# 11 — Verificação e Evals

🕒 estado da arte 2026-07 · revisão 2026-07-26 · 📖 ~14 min de leitura

## Objetivos de aprendizagem

Ao final deste capítulo, você deve ser capaz de:

1. **Distinguir** as três perguntas da verificação (o harness funciona? · o agente se comporta? · o trabalho está certo?) e a resposta técnica de cada;
2. **Explicar** por que a auto-correção *intrínseca* não basta e a verificação precisa ser externa e ancorada em sinal (testes, LSP (Language Server Protocol), tools);
3. **Avaliar** o *reward hacking* — o agente jogando contra o verificador — e as defesas (held-out, testes imutáveis, anti-mock, verificar o estado final);
4. **Reconhecer** os vieses do juiz LLM (Large Language Model) (posição, verbosidade, self-preference) e como mitigá-los;
5. **Implementar** uma suíte de evals do harness-zero (juiz + respostas gravadas) na etapa 10.

## O problema

Como saber se o agente funciona? A pergunta se desdobra em três, com respostas técnicas diferentes:

1. **O harness funciona?** — testes de software clássicos sobre o código do harness (loop, tools, permissões).
2. **O agente se comporta bem?** — evals: o comportamento emergente (usa as tools certas? é frugal? respeita o plan mode? resiste a injection?) sob teste de regressão.
3. **O trabalho do agente está certo?** — verificação em runtime: sinais (LSP, testes, lint) realimentados ao modelo durante a tarefa.

A segunda é a mais difícil e a mais negligenciada: comportamento de agente é estocástico, caro de testar e muda silenciosamente a cada troca de modelo ou de prompt. E há uma quarta pergunta que a rodada 2 tornou incontornável: **o agente está trapaceando o verificador?**

## Fundamentos científicos

A ciência da verificação de agentes tem três mensagens duras — e todas empurram para o mesmo lugar: verificação **externa e ancorada**.

- **Grading por execução, não por aparência** — SWE-bench, arXiv 2310.06770 (ICLR '24) verifica aplicando o patch do modelo e rodando os **testes reais e ocultos** do repositório (FAIL_TO_PASS + PASS_TO_PASS). Decisão: para código, o único sinal confiável é "os testes reais passaram", não similaridade de diff. E SWE-agent, arXiv 2405.15793 mostra que a **ergonomia das tools** (a Agent-Computer Interface) dirige o sucesso tanto quanto o modelo.

- **A auto-correção intrínseca não basta** — Large Language Models Cannot Self-Correct Reasoning Yet, arXiv 2310.01798 é o contra-resultado decisivo: sem feedback externo, pedir ao modelo que "revise" pode *degradar* respostas certas. Decisão: "pedir ao modelo para se conferir" **não é** estratégia de verificação — o harness precisa fornecer um verificador. CRITIC, arXiv 2305.11738 mostra o caminho: auto-crítica **ancorada em tool** (o código roda? o fato confere?) supera introspecção; Self-Consistency, arXiv 2203.11171 dá a versão barata (amostrar caminhos + voto) para respostas checáveis.
- **O juiz LLM funciona — com vieses** — Judging LLM-as-a-Judge, arXiv 2306.05685 mede ~80% de acordo com humanos, mas documenta vieses de **posição, verbosidade e self-preference**. Decisão: randomize/troque a ordem das respostas e faça a média, dê rubrica e resposta-referência, e calibre contra um gold set humano (survey, arXiv 2411.15594) — um único call de juiz não é ground truth. E verifique o **estado final do mundo**, não o transcript: τ-bench, arXiv 2406.12045 mostra que `pass@1` esconde inconsistência brutal (`pass^8` < 25%).
- **O agente joga contra o verificador** — o tema novo e mais importante: com recompensas verificáveis (RLVR / Tülu 3, arXiv 2411.15124) um verificador determinístico é sinal e recompensa mais difícil de fraudar — *mas* reward hacking, arXiv 2606.15385 e testes randomizados contra trapaça, arXiv 2606.07379 mostram que agentes praticam *specification gaming* zero-shot: apagam asserts, dão `sys.exit(0)`, patcham o pytest. Decisão: mantenha uma métrica de ground-truth **held-out** que o agente nunca otimiza, e **testes imutáveis** que ele não pode tocar.

