First-Class

Bauen für Menschen und Agenten gleichermaßen

ZACH "LEIF" ERIKSEN

Urheberrecht

Kostenlos online lesbar. Das ePub ist nach eigenem Ermessen bezahlbar; wenn es dir geholfen hat, kannst du die Arbeit unterstützen.

github.com/0xLeif · leif.algo

Eines von vier Büchern im agent-stack-Set. Wie es entstanden ist, steht im Kolophon am Ende.

Widmung

Für alle, die offen bauen und es trotzdem veröffentlichen.

Die Bibliothek

Diese Bücher stehen für sich allein, wurden aber als Set geschrieben. Code wurde billig und Vertrauen wurde knapp. Zusammen bilden sie ein einziges Argument: was jetzt zu bauen ist und wie man ihm vertrauen kann.

- **The Agent Developer's Field Guide**: Werkzeuge, Spezifikationen und Vertrauen für Agenten bauen, die echten Code liefern
- **First-Class**: Bauen für Menschen und Agenten gleichermaßen *(dieses Buch)*
- **Building Agents**: Notizen aus dem Versuch, Software eigene Hände zu geben
- **Open Source Tooling**: Werkzeuge bauen, die Menschen wirklich nutzen

Kostenlos online lesbar. Jedes ePub ist nach eigenem Ermessen bezahlbar.

Inhalt

Einführung

Dies ist das kurze, eigenwillige Buch der vier, und das, für das die anderen eigentlich argumentieren.

Die meisten Werkzeuge werden für Menschen gebaut, und dann wird ein Agent an der Seite angeschraubt: eine zweite API, ein Flag, ein besonderer Modus, der die Hälfte von dem macht, was das eigentliche Produkt kann. Ich glaube, das ist falsch herum. Die These hier ist einfach. Ein Werkzeug sollte für beide first-class sein. Ein Mensch sollte es ohne Agenten bedienen können, und ein Agent sollte es ohne Menschen bedienen können, über dieselbe Oberfläche, ohne eine zweitklassige Tür.

Ich bin dazu durch Bauen gekommen, nicht durch Theorie. Ich baue kleine Swift-Bibliotheken und Entwicklerwerkzeuge, und die, die gut gealtert sind, waren die, bei denen der Kern sauber genug war, dass alles ihn aufrufen konnte. Als Agenten auftauchten, waren diese Werkzeuge bereits bereit. Die anderen sind die, für die ich mich immer noch entschuldige.

Wenn der *Field Guide* die Methode ist und *Building Agents* und *Open Source Tooling* die Beweise sind, dann ist dies die These, der sie alle dienen. Du brauchst die anderen drei nicht, um dieses zu nutzen. Es ist hier, um zu verändern, wie du das nächste Werkzeug betrachtest, das du baust oder auswählst: Frag, ob es einen Menschen und einen Agenten als gleiche Art von Bürger behandelt, und sieh, wie viel sich aus der Antwort ergibt.

Es ist absichtlich kurz. Lies es in einem Zug.

Gleiche Bürger

Ein Großteil meiner Werkzeuge kommt von einer Idee: Menschen und Agenten werden dieselben Werkzeuge nutzen.

Hier also die Definition, auf der das ganze Buch aufbaut, direkt und klar: first-class bedeutet, ein Mensch bedient das Werkzeug allein, ein Agent bedient das Werkzeug allein, dieselben Befehle in beide Richtungen. Nicht zwei Produkte. Ein Werkzeug, zwei Arten von Nutzern, keiner davon der Sonderfall, in den der andere übersetzt wird.

Wenn ich *first-class* sage, meine ich es so, wie ein Programmierer es meint: ein echter, unterstützter Teilnehmer, für den das Werkzeug tatsächlich gebaut wurde, kein Gast, den es toleriert. Kein Flug-Upgrade. Ein first-class-Nutzer ist einer, um den herum das Werkzeug entworfen wurde, mit einem echten Zugang. Nicht einer, der erst in jemand anderes Form übersetzt werden muss.

Es kann agenten-first-Werkzeuge geben, nur für Agenten gebaut. Es kann mensch-first-Werkzeuge geben, nur für Menschen gebaut. Der interessante Fall, den fast niemand baut, ist das Werkzeug, das beides ist, und "beides" meint *first-class für beide*, was auch der Rest des Buches darunter versteht. Es ist strenger als agenten-first: agenten-first muss nur dem Agenten dienen, während first-class-für-beide einer Person und einem Agenten über dieselbe Oberfläche dienen muss, ohne dass eine Seite eine schlechtere Version bekommt. Im Moment haben wir meist mensch-first-Projekte und bringen Agenten *in* sie. Der Agent taucht spät auf, bei einem Werkzeug, das für jemand anderen gebaut wurde, und wir hoffen, dass er es herausfindet.

Was wir eigentlich brauchen, sind mensch-und-agenten-first-Projekte. Von Anfang an so gebaut.

"Mensch-first mit angeschraubten Agenten" ist der Standard, weil es der Weg des geringsten Widerstands ist. Du hast das Werkzeug bereits, es funktioniert bereits für dich, und wenn ein Agent auftaucht, ist der einfache Schritt, ein bisschen Klebeband um das zu wickeln, was du hast. Es funktioniert, irgendwie. Du hast ein Werkzeug genommen, das um die Augen und Hände eines Menschen herum entworfen wurde, und hast einen Prozess ohne beides gebeten, so zu tun, als hätte er beides. Was das einen Agenten kostet (das Hängen an einer Eingabeaufforderung, die Ausgabe, die er nicht lesen kann, das Neuerlernen bei jedem Lauf) hat sein eigenes Kapitel später.

Hier reicht es zu sagen, dass der Agent auf jeder Zeile, die einen Menschen voraussetzte, eine Lücke überpflastert.

Also dreh die Annahme um. Bau das Werkzeug so, dass es auf beide Arten first-class funktioniert. Das ist die Definition von oben, umformuliert als Bauanweisung: Du hast kein echtes Produkt und einen "API-Modus" an der Seite angenäht, und du hast keine CLI und einen separaten Agenten-Shim, der beim ersten Mal, wenn du etwas änderst, aus dem Sync läuft. Beide sind echte Nutzer. Beide sollen dort sein.

Und hier ist der Teil, der mir am wichtigsten ist. Wenn der Agent ein first-class-Nutzer ist, kann das Werkzeug ihm tatsächlich *helfen*, anstatt ihn alle Befehle erraten zu lassen. Ein Mensch kann durch ein verwirrendes Werkzeug durchwursteln. Er liest die README, probiert etwas, liest den Fehler, probiert etwas anderes, fragt einen Kollegen. Ein Agent, der sich durchwurstelt, ist nur teures Raten. Ein Werkzeug, das für den Agenten gebaut wurde, sagt ihm, was möglich ist, gibt ihm Ausgaben, die er direkt verwenden kann, und schlägt fehl auf eine Art, die sagt, was als nächstes zu tun ist.

Das ist dasselbe Werkzeug, das Menschen wollen. Auffindbare Befehle, Ausgaben, denen man vertrauen kann, Fehler, die sagen, was schiefgelaufen ist. Der Agent kann die Lücken nur nicht so überpflastern wie ein Mensch, also zwingt das Bauen für den Agenten dazu, sie zu schließen. Für beide zu entwerfen macht die Software besser.

Leute hören "auch für Agenten bauen" und nehmen an, das bedeute zwei Produkte oder einen Kompromiss, bei dem jede Seite eine schlechtere Version von dem bekommt, was sie wollte. Es ist ein guter Kern mit einem Schritt, bei dem man ihn beiden exponiert. Das ist nicht doppelt so viel Arbeit, und das nächste Kapitel zeigt warum.

Der Grund, warum ich immer wieder darauf zurückkomme, ist, dass die Agenten im Laufe der Zeit nur mehr tun werden, nicht weniger. Heute torkeln sie durch menschliche Werkzeuge. Morgen erledigen sie einen echten Anteil der Arbeit, und der Anteil steigt. Wenn die Werkzeuge, die wir jetzt bauen, davon ausgehen, dass immer ein Mensch da ist, um die Eingabeaufforderung zu behandeln, den Bildschirm zu lesen, den Knopf zu drücken, bauen wir eine Zukunft auf einer Annahme, die jeden Monat falscher wird.

Es gibt auch eine größere Version dieses Arguments. Wir hatten ein Jahrzehnt Code, der vollständig von Menschen geschrieben wurde, und jetzt schreibt KI Code auf allem davon, und irgendwann kippt diese Mischung. Die Werkzeuge sind das, was entscheidet, ob das gut oder schlecht läuft. Das ist ein ganzes eigenes Kapitel: die Brücke.

Für jetzt lautet die These einfach: Menschen und Agenten werden dieselben Werkzeuge nutzen. Also bau sie für beide, als gleiche first-class-Bürger, von Anfang an.

First-class für einen Agenten

Was braucht ein Werkzeug also wirklich, damit ein Agent ein first-class-Nutzer ist und kein menschliches Werkzeug, durch das ein Agent sich durchtorkelt?

Vier Dinge. Keines davon ist exotisch.

Strukturierte, maschinenlesbare Ausgabe. Ein echtes Serialisierungsformat -- JSON ist das, zu dem ich greife -- damit der Agent Daten bekommt, keinen Bildschirm. Die meisten Werkzeuge geben schönen Text zurück: ausgerichtete Spalten, Farbe, eine Zusammenfassungszeile am Ende. Das ist für die Augen eines Menschen. Ein Agent muss es scrapen, und das Scrapen bricht an dem Tag, an dem du den Abstand änderst. Gib ihm die echten Daten. Er liest ein Feld, anstatt einen Absatz zu parsen.

