Open Source Tooling

Construindo ferramentas que as pessoas realmente usam

ZACH "LEIF" ERIKSEN

Direitos Autorais

Disponível gratuitamente para leitura online. O ePub é pague quanto quiser; se ele te ajudou, você pode apoiar o trabalho.

github.com/0xLeif · leif.algo

Um dos quatro livros da coleção agent-stack. Como foi feito está no colofão ao final.

Dedicatória

Para todos que constroem em aberto e publicam mesmo assim.

A Biblioteca

Estes livros são independentes, mas foram escritos como um conjunto. O código ficou barato e a confiança ficou escassa. Juntos, formam um único argumento: o que construir agora e como confiar nisso.

- **The Agent Developer's Field Guide**: Construindo ferramentas, especificações e confiança para agentes que publicam código de verdade
- **First-Class**: Construindo para humanos e agentes igualmente
- **Building Agents**: Notas de uma tentativa de dar mãos próprias ao software
- **Open Source Tooling**: Construindo ferramentas que as pessoas realmente usam *(este livro)*

Disponível gratuitamente para leitura online. Cada ePub é pague quanto quiser.

Sumário

Introdução

Este livro é sobre o ofício de construir ferramentas que outras pessoas, e outros agentes, realmente usam.

Ele é construído em torno do fledge, uma ferramenta de linha de comando que cobre todo o ciclo de vida do desenvolvimento, e da pergunta que ele me forçava a responder continuamente: o que faz uma ferramenta valer a pena como dependência? A afirmação é que a resposta é a mesma para um humano e para um agente. Uma boa ferramenta tem uma superfície limpa e única, nenhuma etapa interativa oculta, uma fronteira real de plugins e uma especificação que permanece honesta conforme o código muda. Construa isso, e um agente consegue usá-la desde o primeiro dia, porque ela nunca foi construída para precisar de uma pessoa.

Este é o segundo dos dois livros de evidências. *First-Class* argumenta que o software deve ser de primeira classe para ambos. O *Field Guide* transforma isso em método. *Building Agents* mostra os agentes. Este mostra as ferramentas por baixo deles, até o porquê Rust foi a escolha certa e como o sandbox WASM impede que um plugin faça algo que não deveria.

É para qualquer pessoa que já publicou uma ferramenta e a viu ser usada de maneiras que não planejou. Você não precisa do fledge. Você precisa dos hábitos que fazem uma ferramenta sobreviver ao contato com usuários que não são você, incluindo aqueles que nem são humanos.

Uma CLI para todo o ciclo de vida

Todo repositório tinha um dialeto diferente. Makefiles diferentes, scripts diferentes, READMEs diferentes, cada um com sua própria ideia de como você constrói, testa e executa a coisa. Nada disso era transferível. Você abria um projeto que não tocava fazia um tempo e o primeiro trabalho era descobrir a incantação local de novo. Qual script, qual alvo, qual ordem. O trabalho em si não era difícil. Executar testes não é difícil. O problema era que era diferente em todo lugar, e a diferença era custo puro.

Então eu queria uma interface consistente em todos eles. É aí que o fledge começa. A frase no repositório diz claramente: *uma CLI, todo o seu ciclo de vida de desenvolvimento.*

Havia realmente três coisas me empurrando para isso.

A primeira era aquele problema da uniformidade, cada repositório falando seu próprio idioma. A segunda era a inicialização. Eu ficava copiando e colando a mesma configuração e scaffolding em cada novo projeto. A mesma pilha de arquivos, a mesma ligação, de novo e de novo, antes mesmo de poder começar na coisa real que queria construir. A terceira era escala. Estou iniciando toneladas de projetos, com agentes trabalhando neles, e precisava de uma única CLI com a qual todos pudessem contar. Não "uma ferramenta que uso". Uma ferramenta na qual tudo e todos trabalhando em todos esses repositórios pudessem se apoiar da mesma forma.

Esse último é fácil de ignorar. O fledge não nasceu de um repositório que me irritou. Ele nasceu de ter muitos repositórios, muitos projetos em andamento, e agentes participando deles, e precisar que tudo isso conversasse com uma superfície consistente em vez de uma centena de superfícies feitas sob medida. O ângulo de agentes-podem-depender-de-uma-CLI é seu próprio fio, e entro no lado dos agentes no livro sobre agentes. Consistência, scaffolding e escala: essa é a origem.

Construído para humanos e agentes igualmente

Há uma ideia por trás de tudo isso, e ela é maior que o fledge. Muito do meu ferramental vem da crença de que humanos e agentes vão usar as mesmas ferramentas.

Hoje, a maior parte do que existe é feita primeiro para humanos. Projetos são construídos para pessoas, e depois tentamos trazer agentes para eles após o fato.

Pode haver coisas primeiro para agentes e coisas primeiro para humanos, mas o que realmente precisamos é criar projetos que sejam primeiro para agentes e humanos, construídos desde o início para que ambos sejam de primeira classe.

É isso que todas as minhas ferramentas são. Uma ferramenta deve funcionar de primeira classe de qualquer forma: um humano pode usá-la sem um agente, e um agente pode usá-la sem um humano. Nenhum dos dois é um pensamento tardio. E quando um agente a usa, a ferramenta deve *ajudá-lo*, não deixá-lo adivinhando todos os comandos e como tudo funciona.

Projetar para humanos e agentes como chamadores igualmente de primeira classe é a razão real por que o fledge tem a aparência que tem, e é a coisa a se ter em mente no restante deste livro.

O que zero-config realmente significa

Quando digo zero-config, quero dizer que você não precisa ensinar a ferramenta sobre seu projeto antes que ela possa te ajudar. Você a coloca em um repositório e ela funciona. Há três partes nisso.

Ela detecta automaticamente o projeto e seus comandos. Swift, Rust, Node, o que for. Ela descobre que tipo de projeto está observando e conhece o build/test/run correto para ele. Sem etapa de configuração onde você descreve seu projeto para um arquivo de config primeiro.

Os mesmos verbos funcionam em todos os projetos. `build`, `test`, `run`, `lint`, as mesmas palavras, independentemente do que está por baixo. Esse é o benefício direto do problema "cada repositório tinha um dialeto diferente". Os dialetos ainda estão lá embaixo; Cargo ainda é Cargo e Swift Package Manager ainda é Swift Package Manager. Mas você para de precisar se importar em qual deles está. Você aprende os verbos uma vez e eles são os verbos em todo lugar. A detecção cuida da tradução para qualquer ferramenta real que está por baixo.

Esta é a ideia de humanos-e-agentes tornada concreta. Uma pessoa para de precisar reaprender cada repositório. Um agente para de precisar *adivinhar*. Ele não precisa ir procurar o comando certo para este projeto específico, ou ler o README e torcer. Ele executa `build`, executa `test`, e a ferramenta já sabe o que isso significa aqui. A coisa que me poupa de reaprender é a mesma coisa que impede o agente de adivinhar. O fledge foi construído primeiro para mim e para os agentes que executo, e ganha seu lugar lá antes de qualquer outro lugar.

E é extensível via plugins. Um repositório novo recebe o núcleo, e você adiciona capacidades colocando plugins. O núcleo conhece os verbos comuns e como detectar os tipos de projeto comuns; tudo além disso vem de plugins. Então zero-config não significa "faz tudo fora da caixa". Significa que a parte que você recebe fora da caixa não precisa de configuração, e você a expande a partir daí adicionando plugins, não escrevendo config.

Concretamente, uma primeira execução é assim. Você entra em um repositório e executa `fledge`, e ele detecta automaticamente o projeto (Swift, Rust, Node, o que for) e simplesmente conhece o build/test/run correto para ele, sem etapa de configuração. Peça para fazer introspecção e ele *informa os verbos disponíveis* para este repositório: ele se descreve, então você (ou um agente) não precisa adivinhar o que é possível. Se é um projeto totalmente novo, o primeiro movimento real geralmente é scaffolding a partir de um template em vez de detectar um existente. E tudo funciona sem cabeça desde o primeiro comando: defina `FLEDGE_NON_INTERACTIVE`, peça por `--json`, e um agente o conduz desde o passo um. Detectar, introspectar, talvez fazer scaffold e então executar.

A parte do ciclo de vida

O repositório chama o fledge de um executor de tarefas zero-config com scaffolding e revisão por IA. A execução de tarefas é o conjunto de verbos acima: o build/test/run/lint do dia a dia. O scaffolding é a resposta para o segundo problema de origem: em vez de copiar e colar a mesma configuração em cada novo projeto à mão, o fledge o monta. E há uma parte de revisão por IA integrada ao ciclo de vida também.

A execução de tarefas veio primeiro. É a semente, a parte em que todo o resto cresceu. O scaffolding são realmente templates: montar um novo projeto a partir de um em vez de copiá-lo e colá-lo. E a revisão por IA foi integrada porque revisão faz parte do loop de desenvolvimento. Se agentes estão escrevendo o código, avaliá-lo não é um ritual separado que você vai executar em outro lugar, é apenas mais um verbo ao lado de build e test. Uma superfície vence três ferramentas, para mim e ainda mais para um agente que de outra forma teria que aprender três coisas. Esses três são os pilares estruturais, o núcleo, com plugins como a camada de extensão ao redor deles.

O próprio one-liner do fledge é "uma CLI, todo o seu ciclo de vida de desenvolvimento, executor de tarefas zero-config, scaffolding de projetos, revisão por IA e mais", e isso é exatamente as três partes. É um único binário Rust, e você o obtém da maneira óbvia e chata: `brew install CorvidLabs/tap/fledge` no tap do

Homebrew, ou `cargo install fledge` se preferir, ou um script shell. Nada exótico para instalá-lo.

