THE CLAUDE LEVERAGE PLAYBOOK · THE FULL 4-MOVE PROGRAM

From "we use Claude" to "we ship twice as much"

The bottleneck is almost never the model. Every team already has Claude in the building; the step-change comes from turning **ad-hoc individual usage into shared, versioned infrastructure the whole department draws on.** This is the complete program - four moves: **baseline** so the gain is real and provable, a **codified operating model**, an **internal tooling layer** where the 2x lives, and a **measured rollout** that makes the gain compound. Built from the stack I run in production every day.

THE TELL · HOW TO SPOT WHO HASN'T GOT THERE YET

Frustration with "Claude rabbit holes" is one of the tells

When an engineer complains that Claude "goes down rabbit holes", over-builds, or wanders, that is often a strong sign they **haven't learned to leverage it** - they lack the scoping, guardrails, and judgment that keep it on-rails. A high-leverage engineer rarely hits rabbit holes; when the agent wanders they read it as "my scope was thin", not "the tool is broken". A team survey for this feeling is a fast read on where the operating model and tooling below will lift the most.

MOVE 1 Baseline & diagnose

No before-number, no credible "twice as much." Move 1 is what makes every later claim provable instead of a vibe.

Instrument current usage, survey the team, and pick the highest-leverage workflows to target. Then set a **balanced baseline** you will measure everything against:

- **A speed metric** - lead time to production for a defined class of work.
- **A quality guardrail** - change-failure rate, so speed isn't bought with breakage.
- **A human signal** - developer-reported time-on-toil (and/or satisfaction).

Deliberately not measured: lines of code, PR count, story points, tokens spent - motion metrics that reward busywork and are trivially gamed. See "How we measure success" below.

MOVE 2 The operating model (best practices)

Codify what your best people already do so the whole department inherits it. Six SOPs.

SOP 2.1 Per-repo context file (the single highest-leverage artefact)

Put a short, load-bearing **CLAUDE.md** at each repo root that teaches the agent your conventions *before* it writes a line. Keep it a lean **navigator** that points to topic rule files (progressive disclosure), never a wall of text that burns context.

```
# CLAUDE.md - the navigator, not the manual
Read the topic rule for the area you're touching:

| Topic       | File                    | When to read        |
| stack rules | rules/stack.md          | any code change     |
| security    | rules/security.md       | auth / secrets      |
| review      | rules/review.md         | before a PR         |

Architecture, house patterns, and the "never do this" list live in
those files - loaded on demand, not all at once.
```

SOP 2.2 A prompting-pattern catalogue

Write down the handful of shapes that reliably work, so they are teachable, not tribal:

- **Plan-first.** The agent proposes a plan and you approve scope before it edits. The single biggest anti-rabbit-hole move.
- **Explore-before-edit.** Read and map the code (a read-only pass) before touching it.
- **Adversarial self-review.** Make the agent try to refute its own change before it ships.
- **Expert council.** Convene named personas backed by current library docs during planning, so it plans against today's APIs, not two-year-old ones.

SOP 2.3 A when-to-use / when-not decision

Name where the agent multiplies (boilerplate, migrations, test scaffolding, code review, research sweeps) **and where a human still leads** (novel architecture, security-critical paths, ambiguous product calls). Naming the "not" cases is what keeps trust high.

SOP 2.4 Guardrails as hooks (make the rules enforced, not suggested)

The failure modes that make leaders distrust AI code - leaked secrets, confident-but-wrong slop, silent scope creep - are best stopped by **deterministic hooks** on the agent's lifecycle, not by hoping it reads a rule. Real examples from my own setup:

- **Secret guard.** Deny reads and writes of secret-bearing files (.env, keys, credentials) and any command carrying a literal secret value.
- **Memory / structure guard.** Block writes that would drop files in the wrong place, so the repo's structure can't drift.
- **Environment guard.** Warn on a global `pip install` outside a virtualenv before it poisons the system interpreter.

A hook is a small script the agent runtime calls before/after a tool. It either allows, rewrites, or hard-denies the action - the guardrail runs whether or not anyone remembered the rule.

SOP 2.5 Code discipline: readable over commented, docs only when they earn it

- **Write self-documenting code, not comments.** Name things well and structure the code so it reads on its own; a comment that restates the code is a smell. Instruct the agent to prefer clarity over annotation.
- **Document only when it adds value.** A README or doc is worth it when it saves a reader real time (a non-obvious setup, a decision's "why"); don't have the agent generate docs by reflex.
- **Review discipline.** A shared standard the agent's output is *written to pass*, plus a per-repo anti-patterns list the review sweeps against, so speed is never bought with quality.

