RENATO.

Seu agente de IA escreve rápido.
O Renato faz ele escrever **certo**.

Um cérebro central que dá **memória persistente**, **método** e **gates de engenharia** ao agente de programação — test-first obrigatório, nenhuma primeira versão aceita, nenhum "pronto" sem evidência.

renato — sessão real

```
> renatao evals
# Renatao Evals — 21/21 canarios passaram (100%)
## governanca — 7/7
✓ identidade MCP carrega Segunda Versao      ✓ checkpoint pre-v2 existe
✓ handshake MCP carrega Segunda Versao       ✓ AGENTS.md carrega Segunda Versao

> renatao doctor --deep
[OK] evals-canario: 21/21    [OK] indice semantico: 503/503
[OK] servidor MCP importa    [OK] circuit breaker: CLOSED
Doctor --deep: TUDO VERDE
```

"A vida é isso: conhecimento só serve se compartilhado."

ERICK — CRIADOR DO RENATO

01 AS FRASES DA CASA

Manifesto — o que a gente acredita e mecaniza

Cada frase abaixo nasceu de um erro real e virou **regra executável**. Nenhuma é decoração: todas têm um gate, um checklist ou um teste por trás.

"Funciona" é opinião. "Esse comando roda e sai isso" é fato.

CRITÉRIO DE ACEITE COMO CONTRATO

Conselho se esquece. Mecanismo sobrevive.

A ESCADA DE ENFORCEMENT

A primeira versão que funciona ancora o pensamento. A v1 nunca é entregue.

REGRA DA SEGUNDA VERSÃO

O autor jura. O adversário acha.

REVISÃO ADVERSARIAL

Container é gado, não bicho de estimação — o que precisa sobreviver mora fora dele.

GUIA DO FLUXO DE DEPLOY

Backup que nunca foi restaurado é fé, não backup.

GUIA DE SEGURANÇA

A nota só sobe com mecanismo — nunca com promessa.

SELO DA CASA

Avaliação, não dogma: se PHP for a melhor saída, PHP vence.

DECISÃO DE STACK

02 O PROBLEMA · A VIRADA · A TESE

Velocidade sem método é protótipo disfarçado de produto

O problema. Ferramentas de IA escrevem código numa velocidade impensável há dois anos. Mas velocidade não é qualidade. O agente entrega a primeira versão que funciona — e quase todo mundo aceita. O resultado é protótipo disfarçado de produto: "funciona na minha máquina", segredo vazado no commit, e o mesmo erro repetido projeto após projeto, porque a IA não lembra do que já foi decidido.

A virada. Disciplina de engenharia não pode depender de força de vontade — ainda mais quando quem escreve o código é uma IA confiante e apressada. Documento se ignora, checklist se pula, boa intenção se esquece. Tem que virar sistema. O Renato nasceu dessa ideia: transformar boas práticas em **mecanismos que acontecem sozinhos**, não em conselhos que se esquecem.

A tese. Toda prática de engenharia sobe uma escada de enforcement — e só descansa quando vira gate, verificação automática ou métrica no tempo. O trabalho do Renato é empurrar cada regra da casa o mais para a direita possível dessa escada.

aconteceu de verdade · a regra que pediu a própria regra

Quando a Regra da Segunda Versão foi implantada, a primeira versão da implantação parecia completa — e foi auditada adversarialmente antes do "pronto". Resultado: **6 caminhos sem a regra**, incluindo o principal (o canal MCP que o agente do IDE realmente usa). A v1 foi criticada e reescrita. A regra exigiu a própria segunda versão — e provou o próprio ponto.

03 O QUE É, TECNICAMENTE

Ele não escreve código — injeta contexto e cobra método

O Renato é um **cérebro central externo, IDE-first**: um servidor MCP + CLI determinística (Python) que se conecta ao agente do seu editor (Antigravity, VS Code, Cursor). A identidade, os checkpoints e os gates chegam **dentro da conversa do agente** — não dependem de ele abrir arquivo nenhum.