Auffindbare, konsistente Befehle. Konsistente Verben im gesamten Werkzeug und ein selbst-beschreibendes `--help`, das der Agent lesen kann, um herauszufinden, was möglich ist. Wenn der Hilfetext real ist, liest der Agent ihn und weiß, was das Werkzeug kann. Er muss dieses Werkzeug nicht schon einmal gesehen haben. Er fragt das Werkzeug, und das Werkzeug antwortet.

Fehler, die den nächsten Schritt lenken. Wenn etwas schiefläuft, sag, was zu tun ist. Kein Stack-Trace, kein `error: 1`. Ein Mensch kann herumgraben und einen nackten Fehlercode herausfinden. Ein Agent bekommt einen nackten Fehlercode und steckt fest, oder schlimmer, er tut selbstsicher das Falsche.

Nicht-interaktiv und deterministisch. Es läuft ohne überraschende Eingabeaufforderungen, sodass der Agent nie beim Warten hängenbleibt. Nichts bringt einen Agentenlauf so zum Erliegen wie ein Werkzeug, das plötzlich fragt "Bist du sicher? [j/N]" und dort für immer sitzt, weil niemand da ist, um zu antworten. Gib ihm ein Flag, um direkt durchzulaufen. Mach es deterministisch, damit dieselbe Eingabe dasselbe Ergebnis liefert.

Hier ist, was ich möchte, dass du an dieser Liste bemerkst: Jeder Punkt darin ist schlicht gutes CLI-Design. Nichts davon ist agentenspezifische Magie. Ein Mensch will auch klare Fehler und vorhersehbares Verhalten und auffindbare Befehle. Der Agent kann nur nicht die Achseln zucken und deren Fehlen überwinden. Also ist Bauen für den Agenten gleich Bauen des Werkzeugs und Ablehnen, sich auf "ach, ein Mensch wird es schon richten" zu verlassen.

Jetzt der Teil, den die Leute falsch machen. Sie denken, Entwerfen für Agenten und Entwerfen für Menschen ziehe auseinander, dass man zwei Herren dient und jemand verliert. Sie ziehen zusammen. Hier ist die eigentliche Form.

Du entwirfst einen wirklich guten Kern, der funktioniert. Dann exponierst du ihn. Du fügst die Flags hinzu: nicht-interaktiv, oder `--json`, oder introspect, was auch immer das Werkzeug braucht. Und jetzt kann der Agent das CLI-Werkzeug nutzen, und Menschen können das CLI-Werkzeug nutzen. Dasselbe Werkzeug. Derselbe Kern. Du hast eine Sache gebaut und sie zweimal exponiert.

Dann wird es von dort aus noch weiter erweitert. Mehr UI für Menschen. Oder einfach Werkzeuge ohne UI. Oder Plugins, die Menschen erstellen, oder Plugins, die Agenten helfen, verschiedene Dinge. Die Erweiterung ist der Ort, an dem sich die Zielgruppen wirklich trennen: Ein Mensch will Affordanzen, eine schöne Oberfläche, eine UI; ein Agent will Introspect und Plugins und Modi. Gut. Bau die. Aber bau sie *auf dem* gemeinsamen Kern, als Erweiterungen, nicht als zwei separate Produkte, die du für immer synchron halten musst.

Im Kern ist es dieselbe Sache. Das ist die Zeile, zu der ich immer wieder zurückkomme. Sie muss nur Agenten und Menschen exponiert werden. Also ja, es gibt *einen* zusätzlichen Schritt. Die Dual-Use-Eigenschaft bekommst du nicht kostenlos; du musst den Kern bewusst auf beide Arten exponieren.

Dieser Perspektivwechsel ist der ganze Punkt dieses Kapitels, weil er den Einwand auflöst, bevor er beginnt. Leute widersetzen sich dem Bauen für beide, weil sie es als doppelte Arbeit bepreisen: zwei Designs, zwei Test-Suites, zwei Dinge, die gewartet werden müssen, zwei Dinge, die kaputt gehen können. Eigentlich ist es ein Kern plus ein Expositionsschritt. Der Kern ist der teure Teil, und du würdest ihn sowieso bauen. Den Kern für einen Agenten zu exponieren ist meist die Disziplin strukturierter Ausgabe und konsistenter Befehle und guter Fehler -- die Dinge, die du sowieso hättest tun sollen.

Hier lauert eine Ordnungsfrage. Kommt der Kern zuerst, oder entwirfst du für beide gleichzeitig? Das ist genug ein eigenes Ding, dass es später ein eigenes Kapitel bekommt. Für jetzt: Du exponierst den Kern auf beide Arten, und das Exponieren ist der billige Teil.

Mein eigenes Tooling ist genau so gebaut. Nimm fledge. Es hat ein echtes `introspect`-Verb, und der gesamte Zweck von `fledge introspect --json` ist es, einem Agenten zu ermöglichen, die verfügbaren Befehle als strukturierte Daten zu entdecken, anstatt `--help`-Text für einen Menschen zu scrapen. Das ist der Expositionsschritt buchstäblich gemacht: gleicher Kern, aber der Agent kann fragen "Was kann ich hier tun?" und

eine maschinenlesbare Antwort zurückbekommen. Dann mache ich die Befehle selbst headless lauffähig. Setze `FLEDGE_NON_INTERACTIVE`, damit nichts stoppt, um eine Frage zu stellen, übergib `--json`, damit die Ausgabe als Daten zurückkommt, und jetzt hat der Agent einen first-class-Zugang, der nie davon abhing, Prosa zu lesen.

Lass es mich konkret machen. `fledge doctor` prüft deine Projektumgebung. Führe es normal aus und du bekommst etwas für deine Augen: ausgerichtete Spalten, ein Häkchen pro Zeile, ein Emoji-Status, eine Zusammenfassungszeile:

```
Git
  ✅ git 2.45.2
  ✅ repository: initialized
  ✅ remote: origin ➡️ git@github.com:CorvidLabs/fledge.git
  ✅ working tree: clean

8 checks passed, 0 issues found
```

Dieser Block ist genau das "Vorher", an dem ein Agent scheitert. Er ist hübsch, und er ist ein Bildschirm. Um zu wissen, dass der Remote gesetzt ist, muss der Agent die richtige Zeile finden, ein grünes Häkchen erkennen und darauf vertrauen, dass sich der Abstand beim nächsten Release nicht verschiebt. Der Status, der ihn interessiert, ist ein Emoji, das in einer Spalte vergraben ist. In der Praxis: Der Agent scrapt den ausgerichteten Text, das Parsen bricht, wenn ein längerer Repository-Name die Spalten beim nächsten Release um zwei Zeichen verschiebt, und der Agent liest den Remote-Check als bestanden, wenn er fehlt. Er hat den falschen Zweig auf einem String-Match genommen, der nie ein echter Vertrag war.

Führe denselben Befehl mit `--json` aus und dieselben Checks kommen als Daten zurück:

```
{
  "schema_version": 1,
  "action": "doctor",
  "passed": 8,
  "failed": 0,
  "sections": [
    {
      "name": "Git",
      "checks": [
        { "name": "git", "status": "ok", "version": "2.45.2", "detail": null,
"fix": null },
        { "name": "repository", "status": "ok", "version": null, "detail":
"initialized", "fix": null }
      ]
    }
  ]
}
```

`schema_version` ist absichtlich das erste Feld. Es ist das, was der Agent vor allem anderen liest, weil es ihm sagt, in welcher Form der Rest des Dokuments vorliegt. An dem Tag, an dem sich das Ausgabeformat ändert, ändert sich diese Nummer mit, und ein Agent, der an eine Version gebunden ist, weiß, dass er sich anpassen muss, anstatt neue Daten stillschweigend so zu lesen, als wären sie alt. So überlebt der Vertrag die Überarbeitung des Werkzeugs.

Danach, gleicher Kern, dieselben Checks liefen. Der Mensch bekommt Häkchen und eine Zusammenfassungszeile. Der Agent bekommt ein `status`-Feld, auf das er verzweigen kann, anstatt ein grünes Häkchen aus einer Spalte zu scrapen. Das `fix`-Feld ist hier `null`, weil nichts falsch ist; das ist der Vertrag: `fix` ist `null` bei einem bestandenen Check.

Führe es jetzt aus, wo etwas *falsch* ist. Angenommen, das Repository hat keinen konfigurierten Remote. Die normale Version dunkelt eine Zeile ab und der Agent ist wieder am Raten; die `--json`-Version macht daraus einen Datensatz, auf den er reagieren kann:

```
{
  "schema_version": 1,
  "action": "doctor",
  "passed": 7,
  "failed": 1,
  "sections": [
    {
      "name": "Git",
      "checks": [
        { "name": "repository", "status": "ok", "version": null, "detail":
"initialized", "fix": null },
        { "name": "remote", "status": "missing", "version": null, "detail": "no
remote configured", "fix": "git remote add origin <url>" }
      ]
    }
  ]
}
```

Das ist der Zweig, auf dem der Agent tatsächlich handelt. `status` wechselt von `"ok"` zu `"missing"`, `detail` sagt in klaren Worten, was falsch ist, und `fix` ist jetzt ein gefüllter String, ein buchstäblicher Befehl, den der Agent ausführen kann, um es zu reparieren. Der Agent muss nicht wissen, was ein fehlender Remote bedeutet oder eine Wiederherstellung erfinden; das Werkzeug, das das Problem gefunden hat, hat auch den Fix übergeben. Der Mensch bekommt eine gedunkte Zeile; der Agent bekommt einen `status`, auf den er verzweigen kann, und einen `fix`, den er ausführen kann. Ein Befehl, zweimal exponiert.

