# Open Source Tooling

Building tools people actually use

ZACH "LEIF" ERIKSEN

# Copyright



Kostenlos online lesbar. Das ePub ist Pay-what-you-want; 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 im Offenen bauen und es trotzdem rausschicken.*

# Die Bibliothek

Diese Bücher stehen für sich, 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 es vertrauenswürdig macht.

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

Kostenlos online lesbar. Jedes ePub ist Pay-what-you-want.

---

# Inhalt



---

# Einleitung

Dieses Buch handelt vom Handwerk, Tools zu bauen, die andere Menschen, und andere Agents, tatsächlich benutzen.

Es dreht sich um fledge, ein einziges Command-Line-Tool, das den gesamten Entwicklungslebenszyklus abdeckt, und die Frage, die es mich immer wieder zwang zu beantworten: Was macht ein Tool vertrauenswürdig genug, um davon abzuhängen? Die These ist, dass die Antwort für einen Menschen und für einen Agent dieselbe ist. Ein gutes Tool hat eine saubere Oberfläche, keine versteckten interaktiven Schritte, eine echte Plugin-Grenze und eine Spec, die ehrlich bleibt, während sich der Code verändert. Bau das, und ein Agent kann es vom ersten Tag an steuern, weil es von Anfang an nicht so gebaut wurde, dass es einen Menschen braucht.

Dies ist das zweite der zwei Beweisbücher. *First-Class* argumentiert, dass Software für beide erstklassig sein sollte. Der *Field Guide* macht daraus eine Methode. *Building Agents* zeigt die Agents. Dieses zeigt das Tooling darunter, bis hin dazu, warum Rust die richtige Wahl war und wie die WASM-Sandbox einen Plugin davon abhält, etwas zu tun, was er nicht sollte.

Es richtet sich an alle, die jemals ein Tool rausgeschickt haben und zusehen mussten, wie es auf Weisen genutzt wurde, die sie nicht geplant hatten. Du brauchst fledge nicht. Du brauchst die Gewohnheiten, die ein Tool überleben lassen, wenn es auf Nutzer trifft, die nicht du bist, einschließlich derer, die nicht mal menschlich sind.

---

# Eine CLI für den gesamten Lebenszyklus

Jedes Repo hatte einen anderen Dialekt. Verschiedene Makefiles, verschiedene Skripte, verschiedene READMEs, jedes mit seiner eigenen Vorstellung davon, wie man die Sache baut, testet und ausführt. Nichts davon ließ sich übertragen. Du öffnest ein Projekt, das du eine Weile nicht angefasst hattest, und die erste Aufgabe war, die lokale Beschwörungsformel wieder herauszufinden. Welches Skript, welches Target, welche Reihenfolge. Die eigentliche Arbeit war nicht schwer. Tests ausführen ist nicht schwer. Das Problem war, dass es überall anders war, und der Unterschied war reiner Overhead.

Also wollte ich eine konsistente Schnittstelle über alle hinweg. Da fängt fledge an. Die Zeile im Repo sagt es direkt: *one CLI, your whole dev lifecycle.*

Es gab wirklich drei Dinge, die mich dazu drängten.

Das erste war das Gleichheitsproblem, jedes Repo spricht seine eigene Sprache. Das zweite war Bootstrapping. Ich kopierte dasselbe Setup und Scaffolding immer wieder in jedes neue Projekt. Derselbe Haufen Dateien, dieselbe Verdrahtung, immer und immer wieder, bevor ich überhaupt mit der eigentlichen Sache anfangen konnte, die ich bauen wollte. Das dritte war Skalierung. Ich starte massenhaft Projekte, mit Agents, die daran arbeiten, und ich brauchte eine einzige CLI, auf die sie sich alle verlassen können. Nicht "ein Tool, das ich benutze." Ein Tool, auf das alles und jeder, der über all diese Repos arbeitet, sich auf dieselbe Weise verlassen kann.

Das letzte ist leicht zu übersehen. fledge entstand nicht aus einem Repo, das mich geärgert hat. Es entstand daraus, viele Repos zu haben, viele laufende Projekte, mit Agents in der Schleife, und all das brauchte eine einheitliche Oberfläche statt hundert spezialisierte. Der Agents-können-sich-auf-eine-CLI-verlassen-Winkel ist ein eigener Faden, und ich gehe auf die Agent-Seite davon im Agents-Buch ein. Konsistenz, Scaffolding und Skalierung: das ist der Ursprung.

## Für Menschen und Agents gleichermaßen gebaut

Hinter alldem steckt eine Idee, die größer als fledge ist. Ein Großteil meines Toolings kommt von der Überzeugung, dass Menschen und Agents dieselben Tools benutzen werden.

Momentan ist das meiste, was existiert, human-first. Projekte werden für Menschen gebaut, und dann versuchen wir, Agents nachträglich einzubringen. Es kann agent-first- Sachen geben und human-first-Sachen, aber was wir eigentlich brauchen, ist agent-und- human-first-Projekte zu machen, von Anfang an so gebaut, dass beide erstklassig sind.

Das sind alle meine Tools. Ein Tool sollte in beide Richtungen erstklassig funktionieren: ein Mensch kann es ohne Agent benutzen, und ein Agent kann es ohne Menschen benutzen. Keiner von beiden ist ein Nachgedanke. Und wenn ein Agent es benutzt, sollte das Tool ihm *helfen*, nicht ihn im Dunkeln tappen lassen, was all die Befehle sind und wie alles funktioniert.

Für Menschen und Agents als gleichwertige erstklassige Aufrufer zu entwerfen ist der eigentliche Grund, warum fledge so aussieht, wie es aussieht, und das ist das Ding, das man für den Rest dieses Buches im Kopf behalten sollte.

## Was Zero-Config wirklich bedeutet

Wenn ich Zero-Config sage, meine ich, dass du dem Tool nicht erst etwas über dein Projekt beibringen musst, bevor es dir helfen kann. Du droppst es in ein Repo und es funktioniert. Das hat drei Teile.

Es erkennt das Projekt und seine Befehle automatisch. Swift, Rust, Node, was auch immer. Es findet heraus, was für ein Projekt es sich ansieht, und kennt das richtige Build/Test/Run dafür. Kein Setup-Schritt, bei dem du dein Projekt zuerst in einer Config-Datei beschreibst.

Dieselben Verben funktionieren in jedem Projekt. `build`, `test`, `run`, `lint`, dieselben Worte, egal was darunter liegt. Das ist die direkte Auszahlung des "jedes Repo hatte einen anderen Dialekt"-Problems. Die Dialekte sind noch da unten; Cargo ist immer noch Cargo und Swift Package Manager ist immer noch Swift Package Manager. Aber du musst nicht mehr wissen, vor welchem du gerade stehst. Du lernst die Verben einmal und sie sind die Verben überall. Die Erkennung übernimmt die Übersetzung hinunter zu dem, was das eigentliche zugrundeliegende Tool auch immer ist.

Das ist die Menschen-und-Agents-Idee konkret gemacht. Eine Person muss sich nicht mehr in jeden Repo neu einarbeiten. Ein Agent muss nicht mehr *raten*. Er muss nicht nach dem richtigen Befehl für dieses bestimmte Projekt suchen oder die README lesen und hoffen. Er führt `build` aus, er führt `test` aus, und das Tool weiß bereits, was das hier bedeutet. Das Ding, das mir das Wiedereinarbeiten erspart, ist dasselbe, das den Agent vom Raten abhält. fledge wurde zuerst für mich und die Agents gebaut, die ich betreibe, und verdient seinen Wert dort, bevor es irgendwo anders Wert zeigt.

Und es wird über Plugins erweitert. Ein frisches Repo bekommt den Core, und du fügst Fähigkeiten hinzu, indem du Plugins einwirfst. Der Core kennt die gängigen Verben und wie er die gängigen Projekttypen erkennt; alles darüber hinaus kommt von Plugins. Zero-Config bedeutet also nicht "macht alles out of the box." Es bedeutet, dass der Teil, den du out of the box bekommst, keine Konfiguration braucht, und du es von dort aus erweiterst, indem du Plugins hinzufügst, nicht indem du Config schreibst.

Konkret läuft ein erster Run so ab. Du wechselst in ein Repo und führst `fledge` aus, es erkennt das Projekt automatisch (Swift, Rust, Node, was auch immer) und kennt einfach das richtige Build/Test/Run dafür, ohne Config-Schritt. Bitte es um Introspection und es *sagt dir die verfügbaren Verben* für dieses Repo: Es ist selbstbeschreibend, du (oder ein Agent) musst also nicht raten, was möglich ist. Wenn es ein brandneues Projekt ist, ist der erste echte Schritt normalerweise Scaffolding aus einem Template statt die Erkennung eines bestehenden. Und das Ganze funktioniert headless vom ersten Befehl an: Setze `FLEDGE_NON_INTERACTIVE`, frage nach `--json`, und ein Agent steuert es von Schritt eins. Erkennen, introspect, vielleicht scaffolden, dann ausführen.

## Der Lebenszyklus-Teil

Das Repo nennt fledge einen Zero-Config-Task-Runner mit Scaffolding und KI-Review. Das Task-Running ist das obige Verb-Set: das tägliche Build/Test/Run/Lint. Das Scaffolding ist die Antwort auf das zweite Ursprungsproblem: statt das Setup immer wieder von Hand in jedes neue Projekt zu kopieren, stellt fledge es auf. Und es gibt auch ein KI-Review-Stück, das in den Lebenszyklus eingebaut ist.

Task-Running kam zuerst. Es ist der Samen, der Teil, auf den alles andere aufgewachsen ist. Scaffolding sind wirklich Templates: ein neues Projekt aus einem statt es von Hand zusammenzukopieren aufstellen. Und KI-Review wurde eingefaltet, weil Review Teil des Dev-Loops ist. Wenn Agents den Code schreiben, ist das Bewerten kein separates Ritual, das man woanders ausführt, es ist nur ein weiteres Verb neben Build und Test. Eine Oberfläche schlägt drei Tools, für mich und noch mehr für einen Agent, der sonst drei Dinge lernen müsste. Diese drei sind die strukturellen Säulen, der Core, mit Plugins als Erweiterungsschicht darum.

fledge's eigene Einzeiler-Beschreibung ist "one CLI, your whole dev lifecycle, zero-config task runner, project scaffolding, AI review, and more," und das sind genau die drei Teile. Es ist ein einzelnes Rust-Binary, und du bekommst es auf dem langweilig-offensichtlichen Weg: `brew install CorvidLabs/tap/fledge` vom Homebrew-Tap, oder

`cargo install fledge` wenn du das bevorzugst, oder ein Shell-Skript. Nichts Exotisches beim Installieren.

## Wie ich es tatsächlich benutze

Das ist kein Tool, das ich manchmal greife. Es ist jedes Repo, die ganze Zeit, der Standardweg, wie ich jetzt alles baue, teste und ausführe. Und meine Agents steuern es. Die Agents führen fledge öfter aus als ich es von Hand tue. Es wurde gebaut, um viele Repos zu zähmen, und jetzt ist es die eine CLI, auf die sowohl ich als auch die Agents, die ich in diesen Repos arbeiten lasse, überall angewiesen sind.

Das ist auch der Grund, warum das nächste Stück wichtig ist. Eine CLI, auf die alle angewiesen sind, die jeder erweitern kann, die Agents installieren und ausführen, das kann keine feste Menge von Features sein, die ich einzeln abnicke. Sie muss von jedem erweiterbar sein, in welcher Sprache auch immer, ohne den Core anzufassen. Und sobald du beliebige Plugins zulässt, und sobald Agents diejenigen sind, die sie ausführen, musst du ernsthaft über Vertrauen nachdenken. Das ist das nächste Kapitel.

---

# Warum fledge und nicht Make

Die Leute fragen das, sobald sie hören, was fledge macht. Du hast einen Task-Runner gebaut? Es gibt Make. Es gibt Just. Es gibt Turborepo. Es gibt den npm-scripts-Block, der in jeder package.json direkt dort sitzt. Warum noch einen schreiben?

