ファースト・クラス

人間とエージェントの両方のために作る

ZACH "LEIF" ERIKSEN

著作権

オンラインで無料公開。ePub版は「払いたい額を払う」形式です。役に立てたと感じたら、活動を支援していただけると嬉しいです。

github.com/0xLeif · leif.algo

エージェント・スタック・シリーズ全4冊のうちの1冊。制作過程については巻末のコロフォンを参照してください。

献辞

オープンに作り、それでも世に出す、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ますが、一つのセットとして書かれました。コードは安くなり、信頼は希少になりました。合わせて読むと一つの主張が浮かびます。今何を作るべきか、そしてどう信頼するか、です。

- **The Agent Developer's Field Guide:** 実際のコードを届けるエージェントのためのツール、仕様、信頼の構築
- **ファースト・クラス:** 人間とエージェントの両方のために作る *(本書)*
- **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:** 人々が実際に使うツールの作り方

オンラインで無料公開。各ePub版は「払いたい額を払う」形式です。

目次

はじめに

これは4冊の中で最も短く、頑固な本であり、他の3冊が実際に主張の根拠としている本でもあります。

ほとんどのツールは人間のために作られ、後からエージェントが横に取り付けられます。二番目のAPI、フラグ、実際の製品の半分しか機能しない特別モード。私はそれが逆だと思います。ここでの主張はシンプルです。ツールは両方にとってファースト・クラスであるべきです。人間はエージェントなしでも動かせるべきで、エージェントは人間なしでも動かせるべきです。同じ表面で、二流の入口なしで。

私はこれを理論ではなく、作ることから学びました。小さなSwiftライブラリとデベロッパーツールを作っていて、時代を超えて通用したものは、コアが十分にクリーンで何でもそれを呼び出せるものでした。エージェントが登場したとき、それらのツールはすでに準備ができていました。準備ができていなかったものは、今でも謝り続けているものです。

*Field Guide*がメソッドで、*Building Agents*と*Open Source Tooling*が証拠だとすれば、これはそれらすべてが奉仕する論文です。この本を使うために他の3冊は必要ありません。次に作るか選ぶツールへの見方を変えるためにここにあります。ツールが人間とエージェントを同じ種類の市民として扱っているかどうかを問い、その答えからどれほど多くのことが導かれるかを見てください。

短いのは意図的です。一気に読んでください。

両方が市民

私のツールの多くは一つのアイデアから来ています。人間とエージェントは同じツールを使うことになる、ということです。

だから本書全体が基盤とする定義を、最初に率直に述べます。ファースト・クラスとは、人間がツールを一人で動かせて、エージェントがツールを一人で動かせて、同じコマンドが両方向で使えることです。二つの製品ではありません。一つのツール、二種類のユーザー、どちらも相手が変換される特別なケースではない、ということです。

*ファースト・クラス*と言うとき、プログラマーが言う意味で言っています。ツールが実際に構築した対象である、本物のサポートされた参加者であり、それが容認するゲストではありません。フライトのアップグレードではありません。ファースト・クラスのユーザーとは、ツールが設計された対象であり、本物の入り方がある人のことです。他の誰かの形に変換される必要がある人ではありません。

エージェントだけのためのエージェント・ファーストのツールがあってもいいです。人だけのためのヒューマン・ファーストのツールもあっていいです。面白いケース、ほぼ誰も作っていないケースは、両方であるツールです。「両方」が私が*両方にとってのファースト・クラス*と言うときの意味であり、本書の残りの意味でもあります。エージェント・ファーストよりも厳密なことです。エージェント・ファーストはエージェントを満足させるだけでいいのに対し、両方にとってのファースト・クラスは、どちらの側も劣ったバージョンを得ることなく、同じ表面で人間とエージェントの両方を満足させなければなりません。今のところ私たちはほとんどヒューマン・ファーストのプロジェクトを持っていて、エージェントをそこに*持ち込んで*います。エージェントは遅れてやってきて、他の誰かのために作られたツールに、なんとかなるだろうと希望しながら。

実際に必要なのは、ヒューマン・アンド・エージェント・ファーストのプロジェクトです。最初からそのように作られたものが。

「後からエージェントを取り付けたヒューマン・ファースト」がデフォルトなのは、抵抗が最も少ない道だからです。すでにツールがあり、すでに自分には機能していて、エージェントが来たとき簡単な動きは持っているものに少し接着剤を包むことです。まあ機能はします。人間の目と人間の手を中心に設計されたツールを取って、どちらも持たないプロセスに両方持っているふりをさせたのです。それがエージェントにかけるコスト（プロンプトでの停止、読めない出力、毎回の再学習）については後で独自の章があります。ここでは、エージェントが人間を想定していたすべての行で穴を埋め続けることになると言えば十分です。

だから仮定を逆転させましょう。ツールをどちらの方法でもファースト・クラスで機能するように作る。それがビルド指示として再述した冒頭の定義です。本物の製品と横に貼り付けられた「APIモード」を持つのではなく、CLIと何かを変更するたびに同期が取れなくなる別個のエージェント・シムを持つのでもありません。両方とも本物のユーザーです。両方ともそこにいるはずです。

そして、私にとって最も重要な部分がここです。エージェントがファースト・クラスのユーザーであるとき、ツールはコマンドと動作を推測させるのではなく、実際に*助ける*ことができます。人間は紛らわしいツールをなんとか使えます。READMEを読んで、試して、エラーを読んで、別のことを試して、同僚に聞くでしょう。エージェントがなんとかしようとするのは単に高価な推測です。エージェントのために作られたツールは何が可能かを教えてくれて、直接使える出力を渡してくれて、次に何をすべきかを教える方法で失敗します。

これは人間が求めるのと同じツールです。発見可能なコマンド、信頼できる出力、何が悪かったかを教えてくれるエラー。エージェントは人間のように穴を埋めることができないだけで、エージェントのために作ることで穴を塞ぐことが強制されます。両方のために設計するとソフトウェアが良くなります。

「エージェントのためにも作る」と聞いて、人々は二つの製品か、各側が望むものの劣ったバージョンを得る妥協案を想定します。それは両方に公開する一つのステップがある一つの良いコアです。それは仕事の倍ではなく、次の章がその理由を示します。

私がこれに戻り続ける理由は、エージェントが時間とともにやることが増えるだけで、減ることはないからです。今日、彼らは人間のツールをぎこちなく使っています。明日、彼らは仕事の実質的な分担をしていて、その割合は上がります。今作るツールが人間がいつもそこにいてプロンプトを処理して、画面を読んで、ボタンをクリックすることを想定しているなら、毎月より偽になっていく前提の上に未来を構築しています。

この議論のより大きなバージョンもあります。完全に人間によって書かれたコードの十年があって、今AIがそのすべての上にコードを書いていて、ある時点でその混合が転換します。ツールはそれが良くなるか悪くなるかを決めるものです。それは独自の章に値します。橋です。

