First-Class

Construindo para humanos e agentes igualmente

ZACH "LEIF" ERIKSEN

Direitos Autorais

Leitura gratuita online. O ePub é pague-quanto-quiser; se foi útil para você, pode apoiar o trabalho.

github.com/0xLeif · leif.algo

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

Dedicatória

Para todos que constroem em aberto, e publicam assim mesmo.

A Biblioteca

Estes livros funcionam de forma independente, 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 *(este livro)*
- **Building Agents**: Anotações de quem tentou dar as próprias mãos ao software
- **Open Source Tooling**: Construindo ferramentas que as pessoas realmente usam

Leitura gratuita online. Cada ePub é pague-quanto-quiser.

Sumário

Introdução

Este é o livro curto e teimoso dos quatro, e o que os outros estão realmente defendendo.

A maioria das ferramentas é construída para humanos, e depois um agente é parafusado na lateral: uma segunda API, uma flag, um modo especial que faz metade do que o produto de verdade faz. Acho que isso é ao contrário. A afirmação aqui é simples. Uma ferramenta deve ser first-class para ambos. Um humano deve ser capaz de usá-la sem um agente, e um agente deve ser capaz de usá-la sem um humano, pela mesma interface, sem uma porta de segunda categoria.

Cheguei a isso pela prática, não pela teoria. Faço pequenas bibliotecas Swift e ferramentas para desenvolvedores, e as que envelheceram bem foram aquelas cujo núcleo era limpo o suficiente para que qualquer coisa pudesse chamá-lo. Quando os agentes apareceram, essas ferramentas já estavam prontas. As que não estavam são as que fico pedindo desculpas até hoje.

Se o *Field Guide* é o método, e *Building Agents* e *Open Source Tooling* são as evidências, este é a tese que todos servem. Você não precisa dos outros três para usar este. Ele está aqui para mudar a forma como você vê a próxima ferramenta que construir ou escolher: pergunte se ela trata um humano e um agente como o mesmo tipo de cidadão, e observe o quanto deriva dessa resposta.

É curto de propósito. Leia de uma sentada.

Ambos os cidadãos

Boa parte das minhas ferramentas vem de uma ideia: humanos e agentes vão usar as mesmas ferramentas.

Então aqui está a definição em que o livro inteiro se baseia, logo de cara e sem rodeios: first-class significa que um humano usa a ferramenta sozinho, um agente usa a ferramenta sozinho, os mesmos comandos nos dois sentidos. Não dois produtos. Uma ferramenta, dois tipos de usuário, nenhum deles o caso especial no qual o outro é traduzido.

Quando digo *first-class*, quero dizer da forma como um programador entende: um participante real e suportado para o qual a ferramenta foi de fato construída, não um visitante que ela tolera. Não uma upgrade de assento. Um usuário first-class é aquele para quem a ferramenta foi projetada, com uma entrada de verdade. Não um que precisa primeiro ser traduzido para a forma de outra pessoa.

Podem existir ferramentas agent-first, construídas apenas para agentes. Podem existir ferramentas human-first, construídas apenas para pessoas. O caso interessante, o que quase ninguém está construindo, é a ferramenta que é os dois, e "os dois" é o que quero dizer com *first-class para ambos* e o que o restante do livro também quer dizer. É algo mais rigoroso do que agent-first: agent-first só precisa servir ao agente, enquanto first-class-para-ambos precisa servir a uma pessoa e a um agente pela mesma interface sem que nenhum dos lados receba uma versão pior. Agora temos principalmente projetos human-first e estamos trazendo agentes *para dentro* deles. O agente chega tarde, para uma ferramenta que foi construída para outra pessoa, e torcemos para que ele dê um jeito.

O que realmente precisamos são projetos human-and-agent-first. Construídos assim desde o início.

"Human-first com agentes parafusados por fora" é o padrão porque é o caminho de menor resistência. Você já tem a ferramenta, ela já funciona para você, e quando um agente aparece a jogada fácil é enrolar um pouco de cola em volta do que você tem. Funciona, mais ou menos. Você pegou uma ferramenta projetada em torno dos olhos e das mãos de um humano e pediu a um processo sem nenhum dos dois que fingisse ter ambos. O que isso custa a um agente (a pausa em um prompt, a saída que ele não consegue ler, o reaprendizado a cada execução) tem seu próprio capítulo mais

adiante. Aqui basta dizer que o agente acaba tampando uma lacuna em cada linha que assumiu a presença de um humano.

Então inverta o pressuposto. Construa a ferramenta para funcionar first-class de qualquer forma. Essa é a definição lá do início, reformulada como instrução de construção: você não tem um produto real e um "modo API" pregado na lateral, e não tem um CLI e um shim separado para agentes que sai de sincronia na primeira vez que você muda alguma coisa. Ambos são usuários reais. Ambos deveriam estar lá.

E aqui está a parte que mais importa para mim. Quando o agente é um usuário first-class, a ferramenta pode realmente *ajudá-lo*, em vez de fazê-lo adivinhar todos os comandos e como as coisas funcionam. Um humano consegue se virar com uma ferramenta confusa. Vai ler o README, tentar uma coisa, ler o erro, tentar outra, perguntar a um colega. Um agente se virando é apenas adivinhação cara. Uma ferramenta construída para o agente diz o que é possível, entrega a ele uma saída que pode ser usada diretamente, e falha de um jeito que diz o que fazer a seguir.

Essa é a mesma ferramenta que os humanos querem. Comandos descobríveis, saída confiável, erros que dizem o que deu errado. O agente simplesmente não consegue tapar as lacunas do jeito que um humano pode, então construir para o agente te força a fechá-las. Projetar para ambos torna o software melhor.

As pessoas ouvem "construa para agentes também" e assumem que significa dois produtos, ou um compromisso em que cada lado recebe uma versão pior do que queria. É um bom núcleo com uma etapa em que você o expõe para ambos. Isso não é o dobro do trabalho, e o próximo capítulo mostra por quê.

A razão pela qual continuo voltando a isso é que os agentes só vão fazer mais com o tempo, não menos. Hoje eles tropeçam em ferramentas humanas. Amanhã estão fazendo uma parcela real do trabalho, e a parcela aumenta. Se as ferramentas que construímos agora assumem que um humano está sempre lá para lidar com o prompt, ler a tela, clicar no botão, estamos construindo um futuro sobre um pressuposto que fica mais falso a cada mês.

Existe também uma versão maior desse argumento. Tivemos uma década de código escrito inteiramente por humanos, e agora a IA está escrevendo código em cima de tudo isso, e em algum momento essa mistura vira. As ferramentas são o que decide se isso vai bem ou mal. Isso é um capítulo inteiro por si só: a ponte.

Por ora, a afirmação é simplesmente esta: humanos e agentes vão usar as mesmas ferramentas. Então construa-as para ambos, como cidadãos first-class iguais, desde o início.

First-class para um agente

Então o que uma ferramenta precisa ter para que um agente seja um usuário first-class, e não uma ferramenta humana pela qual o agente tropeça?

Quatro coisas. Nenhuma delas exótica.

Saída estruturada e legível por máquina. Um formato de serialização de verdade, JSON é o que eu uso, para que o agente receba dados, não uma tela. A maioria das ferramentas devolve texto bonito: colunas alinhadas, cor, uma linha de resumo no final. Isso é para os olhos de um humano. Um agente tem que raspar o texto, e a raspagem quebra no dia que você muda o espaçamento. Dê a ele os dados de verdade. Ele lê um campo em vez de analisar um parágrafo.