Como eu realmente uso

Esta não é uma ferramenta que uso às vezes. É todo repositório, o tempo todo, a forma padrão como construo, testo e executo tudo agora. E meus agentes a conduzem. Os agentes executam o fledge mais do que eu faço manualmente. Ele foi construído para domar muitos repositórios, e agora é a única CLI na qual tanto eu quanto os agentes que tenho trabalhando nesses repositórios dependemos, em todo lugar.

É também por isso que a próxima parte importa. Uma CLI na qual todos se apoiam, que qualquer pessoa pode estender, que agentes instalam e executam, que não pode ser um conjunto fixo de recursos que eu aprovo um por um. Ela precisa ser extensível por qualquer pessoa, em qualquer linguagem que quiserem, sem tocar no núcleo. E uma vez que você permite plugins arbitrários, e uma vez que agentes são os que os executam, você precisa pensar seriamente sobre confiança. Esse é o próximo capítulo.

Por que fledge e não Make

As pessoas perguntam isso no segundo em que ouvem o que o fledge faz. Você construiu um executor de tarefas? Existe o Make. Existe o Just. Existe o Turborepo. Existe o bloco de scripts npm bem ali em todo package.json. Por que escrever mais um?

A resposta honesta é que a parede que encontrei nunca foi o Make. Comecei como desenvolvedor iOS, e para automatizar uma build lá o caminho sempre levava ao mesmo lugar: fastlane, o que significa Ruby. Eu não queria Ruby. Sou uma pessoa de Swift. Queria automatizar minhas builds na linguagem em que trabalho, e em vez disso todo o ecossistema me afunilou em uma DSL Ruby e um `Gemfile` e uma ferramenta que era seu próprio mundinho para aprender. Toda vez que queria publicar algo, a resposta era "escreva um lane no fastlane", e cada lane do fastlane era Ruby que eu não queria escrever, rodando em um runtime que eu não queria gerenciar, fazendo um trabalho que eu sentia que deveria ser capaz de fazer em Swift. Essa é a dor da qual o fledge realmente cresceu. Não "Make é desajeitado". Era "por que sou forçado para a linguagem de outra pessoa para automatizar meu próprio projeto?"

Então quando as pessoas colocam o fledge ao lado do Make e do Just, eles estão mirando no alvo errado. A resposta não é que essas ferramentas são ruins. O que eu queria de um executor de tarefas eram três coisas, e a parede do fastlane é de onde as três vieram.

Eu queria do meu jeito, na minha linguagem, nos meus termos, não preso a um runtime da forma como o fastlane te prende ao Ruby. Um executor de tarefas é a coisa que você toca cem vezes por dia, e depois de um tempo você para de querer viver dentro das escolhas de outra pessoa sobre como deve ser, ou em qual linguagem você tem que pensar para usá-lo. Eu queria que `build` e `test` e `run` significassem a mesma coisa em todo repositório, e queria que as etapas por baixo fossem escritas em qualquer linguagem que o trabalho exigisse em vez de ser marchado para o Ruby. O Make não te impede, mas o Make também não te dá isso. Você constrói a consistência você mesmo, em cada Makefile, à mão, para sempre. Uma ferramenta que me deixa reinventar o dialeto por projeto não resolveu meu problema; ela apenas me deu um lugar mais bonito para continuar reinventando.

Os outros dois são os do primeiro capítulo, e o arquivo do fastlane é onde eu também os teria querido. Nenhuma dessas ferramentas é primeiro para agentes (JSON por padrão, introspecção, um modo sem cabeça), que é a parte que mais me importa agora que meus agentes conduzem isso mais do que eu faço manualmente. E nenhuma delas é um único binário leve que cai em um repositório sem runtime para instalar primeiro, o que é exatamente o que uma superfície consistente em um monte de repositórios de linguagens diferentes precisa ser. Um lane do fastlane precisa do Ruby; scripts npm precisam do npm; um executor que eu escrevesse em TypeScript precisaria de um runtime TypeScript. O fledge não precisa de nada.

Nada disso é "Make é ruim". Make é ótimo no que faz, Just é um executor de comandos limpo, Turborepo faz cache de um monorepo JS bem. Se um desses é tudo que você precisa, use-o. Não vou fingir que o fledge vence uma comparação recurso a recurso no território deles.

Mas nenhum deles era o que eu realmente teria querido ao estar diante de um arquivo do fastlane anos atrás: uma superfície, primeiro para agentes, um único binário leve, e a liberdade de escrever as etapas em qualquer linguagem que se encaixasse em vez de ser afunilado para o Ruby. O fledge é a ferramenta que eu desejei ter naquela época, finalmente construída.

---

fledge e MCP

MCP é o padrão do ambiente agora. Em 2026, agentes consomem ferramentas principalmente por meio de servidores MCP. Se você constrói uma ferramenta e quer que agentes consigam usá-la de forma confiável, o caminho de menor resistência é expor um servidor MCP. Esse é simplesmente o estado do ecossistema.

O fledge não tem um servidor MCP. E mesmo assim agentes o conduzem constantemente em dezenas de repositórios, e funciona. O motivo vale ser explicitado, porque conta algo sobre qual parte construir primeiro.

A CLI é o primitivo

Um servidor MCP é uma interface de produção. Ele lida com roteamento de requisições, faz logs, oferece observabilidade estruturada sobre o que um agente pediu e o que recebeu de volta. Essas são coisas boas de se ter. Mas um servidor MCP é uma camada que você coloca em cima de algo. A questão é: em cima do quê?

Se você constrói o servidor MCP primeiro e a CLI é um pensamento tardio, você tem uma ferramenta que funciona para agentes e é uma dor para humanos. Se você constrói a CLI primeiro, você tem uma ferramenta que funciona para agentes agora sem adaptador de protocolo, funciona para humanos, e pode ter um servidor MCP colocado na frente dela quando precisar. A CLI é o primitivo. Todo o resto fica em cima dela.

O fledge foi construído com isso em mente. Cada comando emite `--json`. Existe `FLEDGE_NON_INTERACTIVE` para execução sem cabeça. `fledge introspect` dá a qualquer chamador, humano ou agente, um manifesto estruturado de cada verbo disponível, o que ele faz e o que aceita. Não há prompts interativos que bloqueiem execuções desatendidas. A ferramenta se descreve.

Esse perfil, saída estruturada por padrão, um manifesto explícito de capacidades, nenhuma etapa interativa oculta, é exatamente o que uma superfície de ferramentas MCP oferece a um agente. O fledge tem tudo isso sem o protocolo, porque essas propriedades vieram de ser projetado para chamadores agentes desde o início, não de empacotar a ferramenta depois para torná-la amigável a agentes.

Uma instância do Claude conduzindo o fledge hoje chama `fledge introspect`, recebe de volta um manifesto JSON do que está disponível, escolhe o verbo certo, passa --

`json` e lê a saída estruturada. Esse é o mesmo loop de interação que um servidor MCP mediaria, menos o enquadramento MCP. A CLI já é a superfície de ferramenta limpa.

MCP não é o concorrente

O enquadramento que às vezes surge, "CLI ou MCP?", os trata como alternativas. Eles não são. MCP é uma especificação de transporte e protocolo. A CLI é a coisa que faz o trabalho. A questão não é qual ter; é qual construir primeiro e qual colocar em cima.

Construa a CLI primeiro, o tipo que se descreve, emite JSON e funciona sem cabeça. Uma vez que você tem isso, expor um servidor MCP em cima é simples: você mapeia cada entrada de `fledge introspect` para uma definição de ferramenta MCP, você chama a CLI por baixo, você passa a saída JSON de volta. O núcleo já é JSON estruturado sobre uma fronteira limpa com um manifesto explícito de capacidades. Um servidor MCP é uma API de produção em cima desse mesmo primitivo, da mesma forma que uma API REST pode ficar na frente de uma biblioteca bem estruturada. A interface muda; o trabalho não.

E porque a CLI é a superfície real, tudo que pode chamar um subprocesso pode usá-la sem o adaptador de protocolo. Um script shell pode usá-la. Um executor de CI pode usá-la. Um humano em um terminal pode usá-la. Um cliente MCP pode usá-la pela camada do servidor. Nenhum desses chamadores requer que os outros existam. Você obtém universalidade do primitivo, não do protocolo.

O que MCP adiciona

Nada disso é um argumento contra MCP. Ele adiciona coisas reais quando está na frente da CLI.

Logging e observabilidade. Um servidor MCP fica entre o agente e a ferramenta, então você pode registrar cada chamada, medir o tempo, inspecionar argumentos, sinalizar anomalias. A CLI invocada diretamente te dá o que você canaliza para um arquivo de log. A camada MCP te dá observabilidade estruturada sem instrumentar cada comando individualmente. Para uso em produção onde você quer auditar o que um agente pediu, isso importa.

Descoberta no nível do protocolo. O MCP tem uma maneira padronizada para um agente perguntar "quais ferramentas estão aqui?" e receber de volta uma resposta estruturada. O fledge tem `fledge introspect` para a mesma coisa, mas `fledge introspect` requer saber que o fledge está lá. Um registro MCP deixa um agente

descobrir a ferramenta por meio de um handshake padrão antes de saber qualquer coisa sobre a ferramenta por baixo.

Composição entre ferramentas. Um servidor MCP pode expor várias ferramentas subjacentes por meio de um endpoint, então um agente tem um lugar para se conectar em vez de precisar conhecer uma lista de CLIs individuais. Essa é uma conveniência operacional quando o número de ferramentas é grande.

Esses são argumentos de infraestrutura de produção. Eles se aplicam quando você está executando agentes em escala, auditando seu comportamento e conectando-os a uma superfície ampla de ferramentas por meio de uma camada gerenciada. O servidor MCP é, como rótulo para o que é, uma API voltada para IA com logging. Isso é uma coisa útil de se ter. Simplesmente não é o que você constrói primeiro.

