OAK RIDGE National Laboratory | LEADERSHIP COMPUTING FACILITY

Oct. 15, 2025 | HPC Best Practices Webinar

Sustainable HPC Software: A Maintainer's Perspective

Damien Lebrun-Grandié

Computational Sciences and Engineering Division

U.S. DEPARTMENT of ENERGY

ORNL IS MANAGED BY UT-BATTELLE LLC
FOR THE US DEPARTMENT OF ENERGY

Clarifying the Focus: What is “Software Sustainability”?

Working Definition:
The ability for a software project to **remain useful and relevant** for **at least twice its currently projected lifespan**.

Focus: Longevity of the **codebase and community**.
NOT primarily about reducing power consumption (which is Environmental HPC Sustainability).

The Three Pillars of Technical Longevity:

- **Adaptability:** The ability to be ported to new architectures and adapt to new standards/toolchains
- **Maintainability:** Ease of fixing bugs and adding features without breaking existing functionality
- **Resilience:** The health and size of the contributor community

What Does A Maintainer Do?

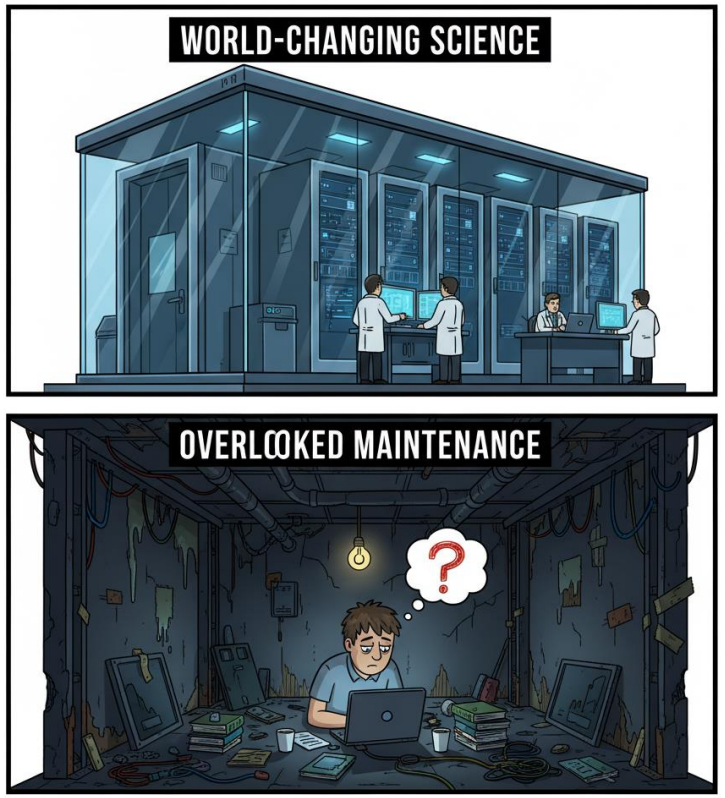
- Loosely aware of the entire project
- Track ongoing work and make sure that it gets reviewed and merged in a timely manner
- Direct the orchestra of **developers** and **reviewers**
- Has final responsibility
    - Reviews when no reviewer can be found for an important contribution
    - Develops when no developer can be found to fix an important bug

If something goes wrong, it's eventually the maintainer's fault

The HPC Software Paradox

The Problem: We rely on complex, foundational HPC software for world-changing science, but the maintenance often lacks resources and spotlight.

The Ask: What does it take to sustain a foundational project?



Talk Roadmap

Bus Factor

Technical Debt

Hyrum's Law

My Perspective: The Kokkos Lens

Kokkos in a few numbers:
14 years old project
250k LoC
162 contributors
30+ active developers from 8 institutions
50% ECP C++ software technologies and applications
2.4k users registered on Slack
2.3k stars on GitHub
1 citation per day

Stewarding the scientific computing software ecosystem presents unique challenges.

