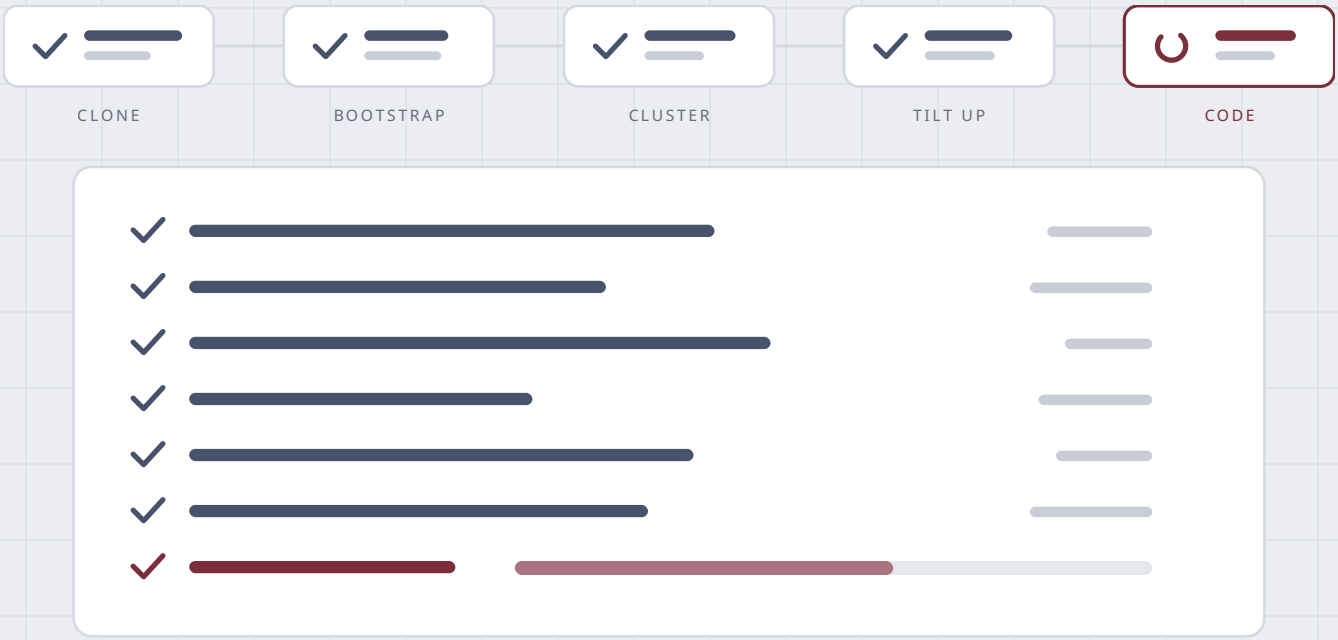
— // DEVELOPER EXPERIENCE · SUSTAIND

Eine Codebasis, in der KI-Agenten *Ende-zu-Ende* entwickeln.

VON SIEBEN REPOSITORIES UND FÜNF DEPLOYMENT-WEGEN ZU EINEM AGENT-READY MONOREPO · NX / TILT / KIND / PNPM / UV

KI-Coding-Agenten sind nur so gut wie die Codebasis, in der sie arbeiten. Als sustaind seine schnell gewachsene Landschaft — sechs Service-Repositories plus eine Serverless-Datenplattform — in ein Monorepo konsolidierte, war das Ziel Developer Experience. Das Ergebnis reicht weiter: ein Arbeitsverzeichnis mit vollem Kontext, Konventionen, die ein Agent ausführen kann, eine fokussierte Testschleife und ein lokaler Cluster zur Verifikation. Menschen bringen das Produkt mit drei Befehlen zum Laufen; Agenten entwickeln darin Features Ende-zu-Ende. Ich habe diese Konsolidierung entworfen und umgesetzt.



