Open Source Tooling

実際に使われるツールを作る

ZACH "LEIF" ERIKSEN

著作権

オンラインで無料公開。ePub版は「払いたい分だけ払う」形式です。役に立てば、活動を支援していただけます。

github.com/0xLeif · leif.algo

agent-stackシリーズ全4冊のうちの1冊。制作の詳細は巻末のコロフォンをご覧ください。

献辞

オープンに作り、それでも出荷し続けるすべての人へ。

ライブラリ

これらの本はそれぞれ単独で読めるが、1つのセットとして書かれた。コードが安くなり、信頼が希少になった。まとめると1つの主張になる。今何を作るべきか、そしてそれをどう信頼するか。

- **The Agent Developer's Field Guide:** 実際のコードを出荷するエージェントのためのツール・スペック・信頼を構築する
- **First-Class:** 人間とエージェントの両方のために作る
- **Building Agents:** ソフトウェアに自分の手を与えようとした記録
- **Open Source Tooling:** 実際に使われるツールを作る *(本書)*

オンラインで無料公開。各ePubは「払いたい分だけ払う」形式。

目次

はじめに

本書は、他の人々、そして他のエージェントが実際に使うツールを作る技術について書かれたものだ。

開発ライフサイクル全体をカバーする1つのコマンドラインツール、fledgeを中心に構成されており、それが常に私に答えさせ続けた問いがある。ツールが依存する価値を持つとはどういうことか？その答えは人間にとってもエージェントにとっても同じだというのが主張だ。良いツールは1つのクリーンなインターフェースを持ち、隠れたインタラクティブなステップがなく、本物のプラグイン境界があり、コードが変わっても誠実であり続けるスペックを持つ。それを作れば、エージェントは初日から動かせる。なぜなら、そもそも人間を必要とするように設計されていないからだ。

本書は2冊のエビデンス本のうちの2冊目だ。*First-Class*はソフトウェアが両方にとって一流であるべきだと主張する。*Field Guide*はその方法論に変える。*Building Agents*はエージェントを見せる。本書はその下にあるツール群を示す。なぜRustが正しい選択だったか、そしてWASMサンドボックスがプラグインにやってはいけないことをどうやって防ぐかも含めて。

ツールを出荷し、自分が想定していなかった使われ方をするのを見たことがある人のために書かれている。fledgeは必要ない。必要なのは、人間ではないユーザーも含めて、自分以外のユーザーとの接触に耐えられるツールを作る習慣だ。

ライフサイクル全体をカバーする1つのCLI

どのリポジトリも異なる方言を話していた。異なるMakefile、異なるスクリプト、異なるREADMEが、それぞれ独自のビルド・テスト・実行の作法を持っていた。何も転用できなかった。しばらく触れていなかったプロジェクトを開くと、最初の仕事はそのローカルな呪文を解読することだった。どのスクリプトか、どのターゲットか、どの順序か。作業そのものは難しくない。テストを実行するのは難しくない。問題は、どこでも違っていて、その違いが純粋なオーバーヘッドだったことだ。

だから、すべてにわたって一貫したインターフェースが欲しかった。それがfledgeの出発点だ。リポジトリのキャッチフレーズはシンプルに言っている。*1つのCLI、開発ライフサイクル全体。*

3つのことが私をそこへ向かわせた。

1つ目はその「同じでない」問題、各リポジトリが独自の言語を話していること。2つ目はブートストラップだ。すべての新しいプロジェクトに同じセットアップとスキャフォールディングをコピーペーストし続けていた。同じファイルの山、同じ配線、実際に作りたいものに取り掛かる前に何度も何度も。3つ目はスケールだ。エージェントが作業するプロジェクトを大量に立ち上げていて、すべてが依存できる1つのCLIが必要だった。「私が使うツール」ではなく。すべてのリポジトリで同じように依存できるツールとして、すべてとすべての人が使えるもの。

最後の点は見落とされやすい。fledgeは1つのリポジトリへの不満から生まれたのではない。多くのリポジトリを持ち、多くのプロジェクトが進行中で、エージェントがそこに加わっており、それらすべてが百個のカスタムインターフェースの代わりに1つの一貫したインターフェースに依存できる必要があったことから生まれた。エージェントが1つのCLIに依存できるという側面はそれ自体のスレッドであり、エージェントの側については agents の本で詳しく述べる。一貫性、スキャフォールディング、スケール、それが起源だ。

人間とエージェントの両方のために作られた

これらすべての下にあるアイデアがあって、それはfledgeより大きい。私のツールの多くは、人間とエージェントが同じツールを使うようになるという信念から来ている。

現在存在するものの多くは人間優先だ。プロジェクトは人のために作られ、後からエージェントを持ち込もうとする。エージェント優先のものもあれば、人間優先のものもある。でも本当に必要なのは、エージェントと人間の両方が最初から一流であるプロジェクトを作ることだ。

それが私のすべてのツールだ。ツールはどちらの方向でも一流に機能すべきで、人間はエージェントなしで使え、エージェントは人間なしで使える。どちらも後付けではない。そしてエージェントがそれを使うとき、ツールはエージェントを*助ける*べきであって、すべてのコマンドとその仕組みを推測させるべきではない。

人間とエージェントを対等なファーストクラスの呼び出し元として設計することが、fledgeが今の形になっている本当の理由であり、本書の残りを読む上で念頭に置くべきことだ。

ゼロコンフィグが本当に意味すること

ゼロコンフィグというとき、プロジェクトについてツールに教えてから助けてもらう必要がないことを意味する。リポジトリに投入すれば動く。3つの要素がある。

プロジェクトとそのコマンドを自動検出する。Swift、Rust、Nodeなど何でも。どんな種類のプロジェクトかを把握して、適切なbuild/test/runを知っている。最初にコンフィグファイルにプロジェクトを説明するセットアップ手順はない。

同じ動詞がすべてのプロジェクトで機能する。`build`、`test`、`run`、`lint`、何が下にあっても同じ言葉だ。それが「各リポジトリが異なる方言を話していた」問題への直接的な回答だ。方言は下に残っている。CargoはまだCargoであり、Swift Package Managerはまだ Swift Package Managerだ。でも、どの前に立っているかを気にしなくて済むようになる。動詞を一度覚えれば、どこでもその動詞が通じる。検出が実際の下位ツールへの翻訳を担う。

これが人間とエージェントというアイデアを具体化したものだ。人間は各リポジトリを再学習しなくて済む。エージェントは*推測しなくて済む*。この特定のプロジェクトで正しいコマンドを探し回ったり、READMEを読んで祈ったりしなくていい。`build`を実行し、`test`を実行すれば、ツールはここでそれが何を意味するかをすでに知っている。再学習の手間を省いてくれるものが、エージェントが推測しなくて済む理由でもある。fledgeはまず私と私が動かすエージェントのために作られ、そこで最初に価値を証明する。

そしてプラグインで拡張できる。新しいリポジトリはコアを手に入れ、プラグインを追加することで機能を加えていく。コアは一般的な動詞と一般的なプロジェクトタイプの検出方法を知っている。それ以上はプラグインから来る。だからゼロコンフィグとは「すべてをすぐに備えている」という意味ではない。すぐに得られる部分は何もコンフィグを必要とせず、そこからコンフィグを書くのではなくプラグインを追加することで拡張していくことを意味する。

具体的には、最初の実行はこのようになる。リポジトリに入って`fledge`を実行すると、プロジェクト（Swift、Rust、Nodeなど何でも）を自動検出し、コンフィグなしで適切なbuild/test/runを知っている。イントロスペクトを求めると、このリポジトリで*利用可能な動詞を教えてくれる*。自己記述的なので、（あなたまたはエージェントは）何が可能かを推

測する必要がない。まったく新しいプロジェクトであれば、最初の本当の動きは既存のものを検出するのではなく、テンプレートからスキャフォールディングすることが多い。そして全体が最初のコマンドからヘッドレスで動く。`FLEDGE_NON_INTERACTIVE`を設定し、`--json`を要求すれば、エージェントがステップ1から動かせる。検出、イントロスペクト、おそらくスキャフォールディング、そして実行。

ライフサイクルの部分

リポジトリはfledgeをスキャフォールディングとAIレビューを備えたゼロコンフィグタスクランナーと呼んでいる。タスク実行とは上記の動詞セットだ。日々のbuild/test/run/lint。スキャフォールディングは2つ目の起源問題への回答だ。毎回手でコピーペーストして同じセットアップを新しいプロジェクトに持ち込む代わりに、fledgeが立ち上げてくれる。そしてライフサイクルにはAIレビューの要素も組み込まれている。

タスク実行が最初だった。それが種で、他のすべてがそこから育った。スキャフォールディングは実質的にテンプレートだ。コピーペーストする代わりにテンプレートから新しいプロジェクトを立ち上げる。そしてAIレビューが組み込まれたのは、レビューが開発ループの一部だからだ。エージェントがコードを書いているなら、採点は別の儀式として別の場所で行うものではなく、buildとtestの隣にある単なる別の動詞だ。1つのインターフェースが3つのツールを上回り、3つのことを別々に学ばなければならないエージェントに対してはさらにそれが顕著だ。この3つが構造的な柱であり、コアであり、プラグインがその周りの拡張層だ。

fledgeの独自のワンライナーは「1つのCLI、開発ライフサイクル全体、ゼロコンフィグタスクランナー、プロジェクトスキャフォールディング、AIレビュー、その他」であり、それがまさにその3つのピースだ。単一のRustバイナリであり、退屈な明白な方法で入手できる。Homebrewのタップから`brew install CorvidLabs/tap/fledge`、好むなら`cargo install fledge`、またはシェルスクリプト。インストールに特別なものは何もない。