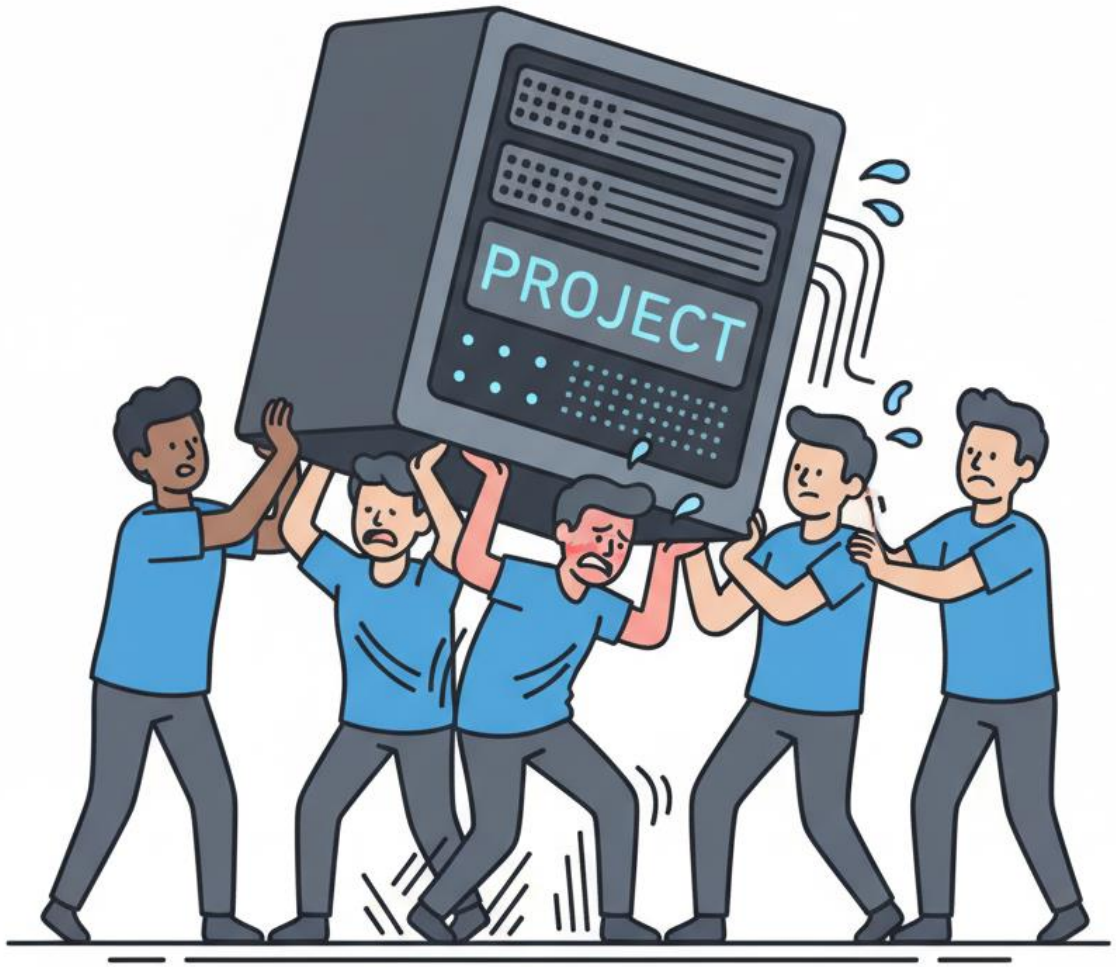
I'll use examples from my experience as Kokkos maintainer to explore these challenges.

My journey:
User -> Contributor -> Developer -> Maintainer/Lead

Kokkos' reach necessitates careful maintenance.
Carelessness: not catastrophic, but costly.

OAK RIDGE National Laboratory | LEADERSHIP COMPUTING FACILITY

Bus Factor: How Vulnerable is Your Project?