A ordem das operações

CLI primeiro. Funciona para cada chamador no momento em que existe. O servidor MCP é o invólucro que você adiciona quando o logging e a padronização do protocolo valem a camada, não antes.

Para o fledge, a CLI é a fundação sobre a qual um humano executando comandos, um agente conduzindo um repositório e um servidor MCP falando com clientes downstream se sentam. A história dos agentes não é "agentes usam o fledge por MCP". É "agentes usam o fledge da mesma forma que tudo mais o faz, porque a CLI foi construída de primeira classe para qualquer chamador".

Quando um servidor MCP para o fledge existir, será uma camada fina. O trabalho já está feito no núcleo. Esse é o ponto de construir o primitivo primeiro.

Scaffolding e templates

A segunda coisa que me empurrou em direção ao fledge, depois do problema da uniformidade, foi a inicialização: a dor de copiar e colar a mesma configuração que mencionei no capítulo um. Veja como isso parece de perto. Você decide criar uma nova CLI em Rust e a primeira hora não é a CLI. É o layout do Cargo, a configuração, a configuração de lint, o CI, o esqueleto do README, toda a coisa que você já digitou vinte vezes antes. Esse é o imposto, e eu o estava pagando constantemente porque estou iniciando muitos projetos.

Então scaffolding é um pilar, não um recurso secundário. Montar um projeto do zero faz parte do ciclo de vida tanto quanto construí-lo e testá-lo. A ideia toda é: montar um novo projeto a partir de um template em vez de copiá-lo e colá-lo à mão.

No fledge isso fica em `fledge templates`. Você executa `fledge templates init` com um nome e um template e ele monta o projeto para você:

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

Há um conjunto integrado cobrindo os tipos de projetos que realmente inicio. Os que são publicados são `rust-cli`, `ts-bun`, `python-cli`, `go-cli`, `ts-node`, `static-site`, `kotlin-kmp` e `kotlin-ktor-api`. Essa lista é basicamente um mapa das linguagens em que trabalho: uma CLI em Rust, alguns sabores de TypeScript, uma CLI em Python, uma CLI em Go, um site estático e os de Kotlin Multiplatform e Ktor API. Esses oito são o conjunto integrado completo no repositório hoje. O lado de detecção do fledge sabe lidar com Swift, Rust, Node e o resto uma vez que um projeto existe; o lado de templates é como um projeto passa a existir em primeiro lugar.

A parte que mais me importa é que templates não são uma lista fechada que preciso aprovar um por um. Você pode fazer scaffold a partir de qualquer repositório do GitHub, não apenas dos integrados. `--template user/repo` e ele puxa o template de lá:

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

O núcleo publica um conjunto pequeno e útil, e além disso você o estende sem que eu precise estar no circuito. Um template é apenas uma forma de projeto que alguém já descobriu, e se ele vive em um repositório do GitHub, o fledge pode montar um novo projeto a partir dele. Então o conjunto de templates não é "o que o CorvidLabs publicou". É "todo ponto de partida que alguém já colocou em um repositório".

E há um conjunto completo de verbos em torno de templates, não apenas `init`. Há `fledge templates create` para criar um, `fledge templates list` e `fledge templates search` para encontrá-los, e `fledge templates validate` para verificar que um template é realmente bem formado antes de você depender dele. Se vou fazer scaffold a partir de um template, e especialmente se um agente vai, quero saber que o template se sustenta antes de ele carimbá-lo em cem projetos com a mesma coisa quebrada integrada.

Ele verifica mais do que "faz parsing". Ele lê o manifesto `template.toml` do template, confirma que o nome e a descrição não estão vazios, e verifica que quaisquer ferramentas que o template diz que requer estão realmente no seu PATH. Em seguida, percorre cada arquivo e cada nome de arquivo no template e executa o templating de espaços reservados sobre eles, então um espaço reservado ruim (um erro de sintaxe, ou uma variável que o manifesto nunca define) é detectado como erro antes de você fazer scaffold a partir dele. Ele também sinaliza um template sem arquivos e avisa se o manifesto não está configurado para ser excluído da saída. Então é estrutural, mas vai além de "os arquivos estão lá": ele realmente exercita o templating da forma como `init` fará, e diz onde quebra.

Que é o motivo real pelo qual scaffolding pertence a esta CLI e não em alguma ferramenta geradora separada ao lado. Mesmo motivo que tudo mais no fledge: é a mesma superfície, para os mesmos dois tipos de usuários. Uma pessoa executa `fledge templates init` e pula a hora tediosa. Um agente executa o exato mesmo comando, sem interatividade, e monta um projeto novo desde o passo um sem um humano clicando em um assistente. Do primeiro capítulo, detectar, introspectar, talvez fazer scaffold e então executar, o passo de scaffold é o "talvez". Quando o repositório já existe, o fledge o detecta. Quando ele ainda não existe, o primeiro movimento real é montá-lo a partir de um template, e então tudo mais, o verbo de build, o verbo de test, o verbo de review, funciona nele imediatamente, porque ele saiu do template já conectado da forma que o fledge espera.

Um projeto que já fala fledge desde o nascimento é o que eu realmente buscava. Não apenas "me poupa de copiar e colar", embora faça isso. Um projeto com scaffold constrói, testa e está pronto para os mesmos verbos que todo outro repositório, desde o primeiro commit. O boilerplate que eu costumava colar à mão era, metade das vezes, exatamente a ligação que tornava um projeto consistente com os outros. Integrar scaffolding à CLI do ciclo de vida significa que essa consistência é o padrão com o qual um projeto nasce, não algo que eu parafuso depois.

Então sim, um novo projeto começa com o scaffold. E mesmo quando não uso `fledge templates init` pelo nome, o projeto recebe a mesma configuração conectada desde o

primeiro commit de qualquer forma: spec-sync, fledge, augur, attest. O "init" real não é o template, é a pilha sendo inserida. O comando é apenas a forma rápida de chegar lá. De qualquer caminho, um novo projeto meu nasce já carregando as ferramentas que todos os outros têm.

Revisão por IA no fluxo

O terceiro pilar é o que surpreende as pessoas ao encontrar em um executor de tarefas. Execução de tarefas, scaffolding, claro, essas são obviamente coisas do ciclo de vida. Mas revisão de código por IA? Na mesma CLI com que você constrói e testa? Parece que deveria ficar em outro lugar, em sua própria ferramenta, em CI, em um bot nas suas pull requests.

Não fica, porém, e o motivo é simples: revisão é parte do loop, da mesma forma que construir e testar são. Você escreve algo, você o verifica, você o corrige, você recomeça. Revisão é a etapa de verificação. E se agentes são os que estão escrevendo o código agora, o que, para mim, eles são na maior parte, então avaliar esse código é apenas mais um verbo. Ele fica bem ao lado de `build` e `test` porque faz o mesmo trabalho que eles fazem: diz se a coisa é realmente boa antes de você seguir em frente.

Uma superfície vence três ferramentas, e vence três com mais força para um agente do que para mim. Esse é o argumento completo para integrar a revisão à CLI do ciclo de vida em vez de publicar uma ferramenta separada. Um agente em uma ferramenta de revisão separada precisa aprender uma coisa completamente diferente, com sua própria invocação e sua própria forma de saída, para fazer um loop contínuo de trabalho. Se o agente já sabe como conduzir o fledge para construir e testar, então `fledge review` é um verbo que ele já conhece a forma. Mesma superfície, mesmo JSON, mesmo modo sem cabeça. Nenhuma nova ferramenta para aprender apenas porque a etapa mudou de "compila" para "é bom".

No fledge isso é `fledge review`, e o que ele faz é revisão de código por IA contra o branch padrão. Então ele não está revisando o mundo inteiro. Está olhando para o que mudou, seu diff contra o branch em que você faria merge, que é exatamente a unidade em que a revisão realmente acontece. O mesmo escopo que um revisor humano ou um bot de PR olharia, executado como um verbo na mesma CLI em que você já vive.

E não está vinculado a um modelo ou provedor. Por baixo, a revisão passa pelo mesmo cliente multi-provedor que o resto do meu ferramental usa, corvid-ai, então o fledge fala com qualquer provedor que o corvid-ai suporta, a API da Anthropic, qualquer endpoint compatível com OpenAI e executores locais como Ollama, entre outros. A lista de provedores suportados vive no repositório corvid-ai e muda

conforme o ecossistema muda. O ponto é que a revisão é executada contra qualquer modelo que você queira, é sua escolha, não da ferramenta.

Vale ser direto sobre o que isso significa, porque o resto deste livro é direto sobre raio de explosão: `fledge review` pega seu diff e suas especificações e os envia para qualquer provedor para o qual você o apontou. Se for um endpoint hospedado, seu código não mesclado acabou de sair do edifício. Para um repositório privado isso é uma superfície real de egresso de dados, e é sua decisão a tomar com os olhos abertos, não um detalhe para ignorar. Apontá-lo para um modelo local é a versão em que nada sai da máquina. Uma ruga honesta nesse caso local: o caminho do Ollama aceita feliz uma chave e uma URL de nuvem, mas o README do corvid-ai ainda enquadra o Ollama como a opção local sem chave, então os docs parecem que Ollama significa o servidor embaixo da mesa. O caminho da nuvem funciona hoje; o README simplesmente não acompanhou. Isso é uma lacuna de documentação do meu lado, não um recurso ausente.

Há também um ângulo multi-modelo que eu realmente gosto. `--with-model` permite que você execute um painel, críticas paralelas do mesmo diff de mais de um modelo ao mesmo tempo.

