

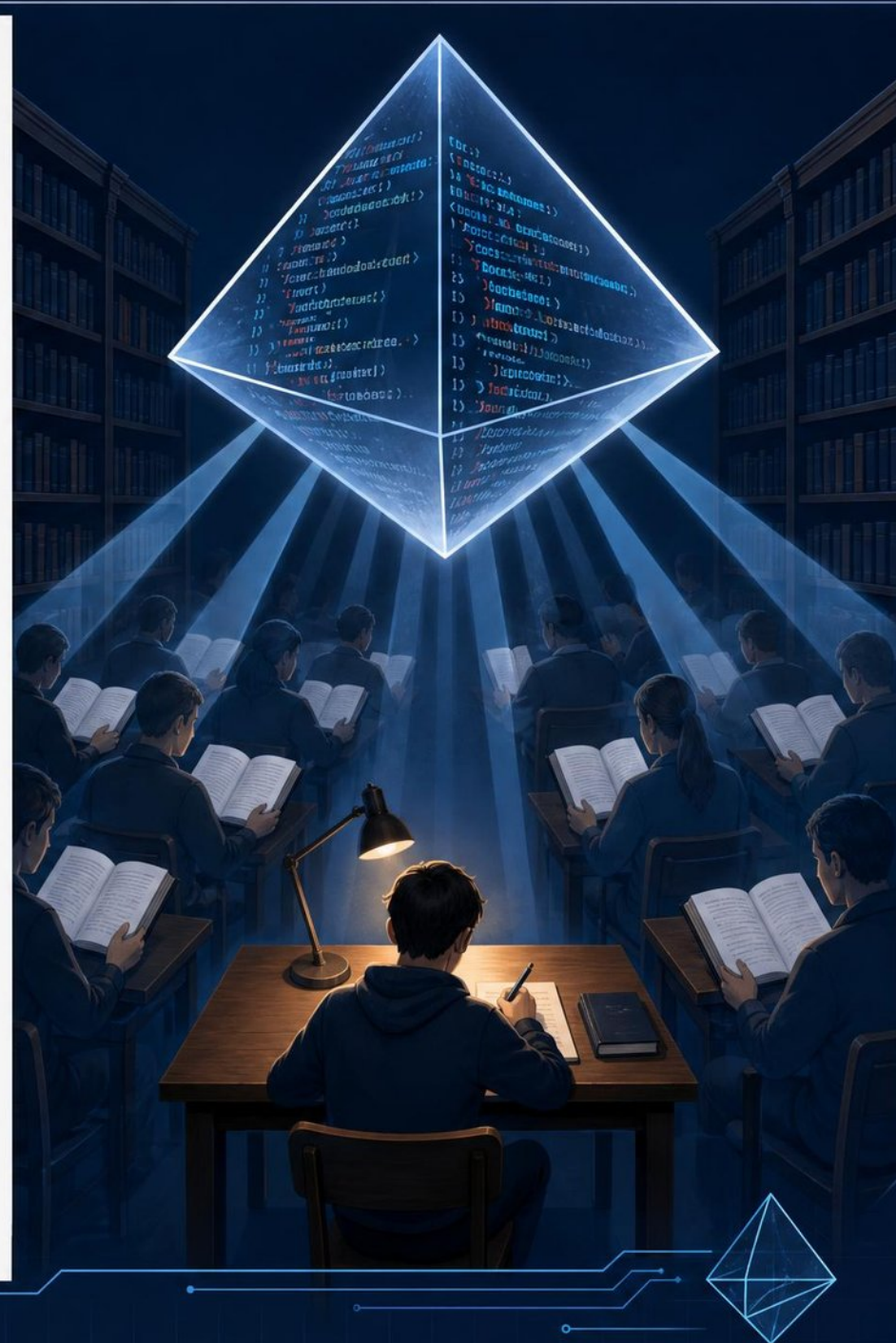
Technical Writing for Software Developers: Course Overview

- - Core developer skill, not an afterthought
- - Covers: doc types, task-oriented instructions, API references, examples, style, maintenance
- - Outcomes: diagnose, rewrite, test, and maintain docs
- - Addresses drift, unclear audience, broken examples
- - Balances reference, guides, tutorials, release notes, real fixes



Why Technical Writing Matters for Software Teams

- Code is read far more than it is written; docs amplify this ratio
- Writing exposes gaps in reasoning, edge cases, and hidden assumptions
- Poor docs raise support load, slow onboarding, and erode developer trust
- Senior engineers produce design docs, RFCs, ADRs, and PR descriptions
- Documentation quality is now a key purchase and adoption signal



Documentation Types and the Jobs They Do

- Diátaxis framework: tutorials, how-to guides, reference, and explanation
- Tutorials teach; how-to guides solve tasks; reference provides facts; explanation builds understanding
- Mixing modes on one page is the most common structural mistake
- Typical inventory: quickstarts, API refs, user guides, troubleshooting, release notes
- Reader task determines page type: get started, complete a task, look up, or understand

TUTORIAL



Teaches foundational concepts and builds knowledge step by step.