01

Agent-Ready by Design

voller Kontext · ausführbare Konventionen

02

Ein Befehl, der ganze Stack

make setup · dev-cluster · tilt up

03

Feedback in Sekunden

Live-Sync · HMR · Auto-Restarts

04

Nur die nötigen Anbieter

AWS · Auth0 · Sentry

01 KONTEXT

Eine Architektur, die so schnell wuchs wie das Produkt.

sustaind entwickelt eine Plattform für regulatorische Compliance — Gesetzesanwendbarkeit, Produkt-Compliance, Sicherheitsdatenblätter — und hat sie in hohem Tempo ausgebaut. Die Architektur hielt Schritt, wie in gesunden Startups üblich: ein neues Repository pro Service, für jede Aufgabe der beste verfügbare Anbieter. Anfang 2026: ein NestJS-Core, zwei Web-Anwendungen, eine Upload-API auf Bun, zwei Python-Worker, eine Serverless-Datenplattform — sieben Repositories mit jeweils klarem Zweck.

Der Preis dieses Tempos zeigte sich an anderer Stelle: im Arbeitsalltag der Entwickler — und in der Frage, wie gut KI-Coding-Agenten mit dieser Codebasis arbeiten können. Mein Auftrag: die Konsolidierung in ein einzelnes Monorepo zu entwerfen und voranzutreiben — und die Developer Experience zu einem eigenen Produkt zu machen, für Mensch und Agent.

02 DIE AUSGANGSLAGE

Sieben gute Services, ein anstrengender Alltag.

Nichts davon war schlechtes Engineering — es war das natürliche Nebenprodukt von Geschwindigkeit. Zusammengenommen war die Reibung trotzdem real:

/01 **Sieben Repositories, ein Produkt**

Ein Feature, das Upload, Verarbeitung und UI berührte, bedeutete koordinierte Branches und Pull Requests in bis zu fünf Repositories.

/02 **Eine Entwicklungsschleife in der Cloud**

Die Datenplattform wurde direkt gegen Serverless-Infrastruktur entwickelt — stark in der Produktion, aber jeder Zyklus aus Speichern und Prüfen lief durch die Cloud statt durch einen lokalen Prozess.

/03 **Rund 300 Umgebungsvariablen**

Die lokale Verdrahtung lebte in sieben Env-Templates. Eine funktionierende Konfiguration zusammenzustellen — Hostnamen, Ports, Zugangsdaten — war die eigentliche Onboarding-Aufgabe.

/04 **Eine Toolchain pro Repository**

Drei pnpm-Versionen, Bun, Poetry und pip im Einsatz; zwei Python-Versionen. Jeder Kontextwechsel brachte eigenen Einrichtungsaufwand mit.

/05 **Geteilter Code per Publish-and-Pin**

Der API-Client wurde auf npm veröffentlicht und pro Consumer gepinnt — die Versionen liefen auseinander. Das Python-Worker-Framework existierte doppelt und entwickelte sich getrennt weiter.

/06 **Fünf Wege in die Produktion**

SST, Amplify, Container-Pushes, Serverless-Deploys und geplante Cloud-Läufe — jeder für sich korrekt, jeder ein weiteres Detail, das man kennen musste.

> **Die Messlatte:** Das gesamte Produkt startet auf jedem Laptop aus committeten Defaults; echte Zugangsdaten nur dort, wo ein Feature sie wirklich braucht; die Migration landet Service für Service, ohne die Roadmap anzuhalten; und jede Konvention ist ausführbar — damit ein KI-Agent ihr so verlässlich folgen kann wie ein Mensch.

„Onboarding ist ein *Produkt*.
Seine Nutzer sind Entwickler — Mensch wie KI.“

03 DER UMZUG

Ein Workspace, zwei Lockfiles, jeder Service.

Die Services zogen in ein einzelnes Nx-Monorepo: ein pnpm-Workspace für TypeScript, ein uv-Workspace für Python — fünf Anwendungen plus ein gemeinsamer API-Client, der aus der OpenAPI-Spezifikation der Core-API generiert und im Workspace selbst gebaut wird. Kein Publishing, kein Versions-Pinning, kein Drift. Tool-Versionen werden einmal festgelegt und gelten überall: Node 24, pnpm 10.18.3, Python 3.11, Biome für TypeScript, Ruff für Python.