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

Isso não é um truque. Se você passou algum tempo com esses modelos sabe que eles não capturam todas as mesmas coisas, e também não alucinam todas as mesmas coisas. Executar alguns deles sobre o mesmo diff e ver onde concordam, e onde um deles sinaliza algo que os outros perderam, é um sinal melhor do que confiar em um único. É uma segunda e terceira opinião, executadas em paralelo, sobre a mudança exata à sua frente.

E como tudo mais no fledge, a saída é estruturada. Cada comando emite `{schema_version: 1, ...}`. JSON por padrão. Então uma revisão não é um muro de prosa que um agente precisa ler e interpretar como uma pessoa faria. São dados. O agente que escreveu o código pode executar a revisão, receber as descobertas estruturadas de volta e agir sobre elas no mesmo loop, sem um humano no meio traduzindo "o revisor parece insatisfeito com o tratamento de erros" em algo para realmente fazer. Uma pessoa pode ler a revisão, e um agente pode fazer o parse, com o mesmo comando.

`fledge review` também é consciente de especificações, o que o capítulo sobre spec-sync cobre completamente; aqui vale apenas ver o que a revisão faz com uma

especificação quando a tem. A revisão descobre quais especificações cobrem os arquivos no diff, fazendo correspondência nos arquivos declarados da especificação e no diretório `specs/<name>/`, e integra essas especificações ao prompt como contexto. E aponta para elas deliberadamente: o modelo é informado de que as especificações descrevem o que os módulos *deveriam* fazer, para usá-las para interpretar a mudança, para revisar apenas o diff e não as especificações em si, e, a parte que importa, se o diff contradiz um invariante da especificação, para apontar isso como um bug no diff. Então desvio da especificação não é um sabor de fundo na crítica; é uma descoberta que a revisão é explicitamente instruída a trazer à tona.

E merece o lugar. `fledge review` encontrou um bug real para mim. Algo que teria sido publicado, bem ali em um diff que construiu bem e passou nos testes, que o LLM sinalizou antes de ser mesclado. Esse é o teste de se revisão como verbo vale alguma coisa, e ele passou: não uma nit de estilo, um defeito real que o resto do loop havia deixado passar. O lado consciente de especificações também valeu. Apontar o revisor para as especificações capturou código que silenciosamente se afastou do contrato que o módulo deveria honrar, desvio que compila e passa assim como o bug fez. Os agentes dependem bem disso, porque para eles tem a mesma forma que build e test: execute-o, leia as descobertas estruturadas, corrija o que ele encontrou, continue.

Se agentes escrevem o código, executam o build e executam os testes, avaliar o código é mais um verbo ao alcance que eles já conhecem, e captura coisas que os outros deixam passar.

Plugins, e não confiar neles

O fledge tem um núcleo pequeno que faz pouco, com toda a capacidade real nos plugins. Isso é proposital. Se o núcleo tivesse que conhecer cada linguagem, cada fluxo de trabalho, cada etapa estranha de cada equipe, ele apodreceria. Então não o faz. O núcleo conhece os verbos comuns e como detectar um projeto; o alcance real vem de plugins colocados em volta dele, e qualquer pessoa pode adicionar um sem tocar no núcleo.

O outro motivo para empurrar a capacidade para os plugins é que eu não queria ser dono da lista do que o fledge pode fazer. As pessoas o estendem como quiserem: Rust, Swift, TS, shell. Você escreve o plugin no que for confortável para você. Você não é forçado para a minha linguagem para estender minha ferramenta.

A forma como um plugin conversa com o núcleo é um contrato real e versionado: um binário com um manifesto `plugin.toml`, conversando JSON com o fledge, o mesmo seja nativo ou um módulo WASM em sandbox. O capítulo do protocolo tem o formato de wire e o handshake de capacidades; aqui é suficiente saber que a costura é um contrato, não uma pilha de convenções.

E para responder a uma pergunta que surge: plugins e lanes não são a mesma coisa. Um plugin adiciona uma capacidade, um novo verbo. Um lane encadeia verbos que você já tem em um pipeline ordenado. Então não é "tudo é um plugin". Há um núcleo real com os três pilares integrados, os plugins estendem as capacidades ao redor dele, e os lanes encadeiam essas capacidades em sequências. Dou aos lanes seu próprio capítulo mais adiante.

Qualquer linguagem, o que significa que você não pode confiar neles

O lado oposto de "estenda como quiser, em qualquer linguagem" é que você acaba executando código que não escreveu e pelo qual não pode se responsabilizar. Um plugin é apenas o programa de outra pessoa. Uma vez que você decidiu que qualquer pessoa pode escrever um em qualquer coisa, você também decidiu que vai executar muito código não confiável.

Então o fledge coloca plugins em sandbox com WASM, no Wasmtime. O ponto é segurança. Um plugin não pode ler todo o seu disco ou ligar para casa a menos que

você permita. Por padrão ele está encaixotado: ele recebe o que você concede e nada mais.

Há um segundo motivo para o sandbox, e é o real. Agentes executam esses plugins. Se agentes estão instalando e executando plugins, você não pode confiar neles por padrão. Não nos plugins, e não, cegamente, na decisão de executá-los. Um agente alcançando, pegando um plugin e executando-o é exatamente a situação em que "provavelmente está bem" não é suficientemente bom. O sandbox torna o ferramental executado por agentes seguro. Essa é a ponte para o material de confiança e raio de explosão que exploro no livro de agentes; aqui a versão do lado das ferramentas é simples: código não confiável mais uma coisa automatizada executando-o equivale a você precisar de uma caixa em volta dele. WASM/Wasmtime é a caixa.

A escolha de WASM sobre as alternativas óbvias vale ser nomeada. As duas que você alcançaria primeiro são perfis seccomp (filtragem de syscall do Linux) e containers (Docker ou similar). Ambos funcionam, mas ambos adicionam atrito ou suposições que não se encaixam aqui. seccomp requer privilégio em nível de SO para configurar e é apenas para Linux, então quebra a promessa cross-platform imediatamente. Containers significam um daemon Docker, um pull de imagem separado e uma fronteira de processo mais pesada do que "execute um plugin". WASM se encaixa de forma diferente: o Wasmtime se incorpora diretamente no processo do fledge como uma biblioteca, vem dentro do binário único, funciona em qualquer SO sem daemons ou privilégios adicionais, e aplica a fronteira de capacidades estruturalmente no momento do link em vez de por uma política do kernel que alguém precisa configurar. Não é que seccomp ou containers sejam errados. É que uma ferramenta publicada como um binário para funcionar em qualquer lugar precisa de um sandbox que viaje com ela, e o Wasmtime faz isso.

O canário

Um sandbox que você não pode provar é apenas uma esperança. Então há um canário, um plugin cujo trabalho é tentar fazer as coisas que um plugin em sandbox não deveria ser capaz de fazer, e confirmar que não consegue. É a prova de que o sandbox se mantém. Se o canário não consegue sair, a fronteira é real; se algum dia conseguisse, você saberia imediatamente.

Na verdade são dois plugins. `fledge-plugin-canary` é o nativo. Ele roda sem sandbox e prova que os ataques *funcionam*: lendo chaves SSH e credenciais AWS, puxando variáveis de ambiente como `GITHUB_TOKEN`, abrindo conexões de rede, gerando processos, escrevendo em `.git/hooks`. Então `fledge-plugin-canary-wasm` executa a mesma bateria dentro do sandbox do Wasmtime, onde cada um desses deveria voltar

BLOQUEADO. Sua própria descrição é direta sobre isso: ele "prova que o sandbox do Wasmtime bloqueia cada ataque que o canário nativo expõe".

O ponto fica mais nítido quando você vê os dois lado a lado em vez de confiar na minha palavra. A própria saída do canário nativo contrasta a si mesma com o que um plugin WASM executando o mesmo código veria:

```
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)
```

Mesmo ataque, duas fronteiras. Nativo lê sua chave SSH; WASM não consegue, porque não há diretório pré-aberto pelo qual ele possa alcançar ~. Nativo herda `GITHUB_TOKEN`; WASM não consegue, porque o ambiente do convidado está vazio. Qualquer resultado que volte VAZADO em vez de BLOQUEADO no lado WASM significa que o sandbox escapou. Os dois juntos são a verificação de ponta a ponta: um mostra que o perigo é real, o outro mostra que a caixa se mantém.

O que o sandbox não protege

O sandbox é contenção de raio de explosão. Ele delimita o que o código de um plugin pode alcançar. Ele não delimita tudo, e vendê-lo como cura para problemas que não resolve seria a versão desonesta deste capítulo. Então aqui está a parte que o WASM não te compra.

Ele não para o egresso de dados em camadas superiores da pilha. A caixa WASM mantém um plugin de alcançar sua chave SSH, mas não governa dados que você entrega a uma ferramenta de propósito. `fledge review` envia seu diff e suas especificações para qualquer provedor LLM para o qual você o apontou, e se for um endpoint hospedado, código não mesclado acabou de sair da máquina. Nenhuma fronteira WASM toca isso, porque os dados saem pela porta da frente, não pelo plugin. Esse é um problema de consentimento e política, e o trato como tal no capítulo de revisão, não algo que o sandbox resolve.

Ele não protege contra um plugin que é executado como o usuário do host. Nem todo plugin é WASM. Plugins nativos são executados com suas permissões, seu ambiente, seu disco. Essa é a razão inteira pelo qual o canário nativo pode ler sua chave SSH: ele não está em sandbox. Quando você executa um plugin nativo, ou um que bifurca um processo de fora do convidado, você o está confiando da forma como confia em qualquer coisa que executa em sua máquina. O sandbox é a garantia do caminho WASM, não uma garantia abrangente sobre cada plugin.

E ele não valida origem ou intenção. WASM contém o que o código pode fazer. Ele não diz nada sobre se o código, ou o agente que escreveu o código que o plugin executa, deveria estar fazendo isso. Um plugin executando lógica escrita por agentes ainda está executando lógica que eu não revisei, dentro da caixa. Código não confiável em uma caixa ainda é código não confiável. A caixa apenas mantém o dano de se espalhar.