私の実際の使い方

これは時々手を伸ばすツールではない。すべてのリポジトリで、常に、今ではすべてをbuild・test・runするデフォルトの方法だ。そして私のエージェントがそれを動かす。エージェントは私が手動でやるより多くfledgeを動かしている。多くのリポジトリを手なずけるために作られ、今は私と私がそれらのリポジトリで動かしているエージェントの両方が依存する1つのCLIになっている。どこでも。

それが次の要素が重要な理由でもある。誰もが依存するCLI、誰でも拡張できる、エージェントがインストールして実行する、それは私が一度に承認する機能の固定セットであってはならない。コアに触れることなく、好きな言語で、誰でも拡張できなければならない。そ

して任意のプラグインを受け入れ、エージェントがそれを実行するなら、信頼について真剣に考える必要がある。それが次の章だ。

Makeではなくfledgeをなぜ選ぶのか

fledgeが何をするかを聞いた瞬間、人々はこれを尋ねる。タスクランナーを作ったのか？Makeがある。Justがある。Turborepoがある。すべてのpackage.jsonに入っているnpmのscriptsブロックもある。なぜもう一つ作るのか？

正直な答えは、私がぶつかった壁はMakeではなかった。私はiOS開発者として育ち、そのビルドを自動化しようとすると同じ場所にたどり着き続けた。fastlane、つまりRubyだ。Rubyは欲しくなかった。私はSwiftの人間だ。自分が使う言語でビルドを自動化したかった。それなのに、エコシステム全体がRubyのDSLと`Gemfile`、独自の世界を持つツールへと私を誘導した。何かを出荷したいたびに答えは「fastlane laneを書け」で、すべてのfastlane laneは書きたくないRubyだった。管理したくないランタイムで動き、Swiftでできるはずだと感じる仕事をしていた。それがfledgeが実際に育った痛みだ。「Makeがめんどくさい」ではない。「自分のプロジェクトを自動化するのになぜ他人の言語を強いられるのか？」だった。

だから人々がfledgeをMakeやJustと並べて比べるとき、ターゲットが外れている。その答えはそれらのツールが悪いということではない。タスクランナーに求めていたものは3つあり、3つすべてはfastlaneの壁から来ていた。

自分のやり方で、自分の言語で、自分の条件で、fastlaneがRubyに縛るように1つのランタイムに縛られずに使いたかった。タスクランナーは1日に何百回も触れるものであり、しばらくすると他人の感覚や使うべき言語についての選択の中に住みたくなくなる。`build`と`test`と`run`がすべてのリポジトリで同じことを意味してほしかった。そして下のステップはRubyに行進させられる代わりに、仕事に合った言語で書けるようにしてほしかった。Makeはそれを止めないが、Makeはそれを提供もしない。一貫性を自分で各Makefileに手で作り、永遠に作り続ける。プロジェクトごとに方言を再発明させてくれるツールは私の問題を解決していない。ただ再発明を続けるより良い場所を与えてくれているだけだ。

他の2つは最初の章から来るものであり、fastlaneファイルにもそれらが欲しかったところだ。それらのツールのどれもエージェントファーストではない（デフォルトでJSON、イントロスペクション、ヘッドレスモード）。それは今やエージェントが私より手動でこれを動かしているので最も気にする部分だ。そしてそれらのどれも、追加のランタイムをインストールせずにリポジトリに投入できる単一の軽量バイナリではない。それはまさに異なる言語のリポジトリの山にわたって一貫したインターフェースを持つために必要なものだ。fastlane laneはRubyが必要だ。npmスクリプトはnpmが必要だ。TypeScriptで書くランナーはTypeScriptランタイムが必要だ。fledgeは何も必要としない。

これはどれも「Makeが悪い」ということではない。MakeはMakeが得意なことが得意で、Justはクリーンなコマンドランナーで、Turborepo はJSモノリポをうまくキャッシュする。それらのどれかが必要なものすべてなら、使えばいい。fledgeがそれらのホームグラウンドでの機能対機能の戦いに勝つとは言わない。

でも、それらのどれも、数年前にfastlaneファイルの前に立ったときに実際に欲しかったものではなかった。1つのインターフェース、エージェントファースト、単一の軽量バイナリ、そしてRubyに誘導される代わりに適した言語でステップを書く自由。fledgeは当時あれば良かったと思っていたツールで、ついに作られた。

---

fledgeとMCP

MCPは今や環境の標準だ。2026年、エージェントはほとんどMCPサーバーを通じてツールを利用している。ツールを作り、エージェントが確実にそれを使えるようにしたいなら、最小抵抗の道はMCPサーバーを公開することだ。それがエコシステムの現状だ。

fledgeにはMCPサーバーがない。それでもエージェントは何十ものリポジトリでfledgeを常に動かしており、うまく機能している。その理由は明示的に述べる価値がある。それが最初にどの部分を作るべきかについて何かを教えてくれるからだ。

CLIがプリミティブだ

MCPサーバーはプロダクション向けのインターフェースだ。リクエストのルーティングを処理し、ログを取り、エージェントが何を求めて何を得たかについて構造化された観測可能性を提供する。それらは持っていて良いものだ。でもMCPサーバーは何かの上に置くレイヤーだ。問いは、何の上に置くか？

MCPサーバーを最初に作ってCLIを後付けにすると、エージェントには機能するが人間には不便なツールができる。CLIを最初に作ると、プロトコルアダプターなしで今すぐエージェントに機能し、人間にも機能し、必要なときはいつでもMCPサーバーを前に置けるツールができる。CLIがプリミティブだ。他のすべてはその上に乗る。

fledgeはそれを念頭に置いて作られた。すべてのコマンドが`--json`を出力する。ヘッドレス実行のための`FLEDGE_NON_INTERACTIVE`がある。`fledge introspect`は呼び出し元が人間でもエージェントでも、利用可能なすべての動詞、その機能、受け付けるものの構造化されたマニフェストを提供する。アンアテンデッドな実行をブロックするインタラクティブなプロンプトはない。ツールは自己記述的だ。

そのプロファイル、デフォルトで構造化された出力、明示的な機能マニフェスト、隠れたインタラクティブなステップがないこと、それがまさにMCPツールインターフェースがエージェントに提供するものだ。fledgeはプロトコルなしでそれらすべてを持っている。なぜならそれらの特性が後からツールをエージェントフレンドリーにラップしたものではなく、最初からエージェントの呼び出し元を念頭に置いた設計から来ているからだ。

今日fledgeを動かしているClaudeのインスタンスは`fledge introspect`を呼び出し、利用可能なもののJSONマニフェストを受け取り、適切な動詞を選び、`--json`を渡し、構造化された出力を読む。それはMCPサーバーが仲介するのと同じインタラクションループで、MCPのフレーミングを除いたものだ。CLIはすでにクリーンなツールインターフェースだ。

MCPは競合相手ではない

時々出てくるフレーミング、「CLIかMCPか？」は両者を代替として扱う。そうではない。MCPはトランスポートとプロトコル仕様だ。CLIは仕事をするものだ。問いはどちらを持つかではなく、どちらを最初に作ってどちらをその上に重ねるかだ。

まず CLI を作れ。自己記述的にし、JSON を出力させ、ヘッドレスで実行できるようにしろ。それが揃えば、MCP サーバーを上に公開するのは簡単だ。各 `fledge introspect` のエントリを MCP ツール定義にマッピングし、内部でCLIを呼び出し、JSON出力を返す。コアはすでに明示的な機能マニフェストを持つクリーンな境界上の構造化JSONだ。MCPサーバーはその同じプリミティブの上のプロダクションAPIだ。よく構造化されたライブラリの前にREST APIが置かれるのと同じように。インターフェースが変わる。仕事は変わらない。

そしてCLIが本物のインターフェースなので、サブプロセスを呼び出せるものはすべて、プロトコルアダプターなしで使える。シェルスクリプトが使える。CIランナーが使える。ターミナルの人間が使える。MCPクライアントがサーバーレイヤーを通じて使える。それらの呼び出し元はどれも他の存在を必要としない。普遍性はプロトコルからではなく、プリミティブから来る。

MCPが追加するもの

これは MCP に反対する議論ではない。MCP は CLI の前に置いたとき、本当に追加するものがある。

ログと観測可能性。MCPサーバーはエージェントとツールの間に座るため、すべての呼び出しを記録し、時間を計測し、引数を検査し、異常にフラグを立てられる。直接呼び出されたCLIは、ログファイルにパイプしたものを提供する。MCPレイヤーは各コマンドを個別に計測することなく構造化された観測可能性を提供する。エージェントが何を求めたかを監査したいプロダクション用途では、それが重要だ。

プロトコルレベルでの発見可能性。MCPにはエージェントが「ここにどんなツールがあるか？」と尋ねて構造化された回答を得る標準化された方法がある。fledgeには同じことのために`fledge introspect`があるが、`fledge introspect`はfledgeがそこにあることを知っている必要がある。MCPレジストリはエージェントが下のツールについて何も知らなくても、標準的なハンドシェイクを通じてツールを発見できるようにする。

ツール間の合成。MCPサーバーは1つのエンドポイントを通じて複数の下位ツールを公開できるため、エージェントは個々のCLIのリストを知る代わりに接続する1つの場所を持てる。それはツールの数が多いときの運用上の利便性だ。

それらはプロダクションインフラの議論だ。エージェントをスケールで実行し、その動作を監査し、管理されたレイヤーを通じて幅広いツールインターフェースに接続するときに適用