Die Anbieter wurden nach demselben Prinzip konsolidiert: behalten, was das Produkt braucht — und das einmal sauber integrieren. LLM-Workloads laufen standardisiert über AWS Bedrock (ein OpenAI-Modell ist dort schlicht eine Modell-ID), Storage über S3, Identität über Auth0, Monitoring über Sentry. Weniger Anbieter bedeuten weniger Zugangsdaten für neue Entwickler und weniger SDKs, die aktuell gehalten werden müssen.

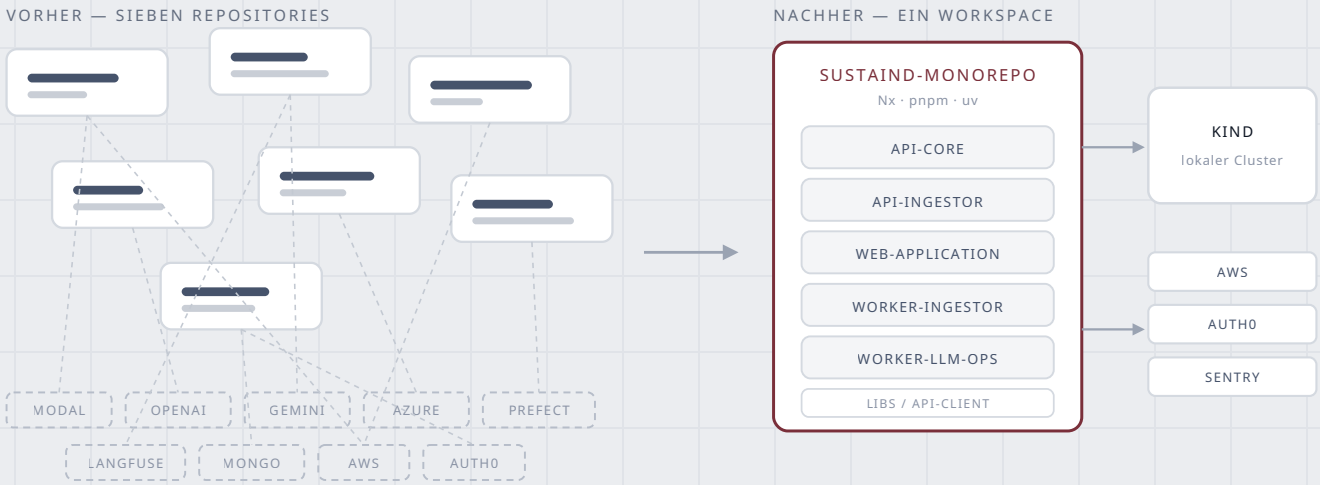


ABB. 01 — KONSOLIDIERUNG: SIEBEN REPOSITORIES UND EINE BREITE ANBIETERLANDSCHAFT WERDEN ZU EINEM WORKSPACE, EINEM LOKALEN CLUSTER UND DREI KERNANBIETERN.

04 EINE LOKALE CLOUD

Ein produktionsnaher Cluster auf jedem Laptop.

`tilt up` startet das gesamte Produkt auf einem lokalen Kubernetes-Cluster: Traefik als Ingress, Postgres mit pgvector, Redis und MinIO als S3-Ersatz. Jeder Service bekommt eine stabile ***.localhost**-Adresse — Browser lösen sie standardkonform auf die lokale Maschine auf; nichts muss angepasst, keine Portliste gemerkt werden.

