Software Integration Challenges

Whitepaper Presentation
Collegeville Workshop, July 21-23, 2020

Todd Gamblin
Advanced Technology Office
Livermore Computing

LLNL-PRES-811108

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344. Lawrence Livermore National Security, LLC

Lawrence Livermore National Laboratory

We build codes from hundreds of small, complex pieces

> ***An essential problem I don't know how to solve:*** *Just when we're starting to solve the problem of how to create software using reusable parts, it founders on the nuts-and-bolts problems outside the software itself.*
>
> P. DuBois & T. Epperly. ***Why Johnny Can't Build***. Scientific Programming. Sep/Oct 2003.

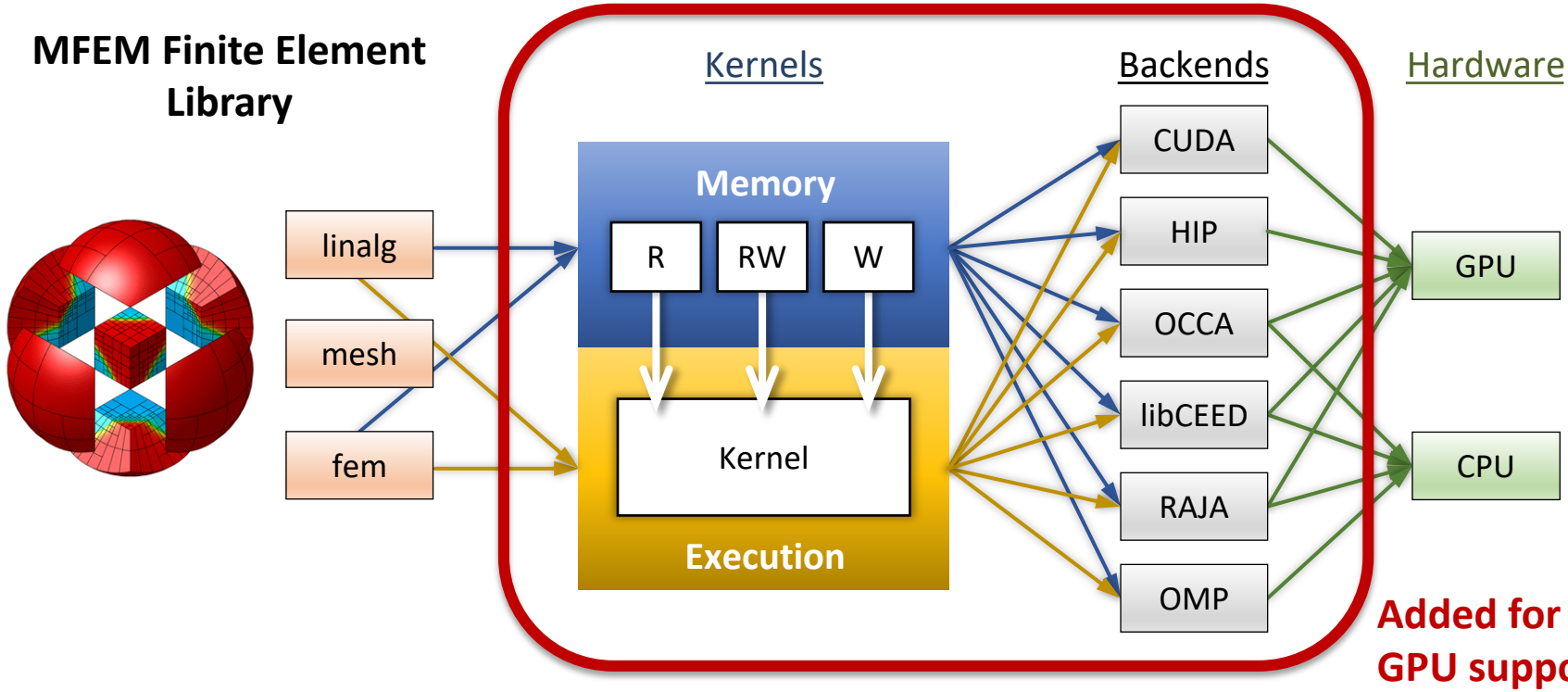
- Component-based software development dates back to the 60's
  - M.D. McIlroy, *Mass Produced Software Components*. NATO SE Conf., 1968
- **Pros are well known:**
  - Teams can and must reuse each others' work
  - Teams write less code, meet deliverables faster
- **Cons:**
  - Teams must ensure that components work together
  - Integration burden increases with each additional library
  - Integration must be repeated with each update to components
- **Managing changes over time is becoming intractable**

Build-time incompatibility; fail fast

Appears to work; subtle errors later

We must automate more of the software integration process.