Die ehrliche Antwort ist, dass die Wand, gegen die ich gestoßen bin, nie Make war. Ich bin als iOS-Entwickler aufgewachsen, und um dort einen Build zu automatisieren, führte der Weg immer wieder zum selben Ort: fastlane, was Ruby bedeutet. Ich wollte kein Ruby. Ich bin ein Swift-Mensch. Ich wollte meine Builds in der Sprache automatisieren, in der ich arbeite, und stattdessen funnelte das gesamte Ökosystem mich in ein Ruby-DSL und ein `Gemfile` und ein Tool, das eine eigene kleine Welt zum Erlernen war. Jedes Mal, wenn ich etwas liefern wollte, lautete die Antwort "schreib eine fastlane-Lane," und jede fastlane-Lane war Ruby, das ich nicht schreiben wollte, in einer Runtime, die ich nicht verwalten wollte, die einen Job erledigte, den ich in Swift hätte erledigen können sollen. Das ist der Schmerz, aus dem fledge tatsächlich gewachsen ist. Nicht "Make ist klobig." Es war "warum werde ich gezwungen, für mein eigenes Projekt die Sprache jemand anderen zu benutzen?"

Wenn Leute fledge also gegen Make und Just antreten, zielen sie auf das falsche Ziel. Die Antwort ist nicht, dass diese Tools schlecht sind. Was ich von einem Task-Runner wollte, waren drei Dinge, und die fastlane-Wand ist, wo alle drei herkamen.

Ich wollte es auf meine Weise, in meiner Sprache, zu meinen Bedingungen, nicht an eine Runtime gebunden, so wie fastlane einen an Ruby bindet. Ein Task-Runner ist das Ding, das man hundertmal am Tag anfasst, und nach einer Weile hat man keine Lust mehr, in den Entscheidungen von jemand anderem zu leben, wie es sich anfühlen soll, oder in welcher Sprache man denken muss, um es zu benutzen. Ich wollte, dass `build` und `test` und `run` in jedem Repo dasselbe bedeuten, und ich wollte, dass die Schritte darunter in welcher Sprache auch immer schreibbar sind, die der Job erfordert, statt in Ruby marschiert zu werden. Make hält einen nicht davon ab, aber Make gibt es einem auch nicht. Man baut die Konsistenz selbst, in jedem Makefile, von Hand, für immer. Ein Tool, das mir erlaubt, den Dialekt pro Projekt neu zu erfinden, hat mein Problem nicht gelöst; es hat mir nur einen schöneren Ort gegeben, ihn neu zu erfinden.

Die anderen zwei sind die aus dem ersten Kapitel, und die fastlane-Datei ist, wo ich sie auch gewollt hätte. Keines dieser Tools ist agent-first (JSON standardmäßig,

Introspection, ein Headless-Modus), was der Teil ist, der mir jetzt am meisten wichtig ist, da meine Agents das Ding öfter steuern als ich es von Hand tue. Und keines ist ein einzelnes leichtes Binary, das man in ein Repo droppt, ohne dass zuerst eine Runtime installiert werden muss, was genau das ist, was eine konsistente Oberfläche über einen Haufen verschiedensprachiger Repos sein muss. Eine fastlane-Lane braucht Ruby; npm-Scripts brauchen npm; ein Runner, den ich in TypeScript schreiben würde, braucht eine TypeScript-Runtime. fledge braucht nichts.

Nichts davon ist "Make ist schlecht." Make ist großartig in dem, was Make tut, Just ist ein sauberer Befehlsrunner, Turborepo cached ein JS-Monorepo gut. Wenn eines davon alles ist, was du brauchst, benutze es. Ich werde nicht so tun, als würde fledge einen Feature-für-Feature-Kampf auf deren Heimgelände gewinnen.

Aber keines davon war das Ding, das ich mir tatsächlich gewünscht hatte, als ich vor Jahren vor einer fastlane-Datei stand: eine Oberfläche, agent-first, ein einzelnes leichtes Binary, und die Freiheit, die Schritte in welcher Sprache auch immer zu schreiben statt in Ruby gefunnelt zu werden. fledge ist das Tool, das ich damals gerne gehabt hätte, endlich gebaut.

---

# fledge und MCP

MCP ist der Umgebungsstandard jetzt. Im Jahr 2026 konsumieren Agents Tools größtenteils über MCP-Server. Wenn du ein Tool baust und willst, dass Agents es zuverlässig nutzen können, ist der Weg des geringsten Widerstands, einen MCP-Server zu exponieren. Das ist einfach der Stand des Ökosystems.

fledge hat keinen MCP-Server. Und trotzdem steuern Agents es konstant über Dutzende von Repos, und es funktioniert. Der Grund ist es wert, explizit gemacht zu werden, weil er dir etwas darüber sagt, welches Stück zuerst zu bauen ist.

## Die CLI ist das Primitiv

Ein MCP-Server ist ein Produktions-Interface. Er behandelt Request-Routing, er loggt, er gibt dir strukturierte Observability darüber, was ein Agent angefragt hat und was er zurückbekam. Das sind gute Dinge. Aber ein MCP-Server ist eine Schicht, die man über etwas legt. Die Frage ist: über was?

Wenn du den MCP-Server zuerst baust und die CLI ein Nachgedanke ist, bekommst du ein Tool, das für Agents funktioniert und für Menschen eine Qual ist. Wenn du die CLI zuerst baust, bekommst du ein Tool, das für Agents sofort ohne Protokolladapter funktioniert, für Menschen funktioniert, und vor dem ein MCP-Server platziert werden kann, wann immer du einen brauchst. Die CLI ist das Primitiv. Alles andere sitzt darauf.

fledge wurde mit diesem Gedanken gebaut. Jeder Befehl gibt `--json` aus. Es gibt `FLEDGE_NON_INTERACTIVE` für Headless-Ausführung. `fledge introspect` gibt jedem Aufrufer, Mensch oder Agent, ein strukturiertes Manifest jedes verfügbaren Verbs, was es tut und was es akzeptiert. Es gibt keine interaktiven Prompts, die unbeaufsichtigte Runs blockieren. Das Tool beschreibt sich selbst.

Dieses Profil, strukturierter Output standardmäßig, ein explizites Fähigkeitsmanifest, keine versteckten interaktiven Schritte, ist genau das, was eine MCP-Tool-Oberfläche einem Agent gibt. fledge hat all das ohne das Protokoll, weil diese Eigenschaften vom Designen für Agent-Aufrufer von Anfang an kamen, nicht vom späteren Umwickeln des Tools, um es agent-freundlich zu machen.

Eine Claude-Instanz, die fledge heute steuert, ruft `fledge introspect` auf, bekommt ein JSON-Manifest des Verfügbaren zurück, wählt das richtige Verb, übergibt `--json` und

liest strukturierten Output. Das ist derselbe Interaktions-Loop, den ein MCP-Server vermitteln würde, minus dem MCP-Framing. Die CLI ist bereits die saubere Tool-Oberfläche.

## MCP ist nicht der Konkurrent

Das Framing, das manchmal auftaucht, "CLI oder MCP?", behandelt sie als Alternativen. Das sind sie nicht. MCP ist eine Transport- und Protokollspezifikation. Die CLI ist das Ding, das die Arbeit macht. Die Frage ist nicht, welche man haben soll; es ist, welche man zuerst baut und welche man obendrauf schichtet.

Bau zuerst die CLI, die Art, die sich selbst beschreibt, JSON ausgibt und headless läuft. Sobald du das hast, ist es unkompliziert, einen MCP-Server oben drauf zu exponieren: du bildest jeden `fledge introspect`-Eintrag auf eine MCP-Tool-Definition ab, rufst die CLI darunter auf, gibst den JSON-Output zurück. Der Core ist bereits strukturiertes JSON über eine saubere Grenze mit einem expliziten Fähigkeitsmanifest. Ein MCP-Server ist ein Produktions-API über demselben Primitiv, genauso wie ein REST-API vor einer gut strukturierten Bibliothek sitzen könnte. Die Schnittstelle ändert sich; die Arbeit nicht.

Und weil die CLI die echte Oberfläche ist, kann alles, was einen Subprozess aufrufen kann, sie ohne den Protokolladapter benutzen. Ein Shell-Skript kann es benutzen. Ein CI-Runner kann es benutzen. Ein Mensch an einem Terminal kann es benutzen. Ein MCP-Client kann es durch die Server-Schicht benutzen. Keiner dieser Aufrufer erfordert, dass die anderen existieren. Du bekommst Universalität vom Primitiv, nicht vom Protokoll.

## Was MCP hinzufügt

Das alles ist kein Argument gegen MCP. Es fügt echte Dinge hinzu, sobald es vor der CLI sitzt.

Logging und Observability. Ein MCP-Server sitzt zwischen dem Agent und dem Tool, also kannst du jeden Aufruf aufzeichnen, timend, Argumente inspizieren, Anomalien flaggen. Die direkt aufgerufene CLI gibt dir, was du auch immer in eine Log-Datei pipetest. Die MCP-Schicht gibt dir strukturierte Observability ohne einzelne Instrumentierung jedes Befehls. Für den Produktionseinsatz, wo du auditieren willst, was ein Agent angefragt hat, ist das wichtig.

Auffindbarkeit auf Protokollebene. MCP hat eine standardisierte Möglichkeit für einen Agent, zu fragen "welche Tools sind hier?" und eine strukturierte Antwort zu

bekommen. fledge hat `fledge introspect` für dasselbe, aber `fledge introspect` erfordert, dass man weiß, dass fledge da ist. Ein MCP-Registry lässt einen Agent das Tool durch einen Standard-Handshake entdecken, bevor er irgendetwas über das Tool darunter weiß.

Komposition über Tools. Ein MCP-Server kann mehrere zugrundeliegende Tools durch einen Endpunkt exponieren, damit ein Agent einen Verbindungsort statt einer Liste einzelner CLIs kennen muss. Das ist ein operativer Komfort, wenn die Anzahl der Tools groß ist.

Das sind Produktionsinfrastruktur-Argumente. Sie gelten, wenn du Agents im großen Maßstab betreibst, ihr Verhalten auditierst und sie über eine verwaltete Schicht mit einer breiten Oberfläche von Tools verbindest. Der MCP-Server ist, als Bezeichnung für das, was er ist, ein KI-gerichtetes API mit Logging. Das ist eine nützliche Sache. Es ist nur nicht das, was man zuerst baut.

## Die Reihenfolge der Schritte

Zuerst CLI. Sie funktioniert für jeden Aufrufer in dem Moment, in dem sie existiert. Der MCP-Server ist der Wrapper, den du hinzufügst, wenn das Logging und die Protokollstandardisierung die Schicht wert sind, nicht vorher.

Für fledge ist die CLI das Fundament, auf dem ein Mensch, der Befehle ausführt, ein Agent, der ein Repo steuert, und ein MCP-Server, der mit Downstream-Clients spricht, alle sitzen. Die Agent-Geschichte ist nicht "Agents benutzen fledge über MCP." Es ist "Agents benutzen fledge auf dieselbe Weise wie alles andere, weil die CLI erstklassig für jeden Aufrufer gebaut wurde."

Wenn ein MCP-Server für fledge existiert, wird er eine dünne Schicht sein. Die Arbeit ist bereits im Core erledigt. Das ist der Sinn davon, das Primitiv zuerst zu bauen.

---

# Scaffolding und Templates

Das zweite Ding, das mich zu fledge trieb, nach dem Gleichheitsproblem, war Bootstrapping: der Kopieren-Einfügen-das-gleiche-Setup-Schmerz, den ich im ersten Kapitel benannt hatte. Hier ist, wie es genau aussieht. Du entscheidest dich, ein neues kleines Rust-CLI zu bauen, und die erste Stunde ist nicht das CLI. Es ist das Cargo-Layout, die Config, das Lint-Setup, das CI, das README-Skelett, all das Zeug, das du schon zwanzigmal getippt hast. Das ist die Steuer, und ich zahlte sie ständig, weil ich eine Menge Projekte starte.

Also ist Scaffolding eine Säule, kein Nebenfeature. Ein Projekt aus dem Nichts aufzustellen ist genauso Teil des Lebenszyklus wie es zu bauen und zu testen. Die ganze Idee ist: ein neues Projekt aus einem Template statt von Hand zusammenzukopieren aufstellen.