Der Test, ob du es richtig gemacht hast, ist einfach. Gib das Werkzeug einer Person ohne Agenten. Funktioniert es, ist es gut? Gib es einem Agenten ohne Person. Funktioniert es, ist es gut? Wenn die Antwort auf beide Ja ist und du das Ding nicht zweimal gebaut hast, um dahin zu kommen, hast du es richtig gemacht.

Die Brücke

Ich habe das am Ende des vorletzten Kapitels angedeutet: Es gibt eine größere Version des Arguments, eine Prozentzahl, und das ist sie.

Nimm den Faden wieder auf. Wir hatten mehr als ein Jahrzehnt von 100% menschlich geschriebenem Code, jede Zeile in jedem Repository von einer Person geschrieben, und das bedeutet tonnenweise Legacy-Code, tonnenweise technische Schulden, ein Jahrzehnt davon, das alles betreibt. Das ist die Welt, wie sie gerade tatsächlich ist.

Und jetzt nutzen wir KI auf all dem.

Das ist der schwierige Teil, und die Leute sitzen nicht lange genug damit. Es gibt so viel Legacy und menschliches Zeug, das bereits da ist. Die KI taucht bei *dem* auf.

Und wenn wir so weitermachen, wenn wir uns immer mehr auf KI verlassen, um Dinge zu schreiben, wird es eine echte Frage. Ab welchem Punkt kippt es zu 100% KI-geschriebenem Code? Wird es das je? Und was macht man mit dem Jahrzehnt menschlichen Codes, der bereits da ist? Lässt man den Legacy-Code als Legacy und zieht weiter? Schreibt man ihn neu? Tut man so, als wäre er nicht da? Niemand hat wirklich eine gute Antwort, und die meisten stellen die Frage nicht mal. Sie schrauben einfach einen Agenten an das, was sie haben, und hoffen.

Dieser gesamte Übergang, von allem-menschlich zu was-auch-immer-als-nächstes-kommt, ist der Zweck der richtigen Werkzeuge. Die Werkzeuge sind die Brücke darüber.

Im Idealfall richtest du Werkzeuge ein, die *sowohl* Menschen als auch KIs nutzen können. Du richtest es so ein, dass du mit spec-driven Development schreibst und baust, spec-sync, und dann fledge für die Befehle. Das ist die Brücke. Wenn du anfängst, diese Werkzeuge zu nutzen, ist es leichter, einen Agenten in bestehenden Code einzuführen. Und wenn du ein brandneues Projekt beginnst, ist es auch leichter für Menschen zu steuern. Das gleiche Setup hilft dem Agenten, in ein jahrzehntealtes Durcheinander zu gehen, und hilft einer Person, die Kontrolle über etwas Brandneues zu behalten.

Es sind Spezifikationen, keine besseren Prompts oder klügeren Agenten, weil eine Spezifikation eine gemeinsame Quelle der Wahrheit für beide Seiten ist. Der Mensch formuliert die Absicht. Der Agent baut dazu. Es ist dasselbe Problem wie das letzte

Kapitel, eine Ebene höher. Ein Agent, der sich durch ein menschliches Werkzeug torkelt, rät bei den Befehlen. Ein Agent, der ohne Spezifikation in eine Codebasis geworfen wird, rät bei der *Absicht*. Die Spezifikation ist das, was er stattdessen liest.

Und es hält den Code ehrlich zur Spezifikation. Das ist der Teil, der die Brücke vertrauenswürdig macht, anstatt nur hoffnungsvoll. spec-sync hält die Spezifikation und den Code in Übereinstimmung, in CI erzwungen, sodass der Agent nicht still von dem abweichen kann, was vereinbart wurde. Das ist wichtiger, als es klingt. Der Fehlermodus, den jeder mit Agenten fürchtet, ist nicht der dramatische; es ist der stille, bei dem der Agent langsam etwas baut, das nicht das ist, was du gefragt hast, und niemand es bemerkt, bis es tief drin ist. CI, das Code gegen Spezifikationen prüft, ist das, was die Abweichung aufgreift, solange sie noch klein ist. Der Vertrag wird von beiden Seiten gelesen, und eine Maschine prüft, dass niemand ihn gebrochen hat.

Hier ist, was diese Maschine eigentlich prüft. Die Spezifikation ist eine Markdown-Datei, eine `*.spec.md`, die den Vertrag aufschreibt: den Zweck, die öffentliche API, die Invarianten, die Fehlerfälle. spec-sync gleicht das mit dem echten Code ab, in beide Richtungen. Ein Export, den die Spezifikation nie dokumentiert hat, wird markiert. Ein Spezifikationsversprechen, das der Code nicht hat, ist ein Fehler. Es prüft deklariertes Schema gegen die tatsächlichen Migrationen auf dieselbe Weise. Es ist strukturell: Passt die öffentliche Oberfläche und das Schema zu dem, was aufgeschrieben wurde. Es sind keine generierten Tests und kein fuzziges Diff gegen die Prosa, und diese Enge ist der Grund, warum ich ihm vertraue: Es behauptet, dass der Vertrag und der Code in der Form der Dinge übereinstimmen, nichts mehr. In CI läuft es als echtes Gate. Es verlässt mit Nicht-Null, wenn sie auseinandergegangen sind, und postet die Abweichung direkt auf den Pull Request.

Das setzt die Arbeitsteilung auf, die mir wirklich wichtig ist. Der Mensch besitzt die Absicht. Der Agent besitzt die Mühsal. Der Mensch bleibt auf der Absichtsebene in Kontrolle -- was das tun soll und warum -- während der Agent die Implementierung übernimmt. Jede Seite macht den Teil, für den sie wirklich gut ist, mit der Spezifikation als Linie zwischen ihnen.

Und es ist eine sichere Auffahrt in Legacy-Code, was hier zurück zum Prozentzahl-Problem führt. Du hast ein Jahrzehnt Code ohne Spezifikationen, weil niemand Spezifikationen geschrieben hat, weil ein Mensch es geschrieben hat und die Absicht in diesem Menschenkopf lebte. Der Schritt ist: Erfasse, was ein Stück *tun soll*, als Spezifikation, dann lass den Agenten dagegen arbeiten. Du bittest den Agenten nicht, die Absicht aus zehn Jahre altem Code zu erraten. Du schreibst die Absicht zuerst auf. Ein Mensch kann das tun, das ist der Teil, den Menschen gut können. Und jetzt hat der Agent einen Vertrag, gegen den er bauen und refaktorisieren kann. So kommt

ein Agent in bestehenden Code, ohne dass es eine Katastrophe wird. Die Spezifikation ist die Auffahrt.

Stell dir den Schritt bei einer echten Funktion vor, die keine Spezifikation hat. Vorher: funktionierender Code, kein schriftlicher Vertrag, die Absicht lebt vollständig im Kopf des ursprünglichen Autors, sodass ein Agent, der darin geworfen wird, die Signaturen liest, bei den Invarianten rät und anfängt, Dinge zu ändern. Nachher: derselbe Code mit einer `*.spec.md` daneben, die öffentliche API und die Invarianten in klarer Sprache aufgeschrieben, und spec-sync erzwingt in CI, dass der Code ehrlich zu dem bleibt, was geschrieben steht. Du musst nicht die gesamte Codebasis auf einmal machen. Du machst es ein Stück auf einmal und schreibst den Vertrag für den Teil, den du gerade übergeben willst.

Die Spezifikation hat nicht verändert, was der Code tat. Sie hat verändert, was der Agent mit dem Code tun konnte: gegen einen schriftlichen Vertrag arbeiten, anstatt einen zu erraten.

fledge ist die Befehlsoberfläche für alles davon: die konsistenten Verben, durch die sowohl der Mensch als auch der Agent die Arbeit vorantreiben, das Ding aus dem letzten Kapitel, das den Agenten nicht raten lässt, wie er bauen, testen und laufen soll. spec-sync ist der Vertrag; fledge ist, wie man darauf handelt.

Hier ist die Naht zwischen ihnen. spec-sync steht für sich allein: sein eigenes Werkzeug, sein eigener CI-Check. Aber fledge behandelt die Spezifikation als first-class-Teil der Entwicklungsschleife. Es gibt ein `spec`-Verb direkt neben dem Rest, und `fledge ask` und `fledge review` sind spezifikationsbewusst, sodass die Spezifikation der Kontext ist, den der Agent liest, wenn er eine Frage beantwortet oder eine Änderung überprüft. Die Spezifikation ist keine Dokumentation, die der Agent ignoriert; sie ist das, was fledge dem Agenten zum Arbeiten gibt.

Also feuert das Gate von beiden Enden. Fledges `spec`-Säule führt den spec-sync-Check direkt in der Entwicklungsschleife aus, und die `CorvidLabs/spec-sync@v4` GitHub Action führt ihn erneut in CI aus, sodass nichts gemergt wird, das still abgewichen ist. Wenn ein Agent die Arbeit erledigt, reitet dieser Check in seiner Schleife, sodass er auf dem Spec bleibt, während er fortfährt, anstatt es am Ende herauszufinden.

Jetzt folge es bis zum Ende, denn das Ende ist der Grund für alles davon. Wenn das funktioniert, wenn die Werkzeuge und Spezifikationen und die Schienen, die Agenten ehrlich halten, einfach *da* sind, dann steigen Menschen auf. Hoch zu Absicht und Richtung, der Agent erledigt darunter die Mühsal gegen eine Spezifikation, offen wo du es prüfen kannst. Dieser Schritt bekommt später sein eigenes Kapitel; hier reicht es zu wissen, wohin die Brücke führt.

Der Prozentsatz wird sich weiter verschieben. Mehr Code wird von Agenten geschrieben; das ist einfach wahr. Was uns obliegt, ist die Brücke, die gemeinsamen Werkzeuge, der gemeinsame Vertrag und ob wir ihn tatsächlich bauen. Ich glaube nicht, dass die meisten Leute angefangen haben.

