KOTROV STUDIO · PLAYBOOK

The AI-Native Operating Model

A decision playbook for engineering leaders choosing what to change when AI arrives

Kotrov Studio — Kraków, Poland
kotrov.com ·

Contents

Why this exists

When AI tools land in an engineering org, two expensive moves get reached for. One restructures the people: pods, an Agent Champion on every team, a single owner for every task, a three-phase rollout. The other restructures the infrastructure: an orchestration platform, policy enforced before every tool call, an audit ledger of record. Both feel like progress. Both cost a quarter and a budget line. And on most teams that make them, the delivery number does not move.

The reason is arithmetic. IDC's 2024 research puts application development at 16% of a developer's time; the other 84% is requirements and tests, security, CI/CD, monitoring, and deployment. A tool that speeds up code generation makes the 16% faster and can make the 84% worse, by producing more code for the same review pipeline to absorb. Reorganizing the people and buying the platform are both aimed at the layer where the constraint usually isn't.

The gain on that 16% is also thinner than the business case assumes. METR's 2025 randomized trial put 16 experienced developers on 246 tasks in repositories they already knew, with frontier tools, and measured them 19% *slower* — after they had forecast a 24% speedup and while they still believed, afterward, that they had been sped up 20%. That is the population the optimistic case assumes benefits most, getting slower and unable to feel it.

This playbook is the alternative to the two expensive moves. It is a way to decide what to change, in what order, starting from evidence about your own system rather than a template. None of it requires a reorganization or a platform contract to begin. It sits above the tactical build — if you want the hands-on version of redesigning the agent loop, that is Making Engineering Agent-First. This one is about the decision that comes first.

The one idea to take away

If you read nothing else, take this: **AI made the 16% faster, but your delivery lives in the 84% around it — find your constraint before you restructure the org or buy the platform.**

Individual, task-level speed does not sum to organizational throughput on its own. The work released by faster generation moves downstream, into review and everything after it, and that is where delivery is actually decided. The cheap first move is to measure where a change spends its time between *written* and *released*, and rebuild that. The expensive, visible moves come last, if at all. Every module here traces back to that sentence.

First, the vocabulary

Four words carry the whole argument, and they get used as if they were interchangeable. They are not.

TERM	WHAT IT MEANS	WHY THE DISTINCTION MATTERS
Productivity	Individual, task-level speed — how fast one developer finishes one task	This is what most AI studies measure, and where the gains are real
Throughput	Organizational delivery — changes shipped, safely, per unit time	This is what the business cares about, and it moves for different reasons
Generation	Producing code	Cheap now, and getting cheaper
Delivery	Getting that code reviewed, verified, and safely released	The constraint, and the part AI tends to congest

Most AI business cases quietly promise the second column and cite evidence from the first. The gap between individual productivity and organizational throughput is where this playbook lives.

How to use this playbook

Each module follows the same shape, so you always know where you are:

- **The Challenge** — the decision the module resolves, and how to recognize it in your own org.
- **Prepare** — what to gather or settle before you decide. The part teams skip.
- **Set up & configure** — the framework, the numbers, and the worked call.
- **The outcome** — what changes when it works, and the failure mode to watch.

The modules are ordered so each depends only on the ones before it. Where a decision needs a hands-on build behind it, this playbook points to Making Engineering Agent-First, which is the tactical companion. Blank templates are at the back.

1

Module 1 — Measure before you move

THE CHALLENGE

The two expensive moves share one flaw: they assume they already know where the problem is. Reorganizing the people assumes the constraint is how the people are arranged. Buying the platform assumes it is missing infrastructure. Neither claim has been tested against your own delivery data, because that data was never pulled.

You'll recognize this if you can't say how long a change waits between *written* and *released*; if "AI adoption" or "percentage of code AI-generated" is a metric someone reports; or if the plan on the table is a restructure with no baseline it's meant to improve.

