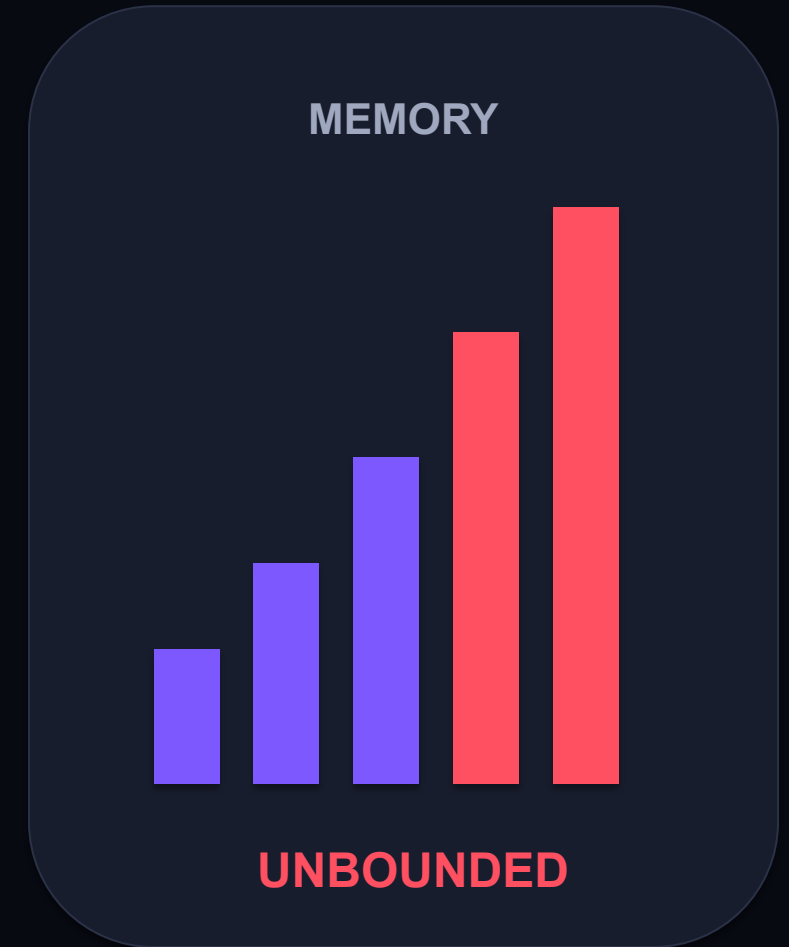


CELERY

A memory leak

users found first.

The defect was reported in 2018.
The fix arrived in 2026.



June 2018: a user reports real impact

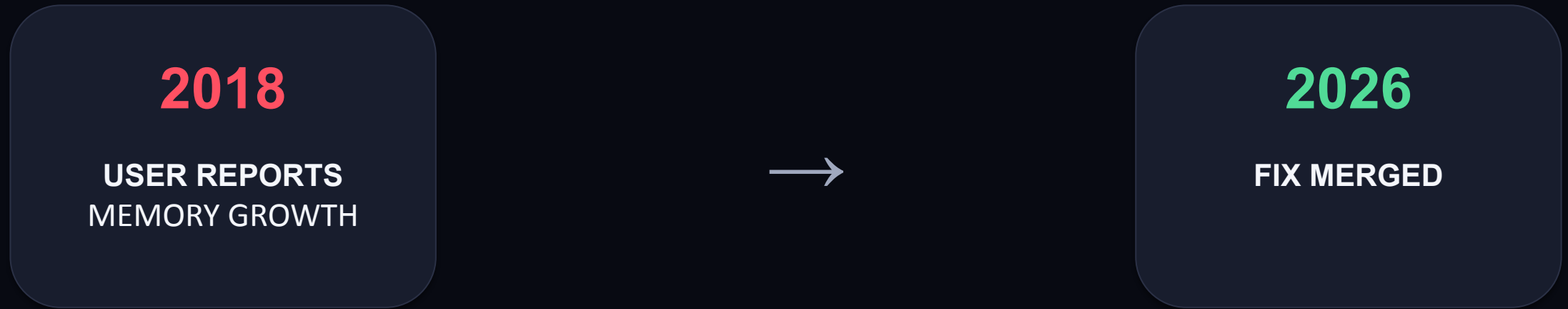
“We've noticed that this program
unboundedly leaks memory”

Celery 4.2.0 • real-time event monitoring • issue #4832

OPENED JUNE 21, 2018

The reporter had already traced the growth toward Celery's event-state bookkeeping.