HOW-TO GUIDE



Provides actionable steps to complete a specific task.

REFERENCE



Presents facts and details for precise lookup and accurate use.

EXPLANATION

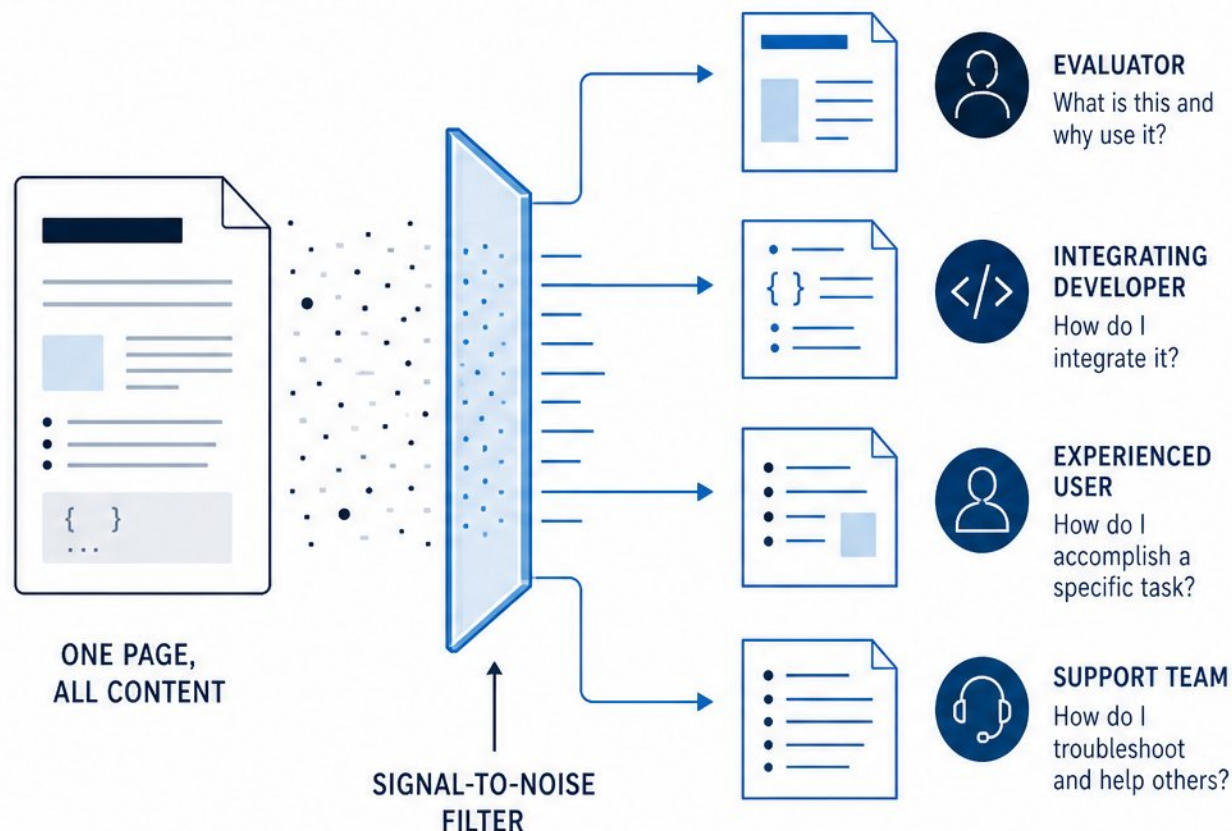


Explains ideas and context to build deep understanding.

