Data

HoverBoard: AI Code documentation

Wave 2

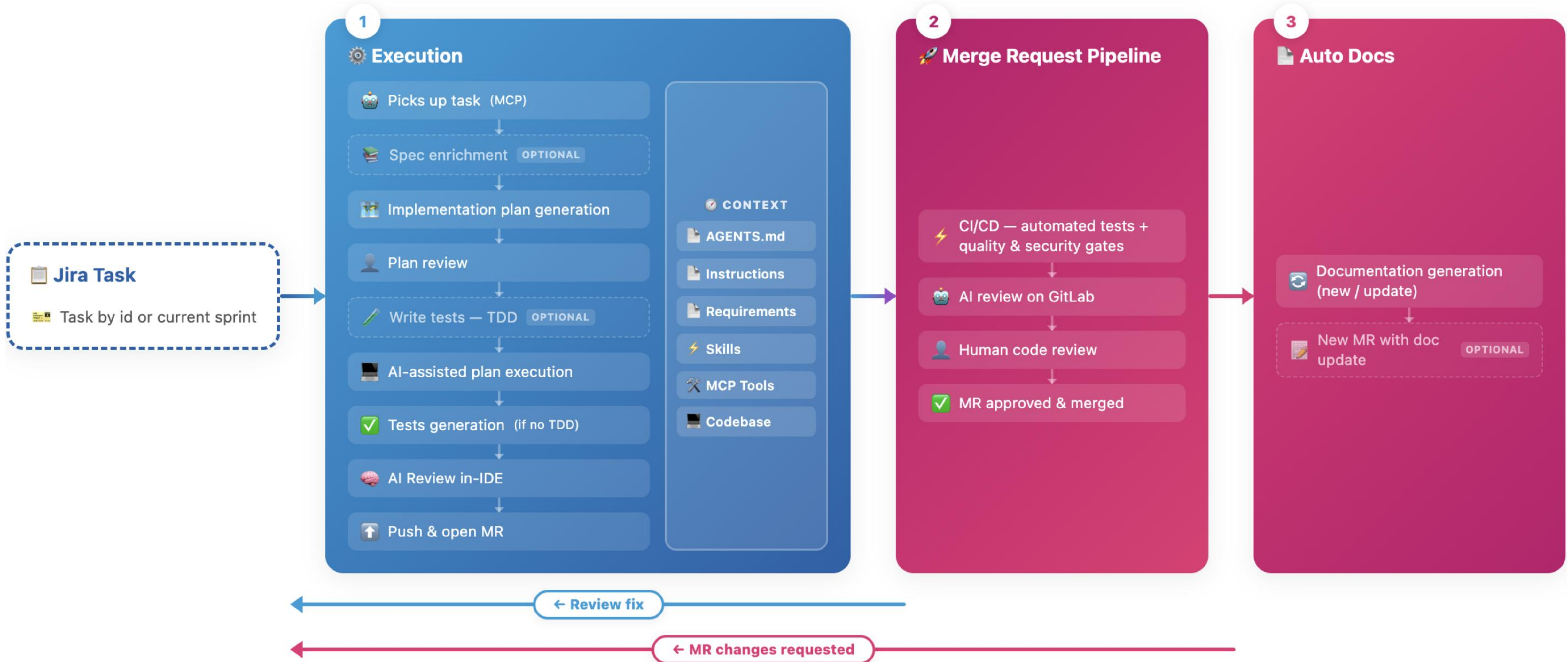
Engineering Documentation Pipelines with GitHub Copilot

Transforming AI from a code completer to an engineered and deterministic system for documentation.

> Status: Executing architectural blueprint...

