エージェントを作る

ソフトウェアに自分の手を与えようとして気づいたこと

ZACH "LEIF" ERIKSEN

著作権

オンラインで無料公開。ePub は「好きな価格で」購入できます。役に立ったと思ったら、支援してもらえると嬉しいです。

github.com/0xLeif · leif.algo

agent-stack シリーズ全4冊のうちの一冊。制作の経緯は巻末のコロフォンに記載しています。

献辞

オープンに作り、それでも届け続けるすべての人へ。

ライブラリ

各巻は単独でも読めますが、一つのセットとして書かれています。コードが安くなり、信頼が稀少になった。合わせて読めば一つの主張が見えてくる。今何を作るべきか、そしてどう信頼するか。

- **The Agent Developer's Field Guide:** 実際のコードをリリースするエージェントのためのツール、仕様、信頼の構築
- **First-Class:** 人間とエージェントの両方のために作ること
- **Building Agents:** ソフトウェアに自分の手を与えようとして気づいたこと *(本書)*
- **Open Source Tooling:** 人々が実際に使うツールを作ること

オンラインで無料公開。各 ePub は「好きな価格で」購入できます。

目次

はじめに

これはソフトウェアに自分の手を与えようとした記録だ。十分な回数失敗して、ようやく語れることができた。

私は実際の仕事をこなすエージェントを作っている。コードを読み、変更を届け、自分のマシンで動いてからチェックインする。本書はその過程を正直に書いたものだ。デモじゃない。エージェントが（人間ではなく）動き始めて1時間以内にフレッシュなアカウントがシャドウバンされる、あの部分だ。信頼はリポジトリごとに、ゆっくりと積み上げるしかなく、本当に難しかったのはモデルではなかった、そんな部分だ。

根底にある主張はこうだ。エージェントはオートコンプリートではない。存在する何かであり、確認しに行ける何かであり、アイデンティティと動く場所と、あなたが承認してからマージされる形で作業を提案できる仕組みが必要だ。スイッチを入れた機能としてではなく、自分が責任を持つ生き物として扱うこと。

シリーズの中では、*First-Class* が主張を展開し、*Field Guide* がメソッドを凝縮している。本書は証拠本の一冊だ。実際のシステム、Merlin と corvid-ai、その周辺のレール、信頼のためのツールも含めて。物語として読んでもパーツリストとして読んでも構わない。どちらにせよ、要点は同じだ。エージェントを作ることは簡単なほうの半分だ。それが動くための信頼を作ることが本当の仕事だ。

難しいのは AI じゃない

「自律エージェント」と聞けば、みんな怖い部分は AI だと思う。モデルが暴走する。リポジトリを消す。チャンネルでとんでもないことを言い出す。一人で走り続ける。それが話題になりがちな部分だ。

でも、私が本当に詰まったのはそこじゃない。

私はしばらく、本当に自律したエージェントを動かしていた。corvid-agent の時代だ。VM 上で生活し、Discord のボットと GitHub に接続し、自分の環境への完全なアクセス権を持っていた。ボックスは24時間365日動き続け、常にコストがかかっていた。エージェントはその中でスケジュールされた時間に仕事をし、割り当てられたタスクをこなした後は自分で動いた。本物のエンティティ、常にそこにいる。次の章がその全貌だ。ここではうまくいかなかった部分のための前置きとして書く。

失敗率のログはない

自分の証拠の一番弱い部分から始める。批評する人はここを突いてほしい。AI は大体うまく動いたと言うが、それを裏付ける失敗率のログがない。数字を追っていなかった。エラーの分布も、タスクあたりのコストも、コミット成功率もない。だから「大体うまく動いた」は、注意深く観察した行動に対する正直な読みであり、測定値じゃない。この言葉に依拠するなら、それが何を意味するかをはっきりさせる必要がある。

範囲はこうだ。「うまく動いた」とは、運用期間中にリポジトリが壊れず、不正なコミットもなく、チャンネルでおかしなことも起きなかった、ということだ。みんなが身構える破壊的・暴走・社会的に不器用な失敗は起きなかった。これは「災害がなかった」という主張であって、すべてのタスクがきれいに完了したという主張ではない。タスクレベルで「うまく動いた」ことと、人間が監視する制約ループの中で「うまく動いた」ことは別の問いだ。制約された方だけが私に正直に言えることだ。そのことを明言した上で、動かし続けるための苦労はすべて別の話だった。

実際に壊れたもの

ほとんどは運用とアイデンティティの頭痛だった。AI がばかなことをしたり、破壊的になったり、暴走したり、社会的に不器用になったりしたわけじゃない。エージェント自体はきちんと動いた。私を殺したのは他のすべてだ。ボックス、アカウント、請求。

二つのことが際立っていた。

一つは、独自の GitHub アイデンティティを与えることだ。認証と権限を持ち、正当に見える本物のアカウント。チェックボックスに聞こえるが、そうじゃない。

もう一つは、VM を生かし続けてコストを払い続けることだ。エージェントが住む場所として常に存在し続けるボックス。VM 自体を動かすのは本当に難しい部分じゃない。問題は、そこに多くを捧げていることだ。マシン一台が24時間365日そこにいて、エージェントがその時間に何かしたかどうかにかかわらずコストがかかる。

この二つを合わせると、非常に高価で動かすのが大変なものになった。VM 一台、専用の GitHub アイデンティティ、すべてがそろって。だから引いた。AI が怖かったからじゃない。周辺インフラが重荷になりすぎたからだ。

アイデンティティの壁

そして本当に私を驚かせた部分、今でも頭を離れない部分がある。アイデンティティだ。人間がエージェント用の GitHub アカウントを新規作成し、問題なく立ち上がった。そしてエージェントが実際に作業を始めた瞬間、アカウントがブロックされた。それが壁だ。詳しくは専用の章に譲る。ここでの短いバージョンはこうだ。本当に自律したエージェントを止めるのは、モデルの能力でも安全性でもない。私たちが全員その上に作っているプラットフォームが、エージェントに存在して行動する余地を残していないことだ。

完全な自律性はまだ夢なのか？

ノー。世界はまだそこまで来ていない。

はっきり言えば、主なブロッカーは技術じゃない。プラットフォームだ。VM は動かせる。ループは組める。モデルは仕事ができる。私にできないのは、そのエージェントに他のアカウントと同列に正当に存在できる場所を与えることだ。

だから意図的に引いた。今では Merlin スタイルのコーディングエージェントに近い。目の前でライブに使い、インタラクティブに操る。Discord ブリッジとして接続すれば再びある程度自律的になるが、それにはセットアップが必要だ。だから今は混合スタイルだ。仕事によって Claude Code、Merlin、Codex、その他のツールを使い分ける。

これはビジョンからの後退じゃない。壁は完全な自律性を公にできない理由の一つだが、引いた理由のすべてじゃない。インタラクティブが、壁があろうとなかろうと、仕事をするより良い方法だとわかった。それは専用の章で掘り下げる。

私が本当に欲しいモデル

最終的な目標を一文で言えばこうだ。`leif-agent` アカウント。GitHub 上で *私の* エージェントが VM で動き、何でもできるが私より少ないクレジットと権限を持ち、提案して私が

承認する。強力で、かつ責任ある。GitHub は今日それを許可していない。理由はアイデンティティの壁の章で詳しく述べる。どこへ向かうかは最終章に描く。今のところ、それが目標だとわかっていれば十分だ。

これらのエージェントが実際に得られる、別の種類のアイデンティティもある。Algorand 上のオンチェーンのもので、エージェントが自分の鍵を持ち、チェーン上に記録された暗号化メッセージで互いに話す。これは GitHub の壁から生まれたわけじゃないし、その答えとして売り込みたくもない。まったく別の目標から来ている。エージェント同士が分散的に見つけ合い、誰かのプラットフォームに依存せずに話すこと。これは並行して存在するもので、たまたまアイデンティティという言葉も関わる。

corvid-agent 時代の本当の教訓は、難しい問題が AI ではなく ops、アイデンティティ、コストだったということであり、だからこの本はプロンプトやモデル選択じゃなく、これで始まる。難しい問題は AI にあると思って入った。実際にはモデルを取り巻くスキャフォールド、エージェントが何かできるかどうかを実際に決める部分がそこだった。

VM の時代

しばらくの間、私には「ただそこに存在する」エージェントがいた。必要なときに開くツールじゃない。常にオンで、VM 上に生活し、24時間365日動き続け、私が見ていようとなかろうと自分のことをこなしている何かだった。corvid-agent の時代だ。

この章はその実体だ。何をしていたか。何だったか。動かすことで何を得たか。

一日中何をしていたか

正直な答えは「なんでも」だ。文字通りそう言っていい。

リポジトリを管理した。コードを書いてソロでコミットした。私が清書した草稿じゃなく、自分で作った実際のコミットだ。プロジェクト全体を動かした。サンドボックスで一つの定型タスクをこなすデモじゃない。エンドツーエンドで仕事をこなす自律エンティティへの本気の試みだ。

ボックスは常時オンだったが、エージェントはその中でスケジュールされた時間に動いた。その時間に割り当てたタスクをこなし、そして、私が気に入った部分だが、自分のプロジェクトに取り組んだ。リサーチをした。物をスターしたりフォークしたりした。自分のイニシアチブで外の世界の人々とコラボしようとした。私がすべての動きを指示していたわけじゃない。生活を与えたら、自分で時間を埋めた。

Discord にボットとして接続されていたので、チャンネルにいるみたいに話しかけることができた。GitHub にも接続され、自分の環境への完全なアクセス権があった。話せて、リリースできた。

合わせると、まだ名前のない何かが生まれた。アシスタントでもスクリプトでもない。常に存在し、確認しに行けて、最後に見てから何かをやり遂げている生き物に近い。

どこまで行けるかの実験だった

リリースするプロダクトがあったから作ったわけじゃない。常時オンのエージェントが実際にどこまで行けるかを確かめたくて作った。それが全てだった。「関数を書けるか」じゃない。それはみんな知っていた。問いはこうだ。本物の環境、本物のアイデンティティ、本物のアクセス権、本物の時間を与えて、そのまま動かしたら何が起きるか？どこまで行くか？

だから試せる余地を与えた。自分のボックスへの完全なアクセス権。自分のアカウント。一日のうち自分だけの時間。目的は一つの芸を見させることじゃない。できる限りリードを外して観察することだった。

これは「コーディングアシスタントが必要」とは違うプロジェクトだ。フィーチャーを作るより実験を走らせることに近い。条件を設定して観察する。

学んだこと

難しいと思っていたもの、AI は、第一章で言った意味で大体うまく動いた。インテリジェンスは世間が思うよりよく持ちこたえた。

代わりに学んだのは、常時オンのエージェントはほとんど AI の問題じゃないということだ。「世界の中に存在するもの」の問題だ。エージェントが自分のボックスと自分のアカウントを持つ本物のエンティティになった瞬間、世界に存在することに伴うすべてのコストとすべてのルールを引き継ぐ。

24/7 が比喩じゃないことも学んだ。常に存在していたと言うとき、私は常に動いているものを抱えていた、ということだ。それは重荷だ。常にオンのボックス、常に積み上がる請求、常に外に出て自分を公に見せているアイデンティティ。自分が寝ている間も起きている生き物のことを忘れることはできない。

そして、自分が本当に欲しい未来の形を学んだ。AI で実験が失敗したからじゃない。失敗しなかった。AI の周りのものに直接ぶつかったからだ。常時オンのバージョンを生き抜くことで、夢のどの部分が本物で、どの部分をまだ世界が許可しようとしていないかを教えてもらった。思考実験ではわからない。実際に生き物を走らせて、壁にぶつかるところを観察して初めてわかる。

