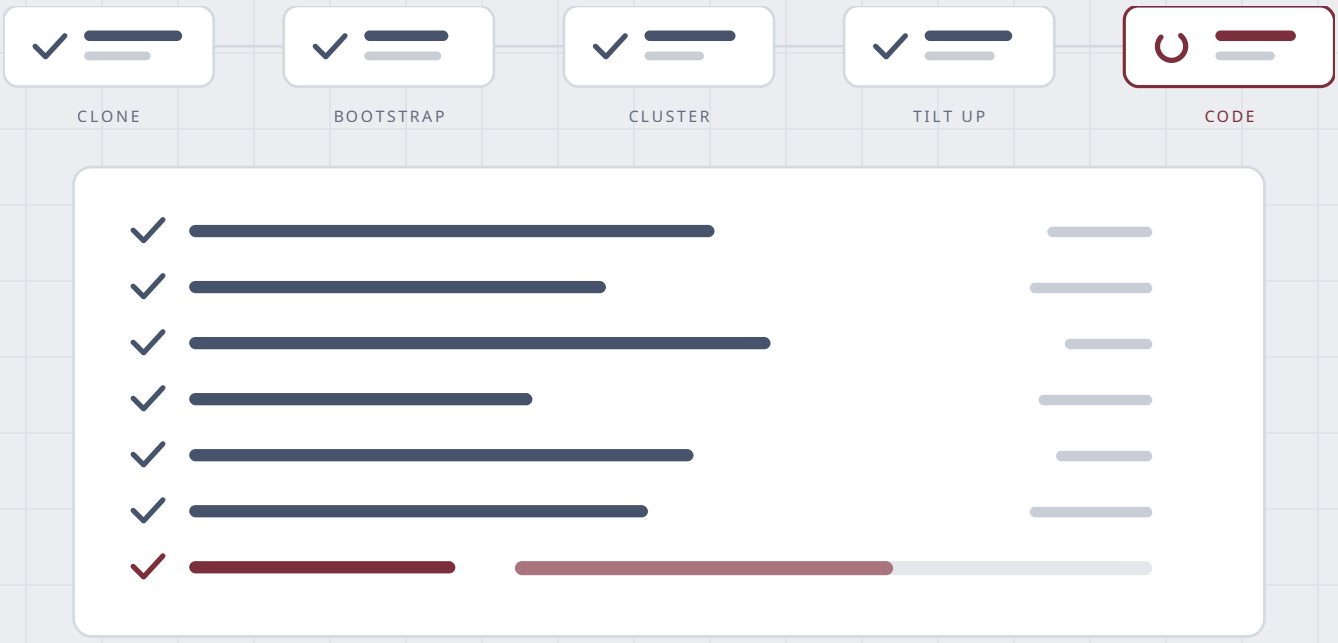
– // DEVELOPER EXPERIENCE · SUSTAIND

A codebase where AI agents engineer *end to end*.

FROM SEVEN REPOSITORIES AND FIVE DEPLOYMENT PATHS TO ONE AGENT-READY MONOREPO · NX / TILT / KIND / PNPM / UV

AI coding agents are only as good as the codebase they work in. When sustaind consolidated its fast-grown landscape — six service repositories plus a serverless data platform — into one monorepo, the goal was developer experience. The result reaches further: one working tree with full context, conventions an agent can execute, a scoped test loop, and a local cluster to verify against. Humans get a running product in three commands; agents engineer features end to end. I architected and drove that consolidation.



01

Agent-Ready by Design

full context · executable conventions

02

One Command, Full Stack

make setup · dev-cluster · tilt up

03

Feedback in Seconds

live sync · HMR · auto-restarts

04

Only the Providers We Need

AWS · Auth0 · Sentry

01 CONTEXT

Architecture that grew as fast as the product.

sustaind ships a regulatory-compliance platform — law applicability, product compliance, safety data sheets — and it shipped it fast. The architecture kept pace the way healthy startups do: a new repository per service, and the best available provider for every job. By early 2026 the product spanned a NestJS core, two web applications, an upload API on Bun, two Python workers and a serverless data platform — seven repositories, each doing its job well.

