AGENT IS AI · AGENTIC WORKFLOWS

The Jira-to-Production Blueprint

Deploying an agentic delivery workflow that takes a requirement from ticket to production code — the architecture, the tools, the pitfalls, and how to pilot it safely.

ISSUE 01 · BIWEEKLY

Omar T Aslam — Founder, Agent is AI

AI & software delivery consultant. 25+ years architecting and delivering enterprise systems for banks, government and global brands. Builds agentic AI and full-stack delivery into complex, regulated organisations.

Contents

SUMMARY

Key findings

> *The tooling to run a requirement-to-production agentic workflow is real and in production in 2026. Whether it pays off is decided almost entirely by the architecture around the agent, not the model inside it.*

What this issue establishes

1. **The agent is the easy part.** Writing code is now a solved 15%. The value and the risk both live in the 85% around it: knowledge base, context, security gates, review and change control.
2. **Context decides everything.** Context-rich agents are markedly more accurate and cheaper to run than context-blind ones (Atlassian benchmarking: 44% more accurate, 48% fewer tokens). Architecting the knowledge base is Step 0, not an afterthought.
3. **Ticket quality is a production control.** A two-line ticket produces a two-line guess. A definition-of-ready is the highest-leverage change you can make, and it sits upstream of any tool.
4. **The gains are real but indirect.** Atlassian reports ~19% more merged pull requests per month for teams adopting its AI-native SDLC (37–51% on lower-activity repos), with 2–4 hours saved per developer per week — gains that come from removing administrative drag, not from typing faster. (Vendor-reported; treat as directional, and measure your own baseline.)
5. **Autonomy is not the metric.** One 2026 analysis found 75% of coding agents broke working code during CI runs. Pick the agent that fits your review capacity and respects repository boundaries — not the one with the biggest autonomy claim.

THE MARKET

The tool landscape in 2026

You don't need to build this stack from scratch — it already exists, and large engineering organisations are running it in production. It splits into three layers: the **orchestration and context layer** (where work and knowledge live), the **coding agent** (the harness that does the work), and the **delivery and governance layer** (git, CI/CD, security). A useful shorthand from the field: **Agent = Model + Harness**. The model supplies intelligence; the harness around it turns that into something you can trust in production. The harness is where the outcome is decided.

Orchestration & context

Tool	What it does	Where it fits
Jira	The system of record for work — epics, stories, acceptance criteria. Increasingly the orchestration anchor for agents.	The requirement enters and every commit traces back here.
Confluence	Specifications, standards, architecture docs — the written knowledge an agent must ground itself in.	A primary knowledge source; also the biggest source of stale-context risk.
Atlassian Rovo / Rovo Dev	Atlassian's AI layer and Teamwork Graph — unifies Jira, Confluence and 50+ connected tools into one context layer, with no-code agents.	The native orchestration + knowledge layer for Atlassian-centric orgs.
Figma	Design source of truth; connectable as context so agents can read intended UI/ flows.	Context input for front-end and product work.
MCP (Model Context Protocol)	The emerging standard for connecting an agent to tools and data sources beyond one vendor.	How you wire non-Atlassian systems into the context layer.

Coding agents (the harness)

Agent	Strength	Best fit
Claude Code	Top coding benchmark scores; deep repo context, terminal-first, MCP-native. Strong Fortune 500 enterprise adoption.	Complex, long-horizon work where reasoning and repo depth matter.
OpenAI Codex	Cloud VMs, parallel jobs, broad IDE support, strong governance. Millions of weekly users.	Async, high-throughput PR work in an OpenAI-standardised org.
GitHub Copilot	Deepest GitHub integration, SSO, audit logs. ~20M users; dominant in Fortune 100 via GitHub/Microsoft procurement.	Teams already centred on GitHub Enterprise.
Cursor	AI-native IDE, parallel sub-agents, strong editor UX.	Editor-first developers; multi-file interactive work.
Devin-class	Most hands-off — accepts a ticket, plans, opens a PR with minimal direction.	Tight, well-defined backlog tasks with a high review bar.

Delivery & governance

Layer	Role
GitHub / Bitbucket	Repository, pull requests, branch protection, CODEOWNERS.
GitHub Actions / Agentic CI/CD	Build, test and deploy pipeline; where security gates run before human review.
SAST / secret scanning / dependency review	Automated security gates — the checks that must pass before agent code reaches a person.