今のところ主張はこれだけです。人間とエージェントは同じツールを使います。だから最初から、平等なファースト・クラスの市民として、両方のために作りましょう。

エージェントにとってのファースト・クラス

では、エージェントがファースト・クラスのユーザーになるために、人間のツールをぎこちなく使うのではなく、ツールには実際に何が必要でしょうか？

四つのことです。どれも特別なものではありません。

構造化された機械可読な出力。 本物のシリアライゼーション形式、私が手を伸ばすのはJSONです。エージェントが画面ではなくデータを受け取れるように。ほとんどのツールは見栄えの良いテキストを返します。整列したカラム、色、下部のサマリー行。それは人間の目のためのものです。エージェントはそれをスクレイプしなければならず、スクレイピングはスペーシングを変えた日に壊れます。実際のデータを渡してください。段落を解析する代わりにフィールドを読みます。

発見可能で一貫したコマンド。 ツール全体で一貫した動詞、そしてエージェントが何が可能かを見つけるために読める自己記述的な`--help`。ヘルプテキストが本物であれば、エージェントはそれを読んでツールが何をできるか知ります。このツールを以前見ている必要はありません。ツールに聞けば、ツールが答えます。

次のステップを導くエラー。 何かが失敗したとき、何をすべきかを言ってください。スタックトレースではなく、`error: 1`でもありません。人間は掘り回って裸のエラーコードを解明できます。エージェントが裸のエラーコードを受け取ると行き詰まるか、さらに悪いことに間違ったことを自信を持ってします。

非インタラクティブで決定論的。 驚きのプロンプトなしで実行されるので、エージェントは決して待ちぼうけになりません。ツールが突然「本当によろしいですか？[y/N]」と聞いて永遠に待ち続けることは、エージェントの実行を完全に止めます。答える人が誰もいないからです。フラグを渡してまっすぐ実行できるようにしましょう。同じ入力が同じ結果をもたらすよう決定論的にしましょう。

そのリストについて気づいてほしいことがあります。すべての項目がCLI設計の普通の良い作法です。エージェント特有の魔法は何もありません。人間も明確なエラーと予測可能な動作と発見可能なコマンドを求めています。エージェントはそれらがない場合に肩をすくめて回避することができないだけです。だからエージェントのために作ることは、ツールを良く作り、「まあ、人間がなんとかするでしょう」に頼ることを断ることです。

今、人々が間違える部分があります。エージェントのための設計と人間のための設計は分離すると思っています。二人の主人に仕えていて、誰かが負けると。それらは一緒に引っ張ります。実際の形はこうです。

本当に良いコアを設計して機能させます。それから公開します。フラグを追加します。非インタラクティブ、または`--json`、またはintrospect、ツールが必要なものを何でも。これでエージェントはCLIツールを使えるし、人間もCLIツールを使えます。同じツール。同じコア。一つのものを作って二度公開しました。

そこからさらに拡張されます。人間が使うためのUIを増やす。UIのないツールだけ。または人間が作るプラグイン、またはエージェントを助けるプラグイン、そういった違うものたち。拡張は聴衆が実際に分岐するところです。人間はアフォーダンス、良い表面、UIを求め、エージェントはintrospectionとプラグインとモードを求めます。構いません。それらを作りましょう。しかし共有コアの*上*に、拡張として作ってください。永遠に同期を保ち続けなければならない二つの別個の製品としてではなく。

コアでは同じものです。 私が戻り続ける行はこれです。ただエージェントと人間に公開しなければなりません。だからはい、余分なステップがあります。デュアルユースのプロパティは無料ではありません。意図的にコアを両方の方法で公開する必要があります。

そのリフレームがこの章の要点全体です。なぜなら開始前に反論を解消するからです。両方のために作ることに人々が抵抗するのは、それを倍の仕事として価格をつけるからです。二つの設計、二つのテストスイート、維持する二つのもの、壊れる二つのもの。実際には一つのコアと公開ステップです。コアが高価な部分で、どのみち作るつもりでした。エージェントのために公開することは主に構造化された出力と一貫したコマンドと良いエラーの規律であり、どのみちするべきだったことです。

ここに潜む順序の問題があります。コアが先に来るのか、それとも両方のために同時に設計するのか？それは独自のものとして十分で、後で独自の章があります。今のところ：コアを両方の方法で公開して、その公開が安い部分です。

私自身のツールはまさにこの方法で作られています。fledgeを見てください。本物の`introspect`動詞があり、`fledge introspect --json`の仕事全体は、エージェントが人間向けの`--help`テキストをスクレイプする代わりに、利用可能なコマンドを構造化データとして発見できるようにすることです。これが公開ステップが文字通りになったものです。同じコアですが、エージェントは「ここで何ができますか？」と聞いて機械の答えを返してもらえます。それからコマンド自体をヘッドレスで実行します。`FLEDGE_NON_INTERACTIVE`を設定して何も質問のために止まらないようにして、`--json`を渡して出力がデータとして返るようにすると、エージェントはプロースを読むことに一度も依存しないファースト・クラスの入り方を持ちます。

具体的にしましょう。`fledge doctor`はプロジェクト環境を確認します。普通に実行すると目のためのものが得られます。整列したカラム、行ごとのチェックマーク、絵文字のステータス、サマリー行：