Generated by Gemini

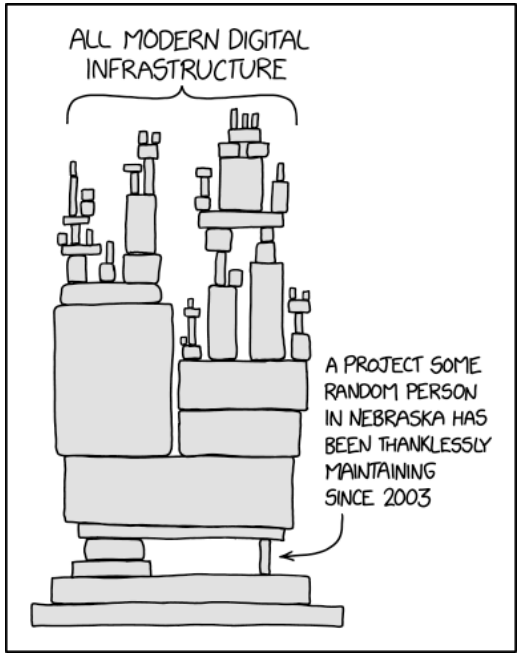
U.S. DEPARTMENT of ENERGY

ORNL IS MANAGED BY UT-BATTELLE LLC
FOR THE US DEPARTMENT OF ENERGY

Managing Critical Vulnerabilities: The Bus Factor

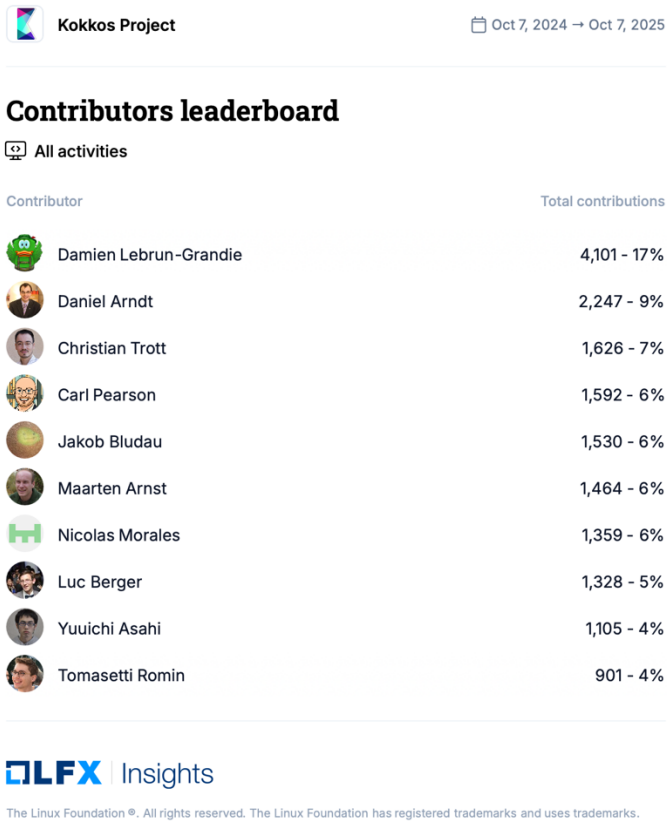
Definition: The risk that a single person (or small group) leaving the project (“hit by a bus”) would halt or severely impair development.

It's not just a joke; it's a structural flaw.



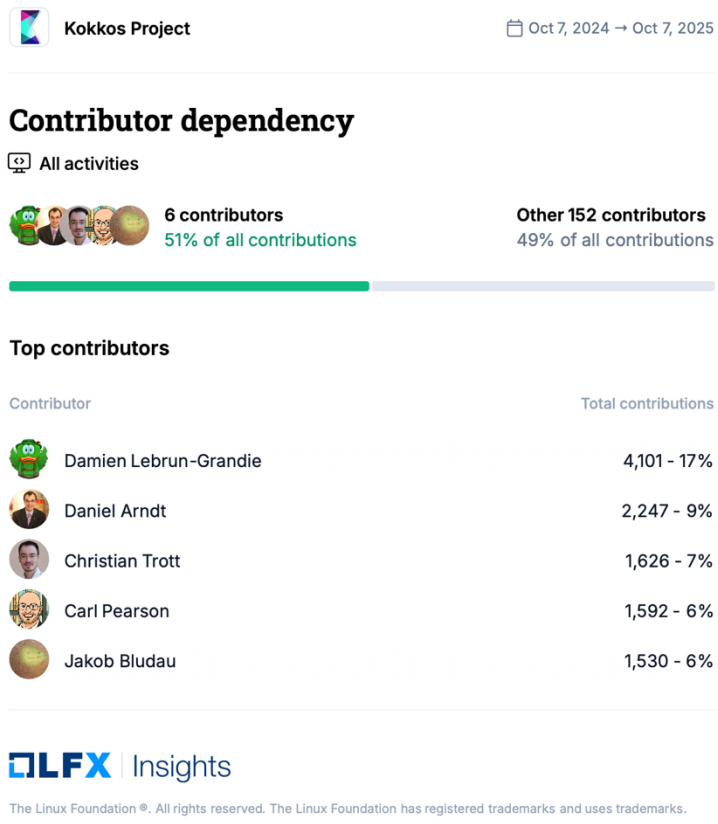
https://xkcd.com/2347/

Analyze Your Contributor Dependency

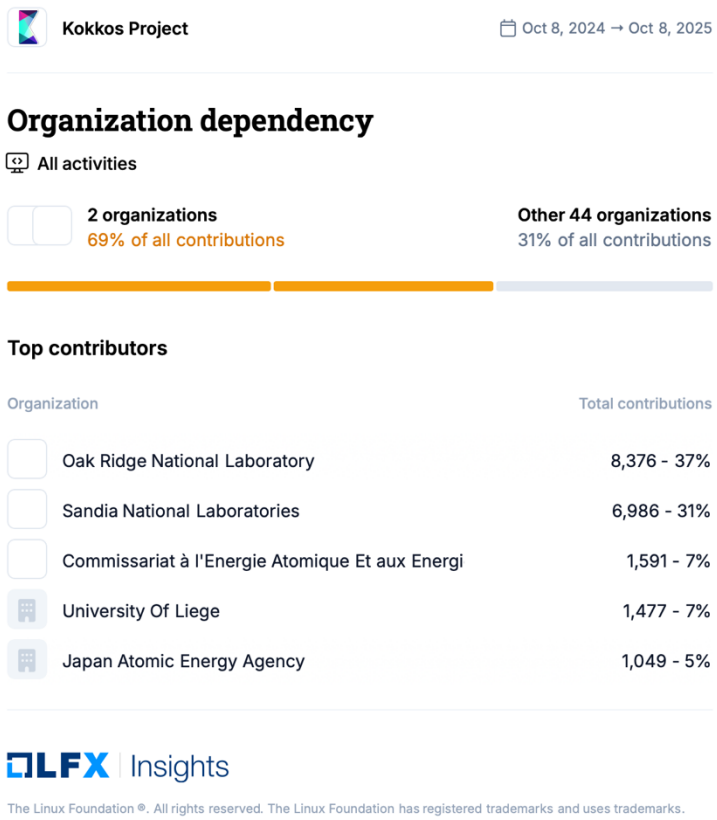


What it is: Measures how much the project depends on its most active individual contributors.