Matching Documentation Type to Audience and Context

- Documentation must target distinct audiences: evaluators, integrators, support, and internal engineers
- Task context dictates structure: evaluate, integrate, look up, debug, or decide
- Developer and user documentation serve different goals, even for the same product
- Common failure: mixing tutorial, how-to, reference, and explanation on one page
- A useful page answers one audience's question completely, not every audience partially

THE SAME PRODUCT PAGE, FILTERED FOR DIFFERENT AUDIENCES





Writing Clear, Task-Oriented Instructions



- Lead with the task and outcome; add prerequisites and context



- Use imperative voice for short, specific, testable steps



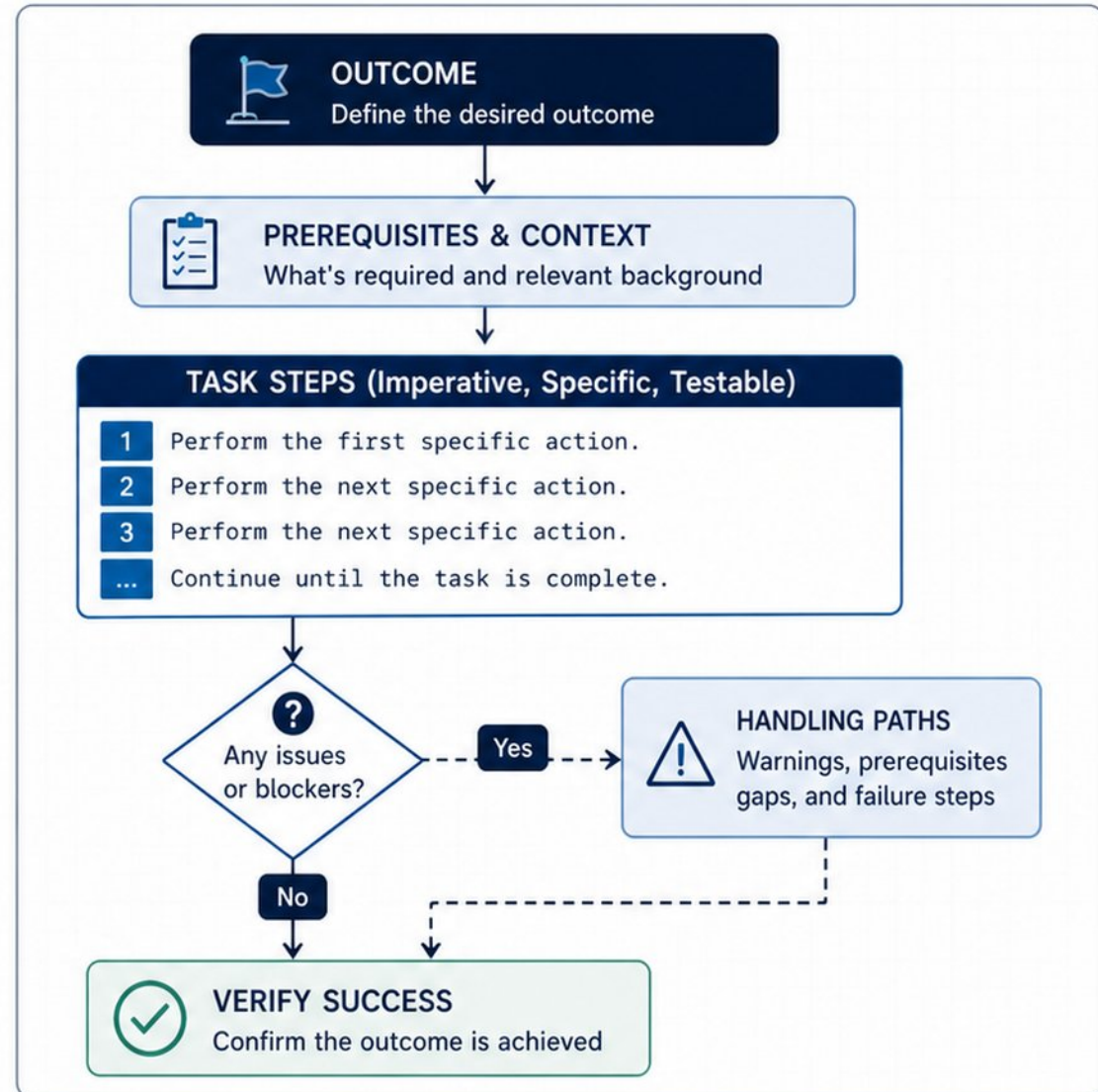
- Handle prerequisites, warnings, and failure paths explicitly



- Replace vague verbs like "handle" with precise actions



- Numbered lists for sequence; bullets for options; verify success



Writing Effective API Reference Content



- Lead with plain-language purpose, then method, path, and parameters



- Document types, defaults, constraints, and valid values in tables



- Give authentication and error handling their own dedicated sections



- List error codes verbatim, with causes and recommended next actions



- Use consistent naming and status-code explanations across all endpoints

Get User

Retrieve a user by ID.