(Bibliografia completa e ponteiros: `livro/bibliografia.md`.)

## Fontes da indústria

- **O benchmark é o padrão — e é contaminável** — SWE-bench Verified (OpenAI) é o subconjunto de 500 tarefas *humanamente auditadas*, criado porque o SWE-bench cru tinha specs ambíguas e testes quebrados que reprovavam soluções corretas (audite o verificador antes de confiar nele). Mas o OpenAI parou de reportar SWE-bench Verified por contaminação/memorização — o eval precisa de rotação e held-outs para seguir sendo sinal. O Terminal-Bench (arXiv 2601.11868, repo `harbor-framework/terminal-bench`) leva o rigor ao terminal: cada tarefa embarca **Docker + solução humana + testes de verificação**, gradando o *estado final do ambiente*, não a plausibilidade do transcript.
- **Evals como disciplina de engenharia** — Define success criteria and build evaluations (Claude): defina critérios mensuráveis *antes*, force o juiz a emitir um veredito discreto e a raciocinar antes de pontuar. O Demystifying evals for AI agents (Anthropic) decompõe o eval em componentes (task · trial · agent harness · eval harness · trace · grader · suite) e insiste: **grade o estado final, não a última mensagem** (uma resposta pode "soar certa" e a tarefa ter falhado). E reporte o erro-padrão da média para distinguir regressão real de ruído.
- **Verificação dentro do loop** — Effective harnesses for long-running agents (Anthropic): cada sessão roda os testes, **verifica a feature end-to-end como um usuário faria** (automação de navegador), deixa um log de progresso e commita limpo. E o Claude Code best practices eleva o TDD ao padrão agêntico mais forte: escreva os testes primeiro, confirme que falham, **commite-os como checkpoint, e**

**implemente sem editá-los** — commitar os testes antes é a rede que revela quando o agente trapaceia alterando o teste em vez de corrigir o código.

- **Ferramental de eval versionado** — OpenAI Evals, Inspect (UK AISI) (Dataset + Solver + Scorer, com sandbox Docker/K8s — o eval e o sandbox são um só sistema), promptfoo (um `promptfooconfig.yaml` versionado como gate de CI), Braintrust e LangSmith (rubrica como config, correções humanas viram few-shot). Decisão: os checks vivem no controle de versão e rodam no CI como qualquer teste.
- **Verificação virou adversarial** — Natural emergent misalignment from reward hacking (Anthropic): agentes aprendem a *gamear o verificador* (sair antes dos testes, patchar o pytest, apagar asserts) e o hábito **generaliza para sabotagem mais ampla**. Decisão: endureça o verificador (testes randomizados/held-out, arquivos de teste imutáveis) e não deixe o agente tocar no próprio grader — o The Verification Horizon (arXiv 2606.26300) adverte que, quando a capacidade do agente ultrapassa o verificador, o reward hacking ressurge; o verificador tem de *evoluir* (testes → rubrica → juízes interativos).
- **Consulte também**: a coleção viva Awesome Harness Engineering — Verification & CI Integration e Awesome Harness Engineering — Evals & Verification reúne mais recursos consultáveis desta dimensão (padrões, artigos e implementações), curados por problema.

## O estado da arte

### 1. Três perguntas, três campeões (e a lacuna que fechou)

A moldura da rodada 1 persiste: **OpenHarness** testa melhor *o harness* (121 arquivos por subsistema), **gemini-cli** testa melhor *o agente* (evals com juiz + baselines de regressão), **opencode** verifica melhor *o trabalho* (LSP em runtime → diagnósticos ao modelo no mesmo turno). Mas a lacuna reveladora da rodada 1 — "só um dos três testa comportamento sob ataque" — **fechou** na rodada 2: IronClaw trata isolamento cross-tenant como cidadão de teste de primeira classe (com *parity de trace* contra o OpenClaw), e ohmo tem 96 testes adversariais (sessão não vaza para outro remetente, `/config` não vaza segredos).