Why it matters: Highlights risk areas if key contributors leave. Useful for sustainability planning and succession strategy.



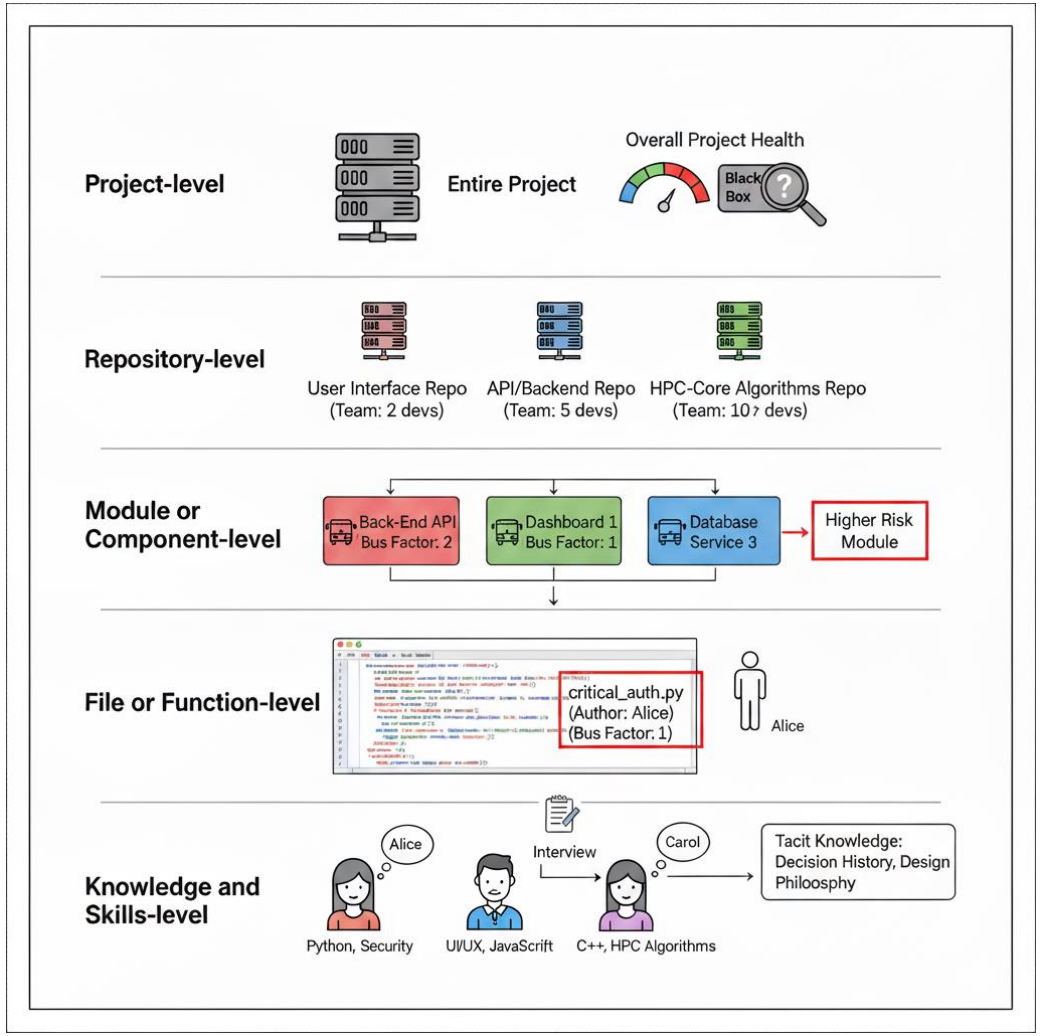
Elephant Factor



What it is: Analyzes how much of the project's contributions come from a small number of organizations.

Why it matters: Reveals potential over-reliance on single entities. Projects with diverse organizational support are generally more robust.

Granularity in Bus Factor Analysis



Bus Factor: Lessons from Kokkos

1. The Foundational Risk: Original Team Vulnerability

The Threat: Lost 2 of 3 original developers over 14 years.
Current Reality: Core resilience still relies heavily on co-lead.
Lesson: Longevity demands proactive knowledge decentralization.

2. The Stress Test: The "Quadruple Hit" Exodus

The Crisis: Lost 4 main developers back-to-back (Google/AMD).
Impact:
- Build system overhaul (departing developer)
- High-visibility subproject with no testing
- Loss of the designated build system backup

Lessons: Multiple single points of failure create systemic risk.

