TELOSIS-RP-2026-008 · ENGINEERING PRACTICE · JULY 2026 · 10 SECTIONS, 8 MIN READ

# Documentation That Survives

TELOSIS Research

Most documentation is written for the present: for the current team, the current architecture, the current understanding of the system. When the team changes, the documentation becomes incomprehensible - or worse, misleading. This paper presents patterns for writing documentation that survives: documentation designed to be understood by someone who joins the project five years after the original authors have left. We define the documentation stack, identify what must be documented and what should not be, and present the documentation standards adopted across TELOSIS brands.

documentation | institutional memory | technical writing | knowledge management | software maintenance

Writing technical documentation that outlasts the original team. Documentation as institutional memory. Patterns for maintainable docs. What to document and what not to.

## 1. Introduction

Code is what the system does. Documentation is what the system was intended to do. When they diverge, the reader cannot know which is correct - the code or the documentation. In most cases, the code is the ground truth. The documentation is wrong.

This is not a failure of writing. It is a failure of design. Documentation that cannot be maintained will not be maintained. Documentation that is not maintained becomes harmful: it misleads the reader into believing something about the system that is no longer true.

Documentation that survives is documentation designed for maintenance. It is minimal. It is structured. It lives close to the code. It is tested. It is written for a reader who knows nothing about the system except what the documentation tells them.

## 2. The Documentation Stack

Documentation is not one thing. It is a stack of layers, each serving a different reader at a different moment.

### Layer 1: Overview

**Reader:** Someone who wants to know what the system is and whether it is relevant to them.

**Contents:** What the system does. Why it exists. What problem it solves. High-level architecture diagram. Link to getting started.

**Length:** One page.

**Rule:** If a new team member cannot understand the system's purpose after reading the overview, the overview has failed.

### Layer 2: Getting Started

**Reader:** Someone who wants to use the system for the first time.

**Contents:** Installation. Configuration. First run. Minimal working example. Common first errors and their solutions.

**Length:** As long as needed, but no longer. The goal is a working system, not comprehensive knowledge.

**Rule:** Getting started must be tested by someone who has never seen the system before. If they cannot complete it, it is wrong.

### Layer 3: Reference

**Reader:** Someone who is using the system and needs to look up a specific detail.

**Contents:** Every configuration option. Every API endpoint. Every parameter. Every error code. Every default value.

**Length:** Comprehensive. This is the only layer where completeness is the goal.

**Rule:** Reference documentation must be generated from the code where possible. Hand-written reference documentation is always out of date.

### Layer 4: Guides

**Reader:** Someone who knows the system and wants to accomplish a specific task.

**Contents:** Step-by-step instructions for common tasks. “How to deploy to production.” “How to configure single sign-on.” “How to migrate from version 2 to version 3.”

**Length:** One task per guide. Multiple guides.

**Rule:** Guides must be tested against the current version. A guide that references a deprecated feature is worse than no guide.

### Layer 5: Architecture

**Reader:** Someone who needs to understand why the system works the way it does. A new team member. A future maintainer. A researcher.

**Contents:** Architecture decisions. Trade-offs. Rejected alternatives. Data flow diagrams. Component relationships. Invariants.

**Length:** As long as needed. Architecture documentation is never skimmed. It is studied.

**Rule:** Architecture documentation must explain why, not just what. “We use PostgreSQL” is a fact. “We chose PostgreSQL because we needed strict schema enforcement and triggers for audit trails” is documentation.