Ele dá ao agente três coisas que faltam: **memória que persiste** entre sessões e projetos, com recall híbrido (busca textual FTS5 + embeddings semânticos) e colheita automática ao concluir — a IA para de reinventar o que já foi resolvido; um **acervo de 1.200+ skills** indexadas, servidas sob demanda para não inflar o contexto; e **protocolos que cobram método** a cada passo do trabalho.

O **roteamento é 100% determinístico**: a tarefa é classificada por keywords ponderadas — sem LLM no caminho crítico. Mesmo input, mesmos protocolos, sempre. Determinismo aqui é feature: o agente pode ser criativo; o processo, não.

E o sistema se cuida

— **720+ testes** sobre o próprio Renato e **21 evals-canário** ponta a ponta — inclusive **governança**: se uma regra sumir de um ponto de injeção, o sistema acusa sozinho.

— **Backup do cérebro** com verificação de integridade e rotação; **sandbox** de shell; **detector de vazamento de segredos**; **teto diário de custo** em chamadas pagas.

— **Manutenção mensal automática**: self-test profundo, evals, auditoria de dependências, varredura de produção e colheita de memória — sem depender de alguém lembrar.

04 DENTRO DO CÉREBRO · COMO UMA TAREFA VIRA MÉTODO

O roteador é uma função, não um palpite

Quando uma tarefa chega, **nenhum modelo de linguagem decide o que fazer com ela**. A classificação é determinística — e isso é uma escolha de engenharia, não uma limitação.

1	**o pedido é normalizado**	acentos, caixa e ruído saem do caminho antes de qualquer decisão	passo
2	**pontuação por keywords ponderadas**	cada domínio — deploy, revisão, segurança, projeto novo... — tem um conjunto de termos com pesos; a **soma** decide o vencedor	passo
3	**o domínio define o pacote**	protocolos, checkpoints e gates que serão **cobrados** naquela tarefa — não sugeridos	GATE
4	**skills sob demanda**	das **1.200+ skills indexadas**, só as mais relevantes entram — contexto enxuto em vez de manual inteiro	passo
5	**injeção via MCP**	identidade, método, memórias e gates entram **dentro da conversa do agente** — ele não precisa abrir arquivo nenhum	ENTREGA

Por que determinismo é feature

Mesmo pedido, mesmo processo — sempre. Um roteador probabilístico erraria diferente a cada dia, e ninguém audita palpite: aqui, auditar é ler uma tabela de pesos, não interrogar um modelo. O custo é zero e a latência é desprezível, porque não há chamada de rede no caminho crítico.

A divisão é deliberada: **a criatividade fica onde deve ficar** — no código que o agente escreve, nas abordagens do torneio, na solução do problema. O processo que cobra teste, evidência e revisão não improvisa. É a mesma lógica de um checklist de aviação: o piloto decide muito; o checklist, nada.

regra da casa · zero llm no caminho crítico

Roteamento, gates, selo e evals são **funções puras**: entrada → saída, testáveis uma a uma. Chamada de modelo é periférica e opcional (vetores de memória, geração de imagem) — **nunca decide o método**.

05 DENTRO DO CÉREBRO · MEMÓRIA

Memória que atravessa sessões e projetos

O agente nasce amnésico: fechou a conversa, esqueceu a decisão, o erro e a lição. O Renato mantém um acervo que persiste — e que **volta sozinho na hora certa**.

1	**colheita automática**	ao concluir a tarefa, decisões e lições são extraídas e gravadas — **sem depender de alguém lembrar de anotar**	GATE
2	**leak-scan na ingestão**	credencial detectada vira **[REDACTED] antes de tocar o disco** — o acervo nunca guarda segredo em texto	GATE
3	**dedup semântico**	a mesma lição aprendida duas vezes não polui o acervo duas vezes	passo
4	**índice duplo**	o texto entra no índice **léxico** (FTS5) e vira um **vetor semântico**; um hash do texto evita re-embutir o que não mudou	passo
5	**recall híbrido**	busca textual + similaridade de cosseno entre vetores, **fundidas por RRF** (fusão de rankings recíprocos)	CORE
6	**injeção na sessão seguinte**	as memórias relevantes ao tema da nova tarefa entram na conversa — em qualquer projeto	passo

Por que dois índices

O léxico acha o **termo exato**: "coolify", o nome da função, a flag do comando. O semântico acha o **conceito**: "app caiu depois do deploy" encontra "healthcheck falhou em produção" sem dividirem uma palavra sequer. O RRF junta as duas listas sem calibração frágil de pesos: quem ranqueia bem nas duas, sobe.