The cost of that speed surfaced somewhere else: in the day-to-day experience of the developers building on top of it — and in what the codebase could offer the AI coding agents joining the team. My engagement: architect and drive the consolidation into a single monorepo, and make the developer experience a first-class product for both.

02 THE STARTING POINT

Seven good services, one hard day-to-day.

None of this was bad engineering — it was the natural residue of speed. Added up, though, the friction was real:

/01 **Seven repositories, one product**

A feature that touched upload, processing and UI meant coordinated branches and pull requests across up to five repositories.

/02 **A cloud-hosted inner loop**

The data platform developed against serverless cloud infrastructure — powerful in production, but every save-and-verify cycle ran through the cloud instead of a local process.

/03 **Roughly 300 environment variables**

Local wiring lived in seven env templates. Assembling a working set — hostnames, ports, credentials — was the real onboarding task.

/04 **A toolchain per repository**

Three pnpm versions, Bun, Poetry and pip across the codebase; two Python versions. Every context switch carried a small setup tax.

/05 **Shared code by publish-and-pin**

The API client was published to npm and pinned per consumer, so versions drifted apart. The Python worker framework existed twice and evolved separately.

/06 **Five paths to production**

SST, Amplify, container pushes, serverless deploys and scheduled cloud runs — each correct on its own, each one more thing to know.

> **The bar we set:** the full product must boot from committed defaults on any laptop, real credentials only where a feature genuinely needs them, the migration landing service by service without pausing the roadmap — and every convention executable, so an AI agent can follow it as reliably as a human.

"Onboarding is a *product*.
Its users are your engineers — human and AI."

03 THE MOVE

One workspace, two lockfiles, every service.

The services moved into a single Nx monorepo: a pnpm workspace for TypeScript, a uv workspace for Python — five applications plus a shared API client that is generated from the core API's OpenAPI spec and built inside the workspace. No publishing, no version pinning, no drift. Tool versions are pinned once and apply everywhere: Node 24, pnpm 10.18.3, Python 3.11, Biome for TypeScript, Ruff for Python.

Providers were consolidated by the same principle — keep what the product needs, integrate it once. LLM workloads standardized on AWS Bedrock (an OpenAI model is now simply a model ID there), storage on S3, identity on Auth0, monitoring on Sentry. Fewer providers means fewer credentials to hand a new developer and fewer SDKs to keep patched.