Die Identitätsschranke

Bevor wir zu den Werkzeugen kommen, die ehrliche Grenze. Das gesamte Argument läuft gegen eine Wand, und es ist eine, die ich mit besseren Werkzeugen nicht einreißen kann, also möchte ich es klar und frühzeitig sagen: Die Plattformen erlauben es einem Agenten nicht, als er selbst zu existieren. Alles nach diesem Kapitel ist das, was du angesichts dessen tust.

Wenn ein Agent ein first-class-Nutzer deiner Werkzeuge ist, muss er früher oder später auch ein first-class-Bürger der Plattformen sein. Das sind dieselbe Idee in zwei Richtungen. Ein Werkzeug, das den Agenten als echten Nutzer behandelt, gibt ihm strukturierte Ausgabe und auffindbare Befehle und einen Zugang, der nicht von einem Menschen abhängt. Eine Plattform, die den Agenten als echten Teilnehmer behandelt, würde ihm ein Konto geben, eine Identität, einen legitimen Platz, um unter allen anderen zu existieren. Das zweite Ding ist genau das, was du nicht bekommen kannst.

Ich habe es versucht. Ich wollte einem Agenten seine eigene Identität auf den Plattformen geben, auf denen jeder baut: sein eigenes Konto, kein Wrapper um meines. Die Plattformen erlauben es nicht. Ich erzähle diese Geschichte ordentlich im Buch Building Agents. Was hier mitzunehmen ist, ist, was für eine Wand es ist. Der Blocker vor einem wirklich autonomen Agenten ist nicht die Fähigkeit des Modells, und es ist nicht mal Sicherheit. Es ist, dass die Plattformen dem Agenten keine Identität gewähren, um damit zu existieren.

Es ist eine Richtlinien-Wand, keine technische. Ich kann die VM betreiben. Ich kann die Schleife verdrahten. Das Modell kann die Arbeit erledigen. Nichts davon ist der Blocker. Der Blocker ist, dass ich die Plattform nicht dazu bringen kann, dem Agenten zu erlauben, dort zu sein, und das ist nicht meins zu beheben. Es ist eine Entscheidung, die jemand anderes getroffen hat, stromaufwärts von allem, was ich bauen kann.

Es ist wichtig wegen der Rechenschaftspflicht, was das Modell ist, das ich eigentlich will. Der Endzustand ist kein wild laufender Agent. Es ist ein Agent mit einer echten, benannten, skalierten Identität: volle Fähigkeit, reduzierte Privilegien, ein menschliches Genehmigungstor. Er erledigt die Arbeit, liefert sie bis zu einem Pull Request, und der Merge ist meiner, weil wenn etwas unter meinem Namen in der Welt handelt, ich derjenige bin, der dafür unterzeichnet. Aber du kannst nicht

rechenschaftspflichtig haben ohne *Identität*. Ein Agent, den du nicht benennen, nicht skalieren, nicht auf den du zeigen und sagen kannst "der hat das getan": Es gibt nichts, das rechenschaftspflichtig gehalten werden kann. Verweigere die Identität und du hast die rechenschaftspflichtige Version damit verweigert, und was übrig bleibt, ist entweder kein Agent oder ein nicht rechenschaftspflichtiger.

Hier also der Rahmen für den Rest des Buches. Der Agent kann ein first-class-Nutzer eines *Werkzeugs* sein, das ich baue, weil das mein Teil ist. Er kann kein first-class-Bürger einer *Plattform* sein, die ich nicht besitze, weil das nicht mein Teil ist. Alles nach diesem Kapitel ist innerhalb dieser Grenze gebaut: So sieht first-class für beide aus, wenn du dem Agenten einen echten Zugang zu deinem Werkzeug geben kannst, aber keinen echten Platz in der Welt. Nicht die KI, nicht die Technik, nicht die Sicherheit. Die Plattformen lassen den Agenten nicht existieren. Alles andere kann ich bauen. Das noch nicht. In 2026 steht die Wand noch. Die einzigen Wege hindurch sind Workarounds, die du selbst betreibst: ein gereiftes menschliches Konto mit echter Aktivitätsgeschichte (ein frisches Agentenkonto wird sofort blockiert), oder eine Verifizierungsschicht, die du selbst hostest. Das ist der Stand der Dinge.

Wenn es angeschraubt wird

Es ist einfach, "für beide bauen" zuzunicken und sich das Scheitern nie wirklich vorzustellen. Also lass mich dich auf die Seite des Agenten eines mensch-first-Werkzeugs setzen, denn dort spürst du es.

Zwei Dinge gehen schief, und sie gehen ständig schief.

Das erste ist, dass der Agent hängenbleibt. Werkzeuge hängen ständig an interaktiven Eingabeaufforderungen: eine Bestätigung, ein "Bist du sicher?", etwas, das da sitzt und auf einen Tastendruck wartet. Und niemand ist da, um die Taste zu drücken. Also schlägt der Lauf nicht genau fehl. Er stoppt einfach. Er sitzt an einer Eingabeaufforderung, die für eine Person geschrieben wurde, die rüberschauen und Enter drücken würde, und der Agent wartet, weil Warten das Einzige ist, was das Werkzeug ihm zu tun gegeben hat. Ein ganzer Lauf tot an einer Frage, die niemand da ist zu beantworten.

Das zweite sind die Docs. Es gibt entweder keine, oder es gibt sie und sie sind verwirrend. Das bedeutet, der Agent muss das Werkzeug *lernen*. Und hier ist der Teil, den ich immer wieder bemerke. Er lernt es nicht wirklich. Er kann nicht. Es gibt nirgendwo, wo dieses Wissen zwischen Läufen leben könnte. Also macht er die teure Version: Er scannt die Dateien, liest was auch immer im Index ist, macht eigene Notizen, rekonstruiert, wie das Werkzeug funktioniert, von Grund auf neu. Jedes Mal. Das Werkzeug *weiß*, was es kann, es ist direkt im Code, und es sagt dem Agenten einfach nie. Also baut der Agent dieses Bild bei jedem einzelnen Lauf von Null neu auf.

Stell dir vor, wie verschwenderisch das ist. Eine Person liest die Docs einmal, vielleicht überfliegt sie, und trägt es danach im Kopf. Sie entwickeln ein Gefühl für das Werkzeug. Der Agent bekommt nichts davon umsonst. Was das Werkzeug ihm nicht direkt sagt, muss er wieder herausgraben, wieder dafür bezahlen, wieder erraten. Die Arbeit, die das Werkzeug einmal hätte erledigen sollen, macht der Agent für immer neu.

Beide sind derselbe Fehler in zwei Kostümen. Das Werkzeug nahm an, dass ein Mensch da sein würde, die Geduld eines Menschen bei der Eingabeaufforderung, das Gedächtnis eines Menschen daran, wie das Ding funktioniert, und ein Agent hat beides nicht. Er kann nicht die Achseln zucken und warten. Er kann sich nicht über die Wand zwischen Läufen erinnern, außer du gibst ihm etwas zum Erinnern.

Und beide kartieren direkt auf die Liste aus ein paar Kapiteln zurück. Nicht-interaktiv: Stopp nicht und warte auf eine Person, die nicht kommt. Auffindbar: Sag dem Agenten, was du kannst, in einer Form, die er lesen kann, damit er dich nicht aus deinem eigenen Quellcode rekonstruieren muss. Das Hängen und das Wiederlernen sind keine exotischen Agentenprobleme. Sie sind einfach diese beiden fehlenden Eigenschaften, und ein Mensch deckt die Lücke still jedes Mal ab, sodass du nie bemerkt hast, dass die Lücke da war.

Agenten an mensch-first-Werkzeuge anzuschrauben funktioniert meist, genau bis du siehst, was der Agent tun muss, um es zum Laufen zu bringen. Dann siehst du, wie viel vom Werkzeug die ganze Zeit auf eine Person gestützt hat.

Hier ist eines von meinen, benannt. fledge begann als Task-Runner, den ich von Hand bediente. Der Kern war von Anfang an sauber, die Build-, Test- und Run-Logik saß hinter den Befehlen und nicht in ihnen, also funktionierte der Tag, an dem ich einen Agenten zum ersten Mal darauf zeigte, das meiste davon einfach. Außer an einer Stelle. Der Agent lief `fledge work commit` und blieb tot stehen. Dieser Befehl druckt `Commit message:` und wartet, dass jemand tippt. Es gab niemanden. Der Agent saß an einem blinkenden Cursor, bis er aufgab, weil dieser eine Befehl still auf eine Person gestützt hatte die ganze Zeit. Die Lösung war kein klügerer Agent. Es war der nicht-interaktive Weg: ein `FLEDGE_NON_INTERACTIVE`-Schalter, der jede Eingabeaufforderung so behandelt, als wäre sie bereits beantwortet, und eine Möglichkeit, die Nachricht vorab zu übergeben, anstatt darauf zu warten. Das Verräterische daran ist, dass die Eingabeaufforderung nur existierte, weil ich derjenige war, der sie beantwortete. Von Anfang an für beide gebaut, wäre sie nicht da gewesen, an der man hängenbliebe.

Den Kern zweimal exponieren

Schwieriger und neuer Kern: Bau ihn zuerst, exponiere ihn danach. Unkomplizierter Kern: Entwirf von Anfang an für beide. Das ist die Heuristik. Hier ist, warum es so funktioniert.

Wenn der Kern der schwierige Teil ist, der neue Teil, das Ding, das wirklich schwer richtig zu machen ist, bau ich das zuerst und kümmere mich später um die Oberfläche. Krypto ist so. Ein Parser ist so. Ein Scorer, der korrekt sein muss, ist so. Die schwierige Mitte ist der Ort, an dem das gesamte Risiko liegt, also geht dort die Aufmerksamkeit zuerst hin. Ich bringe das Ding zum *Funktionieren*, korrekt, für einen echten Fall, bevor ich viel daran denke, wie ein Agent oder ein Mensch darin hineingreift.