される。MCPサーバーは、それが何であるかのラベルとして、ログ付きのAI向けAPIだ。持っていて便利なものだ。ただし、最初に作るものではない。

操作の順序

CLI が最初だ。存在した瞬間からすべての呼び出し元に機能する。MCP サーバーはログとプロトコル標準化がレイヤーの価値に見合うときに追加するラッパーであり、それより前ではない。

fledgeにとって、CLIは基盤であり、コマンドを実行する人間、リポジトリを動かすエージェント、ダウンストリームクライアントと話すMCPサーバーがすべてその上に乗る。エージェントの話は「エージェントはMCPを通じてfledgeを使う」ではない。「エージェントは他のすべてと同じようにfledgeを使う。CLIが最初からどんな呼び出し元にも一流として作られているから」だ。

fledgeのMCPサーバーが存在するとき、それは薄いレイヤーになる。仕事はすでにコアで完成している。それがプリミティブを最初に作ることの要点だ。

スキャフォールディングとテンプレート

私をfledgeへ向かわせた2つ目の要素、「同じでない」問題の後、ブートストラップだった。第1章で名前を挙げたコピーペーストの苦痛だ。どんな様子かを近くで見てみよう。新しいRust CLIを作ることにすると、最初の1時間はCLIではない。Cargoのレイアウト、コンフィグ、lintセットアップ、CI、READMEのスケルトン、以前に20回打ち込んだことのあるものすべてだ。それが税金であり、大量のプロジェクトを立ち上げているので常にそれを払っていた。

だからスキャフォールディングは脇役ではなく柱だ。何もないところからプロジェクトを立ち上げることは、buildやtestと同じくらいライフサイクルの一部だ。全体のアイデアはこうだ。手でコピーペーストするのではなく、テンプレートから新しいプロジェクトを立ち上げる。

fledgeではそれが`fledge templates`の下にある。`fledge templates init`を名前とテンプレートで実行すると、プロジェクトを立ち上げてくれる。

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

私が実際に始めるプロジェクトの種類をカバーする組み込みセットがある。出荷されるのは`rust-cli`、`ts-bun`、`python-cli`、`go-cli`、`ts-node`、`static-site`、`kotlin-kmp`、`kotlin-ktor-api`だ。そのリストは基本的に私が作業する言語のマップだ。Rust CLI、いくつかのTypeScriptの種類、Python CLI、Go CLI、静的サイト、そしてKotlin MultiplatformとKtor APIのもの。それら8つが今日リポジトリにある完全な組み込みセットだ。fledgeの検出側はプロジェクトが存在すればSwift、Rust、Nodeなどを処理する方法を知っている。テンプレート側はプロジェクトが最初から存在できるようにする方法だ。

最も気にするのは、テンプレートが私が一度に承認しなければならない閉じたリストではないことだ。GitHubの任意のリポジトリからスキャフォールディングできる。組み込みだけでなく。`--template user/repo`で、そこからテンプレートを引き込む。

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

コアは小さくて便利なセットを出荷し、それ以降は私が介在する必要なく自分自身で拡張できる。テンプレートは誰かがすでに考え出したプロジェクトの形であり、GitHubのリポジトリに存在すれば、fledgeはそこから新しいプロジェクトを立ち上げられる。だからテンプレートのセットは「CorvidLabsが公開したもの」ではない。「誰かがリポジトリに置いたすべての出発点」だ。

そしてinitだけでなく、テンプレートについての動詞が全セットある。テンプレートを作るためのfledge templates create、テンプレートを見つけるためのfledge templates listとfledge templates search、テンプレートが実際に正しい形式かどうかを確認するためのfledge templates validateがある。テンプレートからスキャフォールディングするなら、特にエージェントがそうするなら、同じ壊れたものが百のプロジェクトに焼き込まれる前にテンプレートが成立していることを確認したい。

それは「解析できるか」以上のことを確認する。テンプレートのtemplate.tomlマニフェストを読み、名前と説明が空でないことを確認し、テンプレートが必要とすると言っているツールが実際にPATH上にあることを確認する。そしてテンプレートのすべてのファイルとすべてのファイル名を走査し、それらに対してプレースホルダーのテンプレート処理を実行するので、構文エラーやマニフェストで定義されていない変数のような悪いプレースホルダーが、スキャフォールディングを行う前にエラーとして捕捉される。ファイルがないテンプレートにもフラグを立てる。マニフェストが出力から除外されるよう設定されていない場合は警告する。だから構造的だが「ファイルがある」を超えている。initが行う方法でテンプレート処理を実際に実行し、どこで壊れるかを教えてくれる。

それが本当の理由で、スキャフォールディングが脇にある別個のジェネレーターツールではなくこのCLIに属する。fledgeにある他のすべてと同じ理由だ。同じインターフェース、同じ2種類のユーザーのために。人間はfledge templates initを実行して退屈な1時間をスキップする。エージェントはまったく同じコマンドを非インタラクティブで実行し、ウィザードをクリックする人間なしに最初のステップから新しいプロジェクトを立ち上げる。第1章から、検出、イントロスペクト、おそらくスキャフォールディング、そして実行、スキャフォールディングのステップは「おそらく」だ。リポジトリがすでに存在するとき、fledgeはそれを検出する。まだ存在しないとき、最初の本当の動きはテンプレートから立ち上げることであり、そうすれば他のすべて、buildの動詞、testの動詞、reviewの動詞がすぐに機能する。なぜなら、fledgeが期待する方法ですでに配線された状態でテンプレートから出てきたからだ。

生まれた最初からfledgeを話すプロジェクトが、本当に求めていたものだ。「コピーペーストを省いてくれる」だけでなく、それもするが。スキャフォールディングされたプロジェクトは、build、test、最初のコミットから他のすべてのリポジトリと同じ動詞への準備ができている。手で貼り付けていたボイラープレートは、半分の時間、プロジェクトを他のプロジェクトと一貫させる配線だった。スキャフォールディングをライフサイクルCLIに組み込むことで、その一貫性がプロジェクトが生まれるデフォルトになり、後からボルト留めするものではなくなる。

だから、新しいプロジェクトはスキャフォールディングから始まる。そしてfledge templates initを名前で呼ばなくても、プロジェクトは最初のコミットから同じセットアップが配線される。spec-sync、fledge、augur、attest。本当の「init」はテンプレートでは

なく、スタックが入ることだ。コマンドはそこへの速道だ。どちらの道でも、私の新しいプロジェクトは他のすべてのプロジェクトが持つツールをすでに持って生まれる。

ループの中のAIレビュー

3つ目の柱はタスクランナーの中に見つけて驚く人が多いものだ。タスク実行、スキャフォールディング、確かにそれらは明らかにライフサイクルのものだ。でもAIコードレビュー？buildやtestと同じCLIの中に？それはどこか別の場所、専用のツール、CI、プルリクエストのボットに属するように感じられる。

でもそうではない。理由はシンプルだ。レビューはビルドやテストと同じようにループの一部だ。何かを書き、確認し、修正し、また書く。レビューは確認のステップだ。そしてエージェントがコードを書いている場合、私の場合はほとんどそうだが、そのコードを採点することはただの別の動詞だ。`build`と`test`の隣に座る。なぜならそれらが行うのと同じ仕事をするから。前に進む前にそのものが実際に良いかどうかを教えてくれる。

1つのインターフェースは3つのツールを上回り、私よりエージェントにとってさらに上回る。それがレビューを別のツールとして出荷する代わりにライフサイクルCLIに組み込む全理由だ。別のレビューツールのエージェントは、継続的な作業のループを行うために、独自の呼び出しと独自の出力形状を持つまったく別のことを学ばなければならない。エージェントがすでにfledgeをbuildとtestのために動かす方法を知っているなら、`fledge review`はすでにその形を知っている動詞だ。同じインターフェース、同じJSON、同じヘッドレスモード。ステップが「コンパイルするか」から「良いか」に変わっただけで学ぶべき新しいツールはない。

fledgeではこれが`fledge review`であり、それが行うことはデフォルトブランチに対するAIコードレビューだ。だから全世界をレビューするのではない。変更されたもの、マージ先のブランチに対するdiffを見ている。それがまさにレビューが実際に行われる単位だ。人間のレビュアーまたはPRボットが見るのと同じスコープで、すでに使っている同じCLIの動詞として実行される。

そして1つのモデルやプロバイダーに縛られていない。内部では、レビューは私の他のツールが使う同じマルチプロバイダークライアント、corvid-aiを通じて行われる。だからfledgeはcorvid-aiがサポートするどのプロバイダーとも話せる。AnthropicのAPI、OpenAI互換エンドポイント、Ollamaのようなローカルランナーなど。サポートされるプロバイダーのリストはcorvid-aiリポジトリにあり、エコシステムとともに変化する。要点はレビューが望むどのモデルに対しても実行できること、ツールではなく使う人の選択で。

それが何を意味するかについて率直に言っておくのが価値がある。なぜなら本書の残りの部分はブラスト半径について率直だからだ。`fledge review`はdiffとスペックを受け取り、向けたどのプロバイダーにも送る。それがホストされたエンドポイントなら、マージされていないコードが外部に出た。プライベートリポジトリにとってそれは本物のデータ流出インター

フェースであり、見て見ぬふりをする詳細ではなく、目を開けて行う選択だ。ローカルモデルに向けることが、何も外部に出ないバージョンだ。そのローカルケースについて1つ正直に言うと、OllamaパスはキーとCloud URLを喜んで受け取るが、corvid-aiのREADMEはまだOllamaをローカルのキーなしオプションとしてフレーム化しているので、ドキュメントはOllamaがデスクの下のボックスを意味するように読める。Cloudパスは今日機能している。READMEがまだそこに追いついていないだけだ。私の側のドキュメントのギャップであり、欠けている機能ではない。

