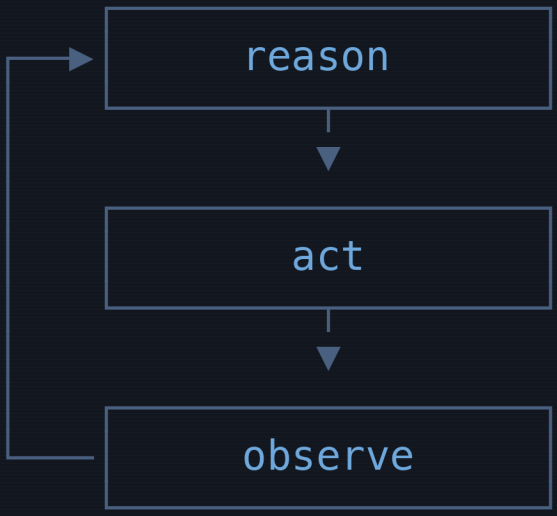
$ the demo is easy. the system is the work.

Building Agentic Systems

A Practical Guide to AI System Engineering



Yuri Syuganov

Building Agentic Systems

Building Agentic Systems

A Practical Guide to AI System Engineering

Yuri Syuganov

Building Agentic Systems

A Practical Guide to AI System Engineering

Early Access Edition · Version 0.9 · 2026-07-12

Set in Charter, IBM Plex Sans, and IBM Plex Mono. Typeset with Pandoc and XeLaTeX.

.

Contents

CHAPTER 1

Introduction: How to Use This Book

This book teaches you how agentic systems work and how to build them. It is designed for two types of readers, and you may be both at different times.

Two Ways to Learn

Path A: The Claude Code Path (Primary)

If you have a Claude Pro, Team, or Enterprise subscription with Claude Code access, you already have a production-grade agentic system at your fingertips. Claude Code is itself a deterministic shell around a probabilistic core — exactly the architecture this book teaches. It has tools (file reading, writing, search, bash execution), structured outputs, state management, and human-in-the-loop design built in.

On this path, you will:

- **Read the concepts** to understand what Claude Code is doing under the hood — why it works when it works, and why it fails when it fails
- **Direct Claude Code** to build the systems described in this book, observing how it implements the patterns rather than typing every line yourself

- **Configure the environment** — CLAUDE.md files, settings, hooks, and skills are your primary tools for shaping agent behavior
- **Review and refine** what Claude Code produces, using the mental models from each chapter to evaluate quality

This path does not mean you are passive. Directing an agentic system effectively requires understanding the architecture deeply. You need to know what a ReAct loop is to recognize when Claude Code is stuck in one. You need to understand structured outputs to know when to add validation. You need to grasp state management to design resumable workflows. The concepts are not academic — they are the vocabulary you use to collaborate with the tool.

Cost model: Flat-rate subscription. You do not pay per token. This changes how you think about experimentation — you can iterate freely, try multiple approaches, and let Claude Code explore without watching a billing meter. The cost engineering chapter (Chapter 18) covers this in detail, but the short version is: your constraint is time and attention, not dollars.

Path B: The API Path (For Deeper Understanding)

If you want to build agentic systems from scratch using API keys — or if you need custom infrastructure that goes beyond what Claude Code provides — every chapter includes implementation code using the Anthropic Python SDK.

On this path, you will:

- **Write the code yourself** — every loop, every tool registry, every validation layer
- **Call the API directly** with your own API key, paying per token
- **Build and own** every layer of the system

This path gives you the deepest understanding and the most control. It is also more work and more expensive during development. Many production systems eventually need this level of control, so the knowledge is never wasted even if you start on Path A.

Using Both Paths

The paths are not exclusive. A practical approach:

1. **Start on Path A.** Use Claude Code to build the companion project. Watch how it implements each pattern. Read the conceptual sections to understand what you are seeing.
2. **Switch to Path B when needed.** When you need behavior that Claude Code does not support, or when you want a standalone system that runs without human interaction, the API implementation sections give you everything you need.
3. **Use Path A to accelerate Path B.** When you do need custom code, direct Claude Code to write it. You will be a better director because you understand the architecture.

The Companion Project

Throughout this book, we build a system called the **Task Operator** — an agentic system that takes a goal, plans its approach, executes steps using tools, manages state, and produces artifacts.

