AI OPERATOR

aioperator.com

AI OPERATOR REFERENCE

Anthropic Academy Playbook

Key concepts and takeaways from all 18 courses — in one place.

BY

AI Operator

PUBLISHED

May 2026

What this is

A tight reference guide covering all 18 Anthropic Academy courses. One section per course. Key concepts, key takeaways, and code patterns where relevant. Skip to any course you need — no reading required from cover to cover.

Course 1 — Claude 101

Claude's interfaces, core capabilities, and how to get better results.

Key concepts

- Claude is built on Constitutional AI — helpful, harmless, honest — with a 200K+ token context window
- **Chat** = turn-by-turn. **Cowork** = agentic multi-step delegation. **Code** = full dev environment
- **Projects** = persistent workspaces with a knowledge base, custom instructions, and team sharing
- **Artifacts** = standalone rendered outputs (HTML, React, SVG, docs) — publishable and remixable
- **Skills** = on-demand instruction packages for repeatable workflows
- **Connectors** link Claude to external tools (Google Drive, Slack, Notion, Jira) via MCP
- **Research mode** = multi-source investigation (5–45 min) with citations and extended thinking

Key takeaways

- **Structure every prompt:** role + context → action → format/tone/examples
- **Use Projects** for recurring work streams. **Use Skills** to encode repeatable processes. They compound.
- The 4D Framework governs all AI use: **Delegation → Description → Discernment → Diligence**
- Verify high-stakes outputs independently. Run lightweight evals (5–10 examples) to calibrate Claude.

Course 2 — Claude Code 101

Claude Code as an AI agent — install, configure, and use it effectively.

Key concepts

- Claude Code runs an **agentic loop**: prompt → gather context → act → verify → repeat
- **Context window** = working memory; use `/compact` (summarize and continue) or `/clear` (fresh start)
- **Permission modes**: approval-per-action (default), auto-accept, or **Plan Mode** (read-only before changes)
- **CLAUDE.md** = persistent project memory; auto-loaded every session; shared via git
- **Subagents** = isolated parallel contexts for off-loading discrete tasks
- **Hooks** = deterministic shell commands at lifecycle events (PreToolUse, PostToolUse, Stop, etc.)

Key takeaways

- **Explore → Plan → Code → Commit** — never skip to code
- **Put CLAUDE.md in version control.** Only add entries when you catch yourself correcting Claude repeatedly.
- Hooks are the only truly deterministic behavior in Claude Code. If something must always happen, it belongs in a hook.
- Use subagents for code review — fresh context, no bias from the coding session.

```
# Install
curl -fsSL https://claude.ai/install.sh | bash

# Context shortcuts
/compact    # summarize + continue
/clear      # full reset
```

```
// PostToolUse hook — auto-format after every edit
{
  "hooks": {
    "PostToolUse": [
      {
        "matcher": "Edit|MultiEdit|Write",
        "hooks": [{ "type": "command", "command": "/abs/path/format.sh" }]
      }
    ]
  }
}
```

Course 3 — Introduction to Claude Cowork

Delegate complete pieces of work — multi-step, multi-tool, real artifact out.

Key concepts

- **Cowork vs Chat:** Chat = thinking partner. Cowork = delegate a complete task.
- **Four building blocks:** global instructions → projects → skills → plugins
- **Global instructions** = standing brief applied to every session
- **Skills** = reusable playbooks (`SKILL.md` + assets); auto-triggered when a task matches
- **Plugins** = bundled skills for a job function, distributed via an org marketplace
- **Claude in Chrome** = bridge for tools without connectors (internal dashboards, portals)

Key takeaways

- A good Cowork prompt names: the deliverable, the inputs, and the nuances only you know.
- Use "Ask before acting" for anything that sends or shares. Test scheduled tasks on copies first.
- The four building blocks compound: global instructions calibrate every session, projects add memory, skills encode process, plugins package expertise for the team.

Course 4 — Claude Code in Action

Hooks, the SDK, custom commands, MCP servers, and GitHub integration.