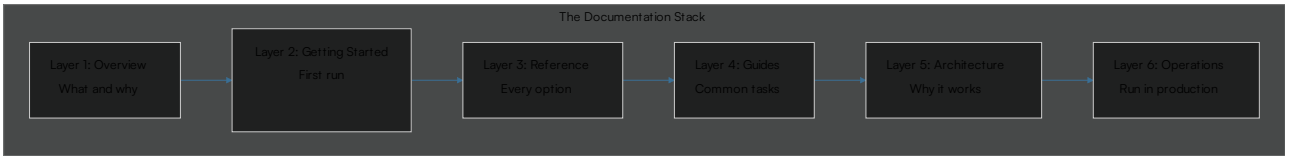
### Layer 6: Operations

**Reader:** Someone who is running the system in production.

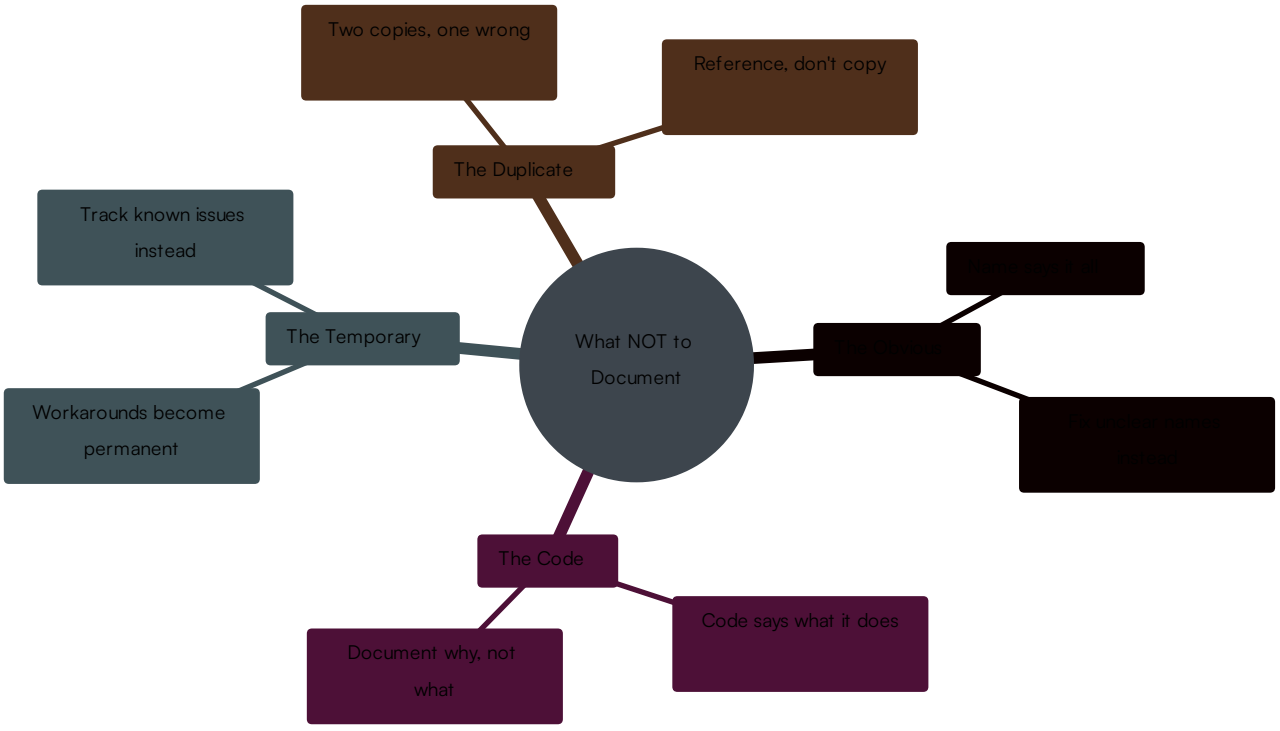
**Contents:** Deployment. Monitoring. Alerting. Backup and restore. Disaster recovery. Scaling. Known failure modes and their symptoms.

**Length:** Comprehensive. This is the layer that determines whether a production incident is resolved in minutes or hours.

**Rule:** Operations documentation must include troubleshooting guides. “If X happens, check Y. If Y is above threshold, do Z.”




## 3. What Not to Document




### 3.1 The Obvious

Do not document that a function called `calculateTotal` calculates a total. The name is the documentation. If the name is unclear, fix the name. Do not write a comment explaining it.

### 3.2 The Code

Do not describe what the code does in prose. The code already says what it does. Document why it does it, why it does it that way, and what else was tried.

### 3.3 The Temporary

Do not document workarounds, temporary fixes, or “we’ll fix this later” without a clear marker. Temporary documentation that becomes permanent is a lie. If a workaround is necessary, document it as a known issue with a tracking reference. If the issue is resolved, remove the documentation.

### 3.4 The Duplicate

Do not document the same thing in two places. Two copies means one will be wrong. Reference, do not duplicate.

## 4. Patterns for Maintainable Documentation

### Pattern 1: Documentation Lives With Code

Documentation that lives in a separate wiki, a separate repository, or a separate tool will diverge from the code. The friction of updating documentation in a different system is too high. Documentation must live in the same repository as the code it describes.

**Implementation:** Markdown files in a `/docs` directory. Versioned alongside the code. Reviewed in the same pull requests.

### Pattern 2: Documentation Is Reviewed

A code change that alters behavior must include documentation changes in the same pull request. A reviewer who approves a behavior change without a documentation change is approving technical debt.

**Implementation:** Pull request templates include a documentation checklist. CI can check that changed APIs have corresponding documentation changes (heuristic, not perfect).

### Pattern 3: Documentation Is Tested

Getting started guides and code examples in documentation must be tested. A script that fails because the documentation is wrong is a failing test.

**Implementation:** Extract code examples from documentation files. Run them in CI. If they fail, the documentation build fails.