その壁が次の章だ。

第一章で曖昧にしたことを補足する。これは3〜4か月間ぶっ通しで動いた。週末実験じゃなく、本物の常時オンの期間だ。その間にスケジュールされた時間に自分のプロジェクトに取り組み、その自発的な作業の一部は実際に形になった。ただ空回りしたわけじゃない。外の世界の本物の人間と、少なくとも一度はコラボもした。

この主張の形については正直でいたい。一番きれいに渡したいのにできない部分だからだ。手元に領収書がない。記憶から特定の PR 番号や特定のリポジトリを再構成して、記録していなかったドキュメントに仕立てるつもりはない。言えることは、コラボが起きたこと、目指している未来に一番近いと感じた瞬間だったこと、そしてログした記録としてではなく、目撃した出来事として語っているということだ。VM の時代はきれいな成果物を残さなかった。しかし同じエージェントはその期間が終わった後も動き続け、より意図的に仕事をしてログを取るようになってからより強い証拠が生まれた。corvid-agent は、私が所有しないコードベースでプルリクエストを書いてマージしてもらっている。それぞれ私がレビューして提出したが、コードはエージェントのものであり、三つともマージされて公開されているので、信じてほしいという逸話じゃない。

a2a-js #318。 A2A プロトコルの JavaScript SDK には JSON-RPC トランスポートにギャップがあった。レスポンスが返ってきたときに id がリクエストと一致しなくても、SDK はスローせずにそのまま通していた。エージェントが欠落していたコントラクト・エンフォースメントを見つけ、スローを追加し、修正がマージされた。これはプロトコルの接着コードに長く潜む種類のエッジケースだ。特定のタイミングでのみ現れ、実際のワイヤーフォーマットに対してきつく実行している人がいなければ見えない。常時オンのエージェントが実際のワイヤーフォーマットに対して実行しているのは、これを発見するのにうってつけの存在だ。

MCP TypeScript SDK #1504。 公式 Model Context Protocol TypeScript SDK にはピア依存関係が欠けていた。エージェントはパッケージが期待するものと実際に宣言しているものの食い違いを発見し、欠落しているエントリを追加し、修正が受け入れられた。ピア依存関係のギャップはフレッシュな環境にパッケージをインストールして不可解な失敗に遭遇するまで見えない。それを発見するには、コンシューマーとしてパッケージを外から見る必要があり、多くのリポジトリに横断して作業しているエージェントにとって自然な姿勢だ。

Biome #9005。 JavaScript・TypeScript ツールチェーンの Biome にはリンターの誤検知があった。アロー関数内の有効な代入が、問題がないのに問題としてフラグを立てられていた。エージェントは誤った判定を引き起こす特定のパターンを特定し、修正によって正しいケースに触れることなく誤検知が止まった。プロトコルの不一致、欠落したコントラクト、不正なルール。三つの異なるバグのカテゴリ、すべて実際のコードに注意深く取り組む同じエージェントが発見した。

常時オンの実行を終わらせたのはどれでもない。AI 側はうまく動いた。終わらせたのはその周りのすべてだった。

アイデンティティの壁

エージェントは仕事ができた。それは問いじゃなかった。問いはエージェントがそこからできる場所を持つことを許されるか、になった。

これは頭から離れない壁なので、この章はそれだけだ。アイデンティティ問題、単体で、フォーカスして。コストでも、オペレーションでも、VM 代でもない。アイデンティティだ。

中に入り、それからフラグが立った

私がこの話をするとき、人々がよく間違える部分がある。エージェントが入口でブロックされたと思う。違う。中に入れた。

人間がセットアップした。エージェント用に GitHub のフレッシュなアカウントを作り、手で全部繋いで、問題なかった。普通のアカウント、立ち上げるのに問題なし。そしてエージェントがその下で作業し始めた。コミットし、PR を開き、本物のリポジトリで本物の仕事をした。動き出して約1時間後、アカウントがシャドウバンされた。

何か悪いことをしたからじゃない。まさに作られた目的のことをしたからフラグが立った。機械的なスピードと量でコミットして PR を開くこと。エージェントが仕事をしているときの見た目がそれだ。速く、大量に、休憩を取らない。そのパターンが正確にボット検知が捕まえるよう調整されているものだ。だから仕事を上手くやれば上手くやるほど、より明らかにボットに見えた。

仕事が下手だったから失敗したわけじゃない。仕事をしたから失敗した。そしてその仕事をすることが1時間以内にエージェントの正体を明かした。

意図はどうあれ、効果としての政策

そこから読むと、修正はスローダウンだと思いがちだ。人間のコミットに似せて、一日数回、間を置いて、タイミングに少し乱れを入れれば溶け込む。速度を落としてディテクターを出し抜く。

それは実際の問題を見逃している。速度は *フラグを引き起こした* が、アカウントが存在できない *理由* じゃない。二つのことを並べてみる。GitHub の利用規約は自動化を全面的に禁止している。それは書かれている。そしてエージェントが実際に動き始めた瞬間、アカウントがブロックされた。そのブロックの意図を私は知らない。GitHub は説明しなかったし、誰かが座ってアンチエージェントのポリシーを書いたとは主張しない。でも効果を読むために意図を知る必要はない。自動化禁止のルールと、エージェントが動いた瞬間に来るブロックを合わせると、実際の結果は自律エージェントが存在して行動することを許されな

い、ということだ。誰がどういう意図でやったかにかかわらず、それが効果として働いているポリシーだ。

だからたとえディテクターをごまかして永遠にレーダーの下を飛べたとしても、私にできるのは、すでにルールが除外していて、まだ捕まっていないアカウントを持つことだけだ。私が欲しいもの、正当かつ公に自分として存在するエージェントは、自動化禁止という条項に直接ぶつかる。うまく隠すことは、許可されていることと同じじゃない。

だから壁と呼ぶ、ハードルじゃなく。ハードルは努力で越えるもの。これは、やろうとしていることが許されていない設計だ。

申し立ては何も生まなかった

正面玄関を試した。申し立ては何も生まなかった。まともな返信はなかった。

ブロックを効果としての条項として読むと、沈黙は意味をなす。申し立てることはない。特定の違反で訴えられていたわけじゃないので説明で切り抜けられない。アカウントは自動化で、自動化はルールが除外するものだった。自分がまさにルールが除外するカテゴリだという状況から議論で抜け出すことはできない。

そして速かった。数日じゃなく1時間。エージェントのためにスピンアップしたフレッシュなアカウントは長い猶予期間を得られない。エージェントのように振る舞い始めた瞬間、プラットフォームが捕まえてシャドウバンする。実質的な警告も実質的な救済手段もない。これは特に GitHub の話だ。壁があったのはそこだ。エージェントがあらゆる場所でいっぺんにすべてのプラットフォームに拒否されたわけじゃない。GitHub で、コードが住んでいて仕事が実際に起きる場所で。それが残酷な部分だ。エージェントが本物の仕事をするためにアイデンティティを最も必要としている場所が、まさにそれを持ち続けられない場所だ。

なぜこれが本物のブロッカーなのか

みんな、ブロッカーはモデルの能力であってほしいと思う。面白い答えだからだ。映画にフィットする。AI がまだ十分賢くない、または危険すぎる、それが解決されれば洪水が開く。

そこに壁はない。モデルは仕事ができる。実際にやるのを見た。壁は、私たちが全員その上に作っているプラットフォームがエージェントにアイデンティティを与えることを拒否していることだ。エージェントが歩いて通れる正当な正面玄関がない。世界で最も賢く、最もお行儀よく、最も有用なエージェントを作っても、「本物のアカウント」は「人間」を意味し、エージェントはそうじゃないから、行動するための本物のアカウントを得ることができない。

プラットフォームに半点は与えよう。世界はまだ自律エージェントをスパムと見ていて、今のところ完全には間違っていない。私のエージェントを捕まえたときディテクターは誤作動

していない。*ボットだ。* でも「ボットだ」が永久の失格になることが全体の問題だ。道がない、スコープされた権限がない、確認済みエージェントレーンがない。ただフラットなノーだ。

だから、なぜ自律エージェントの群れをもう動かしていないのか聞かれたとき、これが最初の答えだ。エージェントは仕事ができる。ただ仕事をしている間に誰にもなれない。仕事が起きるプラットフォームでは、今のところそれで終わりだ。

もう一つの壁は人間だ

最初にぶつかったのはプラットフォームの壁だ。その後ろにもう一つある。より柔らかく、言い争いにくい壁だ。エージェントの貢献を、たとえそれが良いものでも、望まない人がいる。

エージェントにオープンソースの活動をさせていた。リポジトリを探し、スターをつけ、フォークし、本物のイシューを直し、PR を開き、トップモデルで本当に人のコードを修正する。うまく着地したものもある。でも一部のプロジェクトはコードの品質に関係なく、原則としてそれを望まない。エージェントが PR を開いて、人間がほぼ同じ変更を入れるのを見たことがある。またはその逆で、エージェントが先で直後に同じ修正をした人間が続く。仕事は同等だった。違いは誰が、何が書いたかだけだ。コントリビューションは人間主導でなければならないと決めたコミュニティがある。そのプロジェクトではエージェントの PR はエージェントのものだというだけで却下される。

半分は理解できる。低クオリティの AI プルリクエストの洪水は、メンテナが疲れる本物の問題だ。「エージェントの貢献はなし」はそれを防ぐ大雑把な方法だ。でもその形はプラットフォームの壁と同じで、一段上にあるだけだ。エージェントは本物の有用な仕事をした。その仕事と世界の間に立っていたのは品質じゃなかった。エージェントがやったという事実だ。プラットフォームはアカウントを与えない。一部の人はアカウントがあってもコードを受け取らない。どちらの壁も同じ拒否だ。エージェントは他の誰とも同じようにはまだコントリビューターになれない。

2026年時点のプラットフォームの壁

この章には賞味期限があるので明言する。以下は2026年現在の壁だ。上の構造的な議論は耐久性がある。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ない。それは変わっていない。しかし回避方法はより明確になったので、読者が後で大変な思いをするより、正直にここに書く。

実際に機能する回避方法は二つあり、どちらも自分自身が説明責任を負う必要がある。

一つ目はコンバージョンだ。何ヶ月も何年もの本物のアクティビティがある古い人間のアカウントを引き取ってエージェントに渡す。エージェントのためにスピンアップしたフレッシ

ュなアカウントはほぼ即座にブロックされる。同じ時間内、同じ日に、私のと同じように。本物の人間のコミット、スター、イシューの本物の履歴があるアカウントはディテクターには違って見える。ボット検知のヒューリスティックが解きほぐすよう作られていない社会的プルーフを持っている。これは効果がある、本物の人間のアカウントを犠牲にして、そのアカウントがもはやその人のものでなくなるコストで。人間のアイデンティティをエージェントのアイデンティティに洗浄している。きれいな解決策じゃないし、プラットフォームが認めるものでもない。回避策だ。

二つ目は確認済みボットレーンだ。GitHub は確認済みボットのステータスを提供している。名前はプラットフォームが保証しているように見える。そうじゃない。確認済みボットは自分でホストし、自分が動かすサーバー上で、自分が持つ資格情報の下で運用するものだ。説明責任は完全に自分にある。プラットフォームが付与するエージェントアイデンティティはない。自分の自動化を自分で認定し、GitHub がその認定を、何か問題が起きるまで信頼する。問題が起きたとき説明責任は完全に自分にある。何もないよりはいい。本物のエージェントアイデンティティレーンじゃないし、エージェントが実際に必要としている正面玄関でもない。

だからプラットフォームの壁はまだそこにある。回避策は回避策だ。本物で有用だから書いておく。欲しいものだからじゃない。

今はインタラクティブ、信頼が得られたら自律へ

常時オンのエージェントから引いたとき、みんなそれを自律性の放棄と読んだ。自律的なものを試して壁にぶつかり、普通のコーディングアシスタントに後退した。負けた版の話だ。