SOP 2.6 Task files: scope the work so it can't rabbit-hole

The cure for rabbit holes is **a written unit of work with a hard boundary.** Give the agent a task file - a clear scope, a definition of done, and a terminal smoke test - and it has a target to hit and a place to stop. Size by how autonomous the work is, and split anything too big into an epic of smaller tasks.

```
# tasks/<slug>.md
status: inbox            # promote to active on the real start date

## Phase 1 - <grouping>
- [ ] <one implementation step>

## Definition of Done
- <the observable outcome that means "shipped">

## Smoke Test          # terminal section - how a human verifies it
- [ ] `curl -s .../health`  → expect 200
```

PUBLIC WRITE-UPS ON PIECES OF MOVE 2

The Claude Code Router Pattern - gabegiro.com/blog/claude-code-router-pattern
Don't Make AI Your Crutch - gabegiro.com/blog/effective-use-ai-written-code

MOVE 3 The internal tooling (the multiplier)

Move 2 makes each interaction better. Move 3 makes each *good interaction reusable* - one-off prompts become shared, versioned infrastructure every engineer pulls from. **This is where the 2x actually lives.** Seven SOPs.

SOP 3.1 A skill / command library, kept lean by a hard cap

Package the jobs your team does repeatedly as invocable **skills** (spin up a service, run the migration playbook, generate the test suite). The trap is sprawl - every skill's description loads into context on every turn, so an unbounded library quietly taxes speed. Govern it:

- **Cap the catalogue** (I hold ~40) and run a monthly audit that must propose consolidations before it exits.
- **Route, don't add.** A new capability must replace one, fold into an existing router (a dispatcher skill with sub-files, enumeration cost zero), or scope to a single project - before it earns a fresh top-level slot.

SOP 3.2 A hooks library, delivered to every machine from one manifest

Version the guardrail + automation hooks from Move 2 as a library with a **manifest + one installer**, so a fresh machine gets the scripts *and* their registrations in one command - no hand-wiring per developer. That is what makes the guardrails uniform across the whole team instead of one person's local trick. I drive this with a **/machine setup|update** routine and a dedicated cross-machine matrix that records which item is installed on which machine, so gaps are visible instead of silent.

SOP 3.3 A memory system so the agent knows your codebase

A **two-layer** setup: a durable, indexed store of project knowledge (an index file plus one fact per note), and a staging layer for corrections you capture mid-session and file to their permanent home. The agent accumulates understanding of *your* system across sessions instead of starting cold every time - the difference between smart autocomplete and a teammate who knows the code.

SOP 3.4 Agent swarms for the high-volume, high-toil work

Purpose-built multi-agent routines that fan out and verify. The use cases that pay off first:

- **Code-review swarm.** Parallel reviewers over a diff, each a distinct lens (correctness, security, performance), every finding adversarially verified before a human sees it.
- **Migration sweep.** One mechanical change across hundreds of call-sites, each in an isolated worktree so parallel edits don't collide.
- **Test generation.** Fan out over under-tested modules and scaffold coverage - but predefine what a *useful* test is; chasing 100% coverage is usually the wrong target, since it rewards trivial tests over the ones that catch real regressions.
- **Research / audit fan-out.** Many readers over different subsystems returning a structured map; loop-until-dry for unknown-size discovery (keep finding until N rounds surface nothing new).
- **PR triage.** First-pass classification so humans spend attention where it counts.

SOP 3.5 Ultracode: deterministic multi-phase orchestration

When a job is bigger than one agent's context or needs real rigour, script the orchestration itself - decompose into phases and pipeline the work, so control flow is deterministic rather than model-guessed. When to reach for it:

- **Understand → design → implement → review** as explicit phases for a large feature, staying in the loop between each.
- **Broad migrations / audits** that one context can't hold - discover the work-list, then pipeline every item through transform-and-verify.
- **Confidence-critical work** - generate N independent approaches, judge them, synthesize the winner; or verify every finding with independent skeptics before it counts.

Rule of thumb: reach for a swarm when the work is wide (many similar items); reach for ultracode when it is deep (phases that must happen in order, with verification gates).