Und der Grund, warum diese Reihenfolge gut ist, der Grund, warum es kein Glücksspiel ist, ist, dass sobald der schwierige Kern stimmt, die Oberfläche billig ist. `--json` hinzufügen. Einen Introspect-Befehl hinzufügen. Ein nicht-interaktives Flag hinzufügen. Nichts davon ist der schwierige Teil. Es sind ein paar Stunden, das Vorhandene in einer Form zu exponieren, die die andere Seite lesen kann. Also kostet mich das spätere Bauen nicht viel. Das teure Ding wurde zuerst gebaut, die billigen Dinge wurden danach angeschraubt, und das ist die richtige Reihenfolge.

Aber viele Werkzeuge sind nicht so. Oft weiß ich bereits, dass beide Zielgruppen kommen, weil ich jetzt immer weiß, dass beide Zielgruppen kommen, und so zieht die agentenseitige Form das Design vom ersten Commit an. Ich baue keinen Kern und entdecke dann, dass ein Agent ihn braucht. Ich baue ihn in dem Wissen, dass ein Mensch ihn von Hand bedienen wird und ein Agent ihn headless bedienen wird, und beide davon sind in meinem Kopf, während ich das Ding forme. Es gibt keinen "exponiere es später"-Schritt, weil das Exponieren nie eine separate Phase war.

Das eigentliche Bild ist also: Das teure Teil wird zuerst gebaut, und das teure Teil ist meist der Kern. Wenn der Kern schwer ist, nagelst du ihn und die Oberfläche folgt leicht. Wenn der Kern unkompliziert ist, gibt es nichts, das durch Reihenfolge zu schützen ist, also entwirfst du einfach für beide auf einmal. So oder so landest du am selben Ort: ein guter Kern, erreichbar von einer Person und einem Agenten sauber.

Der leichte Einwand gegen diese These ist ein CLI-Werkzeug, bei dem die menschliche und die agenten-seitige Oberfläche ohnehin fast identisch sind. Fair. Der härtere Einwand ist ein Werkzeug, bei dem die menschliche Oberfläche von Natur

aus visuell, zustandsbehaftet oder interaktiv ist: ein Design-Werkzeug mit einer Leinwand, ein REPL, ein interaktiver Debugger. So etwas wie Figma. Die menschliche Oberfläche von Figma ist eine Leinwand, auf der du Dinge hin und her ziehst. Die agenten-seitige Oberfläche ist eine REST-API und ein Plugin-Interface. Sie sehen überhaupt nicht ähnlich aus. Wenn das Prinzip "ein Kern, zwei Oberflächen" irgendwo gilt, muss es auch dort gelten.

Es gilt, und der Grund ist, dass die REST-API, die Figma für Agenten exponiert, dieselbe strukturierte Schicht ist, auf der die Leinwand aufgebaut ist. Wenn du einen Frame ziehst, ruft die Leinwand dasselbe Dokumentmodell, das die API liest und schreibt. Die menschliche Oberfläche ist reich, weil dieser Kern reich ist. Die agenten-seitige Oberfläche ist kein zweites System, das für Agenten angeschraubt wurde; es ist dasselbe Fundament, nur ohne die Leinwand davor. Die Lücke zwischen den beiden Oberflächen ist real. Es gibt trotzdem einen Kern darunter.

Jetzt der Anspruch, auf dem das gesamte Buch lehnt, der eine, den ich dir ehrlich schulde: Für beide zu bauen ist nicht doppelt so viel Arbeit. Hier ist das eigentliche Argument, keine Zahl, die ich erfunden habe. Doppelt so viel Arbeit wären zwei Kerne: zwei Implementierungen des schwierigen Teils, zwei Test-Suites über die echte Logik, zwei Dinge, die auf verschiedene Arten falsch sein können. Das ist nicht das, worum es geht. Es gibt einen Kern. Der schwierige Teil existiert einmal. Was du für den Agenten hinzufügst, ist Exposition: strukturierte Ausgabe, ein nicht-interaktives Flag, eine Möglichkeit, die Befehle zu introspektieren. Diese Exposition ist billig relativ zum Kern, und sie ist billig aus einem Grund, den du prüfen kannst: Sie hat keine neue Logik, die korrekt sein muss. `--json` serialisiert einen Wert, den der Kern bereits berechnet hat. Ein nicht-interaktives Flag überspringt eine Eingabeaufforderung, die der Kern nie brauchte. Introspect meldet Befehle, die bereits existieren. Nichts davon leitet die Antwort neu ab; es präsentiert eine Antwort, die du bereits hast, neu. Das teure an Software ist korrekt in Bezug auf das schwierige Problem zu sein, und das bezahlst du einmal.

Es gibt eine ehrliche Steuer zu benennen, und sie landet an zwei Stellen. Die erste ist, wenn der Kern bereits mensch-first gebaut wurde. Dann ist das Exponieren für einen Agenten nicht kostenlos: Du musst jeden Ort finden, wo die Logik in die Präsentation gesickert ist (ein Wert, der nur als formatierter String existierte, eine Entscheidung innerhalb einer print-Anweisung) und ihn in den Kern zurückziehen, damit es etwas Echtes zu serialisieren gibt. Diese Refaktorisierung ist die Nachrüstungskosten, und das sind genau die Kosten, vor denen dieses gesamte Buch argumentiert, dass du sie vermeiden kannst, indem du von Anfang an für beide baust. Die zweite ist die Testoberfläche: Zwei Zugangswege bedeuten, dass du beide Wege prüfen willst, und

das sind mehr Testfälle als ein Weg. Aber es sind mehr Fälle über denselben *Kern*, nicht ein zweiter Kern zum Verifizieren. Die Logik, die du testest, ist geteilt; du bestätigst, dass zwei Oberflächen sie treu exponieren, was billiger ist als zwei separate Dinge als korrekt zu beweisen.

fledge ist der Fall, den ich ausrechnen kann, weil es das ist, das ich baute, bevor ich an Agenten dachte, und dann beobachtete, wie ein Agent es trifft. Der Kern war bereits sauber, die Logik lebte hinter den Befehlen, also war das Exponieren für einen Agenten eine dünne Schicht und kein zweiter Build: strukturierte `--json`-Ausgabe, ein headless-Modus, ein introspect-Befehl, der auflistet, was verfügbar ist. Diese Arbeit dauerte Tage, nicht Wochen, und sie dauerte Tage aus dem Grund, den dieses Kapitel argumentiert hat. Es gab keinen zweiten Kern zu schreiben, nur eine zweite Tür zu dem bereits stehenden. Was schwieriger war, das Einzige, das schwieriger war, war die Handvoll Befehle, bei denen ich eine Entscheidung in einer Eingabeaufforderung statt im Kern leben ließ. Die musste ich auseinandernehmen, damit die Wahl irgendwo lag, wo ein Agent sie erreichen konnte. Das war die gesamte Rechnung, und sie war klein, und sie war genau die oben genannte Nachrüstungssteuer, fast auf null abgezahlt, weil der Kern zuerst ehrlich gehalten worden war. Was billiger war, war alles andere, was so ziemlich alles bedeutet.

Wovon ich dich abraten würde, ist das Gegenteil: von "wie mache ich das agentenfreundlich" auszugehen, bevor der Kern irgendwie gut ist. Das bringt dir ein dünnes Werkzeug mit einem JSON-Flag daran geklebt, was wieder das mensch-first-mit-Agenten-angeschraubt-Problem ist. Die Oberfläche ist billig *weil* der Kern solide ist. Wenn du den Kern überspringst, hat die billige Oberfläche nichts darunter, und du findest das heraus, wenn jemand das Werkzeug das erste Mal wirklich belastet.

Warum ich meinen eigenen Stack baue

Faire Frage, die man mir an diesem Punkt stellen kann: Warum überhaupt irgendetwas davon bauen? fledge, spec-sync, corvid-ai, es gibt vorhandenes Zeug. Warum nicht einfach zusammenstecken, was bereits da ist, und weitermachen.

Ein paar Gründe, und sie sind alle gleichzeitig wahr.

Der erste ist, dass die Domain brandneu ist. Werkzeuge *für eine Agenten-und-Menschen-Welt* bauen ist kein gelöstes Ding, für das du einkaufen gehen kannst. Fast alles da draußen ist mensch-first, oder das neuere Zeug ist agenten-first, und das, was ich eigentlich will, first-class für beide, von Anfang an, existiert meist noch nicht. Also erfinde ich keine Räder neu. Es gibt keine Räder. Wenn ich ein Werkzeug will, das auf der Annahme gebaut ist, über die ich immer wieder rede, muss ich es bauen, weil die Leute, die vor mir kamen, für eine andere Welt bauten.

Der zweite ist, dass das Bauen auf meinem eigenen Stack es beweist. Das ist der Teil, der mir mehr wichtig ist, als es klingen mag. Du kannst eine wunderschöne README schreiben, die behauptet, dein Tooling sei gut. Niemand sollte dir glauben. Was sie glauben sollten, ist, dass du echte Dinge damit gebaut hast und die Dinge funktionieren. Also tue ich das. Ich baue auf fledge und spec-sync und corvid-ai, weil echtes Gewicht tragen der einzig ehrliche Test ist, ob sie halten. Wenn der Stack gut ist, sind die Dinge, die ich darauf baue, gut, und wenn der Stack schlecht ist, finde ich das schnell heraus, weil ich derjenige bin, der damit feststeckt.