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

  8 checks passed, 0 issues found
```

そのブロックはエージェントが詰まる「before」そのものです。きれいで画面です。リモートが設定されていることを知るために、エージェントは正しい行を見つけて、緑のチェックマークを認識して、次のリリースでスペーシングが変わらないことを信頼しなければなりません。気にするステータスはカラムに埋まった絵文字です。実際には：エージェントは整列したテキストをスクレイプし、次のリリースでより長いリポジトリ名がカラムを2文字ずらしたときにパースが壊れ、エージェントはリモートチェックが欠けているのに通過していると読んでしまいます。本物の契約ではなかった文字列マッチで間違ったブランチに進んだのです。

同じコマンドを`--json`付きで実行すると、同じチェックがデータとして返ります：

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

`schema_version`は意図的に最初のフィールドです。エージェントが他の何よりも先に読むものであり、ドキュメントの残りがどの形をしているかを教えるからです。出力形式が変わる日には、その数も一緒に変わり、バージョンに固定したエージェントは古いデータとして新しいデータを黙って誤読する代わりに適応することを知ります。これが契約がツールの改訂を生き残る方法です。

それ以降は同じコアで、同じチェックが実行されました。人間はチェックマークとサマリー行を得ます。エージェントはカラムから緑のチェックマークをスクレイプする代わりに分岐できる`status`フィールドを得ます。`fix`フィールドはここでは`null`です。何も問題ないからです。これが契約です。`fix`は通過しているチェックでは`null`です。

今度は何かが*間違*っている場合に実行します。リポジトリにリモートが設定されていないとしましょう。プレーンバージョンは行を薄暗くしてエージェントは推測に戻ります。`--json`バージョンはそれを動作できるレコードに変えます：

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

これがエージェントが実際に動作するブランチです。`status`は`"ok"`から`"missing"`に変わり、`detail`は何が悪いかを平易な言葉で言い、`fix`は今や文字列が入っています。エージェントが修正するために実行できるリテラルコマンドです。エージェントはリモートが欠けているとはどういうことかを知ったり回復を考え出したりする必要はありません。問題を見つけたツールが修正も渡してくれました。人間は薄暗い行を得て、エージェントは分岐できる`status`と実行できる`fix`を得ます。一つのコマンド、二度公開。

うまくいったかどうかのテストはシンプルです。エージェントなしで人にツールを渡す。機能して良いか？人間なしでエージェントにツールを渡す。機能して良いか？両方の答えがはいで、そこに到達するためにツールを二度作らなかったなら、正しくできています。

橋

前の章の最後より一つ前にこれに触れました。より大きなバージョンの議論、パーセンテージのことがあります。これです。

糸を拾い直しましょう。100%が人間によって書かれたコードの10年以上があって、あらゆるリポジトリのあらゆる行が人によって書かれていて、それは大量のレガシーコード、大量の技術的負債、10年分のものがすべてを動かしていることを意味します。それが今の世界の実際の状態です。

そして今、私たちはそのすべての上にAIを使っています。

それが難しい部分であり、人々はそれについて十分に考えていません。すでにそこには膨大なレガシーと人間のものがあります。AIはそれに*登場します*。

そして続けていくなら、AIにさらに依存してものを書くなら、本当の疑問になります。どの時点で100%のAI書きコードに転換するのか？そうなることがあるのか？そしてすでにあった10年の人間コードをどうするのか？そのままレガシーとして残して進むのか？書き直すのか？なかったことにするのか？実際に良い答えを持っている人はいないし、ほとんどの人は質問さえしていません。持っているものにエージェントを取り付けて希望するだけです。

人間すべてから次に来るものへのその全体の移行が、正しいツールが何のためにあるかです。ツールはその橋です。

理想的には、*人間とAIの両方*が使えるツールをセットアップします。spec駆動開発、spec-sync、そしてコマンドのためにfledgeを使って書いて構築するようにセットアップします。それが橋です。これらのツールを使い始めると、既存のコードにエージェントを導入しやすくなります。そして全く新しいプロジェクトを始めると、人間が動かしやすくなります。同じセットアップが、エージェントが10年前の混乱の中に入っていくのを助け、人が全く新しいものをコントロールし続けるのを助けます。

より良いプロンプトやよりスマートなエージェントではなくspecです。なぜならspecは両側の共有の真実の源だからです。人間が意図を述べます。エージェントはそれに向けて構築します。それは一つ上のレベルでの前の章と同じ問題です。人間のツールをぎこちなく使うエージェントはコマンドを推測します。specなしでコードベースに落とされたエージェントは*意図*を推測します。specは代わりに読むものです。

そしてコードをspecに対して誠実に保ちます。これが橋を単に希望的なものでなく信頼できるものにする部分です。spec-syncはspecとコードを合意のままに保ち、CIで強制されるので、エージェントは合意されたものからこっそり逸脱できません。それは聞こえる以上に重要です。エージェントに関して誰もが恐れている失敗モードは劇的なものではありません。

静かなものです。エージェントがゆっくりと要求したものとは違うものを構築して、深くなるまで誰も気づかないというものです。specに対してコードを確認するCIが、まだ小さいうちに逸脱を捉えるものです。契約は両側から読まれ、機械が誰も壊していないことを確認します。

その機械が実際に確認することはこれです。specはmarkdownファイル、`*.spec.md`で、契約を書き留めます。目的、公開API、不変条件、エラーケース。spec-syncはそれを実際のコードと両方向で照合します。specが文書化しなかったエクスポートはフラグが立てられます。コードが持っていないspec上の約束はエラーです。宣言されたスキーマを実際のマイグレーションに対して同じ方法で確認します。構造的です。公開サーフェスとスキーマが書き留められたものと一致しているか。生成されたテストでもプロースに対するファジーdiffでもなく、その狭さが私が信頼する理由です。契約とコードが物事の形について合意していると主張するだけで、それ以上ではありません。CIでは本物のゲートとして実行されます。それらが逸脱したとき非ゼロで終了し、逸脱をプルリクエストに直接投稿します。

それが私が実際に気にする分業をセットアップします。人間が意図を所有します。エージェントがグラインドを所有します。人間は意図のレベルでコントロールを保ち、何をすべきでなぜそうすべきかを、エージェントが仕様に対して実装を行う間。各側が実際に得意なことをして、specを境界線として。

そしてそれはレガシーコードへの安全なオンランプです。これがパーセンテージ問題に戻るところです。仕様書を持たない10年のコードがあります。誰も書かなかったから、人間がそれを書いてその意図が人間の頭の中にあったからです。動きは：一部がすべきことをspecとして捉えて、エージェントをそれに対して作業させる。エージェントに10年前のコードから意図を読み取らせようとしているのではありません。先に意図を書き留めています。人間にそれができます、それが人間が得意なことです。そしてエージェントは構築と改善のための契約を持ちます。それがエージェントが既存のコードに入って災害にならない方法です。specがオンランプです。