### Pattern 4: Documentation Has a Single Owner

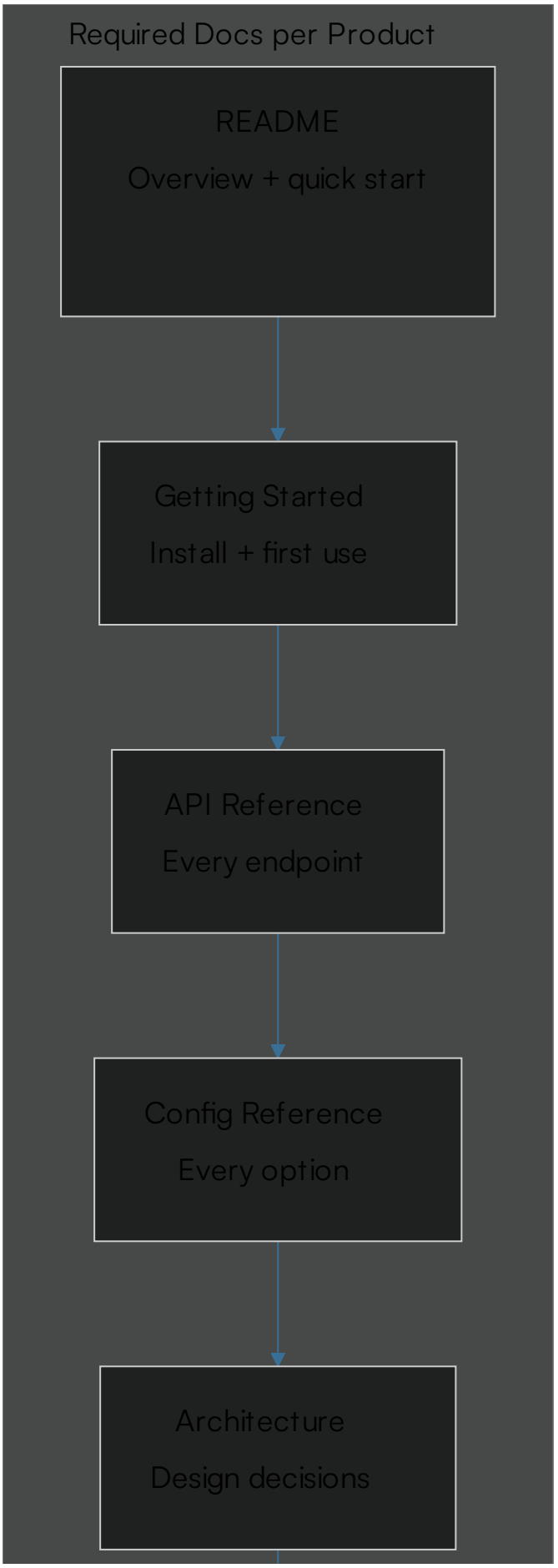
Collective ownership means no ownership. One person is responsible for documentation quality per product. They do not write all documentation. They ensure it exists, it is accurate, and it follows the standards.

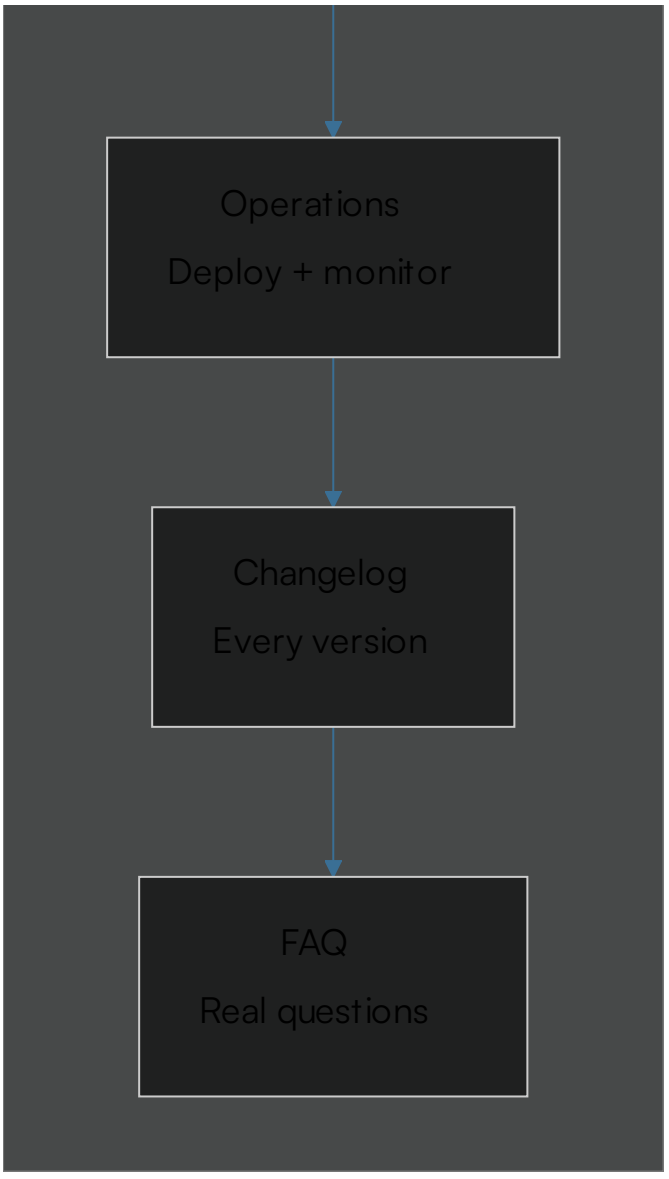
### Pattern 5: Documentation Decays Visibly