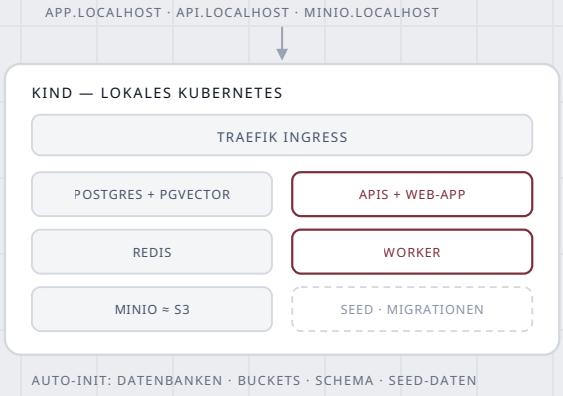


ABB. 02 — ALLES, WAS DAS PRODUKT BRAUCHT, IN EINEM LOKALEN CLUSTER.

Der Cluster initialisiert sich selbst: Datenbanken und Buckets entstehen beim ersten Start, Schema-Migrationen laufen als Init-Container — ein Pod, der Ready meldet, ist bereits auf dem neuesten Schema — und ein idempotenter Seeder befüllt das System einmalig. Für KI-Features authentifiziert sich Bedrock über die eigene AWS-SSO-Session des Entwicklers, vom Host eingebunden; ein lokaler llama.cpp-Server deckt die Entwicklung ohne Cloud-Kosten ab.

05 SCHNELLES FEEDBACK

Datei speichern — der Cluster führt sie schon aus.

Builds folgen einem Drei-Schichten-Modell. Basis-Runner-Images enthalten die Abhängigkeiten und werden nur neu gebaut, wenn sich ein Lockfile ändert. App-Images ergänzen nur Quellcode und Konfiguration — keine Installationsschritte. Während der Arbeit synchronisiert Tilt geänderte Dateien direkt in die laufenden Container: Python- und Node-Prozesse starten automatisch neu, die Bun-API lädt im Prozess nach, die Web-Anwendung aktualisiert sich per Vite-HMR.

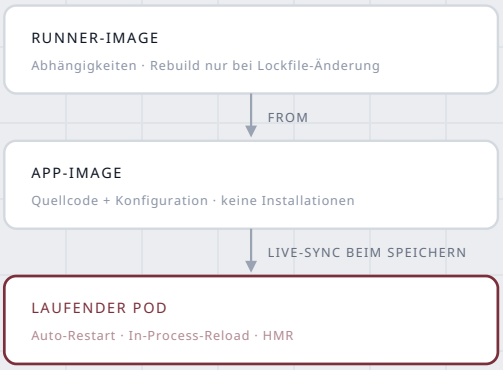


ABB. 03 — DREI-SCHICHTEN-BUILD: ABHÄNGIGKEITEN, QUELLCODE, LAUFENDER PROZESS.

Dieselbe Maschinerie hält die Typen über den gesamten Stack konsistent: Beim Speichern einer Änderung an der Core-API wird die OpenAPI-Spezifikation exportiert, der gemeinsame Client daraus neu gebaut — und die Web-Anwendung übernimmt die neuen Typen. Eine Kaskade, ohne einen einzigen manuellen Publishing-Schritt.

Einen API-Endpunkt ändern — und die Typen im Frontend sind aktualisiert, bevor man das Fenster gewechselt hat.

06 KONVENTION STATT KONFIGURATION

Defaults, die funktionieren; Overrides, die gewinnen.

Die Laufzeitkonfiguration jeder App entsteht aus zwei Schichten: einem committeten Template, mit dem eine frische Umgebung unverändert startet, und einer optionalen, gitignorierten Override-Datei für alles Persönliche — echte Auth0-Zugangsdaten, ein AWS-Profil, eine Sentry-DSN. Tilt führt beide zu Kubernetes Secrets zusammen und erzeugt sie neu, sobald eine der Dateien gespeichert wird.

ABB. 04 — LAUFZEITKONFIGURATION: COMMITTETE DEFAULTS, PERSÖNLICHE OVERRIDES.

Den Rest tragen Konventionen: feste Host-Ports für Datenbank-Tools, ein ***.localhost**-Hostname pro Service, ein Seeder, der sich aus der Tilt-UI erneut anstoßen lässt, und ein `make`-Target für jede Routineaufgabe. Das Bootstrap-Skript installiert die komplette Toolchain auf macOS oder Ubuntu — idempotent und jederzeit gefahrlos wiederholbar.