私が本当に気に入っているマルチモデルの角度もある。`--with-model`で同じdiffに対して複数のモデルから並行批評のパネルを実行できる。

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

これはギミックではない。これらのモデルに時間を費やしたことがある人なら、それらが同じものを捕まえず、同じことを幻覚しないことを知っている。同じdiffに対して2、3個実行し、どこで一致するか、そして1つが他が見逃したものにフラグを立てるか確認することは、どれか1つを信頼するより良いシグナルだ。同じ変更に対して並行して実行される2番目と3番目の意見だ。

そしてfledgeの他のすべてと同様、出力は構造化されている。すべてのコマンドが`{schema_version: 1, ...}`を出力する。デフォルトでJSON。だからレビューは人間が読み返して解釈しなければならない散文の壁ではない。データだ。コードを書いたエージェントはレビューを実行し、構造化された所見を受け取り、同じループで行動できる。「レビュアーがエラーハンドリングについて不満そう」を実際に何をすべきかに変換する人間が途中にいなくていい。人間はレビューを読め、エージェントはそれを解析できる。同じコマンドで。

`fledge review`はspec対応でもある。spec-syncの章が詳しくカバーするが、ここではスペックがあったときにレビューが何をするかを見るだけでいい。レビューはdiff内のファイルをカバーするスペックを特定し、スペックの宣言されたファイルとスペックの`specs/<name>/`ディレクトリでマッチングし、それらのスペックをバックグラウンドとしてプロンプトに組み込む。そして意図的にそれらに向けられる。モデルはスペックがモジュールがすべきことを説明していること、変更を解釈するためにそれらを使うこと、スペック自体ではなくdiffだけをレビューすること、そして重要な部分として、もしdiffがスペックの不変条件に矛盾するなら、それをdiff内のバグとして指摘するよう伝えられる。だからスペックからの逸脱は批評のバックグラウンドフレーバーではない。レビューが明示的に表面化するよう指示される所見だ。

そしてその価値を証明している。`fledge review`は私にとって本物のバグを捕まえた。ビルドが通り、テストに合格したdiffの中に普通に存在していたが、マージ前にLLMがフラグを

立てたものだ。それがレビューを動詞として持つことに価値があるかどうかのテストだ。そして合格した。スタイルの指摘ではなく、ループの残りがスルーした実際の欠陥だ。spec対応の側も報われた。スペックにレビュアーを向けることで、モジュールが守るべき契約から静かに逸脱したコードを捕まえた。バグと同様にコンパイルし、同様にパスする逸脱だ。エージェントはそれをうまく活用する。なぜならそれらにとってbuildとtestと同じ形だからだ。実行し、構造化された所見を読み、見つかったものを修正し、また実行。

エージェントがコードを書き、buildを実行し、testを実行するなら、コードの採点はそれらが知っている届く範囲のもう一つの動詞であり、他が通したものを捕まえる。

プラグイン、そしてそれを信頼しないこと

fledgeには少しの機能のみを持つ小さなコアがあり、本当の機能はすべてプラグインにある。これは意図的だ。コアがすべての言語、すべてのワークフロー、すべてのチームの奇妙なステップを知らなければならないなら、腐敗する。だからそうしない。コアは一般的な動詞と検出方法を知っている。実際のリーチはその周りに投入されるプラグインから来る。誰でもコアに触れることなく追加できる。

機能をプラグインに押し出すもう一つの理由は、fledgeができることのリストを所有したくなかったからだ。人々は好きなように拡張できる。Rust、Swift、TS、シェル。好きな言語でプラグインを書く。私のツールを拡張するために私の言語を強いられない。

プラグインがコアと話す方法は本物のバージョン管理された契約だ。`plugin.toml`マニフェストを持つバイナリで、JSONでfledgeと話す。ネイティブでもサンドボックス化されたWASMモジュールでも同じだ。プロトコルの章にワイヤーフォーマットと機能ハンドシェイクがある。ここではその継ぎ目が規約の山ではなく契約であることを知れば十分だ。

そして出てくる質問を先取りしておく。プラグインとレーンは同じものではない。プラグインは機能、新しい動詞を追加する。レーンはすでに持っている動詞を順序付けられたパイプラインに連鎖させる。だから「すべてはプラグインだ」ではない。3つの柱が組み込まれた本物のコアがあり、プラグインがその周りの機能を拡張し、レーンがそれらの機能を順序にまとめる。レーンについては後の章で詳しく述べる。

どんな言語でも、つまり信頼できない

「好きなように、どんな言語でも拡張できる」の裏返しは、自分が書いておらず保証できないコードを実行することになるということだ。プラグインは単に誰か他の人のプログラムだ。誰でも何でもでそれを書けると決めたら、大量の信頼されないコードを実行することになると決めたことにもなる。

だからfledgeはWASMでプラグインをサンドボックス化する。Wasmtimeで。要点は安全性だ。許可しない限り、プラグインはディスク全体を読んだり外部に通知したりできない。デフォルトでは箱の中にいる。付与したものだけを得て、それ以外は何もない。

サンドボックスのもう一つの理由があり、それが本当の理由だ。エージェントがこれらのプラグインを実行する。エージェントがプラグインをインストールして実行するなら、デフォルトでは信頼できない。プラグインを信頼するのではなく、それを実行する決定を盲目的に信頼するのでもない。エージェントが手を伸ばしてプラグインを取得して実行するのは、まさに「おそらく大丈夫」では不十分な状況だ。サンドボックスはエージェントが実行するツ

ールを安全にする。それがエージェントの本の信頼とブラスト半径の話への橋だ。ここではツール側のバージョンはシンプルだ。信頼されないコードとそれを実行する自動化されたものは、その周りにボックスが必要になる。WASM/Wasmtimeがそのボックスだ。

明白な代替案よりWASMを選ぶ理由は明示する価値がある。最初に手を伸ばすであろう2つはseccompプロファイル（Linuxシスコールフィルタリング）とコンテナ（Dockerや同様のもの）だ。どちらも機能するが、どちらもここにはフィットしない摩擦または前提を追加する。seccompは設定するためにOSレベルの特権を必要とし、Linux専用なので、即座にクロスプラットフォームの約束を破る。コンテナはDockerデーモン、個別のイメージプル、「プラグインを実行する」より重いプロセス境界を意味する。WASMは異なるフィットをする。Wasmtimeはライブラリとしてfledgeプロセスに直接組み込まれ、単一バイナリの中に出荷される。追加のデーモンや特権なしに任意のOSで実行される。そして設定が必要なカーネルポリシーではなく、リンク時に構造的に機能境界を強制する。seccompやコンテナが間違っているのではない。1つのバイナリとしてどこでも実行するツールには、それと一緒に移動できるサンドボックスが必要であり、Wasmtimeはそれを行う。

カナリア

証明できないサンドボックスは単なる希望だ。だからカナリアがある。サンドボックス化されたプラグインができてはいけないことを試みて、できないことを確認することが仕事のプラグインだ。それがサンドボックスが機能することの証明だ。もしカナリアが脱出できなければ、境界は本物だ。もし脱出できたなら、すぐにわかる。

実は2つのプラグインだ。`fledge-plugin-canary`はネイティブのものだ。サンドボックスなしで実行され、攻撃が*機能*することを証明する。SSHキーとAWS credを読み、`GITHUB_TOKEN`のような環境変数を取得し、ネットワーク接続を開き、プロセスを生成し、`.git/hooks`に書き込む。そして`fledge-plugin-canary-wasm`がWasmtimeサンドボックス内で同じバッテリーを実行し、そこではすべてがBLOCKEDとして返ってくるべきだ。その独自の説明は率直に言っている。「Wasmtimeサンドボックスがネイティブカナリアが露わにするすべての攻撃をブロックすることを証明する。」

2つを並べると、私の言葉を信じるより要点がより鋭くなる。ネイティブカナリアの独自出力は、同じコードを実行するWASMプラグインが見るものと自分自身を対比させる。

```
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)
```

同じ攻撃、2つの境界。ネイティブはSSHキーを読む。WASMはできない。~に届くためのプリオープンされたディレクトリがないから。ネイティブは`GITHUB_TOKEN`を継承する。WASMはできない。ゲストの環境が空だから。WASM側でBLOCKEDの代わりにLEAKED

として返ってくる結果はサンドボックスが脱出されたことを意味する。2つを合わせるとエンドツーエンドのチェックだ。1つは危険が本物であることを示し、もう1つはボックスが機能することを示す。

サンドボックスが守らないもの

サンドボックスはブラスト半径の封じ込めだ。プラグインのコードが届ける範囲を制限する。すべてを制限するわけではない。それが解決しない問題の治療薬として売るのはこの章の不誠実なバージョンだ。だからWASMが買わないものを示す。

スタックの上位でのデータ流出は止めない。WASMボックスはプラグインがSSHキーに届くことを防ぐが、意図的にツールに渡すデータを管理しない。`fledge review`はdiffとスペックを向けたどのLLMプロバイダーにも送る。そしてそれがホストされたエンドポイントなら、マージされていないコードが外部に出た。そのデータは正面玄関から出るため、WASM境界はそれに触れない。プラグインを通じてではない。それは同意とポリシーの問題であり、サンドボックスが解決するものとしてではなく、レビューの章でその通りに扱う。

ホストユーザーとして実行するプラグインからは守らない。すべてのプラグインがWASMではない。ネイティブプラグインはあなたの許可、あなたの環境、あなたのディスクで実行される。それがネイティブカナリアがSSHキーを読める全理由だ。サンドボックス化されていないから。ネイティブプラグインを実行するとき、またはゲストからプロセスをフォークするものを実行するとき、マシン上で実行するものを信頼するのと同じ方法でそれを信頼することになる。サンドボックスはWASMパスの保証であり、すべてのプラグインに対する包括的なものではない。