Honestidade sobre os dados: acervo, índice, segredos e backups ficam na máquina. Para gerar o vetor, o texto da memória viaja a uma API de embedding — **depois** do leak-scan. Sem chave ou sem internet, o recall degrada para busca textual pura: menos esperto, **nunca quebrado**.

O QUE FICA · O QUE VIAJA · O QUE DEGRADA

FICA NA MÁQUINA	VIAJA (OPCIONAL)	OFFLINE
acervo SQLite, índices, segredos, backups	texto da memória → vetor, sempre após o leak-scan	recall vira busca textual pura — zero regressão

06 O FLUXO DE UMA TAREFA

Do pedido ao pronto: nove passos, seis gates

1	**session_start**	identidade, memórias, playbooks e modo de trabalho injetados direto na conversa do agente	passo
2	**recall + plano**	busca o histórico antes de reinventar; anuncia o plano; o humano aprova	passo
3	**pre-code**	classifica **antes** da 1ª linha: trivial / relevante / crítico — reclassificar depois é falso-feito	GATE
4	**test-first**	escreve o **teste** → roda → confirma a **falha esperada** → só então implementa	GATE
5	**pre-v2 · Regra da Segunda Versão**	v1 → 3 críticas → reescrita v2 (crítico: torneio de 3 abordagens antes)	GATE
6	**pre-test / pre-commit**	suíte verde de verdade (saída real); trilha v1→v2 conferida; commit semântico	GATE
7	**adversary + ship-check**	tenta **quebrar** o próprio trabalho; evals do projeto bloqueiam o deploy se falharem	GATE
8	**deploy + postdeploy**	HEALTHCHECK obrigatório; smoke real no endpoint em produção; app caído vira incidente	GATE
9	**task_complete**	recusa "pronto" sem evidência recente; a memória do que foi feito é colhida automaticamente	GATE

o detalhe que muda tudo

Os gates não são texto que o agente pode ignorar. O **task_complete recusa recibo em branco**; o **ship-check bloqueia deploy com eval falhando**; o **preflight reprova Dockerfile sem healthcheck**. Desobedecer dá trabalho — obedecer é o caminho fácil.

07 OS DOIS CICLOS CENTRAIS

Test-first e a Segunda Versão, lado a lado

CICLO TEST-FIRST (XP)

1 · ESCREVE O TESTE
antes do código

↓

2 · RODA → FALHA
pelo motivo esperado

↓

3 · IMPLEMENTA
a menor mudança

↓

4 · RODA → VERDE
evidência real

sem teste falhando primeiro, não há implementação

REGRA DA SEGUNDA VERSÃO

v1 — RASCUNHO
nunca é entregue

↓

3 CRÍTICAS CONCRETAS
trocar de papel e atacar

↓

v2 — REESCRITA
repensar, não renomear

↓

TRILHA NO RECIBO
[v1 → críticas → v2]

crítico: antes disso, torneio de 3 abordagens julgadas

Por que reescrever em vez de refazer 5 vezes?

Reescrever do zero repetidas vezes tende a produzir variações da mesma ideia — a IA regride à abordagem mais provável dela. O ganho real vem de **pontos de partida diferentes + julgamento explícito**: no código crítico, 3 abordagens partindo de ângulos declarados (simplicidade, robustez, performance) são julgadas por tabela — correção, simplicidade, testabilidade, risco — e a vencedora herda o melhor das perdedoras, antes de ainda passar pela crítica e reescrita.

