RTMX 101

Introduction to Requirements Traceability with RTMX

Author
RTMX Engineering

Version
1.0

Date
February 10, 2026

rtmx.ai • dev@rtmx.ai

Introduction

Modern software development moves fast. Teams ship features daily, AI agents write code alongside humans, and requirements live in scattered documents, tickets, and tribal knowledge. In this chaos, a fundamental question often goes unanswered:

> **NOTE**
>
> *"Does this code actually meet our requirements?"*

RTMX answers this question with **closed-loop verification**: a system where requirement status is derived from evidence, not opinions.

The Problem

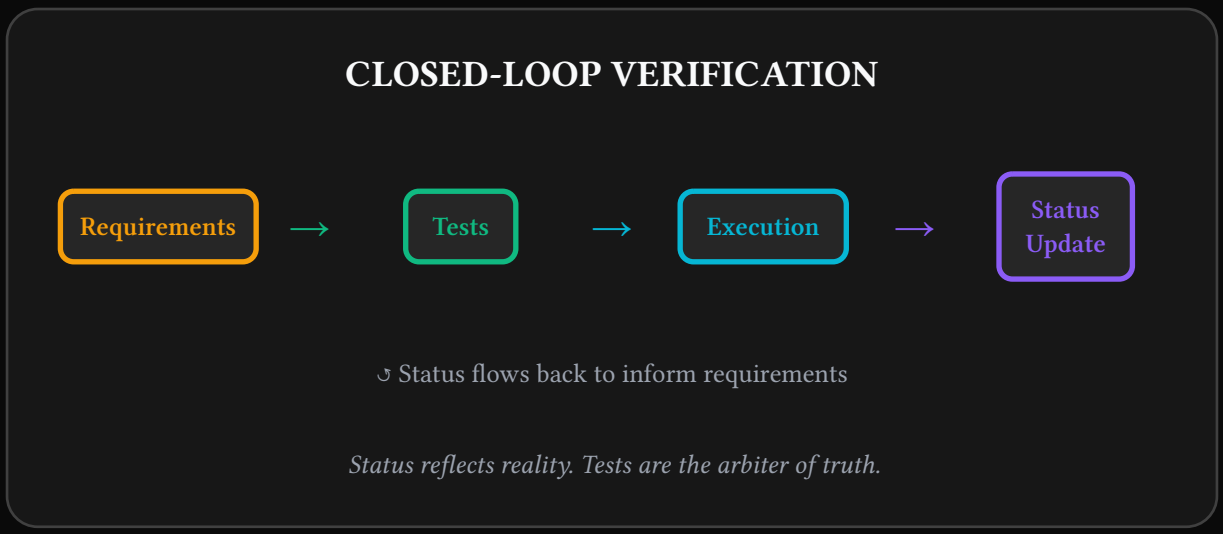
Traditional requirements management suffers from three critical failures:

1. **Manual Status Updates** — Someone claims a requirement is "done" without proof
2. **Disconnected Tests** — Tests exist, but nobody knows which requirements they verify
3. **Stale Documentation** — Requirements documents drift from reality within weeks

These failures compound. When you can't trust your requirements status, you can't trust your release readiness. When tests aren't linked to requirements, passing tests don't prove anything about feature completeness.

The RTMX Solution

RTMX takes a radically simple approach:



Tests are the arbiter of truth. If tests pass, the requirement is complete. If tests fail, it isn't. No exceptions, no overrides, no "trust me, it works."

Core Concepts

The RTM Database

RTMX stores requirements in a CSV file—human-readable, Git-friendly, and AI-parseable:

```csv
req_id,category,requirement_text,status,test_module,test_function
REQ-AUTH-001,AUTH,User can log in,COMPLETE,tests/test_auth.py,test_login
REQ-AUTH-002,AUTH,User can reset password,MISSING,,,
```

Why CSV? Because:

- Every tool can read it (Excel, pandas, grep)
- Git diffs show exactly what changed
- AI agents can parse and update it
- No database server required

Test Markers

Tests link to requirements using pytest markers:

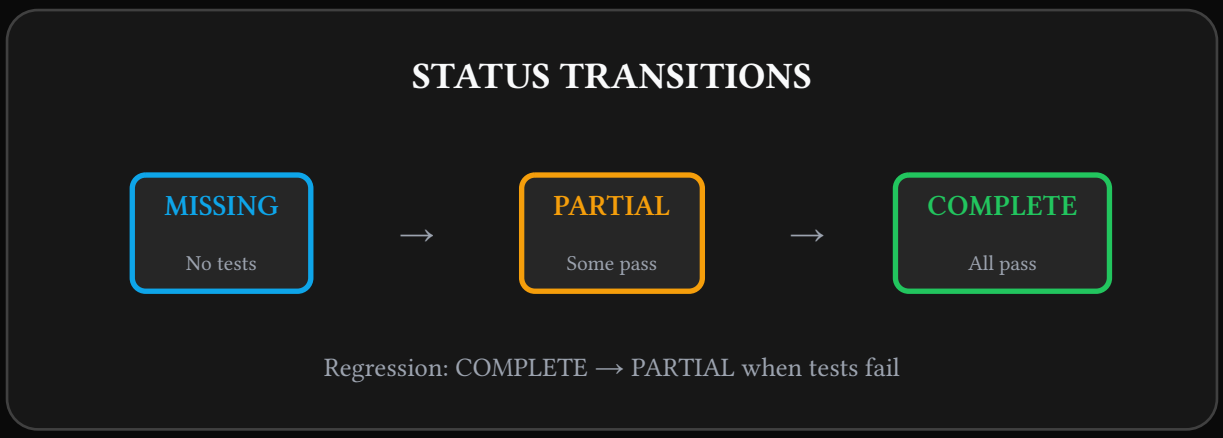
```python
@pytest.mark.req("REQ-AUTH-001")
@pytest.mark.scope_unit
def test_user_can_login():
    user = create_test_user()
    result = login(user.email, user.password)
    assert result.success
```

The `@pytest.mark.req()` decorator creates bidirectional traceability:

- From requirement → find its tests
- From test → find its requirement

Status Lifecycle

Requirements flow through four statuses:



Status	Meaning	Source
`NOT_STARTED`	Work hasn't begun	Initial state
`MISSING`	Tests don't exist or haven't run	No evidence

PARTIAL	Some tests pass, some fail	Mixed results
COMPLETE	All tests pass	Verified

> **KEY INSIGHT**
>
> Status transitions are **automatic**. When `rtmx verify --update` runs, status changes based on test results—not human claims.

Quick Start

Installation

```bash
pip install rtmx
```

Initialize Your Project

```bash
cd your-project
rtmx init
```

This creates:

- `rtmx.yaml` — Configuration file
- `docs/rtm_database.csv` — Requirements database
- `docs/requirements/` — Specification files

Check Status

```bash
rtmx status
```

Output shows completion percentage by category:

```text
═══════════════════ RTM Status Check ═══════════════════

Requirements: [████████████▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒]  45.2%

✓ 14 complete  ⚠ 2 partial  ✗ 15 missing
```

Run Verification

This is the core command—the closed loop:

```bash
rtmx verify --update
```

What happens:
1. Runs all tests with `@pytest.mark.req()` markers
2. Maps test results to requirements
3. Updates status in `rtm_database.csv`
4. Reports changes

The Closed Loop

The most important concept in RTMX is **closed-loop verification**. Let's contrast it with the anti-pattern:

PATTERN

Run `rtmx verify --update`
Tests determine status
Evidence-based progress

ANTI-PATTERN

Edit CSV: `status: COMPLETE`
Human claims status
Opinion-based progress

Why Closed-Loop Matters

When status is derived from tests:

- **Releases are trustworthy** — 100% means all tests pass
- **Regressions are detected** — Failed tests downgrade status
- **AI agents can't lie** — They can't claim completion without passing tests
- **Progress is auditable** — Git history shows who changed what and when