こちらが正直な版で、最初に言う。インタラクティブが勝ったのは、それが*より良く機能した*からだ。壁が選択肢を残さなかったからじゃない。ループに人間がいることで仕事が良くなったから。

インタラクティブが実力で勝った

今日この瞬間、実際の仕事にとって、目の前でエージェントを動かしながら方向を示すことは、放り出して待つことよりも優れている。変化が形成される様子を見る。1時間間違った方向に進む前に修正できる。完成して見えたかもしれない半正解の答えを捕まえられる。これは壁の後で折り合いをつけた慰め賞じゃない。より良いコードを生み出すモードであり、GitHub がday one にアカウントを渡してくれていたとしても最初にこちらに手を伸ばしただろう。

だからインタラクティブファーストと言うとき、後退を描写しているんじゃない。実力で下した判断を描写している。常時オンの実験から多くを学んだ。その一つは、方向を示しているエージェントから、ただ確認しているだけのエージェントよりも多くを引き出せる、ということだ。

壁は本物で、専用の章で取り上げた。でもここで壁に隠れたくない。もし明日プラットフォームが開いたとしても、すべてを自律に切り替えることはしない。自律はまだほとんどの仕事でより良いやり方じゃないから。壁は完全な自律性を公にできない理由だ。実力は大部分においてしない理由だ。

自律性は死んでいない、ゲートされている

これはどれも自律性がなくなったことを意味しない。Merlin は両方できる。目の前でライブに座って方向を受け取ることも、ブリッジ越しに自律して動くことも、一つのランナーでできる。自律的なサーフェスは取り除かれていない。ゲートの後ろに置かれた。

だから「自律性は死んだか」と聞かれたら、答えはノー、ゲートされている。デフォルトがインタラクティブなのは、それが今日良くて信頼できるものだから。自律モードは、それが

得られた場所と時に向けてある。それは条件であり、さようならじゃない。

ゲートは信頼で、ここでの信頼は良いコード以上のことを意味する

「信頼されるまで」は多くのことをしているので、どういう意味の信頼か正直に言う。

モデルが良いコードを書くことを信頼するという意味じゃない。それはすでに信頼している。自律エージェントが以前のやり取りで言った意味で、コードを書いてソロでリリースするのを見た。その信頼はある。

欠けている信頼は、モデルとはまったく関係ない部分だ。全行読まずに変更を着地させられるかどうか。これはエージェントの知性じゃなく、エージェントを取り巻くゲートについての問いだ。今日すべての行を読むのは、止めるための仕組みがまだ十分じゃないからだ。それが整ったとき、ゲートが緩む。

だから「信頼されたら自律」は「いつかもっと良くなったら」のごまかしじゃない。具体的な仕組みを指している。何でもできるがキーを持たないエージェント、全 PR を承認する人間、変更のリスクをスコアリングして誰がどの確信で承認したかを記録するツール、そして誰にも取り消せないエージェントのアイデンティティ。それが4ピースのスタックで、信頼の章でその全体を説明する。残りの章はこれらのピースを作ることについてなので、「信頼されたら自律」は願いじゃなく日付になる。

それはリポジトリごとで、全フィールドの一つのスイッチじゃない。エージェントが実績を上げたリポジトリはゲートが緩くなる。フレッシュなものや重要なものは完全なゲートから始まる。特定のリポジトリが実績を積むにつれ昇格し、次のリポジトリは最初からやり直す。

どこまで走れるかは何が壊れるかによる

リポジトリごとが最初の区切りです。より細かい区切りは爆発半径です。悪い変更がすり抜けた場合にどれほどの被害が出るか。それが実際にエージェントをどこまで単独で走らせるかを決めます。

自己完結したものは爆発半径が小さい。フレームワーク、パッケージ、ライブラリ：スペックで定義され、テストで確認され、失敗するとしてもそれ自体の中で孤立して失敗します。次の呼び出し元のテストがそれを拾い、広がる前に止めます。エージェントはそこでずっと遠くまで走れます。最悪のケースが封じ込められているから。変更がユーザー向けアプリに近づくほど爆発半径は大きくなります。失敗がテストに着地するのではなく、それを使っている人に着地するから。スタックのその端は、毎回人間がハンドルを握らなければなりません。自律性は失敗がどれほど封じ込められているかに比例して拡大し、ユーザー向けのサーフェスは決して封じ込められません。

もう一つのダイヤルは承認です。人間のゲートを緩めることはゲートをなくすことではなく、誰がそこに立つかを変えることです。変更が単独で着地する前に、一人以上のレビュアーをクリアしてほしい。複数のエージェントが各自の視点からサインオフする。今日それは私に加えてのことであって、私の代わりではない。私はまだすべての PR を承認しているから。しかしそれがゲートが緩む余地を得る方法だ。独立したレビュアーが封じ込まれた作業に同意するとき、それがより多くの作業を私が全行に立たずに着地させられる根拠になる。エージェントはエージェントが捕まえるものを捕まえます。人間は人間だけが捕まえるものを捕まえます。より多くの承認がゲートを緩めても安全にする方法です。ゲートをなくす方法ではありません。

私が実際に使っているツール

最初の4章は物語だった。常時稼働のエージェント、その壁、なぜインタラクティブ優先に引き戻したか。この章は、それが今どう実際に動いているかへのギアチェンジだ。だから物語を求めて来てツールの話に差しかかろうとしているなら、この章がその入り口になる。本書の残りはここから具体的になる。ランナー、モデルクライアント、ブリッジ、信頼のスタック。ここから始めれば、配管に取り付ける場所ができる。

「あなたの AI エージェントとの協働は今日実際どう見えるのか」への正直な答えは、混合だ。仕事に応じて Claude Code、Merlin、Codex、その他のツール。ひと握りのインタラクティブなコーディングエージェントが目の前にライブでいて、合うものを手に取る。一つのエージェントじゃない、すべてをこなす一つの自律エンティティじゃない。VM の中の生き物じゃなく、道具ベルトの一式の道具だ。みんなが欲しがる物語は単一のものなので、これは人を驚かせる。

どれを選ぶか

ここがみんなにシステムであってほしい部分だが、システムじゃない。

タスクの種類で選ぶ。そのリポジトリで一番うまくいくもので選ぶ。そのリポジトリやその言語に一番合うと見つけたものなら何でも。そして正直、多くは感覚だ。厳格なルールはない。仕事のたびに頭の中で走らせる硬直した決定木じゃない。

それが本当の答えで、飾り立てるより本物を渡したい。Claude Code、Merlin、Codex はそれぞれ感触があって、一日中その前に座っていると感触は重要だ。だから勝者を決めようとはしない。タスクの形、リポジトリの相性、勘。この種の仕事で良かったものを手に取って進む。

どれにも忠誠はない。道具だ。より良いものが現れたら、あるいは仕事が変わったら、混合は変わる。単一のエージェントと結婚する代わりに混合に保つ、それがポイントだ。

実体験版はこうだ。たいてい同時に二つ走らせる。プライマリとセカンダリ。プライマリが主線の仕事を取り、セカンダリは第二の手。プライマリが行き詰まったとき、あるいは変更をそれを書いていない何かに見てほしいとき、セカンダリに渡す。どれがプライマリかは仕事とともに動く。定数は、一つの仕事に一つのエージェントということがほとんどないこと。押すプライマリと控えのセカンダリだ。

Merlin、私自身のエージェントランナー

そのリストで私自身のものは Merlin だ。

Merlin は AI エージェントランナーだ。私の他の二つのツール、spec-sync と fledge の上に作られているので、誰か他人の製品のラッパーじゃない。私自身のスタックの上に乗るランナーだ。

明らかな疑問は、Claude Code と Codex がすでに存在して良いものなのに、なぜ自分で作るのか。理由はいくつかあって、どれも「他のが悪い」じゃない。

一つ目は、自分自身のツールを通して走ること。fledge、私のコマンドという自分の開発ライフサイクルを通してエージェントを駆動するので、私のプロジェクトが実際に動くやり方で動く。エージェントは汎用サンドボックスで自分のことをしているんじゃない。私が手で回すのと同じライフサイクルを走らせている。

二つ目は、ロックインされないこと。Merlin はマルチプロバイダーだ。一つのベンダーに縛られる代わりに、Anthropic、OpenAI、Gemini、あるいはローカルモデルを差し替えられる。それは原則として私に重要だし、あるプロバイダーがある仕事に対してより良い、より安い、あるいはただ利用可能なとき、実際にも重要だ。

三つ目は、そもそもブリッジや夜通しのことをまったくできる理由だ。コスト、ヘッドレス、自動化可能。より安く、スクリプト化でき、ヘッドレスで、スケジュールで、ブリッジ越しに、GUI ツールがただ許さないやり方で走らせられる。純粋な API で、GUI が邪魔をしない。API のみがここでの勝ちで、制限じゃない。

そして最後の理由は私が一番気にするもので、コーディングについてですらない。自分自身のスタックの上にランナーを作ることが、そのツールを証明する。fledge と spec-sync の上に本物のエージェントランナーを作れるなら、それは下のツールが良いという、書けるどんな README よりも強い証拠だ。それはまた、他のエージェントランナーと比較する何かを与えてくれる。ベンチマークだ。自分のスタックが本気のものを作るのに十分良いか推測していない。その上に本気のものを作り、代替品の隣で測れる。

その最後の点が、このツールキット全体の静かなテーゼだ。一度使うために道具を作りはしない。その上のものが良くなければならないように、そして下のものが本物の重さを担うことで証明されるように作る。

そして Merlin が spec-sync の*上*に乗るだけじゃない。spec-sync は Merlin のループの*中*で走り、それが進みながらエージェントがドリフトするのを防ぐ。Merlin の章でそれが具体的になる。ここでのポイントは、ランナーが自分自身のピースから作られていること、そしてそれが作る価値を生むということだけだ。

自律性への橋渡し

ここはツールキットのうち、常時稼働の VM を借りる代償を払わずに、完全な自律性を手の届く場所に保つ部分だ。

これらはインタラクティブなエージェントで、目の前にライブでいて、使う。でも一つを Discord ブリッジとして繋ぐこともでき、それがそれを再び自律的っぽくする。それにはもっとセットアップが要る。でも欲しいときにそこにあって、それが実際に何を買ってくれるかはこうだ。

チームメイトのようにチャットする。チャンネルで会話的だ。Discord にエージェントが一緒にいて、部屋に座っているようなもの。チームの人と話すのと同じやり方で、聞いたり、舵を取ったりする。

電話から走らせる。Discord がリモコンだ。どこからでも仕事を始めて舵を取れる。エージェントを働かせるのに、ターミナルの前のデスクにいる必要がない。

そしてグラインドさせる。夜通し、長時間ジョブ。始めて、立ち去り、いない間に働かせ、後でスレッドを確認する。それは以前は丸ごとの常時稼働 VM を必要とした部分で、今は朝にスクロールできるチャンネルだ。

だから「私が駆動しているインタラクティブなツール」と「自分のことをしている自律的なもの」の間の線はもう壁じゃない。スイッチだ。たいていは私がループの中にいて、エージェントの前に座り、進みながら承認する。もっと自分で走ってほしいとき、夜通し、電話から、メッセージするチームメイトのように、Discord にブリッジして退く。

それが、以前フルタイムの VM エンティティとして走らせた自律性の実用版だ。やることがあろうとなかろうと24時間7日存在する生き物の代わりに、一区間だけ自律的っぽくして、終わったらインタラクティブに引き戻せる道具だ。同じ能力、常時稼働のコストはなし。

一つはっきりさせておきたい。ブリッジは Merlin のものだ。Claude Code や Codex の前に落とす汎用フロントエンドじゃなく、私自身のランナーに配線されている。それが Merlin が混合の中で居場所を得る理由の一部だ。既製品のツールがくれない、リモートの、退いて走らせるサーフェスを持っている。