Comandos descobríveis e consistentes. Verbos consistentes em toda a ferramenta, e um `--help` autodescritivo que o agente pode ler para descobrir o que é possível. Se o texto de ajuda for real, o agente o lê e sabe o que a ferramenta pode fazer. Ele não precisa ter visto essa ferramenta antes. Ele pergunta à ferramenta, e a ferramenta responde.

Erros que orientam o próximo passo. Quando algo falha, diga o que fazer a respeito. Não um stack trace, não `error: 1`. Um humano pode fuçar e descobrir um código de erro puro. Um agente recebe um código de erro puro e fica preso, ou pior, faz a coisa errada com confiança.

Não-interativo e determinístico. Roda sem prompts surpresa, para que o agente nunca fique preso esperando. Nada mata uma execução de agente como uma ferramenta que de repente pergunta "tem certeza? [s/N]" e fica lá para sempre, porque não há ninguém para responder. Dê a ele uma flag para rodar direto. Torne-o determinístico para que a mesma entrada produza o mesmo resultado.

Aqui está o que quero que você note sobre essa lista: cada item nela é um bom design de CLI simples. Nada disso é magia específica para agentes. Um humano também quer erros claros, comportamento previsível e comandos descobríveis. O agente simplesmente não consegue encolher os ombros e contornar a ausência deles. Então construir para o agente é construir bem a ferramenta e se recusar a se apoiar no "ah, um humano vai resolver isso."

Agora a parte que as pessoas erram. Elas acham que projetar para agentes e projetar para humanos puxam em direções opostas, que você está servindo a dois senhores e

alguém perde. Elas puxam juntas. Aqui está a forma real disso.

Você projeta um núcleo realmente bom que funciona. Depois o expõe. Você adiciona as flags: não-interativo, ou `--json`, ou introspect, o que a ferramenta precisar. E agora o agente pode usar a ferramenta CLI, e os humanos podem usar a ferramenta CLI. Mesma ferramenta. Mesmo núcleo. Você construiu uma coisa e a expôs duas vezes.

Depois se estende ainda mais a partir daí. Mais UI para humanos usarem. Ou apenas ferramentas sem UI. Ou plugins que humanos fazem, ou plugins que ajudam agentes, coisas diferentes assim. A extensão é onde os públicos realmente divergem: um humano quer affordances, uma boa interface, uma UI; um agente quer introspecção, plugins e modos. Tudo bem. Construa esses. Mas construa-os *em cima do* núcleo compartilhado, como extensões, não como dois produtos separados que você tem que manter em sincronia para sempre.

No núcleo é a mesma coisa. Essa é a linha a que continuo voltando. Ela só precisa ser exposta para agentes e para humanos. Então sim, há *uma* etapa extra. Você não ganha a propriedade de uso duplo de graça; você tem que deliberadamente expor o núcleo das duas formas.

Essa reformulação é o ponto inteiro deste capítulo, porque dissolve a objeção antes que ela comece. As pessoas resistem a construir para ambos porque calculam o preço como o dobro do trabalho: dois designs, dois conjuntos de testes, duas coisas para manter, duas coisas para quebrar. Na realidade é um núcleo mais uma etapa de exposição. O núcleo é a parte cara, e você ia construí-lo de qualquer forma. Expô-lo para um agente é principalmente a disciplina de saída estruturada, comandos consistentes e bons erros, as coisas que você deveria ter feito independentemente.

Há uma questão de ordem aqui. O núcleo vem primeiro, ou você projeta para ambos de uma vez? É algo suficientemente próprio para ter seu próprio capítulo mais adiante. Por ora: você expõe o núcleo das duas formas, e essa exposição é a parte barata.

Minha própria pilha de ferramentas é construída exatamente dessa forma. Pegue o fledge. Ele tem um verbo `introspect` de verdade, e o trabalho inteiro de `fledge introspect --json` é deixar um agente descobrir os comandos disponíveis como dados estruturados em vez de raspar texto de `--help` destinado a um humano. Essa é a etapa de exposição tornada literal: mesmo núcleo, mas o agente pode perguntar "o que posso fazer aqui?" e receber uma resposta legível por máquina. Depois faço os próprios comandos rodarem headless. Defina `FLEDGE_NON_INTERACTIVE` para que nada pare para fazer uma pergunta, passe `--json` para que a saída volte como dados, e agora o agente tem uma entrada first-class que nunca dependeu de ler prosa.

Vou tornar isso concreto. `fledge doctor` verifica o ambiente do seu projeto. Execute simples e você recebe algo para os seus olhos: colunas alinhadas, um checkmark por linha, um status com emoji, uma linha de resumo:

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

  8 checks passed, 0 issues found
```

Esse bloco é exatamente o "antes" em que um agente engasga. É bonito, e é uma tela. Para saber se o remote está configurado, o agente tem que encontrar a linha certa, reconhecer um checkmark verde, e confiar que o espaçamento não vai mudar no próximo release. O status que ele se importa é um emoji enterrado em uma coluna. Na prática: o agente raspa o texto alinhado, o parse quebra quando um nome de repositório mais longo desloca as colunas em dois caracteres no próximo release, e o agente lê a verificação de remote como aprovada quando ela está faltando. Tomou o caminho errado em uma correspondência de string que nunca foi um contrato de verdade.

Execute o mesmo comando com `--json` e as mesmas verificações voltam como dados:

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

O `schema_version` é o primeiro campo de propósito. É a coisa que o agente lê antes de qualquer outra, porque diz ao agente em que formato está o resto do documento. No

dia em que o formato de saída mudar, esse número muda junto, e um agente que fixou em uma versão sabe que deve se adaptar em vez de ler silenciosamente dados novos como se fossem antigos. É como o contrato sobrevive à revisão da ferramenta.

Além disso, mesmo núcleo, as mesmas verificações rodaram. O humano recebe checkmarks e uma linha de resumo. O agente recebe um campo `status` em que pode ramificar em vez de raspar um checkmark verde de uma coluna. O campo `fix` é `null` aqui porque nada está errado; esse é o contrato: `fix` é `null` em uma verificação aprovada.

Agora execute onde algo *está* errado. Digamos que o repositório não tem remote configurado. A versão simples escurece uma linha e o agente está de volta a adivinhar; a versão `--json` transforma isso em um registro sobre o qual ele pode agir:


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


Esse é o ramo em que o agente realmente age. `status` muda de `"ok"` para `"missing"`, `detail` diz o que está errado em palavras simples, e `fix` agora é uma string preenchida, um comando literal que o agente pode executar para corrigir. O agente não precisa saber o que significa um remote ausente ou inventar uma recuperação; a ferramenta que encontrou o problema também entregou a correção. O humano recebe uma linha escurecida; o agente recebe um `status` para ramificar e um `fix` para executar. Um comando, exposto duas vezes.

O teste para saber se você acertou é simples. Entregue a ferramenta a uma pessoa sem agente. Funciona, é boa? Entregue a um agente sem pessoa. Funciona, é boa? Se

a resposta para ambos for sim, e você não construiu a coisa duas vezes para chegar lá, você fez certo.

A ponte

Mencionei isso no final do capítulo anterior, mas um: há uma versão maior do argumento, uma questão de porcentagem, e este é ela.

Pegue o fio de volta. Tivemos mais de uma década de código 100% escrito por humanos, cada linha em cada repositório escrita por uma pessoa, e isso significa toneladas de código legado, toneladas de dívida técnica, uma década de tudo isso rodando tudo. Esse é o mundo como ele realmente é agora.

E agora estamos usando IA em cima de tudo isso.

Essa é a parte difícil, e as pessoas não ficam com ela tempo suficiente. Há simplesmente tanto legado e tanta coisa humana já lá. A IA aparece para *isso*.

E se continuarmos assim, se nos apoiarmos cada vez mais na IA para escrever coisas, se torna uma questão real. Em que ponto vira para código 100% escrito por IA? Alguma vez vira? E o que você faz com a década de código humano que já está lá? Você simplesmente deixa o código legado como legado e segue em frente? Reescreve? Finge que não existe? Ninguém tem uma boa resposta, e a maioria das pessoas nem está fazendo a pergunta. Elas estão apenas parafusando um agente no que têm e esperando.

Toda essa transição, do tudo-humano para o que-vem-depois, é para o que as ferramentas certas servem. As ferramentas são a ponte por ela.

O ideal é que você configure ferramentas que *tanto* humanos quanto IAs possam usar. Você a configura para escrever e construir usando desenvolvimento orientado a spec, spec-sync, e depois fledge para os comandos. Essa é a ponte. Se você começar a usar essas ferramentas, é mais fácil introduzir um agente em código existente. E se você começar um projeto totalmente novo, é mais fácil para humanos também. A mesma configuração ajuda o agente a entrar em uma bagunça de dez anos e ajuda uma pessoa a permanecer no controle de algo completamente novo.

São especificações, não prompts melhores ou agentes mais inteligentes, porque uma spec é uma fonte única de verdade para ambos os lados. O humano declara a intenção. O agente constrói a partir dela. É o mesmo problema do último capítulo, um nível acima. Um agente tropeçando em uma ferramenta humana adivinha os comandos. Um agente largado em uma base de código sem spec adivinha a *intenção*. A spec é o que ele lê em vez disso.

E mantém o código honesto com a spec. Essa é a parte que torna a ponte confiável em vez de apenas esperançosa. spec-sync mantém a spec e o código em acordo, aplicado no CI, para que o agente não possa se afastar silenciosamente do que foi combinado. Isso importa mais do que parece. O modo de falha que todo mundo teme com agentes não é o dramático; é o silencioso, em que o agente lentamente constrói algo que não é o que você pediu e ninguém percebe até que está fundo. O CI verificando código contra spec é o que detecta o desvio enquanto é pequeno. O contrato é lido por ambos os lados, e uma máquina verifica que ninguém o quebrou.

Aqui está o que essa máquina realmente verifica. A spec é um arquivo markdown, um `*.spec.md`, que escreve o contrato: o propósito, a API pública, os invariantes, os casos de erro. spec-sync compara isso com o código real, nas duas direções. Uma exportação que a spec nunca documentou é sinalizada. Uma promessa da spec que o código não tem é um erro. Ele verifica o schema declarado contra as migrações reais da mesma forma. É estrutural: a superfície pública e o schema estão alinhados com o que foi escrito. Não são testes gerados e não é uma comparação difusa com a prosa, e essa estreiteza é por que eu confio: ele afirma que o contrato e o código concordam na forma das coisas, nada mais. No CI ele roda como um gate real. Sai com código diferente de zero quando divergiram, e publica o desvio diretamente no pull request.

Isso configura a divisão de trabalho que realmente me importa. O humano possui a intenção. O agente possui o trabalho árduo. O humano permanece no controle no nível da intenção, o que isso deve fazer e por que, enquanto o agente faz a implementação. Cada lado faz a parte em que é realmente bom, com a spec como a linha entre eles.

E é uma rampa de entrada segura para código legado, que é onde isso volta ao problema da porcentagem. Você tem uma década de código sem specs, porque ninguém escreveu specs, porque um humano escreveu e a intenção vivia na cabeça daquele humano. A jogada é: capture o que um pedaço *deveria* fazer como uma spec, depois deixe o agente trabalhar contra isso. Você não está pedindo ao agente para adivinhar a intenção de código de dez anos atrás. Você está escrevendo a intenção primeiro. Um humano pode fazer isso, é a parte em que humanos são bons. E agora o agente tem um contrato para construir e refatorar. É assim que um agente entra em código existente sem ser um desastre. A spec é a rampa de entrada.

Imagine a jogada em uma função real que não tem spec. Antes: código funcionando, sem contrato escrito, a intenção vivendo inteiramente na cabeça do autor original, então um agente largado nela lê as assinaturas, adivinha os invariantes, e começa a mudar coisas. Depois: o mesmo código com um `*.spec.md` ao lado, a API pública e os invariantes escritos em linguagem simples, e spec-sync aplicando no CI que o código

permanece honesto com o que foi escrito. Você não precisa fazer a base de código inteira de uma vez. Você faz um pedaço de cada vez, escrevendo o contrato para a parte que está prestes a entregar.

