Blueprint: Workflow Padrão para Coding Agents — Context Engineering + RAG + Reranker

> Objetivo: aumentar **confiabilidade**, **reduzir retrabalho** por alucinações/perda de contexto e **padronizar** o uso de agentes (GPT-5 p/ planejamento/revisão/ultra-thinking e Claude Sonnet 4.5 p/ implementação).

---

1) Arquitetura de Alto Nível

Papéis (multi-modelo):

- **Planner/Architect (GPT-5):** decomposição, plano tático, contrato de contexto, revisão e gate de qualidade.
- **Implementer (Claude Sonnet 4.5):** gerar/editar código conforme plano e contrato.
- **Critic (GPT-5):** verificação estática/semântica e gate de merge.

Serviços:

- **Orquestrador de Agentes (LangGraph ou equivalente):** estado persistente, retries, rotas condicionais.
- **Context Service (RAG):** indexação do código/projetos, busca híbrida (BM25 + vetor), reranker cross-encoder, cota por tipo de contexto.
- **MCP Connectors (opcional):** repositórios, CI, wikis, issue tracker.
- **Quality Gates:** testes, linters, build, checagens anti-alucinação, políticas.

Fluxo resumido:

```
Issue -> [Planner] Plano + Contrato de Contexto
      -> [Retriever] Busca híbrida (k=50) -> [Reranker] top N (10–15)
      -> [Implementer] Patch/diff + testes gerados/atual.
      -> [Critic] Code review + self-checks estruturados.
      -> PR com descrição, riscos, migração e casos de teste.
      -> CI + Human-in-the-loop.
```

---

2) Contrato de Contexto (Context Contract)

> **Regra de ouro:** todo trabalho do agente deve iniciar com um contrato de contexto **estruturado** e versionado.

Template (preencha automaticamente no Planner):

```
contract_version: 1
issue: <link/ID>
objective: <1-2 frases claras>
acceptance_criteria:
  - <critério 1>
  - <critério 2>
out_of_scope:
  - <fora do escopo 1>
constraints:
  coding_standards: <ex.: PEP8, ESLint config>
  security: <ex.: evitar libs X; no secrets>
  performance_budget: <ex.: p95 < 50ms>
context_pack:
  repo_fingerprint: <commit SHA / tag>
  code_conventions: <resumo>
  key_symbols: [ {name, kind, file, sig, lines} ]
  related_files: [ "path/fileA:10-80", "path/fileB" ]
  knowledge_refs: [ "ADR-012", "design/README.md" ]
retrieval_plan:
  query_seeds: ["<queries>"]
  k_candidates: 50
  rerank_top_n: 12
  must_include: ["file/critical.ts"]
outputs_expected:
  - type: PLAN_JSON
  - type: DIFF_UNIFIED
  - type: TESTS
  - type: PR_SUMMARY
uncertainty_policy:
  allow_idk: true
  require_citations: true
risk_checklist:
  - migrations?
  - backwards_compat?
  - security_footguns?
```

Prompt-base (Planner -> Implementer):

```
Você é um engenheiro que **NÃO** inventa APIs ou símbolos.
Use **apenas** símbolos presentes no Context Pack.
Se faltar contexto, pare e peça `MISSING_CONTEXT` com lista específica.
Responda **estritamente** no formato solicitado.
```

---

3) RAG de Código (Indexação, Busca, Re-Rank)

Ingestão & Indexação

- Chunking hierárquico (repo → diretório → arquivo → função/método), com metadados: linguagem, AST, assinatura, imports, grafos de referência, testes relacionados.
- Enriquecer chunks com *symbol graph* (def-use, call graph) e *repo fingerprint* (SHA + branch).

Busca Híbrida

- BM25/keyword para nomes exatos + vetor para semântica.
- Consultas expandidas com sinônimos/aliases a partir de símbolos.

Reranker

- Cross-encoder para ordenar os candidatos (top 10–15) com foco em **linhas** e **assinaturas** que casam com o pedido.

Políticas

- Cada contexto entregue ao modelo **deve citar e ancorar por SHA**: `file\:path\:ln-start..ln-end@\<commit-SHA>`.