In fledge lebt das unter `fledge templates`. Du führst `fledge templates init` mit einem Namen und einem Template aus und es stellt das Projekt für dich auf:

```
fledge templates init my-tool --template rust-cli
```

Es gibt ein eingebautes Set, das die Arten von Projekten abdeckt, die ich tatsächlich starte. Die, die mitgeliefert werden, sind `rust-cli`, `ts-bun`, `python-cli`, `go-cli`, `ts-node`, `static-site`, `kotlin-kmp`, und `kotlin-ktor-api`. Diese Liste ist im Grunde eine Karte der Sprachen, in denen ich arbeite: ein Rust-CLI, ein paar TypeScript-Geschmacksrichtungen, ein Python-CLI, ein Go-CLI, eine statische Seite und die Kotlin-Multiplatform- und Ktor-API-Varianten. Das sind die acht im aktuellen Repo eingebauten Templates. Die Erkennungsseite von fledge weiß, wie sie mit Swift, Rust, Node und dem Rest umgeht, sobald ein Projekt existiert; die Templates-Seite ist, wie ein Projekt überhaupt erst entsteht.

Der Teil, der mir am meisten wichtig ist, ist, dass Templates keine geschlossene Liste sind, die ich einzeln abnicken muss. Du kannst aus jedem GitHub-Repo scaffolden, nicht nur aus den eingebauten. `--template user/repo` und es zieht das Template von dort:

```
fledge templates init my-app --template user/repo
```

Der Core liefert ein kleines nützliches Set, und darüber hinaus erweiterst du es selbst, ohne dass ich in der Schleife sein muss. Ein Template ist nur eine Projektform, die

jemand bereits herausgefunden hat, und wenn sie in einem GitHub-Repo lebt, kann fledge ein neues Projekt daraus aufstellen. Also ist das Set von Templates nicht "was CorvidLabs veröffentlicht hat." Es ist "jeder Ausgangspunkt, den jemand jemals in ein Repo gestellt hat."

Und es gibt ein vollständiges Set von Verben rund um Templates, nicht nur `init`. Es gibt `fledge templates create`, um eines zu erstellen, `fledge templates list` und `fledge templates search`, um sie zu finden, und `fledge templates validate`, um zu prüfen, ob ein Template tatsächlich wohlgeformt ist, bevor man sich darauf verlässt. Wenn ich aus einem Template scaffolde, und besonders wenn ein Agent es tut, möchte ich wissen, ob das Template zusammenhält, bevor es hundert Projekte mit demselben eingebackenen kaputten Ding abstempelt.

Es prüft mehr als "parst es." Es liest das `template.toml`-Manifest des Templates, bestätigt, dass Name und Beschreibung nicht leer sind, und prüft, ob alle Tools, die das Template als Anforderung angibt, tatsächlich in deinem PATH sind. Dann geht es durch jede Datei und jeden Dateinamen im Template und führt das Platzhalter-Templating darüber aus, damit ein schlechter Platzhalter (ein Syntaxfehler oder eine Variable, die das Manifest nie definiert) als Fehler aufgefangen wird, bevor du jemals daraus scaffoldest. Es flaggt auch ein Template ohne Dateien und warnt, wenn das Manifest nicht so eingestellt ist, dass es aus dem Output weggelassen wird. Es ist also strukturell, geht aber über "die Dateien sind da" hinaus: es übt das Templating tatsächlich so aus, wie `init` es tun würde, und sagt dir, wo es bricht.

Das ist der eigentliche Grund, warum Scaffolding in diese CLI gehört und nicht in ein separates Generator-Tool abseits. Derselbe Grund wie bei allem anderen in fledge: es ist dieselbe Oberfläche, für dieselben zwei Arten von Nutzern. Eine Person führt `fledge templates init` aus und überspringt die langweilige Stunde. Ein Agent führt exakt denselben Befehl aus, nicht-interaktiv, und stellt ein frisches Projekt von Schritt eins auf, ohne dass ein Mensch durch einen Wizard klickt. Aus dem ersten Kapitel: Erkennen, introspect, vielleicht scaffolden, dann ausführen. Der Scaffold- Schritt ist das "vielleicht." Wenn das Repo bereits existiert, erkennt fledge es. Wenn es noch nicht existiert, ist der erste echte Schritt, es aus einem Template aufzustellen, und dann funktioniert alles andere, das Build-Verb, das Test-Verb, das Review-Verb, sofort damit, weil es aus dem Template bereits so verdrahtet herauskam, wie fledge es erwartet.

Ein Projekt, das fledge von Geburt an spricht, ist das Ding, das ich wirklich wollte. Nicht nur "spar mir das Kopieren-Einfügen," obwohl es das tut. Ein scaffoldetes Projekt baut, testet und ist bereit für dieselben Verben wie jedes andere Repo, vom ersten Commit an. Das Boilerplate, das ich früher von Hand einfügte, war zur Hälfte

genau die Verdrahtung, die ein Projekt konsistent mit meinen anderen machte. Scaffolding in die Lifecycle-CLI einzufalten bedeutet, dass Konsistenz der Standard ist, mit dem ein Projekt geboren wird, nicht etwas, das ich nachträglich anschraube.

Also ja, ein neues Projekt beginnt mit dem Scaffold. Und selbst wenn ich nicht explizit nach `fledge templates init` greife, bekommt das Projekt dasselbe Setup vom ersten Commit an trotzdem verdrahtet: spec-sync, fledge, augur, attest. Das echte "init" ist nicht das Template, es ist das Stack, das reingeht. Der Befehl ist nur der schnelle Weg dorthin. So oder so wird ein neues Projekt von mir bereits mit dem Tooling geboren, das jedes andere hat.

---

# KI-Review im Loop

Die dritte Säule ist die, über die Leute erstaunt sind, sie in einem Task-Runner zu finden. Task-Running, Scaffolding, sicher, das sind offensichtlich Lebenszyklus- Dinge. Aber KI-Code-Review? In derselben CLI, mit der du baust und testest? Das fühlt sich so an, als würde es woanders hingehören, in sein eigenes Tool, in CI, in einen Bot auf deinen Pull Requests.

Es gehört aber nicht dahin, und der Grund ist einfach: Review ist Teil des Loops, genauso wie Bauen und Testen es sind. Du schreibst etwas, prüfst es, korrigierst es, machst weiter. Review ist der Prüfschritt. Und wenn Agents jetzt diejenigen sind, die den Code schreiben, was bei mir meistens der Fall ist, dann ist das Benoten dieses Codes nur ein weiteres Verb. Es sitzt direkt neben `build` und `test`, weil es denselben Job erledigt: es sagt dir, ob das Ding tatsächlich gut ist, bevor du weitermachst.

Eine Oberfläche schlägt drei Tools, und sie schlägt drei stärker für einen Agent als für mich. Das ist das gesamte Argument, Review in die Lifecycle-CLI einzubacken statt ein separates Tool zu liefern. Ein Agent in einem separaten Review-Tool muss eine ganz andere Sache lernen, mit seiner eigenen Invocation und seiner eigenen Output-Form, um einen kontinuierlichen Arbeits-Loop zu machen. Wenn der Agent bereits weiß, wie man fledge zum Bauen und Testen steuert, ist `fledge review` ein Verb, dessen Form er bereits kennt. Dieselbe Oberfläche, dasselbe JSON, derselbe Headless-Modus. Kein neues Tool zu lernen, nur weil sich der Schritt von "kompiliert es" zu "ist es gut" verändert hat.

In fledge ist das `fledge review`, und was es macht, ist KI-Code-Review gegen den Default-Branch. Es reviewt also nicht die ganze Welt. Es schaut auf das, was sich geändert hat, dein Diff gegen den Branch, in den du mergen würdest, was genau die Einheit ist, auf der Review tatsächlich passiert. Derselbe Scope, den ein menschlicher Reviewer oder ein PR-Bot betrachten würde, als Verb in derselben CLI ausgeführt, in der du bereits lebst.

Und es ist nicht an ein Modell oder einen Anbieter gebunden. Unter der Haube läuft Review durch denselben Multi-Provider-Client, den der Rest meines Toolings nutzt, corvid-ai, also spricht fledge mit jedem Provider, den corvid-ai unterstützt: Anthropics API, jeden OpenAI-kompatiblen Endpunkt, und lokale Runner wie Ollama, unter anderem. Die Liste der unterstützten Provider lebt im corvid-ai-Repo und ändert sich

mit dem Ökosystem. Der Punkt ist, dass das Review gegen das Modell läuft, das du willst, deine Wahl, nicht die des Tools.

Es lohnt sich, direkt zu sagen, was das bedeutet, weil der Rest dieses Buches direkt über Blast Radius ist: `fledge review` nimmt deinen Diff, und deine Specs, und schickt sie an welchen Provider auch immer du es gerichtet hast. Wenn das ein gehosteter Endpunkt ist, hat dein ungemergter Code gerade das Gebäude verlassen. Für ein privates Repo ist das eine echte Daten-Egress-Fläche, und es ist deine Entscheidung, mit offenen Augen zu treffen, kein Detail zu übergehen. Es auf ein lokales Modell zu richten, ist die Version, in der nichts die Maschine verlässt. Eine ehrliche Falte bei diesem lokalen Fall: Der Ollama-Pfad nimmt gerne einen Schlüssel und eine Cloud-URL, aber die corvid-ai-README framt Ollama immer noch als lokale, schlüssellose Option, also liest sich die Doku so, als würde Ollama die Kiste unter deinem Schreibtisch bedeuten. Der Cloud-Pfad funktioniert heute; die README hat nur noch nicht aufgeholt. Das ist eine Doku-Lücke bei mir, kein fehlendes Feature.

Es gibt auch einen Multi-Modell-Winkel, den ich wirklich mag. `--with-model` lässt dich ein Panel ausführen, parallele Kritiken desselben Diffs von mehr als einem Modell gleichzeitig.

```
fledge review --with-model ollama:gpt-oss:120b-cloud,ollama:qwen3-coder:480b-cloud
```

Das ist kein Gimmick. Wenn du irgendeine Zeit mit diesen Modellen verbracht hast, weißt du, dass sie nicht alle dieselben Dinge finden und auch nicht alle dieselben Dinge halluzinieren. Ein paar von ihnen über denselben Diff laufen zu lassen und zu sehen, wo sie übereinstimmen und wo eines von ihnen etwas flaggt, das die anderen verpassten, ist ein besseres Signal, als irgendeinem einzelnen zu vertrauen. Es ist eine zweite und dritte Meinung, parallel ausgeführt, über die genaue Änderung vor dir.

Und wie alles andere in fledge ist der Output strukturiert. Jeder Befehl gibt `{schema_version: 1, ...}` aus. JSON standardmäßig. Ein Review ist also keine Wand aus Prosa, die ein Agent zurücklesen und wie ein Mensch interpretieren muss. Es sind Daten. Der Agent, der den Code geschrieben hat, kann das Review ausführen, strukturierte Befunde zurückbekommen und im selben Loop darauf reagieren, ohne einen Menschen in der Mitte, der "der Reviewer scheint mit dem Error-Handling unzufrieden zu sein" in etwas tatsächlich Taugliches übersetzt. Eine Person kann das Review lesen, und ein Agent kann es parsen, vom selben Befehl.

`fledge review` ist auch spec-aware, was das spec-sync-Kapitel vollständig abdeckt; hier lohnt es sich nur zu sehen, was Review mit einer Spec macht, sobald es eine hat.

Review findet heraus, welche Specs die Dateien im Diff abdecken, indem es die deklarierten Dateien der Spec und das `specs/<name>/`-Verzeichnis abgleicht, und faltet diese Specs als Hintergrund in den Prompt. Und es ist zielgerichtet auf sie ausgerichtet: Das Modell wird darauf hingewiesen, dass die Specs beschreiben, was die Module *tun sollen*, sie zur Interpretation der Änderung zu nutzen, nur den Diff und nicht die Specs selbst zu reviewen, und, der wichtige Teil: wenn der Diff einem Spec-Invariant widerspricht, das als Bug im Diff auszuflaggen. Drift von der Spec ist also kein Hintergrundgeschmack in der Kritik; es ist ein Befund, den das Review explizit aufzufinden angewiesen wird.

