18 — Harness

18 — Harness

> Disciplina de sessão: guias versionados, estado **por branch**, tudo versionado exceto preferências pessoais do dev. Ao criar em projeto novo, invoque a skill `anthropic-skills:curio-harness-bootstrap` — ela é a fonte canônica dos templates.
>
> **Reestruturado em 2026-08-25** (branch-primeiro, sem `global_feature_list.json`) — ver nota de migração ao final.

O problema que isso resolve

Em projeto grande tocado por várias pessoas com Claude Code, três coisas quebram:

1. **Contexto se perde entre sessões.** Cada sessão redescobre o que a anterior já sabia.
2. **Dois devs pisam no mesmo trabalho.** Ninguém vê o que o outro está fazendo.
3. **"Está pronto" vira afirmação sem evidência.** Código escrito é confundido com código verificado.

A solução não é framework — é convenção de arquivos. Guias versionados (o que o Claude precisa saber antes de mexer), estado de sessão **por branch** (sem conflito entre branches), e um `init.sh` que verifica baseline antes e depois.

Estrutura

Fica na **raiz do repositório**, ao lado de `CLAUDE.md` e `init.sh`. Não dentro de `src/` nem de `docs/`.

```
projeto/
```

```
├── CLAUDE.md                    ponto de entrada (carregado automaticamente)├──
init.sh                       verificação de baseline — ver 19
└── harness/
    ├── guides/                  referência técnica — VERSIONADO
    ├── _examples/    │   ├── state/                templates: feature_list.example.json,
progress.example.md,
    │   │                        session-handoff.example.md    │   ├── handoffs/
template: handoff.example.md
    │   └── user_preferences.example.md    ├── <branch>/                tudo que a branch
produziu — VERSIONADO    │   ├── state/                feature_list.json, progress.md, session-
handoff.md
    │   ├── plans/               planos de implementação (task-by-task)    │   ├──
specs/                specs de design
    │   └── handoffs/            handoffs de sessão    └── user_preferences.md
preferências do dev — ÚNICO ARQUIVO LOCAL (gitignored)
```

`<branch>` é o nome literal da branch (`git rev-parse --abbrev-ref HEAD`). Branch com `/` no nome vira subpasta naturalmente — `feature/x` → `harness/feature/x/`.

A distinção que importa

Versionado	**Local (gitignored)**
`guides/`, `_examples/`, tudo dentro de `<branch>/`	`user_preferences.md`
Todo o trabalho de sessão, por branch	**Preferência pessoal do dev, não é da branch**

Trocar de branch agora troca de estado automaticamente — `git checkout` já isola o `harness/<branch>/` de qualquer outra. Não existe mais "estado local do dev": o estado é da branch, e é versionado com ela.

`.gitignore`

```
# Harness (Claude Code) — so preferencia pessoal e' local
harness/user_preferences.md
```

Todo o resto — `guides/`, `_examples/`, e tudo dentro de `harness/<branch>/` — **é versionado**.

`guides/` — quais escrever

Um projeto fullstack usa guias de backend além dos de frontend. **Um projeto só de frontend usa um subconjunto.**

Escrever

Guia	Conteúdo
`commands.md`	Build, run, ambientes, variáveis. Comandos **reais**, confirmados
`architecture.md`	Camadas do front, fluxo até o backend, autenticação, roteamento
`curio-framework-guide.md`	Sessão, caso de uso, `useCurioMutation`, React Query, tratamento de erro
`dev-environment.md`	Node/fnm, `.env`, acesso aos servidores por ambiente
`husky-git-hooks.md`	O que cada hook faz, como configurar, armadilhas de `sh -e` e `source`

Não escrever (são de backend)

`curio-backend-syntax.md`, `curio-spring-boot-guide.md`, `staruml-tooling.md`, `test-data-scripts.md`. Se o projeto novo tiver backend próprio, aí sim — mas isso está fora do escopo deste guia.

Condicionais

- `gitlab-access.md` — só se scripts do projeto usarem a API do GitLab (`GITLAB_PRIVATE_TOKEN`). **Nunca commitar o valor do token**, só a variável de ambiente.
- Glossário de domínio (ex.: um `arquitetura-dominio-<assunto>.md`) — só se o domínio for genuinamente intrincado. É investigação cara; **pergunte antes de criar**.

