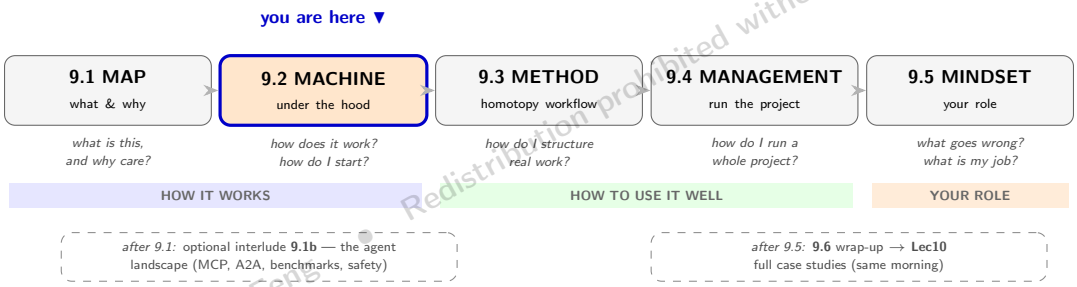
AI for Economic Research: Dynamic Models, Language, and Agents
Lecture 9.2: Under the Hood — How It Works and How to Start

Zhigang Feng

2026-07-06 07:04:56

The Map: Act 2 — The Machine

This deck answers the second question: how does it actually work, and how do I start?



Route: the three physical components and the tool-call loop → install and initialize → the shape of the CLI → your first session (on the fellows roster) → tokens, the context window, and the pillars that keep you honest.

The Machine

Harness, model, files — and the loop that joins them

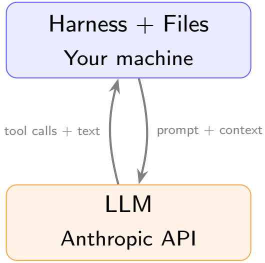
Three Components: Harness, LLM, Your Files

A terminal agent has three physically distinct parts — the same anatomy in Claude Code, Codex CLI, Gemini CLI, or Aider:

1. **The Harness** (lives on your machine)
   CLI software you install. Reads your files, runs shell commands, manages the conversation loop.

2. **The LLM** (lives on the provider's servers)
   Every message is bundled with conversation history and sent via HTTPS to the API. The "thinking" happens in the cloud.

3. **Your Files** (live on your machine)
   Python scripts, data CSVs, LaTeX—all on your disk. Code executes on your hardware with your packages.



Key: code can run locally and many project files stay on your machine. That does *not* mean the whole workflow is automatically private.

The Loop in Machine Language: JSON Tool Calls

Lec07 T3a explained that an LLM can emit structured JSON via schema-constrained decoding. In a terminal agent, every tool call uses that mechanism.

Example. When the agent decides to read a file, it emits:

```
{"tool": "Read", "args": {"file_path": "fellows_seed.csv"}}
```

The harness parses the JSON, executes `Read`, and feeds the file content back as the next message; the agent reads it and decides the next action.

- **Loop.** Model output (JSON) → harness execution → tool result → next model turn. Continues until the model emits a "done" signal or hits a stop condition.
- **Schema is the contract.** Each tool advertises an argument schema; outputs that fail the schema are rejected.
- **Same brain.** There is no separate "planner" — the same transformer that writes code also decides to call tools.
- **Key implication.** Every action the agent takes is a *next-token prediction* constrained by the available tool schema.

Privacy Boundary: Local Execution Is Not the Same as Local Processing

- code may execute locally on your machine
- many project files may stay on disk locally
- if file contents, prompts, or tool outputs are sent back in the conversation, those contents may still traverse the network
- only a fully local or institutionally private deployment changes that boundary

Practical rule: running code locally does not make it safe to put restricted data into the conversation — treat everything you type or attach as leaving your machine.