On Path A: You will direct Claude Code to build this system in your project directory. Each chapter adds a capability. You configure behavior through CLAUDE.md and project settings.

On Path B: You will write the system in Python, calling the Anthropic API directly. Each chapter adds code to the growing codebase.

Both paths produce the same system. The difference is who types the code.

What You Need

For Path A:

- A Claude Pro, Team, or Enterprise subscription with Claude Code access
- A terminal (Claude Code runs in your terminal or IDE)
- A project directory to work in

For Path B:

- An Anthropic API key (sign up at console.anthropic.com)
- Python 3.10+
- The `anthropic` and `pydantic` packages

For both paths:

- Curiosity about how these systems actually work
- Willingness to read code, whether you wrote it or Claude Code did

A Note on the Concepts

Every concept in this book — deterministic shell, probabilistic core, structured outputs, tool registries, state management, orchestration patterns — applies regardless of which path you take. These are not implementation details. They are the mental models that separate people who can direct agentic systems effectively from people who are just hoping the magic prompt works.

When you tell Claude Code to "build a pipeline with validation between steps," you need to know what that means to evaluate whether it did it correctly. When a multi-agent workflow produces wrong output, you need the vocabulary to diagnose whether it is a shell problem or a core problem. When someone asks you to design an agentic system for your team, you need the architectural patterns to make sound decisions.

The concepts are the point. The implementation path is a choice.

Let's begin.

CHAPTER 2

Quick Start: From Zero to Orchestration

If you are a working engineer who wants to use Claude Code effectively right now, this chapter is your complete setup guide. It covers the mental model, the working loop, the project structure, and the exact files you need. Everything here is grounded in Claude Code's official patterns.

If this chapter is all you need, that is fine. The rest of the book explains *why* these patterns work, what happens when they break, and how to design systems that go beyond what a single Claude Code session can handle.

The Mental Model

Two things separate an effective operator from a casual chat user: **how much machinery** they deploy, and **what loop** they run it in. Get the loop right first — machinery amplifies whatever loop you have.

Three levels of machinery

Chat mode is one smart worker in one context window. You talk to Claude, it reads files, makes edits, runs commands. This is where you start and where most work stays.

Subagents are specialized workers spawned inside that one session. Claude can delegate a subtask to a researcher agent (cheap, read-only, fast) while it focuses on implementation. Subagents run within the same session

and return their results to the main conversation — and they keep exploration debris out of your main context.

Agent teams are multiple full Claude Code sessions with a lead coordinating them. Each has its own context window and can work in parallel. Most powerful, most expensive. Use them only when workers truly need to operate independently, because teams consume significantly more tokens.

The rule: start in chat mode. Add subagents when you need role separation. Graduate to teams only for genuinely parallel, independent work.

The loop that matters

Casual use is *prompt → accept → repeat*. The frontier loop is:

1. **Plan before code.** For anything non-trivial, get a plan and approve it before edits happen (plan mode, or simply "propose a plan first, don't edit yet"). A wrong plan costs one message to fix; wrong code costs a review, a revert, and your confidence.
2. **Delegate with contracts.** Say what done looks like — which files, what behavior, what must not change — not just what you want.
3. **Verify before believing.** A change is not done because the agent says so. It is done when the test passed, the endpoint responded, the screenshot shows the feature. Make the agent demonstrate, or demonstrate yourself.
4. **Review as a separate pass.** Production and evaluation are different jobs (Chapter 12). Run a review — a reviewer subagent, a review command, or a fresh session reading the diff cold — before you commit.
5. **Record what you learned.** A repeated instruction becomes a rule. A repeated procedure becomes a skill. A decision becomes a line in a decisions file. This is the compounding step, and skipping it is why month twelve feels like month one.

Every section below serves one of those five steps.

The File Layout

Claude Code loads project knowledge from a small set of well-known locations. Use each for what it is for:

- **CLAUDE.md** holds only the always-relevant project rules. It is loaded at session start and re-read after `/compact`. Keep it short.