Regra ao escrever um guia

Investigue o código real. **Não escreva de memória do que "Curio costuma ser"** — pode ser outra versão do framework, outra convenção, outro layout. Se um comando não foi confirmado (rodado ou lido do `package.json`), não o documente.

`<branch>/state/` — estado da branch

`feature_list.json`

```
{
  "project": "{{NOME_DO_PROJETO}}",
  "project_type": "frontend",
  "last_updated": "YYYY-MM-DD",
  "rules": {
    "passing_requires_evidence": true,
    "do_not_skip_verification": true
  },
  "status_legend": {
    "not_started": "Work has not begun.",    "in_progress": "The feature is the current active task.",
    "blocked": "Work cannot continue until a documented blocker is resolved.",    "passing": "Required verification has passed and evidence is recorded."
  },
  "features": [
    {
      "id": "exemplo-001",
      "type": "ui",
      "area": "nome-da-area",
      "title": "Título curto da feature",      "behavior": "Descrição do comportamento esperado/pendência.",      "verification": "Como confirmar que está correto (comando, tela, checklist).",
```

```
      "status": "not_started",      "evidence": "Preenchido só quando houver evidência real de
verificação."
    }
  ],
  "completed": []
}
```

Num projeto só de frontend, `type` é predominantemente `"ui"` — a verificação é abrir a tela e conferir o fluxo, não rodar teste.

O array `features` guarda o que **ainda não** é `passing`. Ao concluir, a entrada sai de `features` e entra em `completed` — histórico da branch, versionado, sem cópia para nenhum arquivo global.

progress.md

```
# Session Progress Log (versionado — por branch)
## Current State
**Last Updated:** YYYY-MM-DD
**Active Feature:** o que você está trabalhando agora
## Sessão YYYY-MM-DD (parte N) — título curto
- O que foi feito, decisões tomadas, bugs encontrados.- Comandos de verificação rodados e
resultado.
- Pendências para a próxima sessão.
## Status
### What's Done
- [ ] ...
### What's In Progress
- [ ] ...
### What's Next
1. ...
## Blockers / Risks
- [ ] ...
```

session-handoff.md

```
```

```
# Session Handoff (versionado — por branch)
## Current Objective
- Goal: ...
- Active feature (from harness/<branch>/state/feature_list.json): ...
- Branch / commit: ...
## Completed This Session
- [x] ...
## Verification Evidence
| Check | Command | Result | Notes |
| ----- | ------- | ------ | ----- |
| ...   | ...     | ...    | ...   |
## Files Changed
- ...
## Decisions Made
- ...
## Blockers / Risks
- ...
## Next Session Startup
1. Ler `CLAUDE.md` (seção "Harness")2. Resolver a branch atual (`git rev-parse --abbrev-ref HEAD`) e ler `harness/user_preferences.md`   (se existir) e `harness/<branch>/state/progress.md`
3. Rodar `./init.sh` (FAST) antes de editar
## Recommended Next Step
- ...
```

Regra inegociável

`feature_list.json` e `progress.md` nunca nascem pré-preenchidos. Só os `_examples/` são criados no bootstrap. `harness/<branch>/state/` é criado (scaffolding a partir dos `_examples/`) na primeira sessão daquela branch — mas o conteúdo é preenchido por quem trabalha, não pelo Claude "para ajudar".

Se `harness/<branch>/state/` não existir ainda, o Claude deve **criar a pasta a partir dos `_examples/`** (mecânico) e **recomendar preencher** — não inventar `feature_list.json` / `progress.md` com valores supostos.

`plans/` e `specs/`

- **`<branch>/plans/*.md`** — planos de implementação task-by-task, com checkboxes. Nome: `YYYY-MM-DD-nome-da-feature.md`.
- **`<branch>/specs/*.md`** — specs de design para decisões pontuais. Nome: `YYYY-MM-DD-nome-do-assunto-design.md`.

Quando existir plano ou spec da feature ativa, ele **tem precedência** sobre o `feature_list.json` genérico para o detalhe fino. Atualize os dois ao final da sessão.

Regra de proveniência: um plano/spec fica na branch onde foi criado, mesmo que o trabalho continue depois em outra branch. Não é para realocar quando isso acontecer — é só registro de onde nasceu.