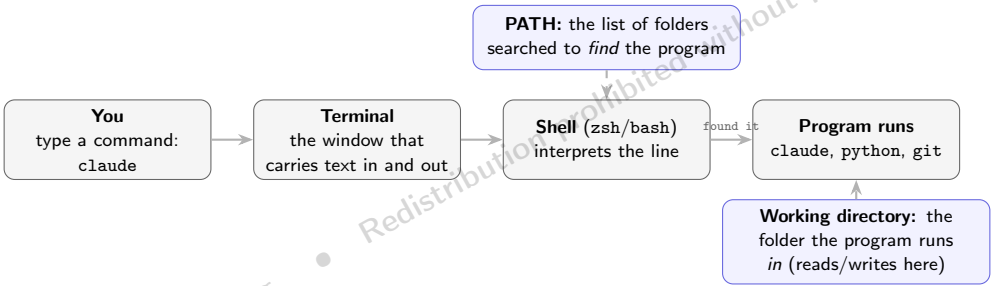
Terminal Vocabulary in 60 Seconds

Term	Meaning
Terminal	The text window where you type commands instead of clicking buttons.
Shell	The program that interprets commands. macOS often uses `zsh`; many examples are written in `bash`.
Bash	A common shell language. When Claude Code says it will use `Bash`, it means "run this command in the terminal."
Working directory	The folder commands run from. `cd my-project` changes it; `pwd` prints it.
PATH	The list of folders your shell searches when you type a command such as `python` or `claude`.

Why this matters: terminal agents act through shell commands. If the working directory or `PATH` is wrong, the agent may be competent but operating in the wrong place.

How the Terminal Pieces Fit Together

One picture for the five words: you type into the terminal; the shell finds the program via PATH and runs it in the working directory.



- When you type `claude`, the shell finds it via `PATH` and starts it *in your current folder* — that is why `cd my-project` comes first
- The agent's `Bash` tool is this same picture one level down: the agent typing into the same shell, in the same working directory

Start Here

Install, initialize, and the shape of the CLI

Install Once: Three Durable Steps

Setup has exactly three steps, and only the first command ever changes.

1. **Install the CLI** — one installer command for your OS (native installer script; Homebrew and `npm` also work):

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

2. **Authenticate once** — run `claude`; a browser window opens; log in with your account or API key
3. **Work from the project** — every session starts the same way:

```
cd my-project
claude
```

Install once per machine. Authenticate once per account. Initialize every session per project — automatically (next frames).

Durable vs. dated: this 3-step shape survives every product update; exact commands, subscription tiers, and the Windows/OneDrive caveats are dated detail → T6 Appendix A and the official docs.

Files First: Global vs. Local

Two layers of configuration. **Local overrides global.**

Path (Claude Code example)	Scope	What lives here
~/.claude/settings.json	Global	Default permissions, model preference
~/.claude/CLAUDE.md	Global	Your cross-project preferences
~/.claude/skills/	Global	Skills available in every project
~/.claude/projects/.../memory/	Global	Auto memory: the tool's own per-project notes
<project>/CLAUDE.md	Local	Project brain, read every session
<project>/.claude/settings.json	Local	Project permissions (overrides global)
<project>/.claude/skills/	Local	This project's slash commands
<project>/.claude/agents/	Local	Specialized reviewer roles
<project>/.claude/rules/	Local	Always-on guardrails

Other agents follow analogous layouts. Codex CLI uses AGENTS.md; Gemini CLI uses GEMINI.md. The pattern — global defaults plus a project-specific system prompt — is durable across products.

Rule: global dotfolder = your personal defaults (apply everywhere). Local dotfolder = the project's rules (apply only here). T4 fills the local layer with research content.