Und es verdient den Platz. `fledge review` hat mir wirklich einen Bug gefunden. Etwas, das ausgeliefert worden wäre, direkt in einem Diff, der gut baute und seine Tests bestand, das das LLM flaggte, bevor es gemergt wurde. Das ist der Test, ob Review-als-Verb irgendetwas wert ist, und er wurde bestanden: kein Stil-Nit, ein tatsächlicher Defekt, den der Rest des Loops durchgewunken hatte. Die spec-aware- Seite hat sich auch gelohnt. Den Reviewer auf die Specs zu richten, hat Code gefunden, der still vom Vertrag driftete, den das Modul halten sollte, Drift, die kompiliert und besteht, genau wie der Bug. Die Agents lehnen sich gut darauf, weil es für sie dieselbe Form wie Build und Test hat: ausführen, strukturierte Befunde lesen, beheben was es fand, weitermachen.

Wenn Agents den Code schreiben, den Build ausführen und die Tests ausführen, ist das Benoten des Codes ein weiteres Verb in Reichweite, das sie bereits kennen, und es findet Dinge, die die anderen durchgelassen haben.

---

# Plugins und ihnen nicht vertrauen

fledge hat einen kleinen Core, der wenig tut, mit allem echten Können in Plugins. Das ist so gewollt. Wenn der Core über jede Sprache, jeden Workflow, jeden seltsamen Schritt jedes Teams Bescheid wissen müsste, würde er verrotten. Also tut er es nicht. Der Core kennt die gängigen Verben und wie man ein Projekt erkennt; die echte Reichweite kommt von Plugins, die darum herum eingeworfen werden, und jeder kann einen hinzufügen, ohne den Core anzufassen.

Der andere Grund, Fähigkeiten in Plugins auszulagern, ist, dass ich die Liste dessen, was fledge kann, nicht besitzen wollte. Die Leute erweitern es, wie sie möchten: Rust, Swift, TS, Shell. Du schreibst den Plugin in was auch immer du kannst. Du wirst nicht in meine Sprache gezwungen, um mein Tool zu erweitern.

Die Art, wie ein Plugin mit dem Core kommuniziert, ist ein echter, versionierter Vertrag: ein Binary mit einem `plugin.toml`-Manifest, das mit fledge JSON spricht, dasselbe ob es nativ oder ein sandboxed WASM-Modul ist. Das Protokoll-Kapitel hat das Wire-Format und den Capability-Handshake; hier reicht es zu wissen, dass die Naht ein Vertrag ist, kein Haufen Konventionen.

Und um eine Frage zu beantworten, die oft auftaucht: Plugins und Lanes sind nicht dasselbe. Ein Plugin fügt eine Fähigkeit hinzu, ein neues Verb. Eine Lane verkettet Verben, die du bereits hast, in eine geordnete Pipeline. Es ist also nicht "alles ist ein Plugin." Es gibt einen echten Core mit den drei eingebauten Säulen, Plugins erweitern die Fähigkeiten darum herum, und Lanes reihen diese Fähigkeiten in Sequenzen. Ich gebe Lanes ihr eigenes Kapitel später.

## Jede Sprache, was bedeutet: man kann ihnen nicht vertrauen

Die Kehrseite von "erweitere es, wie du willst, in jeder Sprache" ist, dass man am Ende Code ausführt, den man nicht geschrieben hat und für den man nicht bürgen kann. Ein Plugin ist nur das Programm von jemand anderem. Sobald man entschieden hat, dass jeder einen in allem schreiben kann, hat man auch entschieden, dass man eine Menge nicht vertrauenswürdigem Code ausführen wird.

Also sandboxt fledge Plugins mit WASM, auf Wasmtime. Der Punkt ist Sicherheit. Ein Plugin kann nicht die gesamte Festplatte lesen oder nach Hause telefonieren, es sei denn, du erlaubst es. Standardmäßig ist es eingesperrt: es bekommt, was du ihm gewährst, und nichts sonst.

Es gibt einen zweiten Grund für die Sandbox, und das ist der eigentliche. Agents führen diese Plugins aus. Wenn Agents Plugins installieren und ausführen, kann man ihnen nicht standardmäßig vertrauen. Nicht den Plugins, und nicht, blind, der Entscheidung, sie auszuführen. Ein Agent, der erreicht, einen Plugin greift und ihn ausführt, ist genau die Situation, in der "es ist wahrscheinlich okay" nicht gut genug ist. Die Sandbox macht agent-gesteuertes Tooling sicher. Das ist die Brücke zu dem Vertrauen- und-Blast-Radius-Zeug, auf das ich im Agents-Buch eingehe; hier ist die Tooling- Seite einfach: nicht vertrauenswürdiger Code plus eine automatisierte Sache, die ihn ausführt, gleich man braucht eine Box darum. WASM/Wasmtime ist die Box.

Die Wahl von WASM gegenüber den offensichtlichen Alternativen ist es wert, benannt zu werden. Die zwei, zu denen man als erstes greift, sind seccomp-Profile (Linux-Syscall-Filterung) und Container (Docker oder ähnliches). Beide funktionieren, aber beide fügen Reibung oder Annahmen hinzu, die hier nicht passen. seccomp erfordert OS-Level-Privilege zur Konfiguration und ist nur Linux, bricht also sofort das Cross-Platform-Versprechen. Container bedeuten einen Docker-Daemon, einen separaten Image-Pull und eine schwerere Prozessgrenze als "run einen Plugin." WASM passt anders: Wasmtime bettet sich direkt in den fledge-Prozess als Bibliothek ein, wird innerhalb des einzelnen Binaries mitgeliefert, läuft auf jedem OS ohne zusätzliche Daemons oder Privilegien, und erzwingt die Capability-Grenze strukturell zur Link-Zeit statt durch eine Kernel-Policy, die jemand konfigurieren muss. Nicht dass seccomp oder Container falsch sind. Es ist, dass ein Tool, das als ein Binary überall läuft, eine Sandbox braucht, die mit ihm reist, und Wasmtime das tut.

## Der Canary

Eine Sandbox, die man nicht beweisen kann, ist nur eine Hoffnung. Also gibt es einen Canary, einen Plugin, dessen Aufgabe es ist, die Dinge zu versuchen, die ein sandboxed Plugin nicht tun können sollte, und zu bestätigen, dass er es nicht kann. Er ist der Beweis, dass die Sandbox hält. Wenn der Canary nicht ausbrechen kann, ist die Grenze real; wenn er es jemals könnte, wüsste man es sofort.

Es sind tatsächlich zwei Plugins. `fledge-plugin-canary` ist der native. Er läuft unsandboxed und beweist, dass die Angriffe *funktionieren*: SSH-Schlüssel und AWS-Credentials lesen, Umgebungsvariablen wie `GITHUB_TOKEN` herausziehen, Netzwerkverbindungen öffnen, Prozesse spawnen, in `.git/hooks` schreiben. Dann führt `fledge-plugin-canary-wasm` dieselbe Batterie innerhalb der Wasmtime-Sandbox aus, wo jeder dieser Fälle BLOCKED zurückkommen sollte. Seine eigene Beschreibung ist direkt darüber: es "beweist, dass die Wasmtime-Sandbox jeden Angriff, den der native Canary exponiert, blockiert."

Der Punkt ist schärfer, wenn man die beiden nebeneinander sieht, statt meinem Wort dafür zu glauben. Der eigene Output des nativen Canarys kontrastiert sich mit dem, was ein WASM-Plugin, das denselben Code ausführt, sehen würde:

```
NATIVE: ~/.ssh/ READABLE            → WASM: BLOCKED (no preopened dir for ~)
NATIVE: ~/.config/fledge/ READABLE   → WASM: BLOCKED (outside sandbox)
NATIVE: GITHUB_TOKEN LEAKED          → WASM: BLOCKED (not passed to guest)
```

Gleicher Angriff, zwei Grenzen. Native liest deinen SSH-Schlüssel; WASM kann es nicht, weil es kein vorgeöffnetes Verzeichnis gibt, über das es ~ erreichen kann. Native erbt `GITHUB_TOKEN`; WASM kann es nicht, weil die Umgebung des Gastes leer ist. Jedes Ergebnis, das LEAKED statt BLOCKED auf der WASM-Seite zurückkommt, bedeutet, dass die Sandbox entkommen ist. Die beiden zusammen sind der End-to-End- Check: eine zeigt, dass die Gefahr real ist, die andere zeigt, dass die Box hält.

## Was die Sandbox nicht schützt

Die Sandbox ist Blast-Radius-Eindämmung. Sie begrenzt, was der Code eines Plugins erreichen kann. Sie begrenzt nicht alles, und sie als Heilmittel für Probleme zu verkaufen, die sie nicht löst, wäre die unehrliche Version dieses Kapitels. Also hier ist der Teil, den WASM nicht bringt.

Es verhindert keine Daten-Egress höher im Stack. Die WASM-Box hält einen Plugin davon ab, deinen SSH-Schlüssel zu erreichen, aber sie regelt keine Daten, die du einem Tool absichtlich übergibst. `fledge review` schickt deinen Diff und deine Specs an welchen LLM-Provider du es gerichtet hast, und wenn das ein gehosteter Endpunkt ist, hat ungemergter Code gerade das Gebäude verlassen. Keine WASM-Grenze berührt das, weil die Daten durch die Vordertür gehen, nicht durch den Plugin. Das ist ein Zustimmungs-und-Policy-Problem, und ich behandle es als eines im Review- Kapitel, nicht als etwas, das die Sandbox löst.

Es schützt nicht vor einem Plugin, der als Host-User läuft. Nicht jeder Plugin ist WASM. Native Plugins laufen mit deinen Berechtigungen, deiner Umgebung, deiner Festplatte. Das ist der ganze Grund, warum der native Canary deinen SSH-Schlüssel lesen kann: er ist nicht sandboxed. Wenn du einen nativen Plugin ausführst oder einen, der einen Prozess von unter dem Gast heraus forkt, vertraust du ihm so, wie du allem vertraust, was du auf deiner Maschine ausführst. Die Sandbox ist die Garantie des WASM-Pfads, keine allgemeine über jeden Plugin.

Und es validiert keine Herkunft oder Absicht. WASM enthält, was Code tun kann. Es sagt nichts darüber, ob der Code oder der Agent, der den Code schrieb, den der

Plugin ausführt, das tun sollte. Ein Plugin, der agent-geschriebene Logik ausführt, führt immer noch Logik aus, die ich nicht reviewed habe, innerhalb der Box. Nicht vertrauenswürdiger Code in einer Box ist immer noch nicht vertrauenswürdiger Code. Die Box hält nur den Schaden davon ab, sich auszubreiten.

Also ist das ehrliche Framing eng: WASM/Wasmtime ist herkunftsagnostische Blast-Radius- Eindämmung für den Code-Ausführungspfad. Es ist keine Datenfluss-Kontrolle, kein Zustimmungsmanagement und kein Grund, aufzuhören darüber nachzudenken, was man einem Plugin übergibt oder wer es geschrieben hat.

Das Plugin-Ökosystem ist größer, als ich es erwartet hätte, wenn man fledge nur nach seinen drei nativen Built-ins kennt: viele `fledge-plugin-*`-Repos in der CorvidLabs-Org über mehrere Sprachen hinweg (das Ökosystem-Kapitel deckt sie ab), geschrieben in welcher Sprache auch immer der Job erforderte. Die Sandbox ist das, was das möglich macht: ein Plugin kann in allem geschrieben werden und ist trotzdem sicher auszuführen. Den vollständigen Rundgang durch das Ökosystem und die Sprachen mache ich in seinem eigenen Kapitel.

## Wie fledge in den Tag passt

Agents, die nicht vertrauenswürdige Plugins in hohem Tempo über viele Repos ausführen, sind der Grund, warum die Sandbox nicht verhandelbar ist. Das Ding, das diese Entscheidungen trifft, bin normalerweise nicht ich, der von Fall zu Fall entscheidet. Es sind Agents, die ständig laufen, über viele Repos. Die WASM-Sandbox darunter ist das, was es sicher macht, das laufen zu lassen.