- **Limiar de confiança do reranker: 0.40 (default)**. Se `< 0.40`, retornar **`MISSING_CONTEXT`** junto com a lista de arquivos/assinaturas necessários.

4) Saídas Estruturadas (Schemas)

Plano (Planner → JSON)

```
{
  "steps": [
    {"id": "S1", "title": "Analisar símbolos", "owner": "Planner", "criteria":
["lista de símbolos existente"], "estimated_tokens": 800},
    {"id": "S2", "title": "Gerar patch", "owner": "Implementer", "criteria":
["compila", "tests passing"], "estimated_tokens": 2000}
  ],
  "files_to_touch": ["src/x.ts", "src/y.ts", "tests/x.spec.ts"],
  "risks": ["regressão em y"],
  "fallbacks": ["feature flag", "rollback plan"]
}
```

Patch (Implementer → DIFF)

```
--- a/src/x.ts
+++ b/src/x.ts
@@
- código antigo
+ código novo
```

PR Summary (Critic → Markdown)

- Objetivo, escopo, riscos, casos de teste, roll-out, rollback, métricas.

---

5) Orquestração (LangGraph – exemplo de nós)

- **Nó A: Build Contract** (GPT-5)
- **Nó B: Retrieve** (BM25+Vector k=50)
- **Nó C: Rerank** (top 12)
- **Nó D: Implement** (Sonnet 4.5)
- **Nó E: Self-Check** (GPT-5): checagens formais, lint, segurança
- **Nó F: Test & Bench** (CI): unit/integration, snapshot perf
- **Nó G: Draft PR** (GPT-5) + etiquetas e riscos
- **Nó H: Human Gate**

Regras de rota:

- Falha em testes → volta para **Implement** com diffs e logs.
- Baixa confiança do reranker → volta para **Retrieve** com query expandida.

---

6) Quality Gates e Anti-Alucinação

- **Undefined Symbol Guard:** se `symbol not found` → bloqueia merge.
- **Context Drift Guard:** validar SHA das fontes citadas.
- **Grounded Citations:** as respostas devem incluir `file:path:lines`.
- **Security/Licensing Guard:** bloqueio de dependências proibidas, verificação de headers/licenças.
- **Test Coverage Gate:** patch deve manter/elevar cobertura dos arquivos tocados.

Checklist (por PR):

-

---

7) Métricas & Evals

Métricas operacionais

- Pass\@1 (tarefas de manutenção), taxa de retrabalho (PR reaberto), tempo até verde no CI, taxa de `MISSING_CONTEXT`.
- Indicadores de alucinação: símbolos inexistentes em diffs, testes quebrados por referências erradas.

Avaliações periódicas (semanais)

- Conjunto fixo de issues (mini SWE-bench interno).
- Benchmarks controlados para refatorações, bugs e features pequenas.

---

8) Integrações (GitHub & CI)

- **PR Bot:** cria PR com rótulos, checklists e matriz de riscos.
- **Code Review Automático:** gerar comentários, sugestões aplicáveis e verificação de políticas.
- **Rulesets:** exigir PR + revisão humana para branches protegidos.

---

9) Operação Diária (Runbook)

1. Abrir issue com escopo e critérios.
2. Rodar **Planner** → gera Contrato de Contexto.
3. **Retriever + Reranker** preparam Context Pack.
4. **Implementer** gera patch+tests.
5. **Critic** redige PR e checklist.
6. CI roda testes/linters/segurança.
7. Revisão humana e merge.