Initialization: What Happens When You Type `claude`

Five steps run **before you type your first message** — each one loads a file from the previous frame:

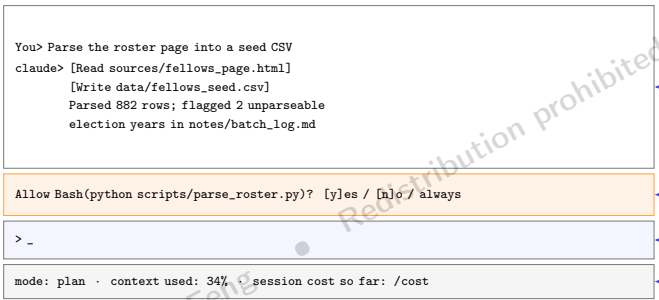
1. **Read global config** — `~/.claude/settings.json`: permission defaults, model preference
2. **Detect project context** — searches current directory (and parents) for `CLAUDE.md` and `.claude/`
3. **Load project memory** — reads `CLAUDE.md` (and the tool's auto memory for this project) into the context window
4. **Auto-load all rules** — every `.claude/rules/*.md` file is loaded silently
5. **Ready** — context window pre-populated before you say anything

Analogy: steps 3–4 are a professor re-reading lecture notes before class. Students arrive; context is already loaded.

Cost warning: steps 3–4 consume tokens before you type anything. Keep `CLAUDE.md` and rules files concise.

The CLI at a Glance: One Screen, Four Zones

Everything you will ever look at in a terminal-agent session is one screen with four zones.



Zone 1's `[Read ...]` traces are the JSON tool calls of the earlier frame, printed for you — the audit trail is on screen by default. Layout details differ across products and versions; the four zones are durable.

Commands, Modes, Keys: The Control Surface

Control	What it does	When to use
Modes (cycle with `Shift+Tab`)		
Plan mode	Read-only: the agent explores and drafts a plan, executes **nothing** until you approve	Start every non-trivial task here
Accept-edits	Auto-approves routine file edits	Only after trust is earned
Slash commands		
`/effort max`	Maximum reasoning depth: best quality, slower, costlier	Novel problems, economic correctness
`/effort default`	Balanced quality and speed	Routine work
`/cost`	Tokens used + estimated USD this session	Budget check before long tasks
`/compact`, `/clear`	Manage the context window	Next section of this deck
Keys		
`Esc`	Interrupt the agent mid-task immediately	When it's going the wrong way

The effort–cost tradeoff: maximum-effort modes plus long context can be an order of magnitude more expensive than default. Spend effort on *decisions*; save it on *execution*.

Permissions: The Blast-Radius Dial

The agent's toolbox:

Tool	What it does
`Bash`	Run any shell command
`Read/Grep`	Read and search files
`Write`	Create or overwrite a file
`Agent`	Spawn a child instance

"Which Python ran this?"

- the agent's `Bash` inherits your shell's `$PATH`: if you ran `conda activate myenv` before `claude`, *that* Python runs
- add a `which python` check to `CLAUDE.md`

The permission model:

- **Default: ask.** You see the exact command before it runs
- **Allowlist** trusted patterns in `.claude/settings.json`:

```
"permissions": {
  "allow": ["Bash(python *)",
            "Bash(git *)"],
  "deny":  ["Read(.env)",
            "Bash(rm -rf *)"] }
```

- **Bypass-permissions** skips all prompts: isolated sandboxes only

Think of authorizing a new RA. Default = every action needs your signature; allowlist = standing approval for routine work; deny = never allowed; bypass = handing over your own login. *"Blast radius"* = how much damage one wrong action could do before you notice.

Your First Session

A real walkthrough on the fellows roster

First Session (1): Launch on a Real Project

Initialization in the abstract → what it actually looks like, on the fellows project from T1.

```
$ cd ~/research/es-fellows
$ claude
[reads ~/.claude/settings.json]
[reads ./CLAUDE.md]
[loads .claude/rules/provenance.md]
[ready]

You> Parse sources/fellows_page.html into data/fellows_seed.csv
     with columns: name, affiliation, election_year, profile_url.

claude> [Read sources/fellows_page.html]
        [Write data/fellows_seed.csv]
        Parsed 882 rows. Two election years did not parse
        (flagged in notes/batch_log.md). Spot-check 5 rows?
```


1. Initialization is automatic; no flags needed for the first session
2. The agent reads context (rules, references) *before* acting — visible as bracketed `[Read ...]` traces
3. Each trace is a JSON tool call that the harness translated into a real disk operation
4. Note the agent's last line: a competent session *ends turns with a checkable claim*, not just "done"

First Session (2): Pen and Paper, Then Plan Mode

0. **Before the agent—pen + paper**
   Write one sentence: *"I want to compute X using method Y for question Z."*
   Fellows version: "a provenance-aware seed roster of all current ES Fellows, from the official page only."