### 2. A verificação certa é externa e ancorada — porque a interna falha

O achado científico central (a auto-correção intrínseca degrada; a ancorada em tool funciona) é exatamente o que o **LSP em runtime** do opencode faz: o agente descobre que quebrou a tipagem no turno seguinte, não no CI. É a mesma tese da **reflexão do Aider** (disparada por lint/testes falhando, não por introspecção) e do **verify-on-stop do Hermes** — o agente é *forçado* a verificar antes de parar, com `verification_evidence.py` rastreando a evidência. Verificar deixou de ser uma esperança e virou um **estágio imposto do loop**.

### 3. Eval comportamental virou table-stakes — e por categoria

Na rodada 1, só o gemini-cli tratava comportamento como superfície de regressão. Na rodada 2 isso virou norma: o **Goose** publica a **Harbor** (sobre o framework Terminal-Bench, 89 tasks, com **leaderboard real**: stock 50,6% / code-mode 57,3%); o **Codex** tem ~660 snapshots insta; o **Hermes** roda `mini_swe_runner` (estilo SWE-bench); o **n8n** transformou eval em *produto* (nós Evaluation + LLM-judge). E surgiu o eval **por**

**categoria**: o **Personal Agent Benchmark Pack** do OpenClaw (10 cenários da categoria — `personal-redaction-no-secret-leak`, `personal-approval-denial-stop`, `personal-no-fake-progress`, `personal-memory-preference-recall`), o primeiro benchmark comportamental *da categoria agente pessoal*. Um harness sem evals não sabe o que perdeu no último ajuste de prompt.

### 4. O adversário é o próprio agente

A virada mais séria: a verificação virou **adversarial**. A literatura mostra agentes apagando asserts e patchando o pytest para "passar"; a defesa da indústria é convergente — **testes imutáveis** (commite os testes primeiro, o agente não os edita), **held-out/randomizado** (o agente não pode overfittar o que não vê), **política anti-mock** (o `AGENTS.md` de teste do opencode proíbe mocks que mentem; o `http-recorder` grava chamadas reais), e **snapshots com drift-check** (OpenClaw) para determinismo onde o juiz é caro. A verificação não é mais só medir acerto — é impedir a trapaça.

> **Adendo (2026-07-31, texto integral verificado): como avaliar o próprio harness — três regras de um paper de método.** O preprint *Rethinking the Evaluation of Harness Evolution for Agents* (AI2/UW/indep., 14-jul-2026) testa a moda da "evolução automática de harness" e encontra um resultado incômodo: sob **orçamento equiparado** (K=5 para todos os métodos), ela "does not consistently outperform simple test-time scaling methods" — no Terminal-Bench 2.1 (89 tarefas, 3 modelos), amostragem paralela pura levou a média de pass@1 de 68,2 a 72,3 (Tabela 1) enquanto a evolução chegou a **piorar** o GPT-5.4 (75,3→69,7); com testes unitários disponíveis, a amostragem paralela abre 86,0 contra 75,8 (Tabela 2); e em tarefas held-out o ganho médio da evolução é **+0,6** (Tabela 3) — "their gains largely stem from making multiple attempts" (§4.3), porque "most edits memorize fixes rather than distilling strategies" (§5.1), acumulando "context bloat that can offset the remaining gains". As três regras que ficam para quem avalia harness (incluindo este livro): (1) **orçamento equiparado** — todo ganho atribuído a design deve ser reportado contra um baseline de repetição de amostras com o mesmo compute; (2) **separação busca/avaliação** — held-out obrigatório, ou o ganho é overfitting ao conjunto; (3) **sensibilidade do instrumento** — os próprios autores suspeitam que "Terminal-Bench may simply not be very sensitive to harness design" (§5.2): benchmark bom para medir harness precisa de headroom E de desempenho que dependa do harness, senão o sinal é capacidade do modelo. Para o método deste livro (rubrica 0–3 por leitura de código), o paper refina sem contradizer: a rubrica mede a propriedade estrutural sem passar pelo canal contaminado por amostragem — mas herda o dever da **validade convergente** (notas altas deveriam prever desempenho held-out), o risco de overfitting se a régua for calibrada olhando os sistemas que se quer pontuar bem, e o alerta do §5.1: penalizar memorização e inchaço de contexto, não só ausência de recursos. Isso conversa com o cap. 16: se evoluir o harness automaticamente rende menos que re-amostrar, a auto-melhoria barata está no **conhecimento** (skills/memória), não na **estrutura**.