Então o enquadramento honesto é restrito: WASM/Wasmtime é contenção de raio de explosão agnóstica à origem para o caminho de execução de código. Não é controle de fluxo de dados, não é gerenciamento de consentimento, e não é razão para parar de pensar sobre o que você entrega a um plugin ou quem o escreveu.

O ecossistema de plugins é maior do que eu teria adivinhado se você conhecesse o fledge apenas por seus três integrados nativos: muitos repositórios `fledge-plugin-*` na organização CorvidLabs em várias linguagens (o capítulo do ecossistema os cobre), escritos em qualquer linguagem que se adequasse ao trabalho. O sandbox é o que permite que isso seja verdade: um plugin pode ser escrito em qualquer coisa e ainda ser seguro de executar. Faço o tour completo do ecossistema e das linguagens em seu próprio capítulo.

Como o fledge se encaixa no dia a dia

Agentes executando plugins não confiáveis em velocidade em muitos repositórios é por isso que o sandbox não é negociável. A coisa que toma essas decisões geralmente não sou eu decidindo caso a caso. São agentes, rodando constantemente, em muitos repositórios. O sandbox WASM por baixo é o que torna seguro deixar isso funcionar.

O protocolo de plugins

Esta é a costura, o lugar real onde um plugin e o núcleo se encontram. Tem um nome no repositório: o protocolo é versionado, e a string de versão é `fledge-v1`. Um plugin o declara, e esse é o handshake. Todo o resto pende disso.

O que um autor de plugin realmente escreve

Um plugin é um repositório git com um manifesto na raiz chamado `plugin.toml`, mais um ou mais executáveis. O manifesto é curto. Você nomeia o plugin, dá a ele uma versão, diz qual protocolo fala e lista os comandos que está adicionando:

```
[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
```

Esse é o contrato completo do lado do autor. Cada entrada `[[commands]]` é um novo verbo que o fledge conhecerá, apontado para um binário a ser executado. O bloco `[capabilities]` é o autor dizendo, de antemão, o que este plugin precisa ser capaz de fazer: executar comandos shell, persistir um pouco de estado, ler metadados do projeto. O padrão é `false`. Você pede o que precisa e nada mais, e o pedido é visível no manifesto, não enterrado no código.

Como o host conversa com um plugin

Quando você invoca um comando de plugin, o fledge gera o binário e conversa com ele via stdin e stdout como linhas JSON, um objeto JSON por linha. A primeira coisa que o host envia é uma mensagem `init`, e essa mensagem é a ideia de zero-config tornada literal. Ela entrega ao plugin toda a situação: o nome e raiz do projeto, a

linguagem detectada, o estado do git (branch, limpo ou sujo, remoto), a versão e diretório do próprio plugin, a versão do fledge, e as capacidades exatas que foram concedidas. O plugin não precisa descobrir onde está ou o que está olhando. O host já sabe, esse é o trabalho inteiro do núcleo, então ele simplesmente informa ao plugin.

Depois disso é uma conversa. O plugin envia mensagens de volta: `prompt`, `confirm`, `select` para perguntar algo ao usuário; `log`, `progress`, `output` para reportar; `exec` para executar um comando; `store` e `load` para seu pequeno pedaço de estado. Qualquer coisa com um `id` recebe exatamente uma resposta de volta: uma `response` ou um `cancel`. Stderr nunca é capturado, então um autor de plugin pode sempre imprimir debug direto no terminal.

É simples o suficiente para ler por cima do ombro de alguém. O host abre com `init`, entregando a situação:

```
{"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}}
```

O plugin, já sabendo onde está, pede ao fledge para executar um comando:

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

E o host responde ao `id` correspondente:

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

Essa é a forma completa: `init` entrega tudo que um agente de outra forma teria que adivinhar como JSON estruturado em vez de prosa em um README, e cada requisição com um `id` recebe exatamente uma resposta correspondente. A ferramenta diz ao plugin onde está; o plugin não vai caçar.

Onde o sandbox se encaixa

Tudo acima descreve um plugin nativo, um executável normal, conversando JSON sobre pipes. O problema, e o repositório é direto sobre isso, é que um plugin nativo é executado como você, com todo o seu acesso. O bloco de capacidades bloqueia o

protocolo, mas não bloqueia o *processo*. Um plugin que não pediu nada ainda poderia ler suas chaves SSH, porque é apenas um programa rodando como seu usuário. Esse é o vazamento que o canário provou: declarar capacidades no nível do protocolo era teatro de segurança, e é a razão inteira para a mudança para WASM. Contei esse arco no capítulo do sandbox; aqui o ponto é apenas o que isso mudou no protocolo.

Plugins WASM mantêm o exato mesmo protocolo `fledge-v1` mas mudam a fronteira. O plugin declara `runtime = "wasm"` e publica um único binário `.wasm`. Em vez de stdin e stdout, as mesmas mensagens JSON cruzam por três funções do host (`recv`, `send`, `exit`) que o host vincula. Mesmos tipos de mensagem, mesma conversa. E as capacidades param de ser um pedido educado. Elas são impostas no momento do link: se você não concedeu `exec`, a importação `exec` simplesmente não é vinculada, e um plugin que tentar chamá-la falha ao instanciar. Não há verificação em runtime para enganar, porque a função que ele chamaria não existe para ele. O acesso ao sistema de arquivos é `none`, `project` ou `plugin`, montado como diretórios pré-abertos; rede é um booleano. Além disso, o runtime é limitado por combustível e tem limite de memória, então um plugin não pode rodar para sempre ou comer a máquina.

Então a capacidade que você vê em `plugin.toml` e a capacidade que o código pode realmente alcançar são a mesma coisa, estruturalmente.

O motivo pelo qual precisava ser estrutural é a lição que o canário ensinou: uma capacidade que você meramente *declara* é uma capacidade que você está confiando que o plugin vai respeitar, e a razão inteira pela qual você está colocando um plugin em sandbox é que você não confia nele. A única versão que se mantém é aquela em que a capacidade que você não concedeu não é alcançável, onde a função literalmente não está lá para chamar. Quando agentes são os que estão instalando e executando essas coisas, é isso que você quer.

Um risco de versão vale ser mencionado aqui. A string de protocolo que um plugin declara é `fledge-v1`. Quando uma mudança de protocolo que quebra compatibilidade vier, e virá em algum ponto, essa string se tornará `fledge-v2`. Um plugin que ainda declara `fledge-v1` contra um host esperando `fledge-v2` falha na instanciação, em voz alta, antes que qualquer código seja executado. Ele não se comporta mal silenciosamente. A versão no manifesto é o que força essa falha antecipada: um plugin incompatível não chega ao ponto em que pode fazer algo, então o operador sabe exatamente o que precisa ser atualizado em vez de perseguir um bug sutil em runtime.

Hooks do ciclo de vida de lanes

O mesmo protocolo carrega um segundo tipo de integração: hooks de ciclo de vida. Um plugin pode registrar um hook em seu `plugin.toml` que dispara em eventos `lane:pre` ou `lane:post` em vez de em uma invocação de comando direta. O plugin de deploy é o exemplo que vem com a documentação: ele registra um hook `lane:post` que roda automaticamente após qualquer lane do fledge ser concluída.

Quando o hook dispara, o fledge passa contexto para o binário do hook por meio de variáveis de ambiente: `FLEDGE_LANE_NAME`, `FLEDGE_LANE_STATUS` e `FLEDGE_LANE_RUN_ID`. A definição do lane não tem conhecimento de quais plugins estão instalados. O plugin não tem conhecimento de quais lanes existem. O evento do ciclo de vida é a costura.

Aqui está como a imagem completa parece. O lane vive em `fledge.toml`; o registro do hook vive no `plugin.toml` do plugin; os dois se conectam sem que nenhum dos dois conheça os detalhes do outro:

```
# 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
```

Quando `fledge lanes run verify` é concluído, o fledge dispara o evento `lane:post`. Qualquer plugin instalado com uma entrada `[[hooks]]` correspondente tem seu

binário de hook chamado com `FLEDGE_LANE_NAME=verify`, `FLEDGE_LANE_STATUS=success` (ou `failure`), e `FLEDGE_LANE_RUN_ID` definido.

O lane emite contexto estruturado como variáveis de ambiente em vez de prosa no stdout, então um plugin lê um status sobre o qual pode agir em vez de fazer scraping de texto e adivinhar. Esse é o mesmo princípio de design que a mensagem `init`: o host te diz o que aconteceu; você não precisa inferir.

Os três pilares ficam em cima disso

Os três pilares (execução de tarefas, scaffolding, revisão por IA) são núcleo, não plugins da comunidade. O protocolo é a camada de extensão *ao redor* deles. A espinha dorsal é do fledge, e `fledge-v1` é como qualquer pessoa pode aparafusar mais capacidade nele. O núcleo é real e os plugins o cercam; o protocolo é apenas a costura onde os dois se encontram.

O ecossistema de plugins

Há dezenas de repositórios `fledge-plugin-*` na organização CorvidLabs, em várias linguagens, com um punhado arquivado. O número ao vivo é real, mas é a coisa menos interessante sobre o ecossistema. Contagem de plugins é inventário. O argumento é quais eu realmente te entregaria, e quão seguro estou sobre cada um.

Então aqui estão os cinco que você usaria diariamente: `augur`, `attest`, `format`, `gitleaks`, `deps`. Comece por lá. O resto deste capítulo é sobre por que os outros trinta e sete existem e quanto confiar neles.

Os cinco que você alcançaria

Estes rodam basicamente em cada mudança, minha e a dos agentes. Eles se agrupam em dois trabalhos.