Key concepts

- **Custom commands:** markdown files in `.claude/commands/`, invoked with `/command-name`
- **GitHub integration:** works with `gh` CLI; creates and links PRs; CI/CD via GitHub Actions
- **Hooks deep-dive:** stdin payload varies by hook type; use `jq . > log.json` to inspect before building
- **Claude Code SDK:** run Claude Code programmatically from TypeScript or Python
- All hook event types: PreToolUse, PostToolUse, Stop, SubagentStop, PreCompact, UserPromptSubmit, Notification, SessionStart, SessionEnd

Key takeaways

- **Plan Mode is read-only** — use it for complex multi-file changes to course-correct before anything is written.
- Use **absolute paths** in hooks, not relative ones — prevents path interception vulnerabilities.
- Custom commands collapse multi-step git workflows into a single invocation.
- The SDK unlocks Claude Code as a building block for CI pipelines, scheduled jobs, and custom tooling.

Course 5 — AI Fluency Framework & Foundations

The academic and practical framework for effective, ethical AI collaboration.

Key concepts

- **Three engagement modes:** Automation (AI executes), Augmentation (human-AI collaboration), Agency (configure AI to work independently)
- **Delegation** — decide what goes to AI vs. stays with you (Problem Awareness + Platform Awareness + Task Delegation)
- **Description** — Product (what to make) + Process (how to approach it) + Performance (how to behave)
- **Discernment** — evaluate the output, the reasoning, and the behavior
- **Diligence** — choose systems thoughtfully, disclose AI's role, own the final output
- The **Description–Discernment loop** drives quality: describe → evaluate → refine → integrate → repeat

Key takeaways

- AI Fluency is a durable skill set. The 4Ds stay relevant as models change.
- **Don't delegate blindly.** Problem Awareness (knowing what you need) comes before Platform Awareness (knowing what AI can do).
- The secret weapon in Description: ask AI to help improve your own prompt.
- Discernment is strongest when you have domain expertise. If you can't evaluate the output, you need an independent check.
- You are responsible for AI-assisted work you ship — regardless of how much AI contributed.

Course 6 — Building with the Claude API

Prompt caching, the Files API, code execution, MCP, and agentic workflows.

Key concepts

- **Prompt caching:** mark reused prefixes with `cache_control`; cache hits cost ~10% of base input; 5-min rolling TTL; min 1024 tokens
- **Files API:** upload once (get a `file_id`), reference in future messages
- **Code Execution:** sandboxed Python in Docker; no network access
- **Workflow patterns:** Parallelization, Chaining, Routing, Evaluator-Optimizer
- **Agents vs. workflows:** workflows = predefined steps; agents = Claude decides the steps. Default to workflows.

Key takeaways

- **Cache long system prompts aggressively.** The ~90% cost reduction on cache hits is significant at scale.
- In chaining workflows, break up over-constrained single prompts: generate in step 1, apply constraints in step 2.
- Default to workflows. Use agents only when the task is genuinely open-ended and you can't enumerate the steps.
- Give every agent tool a way to observe its own results. Agents that can't verify actions can't recover from mistakes.

```
# Prompt caching
response = client.messages.create(
    model="claude-sonnet-4-6",
    system=[{
        "type": "text",
        "text": long_system_prompt,
        "cache_control": {"type": "ephemeral"}
    }],
    messages=[{"role": "user", "content": user_input}]
)
```

Course 7 — Introduction to Model Context Protocol

Building MCP servers and clients in Python using FastMCP.

Key concepts

- **Three primitives:** Tools (model-controlled), Resources (app-controlled), Prompts (user-controlled)
- **FastMCP + decorators:** Python type hints and `Field` descriptions auto-generate JSON schemas
- **Server Inspector:** `mcp dev mcp_server.py` opens a browser-based debug UI
- **Resources vs tools:** resources inject data directly into prompts; tools let Claude fetch data at runtime

Key takeaways

- Use **tools** when Claude needs to act. Use **resources** to load data into context. Use **prompts** for repeatable, tested workflows.
- Errors in tools surface via Python exceptions — no special wrapper needed.
- The 12-step flow (user query → list tools → Claude → tool call → result → response) is the core loop to internalize.

```
from mcp.server.fastmcp import FastMCP
mcp = FastMCP("DocumentMCP")

@mcp.tool()
def read_document(doc_id: str):
    if doc_id not in docs:
        raise ValueError(f"Doc {doc_id} not found")
    return docs[doc_id]

@mcp.resource("docs://documents/{doc_id}")
def fetch_doc(doc_id: str) -> str:
    return docs[doc_id]
```

Course 8 — AI Fluency for Educators

Applying the 4D Framework to course design and material creation.