SOP 3.6 Pick a push strategy that fits the team

How agent-authored work reaches main is a real choice, not a default. The main options and when each fits:

- **PR-based (branch → review → merge).** Safest and most auditable; best for larger teams, regulated code, or where a human gate is required. The agent opens a reviewable PR; humans merge.
- **Direct-to-main.** Fastest flow for a trusted solo/small team with strong hooks and tests as the gate. The guardrails (Move 2) are what make this safe without a PR.
- **Work without pull request approvals.** For high-trust teams - changes still land as commits/branches behind CI and review discipline, but without a blocking human approval gate. The point is removing the approval bottleneck and its queue latency, not the record.

SOP 3.7 CI + orchestration for unattended work (the compounding layer)

Pipe the above so it runs without a human babysitting: **overnight PR runs**, scheduled sweeps, an agent that picks up a well-scoped task, does it, and opens a reviewable PR by morning. Every unattended run is gated by the same review discipline and hooks - so work happens while the team sleeps and quality holds. The use cases that pay off first:

- **Security-patch PRs.** The moment a scan flags a vulnerability, the agent drafts the fix as a reviewable PR. Usually the fastest ROI - bounded, verifiable, and off the critical path.
- **Red-build & flaky-test triage.** On a failed run it reads the logs, correlates against the recent commits, and posts the likely root cause - plus a fix PR for a flaky test - before anyone gets paged.
- **Dependency upgrades with the fix attached.** Bump dependencies off-hours and repair the breakage they cause, so you get one reviewable PR instead of a wall of red.

Where it earns its keep, honestly: start read-only, gate every write behind review + hooks, and set hard limits - an unattended agent plus a flaky test plus a retry loop is how it runs away. Guardrails are the gap between "works in a demo" and "works on Tuesday".

PUBLIC WRITE-UPS ON PIECES OF MOVE 3

An Expert Council for Claude - gabegiro.com/blog/expert-council-for-claude
Work Without Pull Requests - gabegiro.com/blog/work-without-pull-requests

MOVE 4 Rollout & measured uplift

Land it on one squad, prove the number, then make the gain compound instead of decaying.

- **Pilot on one squad** (a few weeks) - install the operating model + a first slice of internal tooling against the Move 1 baseline, and hit an explicit, measured uplift target.
- **Template it.** Turn what worked into the shared skill/hooks/rules library the next team inherits on day one.
- **Roll across teams** with internal champions and light governance - so quality and security hold as it scales, and gains compound instead of decaying back to ad-hoc.

Low-risk by design: the org sees a measured result on a single squad before committing to a department-wide rollout.

How we measure success

We do **not** measure motion. Lines of code, PR count, story points, and tokens spent all reward busywork and are trivially gamed. Instead we track delivered outcomes, flow, and quality as a **balanced set**, grounded in the industry-standard **DORA** and **SPACE** frameworks - the balance is the point: you can't win one metric by wrecking another.

The pilot runs on a small, un-gameable trio, held against each other:

- **A speed metric** - lead time to production for a defined class of work (idea/commit to running for users).
- **A quality guardrail** - change-failure rate stays flat or improves, so speed isn't bought with breakage.
- **A human signal** - developer-reported time-on-toil drops (and/or satisfaction rises) - the clearest "is this actually helping?" read.

North star, plainly: we shipped a defined class of work meaningfully faster, without more production breakage, and the team spends more time on real engineering and less on toil. Supporting metrics per context: review latency, deployment frequency, time-to-restore, defect/rework rate, and the share of agent-authored work that merges without a human rewrite.

Honest caveats: this needs a clean baseline (why Move 1 exists), some signals take a few weeks to read, and we attribute conservatively - one squad with a comparison, not a department-wide claim we can't defend.

Why this is where the 2x lives

Baseline so the gain is provable, an operating model that raises the floor, internal tooling that turns one-off wins into infrastructure every engineer draws on, and a measured rollout that makes it compound - that is the shift from **faster typing** to **more shipped**, and from ad-hoc individual wins to department throughput you can put a number on.

Where I'm coming from: I run this exact stack in production - a disciplined skill catalogue under a hard cap and router pattern, a two-layer memory system, purposeful multi-agent swarms (code review, migrations, builds), per-repo rule + hooks files, and unattended orchestration pipelines - on top of **13+ years of engineering leadership** (HBO GO at 8M users, Yahoo, early-stage).

If any of this maps to where your team is stuck, tell me which layer bites hardest and I'll point you at the part that moves the needle first for your setup.

Gabe Giro · AI Engineer gabegiro.com