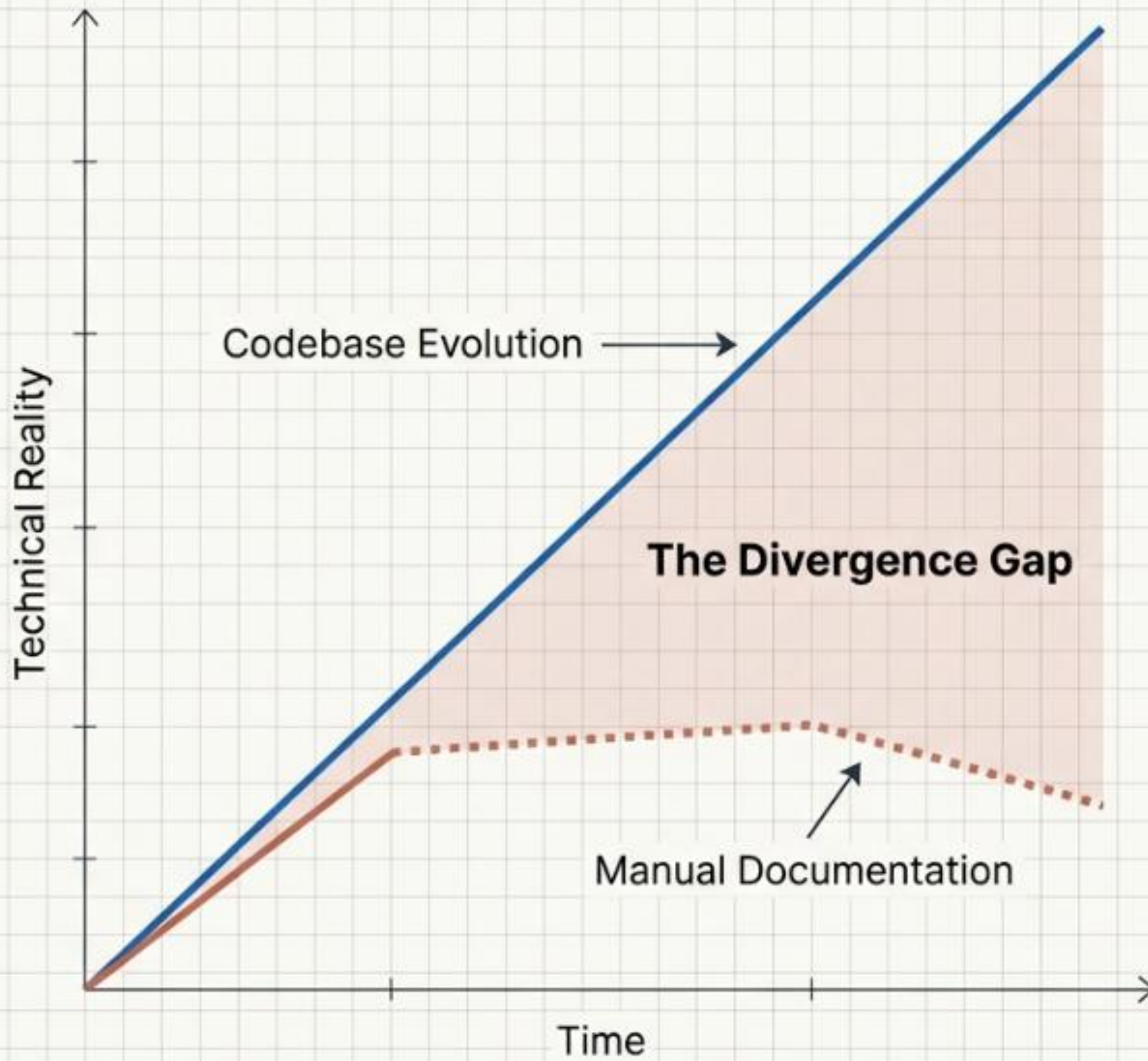
AI-Assisted SDLC

From Jira ticket to production — enhanced by AI at every step



1976 - Ferrari 312T

2023 - Ferrari SF23



The Inevitable Crisis of Documentation Drift

The Reality

Code evolves in rapid release cycles. Manual documentation is inherently obsolete the moment it is published.

The Cost

Degraded team velocity, massive spikes in context-switching, and critical knowledge silos.

The Failure

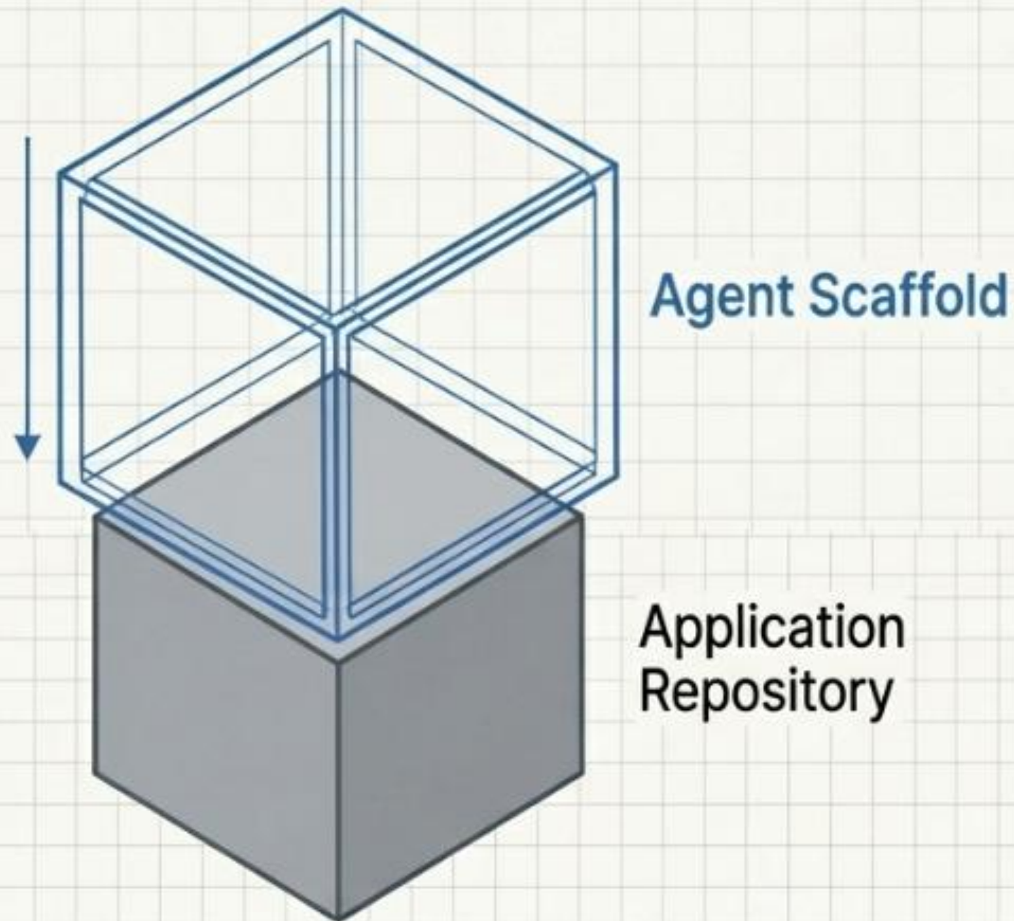
Relying on human memory and ad-hoc post-release chores to maintain system truth is a structurally flawed engineering practice.