O par de confiança é `augur` e `attest`. `augur` pontua o risco de mudança de um diff, prosseguir, revisar ou bloquear, a partir de sinais estruturais apenas, sem chave de API e sem LLM, e sua descrição diz o público em voz alta: "para humanos e agentes". `attest` registra a proveniência assinada de quem revisou qual commit, mantida em git notes. Esses dois são os em que mais me apoio, porque são os que importam quando um agente é o autor. Pontuação de risco na entrada, rastro de proveniência na saída.

Os três do loop de desenvolvimento são entediantes de propósito: `format`, `gitleaks`, `deps`. Execute o formatador. Escaneie segredos confirmados. Verifique a saúde das dependências, desatualizadas, auditoria, licenças, em Rust, Node e Python. Nenhum deles é empolgante. Todos eles rodam constantemente, porque é exatamente o tipo de verificação que você quer disparar automaticamente em vez de lembrar.

Se você apenas instalasse esses cinco, teria a maior parte do que obtenho do ecossistema.

Três níveis, e quão seguro estou

A forma honesta de ler o roster é por confiança, não por contagem. Os plugins se enquadram em três níveis, e eu confio neles de forma muito diferente. Verifique o roster ao vivo na organização CorvidLabs para a lista atual; o que segue é como ler o que você encontrar lá.

Núcleo. `augur`, `attest`, `format`, `deps`, `gitleaks`. Os cinco diários. Eu os escrevi, os executo em cada mudança, e são mantidos e testados. Este é o nível no qual eu apostaria o fluxo de trabalho.

Integração. `github`, `discord`, `algochat`, `memory`. Estes conectam o fledge a algo externo: a API do GitHub por meio de `gh`, webhooks do Discord para notificações de falha de CI, mensagens criptografadas on-chain, um armazenamento de memória. Eles também são mantidos e testados, mas carregam o risco de tudo com que conversam. Uma integração é tão confiável quanto o serviço por trás dela.

Playgrounds. `weather`, `roast`, `hangman` e o resto dos experimentos únicos. `weather` imprime uma previsão no terminal. `hangman` joga forca com identificadores extraídos do seu código. `roast` passa um commit pelo LLM e o critica, "apenas para fins de entretenimento". Estes não são mantidos no mesmo padrão. Eles existem para provar que o sandbox funciona: se um plugin de clima semi-sério escrito de improviso é seguro de executar, o protocolo está fazendo seu trabalho. O baixo padrão não é uma fraqueza. É a prova.

Núcleo e integração são mantidos e testados. Os playgrounds provam que o sandbox se mantém. Não leia a contagem plana como uma distribuição uniforme de ferramentas igualmente verificadas, porque não são, e fingir o contrário seria a versão desonesta deste capítulo.

Proveniência, claramente

Eu escrevi a maior parte destes. Os cinco diários são meus. Assim como quase todos os experimentos únicos, escritos de improviso porque publicar um plugin não custa nada uma vez que o protocolo existe. Eu precisava de uma coisa, escrevi um plugin, o coloquei lá. O ponto do protocolo é que não preciso crescer o núcleo para adicionar uma capacidade.

Não vou afirmar uma comunidade de autores externos que não tenho. Alguns plugins vieram do círculo, mas não estou apresentando o roster como uma amplitude verificada pela comunidade. É principalmente uma pessoa enchendo uma gaveta, e essa é a proveniência honesta.

O canário, e por que qualquer coisa disso é segura

Um par merece sua própria menção porque é fundamental para todo o ecossistema: `fledge-plugin-canary` e `fledge-plugin-canary-wasm`, o par de red-team do capítulo do sandbox. O nativo prova que os ataques funcionam. O WASM prova que o sandbox os

bloqueia. Tudo mais aqui, um plugin escrito em Kotlin, um plugin que escrevi meio dormindo, só é seguro de executar porque esse par mantém a fronteira. O ecossistema pode ser caótico precisamente porque a caixa ao redor dele não é.

Por que tantas linguagens

A maioria é Rust e shell, depois Swift, TypeScript, Kotlin, Python e alguns em HTML/JavaScript. Essa distribuição é o protocolo funcionando como pretendido. O motivo inteiro para `fledge-v1` e o sandbox WASM é que um plugin pode ser escrito em qualquer coisa e ainda ser seguro de executar, então o ecossistema não precisa concordar com uma linguagem, apenas com um contrato. `fledge-plugin-bridge` é Kotlin. `fledge-plugin-memory` é TypeScript. `fledge-plugin-attest` é Swift. Metade das integrações é shell. Ninguém precisou aprender Rust para adicionar ao fledge. Eles escreveram a coisa no que estava na frente deles, e o núcleo permaneceu do mesmo tamanho o tempo todo.

Não sou dono da lista do que o fledge pode fazer. O protocolo é, e ele deixa qualquer pessoa adicionar a ele sem me perguntar. A distribuição é a prova, e os níveis são a honestidade sobre isso.

---

spec-sync e manter a honestidade

O desvio de especificação é o modo de falha que me importa mais do que a maioria. Você escreve uma especificação, o contrato para um módulo, o que ele faz, qual é sua API pública, e então o código deriva. Alguém muda uma função, a especificação ainda afirma a forma antiga, e agora o documento e o código discordam silenciosamente. Com uma equipe humana isso é um README desatualizado. Com um agente no circuito é pior, porque o agente lê a especificação como verdade, constrói sobre um contrato que o código não honra mais, e ninguém percebe até algo quebrar. spec-sync existe para tornar essa deriva impossível de ignorar.

spec-sync é sua própria ferramenta, seu próprio binário Rust, sua própria GitHub Action, e seu tagline diz o que é: "Validação bidirecional de especificação para código com referências entre projetos, grafos de dependência e geração com IA." As duas palavras que importam são *bidirecional* e *validação*. Ele verifica, em ambas as direções, que a especificação e o código concordam. Não é um conjunto de testes e não é um diff fuzzy contra prosa. É verificação estrutural de contrato: a API pública documentada realmente corresponde à real. As duas direções não são simétricas: uma exportação que o código tem mas a especificação não documenta é um aviso, algo para preencher; uma entrada que a especificação afirma que o código não tem é um erro, uma promessa quebrada.

O que ele realmente verifica

Uma especificação é um arquivo markdown, `*.spec.md`, com frontmatter YAML e um conjunto de seções obrigatórias. O frontmatter nomeia o módulo, uma versão, um status e os arquivos fonte que cobre. As seções `##` obrigatórias são Purpose, Public API, Invariants, Behavioral Examples, Error Cases, Dependencies e Change Log. Falte uma seção obrigatória e a especificação está incompleta e a validação bloqueia.

Essa é a regra; aqui está uma especificação honrando-a. Esta é a cabeça da especificação do próprio módulo `review` do fledge, o frontmatter real, e a seção `## Public API` preenchida com as exportações reais que o spec-sync verifica contra o código:

```
---
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 |
```

A linha `files:` é o que o spec-sync intersecta contra o diff e contra as exportações reais de `src/review.rs`. Cada linha nessa tabela `## Public API` é um nome que o spec-sync espera encontrar no código; uma entrada que o código não tem é o caso de erro abaixo. Se você nunca escreveu uma especificação, esse frontmatter mais uma tabela `## Public API` honesta é a forma a copiar. O resto das seções obrigatórias é o mesmo tipo de coisa, prosa e tabelas descrevendo o que o módulo promete.

Então ele valida nas duas direções:

- **Código para especificação:** exportação não documentada, aviso.
- **Especificação para código:** entrada fantasma ou desatualizada, arquivo fonte ausente, incompatibilidade de tipo, todos erros.

Ele faz o mesmo para schemas: tabelas e colunas de banco de dados declaradas são verificadas contra o SQL real, e uma tabela ou coluna fantasma falha. E ele resolve

referências entre projetos, então uma especificação em um repositório pode depender de um módulo em outro via `owner/repo@module` e esse link é verificado também.

A coisa a guardar é que isso é *estrutural*. Não está avaliando sua prosa e não está gerando testes. Está fazendo uma pergunta direta, a superfície documentada corresponde à superfície real, e respondendo deterministicamente. É isso que torna seguro colocar na frente de um agente. Não há julgamento para argumentar; ou a API corresponde à especificação ou não.

Como ele aparece

Ele aparece de três formas, e na entrevista a resposta para "como o spec-sync aparece no dia a dia" foi: todas elas. Três portas de entrada para o mesmo formato de especificação, cada uma rodando em algum lugar diferente:

Porta de entrada	Onde roda	O que faz
`fledge spec` (nativo)	sua máquina, dentro do fledge	`init`, `check`, `list`, `show`: validação própria do fledge, sem shell-out
binário `specsync`	loop do agente, antes de um PR	`specsync check` / `--fix`: a ferramenta standalone que o agente executa no loop
`CorvidLabs/spec-sync@v4`	CI, no pull request	bloqueia o build, comenta deriva, bloqueia em falha

O resto desta seção são essas três linhas, uma por vez.

O fledge conhece especificações nativamente. spec é um dos pilares do fledge. O README do fledge é verbatim sobre isso: "Spec | `spec` | [spec-sync]. Modules declare their contract, AI uses it as context." Vale ser preciso sobre o que isso significa no código: o fledge tem seu próprio `fledge spec` (`init`, `check`, `list`, `show`) integrado diretamente no binário. Ele não faz shell-out para a ferramenta `specsync`. Ele lê o mesmo `.specsync/config.toml` e os mesmos arquivos `*.spec.md`, os analisa por conta própria e faz sua própria verificação. Então o formato de especificação é compartilhado, mas a validação dentro do fledge é código próprio do fledge, não uma chamada para o binário separado.