3. Mitigation: Decentralizing Knowledge & Infrastructure

Auto-Tuning Subproject Success:
- Initial risk from maintainer departure (KokkosTools).
- **Mitigation:** Design documentation and external consultant enabled successful handover and blog-post-worthy advancement.
- **Insight:** Process & shared design prevent collapse.

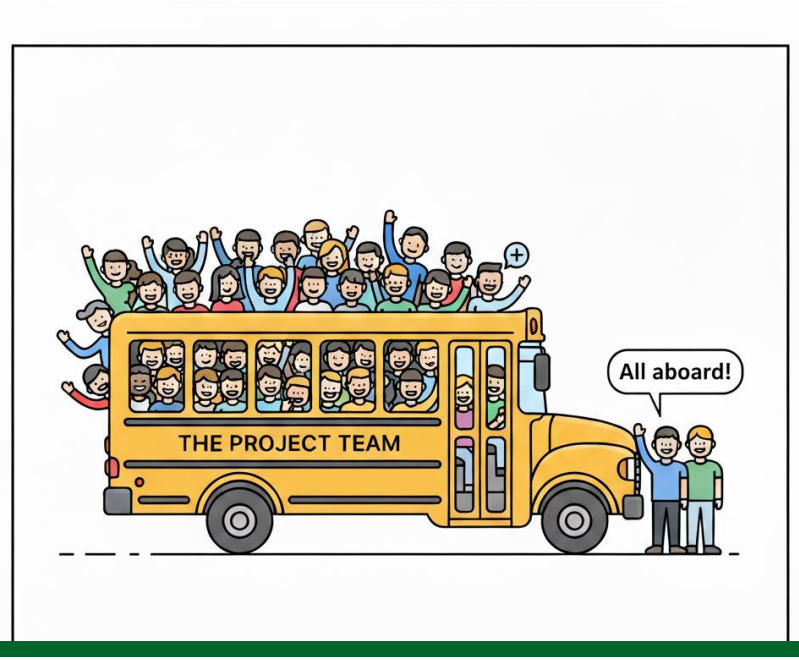
Testing Infrastructure Resilience:
- **Problem:** Single person handled all testing.
- **Solution:** Diversified testing load across partner organizations (on-prem, open-source, HPSF).
- **Impact:** Engaged more developers in maintenance; mitigated site failures.
- **Insight:** Redundancy in both people and systems.

Mitigation Strategy: Documentation & Process

Actionable Strategy 1: Mandatory, reviewed documentation for all new features.

Actionable Strategy 2: Enforced code review across different teams/individuals.

Actionable Strategy 3: Create a clear path for new contributors to become maintainers (i.e., **growing the bus**).