---

10) Templates de Prompt (prontos p/ uso)

10.1 Planner (GPT-5)

```
SISTEMA: Você é arquiteto de software. Produza um CONTRATO DE CONTEXTO v1.
REGRAS: não inventar símbolos; pode responder MISSING_CONTEXT.
ENTRADA: <issue + links>
SAÍDA: YAML conforme schema do contrato.
```

10.2 Implementer (Sonnet 4.5)

```
SISTEMA: Você é implementador rigoroso. Gere um patch unificado + testes.
REGRAS: usar apenas símbolos do Context Pack; incluir citações file:line.
ENTRADA: Contrato + Context Pack (top 12) + plano.
SAÍDA: DIFF + lista de arquivos tocados.
```

10.3 Critic (GPT-5)

```
SISTEMA: Você é revisor. Valide riscos, segurança e compatibilidade.
SAÍDA: PR.md (objetivo, impacto, riscos, rollback, test plan) + checklist OK/
FAIL.
```

---

11) Parametrização recomendada (start)

- `k_candidates = 50`, `rerank_top_n = 12`.
- Tamanho de chunk função/método com overlap 15–30 linhas.
- Limiar de confiança do reranker: 0.35–0.45; abaixo disso retornar `MISSING_CONTEXT`.
- Orçamento de tokens por role: Planner 1.5–3k; Implementer 2–6k; Critic 1–2k.

---

12) Roadmap de Adoção (30 dias)

Semana 1: instrumentação (indexador do repo, LangGraph esqueleto, CI hooks).
Semana 2: contratos + reranker + PR bot, piloto em 1 repo.
Semana 3: quality gates, métricas, rulesets.
Semana 4: ampliar para 3–5 repositórios; criar benchmark interno.

---

13) Anexos

- Checklists operacionais (por PR e por release)
- Esquemas JSON reutilizáveis (Plano, Patch Metadata, PR Summary)
- Tabelas de mapeamento: linguagem → linters/tests preferidos

---

Anexo A — Adaptação da Metodologia IA³ ao Workflow de Coding (RAG + Reranker)

A1. Mapeamento do Ciclo IA³ → Pipeline de Engenharia de Código

Conectar → Conectores MCP para GitHub, Wiki/ADR, Issue Tracker e FS. Indexação contínua do repositório (fingerprint por SHA), extração de símbolos/AST e criação de *Context Packs* versionados.

Analisar → Planner (GPT-5) gera Contrato de Contexto; Retriever (BM25+Vetores) reúne k=50; Reranker filtra top 12; Critic faz análise estática (símbolos, imports, riscos, compatibilidade) e validação de escopo.

Agir → Implementer (Sonnet 4.5) produz patch unificado + testes; PR Bot gera PR.md com objetivos, riscos, plano de rollout/rollback; CI executa testes, linters e checagens de política.

Aprender → Telemetria e Evals semanais (pass\@1, TTR até CI verde, taxa de `MISSING_CONTEXT`, reabertura de PRs). Ajuste de prompts, queries e *guardrails* baseado nos achados.

A2. Avaliação dos Frameworks (KEEP / ADAPT / DEFER)

1) Frameworks "Agentic"

- **Sequência Estratégica (Prompt Chaining) — KEEP**
- **Aplicação:** Grafo de execução (Planner → Retrieve → Rerank → Implement → Critic → PR). Saídas estritas (JSON/DIFF/MD) e checagens em cada nó.
- **Direcionamento Inteligente (Routing) — KEEP**
- **Aplicação:** Orquestrador decide rotas por tipo de tarefa (bugfix, refactor, feature) e por linguagem (TS/Java/Python), além de *fallbacks* (ex.: baixa confiança → expandir query).
- **Ação Simultânea (Parallelization) — ADAPT**
- **Aplicação:** Em vez de codar tudo em paralelo, paralelizamos **retrieval/estática** e **avaliações**. Geração de testes pode ser paralela ao *diff* **somente** após o plano aprovado para evitar deriva de escopo.
- **Refinamento Contínuo (Evaluator-Optimizer) — KEEP**
- **Aplicação:** Loop curto (máx. 2 iterações) entre Implementer ↔ Critic com critérios de saída: compila, testes passam nos arquivos tocados, sem símbolos inventados, *coverage* não cai.

2) Frameworks Adicionais

- **Memória Ativa (Knowledge Base Integration) — KEEP**
- **Aplicação:** Base de conhecimento = repo + ADR + *playbooks*. Leitura sempre permitida; escrita via PR (semântica e rastreável).
- **Supervisão Estratégica (Human-in-the-Loop) — KEEP**
- **Aplicação:** *Gates* humanos: aprovação de Contrato de Contexto; aprovação final do PR quando há riscos de migração/segurança.