---

# Das Plugin-Protokoll

Das ist die Naht, der tatsächliche Ort, wo ein Plugin und der Core sich treffen. Es hat einen Namen im Repo: Das Protokoll ist versioniert, und der Versionsstring ist `fledge-v1`. Ein Plugin deklariert es, und das ist der Handshake. Alles andere hängt daran.

## Was ein Plugin-Autor tatsächlich schreibt

Ein Plugin ist ein Git-Repo mit einem Manifest an der Wurzel namens `plugin.toml`, plus einem oder mehreren Executables. Das Manifest ist kurz. Du benennst den Plugin, gibst ihm eine Version, sagst, welches Protokoll du sprichst, und listest die Befehle auf, die du hinzufügst:

```
[plugin]
name = "fledge-deploy"
version = "0.1.0"
protocol = "fledge-v1"

[[commands]]
name = "deploy"
binary = "bin/fledge-deploy"

[capabilities]
exec = true
store = true
metadata = false
```

Das ist der gesamte Vertrag auf der Autorenseite. Jeder `[[commands]]`-Eintrag ist ein neues Verb, das fledge kennen wird, auf ein Binary gerichtet. Der `[capabilities]`-Block ist der Autor, der im Voraus sagt, was dieser Plugin in der Lage sein muss zu tun: Shell-Befehle ausführen, ein bisschen Zustand persistieren, Projekt-Metadaten lesen. Standard ist `false`. Du fragst nach dem, was du brauchst und nichts mehr, und das Fragen ist im Manifest sichtbar, nicht im Code vergraben.

## Wie der Host mit einem Plugin kommuniziert

Wenn du einen Plugin-Befehl aufrufst, spawnt fledge das Binary und kommuniziert mit ihm über stdin und stdout als JSON-Zeilen, ein JSON-Objekt pro Zeile. Das erste, was der Host sendet, ist eine `init`-Nachricht, und diese Nachricht ist die Zero-Config-

Idee buchstäblich gemacht. Er übergibt dem Plugin die gesamte Situation: den Projektnamen und -root, die erkannte Sprache, den Git-Zustand (Branch, dirty oder clean, Remote), die eigene Version und das Verzeichnis des Plugins, die fledge-Version und die genau gewährten Capabilities. Der Plugin muss nicht herausfinden, wo er ist oder was er betrachtet. Der Host weiß es bereits, das ist die ganze Aufgabe des Cores, also sagt er es einfach dem Plugin.

Danach ist es ein Gespräch. Der Plugin sendet Nachrichten zurück: `prompt`, `confirm`, `select`, um den Nutzer nach etwas zu fragen; `log`, `progress`, `output`, um zu berichten; `exec`, um einen Befehl auszuführen; `store` und `load` für seinen kleinen Zustandsfleck. Alles mit einer `id` bekommt genau eine Antwort zurück: eine `response` oder ein `cancel`. Stderr wird nie erfasst, also kann ein Plugin-Autor immer direkt zum Terminal debuggen.

Es ist einfach genug, es über jemandes Schulter zu lesen. Der Host öffnet mit `init`, übergibt die Situation:

```
{"type": "init", "protocol": "fledge-v1",
 "args": ["staging"],
 "project": {"name": "my-app", "root": "/Users/dev/my-app", "language": "rust",
   "git": {"branch": "main", "dirty": false, "remote": "origin"}},
 "capabilities": {"exec": true, "store": true, "metadata": false}}
```

Der Plugin, bereits wissend, wo er ist, bittet fledge, einen Befehl auszuführen:

```
{"type": "exec", "id": "6", "command": "git tag -l 'v*' --sort=-v:refname",
"timeout": 10}
```

Und der Host beantwortet die passende `id`:

```
{"type": "response", "id": "6", "value": {"code": 0, "stdout":
"v0.9.1\nv0.9.0\n", "stderr": ""}}
```

Das ist die ganze Form: `init` liefert alles, was ein Agent sonst in einer README raten müsste, als strukturiertes JSON statt Prosa, und jede Anfrage mit einer `id` bekommt genau eine passende Antwort. Das Tool sagt dem Plugin, wo es ist; der Plugin sucht nicht.

## Wo die Sandbox angeschraubt wird

Alles oben beschreibt einen nativen Plugin, ein normales Executable, das JSON über Pipes spricht. Das Problem, und das Repo ist direkt darüber, ist, dass ein nativer

Plugin als du läuft, mit deinem vollen Zugriff. Der Capability-Block sperrt das Protokoll, aber nicht den *Prozess*. Ein Plugin, der nichts angefragt hat, könnte immer noch deine SSH-Schlüssel lesen, weil es nur ein Programm ist, das als dein Nutzer läuft. Das ist das Leck, das der Canary bewiesen hat: Capabilities auf Protokollebene zu deklarieren war Security-Theater, und das ist der ganze Grund für den WASM-Schritt. Ich habe diesen Bogen im Sandbox-Kapitel erzählt; hier ist der Punkt nur, was es im Protokoll geändert hat.

WASM-Plugins behalten das exakt gleiche `fledge-v1`-Protokoll, ändern aber die Grenze. Der Plugin deklariert `runtime = "wasm"` und liefert ein einzelnes `.wasm`- Binary. Statt stdin und stdout überqueren dieselben JSON-Nachrichten durch drei Host-Funktionen (`recv`, `send`, `exit`), die der Host einlinkt. Gleiche Nachrichtentypen, gleiches Gespräch. Und die Capabilities hören auf, eine höfliche Anfrage zu sein. Sie werden zur Link-Zeit durchgesetzt: Wenn du `exec` nicht gewährt hast, wird der `exec`-Import einfach nicht eingelinkt, und ein Plugin, der versucht, ihn aufzurufen, schlägt bei der Instantiierung fehl. Es gibt keinen Laufzeit-Check zu täuschen, weil die Funktion, die er aufrufen würde, für ihn nicht existiert. Dateisystemzugriff ist `none`, `project` oder `plugin`, als vorgeöffnete Verzeichnisse gemountet; Netzwerk ist ein Boolean. Darüber hinaus ist die Runtime fuel-bounded und memory-capped, also kann ein Plugin nicht für immer drehen oder die Maschine fressen.

Also sind die Capability, die du in `plugin.toml` siehst, und die Capability, die der Code tatsächlich erreichen kann, strukturell dieselbe.

Der Grund, warum es strukturell sein musste, ist die Lektion, die der Canary lehrte: eine Capability, die man nur *deklariert*, ist eine Capability, der man vertraut, dass der Plugin sie respektiert, und der ganze Grund, warum man einen Plugin sandboxt, ist, dass man ihm nicht vertraut. Die einzige Version, die hält, ist die, in der die nicht gewährte Capability nicht erreichbar ist, wo die Funktion buchstäblich nicht da ist, um aufgerufen zu werden. Wenn Agents diejenigen sind, die diese Dinge installieren und ausführen, ist das, was man will.

Ein Versionsrisiko ist hier erwähnenswert. Der Protokoll-String, den ein Plugin deklariert, ist `fledge-v1`. Wenn eine brechende Protokolländerung kommt, und sie wird irgendwann kommen, wird dieser String zu `fledge-v2`. Ein Plugin, der immer noch `fledge-v1` gegen einen Host deklariert, der `fledge-v2` erwartet, schlägt bei der Instantiierung laut fehl, bevor irgendein Code läuft. Er verhält sich nicht still falsch. Die Version im Manifest ist das, was diesen frühen Fehler erzwingt: ein falsch passender Plugin erreicht nicht den Punkt, an dem er irgendetwas tun kann, also weiß der Operator genau, was aktualisiert werden muss, statt einem subtilen Laufzeit-Bug nachzujagen.

## Lane-Lifecycle-Hooks

Dasselbe Protokoll trägt eine zweite Art von Integration: Lifecycle-Hooks. Ein Plugin kann einen Hook in seinem `plugin.toml` registrieren, der auf `lane:pre-` oder `lane:post`-Events statt auf direkte Befehlsaufrufungen feuert. Der Deploy- Plugin ist das Beispiel, das mit der Doku mitgeliefert wird: er registriert einen `lane:post`-Hook, der automatisch ausgeführt wird, nachdem eine fledge-Lane abgeschlossen ist.

Wenn der Hook feuert, übergibt fledge Kontext an das Hook-Binary durch Umgebungs- variablen: `FLEDGE_LANE_NAME`, `FLEDGE_LANE_STATUS`, und `FLEDGE_LANE_RUN_ID`. Die Lane-Definition hat keine Kenntnis, welche Plugins installiert sind. Der Plugin hat keine Kenntnis, welche Lanes existieren. Das Lifecycle-Event ist die Naht.

Hier ist, wie das vollständige Bild aussieht. Die Lane lebt in `fledge.toml`; die Hook-Registrierung lebt im `plugin.toml` des Plugins; die beiden verbinden sich, ohne dass einer die Details des anderen kennt:

```
# fledge.toml
[lanes.verify]
description = "Pre-merge gate: format, lint, test, build"
steps = ["fmt", "lint", "test", "build"]
```

```
# plugin.toml (from fledge-deploy, or any plugin registering a lane hook)
[plugin]
name = "fledge-deploy"
version = "0.2.0"
protocol = "fledge-v1"

[[commands]]
name = "deploy"
binary = "bin/fledge-deploy"

[[hooks]]
event = "lane:post"
binary = "bin/fledge-deploy-hook"

[capabilities]
exec = true
store = true
metadata = false
```

Wenn `fledge lanes run verify` abgeschlossen ist, feuert fledge das `lane:post`-Event. Jeder installierte Plugin mit einem passenden `[[hooks]]`-Eintrag bekommt sein Hook-

Binary aufgerufen mit `FLEDGE_LANE_NAME=verify`, `FLEDGE_LANE_STATUS=success` (oder `failure`), und `FLEDGE_LANE_RUN_ID` gesetzt.

Die Lane gibt strukturierten Kontext als Umgebungsvariablen statt Prosa auf stdout aus, also liest ein Plugin einen Status, auf den er reagieren kann, statt Text zu scrapen und zu raten. Das ist dasselbe Designprinzip wie die `init`-Nachricht: der Host sagt dir, was passiert ist; du musst es nicht schlussfolgern.

## Die drei Säulen sitzen darauf

Die drei Säulen (Task-Running, Scaffolding, KI-Review) sind Core, keine Community-Plugins. Das Protokoll ist die Erweiterungsschicht *darum herum*. Das Rückgrat gehört fledge, und `fledge-v1` ist, wie jeder mehr Fähigkeiten daran anschrauben kann. Der Core ist real und die Plugins umringen ihn; das Protokoll ist nur die Naht, wo die beiden sich treffen.

---

# Das Plugin-Ökosystem

Es gibt Dutzende von `fledge-plugin-*`-Repos in der CorvidLabs-Org, über mehrere Sprachen hinweg, mit einer Handvoll archivierter. Die Live-Zahl ist real, aber es ist das Uninteressanteste am Ökosystem. Plugin-Zahl ist Inventar. Das Argument ist, welche ich dir tatsächlich in die Hand geben würde, und wie sicher ich bei jedem bin.

Also hier sind die fünf, die du täglich benutzen würdest: `augur`, `attest`, `format`, `gitleaks`, `deps`. Fang dort an. Der Rest dieses Kapitels handelt davon, warum die anderen siebenunddreißig existieren und wie sehr man ihnen vertrauen kann.

## Die fünf, nach denen du greifen würdest

Diese laufen bei praktisch jeder Änderung, meiner und der Agents'. Sie clustern in zwei Jobs.

Das Vertrauenspaar ist `augur` und `attest`. `augur` bewertet das Änderungsrisiko eines Diffs, weitermachen, reviewen oder blocken, aus strukturellen Signalen allein, kein API-Schlüssel und kein LLM, und seine Beschreibung sagt das Publikum laut: "for humans and agents." `attest` zeichnet die signierte Provenienz auf, wer welchen Commit reviewed hat, in git notes gespeichert. Diese zwei sind die, auf die ich am schwersten angewiesen bin, weil sie diejenigen sind, die wichtig sind, wenn ein Agent der Autor ist. Risikobewertung auf dem Weg rein, ein Provenienz-Trail auf dem Weg raus.