> *The tools are commoditised. The differentiator is how you assemble and govern them — identity, repository access, model routing, sandboxes, review, analytics, compliance. Enterprises don't buy an agent; they buy a system — identity, access, review,*

> *analytics and compliance — that hundreds of people can operate safely.*

THE PATTERN

The workflow, in four stages

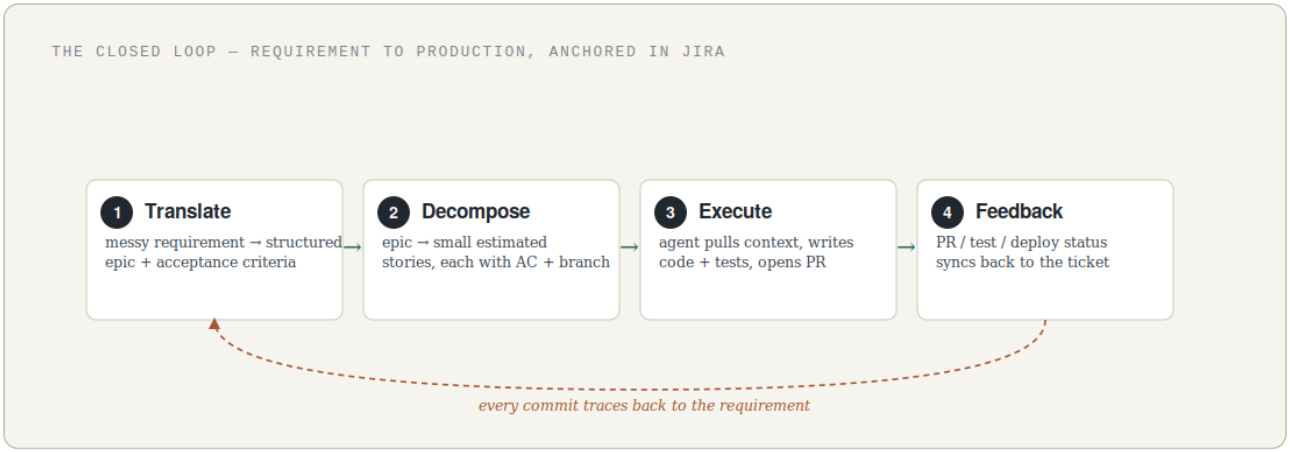
A closed loop anchored in Jira, so every commit traces back to a requirement. Not "point an agent at your backlog" — four deliberate stages:

1 · Translate — an agent turns unstructured requirements (a PM conversation, a rough brief) into a structured epic with real acceptance criteria and edge cases.

2 · Decompose — the epic is broken into small, estimated stories, each with explicit acceptance criteria and a branch.

3 · Execute — a coding agent pulls full context from the graph, writes code and tests, and opens a pull request.

4 · Feedback — PR outcomes, test results and deploy status sync back to update the ticket. The loop closes.



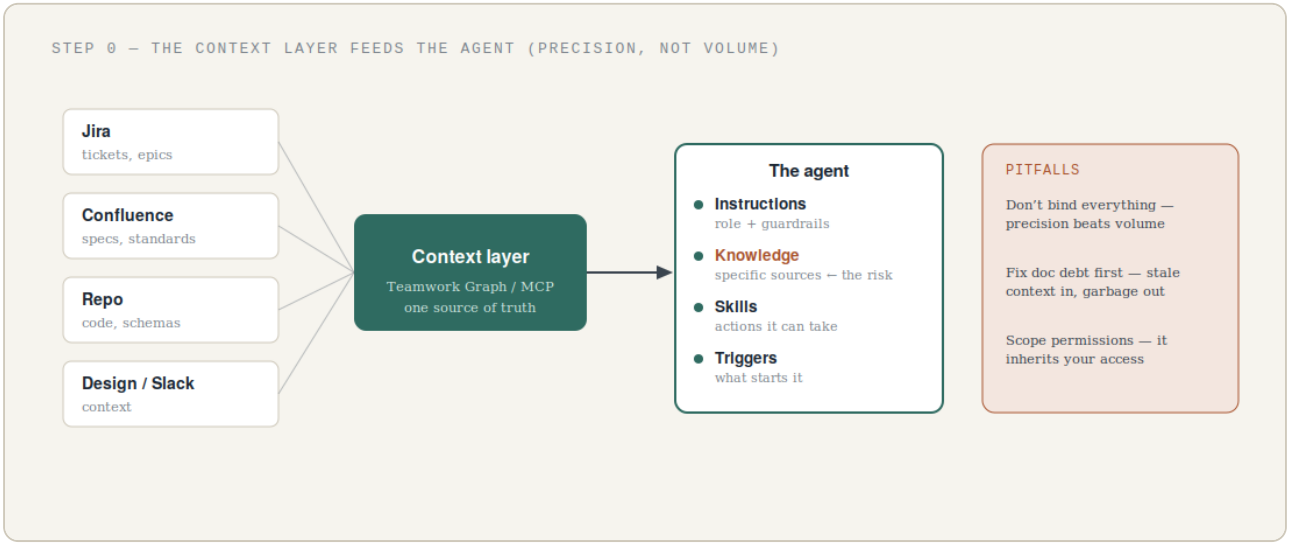
The closed loop — requirement to production, anchored in Jira.