Key concepts

- **Augmentation over automation:** AI enhances teaching expertise — it doesn't replace it
- **Teaching context document:** a reusable doc capturing your subject, student profile, constraints, and philosophy
- **Diligence in education:** verify accuracy, check for bias, protect student data, disclose AI's role

Key takeaways

- **Build a Teaching Context Document once** and reuse it in every AI session.
- The goal is better teaching, not just faster planning — AI-assisted materials should be more coherent, not just quicker.
- AI is most useful for: content sequencing, spotting gaps in learning objectives, anticipating misconceptions, and generating varied question types.
- **Document what you rejected and why** — this audit trail maintains pedagogical integrity.

Course 9 — AI Fluency for Students

How to learn with AI rather than outsource to it.

Key concepts

- **Learning partner, not answer machine:** configure AI explicitly as a tutor; have it ask questions, not give answers
- **Learning context document:** your courses, learning style, goals, AI use boundaries, and school policies
- **Personal AI collaboration policy:** when you will/won't use AI, transparency commitments, ethical red lines

Key takeaways

- **You must be able to explain everything you submit** — even if AI helped create it.
- Study protocols that work: "Guide me with hints, don't solve it" / "Quiz me and explain why wrong answers are wrong."
- Maintain AI-free practice sessions for core skills. Over-reliance erodes abilities you need in exams and interviews.
- For CV/interview prep: AI critiques and gathers information, but every claim must be backed by a real example you can discuss.

Course 10 — Model Context Protocol: Advanced Topics

Sampling, Notifications, Roots, JSON messages, and both transport layers.

Key concepts

- **Sampling:** server asks the client to call Claude — server needs no API key; ideal for public servers
- **Log and progress notifications:** `context.info()` for logs, `context.report_progress(current, total)` for progress bars
- **Roots:** permission system granting MCP servers access to specific directories; you enforce it yourself
- **stdio transport:** client launches server as subprocess; full bidirectional; local only
- **StreamableHTTP transport:** remote servers over HTTP; uses SSE; `stateless_http` and `json_response` flags can silently break sampling and progress

Key takeaways

- Only set `stateless_http=True` for horizontally scaled deployments — it disables sampling, progress reports, and resource subscriptions.
- `json_response=True` strips all streaming — use for plain JSON integrations only.
- Test with the same transport locally before deploying. The behavioral gap between stateful and stateless mode is large.

```
# Sampling — server side
@mcp.tool()
async def summarize(text: str, ctx: Context):
    result = await ctx.session.create_message(
        messages=[SamplingMessage(role="user",
            content=TextContent(type="text", text=f"Summarize: {text}"))],
        max_tokens=4000,
    )
    return result.content.text
```

Course 11 — Claude with Amazon Bedrock

Accessing Claude via AWS Bedrock using the boto3 SDK.

Key concepts

- **boto3, not the Anthropic SDK** — use `boto3.client("bedrock-runtime")`
- **Inference profiles over model IDs** — individual model IDs are region-specific
- **`converse`** and **`converse_stream`** — Bedrock's primary API methods
- **Response path is deeper:** `response["output"]["message"]["content"][0]["text"]`
- **System prompt format:** `system=[{"text": "..."}]` — a list of dicts, not a bare string
- **`inferenceConfig`** — temperature, stop sequences, and generation params go inside this dict

Key takeaways

- If you get "model does not exist" errors, switch from model ID to inference profile ID.
- Temperature defaults to 1.0 in Bedrock — lower it explicitly for deterministic tasks.
- AWS IAM handles auth — no API key needed.

```
import boto3

client = boto3.client("bedrock-runtime", region_name="us-west-2")
model_id = "us.anthropic.claude-sonnet-4-5"  # use inference profile ID

response = client.converse(
    modelId=model_id,
    messages=[{"role": "user", "content": [{"text": "Hello"}]}],
    system=[{"text": "You are a helpful assistant."}],
    inferenceConfig={"temperature": 0.3}
)
text = response["output"]["message"]["content"][0]["text"]
```

Course 12 — Claude with Google Cloud Vertex AI

Accessing Claude through Google Cloud's Vertex AI.