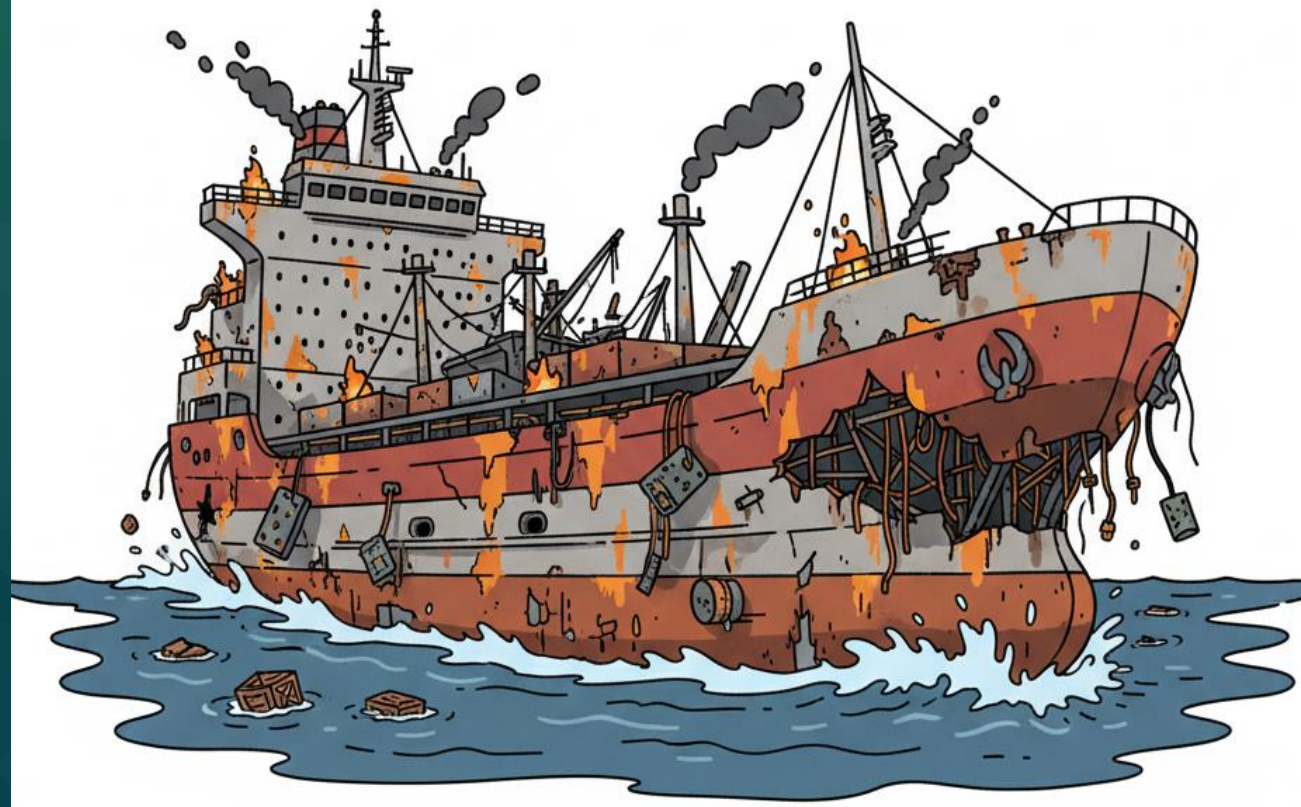


Key Takeaway: Growing the Bus

The goal isn't just to survive; it's to decentralize knowledge and grow the contributor pool.

OAK RIDGE National Laboratory | LEADERSHIP COMPUTING FACILITY

The Silent Drag: Technical Debt



Generated by Gemini

U.S. DEPARTMENT of ENERGY

ORNL IS MANAGED BY UT-BATTELLE LLC
FOR THE US DEPARTMENT OF ENERGY

The Silent Drag: Technical Debt

Definition: Code or architectural choices made quickly today that will slow down development tomorrow.
The Issue: The debt interest (slowed velocity, more bugs) is paid in every subsequent development cycle.

What is it? "Quick fixes" create future rework.
Performance, scalability & maintainability suffer.

Sources: Deadlines, legacy code, evolving hardware, lack of refactoring.

Impact: Slows development, increases bugs, hinders innovation, burns out maintainers.

Maintainer's Reality: Constant patching, frustration, struggling to keep up.

Solution: Prioritize refactoring, testing, documentation, and code reviews.

Technical Debt: Lessons from Kokkos

1. The Architectural Blocker: Exascale Backend Crisis

Mission: Develop new backends (HIP/SYCL) for Frontier & Aurora.
Debt: Rigid architecture with centralized preprocessor directives.
Impact: Blocked incremental development; new backends were untestable until 70% complete.

2. The Fix: Refactoring for Adaptability

Solution: Converted backend "choke points" into a modular plugin system.
Enablement: Developed new incremental tests to guide and validate backend development.
Lesson: Proactive refactoring is an investment in future adaptability.

3. The Human Cost: The "Half-Baked Feature" Trap

Problem: Experimental features pushed out for a publication, then abandoned.
Impact: Maintainers are left to salvage/fix the code with no recognition.
Lesson: Short-term incentives (publications) can create long-term, uncredited maintenance burdens.

4. The Strategic Win: Offloading Debt via Standardization

Strategy: Pushed `Kokkos::View` abstraction to the C++ standard, resulting in `std::mdspan` (C++23).
Benefit: Shared the maintenance cost with the entire C++ community.
Action: Back-ported `mdspan` to C++17 and refactored `Kokkos::View` to use it.
Lesson: The ultimate debt reduction is making your problem a shared, community-supported solution.

Mitigation Strategy: Refactoring as a Feature

Actionable Strategy 1: Treat refactoring as a core feature for every release cycle (e.g., dedicate 10% of effort).

Actionable Strategy 2: Invest in robust, layered testing to ensure refactoring doesn't break performance.

Actionable Strategy 3: Adopt modern C++ standards to shed old, custom workarounds.

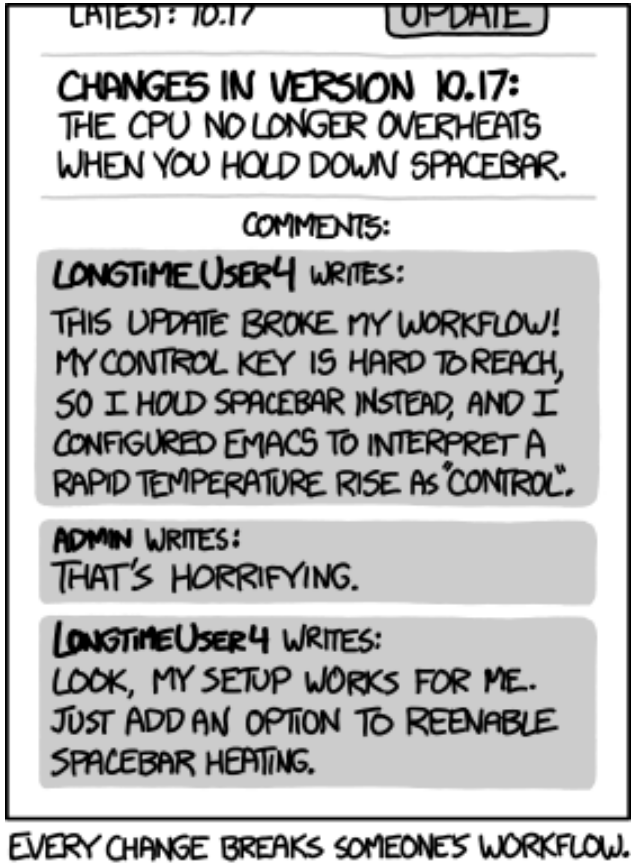
Key Takeaway: Pay Down the Principal

Don't wait for a crisis to fix core issues. **Proactive maintenance is a feature.**