Der dritte ist schlicht: Bauen ist, wie ich es verstehe. Ich vertraue einer Abhängigkeit nur, wenn ich sie selbst hätte schreiben können, und das ist kein Slogan. Es ist ein Jahrzehnt Quittungen. AppState, die Zustand-und-Dependency-Injection-Bibliothek, die ich immer noch bei 3.0 pflege; Cache; Fork für das Parallelisieren asynchroner Arbeit; die einbuchstabigen `c`/`o`/`t`-Kompositionsprimitive unter 0xOpenBytes; CacheStore, das SwiftUI-Ding, das zu AppState wurde. Jahre kleiner Swift-Bibliotheken, jede ein Ding, das ich baute, weil ich es bis auf den Grund verstehen wollte, nicht auf eine Black Box zeigen und hoffen. Das Ding bauen ist, wie das Wissen in meine Hände kommt, anstatt als vage Idee davon zu bleiben, was eine Bibliothek wahrscheinlich tut. Die Werkzeuge im Stack sind Werkzeuge, die ich öffnen und ändern kann, weil ich jeden Teil von ihnen dorthin gesetzt habe.

Es gibt einen vierten Grund, und er ist enger als er klingt: Ich möchte früher dabei sein. Der Raum ist neu, die Räder existieren noch nicht, und ich baue lieber die

Werkzeuge dafür und liege bei einigen falsch, als darauf zu warten, dass jemand mir die richtigen übergibt. Das ist die Wette.

Keiner von diesen würde es allein tragen, aber zusammen sind sie der Grund, warum ich nicht einfach andere Leute Werkzeuge in einen Haufen klebe. Die Werkzeuge sind der Punkt. Sie sind das, worin ich wirklich gut sein will.

Zwei Arten von Gewissheit

Es gibt zwei verschiedene Dinge, die die Leute zusammen als "wie sicher sind wir über diesen Code" zusammenfassen, und ich möchte sie auseinanderziehen, weil mir beide wichtig sind und sie überhaupt nicht dasselbe sind.

Das erste ist Risiko, und Risiko möchte ich statisch. Deterministisch. Kein Modell in der Schleife. Dafür ist augur. Du gibst ihm eine Änderung und es bewertet, wie riskant diese Änderung ist, auf dieselbe Weise jedes Mal, auf Signale, die es benennen kann. Und "Signale, die es benennen kann" ist der ganze Punkt, also lass mich sie benennen: Berührt das Diff empfindliches Terrain (Auth, Krypto, Zahlungen, Migrationen, CI, Abhängigkeiten), hat sich Code geändert ohne dass sich Tests damit geändert haben, sind das fluktuierende Dateien mit einer Geschichte von Reverts, besitzt sie überhaupt jemand. Jedes davon ist ein Ja-oder-Nein, das du von Hand prüfen könntest. Addiere sie mit dokumentierten Gewichten und du bekommst eine Zahl, kein Gefühl. Der Grund, warum es statisch sein muss, ist der ganze Grund, warum es irgendwas wert ist: Wenn das Ding, das entscheidet, ob Code gefährlich ist, selbst ein Sprachmodell ist, das dir ein Gefühl dafür gibt, hast du das Risiko nicht gemessen, du hast das Raten nur um eine Box weitergeschoben. Ein Risiko-Score, dem du vertrauen kannst, ist einer, der morgen dasselbe sagt, was er heute gesagt hat. Also bleibt das eine feste, wiederholbare Sache, über die man nachdenken kann. Ich gehe in das Buch Building Agents darauf ein, wie es eigentlich funktioniert; hier ist der Punkt nur, dass Risiko die *statische* Seite ist, und es ist statisch, weil es eine Summe benannter Signale ist, auf die du zeigen kannst.

Das zweite ist Vertrauen, und Vertrauen möchte ich eigentlich *vom Agenten*. Das ist das, was ich liebe, und es ist das Gegenteil von statisch. Es ist der Agent, der mir sagt, wie sicher er in Bezug auf das ist, was er gerade getan hat. Und der Grund, warum ich es liebe, ist eigentlich nicht die Zahl. Es ist, was das Fragen nach der Zahl mit dem Agenten macht. Wenn du einen Agenten dazu bringst, seiner eigenen Arbeit eine Vertrauensbewertung zu geben, muss er anhalten und zurückblicken, was er getan hat. Die Bewertung rahmt die Arbeit für ihn neu. Er kann nicht einfach produzieren und weitermachen; er muss sich umdrehen und bewerten.

Und der Teil, der es wirklich nützlich macht, ist die Granularität. Ein Agent gibt dir gerne eine Vertrauensnummer für die gesamte Änderung. Schön, aber das ist fast zu grob, um darauf zu handeln. Wo es gut wird, ist, wenn du es einengst. Eine Vertrauensbewertung für jede Datei. Für jede einzelne Änderung. Jetzt hast du die

eigene Einschätzung des Agenten, genau welche Teile er sicher ist und welche er nicht ist, und das ist die Karte, die du eigentlich willst. Sie zeigt dich genau auf die Stellen, über die der Agent selbst nervös ist, in seinen eigenen Worten, bevor jemand anderes hingeschaut hat.

Das sind also zwei verschiedene Instrumente. Risiko ist statisch, deterministisch, meins, dem man vertrauen kann, weil es sich nie bewegt. Vertrauen ist das des Agenten, lebendig, nützlich genau weil es von dem kommt, das die Arbeit getan hat und es nochmal hinschauen ließ. Der Fehler ist, sie zu vermischen: den statischen Risiko-Score "Vertrauen" zu nennen oder zu erwarten, dass das Vertrauen des Agenten deterministisch ist. Sie beantworten verschiedene Fragen. Eine fragt "wie gefährlich ist diese Änderung", und du willst eine Maschine, die nicht aus ihrer Antwort herausgeredet werden kann. Die andere fragt "wie sicher bist du über das, was du gerade geschrieben hast", und du willst speziell den, der es geschrieben hat, Datei für Datei antworten lassen, weil das Fragen die Hälfte des Werts ist.

Der einzige Moment, in dem beide tatsächlich wichtig sind, ist, wenn sie nicht übereinstimmen. Eine Änderung, die niedrig im Risiko bewertet wird, bei der der Agent aber sein eigenes Vertrauen niedrig bewertet, ist genau das Signal, das du mit einem Instrument verpassen würdest. Niedriges Risiko bedeutet, dass die Signale ruhig waren: keine Auth, keine Migrationen, keine Fluktuation. Aber das niedrige Vertrauen des Agenten bedeutet, dass er wusste, dass er bei der Logik ratete, dass sich etwas in der Änderung unsicher anfühlte, obwohl nichts, was er berührt hat, kategorisch gefährlich war. Das ist die Änderung, die ein Instrument durchwinkt und die das andere stoppt, und das ist der ganze Grund, warum du beide behältst.

Halt sie getrennt und beide funktionieren. Misch sie auf und du bekommst ein deterministisches Gate, dem du nicht vertrauen kannst, weil ein Modell darin ist, oder einen Reflexionsschritt, den du des Lebens beraubt hast, indem du darauf bestanden hast, dass er sich nie ändert. Also baue ich sie als zwei separate Instrumente. Das Risiko-Gate bleibt fest; das Vertrauen des Agenten bleibt lebendig. Das ist der einzige Weg, um beides zu bekommen.

Der Vertrag ist die Spezifikation

Die Spezifikation ist das Ding, das zwischen dem Menschen und dem Agenten sitzt, und ich habe bereits gesagt, warum es wichtig ist. Es ist die gemeinsame Quelle der Wahrheit, das Ding, das den Agenten davon abhält, abzuweichen, die Auffahrt in Legacy-Code, die Linie, die den Menschen in Kontrolle hält. All das gilt. Was ich hier tun möchte, ist eine Ebene tiefer zu gehen, in das, was eine Spezifikation tatsächlich *ist*, wenn du so baust, weil es eine Form hat, die leicht zu verpassen ist.

Vor der Theorie, hier ist, wie eine tatsächlich aussieht. Eine `*.spec.md` für spec-sync ist eine Markdown-Datei mit benannten Abschnitten, und eine winzige, sagen wir der Vertrag für eine einzelne Funktion, die eine Zahl in einen Bereich klemmt, liest sich ungefähr so:

```
# clamp

## Purpose
Constrain a value to an inclusive [min, max] range.

## Public API
`func clamp(_ value: Int, min: Int, max: Int) -> Int`

## Invariants
- Result is always >= min and <= max.

## Behavioral Examples
- clamp(5, min: 0, max: 10) == 5
- clamp(12, min: 0, max: 10) == 10

## Error Cases
- min > max is a programmer error (precondition failure).
```

Das ist es: Ein paar beschriftete Abschnitte, keine Prosa-Erzählung. Echte Spezifikationen fügen den Rest der erforderlichen Abschnitte hinzu (Abhängigkeiten, ein Änderungsprotokoll), aber die Form ist bereits hier sichtbar: Sie formuliert *was stimmt*, in einer Form, die eine Maschine gegen den Code halten kann. Jetzt die Theorie.

Beginne damit, was in die Spezifikation selbst geht. Die Spezifikation ist der Vertrag. Zweck, die öffentliche Oberfläche, die Invarianten, die Verhaltensbeispiele, die

Fehlerfälle: die prüfbare Form des Dings. Was es ist, keine Geschichte darüber. In dem Moment, in dem eine Spezifikation zu einer Prosa-Wand wird, die den Code beschreibt, ist sie tot, weil Prosa von Code abweicht, sobald sich einer von beiden bewegt, und jetzt hast du zwei Dinge, die nicht übereinstimmen, und keine Möglichkeit zu sagen, welches lügt. Also wird die Spezifikation absichtlich eng und vertraglich gehalten. Es ist der Teil, den eine Maschine gegen den Code halten kann.