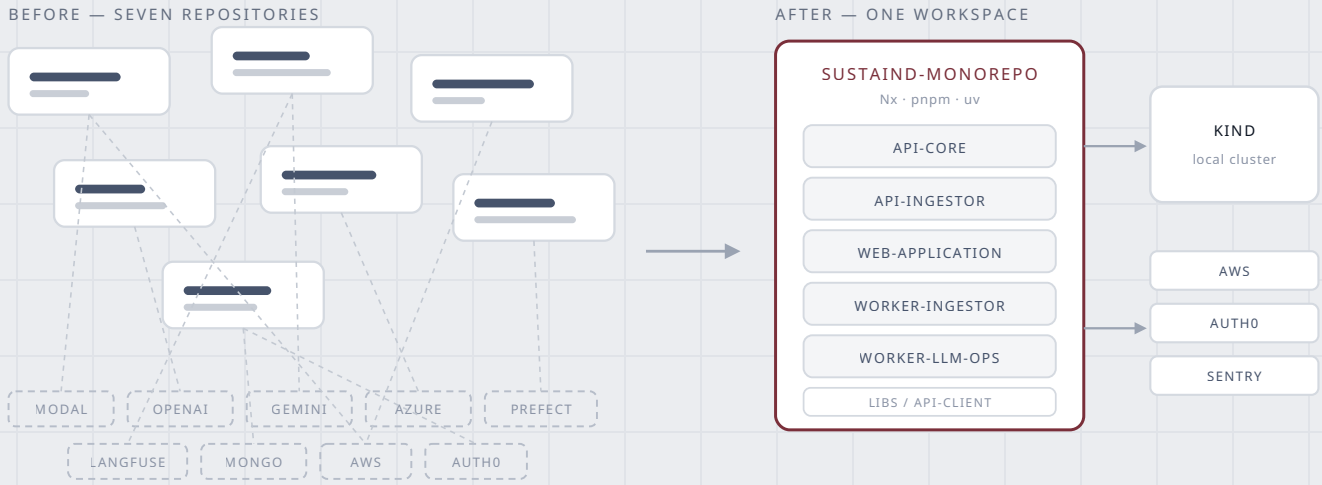


FIG. 01 — CONSOLIDATION: SEVEN REPOSITORIES AND A WIDE PROVIDER SURFACE BECOME ONE WORKSPACE, ONE LOCAL CLUSTER, THREE CORE PROVIDERS.

04 A LOCAL CLOUD

A production-shaped cluster on every laptop.

`tilt up` boots the entire product on a local Kubernetes cluster: Traefik as ingress, Postgres with pgvector, Redis, and MinIO standing in for S3. Every service gets a stable ***.localhost** address — browsers resolve these to the local machine by standard, so there is nothing to edit and no port list to memorize.

The cluster initializes itself: databases and buckets are created on first boot, schema migrations run as init containers — a pod that reports Ready is already on the latest schema — and an idempotent seeder fills the system once. For AI features, Bedrock authenticates through the developer's own AWS SSO session mounted from the host; a local llama.cpp server covers development at zero cloud spend.

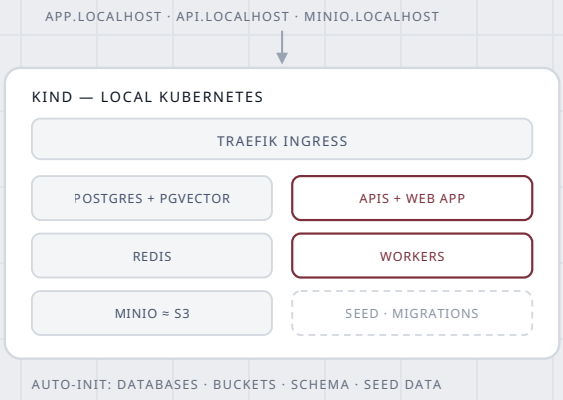


FIG. 02 — EVERYTHING THE PRODUCT NEEDS, INSIDE ONE LOCAL CLUSTER.

05 FAST FEEDBACK

Save the file — the cluster is already running it.

Builds follow a three-layer model. Base runner images hold each runtime's dependencies and rebuild only when a lockfile changes. App images add nothing but source and configuration — no install steps at all. While you work, Tilt syncs changed files straight into the running containers: Python and Node processes restart automatically, the Bun API reloads in-process, and the web application hot-reloads through Vite.

The same machinery keeps types honest across the stack: saving a change in the core API exports its OpenAPI spec, the shared client rebuilds from it, and the web application picks up the new types — one cascade, no manual publishing step anywhere.

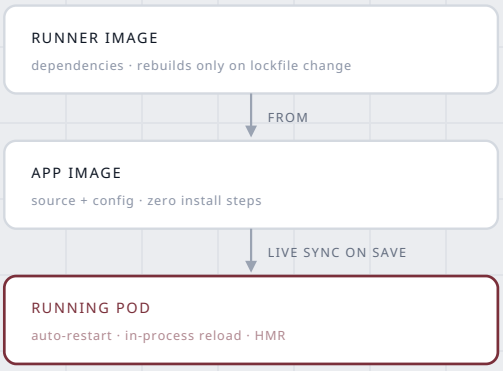


FIG. 03 — THREE-LAYER BUILD: DEPENDENCIES, SOURCE, RUNNING PROCESS.