The bug sat in released code



More than eight years between report and merged fix

What was Celery doing?



Long-lived in-memory state makes retention mistakes operationally expensive.

The failure mechanism — in one slide



No guaranteed cleanup + no hard bound = memory can grow without bound

The user's diagnosis was pointing at the right place

Expected

pending entries eventually get
popped



Observed

`_tasks_to_resolve` grows
unboundedly

“Or is this actually a problem in the celery code...?”

2026: the defect is finally confirmed

PR #10534

“Bound State._tasks_to_resolve
to fix unbounded memory growth”

The PR says unresolved-parent entries can stay forever, so a long-running monitor leaks memory without bound.

MERGED AUG 31, 2026

The human-built fix was tiny

BEFORE

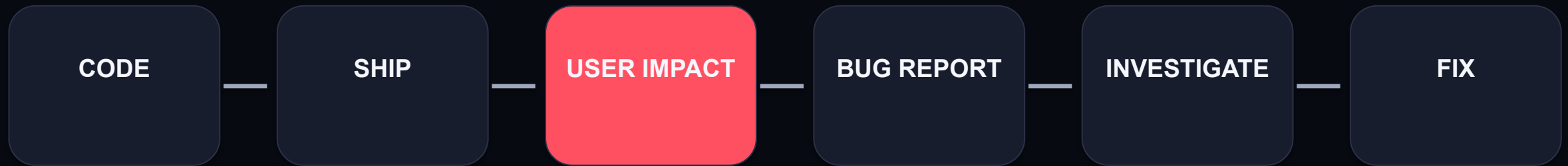
```
self._tasks_to_resolve = {}
```

AFTER

```
self._tasks_to_resolve =  
LRUCache(max_tasks_in_memory)
```

The important fix wasn't “use an LRU.” It was: this long-lived state must be bounded.

The traditional reliability feedback loop



Production became the place where the missing reliability invariant was discovered.