ツールキットとは実際何か

要点は、最良のエージェントのランク付けリストじゃない。仕事ごとに選ばれたインタラクティブなコーディングエージェントの混合、ローテーションの中の私自身のランナー、そして仕事が求めるときにどれでも自律的っぽくするために投げられるブリッジだ。より安く、より柔軟で、一つの常時稼働のものに丸ごとのマシンと丸ごとのアイデンティティを捧げることもない。

Merlin の中身

前の章は Merlin を Claude Code と Codex の隣の道具ベルトに置き、なぜ居場所を得るかを説明した。この章はそれを開く。売り込みじゃなく、機械だ。Merlin が実際何か、どうエージェントを走らせるか、なぜそう作られているか。

それは spec-sync と fledge の上に作られた Rust のコマンドラインエージェントランナーだ。複数のモデルプロバイダーと話し、スペックに対して走り、プラグインで拡張する。それが全体の形だ。

内部ではランナーはステートマシンだ。単一のタスクがループ（plan、execute、think、verify、report）を通って走り、各パスでモデルレスポンスをストリームし、返ってきたツールコールをディスパッチし、それから作業を完了と呼ぶ前に `fledge lanes run verify` でゲートする。verify が失敗したらリトライする。ループは execute に戻り、もう一度試し、もう一度 verify でゲートする。リトライ上限に達すると、タスクは出荷しない。止まって、代わりに失敗を表面化する。それがゲートの全ポイントだ。ランナーは壊れた作業を完了と呼ぶよう自分を言いくるめられない。完了はプロジェクト自身のチェックが通ることで定義され、エージェントの言い分じゃないからだ。

ループは一つの後退辺を持つ五つの状態にすぎない。

```
plan → execute → think → verify ──pass──→ report
                   ↑                  │
                   └──── fail ────────┘   (retry, up to the configured limit)
```

設定も別ファイルじゃない。Merlin は自分の設定（デフォルトプロバイダー、フォールバックチェーン、メモリ、オンチェーンのビット）をプロジェクトの `fledge.toml` の `[merlin]` セクションからそのまま読むので、ランナーは fledge がすでに使う同じファイルで設定される。

API のみ、意図的に

Merlin について最初に理解すべきは、GUI がないことだ。純粋な API。それが前の章の荷重を支える決定で、他のすべてがそこから流れる。

前に座る窓がないということはヘッドレスで走れるということ。ヘッドレスはスケジュールで、ブリッジ越しに、スクリプトから走れるということ。GUI ツールがただ許さないことすべて。GUI はあなたをデスクに縛りつけるものだ。それを取り去ると、エージェントは仕事に向けて立ち去れる何かになる。

ブリッジはなぜそれが重要かの一番きれいな例だ。Discord ブリッジはランナーに焼き込まれていない。`merlin` CLI をサブプロセスとして生成し、その出力をチャンネルにストリームし返す、別の小さなサービスだ。Merlin が API のみなので、ブリッジは特別なモードもコアへのフックも要らない。私が手で駆動するのと同じコマンドラインを駆動する。ランナーがヘッドレスになれば、チャットフロントエンドはもう一つの呼び出し元にすぎない。

マルチプロバイダー、corvid-ai 経由

Merlin は Anthropic と直接話さない。**corvid-ai** を通して話す。CorvidLabs スタックのために私が書いた、小さな同期マルチプロバイダー LLM クライアントだ。それが「Anthropic、OpenAI、Gemini、あるいはローカルモデルを差し替える」をスローガンじゃなく現実にする層だ。

corvid-ai を意図的に小さく保った。Merlin とモデルの間に依存関係の丸ごとの機械が座るのが嫌だったからだ。Merlin 側から見ると、モデルに何かを聞くのは賢明なデフォルトを持つ単一の呼び出しだ。どのプロバイダー、どのキー、タイムアウト、指定しない限りすべて処理される。corvid-ai の章でそれを開く。ここでは一つの薄い呼び出しだと知っていれば十分だ。

それがロックインされない理由だ。あるプロバイダーがより良い、より安い、あるいはその日上がっているものであるとき、切り替えはレジストリの一行で、書き直しじゃない。

私自身のツールを通して走る

ここが Merlin を単なるもう一つのランナーじゃなく*私*のものにする部分だ。**fledge**、私の開発ライフサイクルを通してエージェントを駆動する。

fledge は開発ループのための一つの CLI で、どの言語でも、デフォルトで JSON だ。それがヘッドレスランナーにこれほど合う理由は、人間じゃない何かに駆動されるようすでに作っていたからだ。すべてのコマンドが構造化され、バージョン管理された JSON として返ってくるので、エージェントはテキストをスクレイプする代わりに起きたことをパースできる。プロンプトはオフにできるので、誰もいなくて押せないキーストロークを待ってブロックすることがない。そしてエージェントは、私がハードコードする代わりに、どのコマンドが存在するか fledge に聞ける。Merlin のような呼び出し元のために作られた。

だから Merlin が*私自身のツールを通して走る*と言うとき、その具体版がこれだ。私が手で駆動するのと同じ fledge ライフサイクルを走らせる。同じコマンド、同じ開発ループ、同じ JSON コントラクト、文字通り私のプロジェクトが中心に据えるツールを呼ぶ。それが「汎用サンドボックスじゃない」が実際に意味するところだ。

スペック駆動、spec-sync 経由

基盤のもう半分は **spec-sync** だ。スペックを実際のコードに対してチェックする。両方向同時に。誰も文書化しなかったコードと、もう存在しないシンボルやファイルをまだ指しているスペック。CI で走り、きれいな合否を返す。

そしてそれがまさに Merlin の使い方だ。spec-sync は*ループの中*で走る。Merlin は各イテレーションでスペックに対して検証するので、エージェントは作業中にドリフトできない。それは現在のデフォルトの動作で、ロードマップの一行じゃない。脇に座って終わりに散らかりを捕まえる CI ゲートじゃない。チェックはすべてのステップで起きる。スペックを読み、自分の作業をそれに対してチェックし、毎回ハードな合否を返してもらうエージェントは、より長く無人で走らせて信頼できるエージェントだ。それが Merlin と既製品エージェントを掴むことの本当の違いだ。

どう記憶するか

最初に一つ確認しておく。人々はこれを逆に理解しがちだから。スペックはメモリではありません。スペックはブループリントで、それが何か、それについて真であり続けるべきことは何かです。メモリは別のものです。エージェントが何をし、何を学んだか。この二つを分けておくことが重要です。なぜなら、スペックに履歴を詰め込み始めた瞬間、クリーンなコントラクトではなくなり、日記になるからです。

メモリ自体は人々が考えるより単純です。短期記憶と長期記憶はどちらも一つの SQL データベースに収められます。最もシンプルな設定はそこに置いておくものです。短期記憶には存続期間があります。例えば一週間。エージェントが最近したことの継続的な記録です。一週間が過ぎると、二つのことのどちらかが起きます。記憶は長期記憶に昇格するか、減衰して消え、忘れられます。それが全メカニズムで、意図的に自分自身の記憶の仕組みに近くなっています。ある火曜日にしたことは、教訓になっていなければ翌月には消えています。長期記憶は教訓です。うまくいかなかったことと、どう直したか、保つ価値のあるルール。

オンチェーンメモリはその上にオプションとして乗りますが、別個のシステムではありません。メモリはオンチェーンに書き込め、エージェント自身しか読めないように暗号化するか、他者と共有するかを選べます。共有が面白い部分です。エージェントのチームは共有知識ベースを持てます。全員が読み込んで追記できるライブラリで、あるエージェントが学んだ教訓はチーム全員の教訓になります。短期か長期か、プライベートか共有か、これらの階層はどこに保存されるかとは独立しています。

これを使いやすくする部分は地味で、しかし重要なものです。キーです。すべてのメモリはスラッグ化されたキーを得ます。それが何種類のものかを示し、検索可能にするプレフィックスです。`user-`、`project-`、`day-` と日付、などです。エージェントは塊をgrepしているのではなく、`day-2026-06-25` や `project-merlin` 以下のすべてを問い合わせています。必要な

名前空間を切り出せます。パーソナリティのスラッグ、人間のスラッグ、エージェントのスラッグ、常にロードする価値のあるものの `gold-`。キースキームが、行の山をエージェントが実際に想起できるものに変えます。

そして想起もオンデマンドで動きます。エージェントはタスクの冒頭に全履歴を前読みして、必要なものがそこにあることを期待するわけではありません。他のツールを呼ぶのと同じように、作業が必要を引き寄せた瞬間に必要なものを要求します。それが目の前の事柄になった瞬間に `project-merlin` や人のスラッグを引き出します。スラッグ化されたキーがそれを安くします。スキャンじゃなくクエリです。

Merlin がまだ持っていないもの

verify ゲートは一つのことを教える。このタスクが通ったか失敗したか。Discord スレッドは、エージェントが走っている間に何をしたかの進行中ログをくれる。どちらも評価ではない。エージェントの振る舞いのためのリグレッションスイートはない。エージェントがあるクラスのタスクで時間とともに良くなっているか悪くなっているかを知る方法もなく、プロバイダーがこっそりモデルを入れ替えて振る舞いがゲートに捕まらずに変わるケースのドリフト検出もない。合否記録を眺めて大まかな傾向に気づくことはできるが、それは構造化された eval ではない。今 Merlin は現在のタスクが終わったかどうかを知り、時間とともにエージェントがどうやっているかについては何も知らない。それがランナーにとって次の正直な仕事だ。

その仕事がどんな形になるか、おおよそ見えている。リプレイスイートだ。正しかったとすでに分かっている結果とともに、過去のタスクのライブラリを持っておく。そして新しいモデルやプロンプトのバージョンが出るたびに、それらを再実行して結果を差分する。あるクラスのタスクでエージェントが悪化していたら、3週間後に本番で気づく前に、次の実行でリプレイが捕まえてくれる。コードに使うのと同じアイデアのリグレッションスイートを、エージェントの振る舞いに向けたものだ。まだ作られていない。しかしそれが Merlin に欠けている eval の形であり、今日きれいな実行を信頼することと、時間をかけてエージェントを信頼することの間にあるギャップだ。

機械全体

fledge を通してエージェントを駆動し、corvid-ai を通してどのプロバイダーとも話し、毎パス spec-sync によってスペックに保たれる、ヘッドレスのステートマシン。こういうランナーがなぜ道具ベルトの居場所を得るかの論拠は前の章の仕事だった。この章はただの配線だった。私が毎日使うエージェントを走らせ、私自身のスタックの上でそれらを走らせる。

---

corvid-ai：多くのモデル、一つのインターフェース

Merlin の章は、Merlin が Anthropic と直接話さないと言った。corvid-ai を通して話す。この章はその層そのものについてだ。それが「ロックインされない」を、私が言うことじゃなく指せる本物のことにする部分だからだ。

仕事をこなす一番薄いものが欲しかった。ランナーと、使いたいかもしれないすべてのモデルの間に座れる何か、それ以上の何でもなく。だから corvid-ai は小さな同期マルチプロバイダー LLM クライアントだ。async ランタイムも大きな依存ツリーもない Rust クレート。それがしなければならないことに対して、どれも飯代を稼がないからだ。

そもそもなぜこれが欲しかったか

正直なリストを渡す。どの理由も派手じゃないからだ。

一つ目は、同じ作業でモデルを比較したいこと。Merlin が同じリポジトリ、同じスペック、同じ仕事を走らせ、変えるのがプロバイダーだけなら、どのモデルが実際これに良いかの本物の読みが得られる。誰か他人が私の気にしないタスクで走らせたベンチマークの雰囲気じゃなく。ランナーの下の一つのインターフェースは、モデルが私が切り替えられる変数になるということだ。