The Evolution of Technical Documentation

Manual Documentation	Ad-Hoc AI Prompting	AI Agent Scaffold
Process: Post-release chore	**Process:** Unorchestrated Q&A	**Process:** **Automated**, evidence-based pipeline
Effort: High & unsustainable	**Effort:** Low	**Effort:** Initial setup + seamless generation
Drift Risk: Critical (Immediate obsolescence)	**Hallucination Risk:** Critical (Assumes generic patterns)	**Drift Risk:** **Zero** (Tethered to code state)
Trust Level: Low	**Trust Level:** Zero	**Trust Level:** **Absolute** (Verifiable truth)

A Meta-Tool, Not a Black Box Application

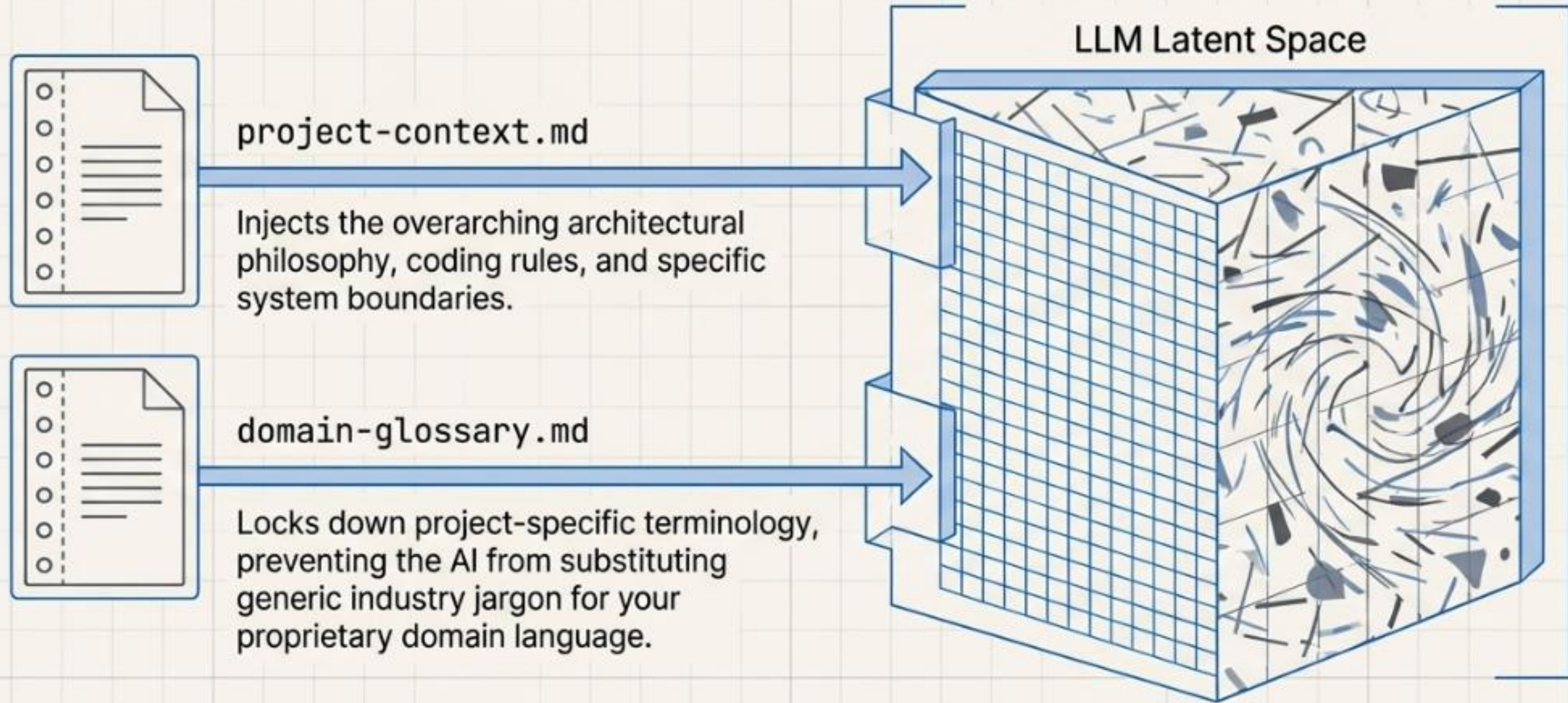
The AI Documentation Agent Scaffold contains zero runtime code.



> It is a reusable configuration framework designed natively for GitHub Copilot.

> It provides structured prompts, specialized AI skills, workflows, and markdown templates.

> It orchestrates AI to operate as an autonomous, rigorous documentation agent embedded within the repository's DNA.


Setup & Context Injection: Anchoring the LLM
The prerequisite for precision: forcibly tethering the LLM to the actual reality of your project.
LLM Latent Space
project-context.md
Injects the overarching architectural philosophy, coding rules, and specific system boundaries.
domain-glossary.md
Locks down project-specific terminology, preventing the AI from substituting generic industry jargon for your proprietary domain language.
Result: Eliminates hallucinations derived from generic assumptions.

Domain-glossary and *project-context.md*

📌 Domain Glossary and Project

The Domain Glossary is:

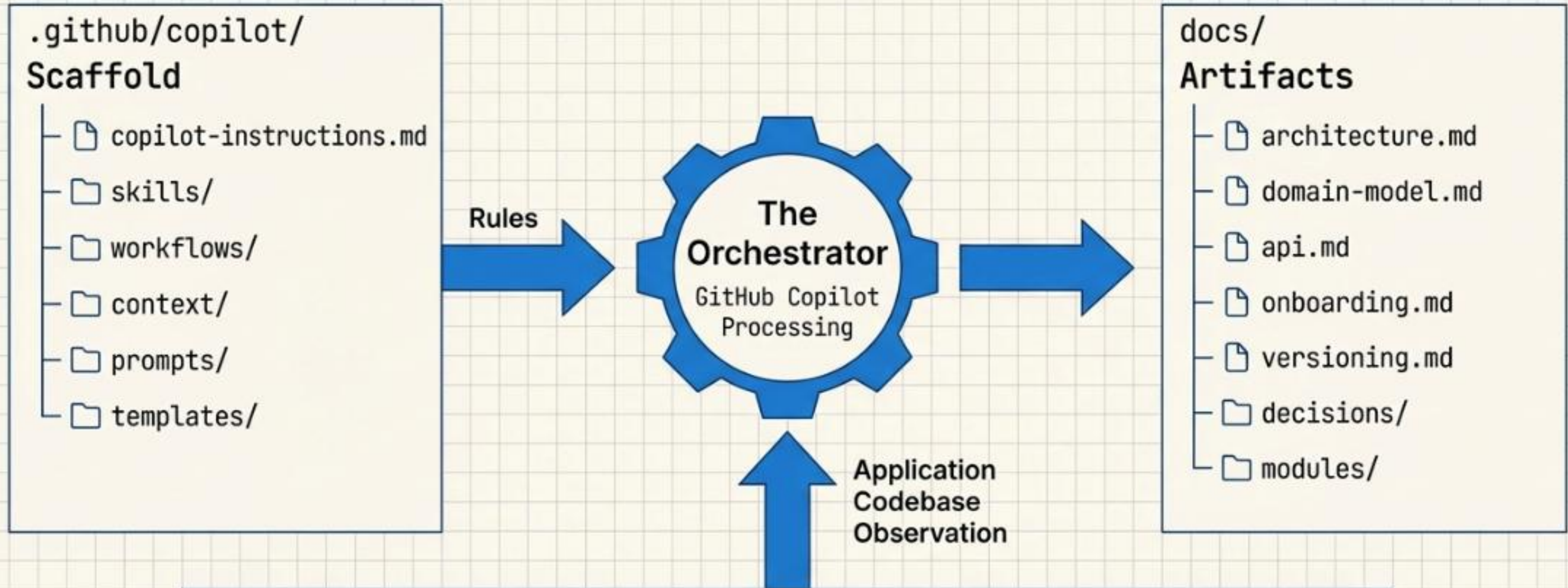
- a **source of linguistic truth**
- a **fundamental input for Copilot**
- a tool to make documentation:
- consistent
- clear
- scalable

🧠 What is *project-context.md*

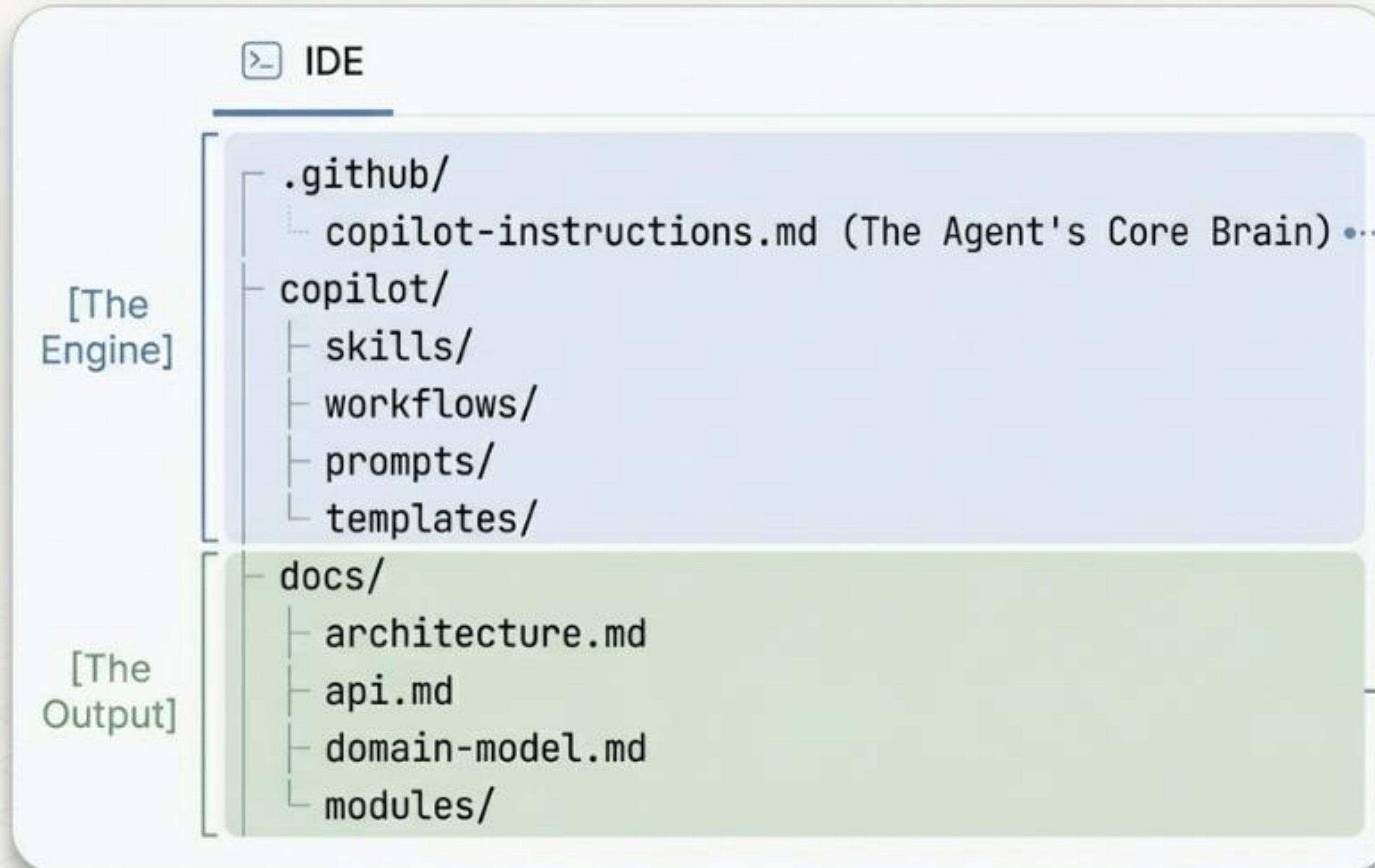
It is a file that:

- describes **the global context of the project**
- guides the AI (e.g., GitHub Copilot) in its responses
- reduces ambiguities when generating documentation, code, or architectural decisions

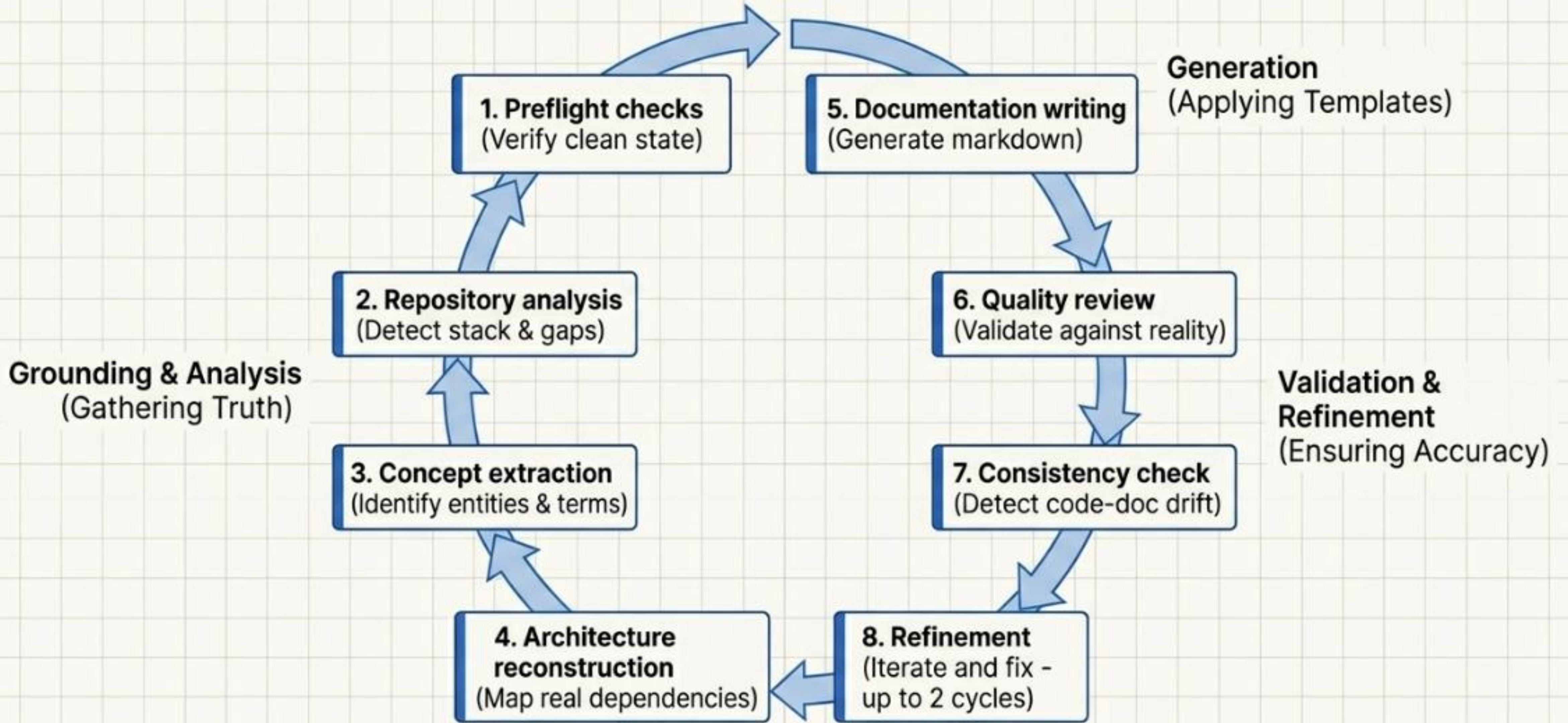
👉 In practice: it is your project's **"persistent prompt"**.