オリジンや意図を検証しない。WASMはコードができることを封じ込める。コードが、またはプラグインが実行するコードを書いたエージェントが、それをすべきかどうかについては何も言わない。エージェントが書いたロジックを実行するプラグインは、ボックスの中で私がレビューしていないロジックを実行している。ボックスの中の信頼されないコードはまだ信頼されないコードだ。ボックスはただ被害が広がるのを防ぐだけだ。

だから正直なフレーミングは狭い。WASM/Wasmtimeはコード実行パスのオリジン非依存ブラスト半径封じ込めだ。データフロー制御ではなく、同意管理でもなく、プラグインに渡すものやそれを書いた人について考えるのをやめる理由でもない。

プラグインエコシステムはfledgeを3つのネイティブ組み込みでしか知らなかったら予想するより大きい。CorvidLabsの組織に複数の言語にわたって多くの`fledge-plugin-*`リポジトリがある（エコシステムの章がカバーする）。仕事に合った何でもの言語で書かれている。サンドボックスがそれを可能にするものだ。プラグインは何でもの言語で書けて、それでも安全に実行できる。エコシステムと言語の全ツアーはその独自の章で行う。

fledgeが1日にどうフィットするか

多くのリポジトリにわたって速く信頼されないプラグインを実行するエージェントが、サンドボックスが交渉の余地のない理由だ。それらの判断を下しているのは、通常ケースバイケースで決める私ではない。常に動き続けるエージェントが、多くのリポジトリにわたって。その下のWASMサンドボックスがそれを安全に走らせられるものだ。

プラグインプロトコル

これが継ぎ目、プラグインとコアが実際に会う場所だ。リポジトリに名前がある。プロトコルはバージョン管理されており、バージョン文字列は`fledge-v1`だ。プラグインはそれを宣言し、それがハンドシェイクだ。他のすべてはそこからぶら下がる。

プラグイン作者が実際に書くもの

プラグインはルートに`plugin.toml`というマニフェストを持つgitリポジトリで、1つ以上の実行ファイルを持つ。マニフェストは短い。プラグインに名前を付け、バージョンを付け、話すプロトコルを言い、追加するコマンドをリストする。

```
[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
```

それが作者側の契約全体だ。各`[[commands]]`エントリはfledgeが知るべき新しい動詞で、実行するバイナリに向けられている。`[capabilities]`ブロックは作者が前もって言う、このプラグインが何をできる必要があるか。シェルコマンドを実行する、少し状態を保持する、プロジェクトメタデータを読む。デフォルトは`false`だ。必要なものを要求し、それ以上は要求しない。その要求はコードに埋まっているのではなく、マニフェストに見える。

ホストがプラグインと話す方法

プラグインコマンドを呼び出すとき、fledgeはバイナリを生成し、JSONライン（1行1JSONオブジェクト）としてstdinとstdoutを通じて話す。ホストが最初に送るのは`init`メッセージで、そのメッセージがゼロコンフィグのアイデアを文字通りにしたものだ。プラグインに全状況を渡す。プロジェクト名とルート、検出された言語、git状態（ブランチ、ダーティかクリーンか、リモート）、プラグイン自身のバージョンとディレクトリ、fledgeバー

ジョン、そして付与された正確な機能。プラグインはどこにいるか、何を見ているかを発見しに行く必要がない。ホストはすでに知っている。それがコアの全仕事なので、ただプラグインに教える。

その後は会話だ。プラグインはメッセージを送り返す。`prompt`、`confirm`、`select`はユーザーに何かを尋ねるため。`log`、`progress`、`output`は報告するため。`exec`はコマンドを実行するため。`store`と`load`は小さな状態のパッチのため。`id`を持つものは、ちょうど1つの答えを返す。`response`または`cancel`だ。stderrは決してキャプチャされないので、プラグイン作者はいつでもターミナルに直接デバッグを出力できる。

誰かの肩越しに読めるくらいシンプルだ。ホストは`init`で始まり、状況を渡す。

```
{"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}}
```

プラグインはすでにどこにいるかを知っているので、fledgeにコマンドを実行するよう求める。

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

そしてホストはマッチングする`id`に答える。

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

それが全体の形だ。`init`はエージェントがREADMEの散文で推測しなければならないものすべてを構造化JSONとして渡す。そして`id`を持つすべてのリクエストがちょうど1つのマッチングレスポンスを得る。ツールはプラグインにどこにいるかを教える。プラグインは探し回らない。

サンドボックスがボルト留めされる場所

上記のすべてはネイティブプラグイン、JSONをパイプ越しに話す通常の実行ファイルを説明している。問題、そしてリポジトリはこれについて率直だが、ネイティブプラグインはあなたとして、完全なアクセスで実行される。機能ブロックはプロトコルをゲートするが、プロセスをゲートしない。何も要求しなかったプラグインはまだあなたのSSHキーを読める。なぜならあなたのユーザーとして実行しているプログラムだから。それがカナリアが証明した漏れだ。プロトコルレベルで機能を宣言することはセキュリティシアターだった。それが

WASM移行の全理由だ。そのアークをサンドボックスの章で話した。ここでの要点はそれがプロトコルで何を変えたかだ。

WASMプラグインは全く同じ`fledge-v1`プロトコルを保持するが、境界を変える。プラグインは`runtime = "wasm"`を宣言し、単一の`.wasm`バイナリを出荷する。stdinとstdoutの代わりに、同じJSONメッセージがホストが組み込む3つのホスト関数（`recv`、`send`、`exit`）を通じて交わされる。同じメッセージタイプ、同じ会話。そして機能は礼儀正しいリクエストではなくなる。リンク時に強制される。`exec`を付与しなかったなら、`exec`インポートは単純にリンクされず、それを呼び出そうとするプラグインはそもそもインスタンス化に失敗する。だます対象のランタイムチェックはない。なぜなら呼び出す関数がそこに存在しないから。ファイルシステムアクセスは`none`、`project`、`plugin`で、プリオープンされたディレクトリとしてマウントされる。ネットワークはブール値だ。それに加えてランタイムはフューエル制限されメモリキャップされているため、プラグインは永遠にスピンしたりマシンを食べたりできない。

だから`plugin.toml`に見える機能と、コードが実際に届ける機能は、構造的に同じものだ。

それが構造的でなければならなかった理由はカナリアが教えた教訓だ。単に*宣言する*機能はプラグインがそれを尊重することを信頼している機能であり、プラグインをサンドボックス化する全理由はそれを信頼していないからだ。機能していない唯一のバージョンは、付与しなかった機能が到達不能なもの、関数が文字通りそこに存在しないもの。エージェントがこれらのものをインストールして実行しているとき、それがあなたが求めるものだ。

バージョンリスクについて言及する価値がある。プラグインが宣言するプロトコル文字列は`fledge-v1`だ。破壊的なプロトコル変更が来るとき、それはいつか来るが、その文字列は`fledge-v2`になる。`fledge-v2`を期待するホストに対してまだ`fledge-v1`を宣言しているプラグインはインスタンス化で、コードが実行される前に、大きな声で失敗する。静かに誤動作しない。マニフェストのバージョンがその早期失敗を強制するもので、ミスマッチしたプラグインは何かできるポイントに達しないため、オペレーターは微妙なランタイムバグを追うのではなく、何を更新する必要があるかを正確に知る。

レーンライフサイクルフック

同じプロトコルが2種類目の統合を担う。ライフサイクルフックだ。プラグインは直接のコマンド呼び出しではなく`lane:pre`または`lane:post`イベントで発火する`plugin.toml`のフックを登録できる。デプロイプラグインはドキュメントと一緒に出荷される例だ。任意のfledgeレーンが完了した後に自動的に実行される`lane:post`フックを登録する。

フックが発火すると、fledgeは環境変数を通じてフックバイナリにコンテキストを渡す。`FLEDGE_LANE_NAME`、`FLEDGE_LANE_STATUS`、`FLEDGE_LANE_RUN_ID`だ。レーン定義はどのプラグインがインストールされているかを知らない。プラグインはどのレーンが存在するかを知らない。ライフサイクルイベントが継ぎ目だ。

全体像はどう見えるか。レーンは`fledge.toml`に住む。フック登録はプラグインの`plugin.toml`に住む。2つはどちらも相手の詳細を知らずにつながる。

```
# 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
```

`fledge lanes run verify`が完了すると、fledgeは`lane:post`イベントを発火する。マッチングする`[[hooks]]`エントリを持つインストールされたプラグインは、`FLEDGE_LANE_NAME=verify`、`FLEDGE_LANE_STATUS=success`（または`failure`）、`FLEDGE_LANE_RUN_ID`が設定されてフックバイナリが呼び出される。

レーンはstdoutの散文ではなく環境変数として構造化されたコンテキストを出力する。プラグインはテキストをスクレイピングして推測する代わりに行動できるステータスを読む。それは`init`メッセージと同じ設計原則だ。ホストは何が起きたかを教えてくれる。あなたが推測する必要はない。

3つの柱はこの上に乗っている

3つの柱（タスク実行、スキャフォールディング、AIレビュー）はコアであり、コミュニティプラグインではない。プロトコルはそれらの周りの拡張層だ。背骨はfledgeのものであり、`fledge-v1`は誰でも機能をボルト留めできる方法だ。コアは本物であり、プラグインはその周りを囲む。プロトコルはただ2つが会う継ぎ目だ。

プラグインエコシステム

CorvidLabsの組織には複数の言語にわたって多くの`fledge-plugin-*`リポジトリがあり、いくつかはアーカイブされている。ライブ数は本物だが、エコシステムについて最も興味深いことではない。プラグイン数は在庫だ。議論は実際に手渡すものと、それぞれについての確信の程度だ。