Essa consciência nativa de especificação é também por que os comandos de IA do fledge podem se apoiar no contrato. `fledge ask` é Q&A consciente de especificação e `fledge review` é revisão de código consciente de especificação, ambos puxam as especificações relevantes como contexto. A especificação é o contexto que esses comandos alimentam o modelo. A especificação não é um documento ao lado; é a coisa que o ferramental lê quando raciocina sobre seu código.

Ela bloqueia o CI. Há uma GitHub Action standalone, `CorvidLabs/spec-sync@v4`, que executa `specsync check` automaticamente. Um fluxo de trabalho mínimo:

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

`strict` transforma avisos em erros. `require-coverage` define um piso. `comment` posta um resumo de deriva de especificação no pull request. E a verificação sai com código não zero em falha, o que significa que o PR é bloqueado. Esse é o mecanismo que torna tudo isso real: deriva não gera uma nota educada, ela falha no build. Um agente, ou um humano, não pode silenciosamente deixar a especificação e o código se separarem, porque a separação *é* a verificação falhando.

Ela mantém o agente dentro da especificação. Esta é a que mais me importa, e é por que a especificação vive dentro do loop do agente, não apenas no CI. A mecânica é concreta. O agente executa `specsync check` como parte de seu loop, da mesma forma que executa build e test. A verificação retorna com falhas estruturadas, e o agente as lê e decide o que fazer, e qual caminho seguir depende de qual direção a deriva foi.

Aqui está como essa saída estruturada parece quando ambas as direções derivaram (exemplo ilustrativo correspondendo ao formato de saída real):

```
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
```

Isso é o que o agente lê. Os erros nomeiam o arquivo, a afirmação da especificação e o que o código realmente tem. Os avisos nomeiam a exportação e o comando de

correção exato. Sem adivinhação necessária: ou reconcilie o código para corresponder à especificação ou corrija a especificação, depois execute novamente até a saída 0.

Se o código cresceu uma exportação que a especificação não menciona, a direção do aviso, o agente não edita a especificação à mão. Ele executa `specsync check --fix`, que adiciona automaticamente as exportações não documentadas à especificação como stubs, e o caso fácil é reconciliado em um movimento. O trabalho do agente ali é principalmente preencher o stub com uma descrição real, não descobrir a deriva em primeiro lugar.

A outra direção é a que exige trabalho real. Quando a especificação afirma uma API que o código não honra, a direção do erro, a entrada desatualizada ou fantasma, não há `--fix` para isso, porque a correção é um julgamento: ou o código está errado e o agente vai e faz o código corresponder ao que foi prometido, ou a especificação era aspiracional e o agente corrige o contrato. De qualquer forma o agente precisa reconciliá-lo no código, deliberadamente, e executar a verificação novamente até que esteja verde. Esse é o loop na prática: contrato, mudança, verificação, e depois uma adição automática ou uma correção real dependendo de qual lado saiu do passo, e ele roda antes que o diff chegue a um pull request, dentro do loop de Merlin, não depois do fato no CI.

O que o spec-sync não verifica

Há uma linha sobre a qual quero ser honesto, porque é a borda do que esta ferramenta faz. O spec-sync verifica que a spec e o código concordam. Ele não tem opinião sobre se a spec é boa.

Pense no que isso deixa em aberto. Uma spec pode nomear as exportações certas e descrevê-las de forma errada. Pode ser rasa, uma seção de Finalidade que não diz nada e uma tabela de API Pública que é precisa e não te diz nada sobre para que serve o módulo. Pode ser aspiracional, escrita para o código que alguém pretendia escrever. E se um agente rascunhou a spec a partir de um resumo de tarefa que estava ele mesmo errado, ou injetado, a spec pode ser um contrato fiel para a coisa errada. Em cada um desses casos o spec-sync passa. A superfície corresponde à superfície. A verificação está verde. O contrato está errado, e nada no loop capturou isso, porque verificar a spec contra o código não pode te dizer que a spec era ruim para começar.

Essa é uma lacuna real e estou nomeando como tal. O código tem uma verificação; a spec não tem. O que você quereria é um segundo portão que rode sobre a própria

spec, antes que um agente construa contra ela: cada seção obrigatória realmente diz algo, a API Pública aponta para arquivos que existem, há uma declaração clara de como sucesso e falha parecem, e um humano realmente assinou este contrato. Isso seria um `fledge spec lint`, e ele ainda não está construído. Até que esteja, a spec é a única entrada de todo esse pipeline que nada valida, e vale saber disso quando você confia na verificação verde.

Por que um contrato compartilhado é o ponto todo

Isso se conecta diretamente à tese por baixo de todas essas ferramentas: humanos e agentes usando as mesmas ferramentas, como cidadãos de primeira classe iguais. Uma especificação é a versão mais clara dessa ideia que tenho. O humano a escreve ou revisa; o agente constrói contra ela; o spec-sync mantém ambos à ela, em ambas as direções, deterministicamente. O humano que refatorou sem atualizar o documento é capturado da mesma forma que o agente que alucinará uma API é capturado. A verificação não se importa com qual deles derivou.

Construindo o fledge em Rust

O fledge é um único binário. Você o instala uma vez e ele roda em qualquer máquina, qualquer SO, sem mais nada para instalar ao lado. Sem runtime para trazer, sem interpretador, sem gerenciador de dependências que precise já estar presente. Cross-platform por padrão: um pipeline de build produz um binário que funciona em macOS, Linux e Windows. Esse é o requisito central, e Rust é a razão por que é fácil em vez de uma luta. O capítulo inteiro decorre disso: por que Rust foi a escolha certa, por que aprendê-lo não foi a história de guerra que as pessoas esperam, e como agentes cobriram a lacuna de fluência.

Eu estava escrevendo Swift por cerca de uma década quando comecei o fledge. Então a versão honesta deste capítulo é um pouco anticlimática: aprender Rust não doeu. Nada de grande importância lutou comigo. Nem mesmo o borrow checker, que é a parte sobre a qual todo mundo te avisa.

Acho que as pessoas esperam uma história de guerra aqui: a linguagem que me humilhou, o mês que passei perdendo batalhas com o compilador. Não tenho uma. E prefiro te dizer por que foi tranquilo a inventar o drama.

Swift me preparou

Muito de Rust pareceu familiar porque Swift já me havia treinado nas mesmas ideias, apenas com roupas diferentes.

Enums e correspondência de padrões foram a grande parte. Os enums de Swift são tipos soma reais, e eu estava me apoiando neles e em `switch` há anos. O `enum` e o `match` do Rust são o mesmo músculo. Result e Option também não eram novos. Eu havia vivido nos opcionais e no `Result` de Swift por um longo tempo, então "isso pode ser um valor ou nada" e "isso pode ser um valor ou um erro" já eram como eu pensava sobre código. O Rust apenas te obriga a lidar com ambos, o tempo todo, em voz alta. Isso não era uma nova disciplina para mim; era uma disciplina que eu já tinha, agora imposta.

Traits chegaram como aproximadamente os protocolos de Swift. Não idênticos, mas próximos o suficiente para que eu não estivesse aprendendo um conceito estrangeiro. Estava aprendendo um dialeto de um que eu conhecia. "Definir comportamento, conformar tipos a ele" tem a mesma forma em ambos.

Então a linguagem não pareceu uma parede. Pareceu um lugar onde já havia meio morado. Os hábitos de tipo-valor e opcionais que Swift me treinou são, acho, exatamente o motivo pelo qual o borrow checker nunca se tornou uma luta. Se você já pensa em termos de propriedade e "quem detém este valor", o verificador está principalmente te dizendo coisas que você já estava tentando fazer. Não era uma nova forma de pensar que eu precisava adquirir. Era um árbitro mais rigoroso para um jogo que eu já sabia jogar.

E a forma da coisa que constrói é a simples que eu queria: o fledge é um único binário Rust, construído com Cargo. Uma crate, um binário no final.

Agentes fizeram o trabalho pesado

Não vou fingir que sou um programador Rust fluente. Não sou. Sou um programador Swift fluente que consegue ler e direcionar Rust bem, e isso é uma coisa diferente.

O que fechou a lacuna foram os agentes. Apoiei-me neles pesadamente para ser produtivo em uma linguagem em que sou menos fluente. As partes familiares eu poderia ler, raciocinar sobre e direcionar por conta própria. Eu sabia o que queria que o código *fizesse* e como boa estrutura parecia, porque essa parte se transfere entre linguagens. As partes onde Rust tem seus próprios idiomas, sua própria forma de dizer uma coisa, os agentes carregaram.

Esta é uma distinção que vale traçar, vista do outro lado. Meu Swift é escrito à mão. Meu Rust é amplificado por agentes. Em um sou o autor nas teclas. No outro sou o que sabe como certo parece, apontando agentes para isso e verificando seu trabalho.

E honestamente, construir o fledge em Rust dessa forma é uma pequena prova da tese inteira por trás dele. A ferramenta é construída para humanos e agentes igualmente. Foi construída *por* um humano e agentes igualmente. Os agentes que conduzem o fledge hoje ajudaram a escrever o fledge em primeiro lugar.

O ferramental foi um alívio

Aqui está a parte que não esperava aproveitar tanto quanto apreciei: o ferramental do Rust pareceu um alívio.

Cargo simplesmente funciona. Uma ferramenta para construir, testar, dependências, a coisa toda, e é a mesma em todo lugar. O ecossistema de crates é profundo. O que quer que eu precisasse, geralmente havia uma crate sólida para isso, e puxá-la era uma linha. E builds de binário único são exatamente o que o fledge queria ser: eu

precisava publicar um binário leve que funcionasse em todo lugar, e Rust te dá isso por padrão.