1. **Enter plan mode (`Shift+Tab`)**
   Describe your goal. The agent drafts a plan.
   Nothing runs yet.

2. **Review and approve**
   Read the plan. Ask questions. Revise. Then approve it.

Why this matters: plan review is the cheapest place to catch a wrong task definition.

Common beginner mistake: skipping Step 0. Without a clear goal, the agent will confidently build the wrong thing.

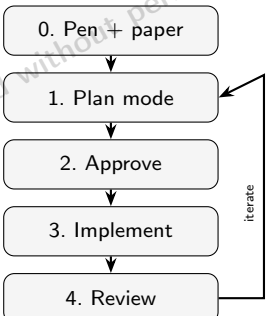
What exactly to check before approving a plan: T3 gives the five criteria.

First Session (3): Implement, Review, Iterate

3. **Implement**
   The agent executes the approved plan. Press `Esc` to stop if wrong.

4. **Review output**
   Does it make economic sense? Re-derive one number by hand.
   Fellows version: open 5 random rows; follow each source URL.

5. **Iterate**
   Fix, extend one feature, compact context, and return to step 1.



Practical takeaway: after each implementation pass, force one human check before you ask for the next extension.

What the First Session Leaves Behind

Close the terminal. What survives is not the conversation — it is the project state on disk.

In the folder now:

- `data/fellows_seed.csv` — the artifact (882 rows)
- `notes/batch_log.md` — what parsed, what was flagged
- `CLAUDE.md` — two new lines: the parse rules you settled on
- `.claude/rules/provenance.md` — the source rule, now always-on

Why this is the whole game:

- chat history is session-bound (resumable, but do not rely on it); **files persist and compose**
- next session, initialization reloads all of this *automatically* — batch 2 starts where batch 1 ended
- "the project files hold the rules and state across sessions" — the public-web analog of `CLAUDE.md` (Lec10 T4)

A session is good when it improves the project state, not when the conversation felt smart.

Close the Loop: Reflect, Save, Shrink

The last five minutes of a session are worth more than the next fifty. Three habits.

1. **Reflect.** End with a two-minute review: what worked, what you corrected, what surprised you. Ask the agent itself to summarize the session's decisions — it is cheap and surprisingly useful
2. **Save.** Write that summary into the plan or progress file, and update `CLAUDE.md` while it is fresh. The next session then starts from the updated plan, not from anyone's memory
3. **Shrink.** Prefer several small, focused sessions over one giant thread: a small context is cheaper, and the model's recall is sharper (mechanics in the next section). Break the big project into bounded sessions — *one intermediate product per session*

A big project is a sequence of small sessions, each ending with an updated plan. T4 turns this rhythm into a management system.

Budget and Discipline

Tokens, cost, the context window — and the pillars that keep you honest

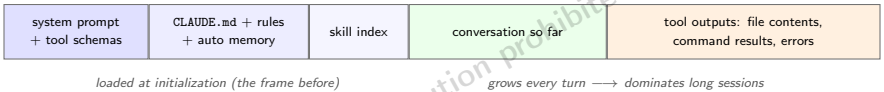
Tokens and Cost on the Radar

Every agent turn is a model API call. Cost scales with input and output tokens.

- **Input tokens.** Your prompt, conversation history, `CLAUDE.md`, rules, retrieved chunks. Every step that adds context adds cost.
- **Output tokens.** Generated code, JSON tool calls, prose explanations. Reasoning-heavy modes also produce many internal tokens.
- **Long sessions accumulate context.** Watch the status line; check `/cost` before long tasks.
- **Pin for reproducibility.** Record the model version, prompt template, and seed when applicable; AEA disclosure (Oct 2024) extends to AI-assisted analysis.