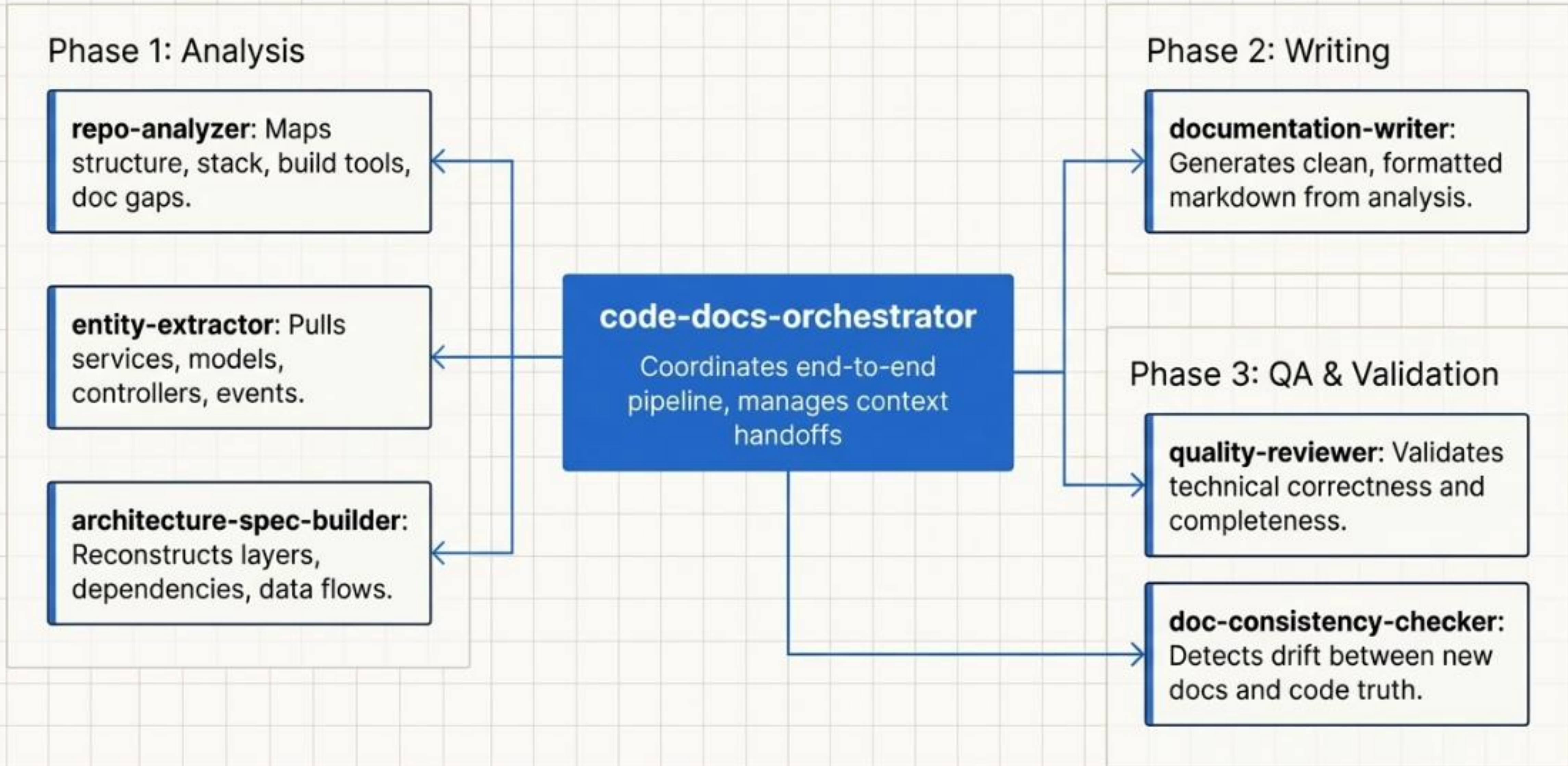

Non-Invasive Architecture: The Engine
.github/copilot/
Scaffold
copilot-instructions.md
skills/
workflows/
context/
prompts/
templates/
Rules
The
Orchestrator
GitHub Copilot
Processing
docs/
Artifacts
architecture.md
domain-model.md
api.md
onboarding.md
versioning.md
decisions/
modules/
Application
Codebase
Observation
The AI acts as a highly informed external observer. It draws deep context
without ever interfering with or polluting the application code.

The Self-Contained Repository Ecosystem



No external SaaS integrations required. The logic lives natively where the code lives.


The 8-Step Verification Loop
1. Preflight checks
(Verify clean state)
5. Documentation writing
(Generate markdown)
Generation
(Applying Templates)
2. Repository analysis
(Detect stack & gaps)
6. Quality review
(Validate against reality)
Grounding & Analysis
(Gathering Truth)
Validation &
Refinement
(Ensuring Accuracy)
3. Concept extraction
(Identify entities & terms)
7. Consistency check
(Detect code-doc drift)
4. Architecture
reconstruction
(Map real dependencies)
8. Refinement
(Iterate and fix -
up to 2 cycles)


The Anatomy of Intelligence: 7 Specialized Skills
Phase 1: Analysis
repo-analyzer: Maps structure, stack, build tools, doc gaps.
entity-extractor: Pulls services, models, controllers, events.
architecture-spec-builder: Reconstructs layers, dependencies, data flows.
code-docs-orchestrator
Coordinates end-to-end pipeline, manages context handoffs
Phase 2: Writing
documentation-writer: Generates clean, formatted markdown from analysis.
Phase 3: QA & Validation
quality-reviewer: Validates technical correctness and completeness.
doc-consistency-checker: Detects drift between new docs and code truth.

Phase 1: Context Extraction

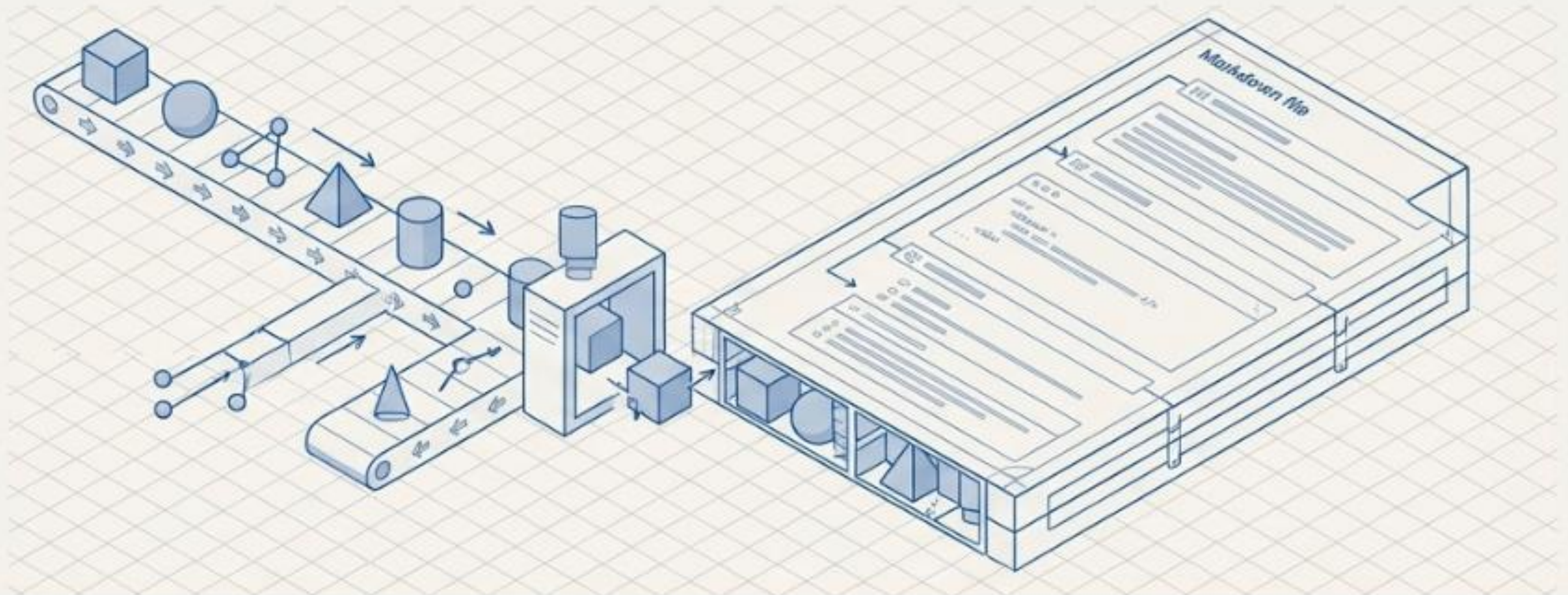
Step 1: Repository Analysis `repo-analyzer`