STEP 0

Architect the knowledge base first

This is the step most guides skip, and the one that decides everything. An agent is only as good as the context it can reach — context-rich agents are markedly more accurate and cheaper to run than context-blind ones (Atlassian's internal benchmarking put this at 44% more accurate on 48% fewer tokens). Getting this right is the actual work.

The context layer (Atlassian's Teamwork Graph, or your equivalent via MCP connectors) links Jira, Confluence, the repo and design docs into one queryable source of truth. But connecting everything is not the same as architecting it. An agent is built from four things: **Instructions** (its role and guardrails), **Knowledge** (the specific sources it may draw on), **Skills** (the actions it can take) and **Triggers** (what starts it).



Sources feed one context layer, which feeds a scoped agent. The risk lives in "Knowledge".

The pitfalls all live in "Knowledge"

- **Don't point it at everything.** Binding the whole Confluence estate floods the agent with stale and contradictory pages. Bind specific spaces and pages per task — precision beats volume.
- **Fix your documentation debt first.** The agent will faithfully reproduce your out-of-date architecture page. Garbage context in, confident garbage out.
- **Scope permissions deliberately.** The agent inherits whatever access you grant it. Too broad and it reaches data it shouldn't; too narrow and it can't see what it needs. A security decision, not a config afterthought.

STEP 0.5

Build the pilot repo properly

You do not point an agent at your production monorepo and hope. You build a controlled environment first — and testing the *whole pipeline*, not just the agent, is the point.

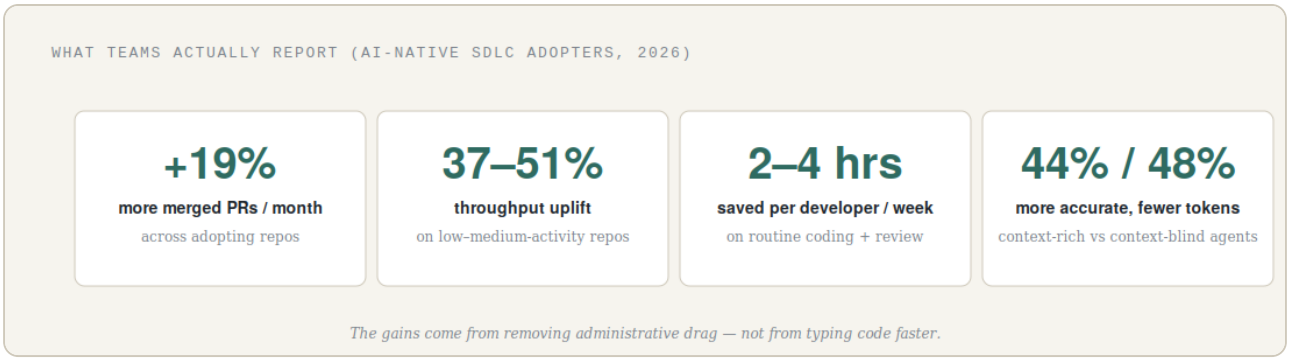
- **Clone into an isolated sandbox** — a fork or dedicated repo the agent operates in, with no path to production and no live credentials.
- **Seed it with real context** — representative code, the actual coding standards, schemas and a dependency manifest. A sandbox with no context tests nothing useful.
- **Wire the gates from day one** — branch protection, required review, CODEOWNERS and CI security scans, exactly as production would enforce them.
- **Use synthetic or masked data.** The agent should never see real customer or production data during a pilot.

Skip this and one of two things happens: the agent touches something it shouldn't, or it works against a toy repo and tells you nothing about real behaviour.

THE EVIDENCE

The numbers: where the value is

The instinct is to measure code-generation speed. Wrong meter. The friction was never typing; it's vague requirements, missing acceptance criteria and fragmented context. Remove that and throughput follows.



Reported outcomes from AI-native SDLC adopters, 2026.