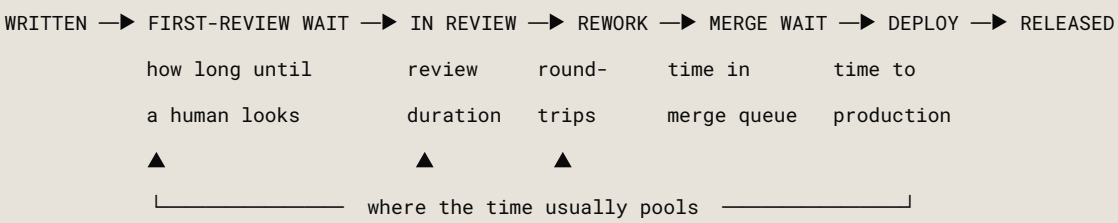
The fix is to measure the pipeline before you touch it. The constraint is usually not where the expensive move is aimed, and an afternoon in your own history will tell you where it actually is.

PREPARE

- **Pull the delivery history you already have.** Most of this is in git and your PR tool: when a change opened for review, when it got its first review, when it merged, when it deployed, and whether it came back. A week of export covers it.
- **Write your real written-to-released path.** The actual stages a change passes through, not the diagram on the wiki. You are going to time each one.
- **Pick flow time as the unit, not counts.** How long a change spends waiting and being worked at each stage. Counts (PRs, lines, adoption) hide the constraint; waiting time reveals it.

SET UP & CONFIGURE

Measure the time a change spends at each stage between written and released:



For each stage, get the median and the spread. The constraint is the stage where the time pools — for most teams that have added AI generation, it sits in the review band and the rework that

follows, because more generated code means more and larger changes arriving at a review step staffed by the same people. If that is where your time goes, then generation speed was never your bottleneck, and a reorganization aimed at generation speed cannot help.

Hold the measurement honestly. A rising count of merged pull requests is not throughput if those changes wait longer, come back more often, or ship more incidents. Measure the flow, not the activity.

THE OUTCOME

You can name your constraint, with a number, from your own system. That single fact decides everything downstream: whether the problem is generation (rare), review (common), or coordination and everything after it, and therefore whether either expensive move addresses it at all.

The failure mode to watch: measuring vanity. Adoption percentage, lines generated, and raw PR count all rise when AI arrives and tell you nothing about delivery. A team that optimizes those numbers is optimizing the inputs the next module shows you should be careful with.

2

Module 2 — Size the gain honestly

THE CHALLENGE

The business case for both expensive moves rests on a single large multiplier — a "2x," a flat "20–45% faster." The real evidence is a spread that cuts both ways and is almost entirely task-level, not delivery-level. Reorganizing a hundred people around the top of that spread is a costly way to discover it was the wrong number.

You'll recognize this if your plan cites one headline gain with no range; if expectations have been set at a round multiple; or if the assumption is that junior engineers, who adopt the tools most, benefit most.

The move is to separate what the evidence actually says by task and by seniority, and set expectations against the real distribution before you invest against it.

PREPARE

- **Separate task-level evidence from delivery-level evidence.** A faster task is not a faster release. Keep the two columns apart when you build the case.
- **Identify who the tools help and who they slow.** The effect is not uniform across your team, and the difference decides where the gain is real.

SET UP & CONFIGURE

The verifiable numbers, with the nuance the headline drops:

STUDY	WHAT IT FOUND	THE CATCH
McKinsey 2023 lab (40+ devs)	45–50% faster docs, 35–45% code-gen, 20–30% refactor	Under 10% on high-complexity work; junior devs 7–10% *slower*
Microsoft Research RCT (4,867 devs)	+26.08% completed PRs per week	Concentrated in junior devs; counts PRs, says nothing of quality
METR 2025 RCT (16 devs, 246 tasks)	19% *slower*, while feeling 20% faster	Experienced devs, familiar repos — the case's core assumption

Read across the row and the honest position is narrow: the gains are largest on simple work and for skilled developers, smallest or negative on complex work and for juniors, and in the one

randomized test on experienced engineers in their own code, negative overall. That is real, and it is worth having. It is not “2x, reorganize now.”

> **“I end up spending less time writing code. I spend more time babysitting the AI and reviewing what it is trying to do.”**
>
> — a developer quoted in DORA’s *2025 State of DevOps* report

The quote names the mechanism. The task got faster and the reviewing got heavier, which is why individual speed and organizational throughput come apart.

THE OUTCOME

You have an expectation that matches the evidence: where AI helps on your team, where it doesn’t, and how much. The business case rests on a range you can defend rather than a multiplier you borrowed.