- **Diálogo Construtivo (Multi-Agent Negotiation) — ADAPT**
- **Aplicação:** *Pair critics* especializados (Performance vs Segurança) negociam recomendações; arbitragem pelo Planner segundo critérios definidos no contrato (ex.: p95 < 50ms, OWASP).
- **Adaptação Inteligente (Dynamic Task Allocation) — KEEP**
- **Aplicação:** Filas por competência (linguagem/módulo). Carga e *budget* de tokens guiando a alocação do Implementer.
- **Evolução Autônoma (Reinforcement Learning) — DEFER** (fase 2)
- **Aplicação futura:** Bandits para escolher *prompts/templates* ou *query expanders*; só após termos telemetria estável.
- **Transparência Ativa (XAI) — KEEP**
- **Aplicação:** PR.md inclui *rationale* vinculado a `file:path:lines` e explicitação de riscos/trade-offs.
- **Inteligência Coletiva (Federated Learning) — DEFER**
- **Aplicação futura:** útil apenas se houver múltiplas unidades com dados sensíveis isolados.
- **Autonomia Contínua (Meta-Learning) — ADAPT**
- **Aplicação:** *Pattern library* por repositório (exemplos vencedores) usada como *few-shots* pelo Planner/Implementer via RAG, não como meta-treino pesado.

A3. Onde IA³ agrega valor direto

1. **Clareza de papéis e artefatos**: IA (agentes), Análise Humana (gates), Ação Estratégica (PR/rollout), todos amarrados pelo **Contrato de Contexto**.
2. **Ciclo fechado de aprendizado**: métricas operacionais → ajuste de prompts/retrieval → novas políticas.
3. **Governança**: transparência (citações de código), trilha de auditoria (PRs e *rulesets*), limites de escopo.

A4. Anti-padrões a evitar na adaptação

- Paralelizar geração de código **antes** do contrato aprovado.
- Contexto sem citações de linhas ou sem SHA, causando *drift*.
- Loops de refinamento abertos (sem orçamento de tokens/iterações).
- Atualizar wikis/KB diretamente sem PR.

A5. Quick Wins (1–2 semanas)

- Substituir o “brief” atual do Speckit por **Contrato de Contexto v1** (template já incluído).
- Ligar *Undefined-Symbol Guard* no CI e exigir citações `file:lines` em toda resposta do Implementer.
- Implementar reranker com *threshold* e retorno obrigatório `MISSING_CONTEXT` quando necessário.
- Criar *pattern library* de PRs exemplares para *few-shots* do Planner.

A6. Checklists de Conformidade IA³ (por PR)

-

Anexo B — Claude (Sonnet 4.5): Workflows, MCPs e Tools úteis ao nosso Blueprint

B1. Recursos do Claude a aproveitar

- **Tool Use (client/server tools)** com schemas JSON; suporte a *tool_choice* e *tool_result*.
- **Claude Code** (terminal) com sub-agentes, hooks, headless/SDK e integração GitHub Actions.
- **MCP (Model Context Protocol)** para conectar GitHub, FS, DBs e APIs como ferramentas padronizadas.
- **Code Execution Tool** (ambiente sandbox Bash/arquivos) p/ testes/linters/builds leves.
- **Computer Use** (beta) quando precisar automatizar UI (opcional, uso criterioso).
- **Prompt Caching** p/ reduzir custo/latência em prompts repetidos (contratos/templates).
- **PDF/Files API** p/ anexar ADRs/diagramas ao contexto do Implementer.

B2. Workflows prontos que casam com nosso pipeline

1. **Implementer no terminal (Claude Code)**
2. `/plan` (ou prompt) → gera plano conforme Contrato de Contexto.
3. `@repo/files` + MCP GitHub/FS → leitura guiada por símbolos; usa **sub-agentes** p/ testes/perf.
4. **Hooks**: após gerar DIFF, rodar script `pre-pr` (lint/tests) e `post-pr` (sincronizar ADR/CHANGELOG).
5. **Headless**: rodar via CI/CD (job “ai-implementer”) com JSONL de turns e artefatos (diff/tests/PR.md).
6. **PR Automation (GitHub Actions)**
7. Action do Claude Code cria comentários, aplica sugestões e roda checklists; roda também como *Critic* automático.
8. **Retriever/Research**
9. **Web Search/Fetch tool** e **Files/PDF** para buscar docs de arquitetura e portáteis como contexto para o Implementer.