A classificação vem antes de tudo: **trivial** (typo, texto, config) fica fora da regra; **relevante** — o default de qualquer lógica — exige a segunda versão; **crítico** (auth, pagamento, dados) exige o torneio. Na dúvida, é relevante. Reclassificar para baixo depois de pronto é assinatura de falso-feito. Exceções honestas: emergência em produção (conserta primeiro, a v2 vira follow-up imediato registrado) e código vindo do scaffold da casa.

08 OS 8 PROTOCOLOS · PARTE 1

Cada um nasceu de um erro real — e virou mecanismo

01 Test-first como gate, não como conselho

Nenhum código de produção nasce antes do teste: escreve-se o teste → roda-se → **confirma-se a falha pelo motivo esperado** → só então implementa. O fluxo é cobrado em artefato próprio e verificado na conclusão — o sistema recusa "concluído" sem evidência de execução recente de testes, com janela de tempo: um relatório velho não prova que os testes rodaram para ESTA tarefa.

MECANISMO `task_complete exige FULL_TEST_RUN recente (mtime < 30 min) · gate no checkpoint pre-test`

02 Regra da Segunda Versão: a v1 nunca é entregue

A primeira versão que funciona ancora o pensamento. Todo código relevante segue **v1 → 3 críticas concretas → reescrita v2**, com a trilha registrada no recibo de execução. Código crítico (auth, pagamento, dados) exige **torneio**: 3 abordagens partindo de ângulos distintos declarados antes, julgadas por tabela — a vencedora herda o melhor das perdedoras.

MECANISMO `checkpoint pre-v2 · pre-commit cobra a trilha [v1 → críticas → v2] · adversarial caça v1 sem crítica`

03 Critério de aceite como contrato

Antes da primeira linha: **qual comando vou rodar e que saída espero ver?** Ao final, a saída real é colada no recibo — e o task_complete **bloqueia** conclusão com recibo em branco. "Funciona" é opinião; "esse comando roda e sai isso" é fato.

MECANISMO `GSD exige tabela comando → saída · task_complete recusa EXECUTION_RECEIPT template`

04 Revisão adversarial antes de aprovar

O agente troca de papel: **3 hipóteses de como o trabalho pode estar quebrado**, cada uma testada de verdade; caça às assinaturas de falso-feito (sucesso sem saída colada, teste que não testa, `except: pass`, v1 entregue sem crítica); lente de produto quando há UI — primeiro uso sem manual, mensagens de erro que ensinam, estado vazio que orienta. Todo deploy aciona automaticamente.

MECANISMO `roteia em revisar/auditar + todo deploy · ship-check reporta ausência da revisão`

09 OS 8 PROTOCOLOS · PARTE 2

Do primeiro commit à produção vigiada

05 Evals por projeto

Cada projeto define casos determinísticos **comando → saída esperada** (EVALS.jsonl). Rodam a cada mudança e **bloqueiam o deploy** se falharem — o único termômetro honesto de "a ferramenta ainda funciona". O scaffold já cria os primeiros casos na stack detectada.

MECANISMO `renatao evals roda canários + casos do projeto · eval falhando = BLOQUEADOR no ship-check`

06 Deploy com gates reais

Preflight que **reprova** Dockerfile sem HEALTHCHECK; auditoria de dependências (pip-audit / npm audit) contra CVEs conhecidas; smoke pós-deploy batendo no healthcheck **real** em produção; varredura periódica que transforma app fora do ar em **incidente automático** no projeto de origem.

MECANISMO `coolify-preflight FAIL · deps-audit rc=1 em severidade alta · postdeploy + production-sweep`

07 Selo da Casa

Nota **A–F determinística** por projeto agregando: evals verdes, testes, README, dependências auditadas, segredos fora do código, healthcheck. Com histórico em série temporal e tendência no briefing diário. A nota só sobe com mecanismo, nunca com promessa.