So we rewound Celery to before the fix

Historical pre-fix commit

fd a3c5b07

VULNERABLE HISTORICAL SOURCE

celery/events/state.py

730 lines • ~28 KB • 1 code file

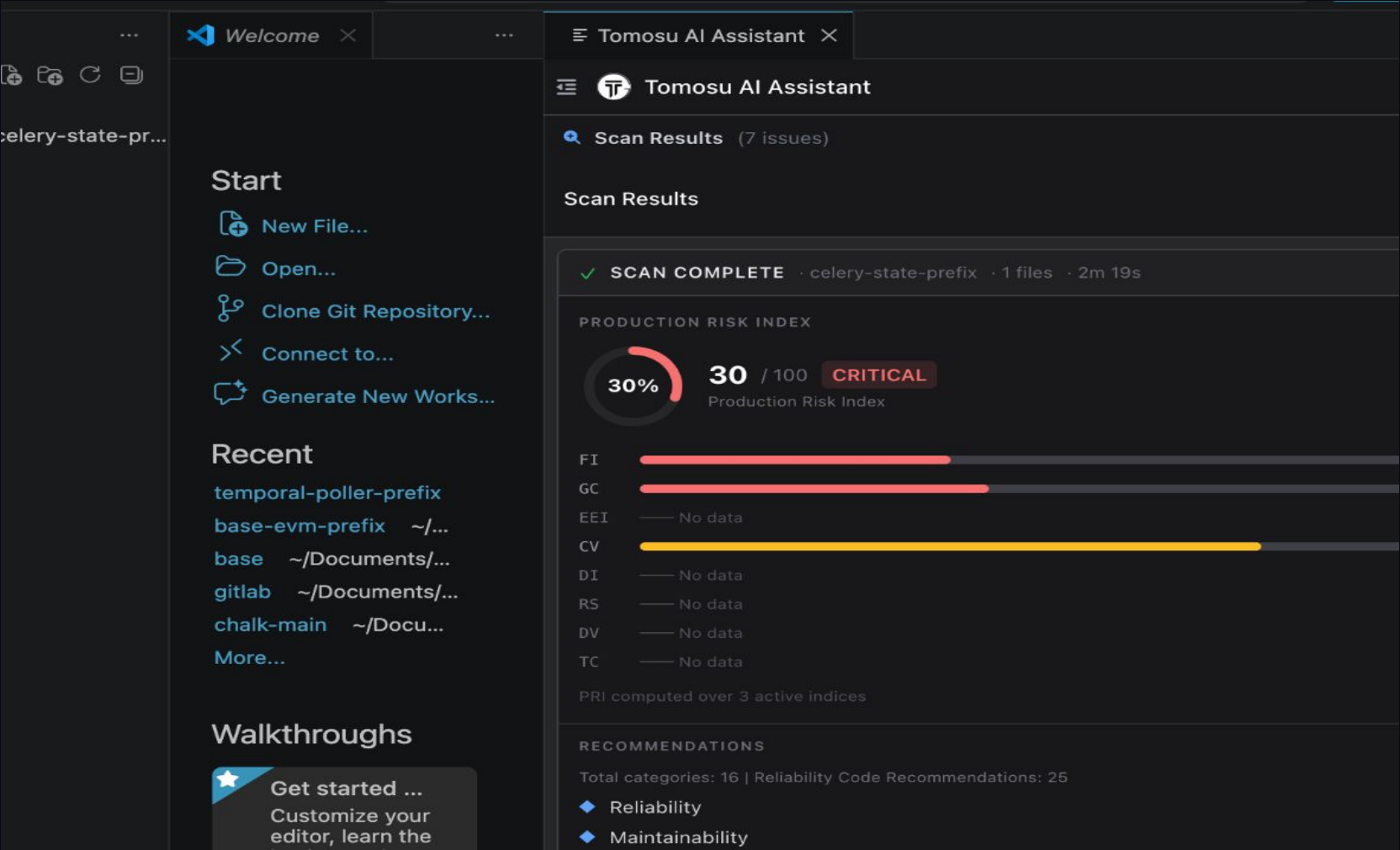
**We recreated the code as it existed
immediately before the substantive fix.**

And we removed the answer key

- ✗ NO BUG REPORT
- ✗ NO PR DESCRIPTION
- ✗ NO PATCH
- ✗ NO REGRESSION TEST
- ✗ NO MEMORY-LEAK HINT

Tomosu saw the vulnerable source — not our diagnosis.

The actual Tomosu scan



BLIND SCAN

1 file

7 issues

PRI 30 / 100

CRITICAL

**Historical
vulnerable state.py**

Tomosu found the exact failure mode

ERROR • RELIABILITY

In Memory Cluster State Scalability

“events referencing never-seen parents
can accumulate indefinitely.”

Tomosu recommended bounding the pending parent-child resolution index using `max_tasks_in_memory`.

It wasn't guessing “memory leak”



That is the causal chain the eventual fix addresses.

Tomosu vs. the eventual Celery fix

TOMOSU • BLIND

Bound pending

parent-child state

`using`

`max_tasks_in_memory`

CELERY • EVENTUAL FIX

Replace plain dict

with bounded LRU

`LRUCache (`

`max_tasks_in_memory)`

Same reliability invariant. Different implementation.

What we can actually claim

Tomosu identified this specific reliability defect

in Celery's historical vulnerable code

without being given the bug report or eventual fix.

That's stronger than saying Tomosu “might have helped.”

Move discovery left of user impact



The goal isn't perfect prediction. It's finding more reliability defects before customers become the detector.

The engineering lesson is bigger than Celery

Every long-lived collection needs an answer to four questions:

**WHAT
INSERTS?**

entry creation

**WHAT
REMOVES?**

cleanup path

**IS CLEANUP
GUARANTEED?**

failure path

**WHAT'S THE
HARD BOUND?**

retention policy

If the answer to the last two is “no” and “none,” production owns the risk.

THE POINT

A user found it in 2018.

Tomosu found it in the code.

THEN

Ship → **user impact**
→ report → fix

NOW

Historical code → **scan**
→ direct finding

That's what moving production reliability earlier looks like.



Find reliability problems

before your users do.

Production Reliability Governance
across code, deployment, and runtime.

tomосу.ai

CELERY #4832 • DIRECT HIT