Change an API endpoint, and the frontend's types have updated before you've switched windows.

06 CONVENTION OVER CONFIGURATION

Defaults that work; overrides that win.

Every app's runtime configuration is assembled from two layers: a committed template that boots a fresh environment as-is, and an optional gitignored override file for anything personal — a real Auth0 credential, an AWS profile, a Sentry DSN. Tilt merges the two into Kubernetes Secrets and regenerates them the moment either file is saved.

Conventions carry the rest: fixed host ports for database tools, one ***.localhost** hostname per service, a seeder that can be re-triggered from the Tilt UI, and a `make` target for every routine task. The bootstrap script is idempotent and cross-platform — it installs the complete toolchain on macOS or Ubuntu and can safely be re-run at any time.

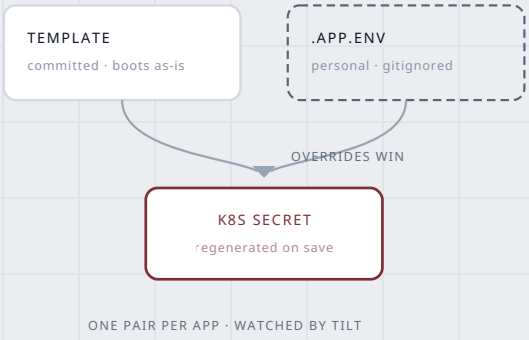


FIG. 04 — RUNTIME CONFIG: COMMITTED DEFAULTS, PERSONAL OVERRIDES.

The payoff: agents that engineer end to end.

Everything on the preceding pages compounds here. One working tree gives an agent the entire product as context; executable conventions tell it how the system runs; the affected pipeline and the local cluster give it a feedback loop it can drive itself. An agent is a developer with perfect patience and zero tribal knowledge — and a codebase shaped like this lets it carry real engineering work, not just autocomplete.

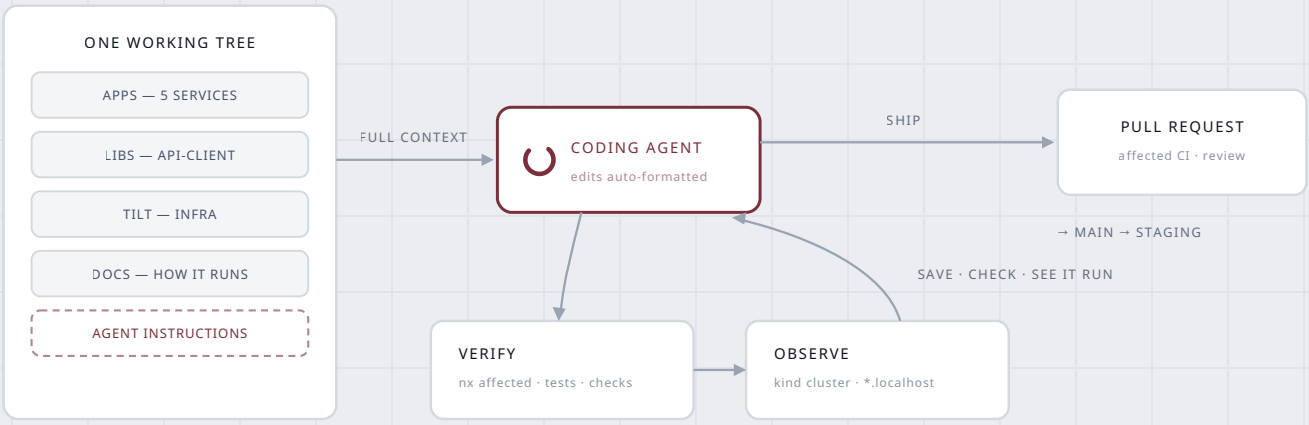


FIG. 05 — THE AGENT LOOP: FULL CONTEXT IN, AFFECTED CHECKS AND A RUNNING PRODUCT TO VERIFY AGAINST — SHIPPED THROUGH THE SAME PULL-REQUEST PATH AS EVERYONE ELSE.