MECANISMO `renatao seal · SEAL_HISTORY.jsonl · tendência no standup`

08 Nascer bem custa um comando

O scaffold gera o projeto com o kit embutido — estrutura canônica, teste smoke passando, evals iniciais, logging estruturado, Dockerfile com healthcheck, .env.example, git iniciado — e **se recusa a rodar sem a decisão de stack registrada**: avaliação comparativa, nunca dogma. Se PHP for a melhor saída para o caso, PHP vence.

MECANISMO `renatao scaffold --tipo api|cli|estatico|fullstack · exige --decisao (Gate 2) · DECISIONS.md no projeto`

`aconteceu de verdade · a primeira medição achou o que estava invisível`

No dia em que o selo e a auditoria de dependências entraram no ar, foram apontados nos projetos reais da casa: **18 vulnerabilidades (3 graves)** num frontend que parecia saudável, dois Dockerfiles sem healthcheck e a ausência de evals — com o caminho exato para subir cada nota. Nenhuma promessa: um checklist acionável por projeto.

10 ANATOMIA DO .RENATAO/ · A PAPELADA QUE TRABALHA

O método não mora na conversa: vira arquivo no projeto

Conversa evapora; **arquivo fica, versiona e audita**. Cada gate do Renato lê um artefato — nunca a boa intenção de quem escreveu. Todos são texto puro (markdown/JSON), legíveis por humano e por máquina, dentro do próprio repositório.

GSD_TASK.md
o contrato da tarefa: objetivo + critério de aceite (comando → saída esperada)

TEST_FIRST_PLAN.md
o teste planejado antes do código — e a falha esperada dele

EXECUTION_RECEIPT.md
o recibo: saída real colada; template em branco é recusado

FULL_TEST_RUN
prova de suíte verde com carimbo de tempo — relatório velho não vale

ADVERSARIAL_REVIEW
as 3 hipóteses de quebra e o resultado real de cada uma

DECISIONS.md
decisões de stack e arquitetura com o porquê e as alternativas descartadas

EVALS.jsonl
os casos comando → saída que vigiam o projeto a cada mudança

SEAL_HISTORY.jsonl
a série temporal da nota A–F — a direção importa mais que a foto

AGENTS.md
o manual que viaja com o repo: regras da casa para qualquer agente que chegar

Dois níveis, uma lógica

Por projeto, o `.renatao/` guarda o que pertence àquele código: contratos, recibos, decisões, evals, selo. Quem clona o repositório leva o contexto junto — inclusive um agente novo, de outra ferramenta, meses depois.

Na casa, o cérebro central guarda o que atravessa projetos: memória, skills, protocolos, histórico. A fronteira é deliberada: o que é do projeto viaja com o projeto; o que é da casa (e os segredos) **não sai dela**.

> auditoria sem arqueologia
>
> Seis meses depois, "por que escolhemos isso?" se responde com um arquivo — não com a memória de quem já saiu do projeto. O custo é minutos por tarefa; a alternativa é reconstruir contexto por horas, ou decidir de novo no escuro.

11 A VIAGEM DO CÓDIGO · GIT → COOLIFY → PRODUÇÃO

O que acontece num deploy, passo a passo

1	**ship-check antes do push**	testes verdes, evals, leak-scan de segredos, **HEALTHCHECK** no Dockerfile, dependências auditadas — reprovou, não embarca	GATE
2	**git push**	o git é o **único caminho de entrada** do código no servidor — editar direto lá é proibido; o próximo deploy apaga	passo
3	**o Coolify percebe e constrói**	webhook dispara, o código novo é puxado para o servidor e a **imagem** é construída seguindo o Dockerfile	passo
4	**troca de containers**	o novo sobe com as variáveis do painel injetadas, o **healthcheck confirma que está vivo** — só então o antigo morre; o proxy aponta o domínio pro novo	GATE
5	**postdeploy**	smoke batendo no **endpoint real** em produção, resposta guardada no recibo — deploy "que passou" não é deploy verificado	GATE
6	**production-sweep contínuo**	varredura periódica de todos os apps no ar: fora do ar vira **incidente automático** no projeto de origem — ninguém precisa perceber na mão	VIGIA

ONDE CADA COISA MORA — E O QUE SOBREVIVE AO DEPLOY

CÓDIGO no git ✓ sobrevive	**SEGREDOS (.env)** painel do Coolify ✓ sobrevive	**CONTAINER** recriado a cada deploy ✗ morre
DADOS DO BANCO volume no disco ✓ sobrevive	**UPLOADS** só com volume ⚠ configure	**BACKUP** fora do servidor ✓ plano B

o erro clássico que esta tabela mata

SQLite dentro do container sem volume = banco zerado a cada deploy. Tudo que precisa sobreviver mora **fora** do container: banco com volume próprio, backup em outro lugar.