B3. MCPs recomendados (por objetivo)

- **GitHub MCP**: issues, PRs, repos, comentários → leitura/escrita controladas por permissões.
- **Filesystem MCP**: leitura segmentada por pasta; *read-only* para contextos; escrita via patch/PR.
- **HTTP/Fetch/Web-search MCP**: buscar docs externos (RFCs, libs) com cache.
- **DB MCP (Postgres/SQLite)**: consultar esquemas/dados não sensíveis para migrações.
- **Tickets (Jira/Linear) MCP**: sincronizar status e critérios de aceitação direto no contrato.

  Política: habilitar somente MCPs **confiáveis**, *allow-list* estrita e escopos mínimos; registro de chamadas e IPs de saída.

B4. Tools do Claude — quando usar

- **Code Execution Tool**: executar `pytest`, `npm test`, linters, gerar *snapshots*; limites de tempo/IO; artefatos retornados ao agente.
- **Computer Use** (beta): apenas para tarefas que exigem GUI (IDE ou navegador corporativo); desligado por padrão.
- **Web Search / Web Fetch**: recuperar docs on-demand; sempre salvar citações (URL + hash do conteúdo) no PR.

B5. Configurações base (exemplos)

.claude/settings.json (projeto)

```
{
  "enableAllProjectMcpServers": false,
  "enabledMcpjsonServers": ["github", "filesystem", "http"],
  "outputStyle": "pull-request",
  "hooks": {
    "pre-pr": "./scripts/ai_pre_pr.sh",
    "post-pr": "./scripts/ai_post_pr.sh"
  }
}
```

MCP Connector via API (quando headless)

```
anthropic-beta: mcp-client-2025-04-04
```

Tool schema (trecho)

```
{
  "name": "run_tests",
  "description": "Executa testes e retorna junit + cobertura",
  "input_schema": {
    "type": "object",
    "properties": {"cmd": {"type": "string"}},
    "required": ["cmd"]
  }
}
```

B6. Guardrails e melhores práticas

- **tool_choice**: restrinja a ferramentas esperadas por etapa (ex.: Implementer não abre tickets).
- **Structured Output / JSON mode** p/ Plano/DIFF/PR; valide com schema.

- **Prompt Caching** para contratos/linhas-guia; 5min/1h conforme frequência.
- **Batch** para re-rodar Evals semanais com desconto de cache.
- **Observabilidade**: logar `tool_use` / `tool_result`, scores do reranker, custo e latência.

B7. Plano de integração (2 semanas)

- **D1–D3**: configurar Claude Code (terminal) + hooks + settings; MCPs (GitHub/FS/HTTP).
- **D4–D6**: implementar *run_tests* (Code Execution) e *retrieve_docs* (Web Fetch); ligar PR Action.
- **D7–D10**: headless no CI; Contrato de Contexto com Prompt Caching; Evals batch.
- **D11–D14**: piloto em 1 repo; ajustar *allow-list* de MCPs e limites de ferramenta.

Anexo C — TDD + Agentes (GPT-5 Planner / Sonnet Implementer)

C1. Posição

TDD continua sendo **excelente** para reduzir regressões e alinhar entendimento. Com agentes, adotamos **TDD++**: testes como *fonte de verdade* + *guardrails* automáticos para evitar "green por engano".