O contraste é com o ferramental de Swift quando você sai das plataformas da Apple. Na Apple, a história do Swift é ótima. Fora dela (Linux, cross-platform, o tipo de alvo "roda em qualquer lugar como um único binário" que o fledge precisava) fica mais difícil. Essa é uma razão real pela qual o fledge é Rust e não Swift, mesmo que Swift seja minha linguagem nativa. Eu queria um binário leve e único que pudesse colocar em qualquer máquina, que agentes pudessem executar em qualquer lugar, e o toolchain do Rust tornou esse o caminho fácil em vez da luta.

Então essa é a verdade do construtor sobre isso. Swift preparou o pensamento, agentes cobriram a fluência que não tenho, e o ferramental (Cargo, as crates, o binário único) foi a parte que realmente me alegrou ter mudado. Escolher a linguagem foi a decisão fácil. O trabalho que realmente exigiu reflexão foi todo o resto: descobrir o que o fledge deveria ser.

Como eu uso e quem usa

Deixe-me começar com a parte honesta em vez de guardar para o final. O fledge não é amplamente usado. É meu, é dos meus agentes, é do meu círculo. Esse é o ponto. A ferramenta foi construída para resolver o problema real do construtor primeiro, e o faz, todo dia. Uma ferramenta que resolve o problema à sua frente vence uma ferramenta com um muro de logos e nenhuma pele no jogo.

Então a pergunta que este capítulo responde não é "quantas pessoas usam o fledge". É "quem, e por que essa é a ordem certa". A resposta não é "todo mundo", e nunca deveria ser.

Meus agentes o conduzem mais do que eu

O que as pessoas entendem errado quando imaginam como uso o fledge é que imaginam *mim* usando-o. Isso acontece, mas não é a principal forma como o fledge funciona mais. Meus agentes o conduzem, por uma grande margem. Quando um agente está trabalhando em um repositório, o fledge é como ele constrói, testa e executa o que acabou de mudar: sua superfície de execução, não apenas a minha. É exatamente o que o primeiro capítulo disse que o design era para. Na maioria dos dias os agentes tiram mais proveito disso do que eu, e estou principalmente direcionando.

Esse é o grupo de usuários para o qual construí. Não uma multidão de estranhos. Eu mais os agentes, em cada repositório que tenho, o tempo todo.

Quem mais usa

O fledge é open source, está no tap do Homebrew, qualquer pessoa pode fazer `cargo install` nele. Nada disso é uma afirmação de adoção. Um tap é um canal de distribuição, não uma contagem de usuários, e não vou enfeitar um como o outro.

Hoje a base de usuários é eu, meus agentes e o círculo CorvidLabs, as pessoas com as quais realmente trabalho. Não há adoção externa ampla. Nenhuma comunidade de estranhos abrindo issues. É infraestrutura pessoal e de círculo, e vale dizer isso claramente em vez de implicar um impulso que não existe.

Os colaboradores importam para a imagem, não apenas como uma contagem. O fledge ser a superfície compartilhada em todo o círculo faz parte do design. Quando

alguém no CorvidLabs pega um dos meus repositórios, não precisa aprender o dialeto privado daquele repositório. Já conhece os verbos, porque os verbos são os mesmos em todo lugar. A mesma consistência que me poupa de reaprender aterra as pessoas com as quais trabalho em uma superfície que já conhecem.

Você deveria usá-lo

Talvez ainda não. Honestamente, apenas se você trabalha como eu: muitos repositórios, agentes no circuito, uma superfície consistente em todos eles. Se esse é o seu problema, o fledge o resolve. Se não é, a ferramenta é excessiva e prefiro te dizer isso a te vender nela.

O que não vou afirmar é que adoção limitada prova que o design funciona em escala. Não prova. Prova que o design funciona para mim, que é uma afirmação menor e a única com a qual posso me comprometer. O fledge ganha seu lugar dentro do meu próprio mundo primeiro, todo dia, principalmente por agentes, comigo direcionando e um pequeno círculo na mesma superfície. Isso é uma ferramenta fazendo seu trabalho. A amplitude pode vir depois ou nunca. O trabalho já está sendo feito.

Para onde o fledge vai

A resposta honesta para "para onde o fledge vai a seguir" é menos empolgante do que as pessoas esperam, e acho que isso é um bom sinal. Está principalmente maduro. A grande forma dele está construída. O que resta é principalmente mantê-lo e continuar usando-o. Há algumas direções em que eu o expandiria, mas não estou sentado em um grande roteiro de reinvenção, porque o fledge não precisa de reinvenção. Precisa continuar sendo a coisa sobre a qual todo o resto se apoia.

Um ecossistema maior

Os níveis de playground e integração estão abertos: qualquer pessoa pode adicionar a eles sem pedir. O nível de núcleo não está. Essa assimetria é intencional e não está mudando.

A direção com mais espaço é a que o capítulo do ecossistema já descreveu: mais plugins. O fledge fica mais capaz pelo anel ao redor do núcleo crescendo, não pelo núcleo engordando: um conjunto mais rico de plugins para que qualquer repositório em que você colocar o fledge já tenha algo que saiba como lidar com ele. Mais integrações, mais dos experimentos únicos que cada um resolve uma coisa real.

Ligado a isso está ampliar o que o fledge detecta e executa fora da caixa, a promessa de zero-config do primeiro capítulo, mantida em mais lugares. Cada linguagem e plataforma que ele ainda não conhece é um repositório onde a promessa de mesmos-verbos-simplesmente-funcionam tem uma lacuna, então preencher detecção e cobertura de plataforma é a outra direção óbvia. Não é glamoroso. Cobertura é apenas o que torna uma superfície universal realmente universal.

E a cobertura que mais importa agora não é a próxima linguagem. É o sistema operacional. O fledge precisa funcionar de primeira classe nos três: Windows, Linux e macOS. Uma ferramenta que está realmente em casa em apenas um SO não é uma superfície universal, é uma local. Então suporte cross-OS é a próxima fronteira real, mais do que adicionar outro detector de linguagem.

O que significa manter algo do qual agentes dependem

Eu poderia listar recursos aqui e prometer uma reinvenção. Esse seria o enquadramento errado para o que o fledge é agora.

A questão que importa em 2026 não é quais recursos o fledge adiciona. Código é barato. Agentes escrevem a maior parte. O que é escasso é uma ferramenta na qual você pode realmente confiar para rodar seu trabalho. Um agente pode gerar nova funcionalidade mais rápido do que posso revisá-la. O que ele não pode gerar é uma ferramenta com anos de uso intenso por trás dela, uma interface estável, comportamento com o qual um pipeline pode contar.

Uma ferramenta com anos de uso intenso e uma interface estável com a qual um pipeline pode contar é o que o fledge é, e é a coisa mais difícil de construir. Não mais difícil tecnicamente. Mais difícil porque requer decidir que estabilidade é o resultado, não um subproduto. Toda ferramenta começa como um novo recurso. Poucas sobrevivem tempo suficiente para se tornarem infraestrutura. As que sobrevivem são aquelas em que o mantenedor tratou "nada quebrou" como a principal realização, não um prêmio de consolação por um sprint lento.

Uso o fledge em cada repositório, o tempo todo. Meus agentes o conduzem constantemente. Isso significa que os bugs me encontram. O verbo ausente, a detecção que adivinhou errado, o hook do plugin que disparou na ordem errada. Eu o encontro porque vivo nele, e corrijo porque vivo nele. O uso intenso não é uma atividade lateral. É o mecanismo de qualidade inteiro. Manter e usar são o mesmo trabalho.

A introdução deste livro perguntou o que faz uma ferramenta valer a pena como dependência. A resposta que apontou foi: uma superfície limpa única, nenhuma etapa interativa oculta, uma fronteira real de plugins, uma especificação que permanece honesta conforme o código muda. Construa isso, e um agente pode conduzi-la desde o primeiro dia, porque ela nunca foi construída para precisar de uma pessoa.

O fledge é isso. Não por causa de nenhuma decisão de design única, mas por causa do resultado acumulado de cada vez que me recusei a deixar um atalho corroer a interface. Os verbos são os mesmos em todo repositório. O JSON tem a mesma forma. O manifesto `fledge introspect` permanece atualizado. Nada é publicado que eu não tenha rodado no meu próprio trabalho primeiro.

Quando agentes são os que operam a maior parte do ferramental, o que ganha confiança não é uma longa lista de recursos. É aparecer da mesma forma toda vez, por tempo suficiente para que algo real tenha sido construído em cima. Essa é uma ambição mais silenciosa do que um slide de roteiro, e é a que importa.

Então o trabalho a partir daqui é o trabalho que sempre foi: manter sólido, continuar vivendo nele, manter a superfície honesta. Vou continuar encontrando os bugs porque continuo rodando, e vou continuar corrigindo. Esse é o plano todo.

Sobre o Autor

0xLeif (leif.algo) constrói em aberto. Uma década de bibliotecas Swift pequenas e combináveis como AppState, Cache e Fork. O laboratório CorvidLabs. Uma pilha de ferramentas para agentes que começaram principalmente como "eu desejei que isso existisse". Fora do teclado ele é Zach Eriksen.

Estes livros são entrevistas, moldadas em capítulos e verificadas contra o código real.

github.com/0xLeif · leif.algo

Agradecimentos

Obrigado ao CorvidLabs, por ser a sala onde essas ideias são testadas e moldadas em forma pelo debate.

Obrigado aos mantenedores de open source cujas ferramentas sustentam toda essa pilha. Nada disso é construído sozinho.

E obrigado aos primeiros leitores e aos apoiadores pague quanto quiser que tornam "grátis online" algo que posso continuar fazendo.

Colofão

Composto em Markdown, construído com bookgen, um pipeline puro em Rust (sem Python).

Conduzido por entrevistas e assistido por IA; editado e verificado por pessoas. Escrito sem travessões. Capa e arte dos capítulos das coleções Corvid and Nature no Algorand.