A documentation page that was last updated three years ago should warn the reader. “This page was last updated on 2023-06-15. It may not reflect the

current version." The reader should never trust outdated documentation by accident.

## 5. The TELOSIS Documentation Standard


Required Docs per Product
README
Overview + quick start
Getting Started
Install + first use
API Reference
Every endpoint
Config Reference
Every option
Architecture
Design decisions




Every product under TELOSIS must ship with documentation before it ships to users. Undocumented features are incomplete features.

## Required Documentation Per Product

| DOCUMENT | CONTENTS | UPDATE FREQUENCY |
|---|---|---|
| README | Overview, quick start, links | Every release |
| Getting Started | Installation, configuration, first use | Every release |
| API Reference | Every endpoint, parameter, response | Generated from code |

| DOCUMENT | CONTENTS | UPDATE FREQUENCY |
|---|---|---|
| Configuration Reference | Every option, default, effect | Every release |
| Architecture | Design decisions, trade-offs, diagrams | When architecture changes |
| Operations | Deployment, monitoring, backup, troubleshooting | Every release |
| Changelog | Every version, every change | Every release |
| FAQ | Honest answers to real user questions | As needed |

## 6. The Reader After You

The test of documentation is not whether it makes sense to the author. It is whether it makes sense to someone who:

- Joined the team six months after the author left
- Has never spoken to anyone who worked on the system
- Has only the documentation and the code

Write for that person. They are the reader that matters.

### The New Developer Test

Give a new team member the documentation. Ask them to deploy the system and make a small change. Observe. Do not answer questions. Note every point where they get stuck. Fix the documentation at those points. Repeat.

## 7. Documentation as Institutional Memory

Software institutions outlast individual memory. The engineer who designed the authentication system leaves. The team that built the original data pipeline moves on. The founder who made the early architectural decisions is no longer involved.

What remains is the code and the documentation.

If the documentation explains why decisions were made, future maintainers can evaluate whether those reasons still apply. They can change the system intelligently rather than cargo-culting decisions whose rationale has been lost.

If the documentation only describes what the system does, future maintainers must reverse-engineer the intent. They will make changes that violate original assumptions because they do not know those assumptions existed.

Documentation is not a courtesy to future developers. It is the mechanism by which an institution preserves its reasoning across generations of builders.

## 8. The Documentation Commitment

TELOSIS brands commit to:

1. Documentation shipped with every product, before it reaches users.
2. Documentation versioned alongside the code.
3. Documentation reviewed in the same pull requests as code changes.
4. Documentation tested where testable (code examples, getting started guides).

5. Documentation written for the reader who knows nothing except what the documentation tells them.

This is not a policy we will enforce someday. It is the standard we apply now.

## 9. Conclusion

Most documentation dies before the product does. It is written once, during the rush to launch, and never touched again. Within months, it is misleading. Within years, it is harmful.

Documentation that survives is designed to survive. It is minimal. It lives with the code. It is tested. It is owned. It decays visibly. It is written for a reader who has never met the author and never will.

The code says what the system does. The documentation says what it was meant to do and why. When both are maintained, future builders can understand the system, change it safely, and preserve what should be preserved.

That is institutional memory. That is documentation that survives.

## References


1. TELOSIS Research. *Exportability as a Structural Property.* TELOSIS-RP-2026-007, 2026.
2. CODECX Engineering. *The Architecture of Trust: How Covenant Structures Professional Commitments.* CODECX Journal, 2026.

3. Aplin, J. *What Nobody Tells You About Documentation.* 2017.

4. Holscher, C. *Write the Docs Guide.* 2024.

---

## RELATED

TELOSIS-RP-2026-007: Exportability as a Structural Property

## CITATION

APA MLA BIBTEX

Copy

```
TELOSIS Research. (2026). Documentation That Survives. TELOSIS-RP-2026-008.
```

---

← RP-2026-009: The Economics of Self-Hosting

RP-2026-007: Exportability as a Structural Property →