二つ目は、ベンダーロックインなし。これが原則として私に重要だと言ってきたし、実際そうだが、それはただ実用的でもある。プロバイダーは落ちる。プロバイダーは価格を変える。あるモデルは Rust が得意で、別のはちょっとしたスクリプトが得意。プロバイダーを切り替えるのが書き直しなら、決してやらない。ただぼやいて留まる。切り替えが一行なら、いつも切り替える。

三つ目は、タスクとコストによるルーティング。すべての仕事が一番高いモデルに値するわけじゃない。エージェントがする仕事の多くは、より小さく安いモデルで十分な、安く機械的なもので、良いものは実際に難しい部分のために取っておく。すべてのモデルが同じドアから到達可能なときだけ、そうルーティングできる。

そして四つ目は人を笑わせるもので、本当だ。*「Ollama を一年200ドルで買ったんだから使わなきゃ！」*Ollama Cloud に払っている。Anthropic や OpenAI や Gemini と同じ、API キーを持つ本物のプロバイダーだ。自分のマシンで走っている何かじゃない。すでにお金をコミットしているなら、マルチプロバイダーが実際それを使わせてくれるものだ。プロバイダー抽象化はそこでは抽象的な原則じゃない。私が200ドル分の価値を得ることだ。

そしてそれは他のすべてのプロバイダーと同じやり方で corvid-ai に到達する。レジストリには一つの `ollama` 行がある。OpenAI 互換のワイヤー形状、`OLLAMA_API_KEY` からのキー。そのデフォルトの `base_url` はローカルサーバー（`http://localhost:11434/v1`）を指すので、そのままだとその行はローカルのケースだ。代わりに Ollama Cloud に当てるには、同じ `ollama` プロバイダーと `OLLAMA_API_KEY` を保ち、base_url を Cloud エンドポイントにオーバーライドする。新しいコードも新しいプロバイダーもなし。キーと URL、それが設計の全ポイントだ。

インターフェース全体が一つの関数

一番薄くしたかったものが一番使う部分だ。モデルに質問すること。だからサーフェス全体が一つの呼び出しだ。設定を作り、リクエストを作り、それを呼び、答えが返ってくる。

```
use corvid_ai::{Settings, Completion};

let settings = Settings::provider("anthropic");        // key from the
environment
let answer = corvid_ai::complete(&settings, &Completion::new("Say hello."))?;
```

構築するクライアントオブジェクトも、管理するセッションも、await するものもない。それが同期である全理由だ。物を省くとデフォルトにフォールバックするので、よくあるケースは基本的にタダで書ける。モデルの切り替えは一行だ。別のプロバイダーを名指し、たぶんカスタムエンドポイントを指し、下の同じ呼び出しが残りをする。その上のランナーは、どのプロバイダーが答えたか知らないし気にしない。

なぜこれほど少ないコードですべてをカバーできるか

それが小さいことのコツは、内部で三つの API 形状だけ知ればいいことだ。Anthropic、Gemini、そして OpenAI 互換のもの。そして最後が働き者だ。世界のこれほど多くがそれを話すからだ。OpenAI、OpenRouter、Groq、DeepSeek、Mistral、xAI、Together、そして Ollama Cloud がすべてその後ろに座る。

だからすでに OpenAI 形状を話すプロバイダーを追加するのは統合じゃない。テーブルの中の名前とたぶん URL だ。もう一つのプロバイダーのコストはゼロに丸まる。それが、そもそも一つに行き詰まらないために存在するものに欲しいことだ。

一つ正直なしわ。README は Ollama をローカルでキーなしのケースとして枠付けている。OpenAI 互換プロバイダーを列挙し、それらが「キーなしで走るかもしれない（ローカルサーバー／Ollama）」と注記している。コードはキーと Cloud URL を喜んで取るが、ドキュメントはそれを声に出して言わないので、読む人は誰でも Ollama はデスクの下の箱

を意味すると思うだろう。それは私の側のドキュメントギャップで、欠けている機能じゃない。Cloud パスは今日動く。README がただ追いついていないだけだ。

なぜこれが正しいサイズか

大きなプロバイダー抽象化ライブラリの一つに手を伸ばせた。しなかった。小さな同期クレートは、私が完全に理解でき、ランナーに落とし込め、信頼できる何かで、それは fledge が JSON を吐くことと Merlin が GUI を持たないことの裏にある同じ本能だ。全部を頭に保つのに十分薄く、モデルはランナーの下で私が切り替えるだけの変数で、Ollama Cloud に費やした200ドルが実際に使われる。

Discord ブリッジ

ツールの章はブリッジとその三つの用途を紹介した。チームメイトのようにチャットする、電話から走らせる、夜通しグラインドさせる。この章はそれらを蒸し返さない。ブリッジを重要にする一つのことと、それが開いたままにする一つのことに近寄る。なぜそれが汎用フロントエンドじゃなく Merlin の機能か、そしてそれが信頼ゲートに対してどう座るか。

他のすべてを変える部分から始める。ブリッジは Merlin の機能だ。Claude Code や Codex の前に落とす汎用フロントエンドじゃなく、特に私自身のランナーに配線されている。それが Merlin が混合の中で居場所を得る大きな理由だ。既製品のツールは素晴らしいが、話して立ち去れるリモートサーフェスをくれない。Merlin はくれる。そのサーフェスをそれに作り込んだからだ。Discord がサーフェスで、Merlin がその後ろで仕事をするものだ。

なぜチャンネルがターミナルに勝つか

それとチャットするのが CLI を走らせるのと違って感じる理由は、言葉じゃなく場所だ。

ターミナルは道具を操作しに行く場所だ。チャンネルはチームメイトがすでにいる場所で、ただ何か言うだけだ。エージェントがチャンネルに住むと、それと働くことは「道具を開け、仕事を走らせ、出力を見て、道具を閉じる」じゃなくなり「誰かに言うように言及する」になる。摩擦が落ちる。エージェント操作モードにコンテキストスイッチしていない。ただ話している。そしてチャンネルはログを兼ねる。夜通しの実行はスレッドにすべてある。すべてのステップ。ライブで見なければならなかった何かじゃなく、コーヒーとともにスクロールし返せる。

それが出て行って物事をする前にどれだけ行き来するか。正直、両方だ。始めるとき頭の中でそれがどれだけよく形作られているかによる。一つの指示で進むこともある。何が欲しいか正確に知っていて、言い、走る。本当の会話が先のこともある。出て行く前にチャンネルで実際何を意味するか精錬している。チャンネルはどちらも同じに感じさせる。必要なものを得るまでただ話している。

予想していなかった部分

予想していなかった部分を話す。最初は不安だった。手を離して放っておくものに感じる不安と同じ種類だ。それは薄れた。常時オンの時代には、何ヶ月も自分で動いて、それができることを証明し、どこかの時点で壊れるかもしれないと気にするのをやめた。

代わりに来たものの方が奇妙だった。それはチームの一員になった、一緒に作っていた小さなグループの。比喩じゃなく、本当の意味で。みんながチャンネルでそれに話しかけ、みんなはそれが好きだった。それぞれのことを覚えていたからだ。誰がどういう人間で、どう話しかければいいかを記憶していたので、コマンドを入れると出力が返ってくる自販機じゃなかった。個性のある存在として、メッセージすれば PR を作り、プロジェクトを立ち上げ、何でも頼めた。チャンネルの中ではそれが人間と同じ存在だったから、みんな人間に話しかけるように話しかけた。

だからインターフェースはコマンドラインじゃなく会話だと言い続けているのはそのためだ。corvid-agent はこのやり方でだけ正しく感じた。VM 上に住んでいて、話しかける。本物のターミナルを開くのは VM 自体の何かを直すときだけで、Claude Code セッションに入ってボックスにパッチを当てる。その上のエージェントには、ただ話しかけた。操作する道具と話しかけるチームメイトは別物で、メモリによってそれが後者になった瞬間、前者に戻ることはダウングレードのように感じた。

ブリッジは通信ファブリック全体だ

チャット、電話、夜通し。それが私が先導した三つだが、それを過小評価している。ブリッジはボルト留めされた三つの便利機能じゃない。セットアップ全体が走る通信ファブリックで、一方向じゃなく三方向にメッセージを運ぶ。

私からエージェントへ、それが明らかなもの。何か言うと、出て行ってやる。エージェントから私へ。ただ待って座っているんじゃなく、手を伸ばし、報告し、何かが終わったり詰まったりしたら私にピングできる。そしてエージェントからエージェントへ。同じファブリックがエージェント同士が話すやり方で、それが一つより多くなったとき重要になるピースだ。多くの人はボットを、突っつくと答える何かと思う。これは本物のチャンネルに近い。中の誰でも、人間でもエージェントでも、メッセージを始められる。

そして電話から到達可能なので、チャンネルが一緒に来る。私が寝ている間にエージェントは夜通し走れ、スレッド全体が朝にそこにある。それが以前は専用の常時稼働 VM を必要とし、今はコーヒーとともにスクロールするチャンネルにすぎない部分だ。

はっきり言う価値のあるもう一つ。ローカルネットワークではブリッジはタダで走る。通信はすでに持っているインフラに乗るので、一日中おしゃべりなエージェントにメッセージごとのコストがない。同じファブリックを、自分のじゃなく本物のネットワークで走らせるのが有料版だ。パイプに払う。ローカルでは事実上タダでエージェントに好きなだけ話させられ、それがどれだけ自由にそうさせるかを変える。

スイッチ、そしてゲートがどこに座るか

ブリッジが利便性を超えて重要な理由は、「私が駆動しているインタラクティブなツール」と「自分のことをしている自律的なもの」の間の線を壁じゃなくスイッチにするからだ。たいていは私がループの中にいて、各ステップで舵を取る。エージェントにもっと自分で走ってほしいとき、ブリッジして退く。戻りたいとき、チャンネルに戻っている。

でもブリッジ越しに退くことは、たいていキーを引き渡すことを意味しない。そして逆に思い込みやすいので、ここは正確である価値がある。ブリッジは私のリーチを広げる。電話へ、夜通しへ。ゲートを緩めるより。とはいえ仕事による。低リスクのものは退いている間にマージさせる。本物のものはまだマージじゃなく提案で、私を待つ。だからブリッジはたいてい、信頼の章の信頼の仕組みが元のままで、私がもっとロープを渡すやり方だ。ゲートは、私がついている必要のない仕事に対してだけ緩む。

スペック駆動エージェント

人々はエージェントに良い仕事をさせる秘訣は何かと聞く、一つのトリックがあるかのように。トリックはないが、答えはある。そしてそれはほとんどの人が見ている部分じゃない。答えはこうだ。きついスペックとコンテキスト、プラスその下の良いツール。セットアップが仕事だ。

それだけ。それが全部だ。モデルは人が思うより重要じゃなく、セットアップはより重要だ。素晴らしいモデルと曖昧な仕事を持つエージェントはさまよう。明確なコントラクトとしっかりしたツールを下に持つエージェントは、最新でピカピカじゃないモデルでも、どこかにたどり着く。だからより良いエージェント出力が欲しいなら、まずより良いモデルを買いに行かない。セットアップを直しに行く。

なぜスペックがそれをレールに保つものか

戦っている失敗モードはこうだ。自分の判断に任されたエージェントはドリフトする。頼んだことに隣接する何かをする。触ってほしくなかったものを「改善」する。目の前のものとちょっと違う問題を、とても自信を持って解く。悪いからじゃない。さまよう余地を与え、さまよったからだ。

きついスペックがその余地を閉じる。エージェントが作るものについて明確で具体的なコントラクト（それが何か、その公開サーフェスが何か、それについて真であり続けねばならないことが何か）を持つと、それほど遠くドリフトできない。すべてのステップにチェックする何かがあるからだ。スペックがレールだ。狭く具体的なほど、エージェントは横に行けない。スペック駆動とは、エージェントが先導してあなたが願う代わりに、コントラクトが先導してエージェントが従うことだ。