The failure mode: treating a task-level number as a delivery-level promise. A 40% faster task inside a pipeline whose constraint is review produces roughly zero extra releases, and a plan sold on the 40% will be judged on the zero.

3

Module 3 — Redesign where the work moved

THE CHALLENGE

When generation gets cheap, the work does not disappear. It moves downstream into review and everything after it, and it gets bigger. The two expensive moves both skip this: renaming teams and buying an orchestrator each leave the review step exactly as it was. This is the redesign both gesture at and neither performs.

You'll recognize this if your review queue is growing faster than your team; if "approved" has quietly come to mean "nobody had time to look"; or if incidents and rework rose after adoption even as the PR count climbed.

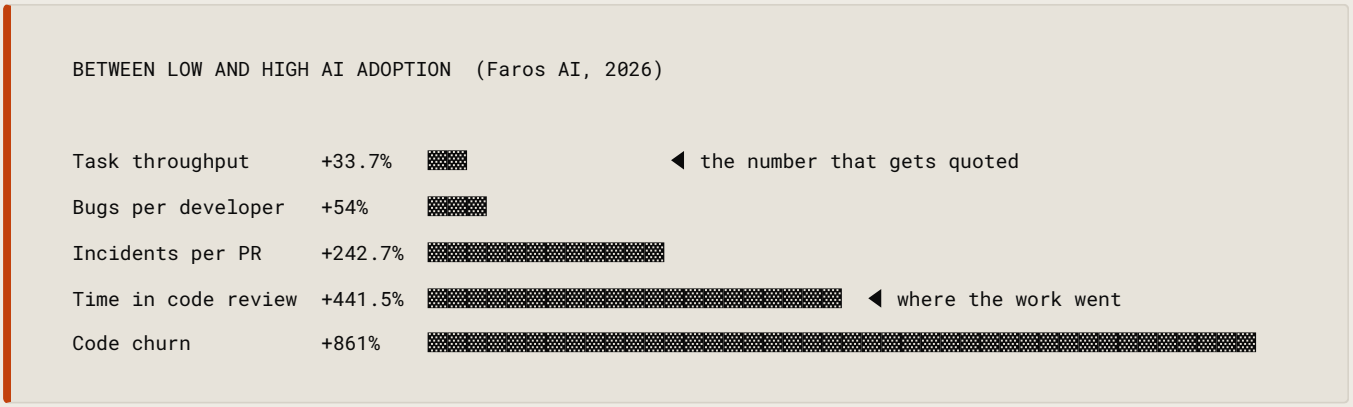
The move is to rebuild the constraint you found in Module 1 — almost always review — instead of the layers around it.

PREPARE

- **Confirm review is the constraint.** Module 1 gives you this. Redesign the stage the data names, not the one that is easiest to reorganize.
- **Decide who owns correctness for every merged change.** AI makes it trivial to produce code no single person owns the correctness of. Ownership is the discipline that survives that, and it predates AI by decades.

SET UP & CONFIGURE

The telemetry on what happens when review is left unredesigned is stark. Faros AI, comparing each organization's lowest and highest AI-adoption periods across 22,000 developers, found the shape of the problem:



Three moves rebuild the stage rather than scaling it:

- **Shrink what a reviewer has to hold in their head.** Automated checks catch the mechanical failures, so a human spends attention on intent while a machine handles syntax. If a person can't reconstruct why a change is correct, it wasn't reviewed, it was waved through.
- **Cap change size.** DORA's 2024 report found each 25% rise in AI adoption correlated with a 1.5% drop in throughput and a 7.2% drop in delivery stability, and traced it to batch size rather than worse code: AI makes large diffs effortless, and large diffs have always been riskier. Prefer many small, independently reviewable, independently revertible changes.
- **Make correctness owned.** Every merged change has one person accountable for why it's right. That is the ownership discipline, kept without the reorganization ritual around it.

The hands-on build behind this — a calibrated auto-approve gate that clears the safe changes and routes the rest to a human — is Making Engineering Agent-First, Module 4. This module is the decision to aim there.

THE OUTCOME

The constraint eases because you rebuilt it, not because you added people to it. Small, owned, well-checked changes move through review at a pace that does not depend on hiring more reviewers.