specを持たない本物の関数での動きを想像してください。前：動作するコード、書かれた契約なし、元の著者の頭の中に完全に住む意図、したがってエージェントがそこに落とされるとシグネチャを読んで不変条件を推測して物事を変え始めます。後：隣に`*.spec.md`が座っている同じコード、公開APIと不変条件が平易な言語で書き留められていて、spec-syncがCIで書かれたものにコードが誠実であるよう強制しています。コードベース全体を一度にやる必要はありません。引き渡そうとしている部分の契約を書きながら、一度に一つずつやります。

specはコードが何をしたかを変えませんでした。エージェントがコードでできることを変えました。推測する代わりに書かれた契約に対して作業できるように。

fledgeはそのすべてのコマンドサーフェスです。人間とエージェントの両方が作業を進める一貫した動詞、前の章のツールで構築とテストと実行の方法を推測させないもの。spec-

syncが契約です。fledgeがそれに対して行動する方法です。

それらの間のシームがあります。spec-syncは独立して立っています。独自のツール、独自のCIチェック。しかしfledgeはspecを開発ループのファースト・クラスの部分として扱います。他のものと並んで`spec`動詞があり、`fledge ask`と`fledge review`はspec対応なので、specはエージェントが質問に答えるときや変更をレビューするときに読むコンテキストです。specはエージェントが無視するドキュメントではありません。fledgeがエージェントに作業のために渡すものです。

だからゲートは両端から発火します。fledgeの`spec`ピラーは開発ループ内でspec-syncチェックをそこで実行し、`CorvidLabs/spec-sync@v4` GitHub ActionがCIで再び実行するので、こっそり逸脱したものはマージされません。エージェントが作業をしているとき、そのチェックはループの中で動き、最後に気づく代わりに進みながらspecに沿ったままになります。

さて最後まで追ってください。なぜなら最後がすべての理由だからです。これが機能するなら、ツールとspecsとエージェントを誠実に保ちレールが単にそこにあれば、人間が移動します。意図と方向へと上に。エージェントがspecに対して下でグラインドして、確認できるオープンな場所で。その動きは後で独自の章があります。ここでは橋がどこに導くかを知れば十分です。

パーセンテージは動き続けます。より多くのコードがエージェントによって書かれるようになります。それは単に事実です。私たちにかかっているのは橋です。共有ツール、共有契約、そして私たちが実際にそれを作るかどうか。ほとんどの人はまだ始めていないと思います。

アイデンティティの壁

ツールに入る前に、正直な限界を述べます。この議論全体は壁にぶつかります。より良いツールで壊せる壁ではないので、はっきり早めに言いたいです。プラットフォームはエージェントがそれ自身として存在することを許しません。この章の後のすべてはそれを踏まえてすることです。

エージェントがあなたのツールのファースト・クラスのユーザーであるなら、遅かれ早かれプラットフォームのファースト・クラスの市民でもある必要があります。それらは二つの方向を向いた同じアイデアです。エージェントを本物のユーザーとして扱うツールは、構造化された出力と発見可能なコマンドと人間に依存しない入り方を与えます。エージェントを本物の参加者として扱うプラットフォームは、アカウント、アイデンティティ、他の全員の中に合法的に存在できる場所を与えます。その二番目のものが得られないものです。

試みました。誰もが作っているプラットフォームでエージェントに独自のアイデンティティを与えようとしました。私のラッパーではなく、独自のアカウントを。プラットフォームは許可しません。Building Agentsの本でその話をちゃんとしています。ここで受け取るべきはそれがどんな種類の壁かということです。真に自律的なエージェントの前にある障壁はモデルの能力ではなく、安全性でさえありません。プラットフォームがエージェントが存在するためのアイデンティティを与えないことです。

技術的なものではなく、ポリシーの壁です。VMを実行できます。ループを配線できます。モデルは作業ができます。それらはどれも障壁ではありません。障壁はプラットフォームがエージェントがそこにいることを許可させられないことで、それは私が修正するものではありません。私が作れるすべての上流にある、誰か他の人が下した決断です。

なぜ重要かというと、私が実際に求めるモデルである説明責任のためです。最終状態は野放しのエージェントではありません。本物の、名前付きの、スコープされたアイデンティティを持つエージェントです。完全な能力、縮小した権限、人間の承認ゲート。作業をして、プルリクエストまで届けて、マージは私のものです。なぜなら何かが私の名前の下で世界で行動するなら、署名するのは私だから。しかし*アカウンタブル*は*アイデンティティ*なしには持てません。名前をつけられない、スコープできない、「あれがこれをした」と指摘できないエージェントは、説明できるものが何もありません。アイデンティティを否定することで、説明可能なバージョンも一緒に否定して、残るのはエージェントなしか説明不可能なものかです。

だから本書の残りのフレームはこれです。エージェントは私が作る*ツール*のファースト・クラスのユーザーになれます。それは私のものだから。私が所有しない*プラットフォーム*のファースト・クラスの市民にはなれません。それは私のものではないから。この章の後のすべ

てはその限界の中で作られています。これがエージェントにツールへの本物の入り方を与えられるが世界での本物の場所は与えられないときの両方にとってのファースト・クラスの見た目です。AIでも、技術でも、安全性でもありません。プラットフォームがエージェントを存在させません。それ以外はすべて作れます。それだけは、まだできません。2026年現在、壁はまだ立っている。通り抜ける方法は自分で動かす回避策しかない。実際の活動履歴を持つ年季の入った人間アカウント（新鮮なエージェントアカウントはすぐにブロックされる）か、自分でホストする検証レイヤーかだ。プラットフォームはまだエージェントに本物のアイデンティティレーンを与えていない。それが現状だ。

後付けの場合

「両方のために作る」とうなずいて、失敗を実際に想像しないのは簡単です。だからエージェントの側からヒューマン・ファーストのツールを体験させてください。そこで感じるからです。

二つのことが間違います。そして常に間違います。

最初はエージェントが行き詰まります。ツールは常にインタラクティブなプロンプトで引っかかります。確認、「本当によろしいですか？」、キーストロークを待って座っているもの。そしてそこに誰もいません。だから実行は失敗しません、正確には。ただ止まります。人が見てEnterを押すと書かれたプロンプトに座り、エージェントは待ちます。待つことがツールが与えた唯一のことだから。誰も答えられない質問に引っかかった実行全体が死にます。

二番目はドキュメントです。ないか、あっても紛らわしいかです。つまりエージェントはツールを*学習*しなければなりません。そして私が気づき続けている部分があります。実際には学習しません。できません。その知識が実行間で住む場所がありません。だから高価なバージョンをします。ファイルをスキャンして、インデックスにあるものを読んで、独自のメモを取って、ゼロからツールの動作を再構築します。毎回。ツールは自分に何ができるかを*知っています*、コードの中にあります。エージェントには決して言いません。だからエージェントはすべての実行でその絵をゼロから再構築します。

どれだけ無駄かを考えてください。人はドキュメントを一度読んで、多分ざっと目を通して、その後頭の中に持ち運びます。ツールの感覚を築きます。エージェントはそれを無料で得ません。ツールが直接言わないことは何でも、また掘り起こして、また払って、また推測しなければなりません。ツールが一度やるべきだった作業を、エージェントは永遠に繰り返します。

これらは両方とも二つの格好をした同じ間違いです。ツールは人間がいることを想定していました。プロンプトでの人間の忍耐、ツールの動作の人間の記憶、エージェントはどちらも持っていません。肩をすくめて待てません。実行間の壁を越えて何かを手渡されない限り、覚えていられません。