Action	Output
Scans languages, frameworks, folder structures, entrypoints, build commands, and test setups.	Repository summary, structure map, documentation gaps.

Step 2: Concept Extraction `entity-extractor`

Action	Output
Identifies controllers, DTOs, jobs, events, and services.	Domain entities, technical components, project glossary.

Phase 2: Synthesis and Generation

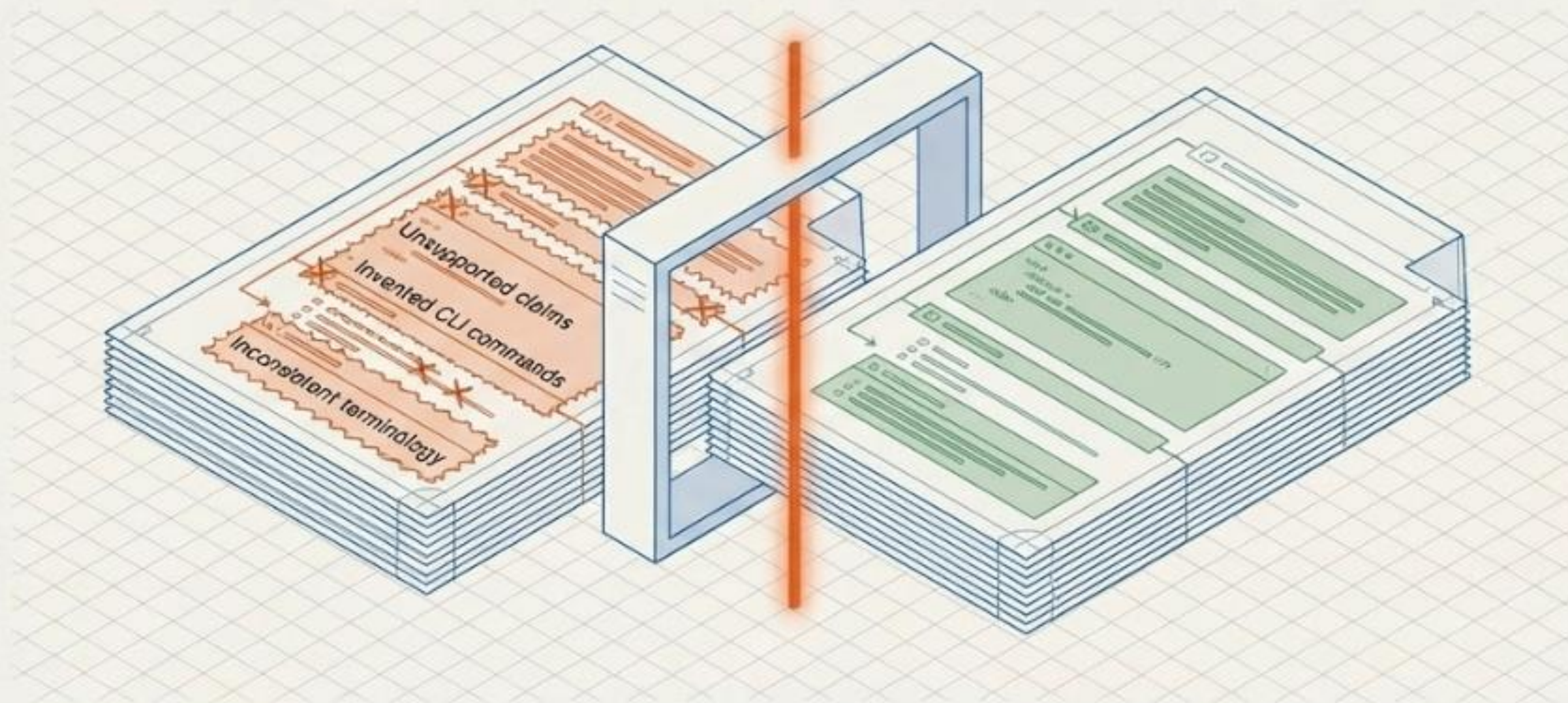


Step 3: Architecture Reconstruction `architecture-spec-builder`

Action: Reverse-engineers dependencies, data flows, and external integrations.
Output: System overview, component diagrams, flow definitions.

Step 4: Documentation Writing `documentation-writer`

Phase 3: Automated Quality Control



Step 5: Quality Review `quality-reviewer`

Actively audits outputs for unsupported claims, invented CLI commands, inconsistent terminology, and missing architectural sections.

→

Step 6: Refinement Orchestration `code-docs-orchestrator`

Applies necessary fixes, assembles the final documentation suite, and flags open architectural questions for human engineers.