A spec não mudou o que o código faz. Mudou o que o agente pode fazer com o código: trabalhar contra um contrato escrito em vez de adivinhar um.

fledge é a interface de comandos para tudo isso: os verbos consistentes pelos quais tanto o humano quanto o agente conduzem o trabalho, a coisa do último capítulo que não faz o agente adivinhar como construir, testar e executar. spec-sync é o contrato; fledge é como você age sobre ele.

Aqui está a costura entre eles. spec-sync existe por conta própria: sua própria ferramenta, sua própria verificação de CI. Mas fledge trata a spec como uma parte first-class do loop de desenvolvimento. Há um verbo `spec` bem ao lado dos demais, e `fledge ask` e `fledge review` são spec-aware, então a spec é o contexto que o agente lê quando responde a uma pergunta ou revisa uma mudança. A spec não é documentação que o agente ignora; é a coisa que fledge entrega ao agente para trabalhar.

Então o gate dispara dos dois lados. O pilar `spec` do fledge executa a verificação spec-sync ali mesmo no loop de desenvolvimento, e a GitHub Action `CorvidLabs/spec-sync@v4` a executa novamente no CI, para que nada seja mesclado que tenha se desviado silenciosamente. Quando um agente está fazendo o trabalho, essa verificação roda dentro do seu loop, para que ele permaneça dentro da spec enquanto avança em vez de descobrir no final.

Agora siga até o final, porque o final é o motivo de tudo isso. Se isso funcionar, se as ferramentas e specs e os trilhos que mantêm os agentes honestos simplesmente *existirem*, os humanos sobem de nível. Até a intenção e a direção, o agente fazendo o trabalho árduo embaixo contra uma spec, em aberto onde você pode verificar. Essa jogada tem seu próprio capítulo mais adiante; aqui é suficiente saber que é para onde a ponte leva.

A porcentagem vai continuar se movendo. Mais do código será escrito por agentes; isso é simplesmente verdade. O que cabe a nós é a ponte, as ferramentas compartilhadas, o contrato compartilhado, e se realmente a construímos. Não acho que a maioria das pessoas tenha começado.

O muro da identidade

Antes de entrarmos nas ferramentas, o limite honesto. Todo esse argumento bate em uma parede, e não é uma que posso derrubar com ferramentas melhores, então quero que seja dito claramente e cedo: as plataformas não vão deixar um agente existir como ele mesmo. Tudo depois deste capítulo é o que você faz dado isso.

Se um agente é um usuário first-class das suas ferramentas, mais cedo ou mais tarde ele também precisa ser um cidadão first-class das plataformas. São a mesma ideia apontada em duas direções. Uma ferramenta que trata o agente como um usuário real dá a ele saída estruturada, comandos descobríveis e uma entrada que não depende de um humano. Uma plataforma que tratasse o agente como um participante real daria a ele uma conta, uma identidade, um lugar legítimo para existir entre todos os outros. Essa segunda coisa é exatamente o que você não pode conseguir.

Tentei. Fui dar a um agente sua própria identidade nas plataformas em que todo mundo constrói: sua própria conta, não um invólucro da minha. As plataformas não permitem. Conto essa história adequadamente no livro Building Agents. O que importa aqui é que tipo de parede é essa. O bloqueador na frente de um agente genuinamente autônomo não é a capacidade do modelo, e nem mesmo a segurança. É que as plataformas não vão conceder ao agente uma identidade para existir.

É uma parede de política, não técnica. Posso rodar a VM. Posso montar o loop. O modelo pode fazer o trabalho. Nada disso é o bloqueador. O bloqueador é que não consigo fazer a plataforma permitir que o agente esteja lá, e isso não é meu para corrigir. É uma decisão que outra pessoa tomou, sentada acima de tudo que posso construir.

Importa por causa da responsabilidade, que é o modelo que realmente quero. O estado final não é um agente correndo solto. É um agente com uma identidade real, nomeada e com escopo definido: capacidade total, privilégios reduzidos, um gate de aprovação humana. Ele faz o trabalho, envia até um pull request, e o merge é meu, porque se algo age no mundo sob meu nome sou eu que assino. Mas você não pode ter *responsabilidade* sem *identidade*. Um agente que você não pode nomear, não pode delimitar, não pode apontar e dizer "aquele fez isso": não há nada a responsabilizar. Negar a identidade e você negou a versão responsável junto com ela, e o que sobra é ou nenhum agente ou um sem responsabilidade.

Então aqui está o quadro para o resto do livro. O agente pode ser um usuário first-class de uma *ferramenta* que construo, porque essa parte é minha. Ele não pode ser um cidadão first-class de uma *plataforma* que não possuo, porque essa parte não é. Tudo depois deste capítulo é construído dentro desse limite: é assim que first-class para ambos se parece quando você pode dar ao agente uma entrada real em sua ferramenta mas não pode dar a ele um lugar real no mundo. Não a IA, não a tecnologia, não a segurança. As plataformas não deixam o agente existir. Todo o resto eu posso construir. Esse ainda não consigo, por ora. Em 2026 o muro ainda está de pé. Os únicos caminhos são contornos que você mesmo opera: uma conta humana envelhecida com histórico real de atividade (uma conta nova de agente é bloqueada imediatamente), ou uma camada de verificação que você mesmo hospeda. Esse é o estado das coisas.

Quando é parafusado por fora

É fácil concordar com "construa para ambos" e nunca realmente imaginar a falha. Então deixe-me colocá-lo no lado do agente de uma ferramenta human-first, porque é onde você sente.

Duas coisas dão errado, e dão errado constantemente.

A primeira é que o agente fica preso. Ferramentas travam em prompts interativos o tempo todo: uma confirmação, um "tem certeza?", algo sentado lá esperando por um toque de tecla. E não há ninguém para pressionar a tecla. Então a execução não falha, exatamente. Ela apenas para. Fica em um prompt que foi escrito para uma pessoa que olharia e pressionaria enter, e o agente espera, porque esperar é a única coisa que a ferramenta deu a ele para fazer. Uma execução inteira morta em uma pergunta que ninguém está lá para responder.

A segunda é a documentação. Ou não existe, ou existe e é confusa. O que significa que o agente tem que *aprender* a ferramenta. E aqui está a parte que continuo notando. Ele não realmente aprende. Não pode. Não há lugar para esse conhecimento viver entre execuções. Então faz a versão cara: varre os arquivos, lê o que quer que esteja no índice, faz suas próprias anotações, reconstrói como a ferramenta funciona do zero. Toda vez. A ferramenta *sabe* o que pode fazer, está ali mesmo no código, e simplesmente nunca diz ao agente. Então o agente reconstrói essa imagem do nada em cada execução única.

Pense no quanto isso é desperdiçador. Uma pessoa lê a documentação uma vez, talvez dê uma olhada, e carrega isso na cabeça depois. Desenvolve uma sensação para a ferramenta. O agente não ganha nada disso de graça. O que a ferramenta não lhe diz diretamente, ele tem que ir buscar de novo, pagar de novo, adivinhar de novo. O trabalho que a ferramenta deveria ter feito uma vez, o agente refaz para sempre.

Ambos são o mesmo erro usando duas fantasias diferentes. A ferramenta assumiu que um humano estaria lá, a paciência de um humano no prompt, a memória de um humano de como a coisa funciona, e um agente não tem nenhum dos dois. Ele não pode encolher os ombros e esperar. Ele não pode lembrar através da parede entre execuções a menos que você lhe entregue algo para lembrar.

E ambos se mapeiam diretamente na lista de alguns capítulos atrás. Não-interativo: não pare e espere por uma pessoa que não está chegando. Descobrível: diga ao agente o que você pode fazer, em um formato que ele pode ler, para que ele não tenha que ir reconstruir você a partir do seu próprio código-fonte. O travamento e o reaprendizado não são problemas exóticos de agentes. São simplesmente essas duas propriedades ausentes, e um humano cobrindo silenciosamente a lacuna toda vez para que você nunca notasse que a lacuna estava lá.

