HPC-BP Webinar Q&A

Sustainable HPC Software: A Maintainer's Perspective

Date: 15 October 2025
Presented by: Damien Lebrun-Grandié (Oak Ridge National Laboratory)

(The slides are available via the link in the page's sidebar.)

Q: How do you think about the bus factor with respect to continued funding? For example many funding sources support new features for scientific value rather than maintenance. Should one try to track funding sources of contributors? Should PIs and funders be counted in the bus factor?

A: That is an excellent point. The Bus Factor must be considered a systemic risk spanning three layers: Personnel, Knowledge, and Funding. If a major contributor loses their specific funding stream, the Bus Factor risk immediately escalates, making the PIs and core funders crucial points of failure that absolutely should be tracked. Our strategy, therefore, must be to diversify the financial commitment across partner organizations, ensuring that maintenance responsibility is written into multiple funding streams to distribute that financial risk.
Addressing the challenge of funding sources favoring new features over maintenance, we must strategically re-frame maintenance as scientific innovation. We package major technical debt payoffs—like standardization or architectural refactoring—as necessary scientific enablers for the next generation of hardware or features. The goal is to sell sustainability not as "keeping the lights on," but as the essential, foundational work required to achieve the next exciting scientific goal.

Q: For the >50% bus factor calculation how far do you find it is useful to go back looking at contributions? Each year or release or something else?

A: When calculating the >50% Bus Factor, the optimal time window depends on whether you are measuring Architectural Risk or Active Maintenance Risk. We find it most useful to track two separate metrics.

For Active Maintenance Risk (i.e., immediate stability and velocity), we focus on a short window of 6 to 12 months or the last two to three releases. This metric identifies who holds the context necessary for quickly fixing the latest bugs, conducting urgent reviews, or finishing current feature work. If one or two people account for over 50% of the meaningful commits or reviewed lines in this period, your immediate project velocity is highly vulnerable. Conversely, to measure Architectural Risk (the capacity for major refactoring), you must look much further back — to the project's inception or at least 5+ years. This identifies the original architects who understand the deep, foundational decisions and legacy dependencies; this knowledge is slow to transfer but critical when paying down major technical debt.

Q: Do you have strategies which have worked for you to entice people to take on additional responsibilities within a project?

A: My journey into Kokkos was as a user. I learned heterogeneous programming. Became a contributor, joined the developer team and became a maintainer. You need to have room for people to become your equal. Not a model where you're the king and have everybody use the little labor. Entice people to work with you. From my experience working on older open source projects and contributing to a standard library implementation, document your processes. We can do better. Our governance defines the rules for developers explaining how things work and responsibilities of how to get from one [role] to the other. Future contributors feel it can also become theirs. HPSF logo is [showing] it's not just a Sandia project: it's a community project. Making clear to people that it's more interesting for them to get involved.

Q: How do you use tooling or language features to enforce API contracts or other other user-library interaction rules & guidelines between the user and your library? Are there any specific language features or specific tools that are especially important in your progress?

A: Enforcing API contracts is a constant battle against Hyrum's Law, and we rely heavily on C++ language features and aggressive static analysis tooling to manage it.

1. C++ Language Enforcement (The "Wall"): The primary tool for defining what is not public is the C++ namespace and header structure. We use the ::Impl:: namespace to wall off all internal implementation details. The "Backwards & Future Compatibility" guidelines strictly forbid users from accessing any symbol

or macro within this Impl space. For macro definition, we use the KOKKOS_IMPL_ prefix to make private implementation clearly visible, even when the macro system bypasses typical C++ encapsulation. If we see a user relying on an Impl artifact, we know they are intentionally going off-contract.

2. Tooling and Diagnostics (The "Warning Shot"): The most critical tool is the compiler itself, specifically how we leverage warnings. For soft deprecation (the slow, painful process mentioned earlier), we use compiler features—often specialized macros—to trigger loud warnings when a user includes a deprecated header or uses a forbidden function. This gives the user a "warning shot" for several releases before the behavior is actually removed. For harder enforcement, we rely on static analysis tools and custom Clang Tidy checks to run against downstream projects (like Trilinos) during nightly testing. This catches most violations before they become widespread problems in the wild.
3. The Diagnostic Tool (The "Lifeboat"): In the most extreme cases (like the "View of Views" incident), no generic tool is sufficient. We build custom, one-off diagnostic tools that run against old versions of the library to scan application codebases for specific semantic violations. This acts as a "lifeboat" for users, allowing them to rapidly fix their code rather than relying on manual inspection.

Q: Are there any guidelines on how to do the documentation? Are there templates to follow? Review criteria? The rationale for decisions is important. How is documenting the rationale enforced?

A: I can tell you how we do things in Kokkos but might not always be the right thing for you. We went down the road of a programming guide with intent for users. Real reference documentation for a feature - there's a balance. "Documentation is also code" - the same way you manage your code base should also be applied to your documentation. Something will be too far off. Kokkos design document is more internal or in code. Not always a good thing to document too much in your API reference; need to make sure to your eyes it's not too heavy. Over time we discover what we have to write down and force ourselves to write why so our users don't get the impression we do things for the pleasure of breaking them.

Q: Do you find it helpful to try and put real cost value on technical debt or know of any project doing that? Or is there another appropriate way of tracking technical debt for non-technical leadership?

A: I wish I could put numbers on some of these things. Occasionally you could point at very simple things and measure how often you do simple things and have fall outs - how often you do bug fixes when you touch anything. Very qualitative. Need to keep an eye on how entangled your code looks like. About strategizing to make sure the things you do don't make things worse. Bigger chokepoints are making a long-term strategy to fix them.

Q: Do all the issues you raise happen only in university and government projects? Is industry more efficient in planning ahead to manage these problems before they happen?

A: No, they're not. We think it's greener. Decently senior members interact with vendors and collaborate. Vendors don't have [research?] papers and ship the feature. Not much better than we are. One place where people are stronger is the standard libraries because of the sheer number of users hammering the interface. Some vendors have better hygiene because they have no choice; it can be risky to their business models to break things their users need. They struggle with the same things as us.

Q: How do you typically accommodate user expectations created by Hyrum's Law? Namely, what percentage of the time do you revise the contract, say "no" to a complaint, or find some middle ground?

A: Case-by-case. Over the years with Kokkos we have different categories now. Become very proactive: will often undo changes if senior developers recognize them as risky. Will do extensive searches downstream to understand the impact. Don't want to break legitimate code because we don't want to fracture the community and leave people at the side of the road. Each minor release involves a 1-1.5 hour going over release notes with bugfixes as well as rubric calling out things that can impact code. When we decide to proceed, explain how they can observe it, how to address it, and why we decided to make that change. Often have to be pragmatic from surveys we have done.

Q: Would you recommend rejecting "Half-Baked-Features" completely or could an "experimental" namespace (C++) be a solution to not give any guarantee to downstream users? Following Hyrum's law the first approach should be recommended?

A: You can’t be entirely strict. Keep in mind when you’re about to accept something you have to ask yourself “is this visible” (in C++). Do you get this feature naturally from Kokkos Core? May affect all your users who may not care about it. If you really care, clearly document promises to users and promises that are experimental. Be careful and don’t accept everything because it can become baggage.