C2. Políticas (obrigatórias)

1. **Failure-first**: toda tarefa começa com **teste falhando**. Se não existir, o Implementer deve **criar/ajustar** o teste e comprovar o *RED* com log do runner.
2. **Sem "test-tampering"**: o agente não pode editar/remover testes para fazê-los passar **na mesma etapa** em que está implementando. Mudança de teste exige justificativa no PR e aprovação humana.
3. **Citações de contexto**: ao propor teste/implementação, referenciar `file:path:lines` (código alvo) e *ADR/issue*.
4. **Cobertura por arquivo tocado**: manter/elevar cobertura dos arquivos alterados.
5. **Mutation Gate**: *mutation score* mínimo (ex.: $\geq$ 60% na fase 1). Abaixo disso → voltar ao Implementer para fortalecer testes.
6. **Flakiness budget**: testes flakey são marcados e devem ser estabilizados antes de merge (no máximo 1 flaky por suíte).

C3. Workflow TDD++ (Red → Green → Refactor com agentes)

1. **Planner (GPT-5)**: traduz critérios de aceitação em **casos de teste** (unitários + integração) e define *test plan* (arquivos, *fixtures*, *mocks* admissíveis).
2. **Implementer (Sonnet) – Fase RED**: gera/edita testes, roda `run_tests`; anexa saída comprovando falha.
3. **Implementer – Fase GREEN**: implementa código mínimo; roda testes; anexa *diff* + logs verdes.
4. **Implementer – Fase REFACTOR**: melhorias internas sem alterar comportamento; roda testes; atualiza *diff*.

5. **Critic (GPT-5)**: revisa *test smells* (over-mocking, asserts fracos), cobertura por arquivo e risco de regressão; sugere testes adicionais.
6. **CI**: unit/integration + **mutation testing**; *ruleset* exige GREEN + Mutation Gate + ausência de flakiness crítica.

C4. Ferramentas recomendadas

- **Runner**: `pytest` / `jest` / `go test` / `mvn test` conforme stack.
- **Mutation testing**: Stryker (JS/TS/Java), PIT (Java), Mutmut (Python).
- **Property-based**: Hypothesis (Py), jqwik/QuickTheories (JVM), fast-check (TS).
- **Golden/snapshot tests**: somente com **approval** humano quando snapshots mudam.

C5. Templates de Prompt

Planner → Test Plan (JSON):

```
{
  "unit_tests": [
    {"file": "src/calc.ts", "name": "should_sum", "cases": ["1+2=3", "-1+2=1"]}
  ],
  "integration_tests": [
    {"name": "sum_endpoint_returns_200", "route": "/sum"}
  ],
  "mocks_allowed": ["HTTP externo"],
  "properties": ["comutatividade", "identidade com 0"],
  "mutation_targets": ["src/calc.ts"],
  "coverage_floor": {"src/calc.ts": 0.8}
}
```

Implementer (Fase RED):

```
Gere apenas TESTES conforme Test Plan. Execute e anexe o log mostrando falhas
esperadas. Não modifique código de produção nesta etapa.
```

Implementer (Fase GREEN):

```
Implemente o mínimo para passar os testes. Anexe DIFF + log verde. Cite
arquivos/linhas usadas do Context Pack.
```

Critic:

```
Avalie força dos asserts, over-mocking, cobertura por arquivo e riscos. Se
houver fraquezas, peça testes adicionais.
```

C6. Anti-padrões

- **Over-mocking** de módulos internos (mocks apenas nos limites do sistema).
- **Snapshots frágeis** para lógica crítica. Prefira asserts semânticos/propriedades.
- **Loops longos** de refinamento. Limite 2 iterações por tarefa antes de revisão humana.

C7. Hooks (pseudo)

pre-pr

```
set -e
npm run test:unit -- --reporter=junit || RED=1
[ -n "$RED" ] && echo "Falhas detectadas: confirmar fase GREEN" && exit 1
npm run mutation -- --min-score=60
```

post-pr

```
python scripts/check_flakiness.py --max-flaky=1
```

Anexo D — Ambiente Agnóstico de Stack (Zero/Low-Config)

> Meta: um único ambiente que funcione bem em qualquer projeto (JS/TS, Python, Java, Go, .NET, Ruby, Rust, PHP etc.) com o **menor número possível de ajustes**.