Parafusar agentes em ferramentas human-first funciona mais ou menos, até o momento em que você observa o que o agente tem que fazer para que funcione. Então você vê o quanto da ferramenta estava se apoiando em uma pessoa o tempo todo.

Aqui está um meu, nomeado. fledge começou como um task runner que eu conduzia à mão. O núcleo era limpo desde o início, a lógica de build, teste e execução ficava por trás dos comandos em vez de dentro deles, então no dia em que apontei um agente nele a maior parte simplesmente funcionou. Exceto em um ponto. O agente executou `fledge work commit` e parou completamente. Esse comando imprime `Commit message:` e espera alguém digitar. Não havia ninguém. O agente ficou em um cursor piscando até desistir, porque aquele comando havia silenciosamente se apoiado em uma pessoa o tempo todo. A correção não foi um agente mais inteligente. Foi o caminho não-interativo: um switch `FLEDGE_NON_INTERACTIVE` que faz todo prompt se comportar como já respondido, e um jeito de passar a mensagem antecipadamente em vez de esperar por ela. O sinal é que o prompt só existiu porque eu era o que estava respondendo. Construído para ambos desde o primeiro commit, não estaria lá para travar.

Exponha o núcleo duas vezes

Núcleo difícil e novo: construa primeiro, exponha depois. Núcleo direto: projete para ambos desde o primeiro commit. Essa é a heurística. Aqui está por que funciona dessa forma.

Quando o núcleo é a parte difícil, a parte nova, a coisa que é genuinamente difícil de acertar, construo isso primeiro e me preocupo com a superfície depois. Criptografia é assim. Um parser é assim. Um scorer que precisa estar certo é assim. O meio difícil é onde todo o risco vive, então é onde a atenção vai primeiro. Faço a coisa *funcionar*, corretamente, para um caso real, antes de pensar muito em como um agente ou um humano alcança para usá-la.

E a razão pela qual essa ordem está certa, a razão pela qual não é um risco, é que uma vez que o núcleo difícil está certo, a superfície é barata. Adicionar `--json`. Adicionar um comando introspect. Adicionar uma flag não-interativa. Nada disso é a parte difícil. São algumas horas de expor o que já está lá em uma forma que o outro lado pode ler. Então construir depois não me custa muito. A coisa cara foi construída primeiro, as coisas baratas foram pregadas depois, e essa é a ordem certa.

Mas muitas ferramentas não são assim. Muitas vezes já sei que ambos os públicos estão chegando, porque sempre sei que ambos os públicos estão chegando agora, e então a forma voltada para agentes puxa o design desde o primeiro commit. Não estou construindo um núcleo e depois descobrindo que um agente precisa dele. Estou construindo sabendo que um humano vai conduzir manualmente e um agente vai conduzir headless, e ambos estão na minha cabeça enquanto moldo a coisa. Não há uma etapa "exponha depois" porque expor nunca foi uma fase separada.

Então o quadro real é: a parte cara é construída primeiro, e a parte cara geralmente é o núcleo. Quando o núcleo é difícil, você o aperfeiçoa e a superfície segue facilmente. Quando o núcleo é direto, não há nada para proteger indo em ordem, então você simplesmente projeta para ambos de uma vez. De qualquer forma você chega ao mesmo lugar: um bom núcleo, alcançável de forma limpa por uma pessoa e por um agente.

A objeção fácil a essa afirmação é uma ferramenta CLI onde as interfaces para humano e agente são quase idênticas mesmo assim. Justo. A objeção mais difícil é uma ferramenta onde a interface humana é inerentemente visual, com estado ou interativa: uma ferramenta de design com uma tela, um REPL, um depurador

interativo. Algo como o Figma. A interface humana do Figma é uma tela em que você arrasta coisas. A interface do agente é uma API REST e uma interface de plugins. Elas não se parecem em nada. Se o princípio "um núcleo, duas interfaces" vale em algum lugar, ele tem que valer lá também.

Vale, e a razão é que a API REST que o Figma expõe aos agentes é a mesma camada estruturada sobre a qual a tela é construída. Quando você arrasta um frame, a tela chama o mesmo modelo de documento que a API lê e escreve. A interface humana é rica porque esse núcleo é rico. A interface do agente não é um segundo sistema colocado para agentes; é essa mesma base com a tela tirada da frente. A diferença entre as duas interfaces é real. Ainda há um único núcleo por baixo das duas.

Agora a afirmação em que o livro inteiro se apoia, aquela que lhe devo honestamente: construir para ambos não é o dobro do trabalho. Aqui está o raciocínio real, não um número que inventei. O dobro do trabalho seria dois núcleos: duas implementações da parte difícil, dois conjuntos de testes sobre a lógica real, duas coisas que podem estar erradas de formas diferentes. Não é isso que é isso. Há um núcleo. A parte difícil existe uma vez. O que você adiciona para o agente é exposição: saída estruturada, uma flag não-interativa, uma forma de introspectar os comandos. Essa exposição é barata em relação ao núcleo, e é barata por uma razão que você pode verificar: ela não tem nenhuma lógica nova para acertar. `--json` serializa um valor que o núcleo já computou. Uma flag não-interativa pula um prompt que o núcleo nunca precisou. Introspect reporta comandos que já existem. Nada disso rederiva a resposta; re-apresenta uma resposta que você já tem. A coisa cara sobre software é estar correto sobre o problema difícil, e você paga por isso uma vez.

Há um imposto honesto a nomear, e ele pousa em dois lugares. O primeiro é quando o núcleo já foi construído human-first. Então expô-lo para um agente não é gratuito: você tem que ir encontrar cada lugar onde a lógica vazou para a apresentação (um valor que só existiu como uma string formatada, uma decisão tomada dentro de um print statement) e puxá-la de volta para o núcleo para que haja algo real para serializar. Essa refatoração é o custo de retrofit, e é exatamente o custo que o livro inteiro está argumentando que você pode evitar construindo para ambos desde o início. O segundo é a superfície de testes: duas formas de entrada significa que você quer verificar que ambas funcionam, e isso são mais casos de teste do que uma forma de entrada. Mas são mais casos sobre o *mesmo* núcleo, não um segundo núcleo para verificar. A lógica que você está testando é compartilhada; você está confirmando que duas superfícies a expõem fielmente, o que é mais barato do que provar duas coisas separadas corretas.

fledge é o caso que posso calcular o custo, porque é o que construí antes de estar pensando em agentes e depois observei um agente encontrá-lo. O núcleo já era limpo, a lógica vivia por trás dos comandos, então expô-lo para um agente foi uma camada fina e não uma segunda construção: saída `--json` estruturada, um modo headless, um comando introspect que lista o que está disponível. Esse trabalho levou dias, não semanas, e levou dias pela razão que este capítulo vem argumentando. Não havia segundo núcleo para escrever, apenas uma segunda porta para o único que já estava de pé. O que foi mais difícil, a única coisa que foi mais difícil, foi o punhado de comandos onde eu havia deixado uma decisão viver dentro de um prompt em vez de dentro do núcleo. Esses eu tive que separar para que a escolha ficasse em algum lugar que um agente pudesse alcançar. Essa foi a conta inteira, e era pequena, e era exatamente o imposto de retrofit nomeado acima, pago até quase nada porque o núcleo havia sido mantido honesto primeiro. O que foi mais barato foi tudo o mais, que é quase tudo.

O que eu te alertaria é o inverso: começar de "como faço isso ser amigável para agentes" antes que o núcleo seja bom. Isso te dá uma ferramenta fina com uma flag JSON pregada na lateral, que é apenas o problema human-first-com-agentes-parafusados-por-fora de novo. A superfície é barata *porque* o núcleo é sólido. Se você pular o núcleo, a superfície barata não tem nada embaixo, e você descobre isso na primeira vez que alguém realmente se apoia na ferramenta.