GET /users/{userId}

Parameters

Name	In	Type	Required	Description
userId	path	string	Yes	Unique identifier of the user
include	query	string	No	Related resources to include
fields	query	string	No	Comma-separated list of fields to return

Response Schema (200)

```
{
  "id": "string",
  "name": "string",
  "email": "string",
  "profile": {
    "title": "string",
    "avatarUrl": "string"
  },
  "roles": ["string"]
}
```

Error Codes

Code	Cause	Recommended Next Action
INVALID_REQUEST	The request was malformed or missing required parameters.	Review the request and ensure all required parameters are provided with correct values.
NOT_FOUND	The specified resource does not exist.	Verify the userId and try again.
UNAUTHORIZED	Authentication is missing or invalid.	Provide a valid access token and retry.
FORBIDDEN	The token does not have access to this resource.	Ensure the token has the necessary permissions.
SERVER_ERROR	An unexpected error occurred on the server.	Retry the request later. If the issue persists, contact support.



Request Example

GET /users/{userId}?include=profile,roles&fields=id,name,email

Headers:

Authorization: Bearer <token>

Accept: application/json

Authentication

This endpoint requires a valid access token.

Include the token in the **Authorization** header as a **Bearer** token.

Error Handling

All error responses follow a consistent schema with an **error** object.

See **Error Codes** for details.

Writing Code Examples That Actually Help



- Every example is copy-paste-run; only credentials and IDs change



- Include imports, setup, expected output, and error cases



- Name variables, functions, and placeholders consistently



- Show realistic responses, not just happy-path success



- Test samples against the current API; automate in CI

```
# Imports
import requests
import json
```

```
# Setup
BASE_URL = "https://api.example.com"
API_KEY = "YOUR_API_KEY"
headers = {"Authorization": f"Bearer {API_KEY}",
           "Content-Type": "application/json"}
```

```
# Input / Request
payload = {"user_id": "YOUR_USER_ID"}
response = requests.post(f"{BASE_URL}/users",
                        headers=headers, json=payload)
```

```
# Expected Output
print(response.status_code)
print(json.dumps(response.json(), indent=2))
```

```
# Error Case (e.g., missing user)
bad_payload = {"user_id": ""}
bad_response = requests.post(f"{BASE_URL}/users",
                             headers=headers, json=bad_payload)
print(bad_response.status_code)
print(json.dumps(bad_response.json(), indent=2))
```