Technical debt is not always avoidable, but it must be managed.
It's a hidden cost that significantly impacts long-term sustainability.

OAK RIDGE National Laboratory | LEADERSHIP COMPUTING FACILITY

Hyrum's Law: Implicit Dependencies Bite

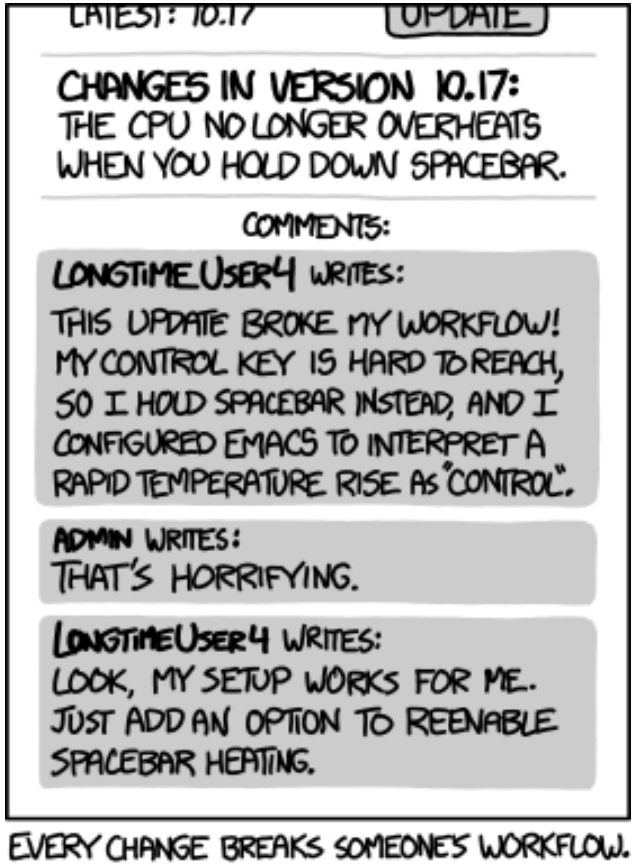


https://xkcd.com/1172/

Hyrum's Law: Implicit Dependencies Bite

"With a sufficient number of users of an API, it does not matter what you promise in the contract:

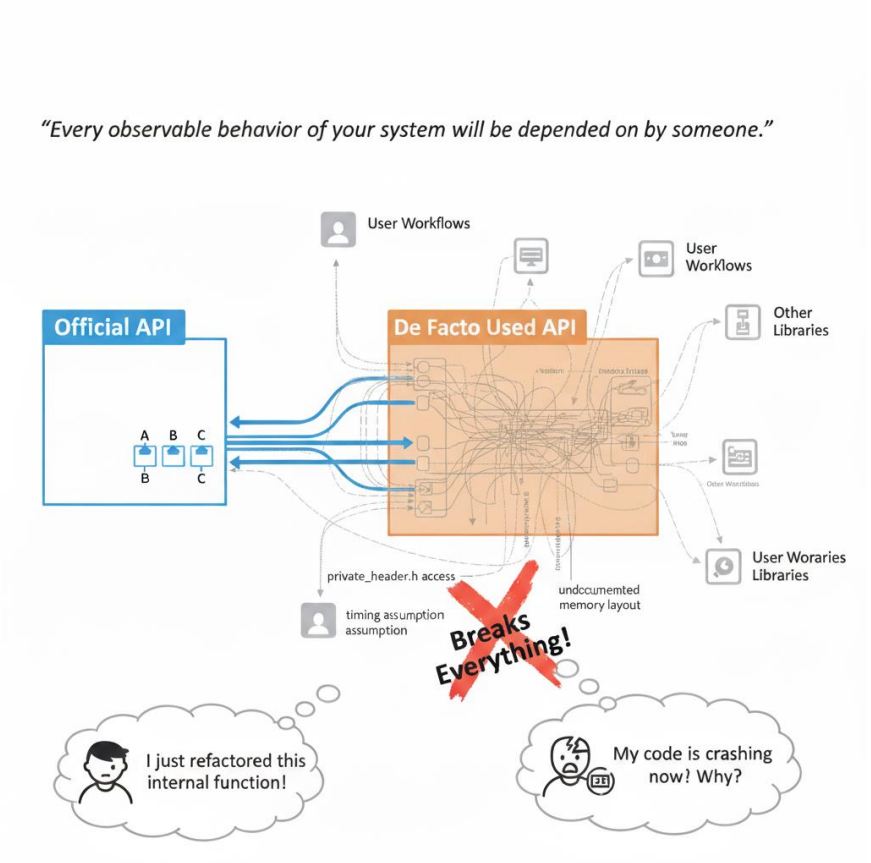
all observable behaviors of your system will be depended on by somebody."