> **Se o projeto já tem planos/specs soltos em outro lugar** (ex.: `docs/superpowers/plans/` e `docs/superpowers/specs/`, de um uso anterior da skill `superpowers`), esse conteúdo passa a viver em `harness/<branch>/plans/` e `harness/<branch>/specs/`.
>
> **Nota de compatibilidade:** a skill `superpowers` pode assumir o caminho antigo `docs/superpowers/`. Se ela não encontrar os arquivos no novo local, ajuste a skill — não volte a mover os arquivos.

`handoffs/` e `user_preferences.md`

- `<branch>/handoffs/*.md` — handoff de uma feature específica, **versionado**. Complementa o `session-handoff.md`, que cobre a sessão mais recente de qualquer feature naquela branch.
- `user_preferences.md` — preferências pessoais do dev (modo padrão do `init.sh`, verbosidade, ambiente local, convenções de commit). **Único arquivo local**, criado a partir de `_examples/user_preferences.example.md`. Não é da branch — é da pessoa, atravessa todas as branches.

Bootstrap em projeto novo

Invoque a skill `anthropic-skills:curio-harness-bootstrap`. Ela traz os templates já validados, o `init.sh.template` e a seção de `CLAUDE.md`. Não reconstrua a estrutura lendo outro projeto na mão — use os templates da skill como fonte, para não divergir silenciosamente.

Passos:

1. Criar `harness/guides/` e `harness/_examples/` (nomes exatos acima — não invente variação).
2. Copiar os templates para dentro de `_examples/` (`state/`, `handoffs/`, `user_preferences.example.md`).
3. Adicionar `harness/user_preferences.md` ao `.gitignore` — só ele.
4. Escrever os guias **a partir do código real** do projeto.
5. Gerar o `init.sh` com os comandos reais — ver 19.
6. Inserir a seção de harness no `CLAUDE.md` — ver 16, incluindo o passo de resolver a branch atual e criar `harness/<branch>/state/` a partir dos `_examples/`.
7. **Rodar o `init.sh`** (FAST) e confirmar que passa.

Não existe mais passo de criar `global_feature_list.json` — foi eliminado (ver nota abaixo).

O que não fazer

- **Não copie os guias de outro projeto** achando que "é o mesmo framework". Pode ser outra versão do Curio, outra linguagem, outras convenções. Sempre reescreva a partir da investigação do projeto alvo.
- **Não crie `<branch>/state/feature_list.json` ou `progress.md` preenchidos** "para ajudar" — só a pasta, a partir dos `_examples/`; o conteúdo é de quem trabalha.
- **Não pule a execução do `init.sh`.** Script quebrado passa despercebido até a primeira sessão real, quando já é tarde.
- **Não recrie `global_feature_list.json`.** Foi removido deliberadamente — ver nota de migração abaixo.

Reestruturação 2026-08-25: por

que branch-primeiro

Problema encontrado: `harness/state/feature_list.json` (+ `progress.md`, `session-handoff.md`) era um único arquivo, compartilhado entre todas as branches — apesar do `.gitignore` marcar `state/*` como local, os arquivos já estavam commitados de antes. Trocar de branch sobrescrevia o estado de outra branch. Essa foi a causa raiz de um bug real observado num projeto que segue este padrão.

Decisão: layout branch-primeiro. Cada branch ganha `harness/<branch>/` com `state/`, `plans/`, `specs/` e `handoffs/` próprios, tudo versionado — sem gitignore para nada disso.

O que foi removido: `global_feature_list.json` (histórico compartilhado entre devs/branches de features `passing`). Decisão explícita: aceitar perder a visibilidade cross-branch em troca de simplicidade — cada branch agora é autocontida. Não recrie esse arquivo achando que é uma omissão.

Migração de histórico pré-existente: se um projeto já tem harness no formato antigo (um `state/` único, `global_feature_list.json`) e vai migrar para este layout, os `plans/` / `specs/` antigos devem ser **reatribuídos** à branch onde nasceram — via `git log --merges` (merge commits guardam a branch de origem mesmo depois dela ser deletada: para cada arquivo, ache o commit de criação e caminhe pelos merges até achar aquele em que o commit é ancestral só do segundo parent, não do primeiro). Não deixe esse histórico solto numa pasta `_legacy` — reatribua de fato.

Revision #3
Created Thu, Aug 20, 2026 5:09 PM by Geraldo Barbosa
Updated Tue, Aug 25, 2026 4:55 PM by Geraldo Barbosa