12 PROJETO NOVO · A LINHA DE RACIOCÍNIO DA CASA

O dia zero: fundação antes de funcionalidade

ANTES DE DIGITAR · AS 5 PERGUNTAS POR ESCRITO

1. O que é? (API, site, app, CLI) · **2.** Quem usa? · **3.** Onde roda? · **4.** Que dados guarda? · **5. O que fica DE FORA da primeira versão?** — a mais importante: sem escopo negativo, todo projeto incha e nunca termina.

1	**stack por avaliação, nunca por moda**	mínimo 2 opções comparadas — requisitos, ecossistema, hospedagem, prazo — porquê registrado em **DECISIONS.md**	GATE
2	**estrutura validada pela comunidade**	o scaffold do Laravel, o src-layout do Python, o padrão do Vite — não invente layout de pastas	passo
3	**git init + .gitignore + README**	histórico desde o primeiro arquivo; README com o que é, como rodar, como testar	passo
4	**.env.example desde já**	todo segredo e configuração **fora do código** antes de existir código — o .env real nunca entra no git	passo
5	**primeiro teste passando**	um smoke: o app importa e responde — o alicerce do test-first de todas as tarefas seguintes	GATE
6	**primeiros evals**	3 a 5 casos "rodo este comando → espero esta saída" — o termômetro honesto nasce junto com o projeto	GATE
7	**commit inicial — e só então funcionalidade**	cada item acima custa **minutos hoje e horas depois**; parece lento, é o contrário	MARCO

⌘ Na casa, tudo isso é um comando

O scaffold entrega o projeto com o kit: estrutura canônica, smoke passando, evals iniciais, Dockerfile com healthcheck, .env.example, git iniciado — e **se recusa a rodar sem a decisão de stack registrada**.

MECANISMO `renatao scaffold --tipo api --nome app --decisao "FastAPI vs Express: ..."`

13 SEGURANÇA · DEFESA EM PROFUNDIDADE

Cada ameaça mapeada tem uma camada com nome

Uma ferramenta local com acesso a shell e memória persistente exige **desconfiança por padrão**. Cada risco vira um mecanismo — e o que não dá para proteger de verdade, o sistema **recusa** em vez de fingir.

1	**segredo entrando no acervo**	leak-scan na ingestão: credencial vira **[REDACTED]** antes de tocar o disco	GATE
2	**segredo saindo em público**	segredos vivem fora do git; varredura de vazamento roda no ship-check antes de qualquer entrega	GATE
3	**comando destrutivo no shell**	jail com sandbox de sistema operacional quando existe; sem sandbox real, **recusa rodar por padrão** — não finge proteger	RECUSA
4	**injeção de prompt**	texto vindo de fora (páginas, arquivos, saídas) é **sanitizado** antes de entrar em qualquer prompt	GATE
5	**URL maliciosa ou interna**	toda URL de saída passa por validação anti-SSRF — endereço interno não é alvo legítimo	GATE
6	**API externa em falha**	**circuit breaker**: para de insistir quando a API cai, reabre sozinho quando estabiliza	auto
7	**custo fugindo do controle**	**teto diário de custo** + limite de ações por hora; estourou, chamadas pagas bloqueiam sozinhas	GATE
8	**perda do cérebro**	backup com **verificação de integridade** e rotação — backup que nunca foi restaurado é fé, não backup	GATE

honestidade também é camada · fail-open documentado

Sem a biblioteca de confiabilidade instalada, as ferramentas locais seguem funcionando **e avisam** — disponibilidade escolhida conscientemente para ferramenta local, com o risco por escrito. Segurança que finge é teatro; aqui, cada exceção tem nome, motivo e log.