だからここに毎日使うであろう5つを示す。`augur`、`attest`、`format`、`gitleaks`、`deps`。そこから始めろ。この章の残りは他の37個がなぜ存在するか、そしてどれだけ信頼するかについてだ。

手を伸ばすであろう5つ

これらは基本的にすべての変更で、私のものもエージェントのものも実行される。2つの仕事にまとまる。

信頼ペアは`augur`と`attest`だ。`augur`はdiffの変更リスクをスコアリングする。進める、レビュー、またはブロック。構造的なシグナルだけから。APIキーもLLMもなし。その説明は対象を声に出している。「人間とエージェントのため。」`attest`はどのコミットを誰がレビューしたかの署名付き証明をgit notesに保持する。その2つが私が最も依存するものだ。エージェントが作者のときに重要なものだから。入力時のリスクスコアリング、出力時の証明トレイル。

開発ループの3つは意図的に退屈だ。`format`、`gitleaks`、`deps`。フォーマッタを実行する。コミットされたシークレットをスキャンする。Rust、Node、Pythonにわたって、古い、監査、ライセンスの依存関係の健全性をチェックする。どれも興奮するものではない。すべてが常に実行される。なぜならそれがまさに手動で覚える代わりに自動的に発火させたいチェックの種類だから。

それら5つだけをインストールしても、エコシステムから得るものの大半が揃う。

3つの層と確信の程度

ロースターを読む誠実な方法は数ではなく確信による。プラグインは3つの層に分かれ、非常に異なる方法で信頼している。CorvidLabsの組織の現在のリストのライブロースターを確認してほしい。以下は見つかったものをどう読むかだ。

コア。 `augur`、`attest`、`format`、`deps`、`gitleaks`。毎日の5つ。私が書き、すべての変更で実行し、保守・テストされている。これがワークフローを賭けられる層だ。

インテグレーション。 `github`、`discord`、`algochat`、`memory`。これらはfledgeを外部の何かに結線する。`gh`を通じたGitHub API、CI失敗通知のDiscordウェブフック、暗号化されたオンチェーンメッセージング、メモリストア。保守・テストもされているが、話しかけるもののリスクを担う。インテグレーションはその背後にあるサービスと同じくらいしか信頼できない。

プレイグラウンド。 `weather`、`roast`、`hangman`、その他の一発ネタ。`weather`はターミナル予報を出力する。`hangman`はコードベースから引き出した識別子でハングマンをする。`roast`はコミットをLLMに通して「娯楽目的のみ」で罵倒する。これらは同じ基準で保守されていない。サンドボックスが機能することを証明するために存在する。気まぐれで書かれた半本気の気象プラグインが安全に実行できるなら、プロトコルが仕事をしている。低い基準は弱みではない。証明だ。

コアとインテグレーションは保守・テストされている。プレイグラウンドはサンドボックスが機能することを証明する。フラットな数を等しくベットされたツールの均等な広がりとして読まないこと。そうではない。そう見せかけることはこの章の不誠実なバージョンだ。

証明書、率直に

私がこれらのほとんどを書いた。毎日の5つは私のものだ。一発ネタのほぼすべてもそうだ。プロトコルが存在するとプラグインの公開にコストがかからないので、気まぐれで書かれた。何かが必要だった。プラグインを書いた。投入した。プロトコルの要点はコアを成長させずに機能を追加できることだ。

持っていない外部作者のコミュニティを主張するつもりはない。いくつかのプラグインはサークルから来たが、ロースターをコミュニティが検証した広さとして提示しない。主に1人の人間が引き出しを埋めており、それが正直な証明書だ。

カナリア、そしてこれが安全な理由

エコシステム全体に対して荷重のかかる一対について独自の言及がある。`fledge-plugin-canary`と`fledge-plugin-canary-wasm`、サンドボックスの章からのレッドチームペアだ。ネイティブのものは攻撃が機能することを証明する。WASMのものはサンドボックスがそれらをブロックすることを証明する。ここにある他のすべてのもの、Kotlinで書かれたプラグイン、半分眠りながら書いたプラグインは、そのペアが境界を保持しているから安全に実行できる。エコシステムはその周りのボックスが機能しているから正確にカオスになれる。

なぜこんなに多くの言語があるか

ほとんどはRustとシェル、次にSwift、TypeScript、Kotlin、Python、いくつかのHTML/JavaScriptのものだ。その広がりはプロトコルが意図通りに機能していることだ。

`fledge-v1`とWASMサンドボックスの全理由は、プラグインが何でもの言語で書けて安全に実行できることで、エコシステムが言語ではなく契約だけで合意する必要があるからだ。`fledge-plugin-bridge`はKotlinだ。`fledge-plugin-memory`はTypeScriptだ。`fledge-plugin-attest`はSwiftだ。インテグレーションの半分はシェルだ。誰もfledgeに追加するためにRustを学ばなくてよかった。前にあるものでそれを書き、コアはその間ずっと同じサイズのままだった。

fledgeができることのリストを私は所有しない。プロトコルが所有し、誰でも私に尋ねずにそれに追加できる。広がりが証明であり、層がそれについての誠実さだ。

---

spec-syncと誠実さを保つこと

スペックドリフトは私が他のほとんどより気にする失敗モードだ。スペックを書く。モジュールの契約、それが何をするか、パブリックAPIは何か。そしてコードが逸れる。誰かが関数を変える。スペックはまだ古い形を主張している。今ドキュメントとコードは静かに不一致だ。人間のチームではそれは古いREADMEだ。エージェントがループにいると悪化する。エージェントはスペックを真実として読み、コードがもはや守らない契約の上に作り上げる。誰も何かが壊れるまで気づかない。spec-syncはその逸脱を無視不可能にするために存在する。

spec-syncはそれ自体のツール、独自のRustバイナリ、独自のGitHub Actionであり、そのキャッチフレーズはそれが何かを言っている。「双方向スペックからコードへの検証、クロスプロジェクト参照、依存関係グラフ、AI駆動の生成。」重要な2つの言葉は*双方向*と*検証*だ。スペックとコードが一致することを両方向で確認する。テストスイートではなく、散文に対するファジーdiffでもない。構造的な契約チェックだ。文書化されたパブリックAPIが実際のものと一致するか。2つの方向は対称ではない。コードが持つがスペックが文書化していないエクスポートは警告、埋めるべきもの。スペックが主張するがコードが持たないエントリはエラー、破られた約束だ。

実際に何をチェックするか

スペックはマークダウンファイル（`*.spec.md`）で、YAMLフロントマターと必要なセクションのセットを持つ。フロントマターはモジュール、バージョン、ステータス、カバーするソースファイルを名前付けする。必要な`##`セクションはPurpose、Public API、Invariants、Behavioral Examples、Error Cases、Dependencies、Change Logだ。必要なセクションを欠くとスペックは不完全で検証がブロックされる。

それがルールだ。それを守っているスペックを見てみよう。これがfledge自身の`review`モジュールスペックのヘッド、本物のフロントマター、そしてspec-syncがコードに対してチェックする実際のエクスポートで埋められた`## Public API`セクションだ。

```
---
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 |
```

`files:`行はspec-syncがdiffと`src/review.rs`の実際のエクスポートに対して交差させるものだ。その`## Public API`テーブルのすべての行は、spec-syncがコードで見つけることを期待する名前だ。コードが持たないエントリが下のエラーケースだ。スペックを書いたことがなければ、そのフロントマターと1つの誠実な`## Public API`テーブルがコピーすべき形だ。必要な残りのセクションはモジュールが約束するものを説明する同じ種類のもの、散文とテーブルだ。

そして両方向で検証する。

- **コード → スペック:** 未文書化のエクスポート、警告。
- **スペック → コード:** ファントムまたは古いエントリ、欠けているソースファイル、型の不一致、すべてエラー。

スキーマについても同様だ。宣言されたデータベーステーブルとカラムは実際のSQLに対してチェックされ、ファントムテーブルやカラムは失敗する。そしてプロジェクト間で参照を

解決するので、あるリポジトリのスペックが`owner/repo@module`で別のリポジトリのモジュールに依存でき、そのリンクも検証される。

これが*構造的だ*ということを覚えておくべきだ。散文を採点しているのでもなくテストを生成しているのでもない。1つの率直な質問をしている。文書化されたインターフェースが本物のインターフェースと一致するか。そしてそれに決定論的に答える。それがエージェントの前に置くのを安全にするものだ。議論すべき判断の余地がない。APIがスペックと一致するかどうかだ。

どう現れるか

3つの方法で現れ、インタビューでの「spec-syncは日々どう現れるか」への答えはすべてだった。同じスペックフォーマット上の3つの正面玄関、それぞれ異なる場所で実行される。

正面玄関	実行場所	何をするか
`fledge spec`（ネイティブ）	あなたのマシン、fledge内	`init`、`check`、`list`、`show`：fledge独自の検証、シェルアウトなし
`specsync`バイナリ	エージェントのループ、PRの前	`specsync check` / `--fix`：エージェントがループ内で実行するスタンドアロンツール
`CorvidLabs/spec-sync@v4`	CI、プルリクエスト上	ビルドをゲート、ドリフトをコメント、失敗でブロック

このセクションの残りはその3行を順に見ていく。

fledgeはスペックをネイティブに知っている。 specはfledgeの柱の1つだ。fledgeのREADMEはそれについて逐語的だ。「Spec | `spec` | [spec-sync]。モジュールは契約を宣言し、AIはそれをコンテキストとして使う。」コードでそれが何を意味するかについて正確にする価値がある。fledgeには独自の`fledge spec`（`init`、`check`、`list`、`show`）がバイナリに直接組み込まれている。`specsync`ツールにシェルアウトしない。同じ`.specsync/config.toml`と同じ`*.spec.md`ファイルを読み、それ自体でパースし、独自のチェックを行う。だからスペックフォーマットは共有されているが、fledge内の検証はfledge独自のコードであり、別のバイナリへの呼び出しではない。