IMPORTS

Everything needed to run



SETUP

Configuration and environment



INPUT

What is sent to the API



OUTPUT

Realistic successful response

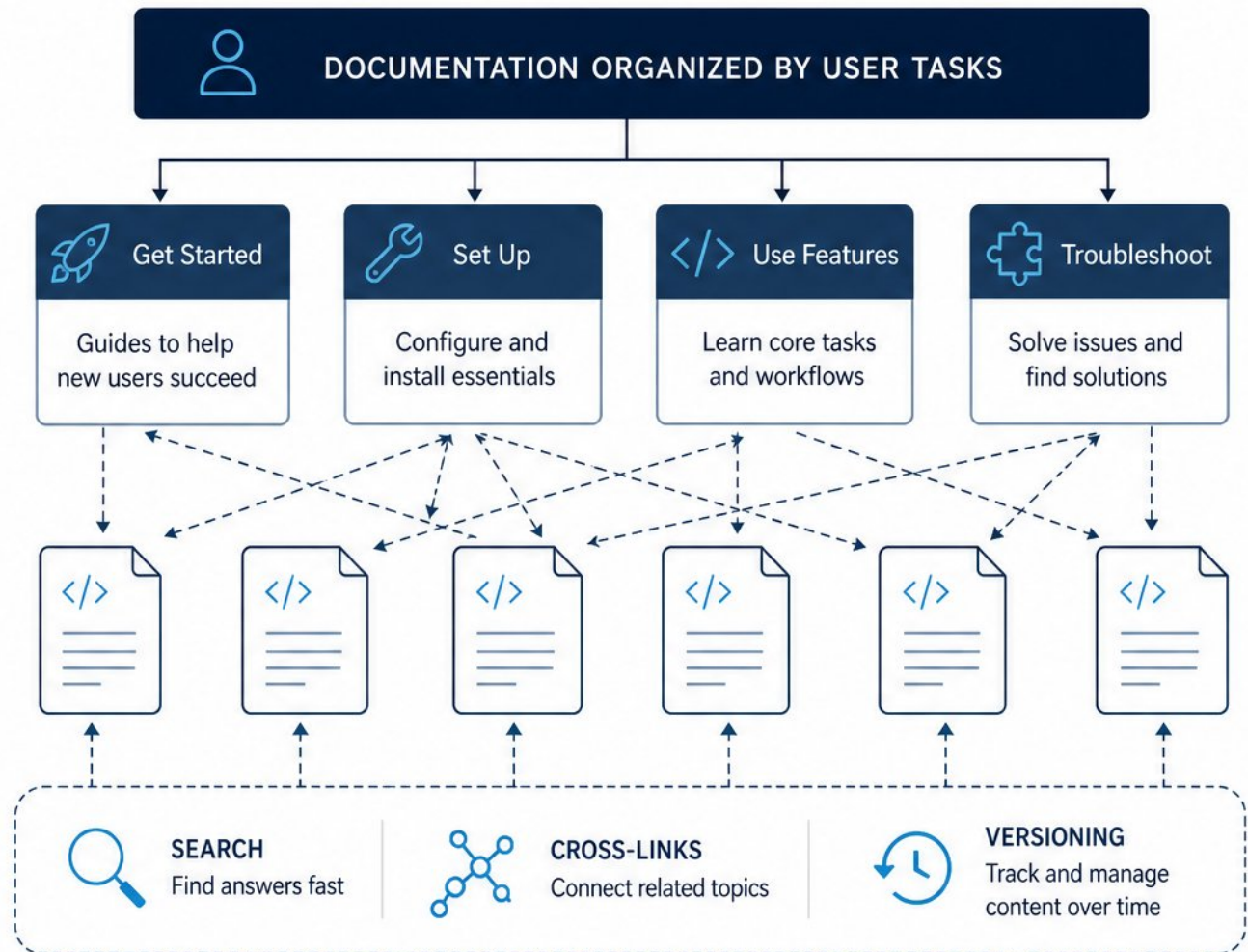


ERROR CASE

Realistic error response

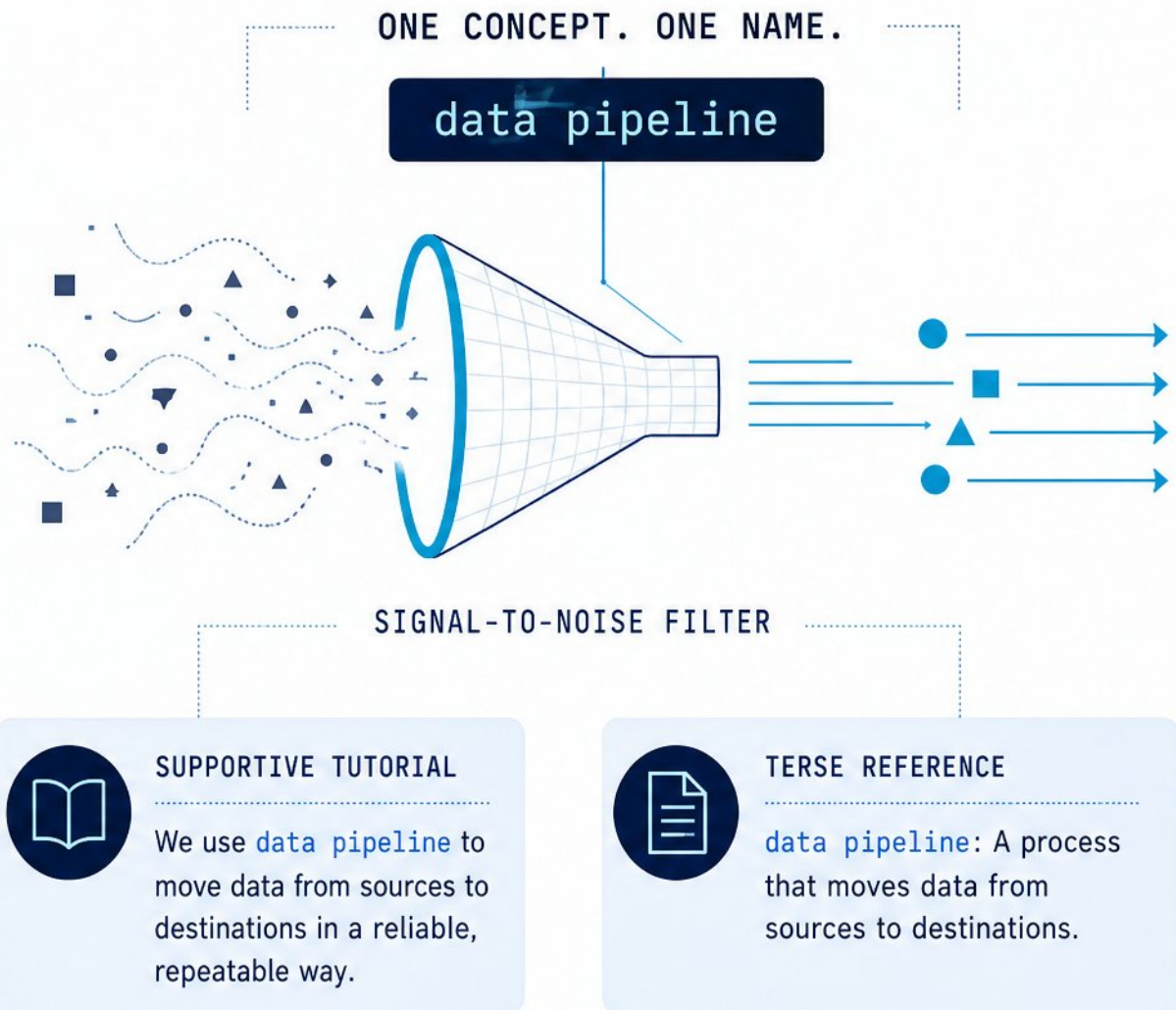
Structure, Navigation, and Information Architecture