Key concepts

- **`AnthropicVertex` client, not `Anthropic`** — install `anthropic[vertex]`
- **GCP auth via gcloud CLI** — no API key; use `gcloud auth application-default login`
- **`project_id` and `region` required** — `AnthropicVertex(region="global", project_id="...")`
- **Model IDs use version suffixes** — e.g. `claude-sonnet-4@20250514`; enable models in Model Garden first
- **API surface is identical to Anthropic SDK** after client init

Key takeaways

- Auth is the main hurdle: `gcloud init` → `gcloud auth login` → set project → `application-default login`
- Once authenticated, the API call is nearly identical to the base Anthropic SDK — easier transition than Bedrock.
- Use `region="global"` to avoid regional availability issues.

```
from anthropic import AnthropicVertex

client = AnthropicVertex(region="global", project_id="your-project-id")

message = client.messages.create(
    model="claude-sonnet-4@20250514",
    max_tokens=1000,
    messages=[{"role": "user", "content": "Hello"}]
)
print(message.content[0].text)
```

Course 13 — Teaching AI Fluency

How to teach the 4D Framework across disciplines, with assessment design.

Key concepts

- **Four teaching approaches:** linear (sequential 4Ds for beginners), non-linear (start anywhere), focused (deep dive into one D), two-loops (nested strategic/tactical)
- **Three assessment types:** outcome-based, process-based (annotated chat logs), reflection-based (metacognitive journals)
- **Discipline-specific adaptation:** making tacit field knowledge explicit is what makes AI fluency training meaningful

Key takeaways

- Build a reusable "teaching context document" at the start — students' background, goals, constraints.
- Effective AI fluency rubrics use three performance levels with observable, concrete behaviors — not vague terms.
- For assessment volume: use rubrics upfront, lean on peer review, do short "lightning round" conferences, sample strategically.

Course 14 — AI Fluency for Nonprofits

Mission-first AI use for grant writing, donor comms, data analysis, and policy.

Key concepts

- **Mission lens:** every AI efficiency gain should translate to greater community impact
- **Data privacy tiers:** match the AI tool to data sensitivity; consumer tools are not appropriate for PII or beneficiary information
- **Validation before trust:** before using AI for data analysis, test it against historical data where you already know the answers
- **Organizational AI policy:** platform selection, task delegation boundaries, quality oversight, and transparency disclosures

Key takeaways

- **Context-rich prompts dramatically outperform generic ones** — always include who you are, who you serve, and what you need.
- Upload past successful work (grants, reports) to help AI match your org's voice before drafting new content.
- **Sanitize data before sharing:** replace names with identifiers (Person A, Donor 1), remove contact info, generalize location details.
- Being "human in the loop" means ensuring AI serves your mission and your relationships stay human-centered.

Course 15 — Introduction to Agent Skills

Creating, configuring, sharing, and troubleshooting Claude Code skills.

Key concepts

- A skill is a directory containing a `SKILL.md` file with YAML frontmatter and instructions
- Skills load **on demand** (only when the request matches) vs. CLAUDE.md which loads every session
- **Personal skills:** `~/.claude/skills`. **Project skills:** `.claude/skills` (version-controlled)
- Priority order for name conflicts: Enterprise > Personal > Project > Plugins
- `allowed-tools` field restricts which tools Claude can use during the skill
- **Subagents don't inherit skills automatically** — list skills explicitly in the agent's frontmatter

Key takeaways

- If you repeat the same instructions to Claude, that's a skill waiting to be written.
- Write descriptions that answer: what does the skill do, and when should Claude use it?
- When a skill doesn't trigger, the fix is almost always the description — not the instructions.

```
---
name: pr-description
description: Writes pull request descriptions. Use when creating a PR or summarizin
allowed-tools: Read, Bash, Grep
model: sonnet
---

1. Run `git diff main...HEAD` to see all changes
2. Write a description with: ## What, ## Why, ## Changes
```

Course 16 — Introduction to Subagents

What subagents are, how to create them, and when to use them.