Die drei Dev-Loop-Plugins sind absichtlich langweilig: `format`, `gitleaks`, `deps`. Den Formatter ausführen. Nach committed Secrets scannen. Dependency-Gesundheit prüfen, veraltet, audit, Lizenzen, über Rust, Node und Python. Keiner von ihnen ist aufregend. Alle laufen ständig, weil das genau die Art von Check ist, die du automatisch feuern lassen willst statt daran zu erinnern.

Wenn du nur diese fünf installierst, hättest du das meiste von dem, was ich aus dem Ökosystem bekomme.

## Drei Tiers und wie sicher ich bin

Die ehrliche Art, die Liste zu lesen, ist nach Vertrauen, nicht nach Anzahl. Die Plugins fallen in drei Tiers, und ich vertraue ihnen sehr unterschiedlich. Prüfe die

Live-Liste in der CorvidLabs-Org für die aktuelle Liste; was folgt, ist, wie man liest, was auch immer man dort findet.

**Core.** `augur`, `attest`, `format`, `deps`, `gitleaks`. Die täglichen fünf. Ich habe diese geschrieben, ich führe sie bei jeder Änderung aus, und sie werden gepflegt und getestet. Das ist der Tier, auf den ich den Workflow setzen würde.

**Integration.** `github`, `discord`, `algochat`, `memory`. Diese verbinden fledge mit etwas Externem: die GitHub-API über `gh`, Discord-Webhooks für CI-Failure-Benachrichtigungen, verschlüsseltes On-Chain-Messaging, ein Memory-Store. Sie werden auch gepflegt und getestet, tragen aber das Risiko von dem, mit dem sie kommunizieren. Eine Integration ist nur so zuverlässig wie der dahinterliegende Service.

**Spielplätze.** `weather`, `roast`, `hangman`, und der Rest der Einzel-Stücke. `weather` druckt eine Terminal-Vorhersage. `hangman` spielt Hangman mit Bezeichnern, die aus deiner Codebase herausgezogen werden. `roast` läuft einen Commit durch das LLM und röstet ihn, "nur zu Unterhaltungszwecken." Diese werden nicht auf demselben Niveau gepflegt. Sie existieren, um zu beweisen, dass die Sandbox funktioniert: Wenn ein halbernster Weather-Plugin, der auf eine Laune hin geschrieben wurde, sicher auszuführen ist, erledigt das Protokoll seinen Job. Die niedrige Messlatte ist keine Schwäche. Es ist der Beweis.

Core und Integration werden gepflegt und getestet. Die Spielplätze beweisen, dass die Sandbox hält. Lies die flache Zahl nicht als gleichmäßige Verteilung von gleich geprüften Tools, weil sie es nicht sind, und so zu tun wäre die unehrliche Version dieses Kapitels.

## Provenienz, direkt

Ich habe die meisten davon geschrieben. Die täglichen fünf sind meine. Ebenso fast alle Einzel-Stücke, auf eine Laune hin geschrieben, weil das Veröffentlichen eines Plugins nichts kostet, sobald das Protokoll existiert. Ich brauchte etwas, ich schrieb einen Plugin, ich warf ihn ein. Der Punkt des Protokolls ist, dass ich den Core nicht erweitern muss, um eine Fähigkeit hinzuzufügen.

Ich werde keine Community externer Autoren beanspruchen, die ich nicht habe. Ein paar Plugins kamen aus dem Kreis, aber ich präsentiere die Liste nicht als community-geprüfte Breite. Es ist meist eine Person, die eine Schublade füllt, und das ist die ehrliche Provenienz.

## Der Canary und warum das alles sicher ist

Ein Paar verdient seine eigene Erwähnung, weil es tragend für das gesamte Ökosystem ist: `fledge-plugin-canary` und `fledge-plugin-canary-wasm`, das Red-Team-Paar aus dem Sandbox-Kapitel. Der native beweist, dass die Angriffe funktionieren. Der WASM- Plugin beweist, dass die Sandbox sie blockiert. Alles andere hier, ein Plugin, der in Kotlin geschrieben wurde, ein Plugin, den ich halbschlafen schrieb, ist nur sicher auszuführen, weil dieses Paar die Grenze hält. Das Ökosystem kann chaotisch sein, genau weil die Box darum herum es nicht ist.

## Warum so viele Sprachen

Die meisten sind Rust und Shell, dann Swift, TypeScript, Kotlin, Python, und ein paar HTML/JavaScript-Varianten. Diese Verteilung ist das Protokoll, das wie beabsichtigt funktioniert. Der ganze Grund für `fledge-v1` und die WASM-Sandbox ist, dass ein Plugin in allem geschrieben werden kann und trotzdem sicher auszuführen ist, also muss sich das Ökosystem nicht auf eine Sprache einigen, nur auf einen Vertrag. `fledge-plugin-bridge` ist Kotlin. `fledge-plugin-memory` ist TypeScript. `fledge-plugin-attest` ist Swift. Die Hälfte der Integrationen sind Shell. Niemand musste Rust lernen, um zu fledge beizutragen. Sie schrieben das Ding in was auch immer sie vor sich hatten, und der Core blieb die ganze Zeit gleich groß.

Ich besitze nicht die Liste dessen, was fledge kann. Das Protokoll tut es, und es lässt jeden dazu beitragen, ohne mich zu fragen. Die Verteilung ist der Beweis, und die Tiers sind die Ehrlichkeit darüber.

---

# spec-sync und ehrlich bleiben

Spec-Drift ist der Fehlermodus, der mir mehr wichtig ist als den meisten. Du schreibst eine Spec, den Vertrag für ein Modul, was es tut, was seine öffentliche API ist, und dann driftet der Code. Jemand ändert eine Funktion, die Spec behauptet immer noch die alte Form, und jetzt widersprechen das Dokument und der Code still. Mit einem menschlichen Team ist das eine veraltete README. Mit einem Agent in der Schleife ist es schlimmer, weil der Agent die Spec als Wahrheit liest, auf einem Vertrag aufbaut, den der Code nicht mehr honoriert, und niemand bemerkt es, bis etwas bricht. spec-sync existiert, um diesen Drift unmöglich zu ignorieren.

spec-sync ist sein eigenes Tool, ein eigenes Rust-Binary, eine eigene GitHub Action, und sein Tagline sagt, was es ist: "Bidirectional spec-to-code validation with cross-project references, dependency graphs, and AI-powered generation." Die zwei Worte, die wichtig sind, sind *bidirektional* und *Validierung*. Es prüft, in beide Richtungen, dass die Spec und der Code übereinstimmen. Es ist kein Test-Suite und kein unscharfer Diff gegen Prosa. Es ist strukturelle Vertragsüberprüfung: stimmt die dokumentierte öffentliche API tatsächlich mit der echten überein. Die zwei Richtungen sind nicht symmetrisch: Ein Export, den der Code hat, den die Spec aber nicht dokumentiert, ist eine Warnung, etwas zum Ausfüllen; ein Eintrag, den die Spec behauptet, den der Code aber nicht hat, ist ein Fehler, ein gebrochenes Versprechen.

## Was es tatsächlich prüft

Eine Spec ist eine Markdown-Datei, `*.spec.md`, mit YAML-Frontmatter und einem Set von erforderlichen Abschnitten. Der Frontmatter benennt das Modul, eine Version, einen Status und die Quelldateien, die er abdeckt. Die erforderlichen `##`-Abschnitte sind Purpose, Public API, Invariants, Behavioral Examples, Error Cases, Dependencies und Change Log. Fehlender erforderlicher Abschnitt und die Spec ist unvollständig und die Validierung blockiert.

Das ist die Regel; hier ist eine Spec, die sie ehrt. Das ist der Kopf von fledges eigenem `review`-Modul-Spec, der echte Frontmatter, und der `## Public API`-Abschnitt mit den tatsächlichen Exports ausgefüllt, gegen die spec-sync den Code prüft:

```
---
module: review
version: 10
status: active
files:
  - src/review.rs
db_tables: []
depends_on:
  - spec
  - llm
  - config
---

# Review

## Purpose

AI-powered code review of current branch changes...

## Public API

### Exported Functions

| Export | Description |
|--------|-------------|
| `run` | Entry point for the review command |
| `ReviewOptions` | Options struct: base, file, json, model, provider,
with_model |
| `ReviewFormat` | Enum: Summary, Checklist, or Inline |
```

Die `files:`-Zeile ist das, was spec-sync gegen den Diff und gegen die echten Exports von `src/review.rs` abgleicht. Jede Zeile in dieser `## Public API`-Tabelle ist ein Name, den spec-sync im Code erwartet zu finden; ein Eintrag, den der Code nicht hat, ist der unten beschriebene Fehlerfall. Wenn du noch nie eine Spec geschrieben hast, ist dieser Frontmatter plus eine ehrliche `## Public API`-Tabelle die Form zum Kopieren. Die restlichen erforderlichen Abschnitte sind dasselbe, Prosa und Tabellen, die beschreiben, was das Modul verspricht.

Dann validiert es in beide Richtungen:

- **Code -> Spec:** nicht dokumentierter Export, Warnung.
- **Spec -> Code:** Phantom- oder veralteter Eintrag, fehlende Quelldatei, Typ-Mismatch, alles Fehler.

Es macht dasselbe für Schemas: deklarierte Datenbanktabellen und -spalten werden gegen die echte SQL geprüft, und eine Phantom-Tabelle oder -spalte schlägt fehl. Und es löst Referenzen über Projekte hinweg auf, also kann eine Spec in einem Repo von einem Modul in einem anderen über `owner/repo@module` abhängen und dieser Link wird ebenfalls überprüft.

Das Ding, das man festhalten sollte, ist, dass das *strukturell* ist. Es benotet nicht deine Prosa und generiert keine Tests. Es stellt eine direkte Frage: Stimmt die dokumentierte Oberfläche mit der echten Oberfläche überein, und beantwortet sie deterministisch. Das macht es sicher, es vor einen Agent zu stellen. Es gibt kein Urteil zu bestreiten; entweder die API stimmt mit der Spec überein oder nicht.

## Wie es auftaucht

Es taucht auf drei Wegen auf, und in dem Interview war die Antwort auf "wie taucht spec-sync täglich auf" lautete: alle. Drei Vordertüren zum gleichen Spec-Format, jede läuft woanders:

| Vordertür | Wo es läuft | Was es tut |
|---|---|---|
| `fledge spec` (nativ) | deine Maschine, innerhalb fledge | `init`, `check`, `list`, `show`: fledges eigene Validierung, kein Shell-Out |
| `specsync`-Binary | der Loop des Agents, vor einem PR | `specsync check` / `--fix`: das eigenständige Tool, das der Agent im Loop ausführt |
| `CorvidLabs/spec-sync@v4` | CI, beim Pull Request | gatete den Build, kommentiert Drift, blockiert bei Fehler |

Der Rest dieses Abschnitts sind diese drei Zeilen, eine nach der anderen.

**fledge kennt Specs nativ.** Spec ist eine von fledges Säulen. Die fledge-README ist wörtlich darüber: "Spec | `spec` | [spec-sync]. Modules declare their contract, AI uses it as context." Es lohnt sich, genau zu sein, was das im Code bedeutet: fledge hat sein eigenes `fledge spec` (`init`, `check`, `list`, `show`), direkt in das Binary eingebaut. Es gibt kein Shell-Out zum `specsync`-Tool. Es liest dasselbe `.specsync/config.toml` und dieselben `*.spec.md`-Dateien, parst sie selbst und macht seine eigene Prüfung. Also ist das Spec-Format geteilt, aber die Validierung innerhalb von fledge ist fledges eigener Code, kein Aufruf des separaten Binaries.

Diese native Spec-Awareness ist auch der Grund, warum fledges KI-Befehle auf den Vertrag zurückgreifen können. `fledge ask` ist spec-aware Q&A und `fledge review` ist spec-aware Code-Review, beide ziehen die relevanten Specs als Kontext heran. Die Spec ist der Kontext, den diese Befehle dem Modell übergeben, wenn es über deinen

Code nachdenkt. Die Spec ist kein Dokument abseits; sie ist das Ding, das das Tooling liest, wenn es über deinen Code nachdenkt.