Example: Catching a Regression

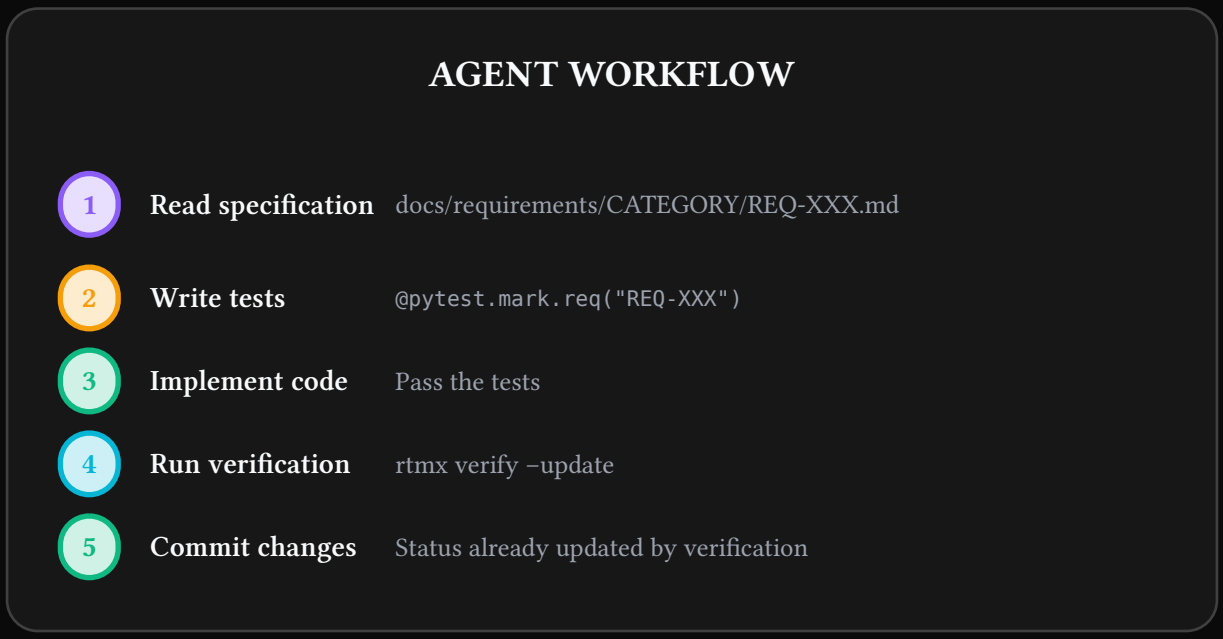
```text
$ rtmx verify --update

Verification Results:
──────────────────────────────────────────────────
  PASSING: 12 requirements
  FAILING: 1 requirements

Status changes:
  REQ-AUTH-001: COMPLETE → PARTIAL

✓ Updated 1 requirement(s) in RTM database
```

The system caught that `REQ-AUTH-001` regressed. No human noticed—the tests did.

Working with AI Agents

RTMX is designed for AI-assisted development. When AI agents work in RTMX-enabled projects:

Agent Workflow



Critical Rule

> **NEVER MANUALLY EDIT STATUS**
>
> Agents must **never** manually edit the `status` field in `rtm_database.csv`. Status is determined by `rtmx verify --update`, not by claims.

This rule is enforced by convention and can be validated in CI:

```yaml
# .github/workflows/ci.yml
- name: Verify Requirements
  run: |
    rtmx verify --update
    git diff --exit-code docs/rtm_database.csv
```

If an agent manually edited status, the diff would fail.

Next Steps

Now that you understand the basics:

1. **Read the Patterns Guide**
   Learn best practices and anti-patterns at rtmx.ai/patterns
2. **Add Requirements to Your Project**
   Start with 3-5 critical requirements and expand
3. **Integrate with CI**
   Run `rtmx verify` on every pull request

4. **Install Agent Prompts**
   Run `rtmx install` to inject RTMX guidance into CLAUDE.md

Summary

Concept	Key Point
RTM Database	CSV file, Git-tracked, AI-readable
Test Markers	`@pytest.mark.req()` links tests to requirements
Status	Derived from tests, never manually set
Closed Loop	`rtmx verify --update` is the source of truth
AI Agents	Implement code, run verify, never claim status

> **REMEMBER**
>
> The RTM is a **verification record**, not a wish list.
> Status must be **earned** through passing tests, not **claimed** through manual edits.

For more information, visit **rtmx.ai**
Questions? Contact **dev@rtmx.ai**