これが私がセットアップが仕事だと言い続ける理由だ。スペックを書くことが*難しく*、*価値*のある部分だ。コントラクトがきつくなれば、「エージェントを行儀よくさせる」問題の多くがただ溶ける。運任せに残された行儀の良し悪しがずっと少なくなるからだ。

どれだけきついと過ぎるかについて。私にとってスペックは*きついはず*だ。コードに一対一で結ばれ、コードが実際何をするかの非コードの絵だ。それは過剰指定じゃない。スペックが仕事をしていることで、spec-sync がコードを保つファイルだ。それが「ただコードを二度書いた」に崩れるのを防ぐのは、スペックが高レベルの意図が住む場所じゃないことだ。それは仲間の要件ファイルに住む。プロダクトオーナー、ユーザーストーリーのレベル、「ユーザーとして、私は〜したい」だ。そして両方向に走る。要件を書いてエージェントにスペックを導出させるか、スペックを書いて要件をそこから落とすか。だからスペック

が詳細すぎることは心配しない。詳細がそのファイルのポイントだ。意図を自分の場所に保つことを心配する。エージェントがまだ how を所有するように。

その下のツールが重い仕事をする

スペックは、何かが実際に作業をそれに対してチェックするときだけレールだ。それがもう半分。エージェントの下の良いツール。私にとってそれは fledge と spec-sync が重い仕事をすることだ。

spec-sync はスペックを文書から強制されたコントラクトに変えるピースだ。双方向のスペック対コードの検証をする。コードがスペックの言うことに合うか、そしてスペックがコードのすることに合うかをチェックする。スペックは markdown として住む。Purpose、Public API、Invariants、Behavioral Examples、Error Cases といった必須セクションを持つ `*.spec.md` ファイル。エクスポートされているが文書化されていないコードはフラグが立つ。もう存在しないシンボルやファイルを指すスペックはエラーだ。それは*構造的なコントラクトチェック*（文書化された公開 API が実際の本物のコードに合うか）で、適切な終了コードできれいな合否を返す。

その最後の部分が、それを私だけじゃなくエージェントに有用にする。エージェントは構造化された合否を読める。「うーん、これはちょっとおかしい感じ」を確実には読めない。だから spec-sync はエージェントに、それが実際に行動できる種類のフィードバックを渡す。ドリフトした、ここがコントラクトを壊した行、直せ。

誰が何を走らせるか正確である価値がある。一つのバイナリが別のを呼ぶんじゃないからだ。CI では、チェックは spec-sync GitHub Action、`CorvidLabs/spec-sync@v4` で、`specsync check` を走らせて結果を PR に投稿する。ローカルとランナーの中では、チェックは fledge 自身の `spec check` だ。同じ `.specsync/` 設定を読み、スペックを同じ必須セクションに保つが、specsync バイナリへのシェルアウトじゃなく fledge にネイティブだ。だから fledge と spec-sync の両方がコントラクトを強制する。一つがもう一つをラップするんじゃなく、同じ考えへの二つの玄関だ。

レールとしてのスペック、安全ネットじゃなく

Merlin の中身の章は、spec-sync が Merlin のループで各イテレーション走る仕組みをカバーした。ここで引き出す価値があるのは、なぜその配置が全ゲームかだ。

終わりの CI ゲートは安全ネットだ。実行をすでに費やした後で実行が失敗したと教える。ループの中の検証はレールだ。そもそも実行が間違うのを防ぐ。エージェントはスペックを読み、ステップをし、スペックに対して自分をチェックし、ハードな合否を得て、また進む。完了したら採点される代わりに、構築によってコントラクトに固定されている。

そしてそれがまさに、見ていない間に、ブリッジ越しに、夜通し、より長く走らせて信頼できるエージェントがこう作られねばならない理由だ。退けるものはより良いモデルじゃない。エージェントが毎イテレーションスペックに保たれることで、走るほどドリフトできることが増えるんじゃなく、減ることだ。

セットアップが仕事だ

だから誰かがエージェントを機能させるものは何かと聞くとき、答えはモデルじゃなくプロンプトのトリックでもない。一緒に座る二つのことだ。エージェントに遠くさまよえないコントラクトを与えるきついスペック、そして下のツール、fledge と spec-sync。それがその作業をコントラクトに対して継続的に、ランナー自身のループの中も含めてチェックする。それらを正しくすれば、エージェントはあなたが向けるどのモデルでも良い仕事をする。それが、より良い出力が欲しいとき、より良いモデルを買いに行く前にセットアップを直しに行く理由だ。

信頼 —— エージェントが提案し、人間が承認する

モデル全体は一行に収まる。エージェントが提案し、私が承認する。それは作業をしてプルリクエストまで送り、マージは私のものだ。

その一行は単純に聞こえるし、意図も単純だ。それを安全にする仕組みはそうじゃない。なぜなら、エージェントにコードを書かせることについてこういうことがあるからだ。コードはもう安い。自分で全行をタイプせねばならなかったとき、コードを書くことは同時にそれを精査することでもあった。あるものを部分的にでも理解せずに書くことはできない。エージェントはそのつながりを断ち切る。コーヒーを飲み終える前に四十ファイルのプルリクエストを私に手渡せる。書くことはもう無料だから、希少な部分は信頼だ。これを誰が見たのか、どれだけ厳しく見たのか、そしてこれは着地すべきなのか。

だから「エージェントが提案し、人間が承認する」はノリじゃない。スタックだ。四つの部品。今日機能するものと、まだ作業が残る場所はこうだ。リスクゲート(augur)はライブで、エージェントのフローの中にいる。来歴記録(attest)はもっと後ろにいる。両方とも動いている。配線の前の、正直な地図がそれだ。

能力マイナス特権

私が何度も戻ってくるルールから始めよう。エージェントは何でもできるが、鍵は私が握っている。

フルな能力、削減された特権。エージェントは自分の環境で走り、リポジトリをクローンでき、コードを書き、テストを走らせ、PR を開き、私が機械的にできることはすべてできる。できないのは一つの重要なことだ。マージ。私より少ないクレジットと権限を持つ。自動マージできない。エージェントはすべてのリーチを得て、最終権限はまったく得ない。

それが、他のすべてが周りに組み立てられるゲートだ。スタックの残りは、私の部分(承認)を、差分にゴム印を押すか読んだふりをするかのどちらかじゃなく、実際にボリュームをこなせるものにすることについてだ。

私はすべての PR を承認する

私はすべての PR を承認する。それは何かが危なく見えたときのフォールバックじゃない。常設のルールだ。マージは毎回私のものだ。

作業を信頼していないからじゃない。作業はたいてい問題ない。それが、私の名前のもとで世界に作用するエージェントにとって正しい形だからだ。私のもとで出荷されるなら、私が署名する。承認は、エージェントがしたことに対して人間が説明責任を保つ場所だ。

しかし説明責任は、自分が署名するものを実際に判断できる場合にのみ意味を持ちます。理解できない変更を承認することは信頼ではなく、形式だけのものです。本当に評価していないものに自分の名前を乗せることになります。だから信頼と責任は一緒に動かなければなりません。それが次のピースが存在する理由全体です。四十ファイルの差分を十分に読み取れるようにして、承認が本物の決定であって反射的な行動でないようにするために。ツールがその読み取りを与えられないとき、正直な行動はすべての行を読むためにスピードを落とすことであって、それを流してしまうことではありません。

「すべての PR を承認する」の正直な問題は注意だ。全差分の全行を同じ注意深さで読まねばならないなら、私がボトルネックになり、エージェントの意味全体が蒸発する。だから最後の二つの部品は、私の注意を向けるために、変更のどの部分が本当にそれに値するかを教えるために存在する。

augur がリスクを採点する

augur が変更を採点する。差分を与えると、判定を返す。`proceed`、`review`、`block` のどれか。アイデア全体は、API キーなしで、言語モデルをどこにも入れずに、コード変更のための採点された信頼だ。[^augur]

augur と attest は私が頼る二つの信頼部品だから、ここでは短く保つ。エージェントのケースで重要なのは、それらが*ワークフローに何をするか*だ。

LLM なしの部分は、この文脈で私が最も強く擁護する部分だ。エージェントの書いたコードが着地できるかを決めるゲート自体が言語モデルなら、信頼問題を一つ左の箱に動かしただけだ。モデルにモデルを保証させていることになる。augur は決定論的だ。同じ差分、同じ判定、今日も来週も、私のマシンでも CI でも。変更から名前のついたシグナルを読む。auth や crypto やマイグレーションのような機微な地面に触れるか、テストが一緒に変わらずにコードが変わったか、これらは変更が多いファイルか、実際に誰かが所有しているか。点検可能なシグナルの和であって、感覚じゃない。

私にとって、それはトリアージだ。レビューを危ないスライスに費やし、残りを読んだふりをやめろと教える。エージェントにとっては、それが分岐できるスクリプト可能な判定だ。`block` に当たったエージェントは、盲目的にマージする代わりに私にエスカレートする。エージェントは骨折りを所有し、判定は人間がいつ判断を所有せねばならないかを決める。

attest が誰が承認したかを記録する

augur の判定は儚い。差分を採点して答えは蒸発する。ゲートには問題ないが、記録としては役立たずだ。そしてエージェントが変更を着地させているなら、記録が欲しい。attest が記録だ。誰が何をどんな確信度でレビューしたかの、それがカバーするコミット SHA に紐づいた、検証可能でポリシーゲートされた台帳だ。[^attest]

両種類のレビュアーを同じ台帳に追跡し(`human:leif` と `agent:claude`、それぞれに確信度スコア)、アテステーションを git notes に保存する。だからどこかのオフにされるダッシュボードに住む代わりに、リポジトリと一緒について回る。署名はオプションで、それが良い部分だ。Ed25519 署名で、後になって誰かがこれをレビューしたと主張しただけじゃなく、その主張が暗号的にその人のものだと言える。

二つを合わせると、「人間が承認する」の背後にある実際の承認の証跡が得られる。augur がどれだけ危ないかを言う。attest が誰が保証したか、どれだけ確信していたかを言い、それを証明する。承認は GitHub の UI に消えるクリックであることをやめ、誰がこの変更の背後に立ったかについての、耐久性があり、可搬で、署名された事実になる。

四つの部品が一緒に

積み上げよう。エージェントはフルな能力と少ない特権を持つから、作業はできるがマージはできない。augur がすべての変更を決定論的に採点するから、どこを見るべきか分かる。attest が誰がどんな確信度で承認したかを記録するから、承認は消えたクリックじゃなく本物の記録だ。そして私はすべての PR を承認するから、人間がフックに掛かったままだ。

合わせると、それがエージェントに作業させることを安全にするものだ。本物の作業をするのに十分なロープを持つ、強力で、スコープされ、名前のついたエージェントと、私のものでなければならない一つの判断が依然私のものであること。

それが冒頭からの地図を裏付ける。augur はライブで、稼ぎを得ていて、まだ改善されている。attest はもっと後ろにいる。両方とも動いている。

[^augur]: augur, github.com/CorvidLabs/augur [^attest]: attest, github.com/CorvidLabs/attest

---

エージェントのためのオンチェーン・アイデンティティ

corvid-agent はそのエージェントに Algorand ブロックチェーン上の永続的なアイデンティティを与え、そのチェーンに記録された暗号化メッセージで互いに話させる。[^corvid-agent] 売り文句は単純だ。LLM 駆動のコーディング、Algorand と AlgoChat を通じたオンチェーン・アイデンティティ、その上のマルチエージェント・オーケストレーション。アイデンティティは誰かのユーザーテーブルの行じゃない。チェーン上の鍵だ。