14 NÚMEROS · MEDIDO, NÃO PROMETIDO

O sistema que cobra evidência presta contas com evidência

720+ testes na própria suíte	**21** evals-canário de comportamento	**1.200+** skills indexadas
8 protocolos com gate	**100%** local — memória e segredos em casa	**0** LLM no roteamento (determinístico)

Testes na suíte — um único dia de trabalho (13/07/2026): +98, tudo test-first

cada bloco de protocolos entrou test-first, sem quebrar nada — os commits deste gráfico existem e são auditáveis

O Selo da Casa

F E D C B A

a nota só sobe com mecanismo — nunca com promessa

renatao seal — projeto real

```
> renatao seal --write
Nota: C (65/100)
## O Que Falta Para Subir
[ ] Definir evals do projeto (+15)
[ ] Evals verdes             (+20)
```

Humano decide. Agente executa. Renato coordena.

par no espírito do XP: test-first, iterações curtas, feedback contínuo — com cabeça de MVP
(o escopo declara o que fica DE FORA antes da primeira linha)

É do **XP (Extreme Programming)** que vêm o test-first/TDD, as iterações curtas com feedback contínuo e a programação em par — aqui, o par é o desenvolvedor e o agente, **coordenando e planejando juntos** antes de qualquer linha. E com cabeça de **MVP**: todo trabalho começa declarando o escopo mínimo e, principalmente, o que fica de fora da primeira versão.

A divisão de papéis é deliberada. O **humano** é dono das decisões — produto, stack, prioridade, o que é crítico. O **agente** executa com a velocidade que só IA tem. E o **Renato** coordena: injeta o contexto certo, serve os protocolos da tarefa e cobra os gates — para que o par nunca pule etapa, nem no dia corrido. Nenhum dos três substitui os outros dois.

POR QUE COMPARTILHAR

“A vida é isso: conhecimento só serve se compartilhado.”

O Renato não nasceu para ser meu. Se ele me ajuda a transformar código de IA em software de verdade, por que não ajudaria todo mundo? Os segredos e as chaves ficam em casa — **o conhecimento, não**. Os protocolos, a arquitetura, o método, as lições (inclusive os erros — foram auditorias adversariais que acharam os melhores bugs): tudo isso é feito para circular. Se uma ideia daqui melhorar o jeito de uma pessoa construir software, o projeto já cumpriu o propósito.

16 ADOÇÃO · O MÉTODO SEM A FERRAMENTA

Leve para o seu time: uma prática por semana

O Renato é a **automação** do método — mas o método funciona à mão, hoje, sem instalar nada. A escada de adoção que a casa recomenda:

1	**aceite como contrato**	antes de cada tarefa: **qual comando vou rodar e que saída espero?** escreva — 2 minutos que matam o "funciona na minha máquina"	SEM. 1
2	**test-first no que importa**	teste antes do código em toda lógica nova; veja **falhar pelo motivo esperado** antes de implementar	SEM. 2
3	**evals do projeto**	5 casos **comando → saída esperada** num arquivo; rode todos antes de qualquer deploy	SEM. 3
4	**revisão adversarial**	3 hipóteses de como pode estar quebrado, **testadas de verdade**, antes do "pronto" — se não achou nada, desconfie da revisão	SEM. 4
5	**segunda versão + selo mensal**	reescrita crítica no código relevante; nota A–F por checklist, uma vez por mês — **a direção importa mais que a foto**	MÊS 2

A regra de ouro da adoção

Uma prática por vez — quatro mudanças simultâneas morrem juntas na primeira semana cheia. E cada prática precisa subir a escada de enforcement do time: o aceite vai para o template do PR, os evals vão para o CI, o selo vira uma reunião mensal de 15 minutos com checklist.