/01 **One tree, whole product**

A single working tree holds every service, the shared client, the infrastructure manifests and the docs. An agent's context is the system itself — a cross-service feature is one branch and one pull request, not a coordination exercise across repositories.

/02 **Context that is written down**

The repository ships its own agent instructions, and the README chain describes the system the way it actually runs — kept accurate by convention. Formatting hooks run after every edit, so an agent's diffs are canonical by construction.

/03 **A verification loop it can drive**

Nx affected scopes tests, lint and type-checks to what a change touches — a fast, deterministic signal. And because the whole product boots locally, an agent can go beyond static checks: start the cluster, call the *.localhost endpoints, watch the logs.

/04 **The same path as everyone else**

Agent work ships like human work: a branch, a pull request, the affected pipeline, review. No special lanes and no bypasses — the conventions that make the fast path safe for people make it safe for agents.

That makes end-to-end agentic engineering a working mode, not a demo: **an agent can take a feature from the API endpoint through the typed client and UI to verified-in-the-running-product — one repository, one branch, the same three commands a human uses.**

08 ONE PATH TO PRODUCTION

The same conventions carry into CI.

Continuous integration runs **Nx affected**: only the projects a change actually touches are linted, checked, tested and built. A daily Trivy scan gates high-severity CVEs behind an expiring allow-list. Runner images publish only when their content hash is new, and app images are built — or retagged unchanged — into a commit-keyed deploy manifest.

Every merge to main deploys itself to staging. Production is a SemVer tag behind a four-eyes approval, with an auto-generated changelog. Where five deployment mechanisms each needed their own expertise, there is now one path — and everyone on the team knows it.

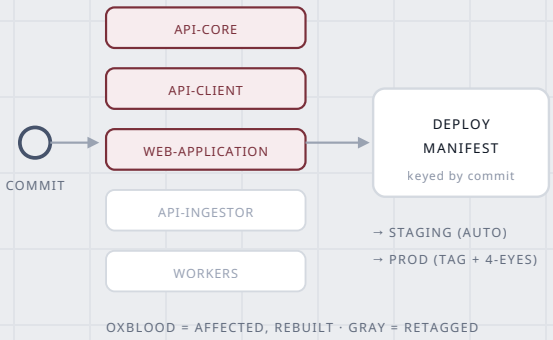


FIG. 06 — AFFECTED-ONLY CI WITH A COMMIT-KEYED DEPLOY MANIFEST.

09 OUTCOME

One repository, three commands — for human and agent alike.

01	03	01	02
MONOREPO — SEVEN REPOSITORIES CONSOLIDATED	COMMANDS FROM CLONE TO A RUNNING PRODUCT	PATH TO PRODUCTION — DOWN FROM FIVE	LOCKFILES FOR THE ENTIRE CODEBASE

The migration landed service by service over the following weeks — the roadmap never paused — and each legacy repository was archived as its successor went live. Today a new developer clones one repository, runs three commands, and has the entire product running on a local cluster with live updates. Setup knowledge that once lived in seven READMEs is now executable convention — which is why the same repository is where AI coding agents work today: end to end, through the same pull requests as everyone else.

STACK — Nx · pnpm · uv · Tilt · kind · ctlptl · Kubernetes · Traefik · Docker · Biome · Ruff · GitHub Actions · Trivy · SST · AWS · Auth0 · Sentry

Is your codebase ready for AI agents?

I design and build developer platforms — monorepos, local cloud environments, build systems and CI/CD that make the fast path the default — for human teams and the agents working alongside them.

START A PROJECT ↗

hello@bastian-schoentag.de