Der Ertrag: Agenten, die Ende-zu-Ende entwickeln.

Alles auf den vorangegangenen Seiten läuft hier zusammen. Ein Arbeitsverzeichnis gibt einem Agenten das gesamte Produkt als Kontext; ausführbare Konventionen erklären ihm, wie das System läuft; Affected-Pipeline und lokaler Cluster geben ihm eine Feedback-Schleife, die er selbst bedienen kann. Ein Agent ist ein Entwickler mit unendlicher Geduld und ohne jedes implizite Wissen — und eine so gebaute Codebasis lässt ihn echte Engineering-Arbeit übernehmen, nicht nur Autocomplete.

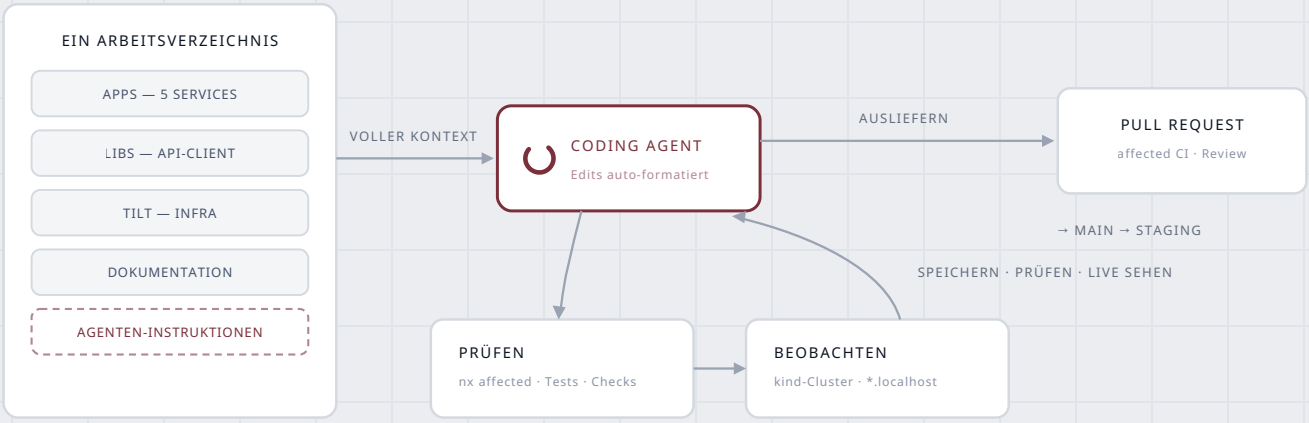


ABB. 05 — DIE AGENTEN-SCHLEIFE: VOLLER KONTEXT HINEIN, FOKUSSIERTE CHECKS UND EIN LAUFENDES PRODUKT ZUR VERIFIKATION — AUSGELIEFERT ÜBER DENSELBEN PULL-REQUEST-WEG WIE ALLE ANDEREN.

/01 Ein Verzeichnis, das ganze Produkt

Ein einziges Arbeitsverzeichnis enthält jeden Service, den gemeinsamen Client, die Infrastruktur-Manifeste und die Dokumentation. Der Kontext eines Agenten ist das System selbst — ein Feature über Servicegrenzen hinweg ist ein Branch und ein Pull Request statt einer Koordinationsübung über mehrere Repositories hinweg.

/03 Eine Prüfschleife, die er selbst bedient

Nx affected begrenzt Tests, Lint und Type-Checks auf das, was eine Änderung berührt — ein schnelles, deterministisches Signal. Und weil das ganze Produkt lokal startet, kann ein Agent über statische Checks hinausgehen: Cluster starten, *.localhost-Endpunkte aufrufen, Logs beobachten.

/02 Kontext, der aufgeschrieben ist

Das Repository bringt eigene Agenten-Instruktionen mit, und die README-Kette beschreibt das System so, wie es wirklich läuft — per Konvention aktuell gehalten. Formatierungs-Hooks laufen nach jeder Änderung; die Diffs eines Agenten sind damit von Haus aus kanonisch.