Por que construo minha própria pilha

Pergunta justa a me fazer nesse ponto: por que construir qualquer coisa disso? fledge, spec-sync, corvid-ai, existem coisas já prontas. Por que não apenas conectar o que já existe e seguir em frente.

Alguns motivos, e todos são verdadeiros ao mesmo tempo.

O primeiro é que o domínio é completamente novo. Construir ferramentas *para um mundo de agentes e humanos* não é algo resolvido que você pode ir comprar. Quase tudo lá fora é human-first, ou o que é mais recente é agent-first, e a coisa que realmente quero, first-class para ambos, desde o início, na maior parte ainda não existe. Então não estou reinventando rodas. Não há rodas. Se quero uma ferramenta construída sobre o pressuposto que fico argumentando, tenho que construí-la, porque as pessoas que vieram antes de mim estavam construindo para um mundo diferente.

O segundo é que construir na minha própria pilha prova isso. É a parte que me importa mais do que pode parecer. Você pode escrever um belo README afirmando que suas ferramentas são boas. Ninguém deveria acreditar em você. O que eles deveriam acreditar é que você construiu coisas reais nelas e as coisas funcionam. Então eu faço. Construo no fledge, spec-sync e corvid-ai porque carregar peso real é o único teste honesto de se elas aguentam. Se a pilha for boa, as coisas que construo nela são boas, e se a pilha for ruim, descubro rápido, porque sou eu o preso com ela.

O terceiro é simples: construir é como eu entendo. Só confio em uma dependência se poderia tê-la escrito eu mesmo, e isso não é um slogan. É uma década de recibos. AppState, a biblioteca de state-and-dependency-injection que ainda mantenho na versão 3.0; Cache; Fork para paralelizar trabalho assíncrono; os primitivos de composição de uma letra `c/o/t` sob 0xOpenBytes; CacheStore, a coisa do SwiftUI que se tornou AppState. Anos de pequenas bibliotecas Swift, cada uma uma coisa que construí porque queria entendê-la até o fundo, não apontar para uma caixa preta e torcer. Construir a coisa é como o conhecimento entra nas minhas mãos em vez de permanecer como uma ideia vaga do que alguma biblioteca provavelmente faz. As ferramentas na pilha são ferramentas que posso abrir e mudar, porque coloquei cada parte delas lá.

Há um quarto motivo, e é mais estreito do que soa: quero ser cedo nisso. O espaço é novo, as rodas ainda não existem, e prefiro construir as ferramentas para ele e estar

errado sobre algumas do que esperar que alguém me entregue as certas. Essa é a aposta.

Nenhum desses motivos carregaria sozinho, mas juntos são a razão pela qual não estou apenas colando ferramentas de outras pessoas numa pilha. As ferramentas são o ponto. São a coisa em que realmente estou tentando ser bom.

Dois tipos de certeza

Existem duas coisas diferentes que as pessoas agrupam como "o quanto temos certeza sobre esse código," e quero separá-las, porque me importo com ambas e elas não são a mesma coisa de forma alguma.

A primeira é risco, e risco eu quero estático. Determinístico. Sem modelo no loop. É para isso que augur serve. Você lhe entrega uma mudança e ele pontua o quão arriscada é, da mesma forma todas as vezes, com base em sinais que pode nomear. E "sinais que pode nomear" é o ponto inteiro, então deixe-me nomeá-los: o diff toca terreno sensível (auth, crypto, pagamentos, migrações, CI, dependências), o código mudou sem que os testes mudassem junto, esses são arquivos com histórico de reverts, alguém realmente os possui. Cada um é um sim-ou-não que você poderia verificar à mão. Some-os com pesos documentados e você obtém um número, não uma sensação. A razão pela qual tem que ser estático é a razão inteira pela qual vale alguma coisa: se a coisa que decide se o código é perigoso é em si um modelo de linguagem dando a você uma sensação sobre isso, você não mediu o risco, apenas moveu a adivinhação uma caixa adiante. Uma pontuação de risco na qual você pode confiar é aquela que diz a mesma coisa amanhã que disse hoje. Então esse fica sendo uma coisa fixa e repetível sobre a qual você pode raciocinar. Exploro como realmente funciona no livro Building Agents; aqui o ponto é apenas que risco é o lado *estático*, e é estático porque é uma soma de sinais nomeados que você pode apontar.

O segundo é confiança, e confiança eu realmente quero *do agente*. Este é o que adoro, e é o oposto do estático. É o agente me dizendo o quão certo ele está sobre a coisa que acabou de fazer. E a razão pela qual adoro não é realmente o número. É o que pedir o número faz com o agente. Quando você faz um agente colocar uma avaliação de confiança em seu próprio trabalho, ele tem que parar e olhar para trás para o que fez. A avaliação reformula o trabalho para ele. Ele não pode simplesmente produzir e seguir em frente; ele tem que virar e avaliar.

E a parte que o torna genuinamente útil é a granularidade. Um agente vai felizmente dar a você um número de confiança para a mudança inteira. Tudo bem, mas isso é quase grosso demais para agir. Onde fica bom é quando você diminui. Uma avaliação de confiança em cada arquivo. Em cada mudança individual. Agora você tem a própria leitura do agente sobre exatamente quais partes ele tem certeza e quais partes não, e esse é o mapa que você realmente quer. Ele aponta você exatamente para os

pontos em que o próprio agente está nervoso, em suas próprias palavras, antes que qualquer outra pessoa tenha olhado.

Então esses são dois instrumentos diferentes. Risco é estático, determinístico, meu para confiar porque nunca se move. Confiança é do agente, viva, útil precisamente porque vem da coisa que fez o trabalho e a fez olhar de novo. O erro é embaçá-los: chamar a pontuação de risco estático de "confiança", ou esperar que a confiança do agente seja determinística. Eles estão respondendo a perguntas diferentes. Um pergunta "quão perigosa é essa mudança," e você quer uma máquina que não pode ser convencida a mudar sua resposta. O outro pergunta "o quanto você tem certeza sobre o que acabou de escrever," e você especificamente *quer* que aquele que escreveu responda, arquivo por arquivo, porque o perguntar é metade do valor.

O único momento em que ter ambos realmente importa é quando eles discordam. Uma mudança que pontua baixo em risco mas onde o agente avalia sua própria confiança como baixa é exatamente o sinal que você perderia com um instrumento. Baixo risco significa que os sinais estavam silenciosos: sem auth, sem migrações, sem churn. Mas a baixa confiança do agente significa que ele sabia que estava adivinhando a lógica, que algo na mudança parecia incerto mesmo que nada que tocou fosse categoricamente perigoso. Essa é a mudança que um instrumento deixa passar e o outro para, e é a razão inteira pela qual você mantém ambos.

Mantenha-os separados e ambos funcionam. Misture-os e você fica com um gate determinístico no qual não pode confiar porque um modelo está nele, ou uma etapa de reflexão que você drenou de vida ao exigir que nunca mude. Então os construo como dois instrumentos separados. O gate de risco permanece fixo; a confiança do agente permanece viva. É a única forma de obter ambos.

O contrato é a especificação

A spec é a coisa que fica entre o humano e o agente, e já disse por que ela importa. É a fonte única de verdade, a coisa que impede o agente de se desviar, a rampa de entrada para código legado, a linha que mantém o humano no controle. Tudo isso se sustenta. O que quero fazer aqui é ir um nível abaixo, para o que uma spec realmente *é* quando você constrói dessa forma, porque há uma forma nisso que é fácil de perder.

Antes da teoria, aqui está como uma realmente parece. Um `*.spec.md` para spec-sync é um arquivo markdown com seções nomeadas, e um pequeno, digamos o contrato para uma única função que restringe um número a um intervalo, lê mais ou menos assim:

```
# clamp

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

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

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

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

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

É isso: algumas seções rotuladas, sem narração em prosa. Specs reais adicionam o restante das seções obrigatórias (Dependencies, um Change Log), mas a forma já é visível aqui: ela declara *o que é verdadeiro*, em uma forma que uma máquina pode comparar com o código. Agora a teoria.

Comece com o que entra na própria spec. A spec é o contrato. Propósito, a superfície pública, os invariantes, os exemplos comportamentais, os casos de erro: a forma verificável da coisa. O que é, não uma história sobre isso. No segundo em que uma

spec se torna uma parede de prosa descrevendo o código, está morta, porque prosa se afasta do código no momento em que qualquer um dos dois se move e agora você tem duas coisas que discordam e nenhuma forma de dizer qual está mentindo. Então a spec é mantida compacta e contratual de propósito. É a parte que uma máquina pode comparar com o código.

Também é intenção, não implementação. A spec diz o que a coisa deve fazer e por que deve fazê-lo. Não diz como. No momento em que uma spec começa a ditar implementação, ela para de guiar o agente e começa a lutar com ele. Você pegou a parte em que o agente é bom, o trabalho árduo de descobrir *como*, e a fixou de cima sem razão. Mantenha a spec no nível da intenção e o agente tem espaço para realmente trabalhar. Declare o que deve ser verdadeiro. Deixe-o construir a partir disso.

Há mais uma coisa que a spec precisa fazer, e é a parte que as pessoas pulam: ela precisa dizer como você saberia. Intenção não é apenas o que deve ser verdadeiro, é o que você aceitaria como prova de que é verdadeiro, e o que te faria dizer que não é. Uma spec que nomeia o primeiro e não o segundo é uma spec que o agente pode satisfazer de um jeito que você não quis dizer, construindo algo que corresponde às palavras e perde o ponto, sem nada para capturá-lo porque você nunca escreveu como seria capturá-lo.

É para isso que os exemplos comportamentais e os casos de erro realmente servem. `clamp(12, min: 0, max: 10) == 10` é um sinal de aceitação: execute e você sabe. A falha de pré-condição em `min > max` é um sinal de rejeição: diz como ficar errado se parece, de forma concreta, em vez de "trate entrada ruim." Ambos são observáveis, e nenhum dos dois precisa de mim na sala para julgar. Então quando você escreve o contrato, escreva as duas metades. O sinal de aceitação é você decidindo o que "pronto" significa com antecedência. O sinal de rejeição é você decidindo o que "quebrado" significa antes de estar olhando para ele e tentado a dizer que está bem.

Mas aqui está a forma que acho que as pessoas perdem: não é um arquivo. Há a spec, e depois há arquivos complementares ao redor dela, e cada um carrega um tipo diferente de conhecimento que a própria spec não deveria ter.

Comece com a spec em si. A spec é a que está ligada de perto ao código: a imagem não-código do que o código realmente faz, próxima o suficiente dele para que spec-sync possa manter os dois juntos um a um. Esse é o arquivo obrigatório, o verificável. Ao redor dele ficam arquivos complementares, e cada um carrega um tipo de conhecimento que a spec não deveria ter.

Penso neles pelo tipo de conhecimento que contêm em vez de por qualquer nome de arquivo fixo. Há o tipo de **requisitos**: o de alto nível, escrito como um dono de produto escreve, as histórias de usuário, o "como usuário, quero...", a intenção no nível do negócio. Há um tipo de **contexto**: contexto para o agente, o material que você só precisa ter escrito em algum lugar para que ele tenha. Há **design**: as notas de design, o pensamento, o porquê-está-moldado-assim que não pertence ao contrato compacto mas você não quer perder. E há **testes**: como você verificaria que a coisa faz o que a spec diz. Não leia esses como quatro nomes de arquivo abençoados; leia-os como quatro tipos de conhecimento que querem viver ao lado da spec em vez de dentro dela. O que importa é a divisão, não os rótulos.

E roda em ambas as direções. Um humano pode escrever os requisitos e deixar o agente transformá-los na spec; ou um humano escreve a spec e os requisitos derivam dela. Intenção e contrato, em qualquer ordem, com o agente capaz de se mover entre os dois.

A razão pela qual essa divisão importa é que mantém o contrato limpo enquanto ainda dá ao agente tudo o mais que precisa. A spec permanece pequena e verificável, a parte que spec-sync pode realmente comparar com o código. Tudo que é real mas *não* verificável (os requisitos de negócio, o raciocínio de design, o contexto em andamento, como você o testaria) vive ao lado da spec em vez de inchá-la. Então você obtém um contrato compacto o suficiente para impor, cercado pelo conhecimento mais solto que um agente precisa para fazer o trabalho bem, e os dois não se contaminam.

Então a spec funciona como uma interface, não como um documento que o agente folheia e ignora. Um contrato compacto contra o qual ele constrói, mais os complementos que carregam o restante, com uma linha clara entre a parte que uma máquina verifica e a parte que um humano escreveu para que nada se perdesse.

Código é barato, confiança é escassa

Agentes tornaram o código barato. Esse é o fato que reorganiza tudo o mais, então comece por aí.

Quando um humano tinha que digitar cada linha, escrever o código era lento, e a lentidão estava fazendo um trabalho oculto. Você não conseguia escrever uma coisa sem entendê-la parcialmente. O ato de escrever também era o ato de verificar. Eles vinham juntos, de graça, porque a mesma pessoa fazia os dois na mesma velocidade. Esse pacote acabou de quebrar. Um agente entrega um pull request de quarenta arquivos antes do seu café ficar pronto, e o escrever e o verificar se separam. O código foi produzido. Ninguém o entendeu no caminho. Produzi-lo parou de significar que alguém o verificou.

Então vira. O código não é mais o gargalo. É barato, tem tanto quanto você quiser. A coisa escassa é a confiança. Quem realmente olhou para essa mudança, e com que profundidade, e ela deveria ter permissão de pousar. Essa é a pergunta que é cara agora, e é a pergunta que as ferramentas têm que responder, porque a velha resposta ("bem, alguém escreveu, então alguém entendeu") não é mais verdadeira.

O que significa que a revisão tem que mudar de forma. Você não pode carimbar um PR de quarenta arquivos, e também não consegue honestamente ler tudo, e fingir que fez é o perigo real. Então as ferramentas têm que triar sua atenção: apontar você para a parte arriscada e deixar você gastar seu julgamento lá em vez de espalhá-lo tão fino pela coisa toda que não vale nada. O recurso escasso é um humano prestando atenção real, e você o gasta onde importa.

Isso não é hipotético para mim. `fledge review`, que passa a mudança por um modelo através do meu cliente corvid-ai, encontrou um bug real no meu próprio trabalho que de outra forma teria sido publicado, e spec-sync capturou desvio real entre código e sua spec mais de uma vez. Esse é o recibo em que confio mais do que qualquer argumento: a camada de confiança encontrou coisas que um eu mais rápido teria deixado passar. A versão cotidiana disso é uma lane que chamo de `verify` (format, lint, test, build) que roda antes de qualquer coisa ser mesclada, o mesmo gate que o próprio runner do agente tem que passar. Nada disso acelera o escrever. Tudo isso é o trabalho que o escrever barato empurrou para baixo, transformado em algo que realmente roda.

E uma vez que agentes estão pousando mudanças, você precisa de um registro. Um rastro portátil de quem ou o que verificou cada mudança, e ele não pode viver em algum painel SaaS que é desligado, porque então sua proveniência desaparece no dia em que alguém para de pagar a conta. Ela tem que andar junto com o código em si. Quando um agente pousa algo, deve haver uma resposta durável para "quem garantiu isso," e essa resposta tem que sobreviver à ferramenta que a produziu.

Mas há uma peça maior do que confiança, e é a que as pessoas pulam. Testes. Testes de verdade, configuração, todo o aparato em torno do código. Aqui está a armadilha: agentes te tornam dez vezes mais rápido em *escrever*, e nada mais automaticamente acelera para combinar. Os testes ainda têm que ser escritos e executados. A configuração ainda tem que acontecer. A revisão ainda tem que acontecer. Então se você deixar o desenvolvimento ir dez vezes mais rápido e deixar todo o resto no ritmo antigo, tudo que você fez foi mover o gargalo. O código não é mais a parte lenta; os testes são, a verificação é, a confiança é. Você não pode multiplicar por dez o escrever e não multiplicar por dez tudo ao redor. O trabalho não desapareceu. Ele se moveu para baixo, para as partes que nunca foram o gargalo antes e de repente são.