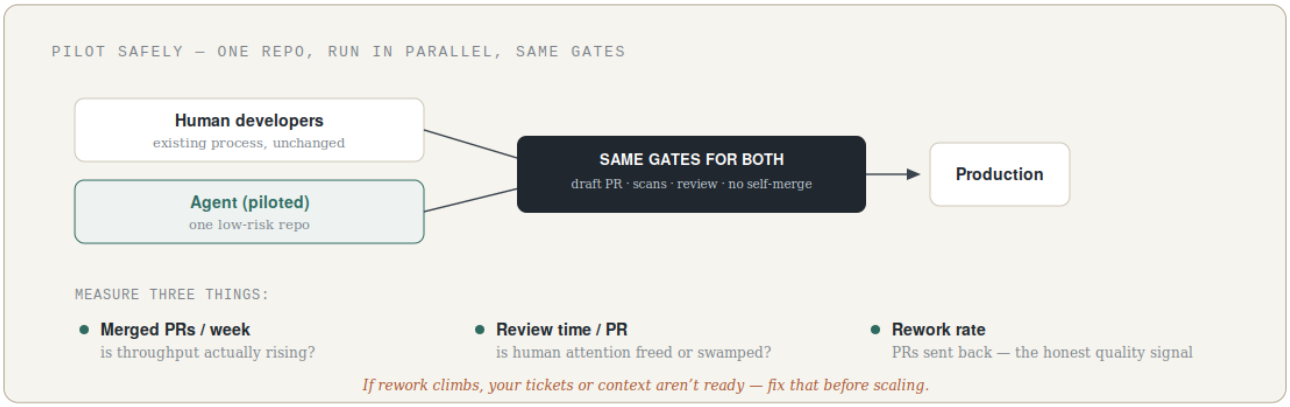
Two cautions sit alongside the upside. First, autonomy is not free: one 2026 analysis found **75% of coding agents broke working code during CI runs** — which is exactly why the gates in the next section are non-negotiable. Second, you cannot manage what you don't measure: adopting AI delivery without tracking throughput, change-failure rate and lead time makes it impossible to tell whether you're creating value or accelerating instability.

DEPLOYMENT

How to pilot it safely

Don't roll this across the org. Run it as a controlled pilot.

- **One team, one repo.** A low-to-medium-activity codebase — biggest uplift, smallest blast radius.
- **Run in parallel.** Agent PRs go through the *same* review and CI gates as human ones. No self-merge, ever.
- **Enforce a definition-of-ready** before any ticket reaches the agent.
- **Measure three things:** merged PRs per week, review time per PR, and rework rate. If rework climbs, your tickets or context aren't ready — fix that before scaling.



Pilot in parallel; both lanes pass the same gates; measure the three signals that matter.

THE BOTTOM LINE

What this means for you

Two things are true at once. This is no longer experimental — large engineering organisations are running requirement-to-production agentic workflows in production today. And most first attempts fail, not quietly but expensively: an agent given poor context and weak gates ships broken code faster than a human would, turning speed into rework and incidents.

So the decision isn't *whether* to adopt this. It's whether you build the unglamorous foundation before you turn it on. Everything that determines the outcome sits in four places:

- **The knowledge base** — scoped, current context, or the agent hallucinates confidently.
- **Ticket quality** — a definition-of-ready, or you automate the production of wrong answers.
- **The gates** — automated security checks and mandatory human review, or bad code merges at machine speed.
- **Measurement** — baseline your throughput, change-failure rate and rework, or you cannot tell whether this is helping or hurting.

Get those right and the reported gains — fewer stalled tickets, faster merges, hours back each week — are real and repeatable. Get them wrong and you have paid for a faster way to create technical debt.

> *The agent writes the code in minutes. Whether that code should ever reach a customer is decided by the four things above — and none of them are the agent.*

Building and running agentic delivery inside real organisations — with the legacy systems, security and compliance that come with them — is what we do at Agent is AI.

Omar T Aslam · Agent is AI · agentisai.co.uk

THE AUTHOR

Omar T Aslam

Founder, Agent is AI

Omar has spent 25+ years architecting and delivering enterprise systems for banks, government and global brands — the kind of complex, regulated environments where "it worked in the demo" and "it runs in production" are very different claims.

Agent is AI is his consultancy for exactly that gap: designing and delivering agentic AI and full-stack software into real organisations, with the access model, security gates, governance and change control that make it safe to run. The firm fronts engagements with senior principals and scales with on-demand teams of AI and software specialists.

WEB agentisai.co.uk

MAIL hello@agentisai.co.uk

IN linkedin.com/in/omar-aslam

Scan to connect →