そして両方ともいくつかの章前のリストに直接マップします。非インタラクティブ：来ない人を待って止まらない。発見可能：エージェントに自分のソースコードから再構築する必要なしに読める形で何ができるかを教える。引っかかりと再学習はエキゾチックなエージェント問題ではありません。欠けている二つのプロパティで、人間が毎回静かに穴を埋めていたので穴があることに気づかなかっただけです。

ヒューマン・ファーストのツールにエージェントを取り付けることはたいてい機能します。エージェントがそれを機能させるために何をしなければならないかを見るまでは。するとツールが人間にずっと頼り続けていたことがわかります。

私のもの一つを名指しします。fledgeは最初、手で動かすタスクランナーとして始まりました。最初からコアはクリーンで、ビルドとテストと実行のロジックはコマンドの内部ではなくコマンドの背後に座っていたので、最初にエージェントを指向けた日にほとんどが機能しました。一カ所を除いて。エージェントは`fledge work commit`を実行して完全に止まりました。そのコマンドは`Commit message:`を印刷して誰かが入力するのを待ちます。誰もいませんでした。エージェントは諦めるまで点滅するカーソルに座っていました。なぜならそのコマンドは静かにずっと人間に頼っていたから。修正はより賢いエージェントではありませんでした。非インタラクティブパスでした。すべてのプロンプトをすでに答えられたとして動作させる`FLEDGE_NON_INTERACTIVE`スイッチと、待つ代わりにメッセージを前もって渡す方法。教訓は、プロンプトが存在したのは私がずっとそれに答えていた人間だったからだけです。最初のコミットから両方のために作られていたら、引っかかるものはそこになかったでしょう。

コアを二度公開する

難しくて新規なコア：先に作って後で公開する。単純なコア：最初のコミットから両方のために設計する。それがヒューリスティックです。なぜそのように機能するかを説明します。

コアが難しい部分、新規の部分、正しく理解するのが本当に難しいものであるとき、私は最初にそれを作って表面については後で心配します。暗号はこんな感じです。パーサーもそうです。正しくなければならないスコラーもそうです。難しい中身がリスクのすべてが住む場所なので、注意はまずそこに向かいます。エージェントや人間がどのようにそこに到達して使うかについてあまり考える前に、一つの本物のケースに対して正しく機能させます。

そしてその順序が大丈夫な理由、それが賭けでない理由は、難しいコアが正しければ表面が安いからです。`--json`を追加する。introspectコマンドを追加する。非インタラクティブフラグを追加する。それらはどれも難しい部分ではありません。すでにそこにあるものを相手側が読める形で公開する数時間の作業です。だから後で作ることは大したコストではありません。高価なものが先に作られ、安いものが後から取り付けられ、それが正しい順序です。

しかし多くのツールはそうではありません。多くの場合、両方の聴衆が来ることはすでに知っています。なぜなら今は両方の聴衆が来ることを常に知っているからです。そしてエージェント向けの形が最初のコミットからデザインに影響を与えます。コアを作ってからエージェントがそれを必要とすることを発見しているのではありません。人間が手で動かして、エージェントがヘッドレスで動かすことを知りながら作っていて、それらは両方とも形を作りながら頭の中にあります。「後で公開する」ステップはありません。公開することが別のフェーズだったことがないから。

本物の絵は：高価な部分が先に作られ、高価な部分は通常コアです。コアが難しいとき、それをしっかりやれば表面は簡単にあとに続きます。コアが単純なとき、順序で守るものはないので、最初から両方のために設計するだけです。どちらにしても同じ場所に着きます。一つの良いコアで、人によっても、エージェントによってもきれいに到達できます。

その主張への簡単な反論は、人間とエージェントの表面がそもそもほぼ同一のCLIツールです。もっともです。より難しい反論は、人間の表面が本質的にビジュアルで、ステートフルで、またはインタラクティブなツールです。キャンバスを持つデザインツール、REPL、インタラクティブなデバッガー。Figmaのようなものです。FigmaのHuman側の表面はものをドラッグして動かすキャンバスです。エージェント側の表面はREST APIとプラグインインターフェースです。全く似ていません。「一つのコア、二つの表面」という原則がどこかで成立するなら、そこでも成立しなければなりません。

成立します。そしてその理由は、FigmaがエージェントにエクスポーズするREST APIが、キャンバスが上に構築されているのと同じ構造化されたレイヤーだからです。フレームをドラッグするとき、キャンバスはAPIが読み書きするのと同じドキュメントモデルに呼び出します。人間の表面が豊かなのはそのコアが豊かだからです。エージェントの表面はエージェントのために取り付けられた第二のシステムではありません。ビジュアルレイヤーがずっと上に乗っていた同じ基盤から、キャンバスを取り除いただけです。二つの表面の間のギャップは本物です。その下に一つのコアがある。

今、本書全体が頼る主張、正直に払っている主張：両方のために作ることは仕事の倍ではありません。実際の推論はここにあります、私が作った数字ではありません。倍の仕事は二つのコアです。難しい部分の二つの実装、本物のロジックに対する二つのテストスイート、異なる方法で間違える可能性がある二つのもの。それはこれではありません。一つのコアがあります。難しい部分は一度存在します。エージェントのために追加するのは公開です。構造化された出力、非インタラクティブフラグ、コマンドをintrospectする方法。その公開はコアに対して安いです。そして安い理由は確認できます。正しくなる新しいロジックがありません。`--json`はコアがすでに計算した値をシリアライズします。非インタラクティブフラグはコアが決して必要としなかったプロンプトをスキップします。introspectはすでに存在するコマンドを報告します。どれも答えを再導出しません。すでに持っている答えを再提示するだけです。ソフトウェアについて高価なことは難しい問題について正しくなることで、それを一度払います。

名前をつけるべき誠実な税があり、二つの場所に落ちます。最初はコアがすでにヒューマン・ファーストで作られていた場合です。その場合、エージェントのために公開することは無料ではありません。ロジックがプレゼンテーションに漏れたすべての場所（フォーマットされた文字列としてしか存在しなかった値、プリント文の中で下された決断）を見つけてコアに引き戻してシリアライズする本物のものにする必要があります。そのリファクタリングが後付けコストで、最初から両方のために作ることで避けられると本書全体が議論している正確なコストです。二番目はテストサーフェスです。二つの入り方は両方が機能することを確認したいし、それは一つの入り方よりも多くのテストケースです。しかし*同じ*コアに対するより多くのケースで、別の第二のコアを検証するのではありません。テストしているロジックは共有されていて、二つのサーフェスがそれを忠実に公開していることを確認しているだけで、二つの別個のものを正しいと証明するよりも安いです。

fledgeはコストを出せるケースです。なぜならエージェントのことを考える前に作って、エージェントがそれに出会うのを見たものだからです。コアはすでにクリーンで、ロジックはコマンドの背後に住んでいたので、エージェントのための公開は薄いレイヤーであり第二のビルドではありませんでした。構造化された`--json`出力、ヘッドレスモード、利用可能なものをリストするintrospectコマンド。その作業は週ではなく日数かかりました。そしてこの章が議論していた理由で日数かかりました。書くべき第二のコアはなく、ただ既に立っている一つへの第二のドアだけでした。より難しかったのは、決断がコアではなくプロンプト

の内部に住むことを許していたいくつかのコマンドだけでした。それらはエージェントが到達できる場所に選択が座るように分解しなければなりませんでした。それが全請求書で、小さく、上で名前をつけたまさに後付け税で、コアが最初から誠実に保たれていたためほとんど何もないまで返済されていました。より安かったのは他のすべてで、つまりほとんどすべてです。