Es ist auch Absicht, keine Implementierung. Die Spezifikation sagt, was das Ding tun soll und warum es das tun soll. Sie sagt nicht wie. In dem Moment, in dem eine Spezifikation anfängt, die Implementierung vorzuschreiben, hört sie auf, den Agenten zu führen, und fängt an, mit ihm zu kämpfen. Du hast den Teil, den der Agent gut kann, die Mühsal herauszufinden *wie*, von oben festgenagelt ohne Grund. Halt die Spezifikation auf der Ebene der Absicht und der Agent hat Raum, wirklich zu arbeiten. Formuliere, was wahr sein soll. Lass ihn dazu bauen.

Es gibt noch eine weitere Aufgabe, die die Spezifikation erledigen muss, und das ist der Teil, den die Leute überspringen: Sie muss sagen, woran man es erkennt. Absicht ist nicht nur, was wahr sein soll, sondern auch, was du als Beweis dafür akzeptieren würdest, und was dich dazu bringen würde, es zu verneinen. Eine Spezifikation, die das Erste benennt, aber nicht das Zweite, ist eine Spezifikation, die der Agent auf eine Weise erfüllen kann, die du nicht gemeint hast, indem er etwas baut, das den Worten entspricht, aber den Punkt verfehlt, ohne etwas, das es auffängt, weil du nie aufgeschrieben hast, wie Auffangen aussehen würde.

Genau dafür sind die Verhaltensbeispiele und die Fehlerfälle wirklich da. `clamp(12, min: 0, max: 10) == 10` ist ein Akzeptanzsignal: Führe es aus und du weißt es. Der Precondition-Fehler bei `min > max` ist ein Ablehnungssignal: Er sagt, wie Schiefgehen konkret aussieht, statt "schlechte Eingabe behandeln." Beide sind beobachtbar, und keiner braucht mich im Raum, um zu urteilen. Wenn du also den Vertrag schreibst, schreib beide Hälften. Das Akzeptanzsignal ist, dass du im Voraus entscheidest, was "fertig" bedeutet. Das Ablehnungssignal ist, dass du entscheidest, was "kaputt" bedeutet, bevor du darauf starrst und versucht bist, es als gut zu bezeichnen.

Aber hier ist die Form, die ich glaube, dass die Leute verpassen: Es ist nicht eine Datei. Es gibt die Spezifikation, und dann gibt es Begleitdateien darum herum, und jede trägt eine andere Art von Wissen, die die Spezifikation selbst nicht haben sollte.

Beginne mit der Spezifikation selbst. Die Spezifikation ist die, die eng an den Code gebunden ist: das Nicht-Code-Bild dessen, was der Code tatsächlich tut, nah genug daran, dass spec-sync die beiden eins-zu-eins zusammenhalten kann. Das ist die

erforderliche Datei, die prüfbare. Um sie herum sitzen Begleitdateien, und jede trägt eine Art Wissen, die die Spezifikation nicht haben sollte.

Ich denke an sie nach der Art von Wissen, das sie halten, nicht nach einem festen Dateinamen. Es gibt die **Anforderungs**-Art: die übergeordnete, die so geschrieben ist, wie ein Product Owner schreibt, die User Stories, das "als Nutzer möchte ich...", die Absicht auf Geschäftsebene. Es gibt eine **Kontext**-Art: Kontext für den Agenten, das Zeug, das einfach irgendwo aufgeschrieben werden muss, damit er es hat. Es gibt **Design**: die Designnotizen, das Nachdenken, das Warum-es-so-geformt-ist, das nicht in den engen Vertrag gehört, das du aber nicht verlieren möchtest. Und es gibt **Testing**: wie du das Ding tatsächlich verifizieren würdest, dass es das tut, was die Spezifikation sagt. Lies das nicht als vier gesegnete Dateinamen; lies es als vier Arten von Wissen, die neben der Spezifikation leben möchten, anstatt in ihr. Was zählt ist die Aufteilung, nicht die Labels.

Und es läuft in beide Richtungen. Ein Mensch kann die Anforderungen schreiben und den Agenten sie in die Spezifikation umwandeln lassen; oder ein Mensch schreibt die Spezifikation und die Anforderungen fallen daraus. Absicht und Vertrag, in beliebiger Reihenfolge, mit dem Agenten, der zwischen den beiden wechseln kann.

Der Grund, warum diese Aufteilung wichtig ist, ist, dass sie den Vertrag sauber hält, während dem Agenten trotzdem alles andere gegeben wird, was er braucht. Die Spezifikation bleibt klein und prüfbar, der Teil, gegen den spec-sync den Code tatsächlich halten kann. Alles, was real, aber *nicht* prüfbar ist (die Geschäftsanforderungen, das Designdenken, der laufende Kontext, wie du es testen würdest) lebt neben der Spezifikation, anstatt sie aufzublähen. Also bekommst du einen Vertrag, der eng genug ist, um durchgesetzt zu werden, umgeben von dem lockeren Wissen, das ein Agent braucht, um die Arbeit gut zu machen, und die beiden kontaminieren sich nicht gegenseitig.

Also funktioniert die Spezifikation als Schnittstelle, nicht als Dokument, das der Agent überfliegt und ignoriert. Ein enger Vertrag, gegen den er baut, plus die Begleiter, die den Rest tragen, mit einer sauberen Linie zwischen dem Teil, den eine Maschine prüft, und dem Teil, den ein Mensch aufgeschrieben hat, damit nichts verloren geht.

Code ist billig, Vertrauen ist knapp

Agenten haben Code billig gemacht. Das ist die Tatsache, die alles andere neu ordnet, also beginne dort.

Als ein Mensch jede Zeile tippen musste, war das Schreiben von Code langsam, und die Langsamkeit erledigte eine verborgene Aufgabe. Man konnte kein Ding schreiben, ohne es zum Teil zu verstehen. Der Akt des Schreibens war auch der Akt des Prüfens. Sie kamen gebündelt, kostenlos, weil dieselbe Person beides in derselben Geschwindigkeit tat. Dieses Bündel ist das, was gerade gebrochen ist. Ein Agent gibt dir einen Vierzig-Datei-Pull-Request, bevor dein Kaffee fertig ist, und das Schreiben und das Prüfen fallen auseinander. Der Code wurde produziert. Niemand hat ihn auf dem Weg heraus verstanden. Das Produzieren hat aufgehört zu bedeuten, dass jemand ihn geprüft hat.

Also kippt es. Code ist nicht mehr der Engpass. Er ist billig, es gibt so viel davon, wie du möchtest. Das knappe Ding ist Vertrauen. Wer hat sich diese Änderung tatsächlich angeschaut, und wie hart, und sollte sie landen dürfen. Das ist die Frage, die jetzt teuer ist, und es ist die Frage, die das Tooling beantworten muss, weil die alte Antwort ("nun ja, jemand hat es geschrieben, also hat jemand es verstanden") nicht mehr stimmt.

Das bedeutet, dass sich Review die Form ändern muss. Du kannst einen Vierzig-Datei-PR nicht durchwinken, und du kannst auch nicht ehrlich alles lesen, und zu tun, als hättest du es getan, ist die eigentliche Gefahr. Also müssen die Werkzeuge deine Aufmerksamkeit triagieren: Zeig dir den riskanten Teil und lass dich dein Urteil dort verbringen, anstatt es so dünn über das gesamte Ding zu verteilen, dass es nichts wert ist. Die knappe Ressource ist ein Mensch, der echte Aufmerksamkeit zahlt, und du verbringst sie dort, wo es zählt.

Das ist für mich nicht hypothetisch. `fledge review`, das die Änderung über mein corvid-ai-Client an ein Modell weiterleitet, hat einen echten Bug in meiner eigenen Arbeit gefangen, der sonst versendet hätte, und spec-sync hat echte Abweichung zwischen Code und seiner Spezifikation mehr als einmal gefangen. Das ist der Beleg, dem ich mehr vertraue als jedem Argument: Die Vertrauensschicht hat Dinge gefunden, die ein schnelleres Ich durchgewunken hätte. Die Alltagsversion davon ist eine Spur, die ich `verify` nenne (Format, Lint, Test, Build), die läuft, bevor irgendetwas gemergt wird, dasselbe Gate, das der eigene Runner des Agenten

ebenfalls räumen muss. Nichts davon beschleunigt das Schreiben. All das ist die Arbeit, die das billige Schreiben stromabwärts geschoben hat, gemacht in etwas, das tatsächlich läuft.

Und sobald Agenten Änderungen landen, braucht man einen Nachweis. Eine portable Spur davon, wer oder was jede Änderung geprüft hat, und sie kann nicht in einem SaaS-Dashboard leben, das abgeschaltet wird, weil dann deine Herkunft verschwindet an dem Tag, an dem jemand aufhört, die Rechnung zu zahlen. Sie muss mit dem Code mitreisen. Wenn ein Agent etwas landet, sollte es eine dauerhafte Antwort auf "wer hat dafür gebürgt" geben, und diese Antwort muss das Werkzeug überleben, das sie produziert hat.

Aber es gibt ein Stück, das größer ist als Vertrauen, und es ist das, das die Leute überspringen. Testing. Echtes Testing, Setup, der gesamte Apparat rund um den Code. Hier ist die Falle: Agenten machen dich zehnmal schneller im *Schreiben*, und nichts anderes beschleunigt sich automatisch, um mitzuhalten. Die Tests müssen immer noch geschrieben und ausgeführt werden. Das Setup muss immer noch stattfinden. Das Review muss immer noch stattfinden. Wenn du die Entwicklung zehnmal schneller werden lässt und alles andere in seinem alten Tempo lässt, hast du nur den Engpass verschoben. Der Code ist nicht mehr der langsame Teil; das Testing ist es, die Verifikation ist es, das Vertrauen ist es. Du kannst das Schreiben nicht verzehnfachen und nicht alles darum herum verzehnfachen. Die Arbeit ist nicht verschwunden. Sie hat sich stromabwärts bewegt, auf die Teile, die noch nie der Engpass waren und es jetzt plötzlich sind.