D1. Princípios

- **Convention > Configuration**: detectar stack e aplicar *presets* padronizados.
- **Imutabilidade/Hermeticidade**: execução em contêiner (Docker/Podman) por padrão; Nix como opção.
- **Interfaces estáveis**: comandos lógicos (**build/test/lint/mutate/coverage/format**) mapeados por *adapters* de stack.
- **Separação de preocupações**: agentes não decidem ferramenta de build; apenas chamam **interfaces**.
- **Fallbacks**: sempre há um comando default sensato caso o repo não tenha scripts.

D2. Arquitetura (camadas)

1. **Exec Layer**: contêiner base + *cache mounts* (npm/pip/maven/gradle/go). Suporte a `devcontainer.json` e modo Nix (opcional).
2. **Detect Layer**: detectores leem `package.json`, `pyproject.toml`, `pom.xml`, `build.gradle`, `go.mod`, `Cargo.toml`, `*.csproj`, `composer.json`, `Gemfile` etc.
3. **Adapter Layer**: para cada stack, um **adapter** expõe a mesma API:
4. `build`, `test`, `lint`, `format`, `coverage`, `mutation`, `package`, `run`
   Cada adapter conhece: binários, *args* padrão, *reporters* (JUnit/LCOV/Cobertura).
5. **Agent Layer**: Planner (GPT-5), Implementer (Sonnet) e Critic (GPT-5) invocam **tools**: `run_tests`, `apply_patch`, `run_lint`, `run_mutation`, `open_pr` via Claude Code/MCP.
6. **Context Layer (RAG)**: indexador (Tree-sitter/ctags) por função/método → OpenSearch/FAISS; busca híbrida + reranker; contexto com citações `file:path:lines`.

D3. Fluxo Zero-Config

1. **ai init**: detecta stack(s) → grava `.ai/stack.json` (opcional; pode rodar sem arquivo).
2. **ai plan**: Planner gera **Contrato de Contexto** e **Test Plan**.
3. **ai run**: Implementer usa adapters p/ `test` (RED) → `apply_patch` → `test` (GREEN) → `lint/format` → `mutation`.
4. **ai pr**: Critic gera PR.md + checklist e aciona `open_pr`.
5. **ai eval**: roda *evals* semanais (batch) com *reports* unificados.

D4. Detecção de Stack (heurística)

- **Node**: `package.json` (priorizar `pnpm-lock.yaml` > `yarn.lock` > `package-lock.json`).
- **Python**: `pyproject.toml` (poetry/pdm), `requirements.txt`.
- **Java**: `pom.xml` / `build.gradle[.kts]`.
- **Go**: `go.mod`.
- **.NET**: `*.csproj` / `*.sln`.
- **Ruby**: `Gemfile`.
- **Rust**: `Cargo.toml`.
- **PHP**: `composer.json`.

Multi-stack (monorepo): detecção por subpastas; cada *package* vira um **target** com seu adapter.

D5. Mapeamento de comandos (defaults sensatos)

Node (JS/TS)

- `build`: `npm run build || tsc -p .`
- `test`: `npm test || jest`
- `lint`: `npm run lint || eslint .`
- `format`: `npm run format || prettier -w .`

- `coverage`: `npm run coverage || jest --coverage --coverageReporters=cobertura,lcov`
- `mutation`: `stryker run` (se presente)

Python

- `build`: `pip install -e .` (se `pyproject`), senão `pip install -r requirements.txt`
- `test`: `pytest -q --junitxml=reports/junit.xml`
- `lint`: `ruff check . || flake8`
- `format`: `ruff format . || black .`
- `coverage`: `coverage run -m pytest && coverage xml`
- `mutation`: `mutmut run` (se presente)

Java

- Maven: `mvn -q -DskipTests=false test surefire-report:report` (gera JUnit)
- Gradle: `./gradlew test jacocoTestReport`
- `mutation`: PIT se plugin disponível

Go

- `test`: `go test ./... -json > reports/go-test.json`
- `coverage`: `go test ./... -coverprofile=coverage.out && gocov convert`
- `lint`: `golangci-lint run`
- `mutation`: gopter/stryker-go (se adotado)

.NET

- `test`: `dotnet test --logger "trx;LogFileName=TestResults.trx" --collect:"XPlat Code Coverage"`

Rust

- `test`: `cargo test --message-format=json`
- `coverage`: `cargo tarpaulin --out Xml`

Todos os adapters produzem **JUnit** (ou TRX), **Cobertura/LCOV** e logs padronizados em `reports/`.