The failure mode: scaling the bottleneck instead of redesigning it — adding reviewers to a review step that is drowning in oversized, unowned changes. More people on an unchanged stage buys you a bigger version of the same jam.

4

Module 4 — Govern by autonomy level

THE CHALLENGE

The moment agents act rather than suggest, governance becomes the thing that decides whether they survive contact with production. Most teams apply one policy to all of them, locked down or fully trusted, and that uniformity is what gets agents pulled. Gartner predicts 40% of enterprises will decommission agents by 2027 for exactly this reason.

You'll recognize this if every agent runs under the same policy regardless of what it can touch; if the choices on offer are "blocked" or "trusted" with nothing between; or if you can't reconstruct what an agent actually did after the fact.

The move is to govern by what each agent is allowed to *do*, not by one blanket rule.

PREPARE

- **Inventory your agents by autonomy.** Sort each one into what it is permitted to do, from reading to acting on its own. The controls follow the level.
- **Separate ability to act from scope of access.** These are two different dials, and treating them as one is the root of the binary trap.

SET UP & CONFIGURE

Four autonomy levels, each with its own controls:

LEVEL	CAN DO	CONTROLS IT NEEDS
1 OBSERVE	reads, reports	read-only scope, audit log
2 ADVISE	proposes changes	no write access, human enacts
3 ACT w/ APPROVAL	executes on approval	scoped writes, human gate, audit
4 ACT AUTONOMOUS	executes unattended	narrow scope, blast-radius cap, replayable audit, kill switch

Access is scoped to the level. Policy is enforced OUTSIDE the model —
a jailbroken agent can only propose; the policy layer decides what runs.

Two engineering principles hold this up. First, the policy that decides what an agent may do must sit outside the model and not depend on the model's cooperation — a standard deny-by-default authorization pattern, argued cleanly in the academic MI9 framework. Second, the audit record is the primary operational log: you reconstruct what an agent could have done and what it did from the ledger, because you will need that the first time one does something you didn't predict.

And the blocker here is trust, not plumbing. LangChain's survey of 1,340 engineers found the top barrier to scaling agents is quality and trust in the output (32%), while most teams already run agents in production and already have observability. Durable execution and audit ledgers are the engineering answer to a trust problem — worth buying when you can state the problem, not before.

> **"Enterprises are treating AI agent governance as binary, either locked down or fully trusted, and that is the root cause of failure."**
>
> — Shiva Varma, Gartner

The reason the answer is structural rather than a better prompt is that agents are non-deterministic in a way traditional software is not.

> **"Traditional software used to be deterministic by nature. Given the same input, the same outcome follows. Agents introduce variability into systems built for repeatability."**
>
> — Arize, *Why AI Agents Break: A Field Analysis of Production Failures*

You cannot instruct your way out of non-determinism. You constrain what the agent can reach, enforce it outside the model, and keep a record you can replay.

THE OUTCOME

Agents stay in production because each one is governed to what it can actually do, and an agent that only observes is not carrying the controls meant for one that acts on its own. You can answer "what did this do" from the ledger.

The failure mode: binary governance. One policy for everything means you either lock agents down until they're useless or trust them until one deletes a production table. Both are the same mistake, and both end in decommissioning.

5

Module 5 — Restructure last, and only for ownership

THE CHALLENGE

The reorganization is the most visible move and the last one you should make. Renaming teams to pods and appointing a Champion changes the org chart, not the system — unless it changes who is accountable for the delivery number. Done for its own sake, it is cost dressed as transformation.

You'll recognize this if the plan adds roles that evangelize a tool; if a Champion's job is to drive adoption toward a code-generation-share target; or if there is a tidy three-phase timeline that would fit any transformation with the nouns swapped.

The move is to make structure serve delivery: change it only where it changes who owns the outcome, and never to optimize an input the delivery data says to be careful with.

PREPARE

- **Have your constraint in hand.** Modules 1 through 4 tell you what actually needs to change. Structure is the last lever, applied to support those changes, not the first.
- **Write down the delivery number a structural change is meant to move.** If you can't name it, the change is theater.