Commanding the Orchestrator

Full Pipeline Execution

```
> Act as code-docs-orchestrator. Run the documentation pipeline defined in
  .github/copilot/workflows/documentation-pipeline.md
```

Action: Triggers the 8-step end-to-end generation for the entire repository.

Granular Module Generation

```
> Act as documentation-writer. Analyze the repository and generate docs/architecture.md
```

Action: Executes targeted updates for a single structural file.

Continuous Quality Assurance

```
> Act as quality-reviewer. Review the repository documentation
```

Action: Audits existing documentation against current code state to flag new drift.

Predictable Evolution: Semantic Versioning (SemVer)

Current Release: Version 1.0.0 (Released 2026-03-14)

`MAJOR (2.0.0)`	Breaking changes to the `.github/` scaffold structure or core architecture.
`MINOR (1.1.0)`	Introduction of new specialized skills, workflows, or markdown templates.
`PATCH (1.0.1)`	Prompt tuning, bug fixes, or internal documentation improvements.

Designed for incremental updates. The scaffold evolves systematically so your documentation capabilities scale safely alongside your codebase.

The Pragmatic Truth of AI Documentation

The Automated Reality (What the System Solves)

- Drastic reduction in documentation debt.
- Frictionless, accelerated onboarding.
- Consistent, explicit architectural records.
- Systematic suppression of AI hallucinations.

The Engineering Reality (What Still Requires Humans)

- Does not replace engineering judgment.
- Requires human architectural review.
- Requires high-level domain context validation.

Ultimate credibility requires refusing to over-promise.

Starting: *Step 4: Documentation writing (documentation-writer)* (5/9)

Let me read the existing docs and README to understand what needs updating vs creating.

Reviewed 8 files and searched for *.md

All existing docs are placeholder stubs (except the README which has duplicated/outdated Italian content). Now I need to read
to ensure accuracy for the API documentation, then write all documents.

Reviewed 4 files

Now I have all the information needed to write the documentation. Let me write all the documents. I'll write all six target files in

Todos (5/9)

- Step 0: Preflight checks
- Step 1: Repository analysis (repo-analyzer)
- Step 2: Concept extraction (entity-extractor)
- Step 3: Architecture reconstruction (architecture-spec-builder)
- Step 4: Documentation writing (documentation-writer)
- Step 5: Quality review (quality-reviewer)

Let me re-read api.md first, then write it. Also need to create the module docs.

Reviewed and updated api.md

Now let me create the module documentation files.

Todos (5/9)

- Step 3: Architecture reconstruction (architecture-spec-builder)
- Step 4: Documentation writing (documentation-writer)
- Step 5: Quality review (quality-reviewer)
- Step 6: Consistency check (doc-consistency-checker)
- Step 7: Refinement (code-docs-orchestrator)
- Step 8: Post-pipeline validation

5 files changed +870 -235

- api.md docs
- architecture.md docs
- domain-model.md docs
- onboarding.md docs
- README.md

+ *workshop.md*

Step 4 is complete. Now let me move to Steps 5 and 6 — Quality Review and Consistency Check — which can run in parallel.

Explore: Quality review of docs

Explore: Consistency check of docs

Starting: *Step 5: Quality review (quality-reviewer)* (6/9)

Steps 5 and 6 identified issues to fix. The quality review found 2 major + 6 minor issues, and the consistency check scored 79/100 (below the 85 threshold). Let me apply refinements (Step 7) to resolve the key issues.

Starting: *Step 7: Refinement (code-docs-orchestrator)* (8/9)

Let me re-read the files I need to fix, then apply all corrections.

Reviewed 4 files

Now I'll apply all the refinement fixes identified in the quality review and consistency check. Let me apply them in parallel.

Todos (8/9)

- [x] Step 3: Architecture reconstruction (architecture-spec-builder)
- [x] Step 4: Documentation writing (documentation-writer)
- [x] Step 5: Quality review (quality-reviewer)
- [x] Step 6: Consistency check (doc-consistency-checker)
- [ ] Step 7: Refinement (code-docs-orchestrator)
- [ ] Step 8: Post-pipeline validation

Final Deliverables

12 documentation files generated/updated:

File	Lines
`README.md`	127
`docs/architecture.md`	175
`docs/domain-model.md`	162
`docs/api.md`	~308
`docs/onboarding.md`	174
`docs/modules/trade-items-read.md`	[illegible]
`docs/modules/trade-items-write.md`	
`docs/modules/common.md`	
`docs/modules/auth.md`	
`docs/modules/dapr.md`	
`docs/modules/health.md`	
`docs/modules/README.md`	

Recommended Next Tasks

1. Add Architecture Decision Records (ADRs) in `docs/decisions/`
2. Document the Hermes Outbox Relay integration and downstream consumer topology
3. Document the end-to-end migration workflow in more detail
4. Complete `.env.example` with all environment variables (many are undocumented there)
5. Add integration/e2e test documentation once tests are written

Todos (9/9)

- ✓ Step 3: Architecture reconstruction (architecture-spec-builder)
- ✓ Step 4: Documentation writing (documentation-writer)
- ✓ Step 5: Quality review (quality-reviewer)
- ✓ Step 6: Consistency check (doc-consistency-checker)
- ✓ Step 7: Refinement (code-docs-orchestrator)
- ✓ Step 8: Post-pipeline validation

TeamSystem

Q&A

“Does it work with any language?”

Yes, because it is **repository-driven** rather than language-specific. The better structured the repo, the better the output.

“How reliable is it?”

Reliable when used as an evidence-based pipeline. It is much safer than a single prompt, but for critical systems human review still matters.

“Can it run in CI?”

Yes. That is actually one of the strongest use cases: drift detection, quality checks, and eventually automated doc generation.

“What about token cost?”

It depends on repo size, but usually the real comparison is not tokens versus zero — it is tokens versus engineering hours lost to missing or outdated documentation.

Reference

[GitLab Repo code documentation]

Thank you!