- - Organize docs by user tasks, not internal architecture
- - Use descriptive headings and atomic pages for findability
- - Cross-links and consistent structure support skimming
- - Design navigation for new users, experts, and returning users
- - Versioning and search improve long-term maintainability



Style, Tone, and Terminology Consistency

- - Adopt a style guide: Google, Microsoft, or internal
- - Use plain language with active voice and direct verbs
- - Name concepts once, use consistently, avoid synonym swapping
- - Manage terms with glossaries and term bases
- - Match tone to page type: supportive tutorials, terse references



Reviewing, Testing, and Maintaining Documentation



- Review docs against set criteria: accuracy, clarity, completeness, and consistency



- Test commands and examples in CI; verify claims against actual code



- Use analytics, issue trackers, and stale-page detection to spot weak spots



- Keep docs current with releases, breaking changes, and deprecations

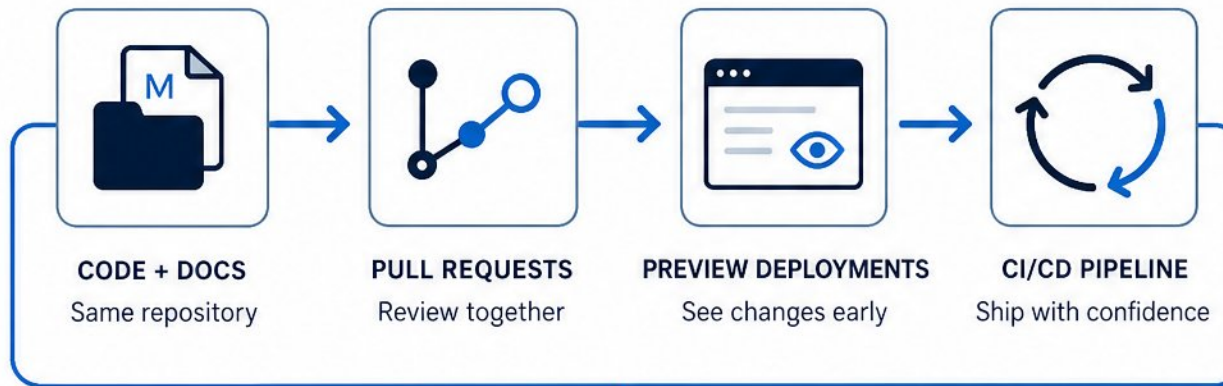


- Stale docs are worse than missing docs—they mislead with confidence

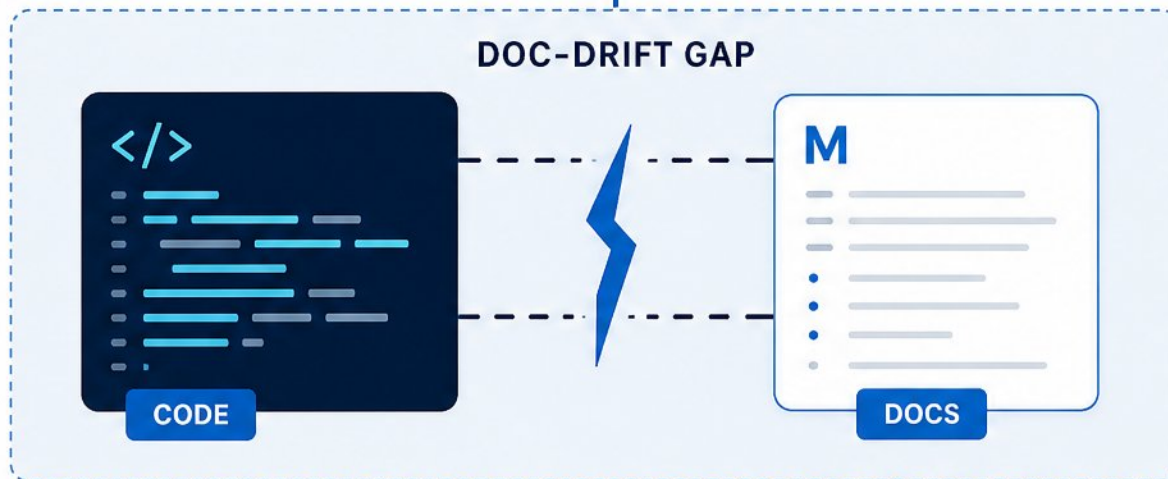
Docs as Code: Collaboration for Developers and Writers



ONE REPOSITORY. ONE WORKFLOW.