SET UP & CONFIGURE

Run every proposed structural change through one test:

```
THE OWNERSHIP TEST
─────────────────────────────────────────────────────────────────

For each proposed change (new role, pod, gate):

   Does this change WHO OWNS the delivery outcome?
      |
      ├── YES ──▶ it may be worth doing. Name the owner,
      |           name the number, make the change.
      |
      └── NO ───▶ it is a role or a ritual that adds
                  cost without changing accountability.
                  Cut it.
─────────────────────────────────────────────────────────────────
```

What survives the test is small: single, unambiguous ownership of an outcome, which is genuinely useful and predates AI by decades, because diffused accountability is where quality goes to die. What fails it is the reorganization ritual — the pod rename, the evangelist role, the phase plan — that leaves accountability exactly where it was.

None of this is a new failure mode. RAND's 2024 study of why AI projects fail, from 65 interviews with senior data scientists and engineers, found AI projects failing at roughly twice the rate of other IT work, with causes that were overwhelmingly organizational: misframing the problem, integration gaps, and a technology-first instinct that reaches for the newest thing when a simpler approach would do. That was written about pre-agent projects. The hype cycle didn't invent the failure; it inherited it. Structure applied to change ownership is the antidote; structure applied for its own sake is the disease.

THE OUTCOME

Your org chart reflects who owns what, and every structural change on it can be traced to a delivery number it was meant to move. The reorganization, when it happens, is small and load-bearing rather than large and decorative.

The failure mode: org-chart theater — adding a role or a pod that changes titles without changing who is accountable, or worse, appointing someone whose success metric is adoption share. Optimizing the percentage of AI-generated code, in an environment where churn and incidents rise with that percentage, is a structure pointed at the wrong number.

Start this week

You don’t need the whole model on day one. Three steps, smallest first:

1. Measure your constraint. Pull a week of delivery history, time each stage from written to released, and find where the time pools. You almost certainly have the data already.

2. Redesign the stage you found. If it’s review — and it usually is — shrink what a reviewer has to hold, cap change size, and give every merge one owner of correctness. Rebuild the constraint, don’t staff around it.

3. Inventory your agents by autonomy level. Sort each into observe, advise, act-with-approval, or act-autonomously, and set the controls to match. Enforce policy outside the model.

Start with step one. Everything else depends on knowing your constraint, and the rest compounds from there.

Templates

DELIVERY-CONSTRAINT AUDIT

STAGE	MEDIAN TIME	SPREAD (p90)	NOTES
First-review wait	____	____	____
In review	____	____	____
Rework round-trips	____	____	____
Merge-queue wait	____	____	____
Deploy to production	____	____	____

The constraint is the stage where time pools: _____

Generation speed is / is not our constraint: _____

REVIEW REDESIGN CHECKLIST

- ☐ Mechanical failures caught by automated checks (humans review intent)
- ☐ Change size capped — small, independently reviewable and revertible
- ☐ Every merged change has ONE named owner of correctness
- ☐ "Approved" means a human reconstructed why it is correct
- ☐ Review flow time measured before and after (not PR count)

CHANGE-SIZE POLICY

A change is review-sized (default) when:

- ☐ ~____ lines of real logic or fewer
- ☐ One reviewable concern, not a bundle
- ☐ Independently revertible / feature-flagged

Larger changes require: _____

Batch-size drop in stability is a DORA 2024 finding — hold this line.

AUTONOMY-LEVEL GOVERNANCE MATRIX

AGENT: _____ LEVEL: 1 2 3 4 (circle one)

☐ Access scoped to this level only (read-only for L1–L2)

☐ Human gate present for L3; blast-radius cap + kill switch for L4

☐ Policy enforced OUTSIDE the model (deny by default)

☐ Actions reconstructable from a replayable audit record

Owner of this agent's behavior: _____

About Kotrov

Kotrov is a hands-on engineering studio in Kraków. We work with B2B teams on the operating decisions behind AI-native engineering — finding where delivery actually stalls, rebuilding that, and governing agents by what they're allowed to do, without a reorganization or a platform contract you don't yet need.

If this playbook maps onto a problem you're carrying, we're glad to talk it through. No pitch — we map the situation, qualify both sides, and recommend the right shape of help.

Book a 30-minute discovery call · hello@kotrov.com · kotrov.com

Kraków, Poland ·