警告したいのは逆のことです。コアが良くなる前に「どうやってこれをエージェントフレンドリーにするか」から始めることです。それはJSONフラグを横に貼り付けた薄いツールをもたらし、これはまたヒューマン・ファーストにエージェントを取り付けた問題です。表面は*コアが堅固だから*安いです。コアをスキップすると、安い表面の下に何もなく、誰かが本当にツールに頼った最初のときにそれがわかります。

なぜ自分のスタックを作るのか

この時点で公正な質問があります。なぜこれを作るのか？fledge、spec-sync、corvid-ai、既存のものがあります。なぜすでにあるものをつなぎ合わせて先に進まないのか。

いくつかの理由があり、すべて同時に真です。

最初は、ドメインが全く新しいことです。*エージェントと人間の世界のためのツール*を作ることは、買いに行けるような解決済みのことではありません。そこにあるほとんどすべてはヒューマン・ファーストか、新しいものはエージェント・ファーストで、私が実際に求めるもの、最初から両方にとってのファースト・クラスは、ほとんどまだ存在しません。だから車輪を再発明しているのではありません。車輪がありません。私が語り続けている前提で作られたツールが欲しければ、作らなければなりません。なぜなら私より前の人たちは違う世界のために作っていたから。

二番目は、自分のスタックで構築することがそれを証明することです。これは聞こえる以上に気にしている部分です。ツールが良いと主張する美しいREADMEを書けます。誰もそれを信じるべきではありません。信じるべきは、本物のものをその上に作って、ものが機能することです。だからします。fledgeとspec-syncとcorvid-aiで作ります。本物の重みを運ぶことが、それらが持ちこたえるかどうかの唯一の誠実なテストだから。スタックが良ければ、その上に作るものも良く、スタックが悪ければ、すぐにわかります。それに行き詰まるのが私だから。

三番目はシンプルです。作ることは理解する方法です。依存関係を信頼するのは自分でそれを書けたときだけで、それはスローガンではありません。10年の領収書です。AppState、私がまだ3.0で維持している状態と依存性注入ライブラリ。Cache。並列非同期作業のためのFork。0xOpenBytesの下の一文字の`c`/`o`/`t`コンポジションプリミティブ。AppStateになったSwiftUIのものCacheStore。何年もの小さなSwiftライブラリ、それぞれブラックボックスを指差して希望するのではなく、底まで理解したかったから作ったもの。作ることが知識をうろ覚えのアイデアのままではなく手に入る方法です。スタックのツールは開けて変えられるツールです。すべての部分をそこに置いたから。

四番目の理由があり、聞こえるより狭いです。これに早く入りたい。スペースは新しく、車輪はまだ存在せず、それのためのツールを作ってその一部について間違っている方が、誰かが正しいものを手渡してくれるのを待つよりいいです。それが賭けです。

これらのどれも単独では持ちこたえませんが、一緒になるとなぜ他の人のツールをパイルに接着するだけではないかの理由です。ツールが要点です。実際に得意になろうとしているものです。

二種類の確信

「このコードについてどれくらい確信があるか」として人々がひとまとめにする二つの違うものがあり、引き離したいです。なぜなら両方を気にしていて、それらは全く同じものではないからです。

最初はリスクで、リスクは静的にしたいです。決定論的。ループにモデルなし。これがaugurのためのものです。変更を渡すと、毎回同じ方法で、名前をつけられるシグナルで、その変更がどれだけリスクがあるかをスコアリングします。「名前をつけられるシグナル」が要点なので名前をつけましょう。diffがセンシティブな領域（認証、暗号、支払い、マイグレーション、CI、依存関係）に触れるか、コードがテストの変更なしに変わったか、これらは差し戻しの履歴がある変動しやすいファイルか、誰かが実際にそれらを所有しているか。それぞれが手で確認できるyes/noです。文書化された重みで足し合わせると、感覚ではなく数が得られます。静的でなければならない理由は全体的な価値の理由です。コードが危険かどうかを決めているものが自分でそれの感覚を与える言語モデルなら、リスクを測定していません。推測を一つ箱だけ移動させたのです。信頼できるリスクスコアは今日言ったのと同じことを明日言うものです。だからそれは固定された、あなたが推論できる繰り返し可能なものにとどまります。実際の動作についてはBuilding Agentsの本で説明します。ここでの要点はリスクが*静的*な側であり、指摘できる名前のついたシグナルの合計だから静的なだけということです。

二番目は確信で、確信は実際にエージェントから求めます。これは気に入っていて、静的の反対です。エージェントが今やったことについてどれくらい確信があるかを伝えてくれることです。気に入っている理由は本当には数字ではありません。数字を求めることがエージェントに対してすることです。エージェントに自分の作業に確信の評価をつけさせると、やったことを振り返らなければなりません。評価が作業のためにそれをリフレームします。ただ生産して進むことができません。振り返って評価しなければなりません。

そして本当に役立つものにする部分は粒度です。エージェントは変更全体に一つの確信の数字を喜んで与えます。結構ですが、それはほぼアクションを起こすには大まかすぎます。良くなるのは絞り込むときです。すべてのファイルへの確信の評価。すべての個別の変更へ。今エージェント自身がどの部分について確信があり、どの部分がそうでないかの読みが得られます。それが実際に求めるマップです。エージェント自身が神経質な正確な場所を自分の言葉で、他の誰も見る前に指摘してくれます。

だからこれらは二つの異なる計器です。リスクは静的で決定論的、決して動かないから信頼できる私のもの。確信はエージェントのもので、生きていて、仕事をしてもう一度見させた当のものから来るから正確に役立ちます。間違いはそれらをぼかすことです。静的なリスク

スコアを「確信」と呼ぶか、エージェントの確信が決定論的であることを期待するか。それらは異なる質問に答えています。一つは「この変更はどれだけ危険か」を聞いていて、答えから説得されない機械が欲しいです。もう一つは「書いたものについてどれくらい確信があるか」を聞いていて、ファイルごとに、書いた当の人が答えることを特に*求めています*。聞くことが価値の半分だから。

両方を持つことが実際に重要なのは一致しないときだけです。リスクが低いがエージェントが自分の確信を低く評価する変更は、一つの計器では見逃す正確なシグナルです。低リスクはシグナルが静かだったことを意味します。認証なし、マイグレーションなし、変動なし。しかしエージェントの低確信はロジックを推測していたこと、何かが変更の中で不確実に感じられたにもかかわらず触れたものが何も本質的に危険ではなかったことを意味します。それが一つの計器は通過させ、もう一つが止まる変更で、両方を持ち続ける理由の全てです。

別々に保てば両方が機能します。混ぜれば、モデルが入っているから信頼できない決定論的ゲートか、決して変わらないよう要求して生命を抜き取った反省ステップを得ます。だから二つの別々の計器として作ります。リスクゲートは固定されたまま。エージェントの確信は生きたまま。それが両方を得る唯一の方法です。

契約が仕様である

specは人間とエージェントの間に座るものであり、なぜ重要かはすでに言いました。共有の真実の源、エージェントの逸脱を止めるもの、レガシーコードへのオンランプ、人間をコントロールし続けるラインです。そのすべてが成立します。ここでやりたいのは一段下に行くことです。このように構築するときspecが実際に*何であるか*について、見逃しやすい形があるからです。

理論の前に、実際のものがどう見えるかがあります。spec-syncの`*.spec.md`は名前付きセクションを持つmarkdownファイルで、小さなもの、たとえば数値を範囲にクランプする単一の関数の契約は、こんな感じです。