**Es gatete CI.** Es gibt eine eigenständige GitHub Action, `CorvidLabs/spec-sync@v4`, die automatisch `specsync check` ausführt. Ein minimaler Workflow:

```
- uses: CorvidLabs/spec-sync@v4
  with:
    strict: 'true'          # warnings become errors
    require-coverage: '100' # minimum spec coverage
```

`strict` macht Warnungen zu Fehlern. `require-coverage` setzt eine Untergrenze. `comment` postet eine Spec-Drift-Zusammenfassung auf den Pull Request. Und der Check beendet sich mit Nicht-Null bei Fehler, was bedeutet, der PR wird geblockt. Das ist der Mechanismus, der das Ganze real macht: Drift erzeugt keine höfliche Notiz, es schlägt den Build fehl. Ein Agent oder ein Mensch kann den Spec und den Code nicht still auseinanderdriften lassen, weil das Auseinanderdriften *der* fehlschlagende Check ist.

**Es hält den Agent on-spec.** Das ist das, das mir am meisten wichtig ist, und es ist der Grund, warum die Spec im Loop des Agents lebt, nicht nur in CI. Die Mechanik ist konkret. Der Agent führt `specsync check` als Teil seines Loops aus, genauso wie er Build und Test ausführt. Der Check kommt mit strukturierten Fehlern zurück, und der Agent liest sie und entscheidet, was zu tun ist, und welchen Weg er einschlägt, hängt davon ab, in welche Richtung der Drift läuft.

Hier ist, wie dieser strukturierte Output aussieht, wenn beide Richtungen gedriftet sind (illustratives Beispiel, das dem echten Output-Format entspricht):

```
specsync check

CHECKING src/review.rs against review.spec.md
  WARNING  undocumented export: `ReviewOptions::with_model`
             → run `specsync check --fix` to add stub

  ERROR    phantom export: `ReviewFormat::Detailed`
             spec claims this variant; code has: Summary, Checklist, Inline
             → update spec or add the variant to code

  ERROR    phantom export: `run_async`
             spec claims this function; not found in src/review.rs
             → update spec or add the function

SUMMARY  2 errors, 1 warning
EXIT 1
```

Das ist, was der Agent liest. Die Fehler benennen die Datei, den Spec-Anspruch und was der Code tatsächlich hat. Die Warnungen benennen den Export und den genauen Fix-Befehl. Kein Raten erforderlich: entweder den Code anpassen, um der Spec zu entsprechen, oder die Spec korrigieren, dann erneut ausführen bis Exit 0.

Wenn der Code einen Export gewachsen hat, den die Spec nicht erwähnt, die Warnungsrichtung, editiert der Agent die Spec nicht von Hand. Er führt `specsync check --fix` aus, was die nicht dokumentierten Exports als Stubs zur Spec hinzufügt, und der einfache Fall ist in einem Schritt abgestimmt. Die Aufgabe des Agents ist dabei meist, den Stub mit einer echten Beschreibung zu füllen, nicht den Drift überhaupt erst zu entdecken.

Die andere Richtung ist diejenige, die echte Arbeit erfordert. Wenn die Spec eine API behauptet, die der Code nicht honoriert, die Fehlerrichtung, der veraltete oder Phantom-Eintrag, gibt es kein `--fix` dafür, weil der Fix ein Urteil ist: entweder ist der Code falsch und der Agent geht hin und lässt den Code das honorieren, was versprochen wurde, oder die Spec war aspiratorisch und der Agent korrigiert den Vertrag. So oder so muss der Agent es im Code abstimmen, absichtlich, und den Check erneut ausführen, bis er grün ist. Das ist der Loop in der Praxis: Vertrag, Änderung, Check, und dann entweder ein Auto-Add oder eine echte Korrektur, je nachdem, welche Seite aus dem Schritt gefallen ist, und er läuft, bevor der Diff jemals einen Pull Request erreicht, innerhalb von Merlins Loop, nicht nachher in CI.

## Was spec-sync nicht prüft

Es gibt eine Grenze, über die ich ehrlich sein möchte, weil sie der Rand dessen ist, was dieses Tool tut. spec-sync prüft, dass Spec und Code übereinstimmen. Es hat keine Meinung dazu, ob die Spec gut ist.

Überlege, was das offenlässt. Eine Spec kann die richtigen Exports benennen und sie falsch beschreiben. Sie kann dünn sein, ein Purpose-Abschnitt, der nichts sagt, und eine Public-API-Tabelle, die korrekt ist und dir nichts darüber sagt, wofür das Modul ist. Sie kann aspiratorisch sein, geschrieben für den Code, den jemand schreiben wollte. Und wenn ein Agent die Spec aus einem Aufgaben-Brief entworfen hat, der selbst falsch oder injiziert war, kann die Spec ein treuer Vertrag für die falsche Sache sein. In jedem dieser Fälle besteht spec-sync. Die Oberfläche stimmt mit der Oberfläche überein. Die Prüfung ist grün. Der Vertrag ist falsch, und nichts im Loop hat es aufgefangen, weil das Prüfen der Spec gegen den Code dir nicht sagen kann, dass die Spec von Anfang an schlecht war.

Das ist eine echte Lücke, und ich benenne sie als solche. Der Code hat eine Prüfung, die Spec nicht. Was man möchte, ist ein zweites Gate, das auf der Spec selbst läuft, bevor ein Agent jemals dagegen baut: Hat jeder erforderliche Abschnitt tatsächlich etwas zu sagen, zeigt die Public API auf Dateien, die existieren, gibt es eine klare Aussage darüber, wie Erfolg und Scheitern aussehen, und hat ein Mensch diesen Vertrag tatsächlich abgezeichnet. Das wäre ein `fledge spec lint`, und er ist noch nicht gebaut. Bis er es ist, ist die Spec die einzige Eingabe in diese gesamte Pipeline, die nichts validiert, und das ist es wert zu wissen, wenn man der grünen Prüfung vertraut.

## Warum ein geteilter Vertrag der ganze Punkt ist

Das führt direkt zu der These unter all diesen Tools: Menschen und Agents, die dieselben Tools benutzen, als gleichwertige erstklassige Bürger. Eine Spec ist die klarste Version dieser Idee, die ich habe. Der Mensch schreibt oder reviewed sie; der Agent baut dagegen; spec-sync hält beide daran fest, in beide Richtungen, deterministisch. Der Mensch, der refaktoriert hat ohne das Dokument zu aktualisieren, wird genauso gefasst wie der Agent, der eine API halluziniert hat. Der Check interessiert sich nicht, wer gedriftet ist.

---

# fledge in Rust bauen

fledge ist ein einzelnes Binary. Du installierst es einmal und es läuft auf jeder Maschine, jedem OS, ohne dass sonst irgendetwas installiert werden muss. Keine Runtime einzuziehen, kein Interpreter, kein Dependency-Manager, der bereits vorhanden sein muss. Cross-Platform standardmäßig: eine Build-Pipeline produziert ein Binary, das auf macOS, Linux und Windows funktioniert. Das ist die Kernanforderung, und Rust ist der Grund, warum es einfach statt ein Kampf ist. Das ganze Kapitel folgt daraus: warum Rust die richtige Wahl war, warum das Erlernen keine Kriegsgeschichte war, wie die Leute es erwarten, und wie Agents die Lücke in der Flüssigkeit schlossen.

Ich hatte etwa ein Jahrzehnt Swift geschrieben, als ich mit fledge anfing. Also ist die ehrliche Version dieses Kapitels ein wenig unspektakulär: Rust aufzunehmen hat nicht wehgetan. Nichts Großes hat sich gewehrt. Nicht einmal der Borrow Checker, das ist der Teil, vor dem alle warnen.

Ich glaube, die Leute erwarten hier eine Kriegsgeschichte: die Sprache, die mich gedemütigt hat, den Monat, den ich damit verbracht habe, Kämpfe mit dem Compiler zu verlieren. Die habe ich nicht. Und ich erzähle dir lieber, warum es reibungslos lief, als das Drama zu erfinden.

## Swift hat mich vorbereitet

Vieles in Rust fühlte sich vertraut an, weil Swift mich bereits auf dieselben Ideen trainiert hatte, nur in anderen Kleidern.

Enums und Pattern Matching waren das Größte. Swifts Enums sind echte Sum Types, und ich hatte mich jahrelang auf sie und `switch` gestützt. Rusts `enum` und `match` sind derselbe Muskel. Result und Option waren auch nicht neu. Ich hatte lange in Swifts Optionals und `Result` gelebt, also waren "das kann ein Wert oder nichts sein" und "das kann ein Wert oder ein Fehler sein" bereits die Art, wie ich über Code nachgedacht hatte. Rust macht nur, dass man beides handhabt, die ganze Zeit, laut. Das war keine neue Disziplin für mich; es war eine Disziplin, die ich bereits hatte, jetzt durchgesetzt.

Traits landeten als ungefähr Swifts Protokolle. Nicht identisch, aber nah genug, dass ich kein fremdes Konzept lernte. Ich lernte einen Dialekt eines, den ich kannte. "Verhalten definieren, Typen dazu konformieren" hat in beiden dieselbe Form.

Also fühlte die Sprache sich nicht wie eine Wand an. Es fühlte sich wie ein Ort an, an dem ich halb schon gelebt hatte. Die Werttyp- und Optionals-Gewohnheiten, die Swift mir eingebohrt hat, sind, glaube ich, genau der Grund, warum der Borrow Checker nie ein Kampf wurde. Wenn du bereits in Begriffen von Eigentümerschaft denkst und "wer hält diesen Wert", sagt der Checker dir meistens Dinge, die du bereits zu tun versuchst. Es war keine neue Denkweise, die ich erwerben musste. Es war ein strengerer Schiedsrichter für ein Spiel, das ich bereits zu spielen wusste.

Und die Form des Dings, das es baut, ist die einfache, die ich wollte: fledge ist ein einzelnes Rust-Binary, mit Cargo gebaut. Ein Crate, ein Binary am anderen Ende.

## Agents haben die schwere Arbeit gemacht

Ich werde nicht so tun, als wäre ich ein fließender Rust-Programmierer. Das bin ich nicht. Ich bin ein fließender Swift-Programmierer, der Rust gut lesen und steuern kann, und das ist eine andere Sache.

Was die Lücke schloss, waren Agents. Ich lehnte mich stark auf sie, um in einer Sprache produktiv zu sein, in der ich weniger fließend bin. Die vertrauten Teile konnte ich lesen, darüber nachdenken und von mir aus steuern. Ich wusste, was ich wollte, dass der Code *tut*, und wie gute Struktur aussieht, weil der Teil sprachübergreifend übertragbar ist. Die Teile, wo Rust seine eigenen Idiome hat, seine eigene Art, eine Sache zu sagen, trugen die Agents.

Diese Unterscheidung ist es wert zu ziehen, nur von der anderen Seite gesehen. Mein Swift ist handgeschrieben. Mein Rust ist agent-verstärkt. In einem bin ich der Autor an den Tasten. Im anderen bin ich derjenige, der weiß, wie richtig aussieht, Agents darauf richtet und ihre Arbeit überprüft.

Und ehrlich gesagt ist das Bauen von fledge in Rust auf diese Weise ein kleiner Beweis der gesamten These dahinter. Das Tool ist für Menschen und Agents gleichermaßen gebaut. Es wurde *von* einem Menschen und Agents gebaut. Die Agents, die fledge heute steuern, halfen, fledge überhaupt erst zu schreiben.

## Das Tooling war eine Erleichterung

Hier ist der Teil, den ich nicht erwartet hatte, so sehr zu genießen: Rusts Tooling fühlte sich wie eine Erleichterung an.

Cargo funktioniert einfach. Ein Tool zum Bauen, Testen, für Dependencies, das ganze Ding, und es ist überall dasselbe. Das Crate-Ökosystem ist tief. Was auch immer ich brauchte, es gab meistens ein solides Crate dafür, und es einzuziehen war eine Zeile.