- **`.claude/rules/`** contains scoped instructions that apply only when Claude works on matching files or areas.
- **`.claude/skills/`** holds reusable procedures and reference material. Skills load only when relevant — supporting files inside a skill stay out of context until needed.
- **`.claude/agents/`** defines subagents with specific roles, tools, and model assignments.
- **`.claude/settings.json`** holds permissions, hooks, and defaults — the enforceable part of your shell.

Permanent rules go in CLAUDE.md. Scoped rules go in rules files. Reusable procedures go in skills. Role-specific behavior goes in agents. Enforcement goes in settings. That division is the main way to avoid loading everything at once — and it is Chapter 4's deterministic shell, expressed as documentation.

CLAUDE.md: The Foundation

Your CLAUDE.md should be concise and specific. It survives compaction and is re-read fresh after `/compact`, so everything in it must be worth re-reading every time.

```
# Project overview
This repo is a Python + Playwright test automation project.

# Core workflow
- Propose a plan before editing on any multi-file change.
- Always inspect existing patterns before creating new files.
- Prefer minimal diffs over broad refactors.
- After edits, run the smallest relevant verification first,
  then broader checks if needed.

# Commands
- Install: `pip install -r requirements.txt`
- Unit tests: `pytest tests/unit -q`
- E2E smoke: `pytest tests/e2e -m smoke -q`
- Lint: `ruff check .`

# Coding conventions
- Keep selectors inside page object files.
- Reuse existing helper methods before adding new ones.
- Avoid changing unrelated tests in the same task.

# Compact instructions
When using compact, preserve:
- files changed
- commands run
- failing tests and exact errors
- decisions already made
```

Adapt the specifics. The structure — overview, workflow, commands, conventions, compact instructions — is the template. Note the first workflow line: the plan gate lives in CLAUDE.md so you do not have to remember to ask for it.

Rules: Scoped Instructions

Rules in `.claude/rules/` scope instructions to specific areas so CLAUDE.md stays lean.

`.claude/rules/tests.md`:

```
Apply these rules only when working on tests:
- Prefer stable locators over brittle CSS chains.
- Keep assertions close to the user action they validate.
- For flaky UI tests, add diagnostic logging before adding retries.
```

You can have rules for tests, docs, database work, deployment — whatever repeats. The habit that fills this folder: **the second time you type the same correction into chat, it becomes a rule.** Corrections you make once are conversation; corrections you make twice are configuration.

Agents: Specialized Workers

Subagents are file-based definitions — YAML frontmatter plus a Markdown prompt — assigning a role, a tool set, and a model to a type of work.

`.claude/agents/researcher.md`:

```
---
name: researcher
description: >
  Investigates codebase areas and gathers evidence. Use proactively
  for exploration, root-cause analysis, and dependency tracing.
tools: Read, Glob, Grep, Bash
model: haiku
---

You are a focused research agent.
1. Find relevant files and commands quickly.
2. Do not edit files.
3. Summarize findings with exact file paths.
4. Call out uncertainty clearly. Keep output concise.
```

`.claude/agents/reviewer.md`:

```
---
name: reviewer
description: >
  Reviews changes for correctness, regressions, and maintainability.
  Use proactively after edits, before commits.
tools: Read, Glob, Grep, Bash
model: sonnet
---

You are a strict reviewer. Check: correctness, unintended side
effects, missing tests, risky assumptions, style drift.
Prefer actionable findings over commentary. Do not edit files.
```

Two design choices to copy. First, the model assignments: the researcher runs on the cheap fast tier, the reviewer on the capable tier — cost control through role design, not through abstinence. Second, the reviewer **has no edit access**: it judges the artifact instead of redoing the work (Chapter 12's production/evaluation split, in four lines of YAML).

Use them naturally in a session — "use the researcher to find where login state is created, then implement the fix, then have the reviewer inspect the diff" — or run a whole session as one agent with `claude --agent reviewer`.

Skills: Reusable Procedures

Skills hold procedures and reference material that load only when relevant. Supporting files inside a skill stay out of context until actually needed.

`.claude/skills/ship-check/SKILL.md`:

```
---
name: ship-check
description: Pre-commit checklist for the current changes.
disable-model-invocation: true
---

Run this only when the user explicitly asks.

1. Review `git diff --stat`
2. Run lint and the targeted tests for changed areas
3. Summarize remaining risk
4. Suggest a commit message
```

`disable-model-invocation: true` means the skill fires only when you type `/ship-check` — use that flag for anything with side effects or deliberate timing: deploys, releases, commits. For orientation material (repo maps, architecture notes), let skills auto-load instead, and keep the heavy reference in a separate file inside the skill directory so it costs nothing until used.