https://xkcd.com/1172/

The Hidden Interface Of Your Software

The Problem in HPC: Users often rely on undocumented **implementation details** for performance tuning or subtle integration with other libraries (e.g., a specific memory layout, a private header file). Changing an internal implementation for the better can break an ecosystem.



Hyrum’s Law: Lessons from Kokkos
The Price of Internal Change

The Daily Grind: Refactors Break the Ecosystem

Problem: Valid internal refactors routinely broke downstream code (especially Trilinos).
Impact: Forced reverts and created a "culture of fear" around necessary internal changes.

Cause: Our Ambiguous Contract

Trilinos Inheritance: Our history as a sub-package set a precedent for disregarding API boundaries.
Our Fault: We failed to clearly define public vs. private headers, leading users to include private files like `<Kokkos_View.hpp>`.

Strategy 1: Building a Formal Contract

Action: Introduced "Backwards & Future Compatibility" guidelines.
The Rules: Forbade using internal symbols (e.g., `Kokkos::Impl::`, `KOKKOS_IMPL_` macros).
The Process: Enforced a slow, painful deprecation process over multiple releases before final changes could be made.

Hyrum’s Law: Lessons from Kokkos
The Ultimate Test

The “View of Views” Saga

The Feature: Fixed a thread-safety issue for concurrent enqueuing.
The Collision: Broke code using "View of Views"—a pattern our programming guide explicitly discourages.
The Discovery: A wider call for feedback revealed the "illegal" pattern was **pervasive across strategic applications**.

Strategy 2: Pragmatism & Adaptation

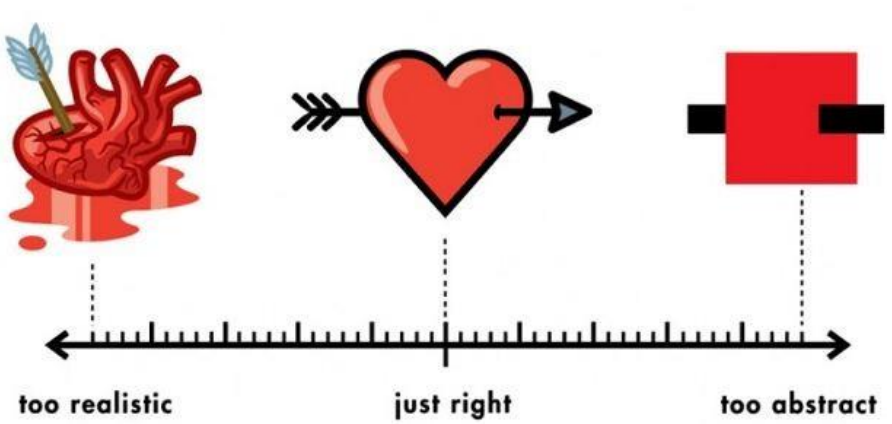
Diagnosis: We wrote a tool to detect semantic violations in older Kokkos versions.
Contingency: Added a "secret" configuration-time option to revert to the old, unsafe behavior (a temporary escape hatch).
Adaptation: Ultimately had to invent new View semantics to officially support the common use case.

Strategy 3: Knowing When to Rollback

Rolled back optimizations (e.g., unpromised fences) when the cost of change was too high for the ecosystem.
The Maintainer's Lesson: The API is what users actually **depend on**, regardless of your documentation.

Mitigation Strategy: Clear Boundaries & Deprecation

Actionable Strategy 1: Clearly define public vs. private interfaces (enforced via namespaces, header files, or tooling).
Actionable Strategy 2: Institute a formal deprecation policy with a fixed "shelf life" for features (e.g., two major releases).
Actionable Strategy 3: Invest in integration tests with key downstream projects to catch hidden breaks before release.



Key Takeaway: The Cost of Change
Every change has a cost. Minimize this cost by rigidly defining the contract and providing a clear transition path.

The Path to Sustainable HPC Software

HPSF
HIGH PERFORMANCE SOFTWARE FOUNDATION

Grow the Bus (Decentralize knowledge).

Pay Down the Principal (Refactor proactively).

Define the Contract (Manage Hyrum's Law with clear boundaries).

Sustainable software requires proactive stewardship, not just heroic coding.

The End

Let's work together to build a future of sustainable, reliable, and impactful HPC software!

Funding Acknowledgments:

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Next-Generation Scientific Software Technologies program, under contract number DE-AC05-00OR22725 (ORNL).

This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.