そのネイティブなスペック認識がfledgeのAIコマンドが契約に依存できる理由でもある。`fledge ask`はスペック対応のQ&Aで、`fledge review`はスペック対応のコードレビューだ。両方が関連するスペックをコンテキストとして引き込む。スペックはそれらのコマンドがコードについて推論するときにモデルに与えるコンテキストだ。スペックは脇にあるドキュメントではない。ツールがコードについて推論するときに読むものだ。

CIをゲートする。 スタンドアロンのGitHub Action、`CorvidLabs/spec-sync@v4`があり、`specsync check`を自動的に実行する。最小限のワークフロー。

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

`strict`は警告をエラーに変える。`require-coverage`はフロアを設定する。`comment`はスペックドリフトの要約をプルリクエストに投稿する。そしてチェックは失敗で非ゼロで終了し、PRがブロックされることを意味する。それが全体を本物にするメカニズムだ。ドリフトは丁寧なメモを生成しない、ビルドを失敗させる。エージェントも人間も静かにスペックとコードを分離させることができない。なぜなら分離が失敗するチェックだからだ。

エージェントをスペック上に保つ。 これが私が最も気にするもので、スペックがCIだけでなくエージェントのループの内部に住む理由だ。メカニクスは具体的だ。エージェントはbuildとtestを実行するのと同じようにループの一部として`specsync check`を実行する。チェックは構造化された失敗を返し、エージェントはそれを読んでどうするかを決め、どちらの方向に進むかはドリフトがどの方向に走るかによる。

構造化出力が両方向にドリフトしているとどう見えるか（本物の出力形式に合わせた例示）。

```
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
```

それがエージェントが読むものだ。エラーはファイル、スペックの主張、コードが実際に持つものを名前付けする。警告はエクスポートと正確な修正コマンドを名前付けする。推測は不要だ。コードをスペックに一致させるか、スペックを修正するか、そしてexit 0になるまで再実行する。

コードがスペックが言及しないエクスポートを持っているなら、警告の方向、エージェントはスペックを手動で編集しない。`specsync check --fix`を実行し、未文書化のエクスポート

をスタブとしてスペックに自動追加し、簡単なケースは1つの動作で解決される。エージェントの仕事はほとんど、最初にドリフトを発見するのではなく、スタブに本物の説明を埋めることだ。

もう1方向が実際の作業を要する。スペックがコードが守らないAPIを主張するとき、エラーの方向、古いまたはファントムのエントリ、`--fix`がない。なぜなら修正は判断の余地があるから。コードが間違っており、エージェントがコードを約束されたものに一致させるか、スペックが期待過剰であり、エージェントが契約を修正するか。どちらにしてもエージェントはコードでそれを意図的に解決し、緑になるまでチェックを再実行しなければならない。それが実際のループだ。契約、変更、チェック、そしてどちらの側が外れたかによって自動追加か本物の修正のどちらか。それはdiffがプルリクエストに届く前に、CIの後からではなく、Merlinのループ内で実行される。

spec-syncが確認しないこと

正直に言いたい一線がある。それがこのツールの端だからだ。spec-syncはスペックとコードが一致することを確認する。スペックが優れているかどうかについては意見を持たない。

それが何を開けたままにするか考えてほしい。スペックは正しいエクスポートを名指しして、それらを間違って説明することがある。薄いこともある。何も言わないPurposeセクション、正確だがモジュールが何のためにあるかを何も教えないPublic APIテーブル。願望的なこともある。誰かが書くつもりだったコードのために書かれたもの。そしてエージェントが間違っていたタスクブリーフ、あるいはインジェクトされたタスクブリーフからスペックを草案したなら、スペックは間違ったものに対する忠実なコントラクトになりうる。それらのケースのどれでもspec-syncはパスする。表面が表面と一致している。チェックは緑だ。コントラクトは間違っていて、ループの中の何もそれを捕まえなかった。コードに対してスペックをチェックすることは、スペックがそもそも悪かったとは教えられないからだ。

これは本物のギャップであり、一つとして名指ししている。コードにはチェックがある。スペックにはない。欲しいのは、エージェントがそれに対してビルドする前にスペック自体に対して実行される二つ目のゲートだ。必要なすべてのセクションが実際に何かを言っているか、Public APIは存在するファイルを指しているか、成功と失敗の明確な記述があるか、人間が実際にこのコントラクトを承認したか。それが`fledge spec lint`であり、まだ作られていない。それが作られるまで、スペックはこのパイプライン全体への入力の中で何も検証しない唯一のものであり、それは緑のチェックを信頼するときに知っておく価値がある。

共有された契約が全体の要点である理由

これはこれらすべてのツールの下にあるテーゼに直結する。人間とエージェントが同じツールを使い、同等の一流市民として。スペックはそのアイデアの最も明確なバージョンだ。人間がそれを書くかレビューする。エージェントがそれに対してビルドする。spec-syncが両

方向で決定論的に両方を守る。リファクタリングをしてドキュメントを更新しなかった人間は、APIを幻覚したエージェントと同じ方法で捕まる。チェックはどちらが逸れたかを気にしない。

RustでfledgeをBuildする

fledgeは単一バイナリだ。一度インストールすれば、他に何もインストールせずに任意のマシン、任意のOSで実行できる。引き込むランタイムも、インタープリタも、すでに存在しなければならない依存関係マネージャーもない。デフォルトでクロスプラットフォーム。1つのビルドパイプラインがmacOS、Linux、Windowsで動作するバイナリを生成する。それがコアの要件であり、Rustはそれを戦いではなく簡単にする理由だ。章全体がそこから続く。なぜRustが正しい選択だったか、なぜそれを始めることが人々が期待する戦争物語ではなかったか、エージェントがどう流暢さのギャップを埋めたか。

fledgeを始めたとき私はSwiftを約10年書いていた。だからこの章の正直なバージョンは少し拍子抜けするかもしれない。Rustを始めることは痛くなかった。大きなものは何も戦わなかった。誰もが警告する借用チェッカーでさえも。

人々はここで戦争物語を期待する。私を辱めた言語、コンパイラとの戦いに費やした月。私にはそれがない。ドラマを作り出すより、なぜスムーズに行ったかを話す方がいい。

Swiftが私を準備した

Rustの多くは親しみがあった。なぜならSwiftが同じアイデアをすでに私に教えていたから、ただ異なる服を着て。

列挙型とパターンマッチングが大きかった。Swiftの列挙型は本物の直和型であり、長年それと`switch`に頼ってきた。Rustの`enum`と`match`は同じ筋肉だ。ResultとOptionも新しくなかった。SwiftのオプショナルとResultに長く住んでいたので、「これは値か何もないかだ」と「これは値かエラーかだ」はすでに私のコードについての考え方だった。Rustはただ両方を常に声に出して扱うことを要求する。それは私にとって新しい規律ではなかった。すでに持っていた規律が今や強制されるようになったものだ。

トレイトはだいたいSwiftのプロトコルとして着地した。同一ではないが、外国の概念を学んでいるほど違わない。知っているものの方言を学んでいた。「振る舞いを定義し、型をそれに準拠させる」は両方で同じ形だ。

だから言語は壁のように感じなかった。半分前に住んでいた場所のように感じた。Swiftが私に染み込ませた値型とオプショナルの習慣が、借用チェッカーが決して戦いにならなかった正確な理由だと思う。すでに所有権と「誰がこの値を持つか」という観点で考えていれば、チェッカーはほとんどすでにやろうとしていたことを伝えているだけだ。習得しなければならない新しい考え方ではなかった。すでに知っているゲームのためのより厳格な審判だった。

そして作るものの形はシンプルで私が望んでいたものだ。fledgeは単一のRustバイナリで、Cargoでビルドされる。1つのクレート、もう一方の端から1つのバイナリ。

エージェントが重い仕事をした

流暢なRustプログラマーのふりはしない。そうではない。流暢なSwiftプログラマーでRustをよく読んで舵を切れるだが、それは異なるものだ。

ギャップを埋めたのはエージェントだった。流暢でない言語で生産的になるために、それらに強く頼った。親しみのある部分は自分で読んで推論して指示できた。コードに何をしてほしいか、良い構造がどんなものかを知っていた。なぜならそこは言語をまたいで転用できるから。Rustが独自のイディオムを持つ部分、物事を言う独自の方法、エージェントが担った。

これは区別する価値があり、反対側から見るとそうだ。私のSwiftは手書きだ。私のRustはエージェント増幅だ。1つでは私がキーを打つ著者だ。もう1つでは正しいものがどんなものかを知っていて、エージェントにそれを向けてその仕事をチェックする人間だ。

正直なところ、この方法でRustでfledgeをビルドすることは、その背後にある全テーゼの小さな証明だ。そのツールは人間とエージェントの両方のために作られた。人間とエージェントの両方によって*作られた*。今日fledgeを動かすエージェントはそもそもfledgeを書くのを助けた。

ツールチェーンは安心感だった

思ったより楽しんだ部分がここにある。Rustのツールチェーンは安心感だった。

Cargoはただ動く。ビルド、テスト、依存関係のための1つのツール、すべて、どこでも同じだ。クレートエコシステムは深い。必要なものは何でも大抵しっかりしたクレートがあり、それを引き込むのは1行だった。そして単一バイナリビルドがまさにfledgeがなりたかったものだ。どこでも実行する1つの軽量バイナリを出荷する必要があり、Rustはそれをデフォルトで手渡してくれる。