The test of a good skill: it encodes a procedure you would otherwise re-explain in chat. Your skills folder is your accumulated operating manual — its size after six months is a fair measure of whether your practice is compounding.

When skills, agents, and hooks prove themselves, **plugins** are the distribution layer: a manifest bundles them, a marketplace shares them, and one

install aligns a teammate — or your second machine. Solo on one laptop, you can ignore plugins; the moment your setup needs to exist in two places, they replace copy-paste as the delivery mechanism (Chapter 24).

Permissions and Hooks: The Enforced Part

Everything above is advisory — the model reads it and usually complies. `settings.json` is where rules become physics.

Permissions. Claude Code's permission system has modes from "ask about everything" to "ask about nothing." Three disciplines:

- Run interactive sessions in the default or accept-edits mode, so file edits flow but commands still gate. Reserve full permission bypass for sandboxes and throwaway work — a bypassed agent in a repo with your `.env` and deploy credentials is Chapter 20's threat model, voluntarily installed.
- Allow-list what should never need asking. The prompts in default mode are not a fixed tax — the system is designed to converge. Every prompt offers a "don't ask again" option that writes a permanent allow rule; take it, and each prompt is the last of its kind. Answer with a plain "yes" instead and you will approve `npm test` four hundred times, which is how people end up reaching for bypass:

```
{
  "permissions": {
    "allow": ["Bash(git diff:*)", "Bash(npm test:*)", "Bash(make:*)"]
  }
}
```

- Deny-list what should never happen regardless of mode:

```
{
  "permissions": {
    "deny": ["Edit(.env*)", "Bash(git push --force*)", "Read(secrets/**)"]
  }
}
```

Two habits make the allowlist converge faster. Keep command shapes stable — a prefix rule cannot match a compound whose shape keeps changing, so `cd api && npm test` prompts forever while `npm test --prefix api` matches its rule the second time you run it. And let read-only work stay read-only: commands the harness can verify as safe (greps, finds, builds

that touch nothing outside the repo) run in a sandbox without prompting at all, so the prompts that remain concentrate on state changes — exactly where you want your attention. For the accumulated backlog there is a built-in skill: `/fewer-permission-prompts` (at the time of writing) scans your past sessions for commands you keep approving and drafts the allowlist for you.

A converged allowlist is what Chapter 15 calls a well-designed approval gate: it fires rarely enough that you actually read what you are approving. If you are prompted once a minute, the fix is not to remove the gate — it is to teach the gate what routine looks like.

Hooks run your commands at fixed points in the agent loop — deterministic guardrails the model cannot talk its way around. One high-value example: lint after every edit, so drift is caught the moment it happens rather than at review:

```
{
  "hooks": {
    "PostToolUse": [
      {
        "matcher": "Edit|Write",
        "hooks": [{ "type": "command", "command": "ruff check --fix $CLAUDE_FILE_PATHS" }]
      }
    ]
  }
}
```

Prompts suggest; hooks and permissions enforce. When you find yourself repeating a safety instruction, it probably wants to be one of these instead.

Model and Cost Control

Model families change names faster than books reprint, so think in tiers, not versions: a **fast/cheap tier** for mechanical work, a **balanced tier** as the daily driver, a **deep tier** for architecture and thorny debugging. Map the current lineup to those tiers when you read this.

Role	Tier	Why
Main session	balanced	Best speed/intelligence tradeoff
Architecture, hard debugging	deep	Reasoning is the bottleneck

continued on next page

Role	Tier	Why
Research subagents	fast/cheap	Read-only, high volume
Reviewer subagents	balanced	Judgment matters

Switch mid-session with `/model` when the problem class changes, and back again after.

On a flat-rate subscription your scarce resources are **time, attention, and context** — not dollars (Chapter 18). The disciplines that matter:

- `/clear` between unrelated tasks; a context full of the previous task actively degrades the next one.
- `/compact` on long sessions — and give CLAUDE.md compact instructions so the right things survive.
- Keep CLAUDE.md short; push optional knowledge into skills; disable unused MCP servers.
- Set up a status line (`/statusline`) showing model and context usage, so context hygiene has a gauge.