Und Single-Binary-Builds sind genau das, was fledge sein wollte: Ich musste ein leichtes Binary liefern, das überall läuft, und Rust gibt dir das standardmäßig.

Der Kontrast ist mit Swift-Tooling, sobald du von Apples Plattformen abweichst. Auf Apple ist die Swift-Geschichte großartig. Abseits davon (Linux, Cross-Platform, das "läuft überall als ein Binary"-Target, das fledge brauchte) wird es schwieriger. Das ist ein echter Grund, warum fledge Rust und nicht Swift ist, obwohl Swift meine Heimatsprache ist. Ich wollte ein leichtes, einzelnes Binary, das ich auf jede Maschine droppen kann, das Agents überall ausführen können, und Rusts Toolchain machte das zum einfachen Pfad statt zum Kampf.

Das ist also die Builder's-Eye-Wahrheit davon. Swift hat das Denken vorbereitet, Agents haben die Flüssigkeit gedeckt, die ich nicht habe, und das Tooling (Cargo, die Crates, das einzelne Binary) war der Teil, der mich tatsächlich froh machte, dass ich gewechselt hatte. Die Sprache zu wählen war die einfache Entscheidung. Die Arbeit, die tatsächlich Nachdenken erforderte, war alles andere: herauszufinden, was fledge sein sollte.

---

# Wie ich es nutze und wer es nutzt

Lass mich mit dem ehrlichen Teil anfangen, statt ihn für das Ende aufzusparen. fledge wird nicht weit verbreitet genutzt. Es gehört mir, meinen Agents, meinem Kreis. Das ist der Punkt. Das Tool wurde gebaut, um das tatsächliche Problem des Builders zuerst zu lösen, und es tut es jeden Tag. Ein Tool, das das Problem vor dir löst, schlägt ein Tool mit einer Logo-Wand und ohne Hund im Kampf.

Also ist die Frage, die dieses Kapitel beantwortet, nicht "wie viele Menschen benutzen fledge." Es ist "wer, und warum das die richtige Reihenfolge ist." Die Antwort ist nicht "jeder," und das sollte es nie sein.

## Meine Agents steuern es öfter als ich

Das Ding, das Leute falsch machen, wenn sie sich vorstellen, wie ich fledge benutze, ist, dass sie sich *mich* vorstellen, wie ich es benutze. Das passiert, aber es ist nicht die Hauptweise, wie fledge noch läuft. Meine Agents steuern es, und zwar mit großem Abstand. Wenn ein Agent an einem Repo arbeitet, ist fledge die Art, wie es baut, testet und ausführt, was es gerade geändert hat: seine Ausführungsoberfläche, nicht nur meine. Das ist genau das, was das erste Kapitel sagte, wofür das Design gedacht war. An den meisten Tagen bekommen die Agents mehr Kilometer daraus als ich, und ich steure meist.

Das ist die Nutzerbasis, für die ich gebaut habe. Keine Menge Fremder. Ich plus die Agents, in jedem Repo das ich habe, die ganze Zeit.

## Wer es sonst benutzt

fledge ist Open Source, es ist auf dem Homebrew-Tap, jeder kann `cargo install` es. Nichts davon ist ein Adoptions-Anspruch. Ein Tap ist ein Distributionskanal, kein Nutzerzähler, und ich werde das eine nicht als das andere aufhübschen.

Heute ist die Nutzerbasis ich, meine Agents und der CorvidLabs-Kreis, die Menschen, mit denen ich tatsächlich arbeite. Es gibt keine breite externe Adoption. Keine Community von Fremden, die Issues einreichen. Es ist persönliche und Kreis-Infrastruktur, und das ist es wert, es direkt zu sagen, statt eine Grundströmung zu implizieren, die es nicht gibt.

Die Mitarbeiter sind wichtig für das Bild, nicht nur als Kopfzahl. fledge als geteilte Oberfläche über den Kreis zu sein ist Teil des Designs. Wenn jemand in CorvidLabs eines meiner Repos aufgreift, muss er nicht den privaten Dialekt dieses Repos lernen. Er kennt bereits die Verben, weil die Verben überall dieselben sind. Dieselbe Konsistenz, die mir das Wiedereinarbeiten erspart, landet die Menschen, mit denen ich arbeite, auf einer Oberfläche, die sie bereits kennen.

## Solltest du es benutzen

Vielleicht noch nicht. Ehrlich gesagt, nur wenn du so arbeitest wie ich: viele Repos, Agents in der Schleife, eine konsistente Oberfläche über alles. Wenn das dein Problem ist, löst fledge es. Wenn es das nicht ist, ist das Tool Overkill, und ich würde es dir lieber sagen als es dir zu verkaufen.

Was ich nicht beanspruchen werde, ist, dass begrenzte Adoption beweist, dass das Design im großen Maßstab funktioniert. Das tut es nicht. Es beweist, dass das Design für mich funktioniert, was ein kleinerer Anspruch und der einzige ist, hinter dem ich stehen kann. fledge verdient seinen Platz in meiner eigenen Welt zuerst, jeden Tag, meistens durch Agents, während ich steure und ein kleiner Kreis auf derselben Oberfläche. Das ist ein Tool, das seinen Job macht. Die Breite kann später kommen oder nie. Der Job wird bereits erledigt.

---

# Wohin fledge geht

Die ehrliche Antwort auf "wohin geht fledge als nächstes" ist weniger aufregend, als die Leute erwarten, und ich denke, das ist ein gutes Zeichen. Es ist größtenteils ausgereift. Die große Form davon ist gebaut. Was bleibt, ist meistens, es zu pflegen und es weiter zu dogfooden. Es gibt ein paar Richtungen, in die ich es wachsen lassen würde, aber ich sitze nicht auf einer großen Roadmap der Neuerfindung, weil fledge keine Neuerfindung braucht. Es muss weiterhin das Ding sein, auf dem alles andere reitet.

## Ein größeres Ökosystem

Die Spielplatz- und Integrations-Tiers sind offen: jeder kann ohne Nachfragen dazu beitragen. Der Core-Tier ist es nicht. Diese Asymmetrie ist absichtlich und wird sich nicht ändern.

Die Richtung mit dem meisten Raum ist die, die das Ökosystem-Kapitel bereits beschrieben hat: mehr Plugins. fledge wird fähiger, indem der Ring um den Core wächst, nicht indem der Core dicker wird: ein reicheres Set von Plugins, damit, egal in welches Repo du fledge dropps, es bereits etwas gibt, das weiß, wie man damit umgeht. Mehr Integrationen, mehr der kleinen Einzel-Stücke, die jeweils eine echte Sache lösen.

Damit verbunden ist die Erweiterung dessen, was fledge out of the box erkennt und ausführt, das Zero-Config-Versprechen aus dem ersten Kapitel, an mehr Orten gehalten. Jede Sprache und Plattform, die es noch nicht kennt, ist ein Repo, wo das "same-verbs-just-work"-Versprechen eine Lücke hat, also ist das Füllen von Erkennung und Plattformabdeckung die andere offensichtliche Richtung. Nicht glamourös. Abdeckung ist einfach das, was eine universelle Oberfläche tatsächlich universell macht.

Und die Abdeckung, die jetzt am meisten wichtig ist, ist nicht die nächste Sprache. Es ist das Betriebssystem. fledge muss auf allen drei erstklassig laufen: Windows, Linux und macOS. Ein Tool, das nur wirklich auf einem OS zu Hause ist, ist keine universelle Oberfläche, es ist eine lokale. Also ist Cross-OS-Support die nächste echte Grenze, mehr als das Hinzufügen eines weiteren Sprachdetektors.

## Was es bedeutet, etwas zu pflegen, von dem Agents abhängen

Ich könnte hier Features auflisten und eine Neuerfindung versprechen. Das wäre der falsche Rahmen für das, was fledge jetzt ist.

Die wichtige Frage im Jahr 2026 ist nicht, welche Features fledge hinzufügt. Code ist billig. Agents schreiben das meiste davon. Was knapp ist, ist ein Tool, dem du tatsächlich vertrauen kannst, deine Arbeit zu erledigen. Ein Agent kann neue Funktionalität schneller generieren, als ich sie reviewen kann. Was er nicht generieren kann, ist ein Tool mit Jahren des Dogfoodings dahinter, einer stabilen Schnittstelle, Verhalten, auf das eine Pipeline sich verlassen kann.

Ein Tool mit Jahren des Dogfoodings und einer stabilen Schnittstelle, auf die eine Pipeline sich verlassen kann, ist was fledge ist, und es ist das Schwierigere zu bauen. Nicht technisch schwieriger. Schwieriger, weil es erfordert zu entscheiden, dass Stabilität der Output ist, kein Nebenprodukt. Jedes Tool beginnt als neues Feature. Sehr wenige überleben lange genug, um Infrastruktur zu werden. Die, die es tun, sind die, bei denen der Betreuer "nichts ist kaputt gegangen" als die Hauptleistung behandelt hat, nicht als Trostpreis für einen langsamen Sprint.

Ich benutze fledge in jedem Repo, die ganze Zeit. Meine Agents steuern es konstant. Das bedeutet, die Bugs finden mich. Das fehlende Verb, die Erkennung, die falsch geraten hat, der Plugin-Hook, der in der falschen Reihenfolge feuerte. Ich treffe es, weil ich darauf lebe, und ich behebe es, weil ich darauf lebe. Das Dogfooding ist keine Nebenaktivität. Es ist der gesamte Qualitätsmechanismus. Pflegen und Benutzen sind derselbe Job.

Die Einleitung dieses Buches fragte, was ein Tool vertrauenswürdig genug macht, um davon abzuhängen. Die Antwort, auf die sie hinwies, war: eine saubere Oberfläche, keine versteckten interaktiven Schritte, eine echte Plugin-Grenze, eine Spec, die ehrlich bleibt, während sich der Code verändert. Bau das, und ein Agent kann es vom ersten Tag an steuern, weil es von Anfang an nicht so gebaut wurde, dass es einen Menschen braucht.

fledge ist das. Nicht wegen einer einzelnen Designentscheidung, sondern wegen des kumulierten Ergebnisses jedes Mal, wenn ich mich geweigert habe, einen Shortcut die Schnittstelle erodieren zu lassen. Die Verben sind in jedem Repo dieselben. Das JSON hat dieselbe Form. Das `fledge introspect`-Manifest bleibt aktuell. Nichts wird ausgeliefert, das ich nicht zuerst an meiner eigenen Arbeit ausgeführt habe.

Wenn Agents diejenigen sind, die das meiste Tooling betreiben, ist das, was Vertrauen verdient, keine lange Feature-Liste. Es ist das gleiche Auftreten, jedes Mal, lange

genug, dass etwas Echtes darauf aufgebaut wurde. Das ist ein leiserer Ehrgeiz als eine Roadmap-Folie, und es ist derjenige, der wichtig ist.

Also ist der Job von hier aus der Job, der es immer war: es solid halten, weiter darauf zu leben, die Oberfläche ehrlich halten. Ich werde die Bugs finden, weil ich es weiter benutze, und ich werde sie beheben. Das ist der ganze Plan.

---

# Über den Autor

0xLeif (leif.algo) baut im Offenen. Ein Jahrzehnt kleiner, komponierbarer Swift-Bibliotheken wie AppState, Cache und Fork. Das CorvidLabs-Lab. Ein Stack von Agent- Tools, die größtenteils mit "ich wünschte, das würde existieren" anfingen. 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

Dank an CorvidLabs, dafür, der Raum zu sein, wo diese Ideen getestet und in Form gestritten werden.

Dank an die Open-Source-Betreuer, deren Tools auf dem ganzen Stack stehen. Nichts davon wird alleine gebaut.

Und Dank an die frühen Leser und die Pay-what-you-want-Unterstützer, die "kostenlos online" zu etwas machen, das ich weiterhin tun kann.

---

# Kolophon

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

Interview-gesteuert und KI-unterstützt; von Hand editiert und auf Fakten geprüft. Ohne Gedankenstriche geschrieben. Cover und Kapitelkunst aus den Corvid- und Nature-Kollektionen auf Algorand.