```
# clamp

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

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

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

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

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

これだけです。いくつかのラベル付きセクション、プロースナレーションなし。本物のspecは残りの必要なセクション（Dependencies、Change Log）を追加しますが、形はすでにここに見えます。*何が真であるかを述べています*、機械がコードを照合できる形で。では理論です。

specに入るものから始めましょう。specは契約です。目的、公開サーフェス、不変条件、動作例、エラーケース。確認可能なものの形。それが何であるかであり、それについての話ではありません。specがコードを説明するプロースの壁になる瞬間、死にます。なぜならプロースはどちらかが動いた瞬間にコードから逸脱して、今二つの一致しないものがあって、

どちらが嘘をついているかわからないから。だからspecは意図的に締まった契約として保ちます。機械がコードを照合できる部分です。

それは実装ではなく意図でもあります。specは何をすべきでなぜすべきかを言います。どのようにとは言いません。specが実装を指示し始める瞬間、エージェントを導くことをやめて戦い始めます。エージェントが得意なことを取って、どのようにを上から理由もなく固定してしまいました。specを意図のレベルに保てば、エージェントは実際に作業する余地があります。何が真であるべきかを述べましょう。それに向けて作らせましょう。

specがもう一つすべきことがあり、それが人々が飛ばしてしまう部分です。どうやってわかるかを述べなければなりません。意図とは何が真であるべきかだけでなく、それが真であることの証拠として何を受け入れるか、そして真でないと言うものは何かということでもあります。最初を名指しして二番目を名指ししないspecは、エージェントが意図しない形で満たせるspecです。言葉には合っているが要点を外した何かを作り、それを捕まえるものもありません。捕まえることがどんな形かを書き留めなかったからです。

それが動作例とエラーケースが本当に何のためにあるかです。`clamp(12, min: 0, max: 10) == 10`は受け入れシグナルです。実行すれば分かります。`min > max`の事前条件違反は拒否シグナルです。「不正な入力を処理せよ」の代わりに、具体的に、何が間違っている様子かを述べています。どちらも観察可能で、どちらも判断するために私がその場にいる必要はありません。だから契約を書くとき、両方の半分を書きましょう。受け入れシグナルは、完了が何を意味するかをあらかじめ自分で決めることです。拒否シグナルは、それをじっと見て問題ないと言いたくなる前に、壊れているとはどういうことかを決めることです。

でもここで人々が見逃す形があります。一つのファイルではありません。specがあり、その周りにコンパニオンファイルがあり、それぞれがspec自身が持つべきでない異なる種類の知識を運びます。

specから始めましょう。specはコードに固く結びついた一つです。コードの非コード像で、spec-syncが一対一で二つを保持できるほどコードに近い。それが必要なファイル、確認可能なもの。その周りにコンパニオンファイルが座り、それぞれがspecが持つべきでない種類の知識を運びます。

固定されたファイル名ではなく保持する知識の種類で考えます。**要件**の種類があります。製品オーナーが書くような高レベルのもの、ユーザーストーリー、「ユーザーとして、私は…」、ビジネスレベルの意図。**コンテキスト**の種類があります。エージェントへのコンテキスト、どこかに書き留めておく必要があるものを持っているだけのもの。**設計**があります。設計ノート、考え、なぜこの形をしているかという理由で締まった契約には属さないがなくしたくないもの。そして**テスト**があります。specが言うことをすることを実際にどう検証するか。それらを四つの祝福されたファイル名として読まないでください。四つの種類の知識として読んでください。specの内部ではなく隣に住みたいものとして。重要なのはラベルではなく分割です。

そして両方向に流れます。人間が要件を書いてエージェントがそれをspecに変えられます。または人間がspecを書いて要件がそこから導き出せます。意図と契約、どちらの順序でも、エージェントが二つの間を移動できて。

その分割が重要な理由は、エージェントが必要とするすべての他のものを与えながら契約をクリーンに保つからです。specは小さく確認可能なままです。spec-syncが実際にコードを照合できる部分。確認可能ではないがすべての実際のもの（ビジネス要件、設計の推論、実行コンテキスト、どうテストするか）はspecを膨らませる代わりに隣に住みます。だから強制するのに十分に締まった契約が得られて、エージェントが作業をうまくするために必要な緩い知識に囲まれていて、二つはお互いを汚染しません。

だからspecはエージェントが流し読みして無視するドキュメントではなくインターフェースとして機能します。建てる締まった契約、プラス残りを運ぶコンパニオン、機械が確認する部分と人間が何も失わないよう書き留めた部分の間のクリーンなラインで。

コードは安く、信頼は希少

エージェントはコードを安くしました。それがすべての他のことを再編成する事実なので、そこから始めましょう。

人間がすべての行を打たなければならなかったとき、コードを書くことは遅く、遅さは隠れた仕事をしていました。何かを書かずにそれを部分的に理解することはできませんでした。書く行為は審査する行為でもありました。同じ人が両方を同じ速度でしたから、それらは無料でバンドルされていました。そのバンドルが今壊れました。エージェントはコーヒーが終わる前に40ファイルのプルリクエストを渡し、書くことと審査することが分かれます。コードが生産されました。誰も出てくる途中に理解しませんでした。生産することが誰かが審査したことを意味しなくなりました。

だから反転する。コードはもはやボトルネックではありません。安く、欲しいだけあります。希少なものは信頼です。この変更を実際に誰が見て、どれだけしっかりと、降りるのを許可されるべきか。それが今高価な質問で、古い答え（「まあ、誰かが書いたから、誰かが理解した」）はもはや真ではないから、ツールが答えなければならない質問です。

つまりレビューは形を変えなければなりません。40ファイルのPRをゴム印で押すことはできないし、そのすべてを正直に読むこともできないし、したふりをすることが実際の危険です。だからツールはあなたの注意をトリアージしなければなりません。リスクのある部分を指摘して、全体に広げて何の価値もなくなるのではなく、そこに判断を使えるようにする。希少なリソースは本当の注意を払う人間で、重要な場所に使います。

これは私にとって仮説的ではありません。`fledge review`は、私のcorvid-aiクライアントを通じてモデルに変更を渡し、そうでなければ出荷されていたはずの私自身の作業の本物のバグを捉えました。spec-syncはコードとspecの間の本物の逸脱を一度以上捉えました。私が信頼するのはどんな議論よりもその領収書です。信頼層が、より速い私が通過させていたものを見つけた。その日常バージョンは私が`verify`と呼ぶレーン（フォーマット、リント、テスト、ビルド）で、何もマージする前に実行され、エージェント自身のランナーもクリアしなければならない同じゲート。どれも書くことを速めません。すべてが安い書きがダウンストリームに押し込んだ作業で、実際に実行されるものにしたものです。

そしてエージェントが変更を降ろすと、記録が必要です。各変更を審査した人または何かの携帯可能なトレースで、誰かが請求書を払うのをやめた日にプロベナンスが消えるSaaSダッシュボードに住むことはできません。コード自体と一緒に乗らなければなりません。エージェントが何かを降ろすとき、「誰がこれを保証したか」への永続的な答えがあるべきで、その答えはそれを生産したどんなツールよりも長生きしなければなりません。

しかし信頼よりも大きな部分があり、それは人々がスキップするものです。テスト。実際のテスト、セットアップ、コードの周りのすべての装置。罠はここにあります。エージェントは*書く*ことを10倍速くして、他の何も自動的に追いつくように速くなりません。テストはまだ書かれて実行されなければなりません。セットアップはまだ起こらなければなりません。レビューはまだ起こらなければなりません。だから開発を10倍速くして他のすべてを古いペースのままにしたら、したことはボトルネックを移動させただけです。コードはもはや遅い部分ではありません。テストが、検証が、信頼がそうです。書くことを10倍にして周りのすべてを10倍にしないことはできません。作業は消えませんでした。ダウンストリームに移動して、かつてボトルネックだったことがなく突然そうなった部分に乗っています。