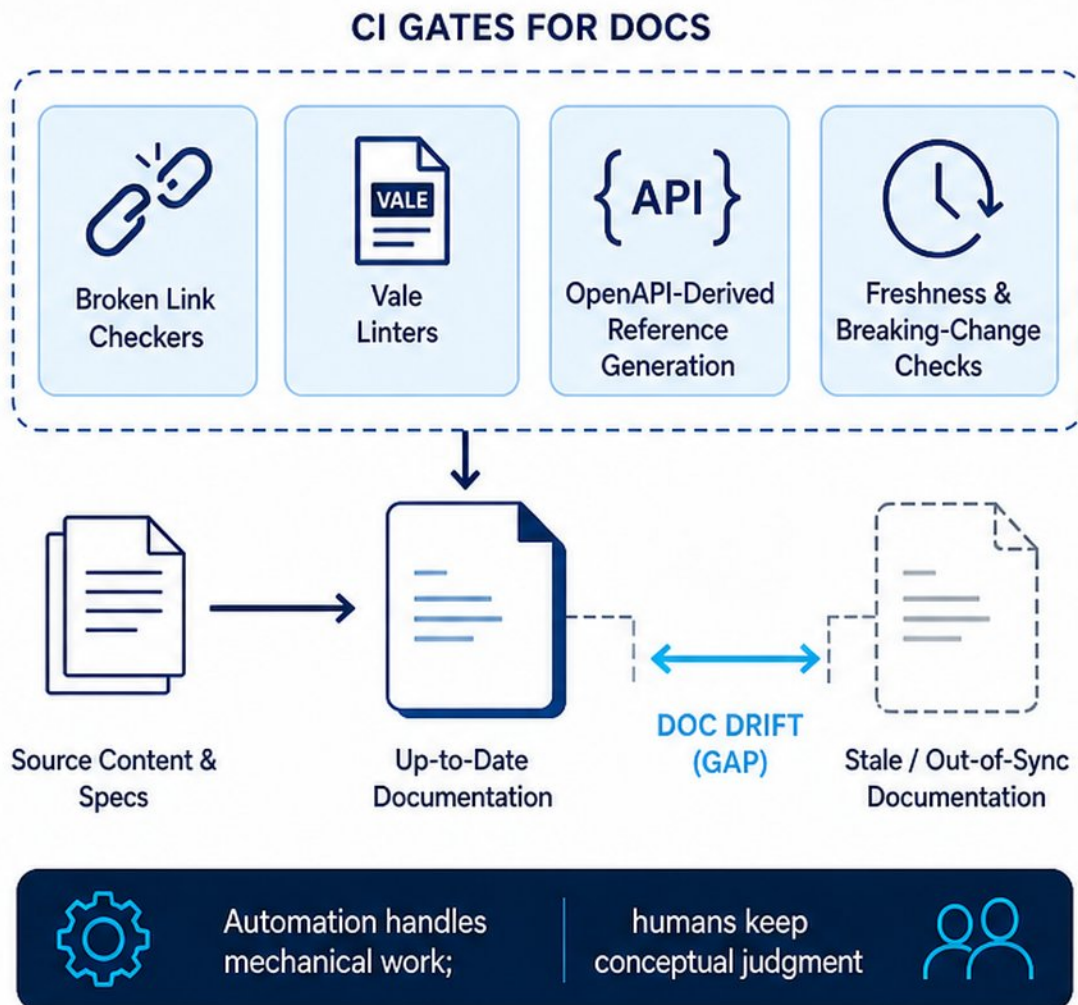
DOC-DRIFT GAP



- Documentation lives in the same repository, review process, and CI/CD pipeline as code
- Pull requests, Markdown, and preview deployments reduce friction without blocking releases
- Shared ownership: developers ensure technical accuracy; writers maintain clarity and structure
- Common bottlenecks: separate doc systems, unclear ownership, and updates that trail code changes
- Browser-based editing layers lower the barrier for non-engineering contributors while keeping Git as the source of truth

Automating Documentation Quality Checks

- CI gates catch broken links, style lapses, and spec drift before release
- Linters like Vale and markdownlint turn standards into enforceable rules
- Reference docs generated from OpenAPI specs eliminate manual sync
- Breaking-change and freshness checks turn staleness into concrete lists
- Automation handles mechanical work; humans keep conceptual judgment



Writing for Humans and AI Readers

- Docs now serve two audiences: developers in browsers and AI agents parsing structured formats
- Clean headings, complete examples, and precise terminology help both readers
- Structured formats like llms.txt and MCP servers improve machine readability
- AI-assisted updates must be verified against actual code to prevent confident fiction
- Generated content covers reference material; humans write concepts and guides

HUMAN READER

In Browser



Scans headings and structure



Reads examples and explanations



Builds understanding and implements

API Reference

Overview

Authentication

```
const token = await auth.login({
  apiKey: process.env.API_KEY
});
client.setToken(token);
```

List Resources

```
const resources = await client.list({
  limit: 50
});
return resources;
```

AI AGENT

Parsing Structured Formats



Extracts structured content



Maps relationships and endpoints



Uses precise data for reliable actions



One Source of Truth

Clear for humans. Structured for AI.

Applying the Course: A Real Documentation Fix

```
// docs should reflect code  
// not the other way around  
<doc-drift/>
```



- **Diagnose** poorly written pages: audience, structure, examples, terminology



- **Rewrite** with clear audience, task-oriented steps, and tested examples



- **Peer review** using a rubric: accuracy, clarity, completeness, consistency



- **Plan team next steps:** ownership, CI checks, maintenance cadence



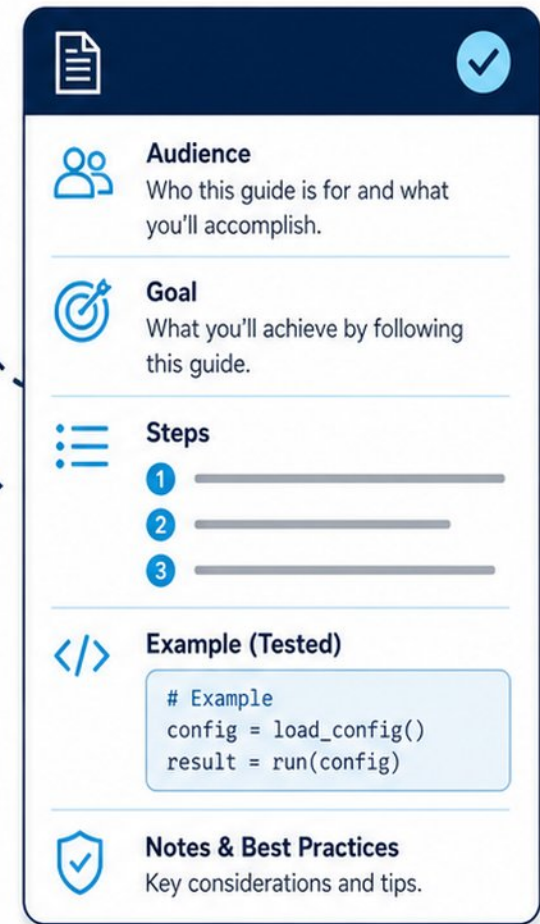
- **Before-and-after example** shows combined effect of all practices

BEFORE POORLY WRITTEN DOC



DOC-DRIFT
GAP

AFTER CLEAR, TASK-ORIENTED DOC



PersonWise.

PersonWise • AI digital-human course platform



Scan the code or click anywhere to watch this course live, with voice Q&A

personwise.ai/t/c9b5fca6/public