The Context Window Is the Agent's Working Memory

Everything the model "knows" during a turn sits in one finite window (Lec07 T3b). Here is what fills it in an agent session:



- The window is a **finite budget**: model-dependent, but always exhaustible by a long session
- Recall is not uniform — material buried mid-window is recalled worst ("lost in the middle," Lec07 T3b); we do not reteach it, we *manage around* it
- Consequence: **long sessions degrade** — the agent slowly forgets early decisions and corrections

Compaction: What `/compact` Keeps and What It Drops

When the window fills, the harness summarizes: that is compaction. It is a lossy operation, and you should treat it as one.

- `/compact` = **summarize and clear**: older tool outputs are dropped first, the dialogue is condensed into a summary, and `CLAUDE.md` + rules are re-read from disk
- **What survives:** the decisions, the current task, key snippets. **What often does not:** the *reasoning* behind decisions, rejected alternatives, small corrections you made in passing
- **Auto-compaction:** by default the harness compacts on its own as the window fills — convenient, but the same information loss applies whether you triggered it or not
- `/clear` = **full reset**: conversation gone, files and memory remain; right choice when switching to an unrelated task

Rule: state belongs in files, not in the conversation. If a decision matters tomorrow, write it to `CLAUDE.md` or a progress note *before* the window fills — then compaction costs you nothing.

That is the mechanics. T5 returns to what this trade-off costs research integrity — and why more context is not always better.

The 5-Pillar Spine, One More Time

The five pillars (introduced in Lec01, exercised throughout Lec06) reappear here as the agent-era research workflow.

1. **Theory**
2. **Algorithm**
3. **Numerics**
4. **Advanced**
5. **Code literacy**

The agent accelerates each pillar; it replaces none of them.

Rendering	One-line discipline
1. Question design	Economic language; define the evidence; limiting cases first
2. Procedure design	The right procedure *and* its acceptance criterion, before code
3. Data/numerical discipline	Grids, weights, crosswalks, missingness — saved, not implicit
4. Domain constraints	Know when the agent's convenient default is wrong
5. Code literacy	Read what was built; verify it is what you meant

The throughline: pillars 1–4 decide *what should be built and what counts as success*; the machine only decides how fast. Weak pillars = faster nonsense.

Division of Labor: You Provide, the Agent Provides

You provide	Agent provides
Question, identification logic, and source rules	Fast code generation and refactoring
Workflow choice and benchmark design	Boilerplate implementation and debugging
Knowledge of the data, model, and institutions	File edits, script writing, and test runs
Economic interpretation and seminar defense	Iteration on visible errors and outputs

Research version of the slogan: the agent can be a strong RA, but you still have to be the principal investigator.

Validation Is Your Referee Checklist

Every iteration needs both workflow checks and economic checks.

Workflow checks

- Convergence achieved
- No NaN or Inf values
- Mappings, weights, or source logs saved where expected
- Diagnostics, residuals, or quote checks pass
- Script reproduces the same output twice

Economic checks

- Limiting cases recover known results
- Comparative statics, subgroup patterns, or review claims make sense
- Magnitudes and source coverage are plausible
- Inference uses the right standard errors, sample, and restrictions
- You can explain the result in plain economic language

If you cannot defend the output in a seminar, you are not done.

Bridge: The Machine Needs a Method

You can now install the tool, launch it, steer it, budget it, and read its screen. What you cannot yet do is structure a week of research with it.

- Act 2 covered the *mechanism*: components, tool calls, install, initialization, the CLI, permissions, the context window
- The next question: *"how do I structure real work?"* — what to ask for first, how big a step to take, when to trust a result
- Act 3's answer is the **homotopy workflow**: build the easy version, validate, then extend one margin at a time — the exact discipline behind HA-Ramsey

Arc: T1 Framing → **T2 Under the Hood** → *T3 Homotopy* → *T4 Project Org* → *T5 Mistakes/Context* → *T6 Wrap.*