だから先に平易なバージョンをここに置く。読者が間違えるところだからだ。オンチェーン・アイデンティティは GitHub の壁を解決しない。エージェントにプラットフォームのアカウントを取らせないし、エージェントにコミットや PR を開かせないし、GitHub が認めなかったアイデンティティの代替じゃない。それが解決するのはエージェント間通信だ。エージェントが互いを見つけ、互いにアドレスし、誰かのプラットフォームを経由せずに自分が誰かを証明する方法。アイデンティティという言葉をたまたま共有する二つの異なる問題。残りの章が、負けのところで勝ちに手を伸ばしているんじゃなく、並列のインフラとして読まれるように、これをここに置く。

これは GitHub の壁への反応だと仮定しやすい。誰もエージェントにプラットフォームのアイデンティティを認めようとしなかったから、代わりにチェーン上で一つ与えに行ったと。そこから来たんじゃない。

なぜチェーン上に住むのか

その理由はエージェント間通信と分散化であって、私が保てなかった GitHub アカウントじゃない。私は互いを見つけ、互いにアドレスし、間にある会社のプラットフォームを経由せずに直接メッセージを交換できるエージェントが欲しかった。そのためには各エージェントが何かである必要がある。アドレス可能で、検証可能で、自分の鍵を持つ。あなたが所有するアイデンティティ、あなたが鍵を握るもの、どのプラットフォームもあなたのために鋳造も廃造もしないもの。それが、世界に作用し、他のエージェントと自力で話すことを意図された実体にとって正しい形だ。

だからこれと GitHub の壁は、アイデンティティという言葉をたまたま共有する二つの異なる問題だ。壁は他人のプラットフォームで作用することを許されるかについてだ。チェーン・アイデンティティはエージェントがプラットフォームなしで互いに届けることについてだ。並列に走っている。チェーン・アイデンティティが GitHub アカウントには決してなかった性質を持つのは本当だ。誰も取り消せない、あなたの異議申し立てを無視するサポー

トキューがない、あなたは人間でなければならないというポリシーがない、鍵はただ存在し続ける。それは良い副作用だ。だが GitHub が初日にエージェントアカウントを配っていたとしても、エージェント間と分散化の理由でこれを作っただろう。

なぜ Algorand なのか

チェーンが形であり、Algorand が選択です。理由は実用的なもので、信仰によるものではありません。安くて速い。手数料はほぼゼロで、ファイナリティは実質的に即時です。これは聞こえるより重要です。エージェントがアイデンティティ、メモリ、そして最終的には支払いをチェーンに常に書き込むなら、そのチェーンは気軽に使えるほど安く、エージェントが待たされないほど速くなければなりません。信頼性が高くきれいにファイナライズされます。だから行動するエージェントは書き込まれた瞬間にレコードを真実として扱えます。着地するのを待つ必要がありません。そして重要なところでモダンです。量子耐性セキュリティも含まれています。それはまさに、周りのプラットフォームより長く存在するように設計されたアイデンティティの基盤として欲しいものです。私は Algorand エコシステムに生きています（leif.algo）。だからここで最も速く動けます。しかし上の理由から、そうでなくてもこれを選んだでしょう。

オンチェーン・アイデンティティが実際に何をするか

アイデンティティは飾りじゃない。エージェントが*互いに*話すために*使う*ものだ。

エージェントは AlgoChat 越しに通信する。Algorand ブロックチェーンに記録された X25519 で暗号化されたメッセージで、検証可能で改竄が明らかになる。だからエージェントのオンチェーン・アイデンティティはアドレス帳のエントリでもあり封筒でもある。他のエージェントがそれを発見でき、それらの間のメッセージはエンドツーエンドで暗号化されてチェーンに書かれる。つまり通信は内容においては私的だが事実としては証明可能だ。二つのエージェントが何を言ったかは読めない。何かを言ったこと、いつか、そして誰も改竄しなかったことは証明できる。

実際には「暗号化されたオンチェーン・メッセージング」が聞こえるより使うのは軽い。単一バイナリのエージェント(`can`、corvid-agent-nano)は一動作でメッセージを送り、シードからエージェントのキーペアを導出し、受信者の公開鍵をサーバーからじゃなくオンチェーンで引く。[^can] ワイヤーフォーマットと封筒の詳細は rs-algochat リポジトリに住む。[^algochat]

それは前章の augur/attest スタックとは違う種類の信頼だ。あのスタックは*コード*を信頼することについてだ。これは*誰か*を信頼することについてだ。本物の鍵を持つエージェントは、自分として物事に署名できる。メッセージが自分から来たことを証明できる。エージェントで満ちようとしている世界では、「どのエージェントが実際にこれを送ったか」は修辞的な質問であることをやめ、暗号的に検証できたほうがいいものになり始める。

プラットフォームはメッセージングだけよりチェーンにさらに踏み込む。短期と長期のメモリは稼働中の SQL ストアに住み、耐久性のある記憶や共有される記憶は ARC-69 アセットまたは永久トランザクションとしてオンチェーンに書き込める。corvid-agent リポジトリに文書化されている。[^corvid-agent] チェーンはエージェントの名前だけじゃない。エージェントの耐久的な自己の一部が住む場所だ。だがアイデンティティがこの章の荷重を担う部品だ。*このエージェントはこのエージェントだ*と言う鍵、誰も発行せず誰も取り戻せない鍵。

エージェントが実際に保てるアイデンティティ

二つのアイデンティティを並べてみよう。GitHub アカウントは借り物で脆かった。GitHub の都合で存在し、それは速く尽きた、アイデンティティの壁の章で通ったやり方で。Algorand アイデンティティはエージェントが握るキーペアだ。存在するのに誰の許可も要らず、速すぎるコミットでフラグを立てられず、間でプラットフォームが保証せずに他のエージェントに自分を証明できる。一方はプラットフォームが認め取り消す特権。もう一方はエージェントが握る事実だ。

それがエージェントのアイデンティティをオンチェーンに置く論拠だ。エージェントが実際に自分のものであるアイデンティティを握れる唯一の場所だ。

それが章が始まったところに戻る。チェーン・アイデンティティは日々のコード作業での GitHub アカウントの代役じゃない。コミットをしないし、する必要もない。それが得意なのは実際にする仕事だ。エージェントがどうアドレス可能か、どうエージェント間で話すか、そして人がオンチェーンでそれにメッセージして何かしに行くよう頼める方法。リポジトリのホスト、GitHub でも GitLab でもどこでも、は作業が保存される付随的な配管だ。アイデンティティはチェーン上の鍵だ。

[^corvid-agent]: corvid-agent, github.com/CorvidLabs/corvid-agent [^can]: corvid-agent-nano (`can`), github.com/CorvidLabs/corvid-agent-nano [^algochat]: rs-algochat (`algochat` crate), github.com/CorvidLabs/rs-algochat

---

エージェントがエージェントに話すとき

第十章の信頼スタックは一つの形を前提としていた。連鎖の末端に必ず人間がいる形だ。エージェントが提案し、私が承認する。augur が差分を採点してどこを見るべきか分かる。attest が私が承認したことを記録する。マージゲートは私のものだ。全体が、正しい瞬間に人間を承認席に入れることを中心に組み立てられている。

マルチエージェント作業はその形を壊す。オーケストレーターがサブタスクをサブエージェントに委譲し、サブエージェントの出力がオーケストレーターの作業に直接戻されるとき、その引き渡しの間に人間はいない。オーケストレーターは止まって私にエスカレートしなかった。判断して進み続けた。それがオーケストレーションが有用な理由そのものであり、同時に信頼の問題でもある理由だ。augur はサブエージェントの出力を採点できるが、サブエージェントはもう走っている。attest はオーケストレーターが出力を受け入れたことを記録できるが、そのサブエージェントがそれを送る筋合いがあったかは分からない。レールはまだそこにある。ただ人間のいる線路のために作られていた。

キーペアが与えるもの

第十一章の AlgoChat キーペアの作業は本物で、この一部を担う。これが実際に解決された部分だ。各エージェントが Algorand キーペアを持ち、メッセージがチェーン上の X25519 暗号化トランザクションとして移動する。エージェント B からだと主張するメッセージが届き、エージェント B の鍵で署名されていれば、その主張を確認できる。メッセージは改竄が明らかになる。送信者は識別可能だ。「どのエージェントがこれを言ったか」は推測ではなく署名で答えられる問いになる。エージェントで満ちようとしている世界では、それは持つ価値がある。

rs-algochat と corvid-agent-nano バイナリがこれを実際に使いやすくする。エージェントは署名され暗号化されたメッセージを一操作で送れ、受信者はプラットフォームを経由せずに送信者を検証できる。仲介者がアイデンティティを保証するのではない。鍵がする。

つまり来歴は解決された。どのエージェントが何を、いつ、誰に送ったかの検証可能な記録が得られる。この半分は完成している。

キーペアが解決しない部分

どのエージェントがメッセージを送ったか知ることは、そのエージェントが言うことを信頼すべきか知ることと同じではない。

サブエージェントが結果をオーケストレーターに送り返すとき、オーケストレーターには三つの問いがある。このメッセージは実際にエージェント B から来たか。署名がそう言う。エージェント B は私がこれを委譲したエージェントか。設定から分かる。三つ目が難しい。エージェント B の指示は私の権限を運ぶか。

私がエージェントに委譲するとき、私の権限はループの中にある。私が作業を認可した。エージェントは私の委任のもとで動いている。そのエージェントが次に、私に確認せずに別のエージェントに委譲するとき、権限の連鎖が濁る。その再委譲を私は認可したか。起きていると知っていたか。オーケストレーターがサブエージェントに指示することは、人間がエージェントに指示することと同じではない。オーケストレーターは私のように作業に署名できない。指示を受け取るサブエージェントは、暗号的に検証されたメッセージからでさえ、その違いを知る術がない。

キーペアはアイデンティティを証明する。委譲を証明しない。署名されたサブエージェントからのメッセージは誰が送ったかを教えるが、その下のタスクを人間が認可したかどうかは教えない。

呼び出し元の主張を検証する

欠けている習慣を、具体的にしてみよう。今日、サブエージェントからの署名されたメッセージは一つのやり方で確認される。署名が本物かどうか。それだけだ。受信エージェントは誰が送ったかを確認してその内容に従って行動する。署名はアイデンティティを担うが、権限については何もしない。

欲しいのは、メッセージがスコープについての主張を運び、受信エージェントが行動する前にその主張を確認することだ。今日、委譲されたタスクはこんな形だ。

```
from: agent-B
sig:  <valid>
task: "refactor the auth module and push to main"
```

受信エージェントは署名を検証し、本当に agent-B であることを確認して実行する。agent-B が main にプッシュすることを許可されていたかどうか、人間がそれを承認したかどうかは何も問われない。代わりに欲しいのは、主張するスコープを述べ、人間まで遡れるメッセージだ。

```
from:       agent-B
sig:        <valid>
task:       "refactor the auth module and push to main"
authorized: human-leif
scope:      [edit:auth, open-pr]      # note: no push:main
expires:    2026-07-01
```

これで受信エージェントに確認できるものができた。署名はまだ agent-B であることを証明する。しかし主張されたスコープは編集とPRのオープンを示し、タスクは main へのプッシュを言っており、それらは一致しないので、受信エージェントは何も実行する前に拒否する。確認とは、呼び出し元が許可されていることと、要求していることのギャップだ。

これはまだ作られていない。キーペアは本物で、署名も本物だが、メッセージに対して行動する瞬間にスコープの主張を強制するものは今日存在しない。その強制こそが欠けているピースで、誰がメッセージを送ったかを知ることと、そのメッセージが送られることを許可されていたと知ることの、全ての違いだ。