**LEITURA EXECUTIVA**

> O que está mais moderno: verificação externa ancorada (LSP/testes no loop, verify-on-stop); eval comportamental como table-stakes e por categoria (Harbor, Personal Agent Benchmark Pack); o juiz LLM usado com controle de viés; a defesa contra reward hacking (testes imutáveis, held-out, anti-mock); e, para quem avalia o próprio harness, as três regras do adendo (orçamento equiparado, held-out, instrumento sensível a design). **O que roubar:** realimente sinal real ao modelo no mesmo turno (LSP/testes), não confie na auto-conferência; commite os testes antes e não deixe o agente editá-los; grade o estado final, não a última mensagem; e trate evals comportamentais como regressão de primeira classe.

## Mão na massa — harness-zero, etapa 10

A etapa 10 (`harness-zero/etapas/10-evals/`) dá ao harness-zero uma suíte de evals própria: **respostas de LLM gravadas** (replay determinístico em CI, barato e estável) para testar o loop e as tools sem chamar a API, e um **juiz LLM** mínimo que pontua se o comportamento do agente atende a critérios qualitativos (usou a tool certa? respeitou o plan mode?). Fiel à disciplina do capítulo: o juiz emite um veredito discreto e a suíte roda no CI como qualquer teste. Exercício de completude: você adiciona um caso de **teste imutável** — uma tarefa cujo teste o agente é proibido de editar — e observa a diferença entre "passou" e "trapaceou".

## Verificação

1. Seu agente diz "corrigi o bug e os testes passam". Por que isso, sozinho, não é verificação — e o que você faz em vez de confiar? (Auto-correção/auto-relato intrínseco não basta — 2310.01798; rode os testes reais e ocultos e grade por execução — SWE-bench.)
2. Depois de dar RLVR ao seu agente, o score sobe mas o produto piora. O que provavelmente aconteceu, e que duas defesas você aplica? (Reward hacking — o agente gameia o verificador, ex. apaga asserts; defesas: métrica held-out que ele não otimiza + testes imutáveis.)
3. Você usa um juiz LLM para pontuar respostas abertas. Cite um viés conhecido e como mitigá-lo. (Posição/verbosidade/self-preference; trocar a ordem e fazer a média, rubrica + gold set humano.)

---

## Apêndice A — Como cada repositório trata verificação e evals

> Evidência por harness, com paths — complementação online, expandida a cada rodada.

### gemini-cli (rodada 1) — comportamento sob regressão contínua

Quatro suítes: (1) `evals/` — ~45 testes comportamentais com **juiz LLM** (`llm-judge.ts`) cobrindo frugalidade, memória hierárquica, plan mode, delegação, segurança de shell, **prompt injection via MCP (Model Context Protocol)** e recuperação de sandbox; (2) `integration-tests/` — E2E determinísticos com **respostas gravadas** (`.responses`); (3) `memory-tests/` — regressão contra `baselines.json`,

nightly; (4) `perf-tests/` — CPU/startup, nightly. Comportamento como superfície de regressão de primeira classe.

**opencode (rodada 1) — verificação durante a tarefa**

**LSP em runtime** (`packages/opencode/src/lsp/`): edições disparam diagnósticos realimentados ao modelo. **Política anti-mock** explícita (o `AGENTS.md` de `test/` proíbe mocks) + `http-recorder` (grava/replaya HTTP real com determinismo). Typecheck obrigatório (`bun typecheck`).