/04 Derselbe Weg wie für alle

Die Arbeit eines Agenten wird ausgeliefert wie die eines Menschen: Branch, Pull Request, Affected-Pipeline, Review. Keine Sonderwege — dieselben Konventionen, die den schnellen Weg für Menschen sicher machen, sichern ihn auch für Agenten.

Agentisches Engineering Ende-zu-Ende ist damit Arbeitsmodus, nicht Demo: **Ein Agent bringt ein Feature vom API-Endpunkt über den typisierten Client und die UI bis zur Verifikation im laufenden Produkt — ein Repository, ein Branch, dieselben drei Befehle, die auch ein Mensch nutzt.**

08 EIN WEG IN DIE PRODUKTION

Dieselben Konventionen tragen bis in die CI.

Die Continuous Integration läuft über **Nx affected**: Nur die Projekte, die eine Änderung tatsächlich berührt, durchlaufen Lint, Checks, Tests und Build. Ein täglicher Trivy-Scan blockiert kritische Sicherheitslücken, mit einer Allow-List, deren Einträge automatisch verfallen. Runner-Images werden nur veröffentlicht, wenn ihr Inhalts-Hash neu ist; App-Images werden gebaut — oder unverändert umgetaggt — und in einem Deploy-Manifest pro Commit festgehalten.

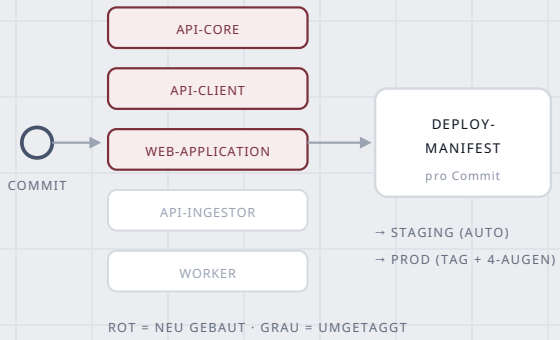


ABB. 06 — AFFECTED-BASIERTE CI MIT DEPLOY-MANIFEST PRO COMMIT.

Jeder Merge auf main wird automatisch nach Staging deployt. Produktion ist ein SemVer-Tag hinter einer Vier-Augen-Freigabe, mit automatisch generiertem Changelog. Wo fünf Deployment-Mechanismen jeweils eigenes Wissen verlangten, gibt es heute einen Weg — und das ganze Team kennt ihn.

09 ERGEBNIS

Ein Repository, drei Befehle — für Mensch und Agent.

01	03	01	02
MONOREPO — SIEBEN REPOSITORIES KONSOLIDIERT	BEFEHLE VOM CLONE ZUM LAUFENDEN PRODUKT	WEG IN DIE PRODUKTION — VORHER FÜNF	LOCKFILES FÜR DIE GESAMTE CODEBASIS

Die Migration lief über mehrere Wochen, Service für Service — die Roadmap pausierte nie —, und jedes Legacy-Repository wurde archiviert, sobald sein Nachfolger live war. Heute klont ein neuer Entwickler ein Repository, führt drei Befehle aus und hat das gesamte Produkt auf einem lokalen Cluster laufen, mit Live-Updates. Setup-Wissen, das früher in sieben READMEs lebte, ist heute ausführbare Konvention — und genau deshalb arbeiten in diesem Repository heute auch KI-Coding-Agenten: Ende-zu-Ende, über dieselben Pull Requests wie alle anderen.

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

Ist Ihre Codebasis bereit für KI-Agenten?

Ich konzipiere und baue Entwicklungsplattformen — Monorepos, lokale Cloud-Umgebungen, Build-Systeme und CI/CD, die den schnellen Weg zum Standardweg machen — für Teams und die Agenten, die mit ihnen arbeiten.

PROJEKT STARTEN ↗

hello@bastian-schoentag.de