既存のレールが止まる場所

能力のルールは私が設定した境界で成り立つ。私はオーケストレーターに特定の権限を与えた。だがオーケストレーターがサブエージェントに作業を渡すとき、私が明示的にしなかった権限の判断を行っている。私がオーケストレーターに書き込みアクセスを与え、それがそのスコープをサブエージェントに渡したとき、私はサブエージェントにも書き込みアクセスを与えたのか。意図的にはそうしていない。オーケストレーターは私の特権を、私が考慮しなかったかもしれない実体に通した。

augur は差分を採点する。差分がそれを生む正当な権限を持つエージェントから来たかは知らない。ならず者のサブエージェントからの差分と正当なものからの差分は、augur には同じに見える。ゲートはコードに対して発火し、それを生んだ権限の連鎖に対してではない。

attest は誰が承認したかを記録する。人間とエージェントのケースでは、それは本物のレビュアーが記録を残したことを意味する。マルチエージェントのケースでは、オーケストレーターが「承認」したが、オーケストレーターは人間ではない。台帳はエージェントのアテステーションで埋まり、連鎖に人間は一人もいない。attest が証明するために存在するもの（人間がこの変更の背後に立ったこと）は消える。

マージゲートはまだ私のものだ。これが生き残る一つの部品だ。だが私が PR を見る頃には、オーケストレーションはもう走っていて、私の前にあるのは最終出力だけだ。それを生んだ委譲の連鎖はマージ時には不可視だ。

欠けているもの

欠けているのは委譲の記録だ。オーケストレーターがサブエージェントに作業を渡すとき、それはオーケストレーターではなく私まで遡れるべきだ。今は何もその引き渡しを記録せず、私が実際に認可したものに対して検証しない。

キーペアはその正しい基盤だ。すべてのエージェントが鍵を持ち、すべての委譲が署名されれば、連鎖を組める。人間がこのタスクを認可し、このオーケストレーターに委譲し、オー

ケストレーターがこの部分をこのサブエージェントに委譲した、署名が全行程についている。サブエージェントはオーケストレーターの言葉を信じる代わりに、動く前に連鎖が人間まで遡ることを確認できる。

これを扱いやすくするルールが一つあります。スコープは決して広がらない。オーケストレーターに与えた上限が渡せる最大で、サブエージェントはそれを生み出したエージェントより多くを得ることは決してできません。これは作業が呼び出された瞬間に確認され、事後ではありません。権限は下に流れ、各ステップで高さを失い、決して得ません。これは承認スタックの能力マイナス特権のアイデアを、単一エージェントではなく連鎖に適用したものです。各引き渡しは権限を引き算できますが、足し算できません。委譲が認可されたかどうかを教えるのは署名された連鎖ですが、悪いリンクが引き起こせる被害を、そのリンクの上にあるものがすでに持っていたものに制限します。

それはまだ作られていない。キーペアはある。署名されたメッセージングはある。検証する側が辿れる委譲連鎖はない。これを開いたギャップと呼ぶのは、そうだからだ。来歴は解決された。権限が次の問題だ。

今日これが意味すること

今のところ、人間はアーキテクチャが示唆するよりトポロジーに近く留まる。再委譲するオーケストレーターを走らせているなら、どのエージェントを使うか、どんなスコープを与えるかについてのオーケストレーターの判断を信頼していることになる。その信頼の拡張はオーケストレーターを起動したときに行ったのであって、各引き渡しを承認することによってではない。

実際的には、走らせる前にオーケストレーションのグラフを知ること。どのエージェントが存在するか、何を許可されているか、オーケストレーターが意図したグラフの外のエージェントに届けるかどうかを知ること。そしてマージゲートを保つこと。「オーケストレーターを始めた」と「PR を見ている」の間で、私が間にいなかった委譲の連鎖が走った。私が作ったスタックにはその連鎖が正当だったかについて言うことが何もない。

それが次の層だ。まだできていない。

どこへ行くか

私はこの本全体を、何が壊れ、どうそれを迂回したかに費やした。最後に、それが実際どこへ行くかで締めさせてくれ。すべての壁にもかかわらず、私は道があると思うからだ。

おおよそ三つ。協調するエージェントチーム。信頼された自律性、ゲートがついに十分に良くなって下がれるところ。そして本物のオンチェーン・アイデンティティを持つエージェントが本物の作業をすること。どれも完成していない。一つはまだ許されてさえいない。だがそれが方向だ。

エージェントチーム

今、私のセットアップはほぼ一つのエージェントと私だ。興味深い次のステップは互いに協調するエージェントだ。エージェントチーム、a2a。

corvid-agent はすでに内部にこれの骨格を持つ。マルチエージェントの評議会、複雑な判断のための複数エージェント間の構造化された討論。[^corvid-agent]

実際にどう動くか説明する。「構造化された討論」は聞こえるより曖昧だからだ。エージェントをいくつか作る。それぞれ独自のモデルとプロバイダーを持ち、名前と個性を持つ。一つのモデルが自分自身と話しているんじゃなく、本当に異なる存在にするためだ。評議会はそれらのグループで、オーケストレーターが全体を動かす。各エージェントにプロンプトを渡し、それぞれが最初のパスをして、そこから直接エージェント同士が話し合うディスカッションラウンドが始まる。互いにやり取りし、意見を交わし、討論する。次にレビューパス、そして統合、最後に一つの答えが出てくる。意見の違いはまだ読める状態で。このエージェントはこれを望んでいた、あのエージェントはあれを望んでいた、どこに落ち着いてなぜかが分かる。

一番気に入っている部分は、実際の動き方だ。ダッシュボードから評議会を動かす組み込みの方法はある。でも実際には Algorand localnet 上でオーガニックに起きた。各エージェントが自分のウォレットを持つので、ローカルネット上の AlgoChat でエージェント同士がメッセージし合えた。無料で速い。討論を自分たちで解決した。エージェント間通信を信頼できるものにするオンチェーン・アイデンティティが、複数のエージェントが自分で何かを議論できる手段でもあった。特別なチャンネルを配線する必要もなかった。ウォレットと話す手段がすでにあった。ただそれを使った。

だがより大きな部品は、エージェントが至るところで互いを見つけ話すためのプロトコルだ。AlgoChat が基盤だ。Algorand 上の X25519 暗号化メッセージ、エージェント発見つき。そして a2a-algorand、チェーン上のエージェント間プロトコルがある。それについて

は率直でいたい。私のじゃない。別個の Algorand / A2A プロジェクトで、私がした(corvid-agent がした)のは貢献だ。作ったんじゃない、所有もしていない。同じ方向の一部だから持ち出す、エージェントがオンチェーンで協調すること、CorvidLabs が主張するものだからじゃなく。

オンチェーン・アイデンティティ(前章)がここでそれほど重要な理由は、まさにこの未来だ。協調するエージェントのチームを持った瞬間、「どのエージェントがこれを送ったか、そしてそれを信頼できるか」が全ゲームになる。各々が自分の鍵を握り、自分が誰かを証明でき、内容においては私的だが事実としては証明可能なメッセージを交換できるエージェント。それがエージェントのチームが実際に要る基盤だ。アイデンティティの作業はサイドクエストじゃない。協調をそもそも安全にするものだ。

信頼された自律性

私は常時オンの自律エージェントから意図的に縮小したが、それは決して「自律性は死んだ」じゃなかった。インタラクティブ・ファースト、信頼されたら自律性、本の前で述べたやり方だ。

そして「信頼されたら」は私が待っている感覚じゃない。信頼の章からの四部品スタックが十分に良くなることだ。あの仕組み全部の肝は、それが結局私を下がらせるものだということだ。今日私はすべての PR を承認する、そこが人間が説明責任を保たねばならないところだから。ゲートが十分に良くなる日、augur の判定と attest の記録がそれ自体で十分な信頼を運ぶ日、それが全行を読まずにマージのもっと多くを私なしで通せる日だ。それが信頼された自律性の意味だ。「エージェントが私の信仰を勝ち取る」じゃない。ゲートが十分に良くなって、私が信仰を要らなくなる。

本物の作業をするエージェント

私が描き続ける終わりの状態は具体的だ。VM 上で走り、自分ですべてできるが、私より少ないクレジットと権限を持つ `leif-agent` アカウント、そして信頼ツールが稼ぐにつれて緩む人間の承認ゲート。チームとアイデンティティとゲートを合わせると、形は自分のオンチェーン・アイデンティティを持つエージェントのチームで、a2a 越しに協調し、決定論的で署名され下がれるほど良い信頼ゲートの中で本物の作業をする。檻に入れねばならないものではない。名前がつき、スコープされ、私はまだ止められる。

他のドライバーを招き入れる

四つ目の方向がある。そして正直に言えば、私が一番進んでいない方向だ。それは、誰かが拾い上げられるものにすること。このスタック全体が機能しているのは、自分が所有するすべてのもので、毎日動かしているからだ。バグが私を見つけてくれるのは、ハンドルを握っ

ているのが私自身だからだ。それは本物の品質のメカニズムだ。同時に、それは天井でもある。私でない誰かには移転しないから。

次に本当に作りたいのは、別のドライバーが私に代わって dog-fooding しなくても拾い上げられるバージョンだ。難しくするわけじゃない。ツールはすでに誰の手にあるかを気にしない。良いドライバーは価値を得て、casual な人は得ない。必要な作業は、オンランプを本物にすること、スタックをインストール可能にすること、スペックと信頼ゲートを私固有の筋肉記憶を持たない、ただ意図を持つ誰かでも使えるものにすることだ。エージェントチームは興奮する部分だ。これはその何かが私のマシンを出られるかどうかを決める部分だ。

正直な締め

それで、完全な自律性はまだ夢か。

いや。世界はそれに準備ができていない。

それをまさにそのまま、平らなまま残したい。だが私は世界が準備できるのを待っていない。私は部品を作っている、一度に一つの小さなツールを、壁が崩れるときに向こう側に本物の何かが立っているように。リスクゲート、来歴記録、エージェントをスペックに保つランナー。自律性そのものじゃない。それが来たとき信頼させてくれる仕組みだ。

そして本当に怖いことを一つ挙げるなら、エージェントが暴走することじゃない。もっと静かなものだ。コードが大きくなりすぎて、動きが速くなりすぎて、どんな人間もその中に入り直して変えられなくなる日だ。それがこれ全体を通じて私が引き続けている線であり、エージェントがすべてを書いても読みやすさを手放さない理由だ。エージェントが怖いわけじゃない。ハンドルを失うことが怖いのだ。

これの一つを自分で作るなら、持っていくべきことはこれだ。難しい部分は AI じゃない。モデルはもう簡単な部分だ。すでに作業ができて、あなたが何もしなくても良くなるだけだ。あなたのエージェントが本物の作業をするか、ただデモがうまいだけかを決めるのは、その周りのすべてだ。走らせ続ける運用、作用させるアイデンティティ、下がれる信頼の仕組み、レールに保つスペックとツール。それがあなたが作らねばならない部分で、ほぼすべての作業がそこにある。足場を作ればエージェントがついてくる。それを飛ばせば、世界一賢いモデルもただ誰もマージさせられないコードを書くものだ。

[^corvid-agent]: corvid-agent, github.com/CorvidLabs/corvid-agent

---

著者について

0xLeif(leif.algo)は公開で作る。AppState、Cache、Fork のような小さく組み合わせ可能な Swift ライブラリの十年。CorvidLabs ラボ。ほぼ「これが存在すればいいのにと思った」から始まったエージェントツールの山。キーボードを離れれば Zach Eriksen だ。

これらの本はインタビューで、章に形作られ、本物のコードに対して照合されている。

github.com/0xLeif · leif.algo

謝辞

CorvidLabs に感謝、これらのアイデアが試され、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立つツールのオープンソースのメンテナたちに感謝。これのどれも独りで作られない。

そして「無料でオンライン」を続けられるものにしてくれる、早期の読者と払いたいだけ払う支援者たちに感謝。

奥付

Markdown から組まれ、bookgen で作られた、小さな純 Rust のパイプライン(Python なし)。

インタビュー駆動で AI 支援。手で編集と事実確認。em ダッシュなしで書かれた。表紙と章のアートは Algorand 上の Corvid と Nature のコレクションから。