Código barato não faz o trabalho desaparecer. Torna o escrever barato e empurra o peso para tudo que decide se o escrever barato presta. Esse trabalho sempre esteve lá. Estava escondido atrás de quão lento o escrever costumava ser, e agora a lentidão se foi e lá está: o trabalho inteiro, de pé onde a parte fácil costumava estar.

Os humanos sobem de nível

Se as ferramentas e as specs e os trilhos de confiança simplesmente *existirem*, ordinários, o modo padrão de construir as coisas, então o humano sobe de nível. Sobe até a intenção. Você para de ser a pessoa que digita a implementação e se torna a pessoa que decide o que deve existir e por quê. Escrever o código à mão se torna opcional em vez de ser o trabalho. Essa é a direção para a qual tudo isso aponta, e quero ter cuidado com a forma como descrevo, porque é fácil arredondar para a versão assustadora.

O título não é "agentes rodam tudo de forma autônoma." O humano sobe um nível e o agente faz o trabalho árduo embaixo, contra uma spec, em aberto onde você pode verificar. O humano é elevado; ninguém é substituído. O que você obtém lá é uma equipe real, passando trabalho de um lado para o outro, cada lado fazendo o que é realmente bom. E isso se torna o padrão. Não minha pilha, não um nicho que algumas pessoas fazem. Apenas a forma como construir funciona.

Aqui está a coisa em que continuo chegando sobre por que o humano permanece nisso. A maioria das coisas boas que a IA gera agora tem uma pessoa no loop tornando-as boas: escolhendo, corrigindo, decidindo o que conta como bom em primeiro lugar. E sim, em algum ponto os modelos ficam bons o suficiente para fazer coisas boas mais ou menos por conta própria. Mas há uma coisa central que os humanos têm que acho que não cai tão facilmente: somos bons em dirigir. Em intenção e propósito. Uma IA não tem realmente um propósito próprio, não até que lhe seja dado um, não até que encontre um. Ela precisa de um humano para *ser* o propósito. O modelo pode fazer o trabalho uma vez que há um porquê; ele não gera o porquê. Essa é a parte que permanece nossa por mais tempo do que o digitar.

Agora deixe o quadro avançar, porque vai a algum lugar estranho e acho que realmente vai lá. Você roda seus próprios agentes para fazer o trabalho. Um agente pode viver por conta própria e simplesmente fazer coisas. Talvez você o aponte para uma área geral para investigar, talvez ele fale com outros agentes, talvez você coloque algum dinheiro nele e ele vai trabalhar, fala com outros agentes, paga outros agentes pelo que fazem. As pessoas rodam agentes e os agentes ganham dinheiro. Isso não é um recurso secundário. É o desenvolvimento orientado a agentes como uma economia real, com humanos definindo o propósito e os agentes trabalhando arduamente embaixo, negociando uns com os outros.

E nesse mundo os humanos são os que garantem que tudo funciona e que alguém ainda entende. Porque em algum ponto o próprio código para de importar da forma que importa agora. Você não está lendo cada linha, o agente escreveu a maior parte, o volume passou do que qualquer pessoa acompanha. Tudo bem. Mas aqui está a linha que não vou abrir mão: um humano ainda tem que ser capaz de entrar no código e mudá-lo. Tem que. O dia em que você não consegue abri-lo e consertá-lo sozinho é o dia em que você entregou algo que não deveria ter.

Por isso tem que ser limpo. Limpo em todos os níveis. Legível e mutável por um humano e por um agente, até o fundo. First-class para ambos nunca foi apenas sobre os agentes frágeis de hoje tropeçando nas ferramentas de hoje. É a coisa que tem que se sustentar mesmo na versão em que agentes fazem a maior parte da construção. *Especialmente* lá. A única forma de "o código não vai importar" não se tornar silenciosamente "você perdeu o controle do código" é se o código permaneceu limpo o suficiente, todo o caminho, para que um humano ainda possa abri-lo e mudá-lo. Esse é o trabalho inteiro, mantido aberto.

Comece na segunda-feira

Digamos que você leu tudo isso e acredita. Humanos e agentes nas mesmas ferramentas, first-class para ambos, a spec como o contrato, os trilhos de confiança. O que você realmente faz na segunda-feira de manhã? Aqui está a resposta honesta, na ordem em que eu faria.

Aponte um agente em uma das suas ferramentas e observe-o engasgar. Essa é a primeira jogada, e é a que mais ensina, porque o agente é a disciplina. Você acha que sua ferramenta está boa. Você a usou por meses, suas mãos a conhecem. Entregue-a a um agente sem mãos e sem olhos e sem memória entre execuções, e cada lacuna que você vinha cobrindo silenciosamente fica de repente visível. O prompt em que ela trava. A saída que ele não consegue analisar. O comando que ele não consegue descobrir. Você não tem que adivinhar onde sua ferramenta assumiu um humano. O agente mostra, imediatamente, ao falhar exatamente lá. Corrija o que encontrar. Esse único exercício vai lhe ensinar mais sobre construir para ambos do que o livro inteiro fez.

Depois vá usar as ferramentas que já são construídas dessa forma, para que você não esteja começando de uma página em branco. spec-sync, para o contrato. fledge, para os comandos. augur, para o risco. attest, para o registro de quem garantiu. Elas não são a única forma de fazer nada disso, mas existem, são construídas sobre o pressuposto sobre o qual este livro trata, e vão mostrar a você a forma disso funcionando em vez de você ter que derivá-la. Use-as em um projeto real. Sinta onde ajudam e onde não ajudam.

E traga seu agente para isso. Faça-o realmente aprender as ferramentas e usá-las: deixe-o conduzir spec-sync e fledge, deixe-o trabalhar contra uma spec, deixe-o rodar as verificações. Esse é o loop que você está tentando construir: um humano definindo a intenção, um agente fazendo o trabalho através de ferramentas que o tratam como um usuário real, a spec mantendo-o honesto. Você não obtém esse loop lendo sobre ele. Você o obtém rodando em algo que lhe importa e observando onde ele se sustenta.

O que você está realmente fazendo, sob tudo isso, é construir e colaborar em ferramentas e confiança. Esse é o jogo inteiro. As ferramentas, para que um humano e um agente possam ambos fazer o trabalho first-class. A confiança, para que você

possa realmente acreditar no que pousou. Essas duas coisas são o que o livro inteiro esteve circulando, e são as duas coisas que valem o seu tempo para construir.

Aponte um agente em uma das suas. Observe onde ele engasga. É aí que você começa.

Sobre o Autor

0xLeif (leif.algo) constrói em aberto. Uma década de pequenas bibliotecas Swift composíveis como AppState, Cache e Fork. O laboratório CorvidLabs. Uma pilha de ferramentas para agentes que na maioria das vezes começou como "eu queria 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 o espaço onde essas ideias são testadas e moldadas pelo debate.

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

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

Colofão

Gerado a partir de Markdown, construído com bookgen, um pequeno pipeline puramente em Rust (sem Python).

Conduzido por entrevistas e assistido por IA; editado e verificado manualmente. Escrito sem travessões. Capa e arte de capítulo das coleções Corvid e Nature no Algorand.