D6. Arquivos padrão (opt-in)

.ai/stack.json (gerado por `ai init`, opcional)

```
{
  "adapters": [
    {"path": "apps/web", "stack": "node"},
```

```
    {"path": "services/api", "stack": "python"}
  ],
  "policies": {
    "mutation_min": 0.6,
    "coverage_floor": 0.8,
    "allow_snapshots": false
  }
}
```

.claude/settings.json (com hooks)

```
{
  "enabledMcpjsonServers": ["github", "filesystem", "http"],
  "outputStyle": "pull-request",
  "hooks": {"pre-pr": "./scripts/ai_pre_pr.sh", "post-pr": "./scripts/
ai_post_pr.sh"}
}
```

D7. Contêiner base

- **Imagem**: `ai/stack-base:latest` com toolchains comuns: Node LTS, Python 3.x, Java 21 (Maven/Gradle), Go, .NET SDK, Rust, bash utils.
- **Volumes de cache**: `~/.cache/pip`, `~/.npm`, `~/.pnpm-store`, `~/.m2`, `~/.gradle`, `~/.cache/go-build`.
- **Entrypoint**: `ai-cli` (descrito abaixo).

docker-compose.yml (trecho)

```
services:
  ai:
    image: ai/stack-base:latest
    volumes:
      - ./:/workspace
      - npm:/root/.npm
      - pip:/root/.cache/pip
      - m2:/root/.m2
    working_dir: /workspace
    entrypoint: ["ai-cli"]
volumes:
  npm: {}
  pip: {}
  m2: {}
```

D8. `ai-cli` (interface única)

Comandos principais: `init`, `detect`, `plan`, `run`, `pr`, `eval`.

- `detect`: imprime JSON com stacks/targets detectados.
- `run --task test|lint|mutation`: chama o adapter apropriado e normaliza relatórios.
- `apply-patch`: aplica *diff* vindo do Implementer de forma segura.
- `reports upload`: publica JUnit/coverage para a CI.

D9. Integração CI (GitHub Actions, genérica)

`.github/workflows/ai.yml`

```
name: AI Pipeline
on: [pull_request]
jobs:
  ai:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: docker/setup-buildx-action@v3
      - name: Run AI Checks
        run: |
          docker compose run --rm ai detect
          docker compose run --rm ai run --task test
          docker compose run --rm ai run --task lint
          docker compose run --rm ai run --task mutation || true
      - name: Publish Reports
        run: docker compose run --rm ai reports upload
```

D10. Integração com Agentes

- **Planner (GPT-5)**: produz Contrato de Contexto + Test Plan (JSON).
- **Implementer (Sonnet via Claude Code)**: invoca `ai-cli run --task test` (Fase RED), `apply-patch`, `run --task test` (GREEN), `run --task lint|format|mutation`.
- **Critic (GPT-5)**: lê relatórios padronizados em `reports/`, gera PR.md e checklist IA³.

D11. Observabilidade

- Logs JSON por etapa (`logs/*.jsonl`) com: tarefa, stack, duração, custo/tokens, pass/fail, *rerank score* médio.
- Métricas por repo/adapter: pass\@1, TTR (até CI verde), taxa de `MISSING_CONTEXT`, cobertura por arquivo tocado, *mutation score*.

D12. Segurança e Governança

- **Allow-list** de comandos por adapter; *no-network* opcional em `test`.
- **Secrets** só via CI/OpenID/Secret Manager.
- **RBAC** para MCPs (GitHub/FS/HTTP).
- **Proveniência**: anexar SHA dos arquivos citados no PR.

D13. Rollout

- **Semana 1**: imagem base + `ai-cli` (detect/run) + Node/Python adapters.
- **Semana 2**: Java/Go adapters + mutation testing.
- **Semana 3**: .NET/Rust + reports unificados + Observabilidade.
- **Semana 4**: afinar defaults, abrir *pattern library* por stack, piloto multi-repo.