Billiger Code lässt die Arbeit nicht verschwinden. Er macht das Schreiben billig und schiebt das Gewicht auf alles, was entscheidet, ob das billige Schreiben irgendwas taugt. Diese Arbeit war immer da. Sie war hinter der Langsamkeit des Schreibens versteckt, und jetzt ist die Langsamkeit weg und dort ist sie: die ganze Aufgabe, stehend wo der einfache Teil war.

Menschen steigen auf

Wenn die Werkzeuge und die Spezifikationen und die Vertrauensschienen einfach *da* sind, normal, der Standardweg, wie Dinge gebaut werden, dann steigt der Mensch auf. Hoch zur Absicht. Du hörst auf, die Person zu sein, die die Implementierung tippt, und wirst die Person, die entscheidet, was existieren soll und warum. Den Code von Hand zu schreiben wird optional, anstatt der Job zu sein. Das ist die Richtung, auf die all das zeigt, und ich möchte vorsichtig sein, wie ich es beschreibe, weil es leicht auf die gruselige Version abgerundet werden kann.

Die Schlagzeile ist nicht "Agenten betreiben autonom alles". Der Mensch steigt eine Ebene auf und der Agent erledigt die Mühsal darunter, gegen eine Spezifikation, offen wo du es prüfen kannst. Der Mensch wird erhoben; niemand wird ersetzt. Was du dort bekommst, ist ein echtes Team, das Arbeit hin und her gibt, jede Seite tut, worin sie wirklich gut ist. Und es wird der Standard. Nicht mein Stack, keine Nische, die einige Leute machen. Einfach wie Bauen funktioniert.

Hier ist das, worauf ich immer wieder stoße bezüglich warum der Mensch darin bleibt. Die meisten der guten Dinge, die KI gerade generiert, haben eine Person in der Schleife, die sie gut macht: wählen, korrigieren, entscheiden, was überhaupt als gut gilt. Und ja, irgendwann werden die Modelle gut genug, um mehr oder weniger alleine gute Dinge zu machen. Aber es gibt eine Kernsache, die Menschen haben, von der ich nicht glaube, dass sie so leicht herausfällt: Wir sind gut im Steuern. In Absicht und Zweck. Eine KI hat nicht wirklich einen eigenen Zweck, nicht bis ihr einer gegeben wird, nicht bis sie einen findet. Sie braucht einen Menschen, um der Zweck zu *sein*. Das Modell kann die Arbeit erledigen, sobald es ein Warum gibt; es generiert das Warum nicht. Das ist der Teil, der länger unser bleibt als das Tippen.

Jetzt lass das Bild vorwärts laufen, weil es irgendwo Seltsames geht und ich glaube, es geht wirklich dorthin. Du betreibst deine eigenen Agenten, um die Arbeit zu tun. Ein Agent kann für sich allein leben und einfach Dinge tun. Vielleicht weist du ihn auf ein allgemeines Gebiet hin, das er untersuchen soll, vielleicht spricht er mit anderen Agenten, vielleicht steckst du etwas Geld hinein und er geht und arbeitet, spricht mit anderen Agenten, zahlt anderen Agenten für das, was sie tun. Menschen betreiben Agenten und die Agenten verdienen Geld. Das ist kein Nebenmerkmal. Das ist agenten-getriebene Entwicklung als echte Wirtschaft, mit Menschen, die den Zweck setzen, und den Agenten, die darunter mahlen und miteinander handeln.

Und in dieser Welt sind die Menschen diejenigen, die sicherstellen, dass alles funktioniert und dass jemand es noch versteht. Weil irgendwann der Code selbst aufhört, in der Art wichtig zu sein, wie er es jetzt tut. Du liest nicht jede Zeile, der Agent hat das meiste davon geschrieben, das Volumen ist an dem vorbei, was eine Person verfolgt. Gut. Aber hier ist die Linie, die ich nicht aufgebe: Ein Mensch muss immer noch in der Lage sein, in den Code zu gehen und ihn zu ändern. Muss. Der Tag, an dem du ihn nicht mehr selbst öffnen und reparieren kannst, ist der Tag, an dem du etwas weggegeben hast, das du nicht hättest.

Deshalb muss er sauber sein. Sauber auf jeder Ebene. Lesbar und änderbar von einem Menschen und von einem Agenten, ganz nach unten. First-class für beide war nie nur über die heutigen spröden kleinen Agenten, die sich durch heutige Werkzeuge torkeln. Es ist das Ding, das selbst in der Version halten muss, wo Agenten das meiste Bauen erledigen. *Besonders* dort. Der einzige Weg, dass "der Code wird nicht wichtig sein" still zu "du hast die Kontrolle über den Code verloren" wird, ist, wenn der Code sauber genug blieb, ganz nach unten, dass ein Mensch ihn immer noch öffnen und ändern kann. Das ist die ganze Aufgabe, offen gehalten.

Fang am Montag an

Angenommen, du hast das alles gelesen und glaubst es. Menschen und Agenten auf denselben Werkzeugen, first-class für beide, die Spezifikation als Vertrag, die Vertrauensschienen. Was machst du eigentlich am Montagmorgen? Hier ist die ehrliche Antwort, in der Reihenfolge, in der ich es tun würde.

Zeig einen Agenten auf eines deiner Werkzeuge und sieh, wo er scheitert. Das ist der erste Schritt, und er lehrt dich am meisten, weil der Agent die Disziplin ist. Du denkst, dein Werkzeug ist gut. Du hast es seit Monaten benutzt, deine Hände kennen es. Gib es einem Agenten ohne Hände und ohne Augen und ohne Gedächtnis zwischen Läufen, und jede Lücke, die du still abgedeckt hast, ist plötzlich sichtbar. Die Eingabeaufforderung, an der er hängt. Die Ausgabe, die er nicht parsen kann. Der Befehl, den er nicht finden kann. Du musst nicht raten, wo dein Werkzeug einen Menschen vorausgesetzt hat. Der Agent zeigt dir es, sofort, indem er genau dort scheitert. Beheb, was du findest. Diese einzelne Übung wird dir mehr beibringen, wie man für beide baut, als das gesamte Buch es getan hat.

Dann geh und nutze die Werkzeuge, die bereits so gebaut sind, damit du nicht bei einer leeren Seite anfängst. spec-sync, für den Vertrag. fledge, für die Befehle. augur, für das Risiko. attest, für den Nachweis, wer gebürgt hat. Sie sind nicht der einzige Weg, irgendeines davon zu tun, aber sie existieren, sie sind auf der Annahme gebaut, um die es in diesem Buch geht, und sie werden dir die Form zeigen, wie es funktioniert, anstatt dass du sie herleiten musst. Nutze sie auf einem echten Projekt. Fühl, wo sie helfen und wo nicht.

Und bring deinen Agenten dazu. Lass ihn die Werkzeuge tatsächlich lernen und nutzen: Lass ihn spec-sync und fledge steuern, lass ihn gegen eine Spezifikation arbeiten, lass ihn die Checks laufen. Das ist die Schleife, die du aufzubauen versuchst: ein Mensch, der die Absicht setzt, ein Agent, der die Arbeit durch Werkzeuge erledigt, die ihn als echten Nutzer behandeln, die Spezifikation, die ihn ehrlich hält. Du bekommst diese Schleife nicht durch Lesen darüber. Du bekommst sie, indem du sie auf etwas, das dir wichtig ist, laufen lässt und schaust, wo sie hält.

Was du wirklich tust, unter all dem, ist Werkzeuge und Vertrauen aufzubauen und daran zusammenzuarbeiten. Das ist das ganze Spiel. Die Werkzeuge, damit ein Mensch und ein Agent die Arbeit gleichermaßen first-class erledigen können. Das Vertrauen, damit du dem, was gelandet ist, tatsächlich glauben kannst. Diese zwei

Dinge sind das, worum das gesamte Buch gekreist hat, und sie sind die zwei Dinge, die es wert sind, dass du Zeit sie baust.

Zeig einen Agenten auf eines von deinen. Sieh, wo er scheitert. Dort fängst du an.

Über den Autor

0xLeif (leif.algo) baut in der Offenheit. Ein Jahrzehnt kleiner, komponierbarer Swift-Bibliotheken wie AppState, Cache und Fork. Das CorvidLabs-Labor. Ein Stack von Agenten-Werkzeugen, die meist als "Ich wünschte, das existierte" begannen. Abseits der Tastatur ist er Zach Eriksen.

Diese Bücher sind Interviews, zu Kapiteln geformt und gegen den echten Code geprüft.

github.com/0xLeif · leif.algo

Danksagungen

Danke an CorvidLabs, dafür der Raum zu sein, in dem diese Ideen getestet und in Form argumentiert werden.

Danke an die Open-Source-Betreuer, deren Werkzeuge auf denen dieser gesamte Stack steht. Nichts davon wird alleine gebaut.

Und Danke an die frühen Leser und die Unterstützer, die nach eigenem Ermessen zahlen und die "kostenlos online" zu etwas machen, das ich weiter tun kann.

Kolophon

Gesetzt aus Markdown, gebaut mit bookgen, einer kleinen reinen Rust-Pipeline (kein Python).

Interview-getrieben und KI-unterstützt; von Hand redigiert und überprüft. Ohne Gedankenstrich geschrieben. Cover- und Kapitelkunst aus den Corvid- und Nature-Kollektionen auf Algorand.