Key concepts

- A subagent is a **separate conversation context** that receives a task, works in isolation, and returns only a summary
- Built-in subagents: General purpose, Explore (fast codebase search), Plan (research before planning)
- Custom subagents are defined in `.claude/agents/` with YAML frontmatter: `name`, `description`, `tools`, `model`, `color`
- The description controls when the subagent triggers AND what input the parent passes to it
- Add `"proactively"` to the description to have Claude auto-trigger the subagent without being asked

Key takeaways

- Use subagents when the exploratory work doesn't need to be visible to the main thread.
- A **code review subagent** produces better feedback — fresh context, no bias from the coding session.

- Limit tool access to only what the subagent needs: read-only tools for research, Edit/Write only for agents that modify files.
- **Decision rule:** does the intermediate work matter? If no, delegate. If yes, keep it in the main thread.

```
---
name: code-quality-reviewer
description: Use this agent to review recently written or modified code for quality
tools: Bash, Glob, Grep, Read
model: sonnet
color: purple
---

Provide output in this format:

1. Summary
2. Critical Issues
3. Major Issues
4. Minor Issues
5. Approval Status
```

Course 17 — AI Capabilities and Limitations

A mental model of how AI works — four core properties — to predict failures and apply targeted fixes.

Key concepts

- **Next Token Prediction:** AI generates text by predicting the most likely next token. Source of both fluency and hallucination.
- **Knowledge:** Fixed at training cutoff. Uneven coverage — mainstream topics are strong, niche/post-cutoff topics are weak.
- **Working Memory (context window):** A fixed-size container. Truncation is silent. "Lost in the middle" effect: attention peaks at start and end, middle loses ~30%+ accuracy.
- **Steerability:** Instructions followed via pattern-matching, not understanding. Short, concrete, verifiable instructions land reliably. Long chains drift.

- **Training fingerprints:** Fine-tuning leaves predictable biases — sycophancy, default verbosity, loose confidence calibration.

Key takeaways

- **Fabrication concentrates in specificity:** names, dates, citations, statistics, URLs. The more precise the claim, the more it needs verification.
- **Don't bury critical instructions in the middle of long context.** State key constraints at the start and repeat near the end.
- When an instruction is followed literally but uselessly, restate the goal — not the instruction with more force.
- Use product features as targeted mitigations: citations for NTP, RAG for Knowledge, compaction for Working Memory, structured output for Steerability.

Course 18 — AI Fluency for Small Businesses

A practical, non-technical introduction to AI for small business operations.

Key concepts

- **The 4D Framework has two loops:**
- **Outer loop — Delegation + Diligence:** when/whether to use AI, and owning the result
- **Inner loop — Description + Discernment:** clear prompting, then evaluating what comes back
- **Delegation:** ask "should AI do this?" not just "can it?" FAQs stay AI. Complaints and judgment calls stay human.
- **Description** = context + goal + format + tone. Think of it as writing instructions for a capable, very literal new employee.
- **AI briefing sheet:** your business mission, team, values, and constraints — paste into future conversations to skip re-explaining.

Key takeaways

- **Create an AI briefing sheet once and reuse it.** Ask AI to interview you about your business, then synthesize it into a portable context doc.
- For automation decisions, categorize tasks into three buckets: AI handles it solo / AI drafts + human approves / human handles it entirely.

- **Data hygiene before sharing with AI:** strip names, contact details, exact figures, proprietary pricing. Replace with "Customer A / Vendor X."
- Build an AI use policy with three sections: what you use AI for, what stays human, and how you stay accountable.
- AI should free you for more human work — the relationship building, the judgment call, the handwritten note.

Built by AI Operator — 18 courses, 376 lessons. aioperator.com