対照的なのはAppleのプラットフォームを離れたときのSwiftツールチェーンだ。Appleでは、Swiftのストーリーはグレートだが Apple以外では（Linux、クロスプラットフォーム、fledgeが必要とした「どこでも1つのバイナリとして実行」のターゲット）難しくなる。それがfledgeがSwiftではなくRustである本物の理由だ。Swiftが私のホーム言語でも。どのマシンにも投入できる軽量の単一バイナリが欲しかった。エージェントがどこでも実行できるもの。そしてRustのツールチェーンがそれを戦いではなく簡単な道にしてくれた。

だからそれがビルダー視点の真実だ。Swiftが考えを準備し、エージェントが私が持っていない流暢さをカバーし、ツールチェーン（Cargo、クレート、単一バイナリ）が切り替えを

喜んだ実際の部分だった。言語を選ぶことは簡単な決断だった。実際に考えを要した仕事は他のすべてだった。fledgeが何であるべきかを理解すること。

私の使い方と、誰が使っているか

正直な部分をリードさせてほしい。最後のために取っておく代わりに。fledgeは広く使われていない。私のもの、私のエージェントのもの、私のサークルのものだ。それが要点だ。ツールはまずビルダーの実際の問題を解決するために作られ、そして毎日解決している。目の前の問題を解決するツールは、ロゴウォールがあって勝負に参加していないツールより優れている。

だからこの章が答える質問は「何人がfledgeを使っているか」ではない。「誰が、そしてなぜそれが正しい順序か」だ。答えは「みんな」ではなく、そうなることを意図していなかった。

私のエージェントが私より多く動かしている

私がfledgeをどう使っているかを想像するとき、人々が間違えるのは、*私*が使っている姿を想像することだ。それも起きるが、それはもうfledgeが実行されるメインの方法ではない。私のエージェントが動かしている、大差で。エージェントがリポジトリで作業しているとき、fledgeは変更したものをどうbuild、test、実行するかだ。私のだけでなく、その実行インターフェース。それがまさに第1章が設計はそのためだと言ったことだ。大抵の日、エージェントは私よりそこから多くを得ていて、私はほとんど舵を切っている。

それが私がビルドしたユーザーベースだ。見知らぬ人の群衆ではない。私と私の持つすべてのリポジトリのエージェントたち、常に。

誰が他に使っているか

fledgeはオープンソースで、Homebrewのタップにあり、誰でも`cargo install`できる。それはどれも採用の主張ではない。タップは配布チャネルであり、ユーザー数ではない。1つを他に見せかけるつもりはない。

今日のユーザーベースは私、私のエージェント、そして実際に一緒に仕事をするCorvidLabsサークルだ。広い外部採用はない。問題を提起する見知らぬ人のコミュニティはない。個人とサークルのインフラストラクチャであり、そう言わずに存在しないうねりを示唆するより、それをはっきり言う価値がある。

コラボレーターが数だけでなく絵に重要だ。fledgeがサークルにわたって共有されたインターフェースであることは設計の一部だ。CorvidLabsの誰かが私のリポジトリの1つを開くとき、そのリポジトリのプライベートな方言を学ぶ必要がない。すでに動詞を知っている、ど

こでも動詞が同じだから。再学習を省いてくれるのと同じ一貫性が、私が一緒に仕事をする人々を彼らがすでに知っているインターフェースに着地させる。

使うべきか

おそらくまだそうではない。正直なところ、私のように仕事をする人だけが使うべきだ。多くのリポジトリ、ループのエージェント、すべてにわたって1つの一貫したインターフェース。それがあなたの問題なら、fledgeはそれを解決する。そうでなければ、ツールは過剰であり、それを売り込もうとするより言う方がいい。

限られた採用が設計がスケールで機能することを証明するとは主張しない。証明しない。設計が私に機能することを証明する。それは小さな主張であり、私が支持できる唯一のものだ。fledgeは毎日、ほとんどエージェントを通じて、私が舵を切って小さなサークルが同じインターフェースにいる、私自身の世界の中でその仕事を稼ぐ。それがその仕事をするツールだ。広さは後で来るかもしれないし、決して来ないかもしれない。仕事はすでに行われている。

---

fledgeの行き先

「fledgeは次にどこへ行くのか」への正直な答えは人々が期待するより刺激的でなく、それは良い兆候だと思う。ほとんど成熟している。大きな形は作られている。残っているのはほとんど保守し、それを使い続けることだ。成長させたい方向がいくつかあるが、fledgeが再発明を必要としないので、再発明の壮大なロードマップの上には座っていない。他のすべてが乗っているものであり続けることが必要だ。

より大きなエコシステム

プレイグラウンドとインテグレーション層は開いている。誰でも尋ねずに追加できる。コア層はそうではない。その非対称性は意図的であり、変わらない。

最も余地がある方向はエコシステムの章がすでに説明したものだ。より多くのプラグイン。fledgeはコアが太るのではなくコアの周りの輪が成長することでより有能になる。より豊かなプラグインのセット。どのリポジトリにfledgeを投入しても、すでにそれを処理する方法を知っているものがある。より多くのインテグレーション、それぞれが1つの本物のことを解決する小さな一発ネタ。

それに結びついているのは、fledgeがすぐに検出して実行するものを広げること、第1章からのゼロコンフィグの約束を、より多くの場所で保つことだ。まだ知らないすべての言語とプラットフォームは、同じ動詞がただ機能する約束にギャップのあるリポジトリだ。だから検出とプラットフォームカバレッジを埋めることが他の明白な方向だ。華やかではない。カバレッジはユニバーサルインターフェースを実際にユニバーサルにするものだ。

そして今最も重要なカバレッジは次の言語ではない。オペレーティングシステムだ。fledgeはWindows、Linux、macOSの3つすべてで一流で実行しなければならない。1つのOSでしか本当に快適でないツールはユニバーサルインターフェースではなく、ローカルなものだ。だからクロスOSサポートが次の本当のフロンティアであり、別の言語検出を追加するより重要だ。

エージェントが依存するものを保守することの意味

ここで機能をリストして再発明を約束することもできる。それは fledge が今何であるかの間違ったフレームだ。

2026年に重要な質問は fledge が何の機能を追加するかではない。コードは安い。エージェントがそのほとんどを書く。希少性は機能ではなく、本当に信頼できる実行インターフェースだ。エージェントは私がレビューできるより速く新しい機能を生成できる。生成できない

のは、何年ものドッグフーディングを経て、安定したインターフェースを持ち、パイプラインが当てにできる既知の動作を持つツールだ。

何年ものドッグフーディングを経て、パイプラインが当てにできる安定したインターフェースを持つツール、それがfledgeであり、ビルドするより難しいものだ。技術的に難しいのではない。安定性が出力であり、副産物ではないと決断することが必要だから難しい。すべてのツールは新しい機能として生まれる。インフラストラクチャになるまで生き残るものはほとんどない。そうするものは保守者が「何も壊れなかった」を遅いスプリントの慰め賞ではなく、主な達成として扱ったものだ。

fledgeを毎日すべてのリポジトリで使っている。私のエージェントが常に動かしている。つまりバグが私を見つける。欠けている動詞、間違って推測した検出、間違った順序で発火したプラグインフック。それを使って生きているから当たり、使って生きているから修正する。ドッグフーディングは副次的な活動ではない。それが全品質メカニズムだ。保守と使用は同じ仕事だ。

本書のはじめにはツールが依存する価値を持つとはどういうことかを問うた。それが指し示した答えは、1つのクリーンなインターフェース、隠れたインタラクティブなステップなし、本物のプラグイン境界、コードが変わっても誠実なスペック。それを作れば、エージェントは初日から動かせる。そもそも人間を必要とするように作られていないから。

fledgeはそれだ。単一の設計決断からではなく、インターフェースを侵食するショートカットを拒む度に積み重なった結果として。動詞はすべてのリポジトリで同じだ。JSONは同じ形だ。`fledge introspect`マニフェストは最新に保たれる。自分の仕事で最初に実行していないものは何も出荷されない。

エージェントがほとんどのツールを操作する世界では、信頼を獲得するものは長い機能リストではない。十分に長い間同じように現れ続けることで、その上に本物のものが作られることだ。ロードマップスライドより静かな野心であり、それが重要なものでもある。

だからここからの仕事はこれまで常にそうであった仕事と同じだ。しっかり保ち、それを使って生き続け、インターフェースを誠実に保つ。バグにぶつかるのは使い続けているからで、修正するのも同じ理由だ。それが全体の計画だ。

著者について

0xLeif（leif.algo）はオープンに作る。AppState、Cache、Forkのような小さくて合成可能なSwiftライブラリを10年間。CorvidLabsのラボ。「これが存在すればいいのに」から始まったエージェントツールのスタック。キーボードから離れると彼はZach Eriksenだ。

これらの本はインタビューで、章に整形されて実際のコードに対して確認されたものだ。

github.com/0xLeif · leif.algo

謝辞

CorvidLabsに感謝する。これらのアイデアがテストされ、議論されて形になる部屋であることに。

このスタック全体が乗っているオープンソースメンテナーたちに感謝する。これは一人では作られない。

そして「オンラインで無料」を続けられるようにしてくれるアーリーリーダーと「払いたい分だけ払う」サポーターに感謝する。

コロフォン

Markdownから組み上げ、bookgen（小さな純Rustパイプライン、Pythonなし）でビルドされた。

インタビュー駆動でAI支援。手で編集・事実確認された。emdashなしで書かれた。カバーとチャプターアートはAlgorand上のCorvidとNatureコレクションから。