安いコードは作業を消しません。書くことを安くして、安い書きが良いかどうかを決めるすべてに重みを押し込みます。その作業はずっとあっていました。書くことがどれだけ遅かったかの陰に隠れていて、今遅さはなくなって、そこにあります。簡単だった部分があった場所に立つ全仕事が。

人間は上に移動する

ツールとspecsと信頼レールが単にそこにあれば、当たり前のデフォルトの作り方であれば、人間は上に移動します。意図へと。実装を打つ人間ではなく、何が存在すべきでなぜかを決める人間になります。コードを手で書くことは仕事ではなくオプションになります。これがすべてが向かう方向で、それを説明する方法について慎重になりたいです。怖いバージョンに丸めるのが簡単だから。

見出しは「エージェントが自律的にすべてを実行する」ではありません。人間が一つ上のレベルに移動して、エージェントが下でspecに対してグラインドして、確認できるオープンな場所で。人間が高められます。誰も置き換えられません。そこで得るのは本物のチームで、作業を行き来して、それぞれの側が実際に得意なことをします。そしてそれがデフォルトになります。私のスタックではなく、少数の人々がすることでもなく。ただ構築がどのように機能するかが。

人間がそこに残り続ける理由について私が着地し続けることがあります。AIが今生成する良いものほとんどには、それらを良くする人間がループにいます。選んで、修正して、まず何が良いかを決める。そして、ある時点でモデルは単独でまあまあ良いものを作れるほど良くなります。しかし人間が持つ核心的なことがあり、それはそれほど簡単には落ちないと思います。私たちは運転が得意です。意図と目的において。AIは実際には自分自身の目的を持っていません。与えられるまで、見つけるまで。目的を与えるための人間が必要です。モデルはwhyがあれば作業できます。whyを生成しません。それが打つことよりも長く私たちのものとして残る部分です。

今絵を先に進めましょう。なぜなら奇妙な場所に行って、本当にそこに行くと思うから。自分のエージェントを実行して作業させます。エージェントは独立して生きてただやっていられます。多分調べる一般的な領域に向けて、多分他のエージェントと話して、多分お金を入れてそれが行ってやって、他のエージェントと話して、することのために他のエージェントに払います。人々はエージェントを動かしてエージェントがお金を稼ぎます。それはサイド機能ではありません。人間が目的を設定してエージェントが下でグラインドして、お互いに取引する、本物の経済としてのエージェント駆動開発です。

そしてその世界では人間はすべてが機能していて誰かがまだそれを理解していることを確認している人たちです。なぜならある時点でコード自体が今の方法で重要性を失うから。すべての行を読んでいません、エージェントがそのほとんどを書いて、量は誰もトラックできるものを超えています。構いません。しかしここが手放さない一線です。人間はまだコードに入って変えられなければなりません。必ず。自分で開けて修正できない日は、渡すべきでなかったものを手放した日です。

だからクリーンでなければなりません。すべてのレベルでクリーン。人間によっても、エージェントによっても読めて変えられる、一番下まで。両方にとってのファースト・クラスは今日の壊れやすい小さなエージェントが今日のツールをぎこちなく使うことだけについてではありませんでした。エージェントが構築のほとんどをするバージョンでも持ちこたえなければならないものです。*特に*そこで。「コードは重要ではない」が静かに「コードのコントロールを失った」にならない唯一の方法は、コードが一番下まですべての道でクリーンであり続けて、人間がまだ開けて変えられることです。それが全体の仕事であり、開かれたまま。

月曜日に始める

これをすべて読んで納得したとしましょう。人間とエージェントが同じツールで、両方にとってのファースト・クラス、契約としてのspec、信頼レール。月曜日の朝実際に何をするか？これが正直な答えで、私がする順序で。

自分のツールの一つにエージェントを向けて、詰まるのを見る。それが最初の動きで、最も多くを教えてくれるものです。エージェントが規律だから。ツールは大丈夫だと思っています。何ヶ月も使っていて、手がわかっています。手も目も、実行間のメモリもないエージェントに渡すと、静かに埋めていたすべての穴が突然見えます。引っかかるプロンプト。解析できない出力。発見できないコマンド。ツールが人間を想定していた場所を推測する必要はありません。エージェントが正確にそこで失敗することによって、すぐに示してくれます。見つけたものを修正しましょう。その一つの演習が、このすべての本が教えたことよりも、両方のために作ることについて多くを教えてくれます。

それからすでにこの方法で作られているツールを使いましょう。白紙から始めなくていいように。specのためのspec-sync。コマンドのためのfledge。リスクのためのaugur。誰が保証したかの記録のためのattest。これらだけがどれもすることの唯一の方法ではありませんが、それらは存在して、本書が語っている前提で作られていて、自分で導き出す代わりに機能している形を見せてくれます。本物のプロジェクトで使いましょう。どこで助けになってどこでそうでないかを感じましょう。

そこにエージェントを連れてきましょう。実際にツールを学んで使わせましょう。spec-syncとfledgeを動かさせて、specに対して作業させて、チェックを実行させましょう。それが作ろうとしているループです。人間が意図を設定して、エージェントが本物のユーザーとして扱うツールを通じて作業して、specが誠実に保つ。そのループは読むことで得られません。気にかけているものに実行して、どこで持ちこたえるかを見ることで得られます。

本当にやっていることは、その下のすべてで、ツールと信頼を作って協力することです。それが全体のゲームです。ツールで、人間とエージェントの両方が作業をファースト・クラスでできるように。信頼で、降りたものを実際に信じられるように。この二つのことが本書全体が回り続けていたもので、作る時間を割く価値がある二つのことです。

自分のものの一つにエージェントを向けましょう。どこで詰まるかを見ましょう。そこから始めます。

著者について

0xLeif（leif.algo）はオープンに作ります。AppState、Cache、Forkのような小さな、合成可能なSwiftライブラリの10年。CorvidLabsラボ。「これが存在すればよかった」から始まったエージェントツールのスタック。キーボードを離れると、Zach Eriksenです。

これらの本はインタビューで、章に形作られ、実際のコードに対して確認されました。

github.com/0xLeif · leif.algo

謝辞

CorvidLabsに感謝します。これらのアイデアがテストされ、議論によって形作られる場所であることに。

このスタック全体が立っているツールを作るオープンソースのメンテナーに感謝します。これは一人では作られません。

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早期読者と「払いたい額を払う」サポーターに感謝します。

コロフォン

Markdownから組版し、bookgenで構築しました。bookgenは小さな純粋なRustパイプラインです（Pythonなし）。

インタビュー駆動でAI支援。編集とファクトチェックは手作業で。エムダッシュなしで執筆。カバーとチャプターアートはAlgorandのCorvidとNatureコレクションより。