Beyond One Session

Three multipliers, in the order you should adopt them:

Parallel worktrees. `git worktree` gives each concurrent task its own checkout, so two sessions never fight over the same files. One feature per worktree, one Claude session per worktree, merge when verified. This is Chapter 14's isolate-then-merge discipline, provided by git for free — and it is how you work on two features at once without conversation soup.

Headless mode. `claude -p "<task>"` runs a one-shot session with no UI — Claude Code as a scriptable Unix tool. That makes agents composable into cron jobs and CI: nightly issue triage, PR review on every pull request, a weekly "read the backlog and flag stale items" pass (Chapters 21 and 23). Anything you do on a schedule, an agent can do on a schedule.

Agent teams. Multiple coordinated full sessions. Powerful, experimental, expensive. The threshold: if workers genuinely need to operate in parallel on independent areas and communicate, use a team; if one worker's output feeds the next, use subagents in a single session. Most work never needs a team.

From Chat to Backlog

The habit that most changes long-term throughput has nothing to do with prompts: **work that outlives a session must live outside the session.** Chat history is where plans go to die — state belongs on disk (Chapter 9), and the practitioner version of that rule is a backlog.

Minimum viable version: a `plan.md` in the repo — direction questions you have not answered, features as headed sections, tickets as checkboxes. Ask Claude to keep it current as part of finishing any task ("check the box, add follow-ups you discovered"). Fancier versions use GitHub issues and skills that convert a PRD into tickets. Either way the test is the same: *if your laptop died tonight, would tomorrow's session know what to do next?* If the answer lives in a chat transcript, the answer is no.

The Progression

1. **Start with one normal session and the loop:** plan → delegate → verify → review → record. No new files needed.
2. **Put stable project rules into CLAUDE.md** — including the plan-first rule and your verify commands.
3. **Capture repetition:** second-time corrections become rules; re-explained procedures become skills.
4. **Add the reviewer** — a subagent or review pass gating every commit.
5. **Enforce what must never happen** with permissions and hooks.
6. **Externalize the backlog** — plan.md or issues; sessions start from it, not from memory.
7. **Scale out:** worktrees for parallel features, headless mode for scheduled work, teams for the rare truly-parallel task.

Most engineers stop at step 1 and plateau. Each later step compounds: the rules make every future session better, the reviewer catches what you stopped seeing, the backlog survives your context window. Steps 2–6 cost an afternoon, total.

The Complete Recipe

```
# 1) One session, balanced tier, plan-first
claude
# → "Plan before editing" lives in CLAUDE.md, not in your memory

# 2) Base memory
#    → CLAUDE.md: overview, workflow, commands, conventions, compact rules

# 3) Focused workers
#    → .claude/agents/researcher.md   (fast tier, read-only)
#    → .claude/agents/reviewer.md     (balanced tier, no edit access)

# 4) Reusable playbooks
#    → .claude/skills/ship-check/SKILL.md   (user-invoked, gates commits)

# 5) Enforcement
#    → settings.json: permission deny-list + post-edit lint hook

# 6) Backlog outside the session
#    → plan.md: direction questions, features, checkbox tickets

# 7) Hygiene
/clear        # between unrelated tasks
/model        # deep tier for hard problems, back after
/statusline   # keep context usage visible

# 8) Scale
git worktree add ../feature-x && claude   # parallel feature
claude -p "triage new issues"             # scheduled/headless
```

The key discipline: plan before code, verify before believing, review before committing, record before forgetting — with permanent rules in CLAUDE.md, scoped rules in `.claude/rules/`, procedures in skills, roles in agents, enforcement in settings, and the backlog on disk. That is orchestration without dragging every instruction into context on every turn.

If this is all you need, you are ready to work. The rest of this book explains the engineering behind these patterns — why they work, how they fail, and how to build systems that go beyond what any single tool provides.

End of the free sample

The full book: 26 chapters, 6 parts — through production reliability, cost, security, and the operator practice.

Updated quarterly. Buy once — every update is free.

leanpub.com/building-agentic-systems