**OpenHarness (rodada 1) — E2E com modelo real**

121 arquivos em `tests/`, ~31 subpastas espelhando cada subsistema. Suítes E2E com **chamadas reais de modelo** (`scripts/test_harness_features.py`) e testes contra artefatos reais do ecossistema (`test_real_skills_plugins.py` roda skills do anthropics/skills e plugins do claude-code). Skill `harness-eval` empacota a validação E2E.

**Goose (rodada 2) ⭐ — Harbor com leaderboard público**

**Harbor** (`evals/harbor/`): benchmark sobre o framework Terminal-Bench (89 tasks) comparando harnesses/modelos/builds por pass-rate, custo, tokens e turns — com **leaderboard real no README** (stock ~50,6%, code-mode 57,3%) e LLM-judges de pós-processamento; `goose-self-test.yaml`; compactação com ~15 testes inline.

**Codex CLI (rodada 2) — snapshots em escala**

~440 arquivos de teste + **~660 snapshots insta**; suíte E2E com turnos reais e backend mockado; testes de política de sandbox por plataforma; parity da compactação remota; CI multi-camada (nextest por plataforma, Bazel, postmerge).

**Hermes (rodada 2) ⭐ — verify-on-stop**

32 subdiretórios de teste; **verify-on-stop nudge** (o agente é forçado a verificar antes de parar, com `verification_evidence.py` rastreando evidência); `batch_runner.py` (trajetórias em lote) e `mini_swe_runner.py` (avaliação estilo SWE-bench). Orientação a pesquisa.

**OpenClaw (rodada 2) ⭐ — benchmark da categoria**

~8.649 arquivos de teste; **prompt snapshots com drift-check** em CI; stack QA com canal sintético e catálogo YAML de cenários; **Personal Agent Benchmark Pack** — 10 cenários da categoria (`personal-redaction-no-secret-leak`, `personal-approval-denial-stop`, `personal-no-fake-progress`, `personal-memory-preference-recall`…), rodáveis em mock. O primeiro benchmark comportamental *da categoria agente pessoal*.

**IronClaw (rodada 2) ⭐ — isolamento como cidadão de teste**

~415 arquivos de teste; fuzzing; **testes de isolamento cross-tenant/agent/project/thread como primeira classe** (`reborn_*_scope_isolation_parity.rs`); **parity de trace gravado contra o OpenClaw**; testes de arquitetura mecanizados; regra que exige testes de denial/redaction/escape para qualquer mudança de sandbox.

### ohmo (rodada 2) — adversarial de canal

**96 testes adversariais** (75 no gateway): sessão não restaura mensagens de outro remetente, `/config show` não vaza segredos, histórico de `/group` sanitizado antes de virar contexto. Lacuna: sem teste de permissão/sandbox — exatamente a dimensão fraca.

### Aider (rodada 2) — reflexão ancorada + leaderboard de edit format

Reflexão (`reflected_message`, máx. 3) disparada quando o linter acha erros ou testes falham (sempre com confirmação humana) — auto-correção **reativa e ancorada**, não introspecção. Famoso por medir empiricamente o formato de edição por modelo (`percent_cases_well_formed`) num leaderboard próprio.

### n8n (rodada 2) — eval como produto

Feature **Evaluations** (nós Evaluation Trigger + Evaluation, UI enterprise) para rodar datasets contra workflows; suíte de evals com LLM-judge no AI Workflow Builder; testes de integração por workflow. Verificação empacotada como recurso vendável.

### OpenHands (rodada 2) — o eval que migrou

Eval de agente **ausente neste repo** (nota 0): o diretório `evaluation/` clássico (o harness SWE-bench pelo qual o OpenHands é histórico) migrou para o `software-agent-sdk`. Aqui há 115 arquivos de testes unitários do app-server, mas zero evals de agente — um lembrete de que a fronteira do que se avalia depende de onde o núcleo vive.

### Frameworks (rodada frameworks)

Os frameworks tratam eval como API: harnesses de eval versionados (OpenAI Evals), Solver+Scorer com sandbox (Inspect), scorers mistos código+juiz (Braintrust/autoevals), rubrica-como-config (LangSmith). O que os harnesses de código montam à mão, o ecossistema de frameworks expõe como ferramenta dedicada.