications: Portable, composable skills that do what traditional apps do, but more flexibly.

Discussion Questions

- What tasks in your work would benefit most from skills?
- How might skills change how your team shares knowledge?
- What skills would you want to find in a marketplace?
- How do you see skills evolving in the next year?

Resources

Specification and Documentation

- **Agent Skills Specification**: https://agentskills.io/specification
- **Claude Code Skills Documentation**: https://code.claude.com/docs/en/skills

Skill Collections

- **Anthropic Skills Repository**: https://github.com/anthropics/skills
- **OpenAI Skills Repository**: https://github.com/openai/skills
- **Hugging Face Skills**: https://github.com/huggingface/skills
- **Vercel Labs Agent Skills**: https://github.com/vercel-labs/agent-skills
- **Notion Skills for Claude**: https://notiondevs.notion.site/notion-skills-for-claude
- **Eleanor's Example Skills**: https://everything.intellectronica.net/p/agent-skills-changed-how-i-work-with

Runtime Environments

- **uv (Python)**: https://docs.astral.sh/uv/
- **Node.js**: https://nodejs.org/
- **Bun**: https://bun.sh/

Agent Platforms

- **Claude Code**: https://code.claude.com/
- **GitHub Copilot**: https://github.com/features/copilot
- **Codex CLI**: https://github.com/openai/codex
- **OpenCode**: https://opencode.ai/
- **Antigravity**: https://antigravity.google/
- **Claude Code for Non-Coders**: https://everything.intellectronica.net/p/claude-code-for-non-coders
- **What Tool Should I Use?**: https://elite-ai-assisted-coding.dev/p/a-straightforward-answer-to-what-tool-should-i-use

Platform Skills Documentation

- **GitHub Copilot**: https://docs.github.com/en/copilot/concepts/agents/about-agent-skills
- **OpenCode**: https://opencode.ai/docs/skills/
- **Gemini CLI**: https://geminicli.com/docs/cli/skills/
- **Codex**: https://developers.openai.com/codex/skills/
- **Cursor**: https://cursor.com/docs/context/skills
- **Antigravity**: https://antigravity.google/docs/skills
- **Goose**: https://block.github.io/goose/docs/guides/context-engineering/using-skills/
- **Amp**: https://ampcode.com/news/agent-skills