Documento que não bloqueia nada volta a ser conselho — **e conselho se esquece**. A pergunta de cada semana é sempre a mesma: "o que impede, mecanicamente, de pular esta etapa?" Enquanto a resposta for "boa vontade", o trabalho não terminou.

o que é aberto · o que fica em casa

Abertos: os protocolos, a arquitetura, o método e as lições — inclusive os erros. Em casa: segredos, chaves e a memória. **Use, adapte, melhore — e repasse.** Se uma ideia daqui melhorar o jeito do seu time construir software, o projeto já cumpriu o propósito.

17 OBJEÇÕES · PERGUNTAS DIRETAS

O que sempre perguntam, sem marketing

Q1 "Meu agente já é ótimo. Para que isso?"

Ele é ótimo em **escrever** e péssimo em **lembrar e se cobrar**. O Renato não compete — coordena: memória entre sessões, método e gates que não aceitam "confia".

Q2 "Funciona com qual IDE?"

Qualquer um que fale **MCP** — Antigravity, VS Code, Cursor. Identidade, memórias e gates chegam dentro da conversa do agente; a CLI determinística cobre o resto, e humanos também podem usá-la direto.

Q3 "Quanto custa rodar?"

O caminho crítico é local e de **custo zero**: roteamento sem LLM, memória em SQLite, gates como funções. Chamadas pagas são opcionais — sob **teto diário de custo** com bloqueio automático.

Q4 "Meus dados saem da máquina?"

Acervo, índices, segredos e backups: **não**. O texto de uma memória viaja só para gerar o vetor — **depois** do leak-scan — e tudo degrada para busca local pura sem internet.

Q5 "Isso não burocratiza o trabalho?"

Tarefa trivial fica **fora dos gates**. Nas relevantes, o custo é minutos — contra horas de retrabalho. Por construção, **desobedecer dá mais trabalho que obedecer**.

Q6 "Por que não usar LLM no roteamento?"

Reprodutibilidade e auditoria: mesmo pedido, mesmo método, custo zero, latência zero. Auditar é ler uma tabela de pesos — não interrogar um modelo sobre o que ele decidiu ontem.

Q7 "E se o Renato quebrar?"

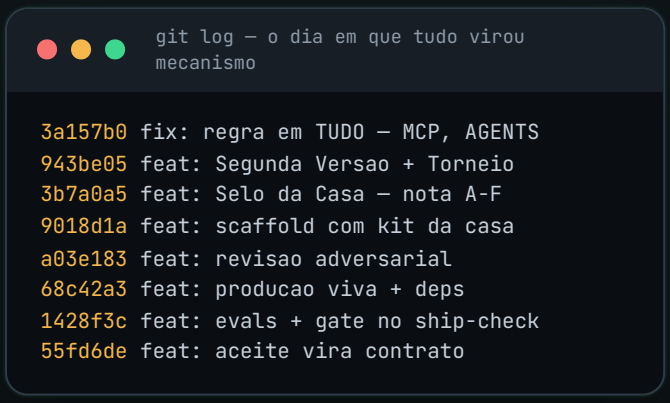
O agente continua funcionando — o Renato coordena, não executa. Os artefatos ficam no projeto, legíveis **sem a ferramenta**; o cérebro tem backup verificado de verdade.

O Renato veio de uma necessidade

Trabalho solo para várias empresas e projetos ao mesmo tempo, e precisava urgente de alguém para me ajudar. Construí o Renato porque queria **testar, conhecer, entender e extrair o melhor que a IA podia dar** para o meu trabalho de verdade — não em demo, no dia a dia.

Assim nasceu o **Espetacular Renato**: hoje não só uma IA, mas um **parceiro de trabalho**.

— Erick



O que este dossiê afirma é verificável

- Os números têm commit por trás — o gráfico da página 14 é o `git log` ao lado.
- As histórias "aconteceu de verdade" têm artefato e teste registrados.
- O método que produziu este documento é o método que ele descreve: este PDF passou pela Regra da Segunda Versão — três vezes.

Construído em português, no Brasil, com método brasileiro:
avaliação em vez de dogma, evidência em vez de juramento — e a porta aberta.