Supporting accelerated architectures will require more components, and great complexity in the software stack



(pre) Exascale Machines

- MFEM has redesigned memory management and added 6 libraries
  - MFEM has not yet started to support the Intel GPUs on Aurora
- Exascale machines will have a new, rapidly changing set of libraries, compilers, and language runtimes

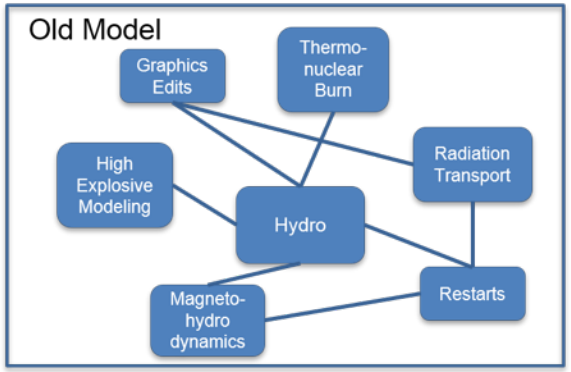
A 2015 LLNL Strategic Planning exercise embraced modularity as a fundamental approach to code development

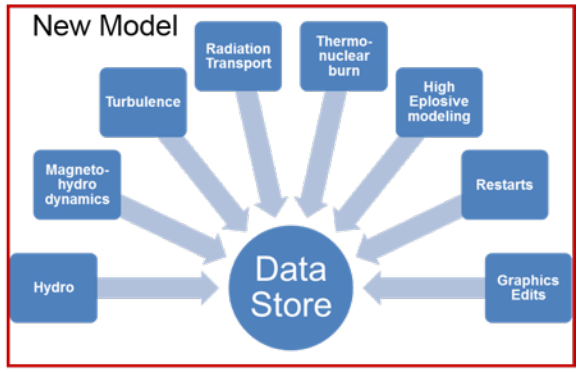
From recent WSC/CP presentation reflecting on increased code modularly as a major pillar of the 2015 next-gen code strategy.

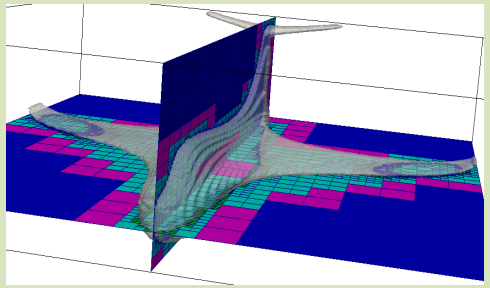
It was recently revisited and confirmed as the right approach, despite challenges with integration.

ECP, E4S, xSDK, many other projects are also embracing modularity!

Physics packages, including hydro, are developed as modular capabilities





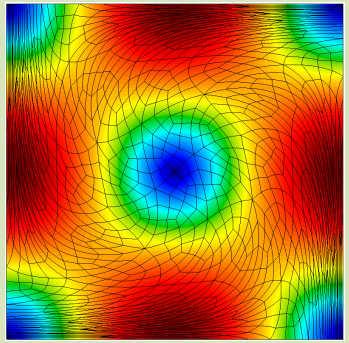


Sidre
(in-core datastore)

Blueprint
("Mesh-aware" data description)

Conduit
(In-core data schema)

Relay
(I/O – HDF5, etc.)



AXOM Toolkit provides a modular CS infrastructure

Quest & Primal
- computational geometry
- surface queries,
- point containment,
- spatial acceleration structures

SLIC & Lumberjack
- unified parallel logging
- message filtering for multi-physics applications

Mint & SLAM
- mesh data model
- abstractions for integrating meshes and numerical algs

Courtesy: Chris Clouse, WSC Program Lead for Computational Physics

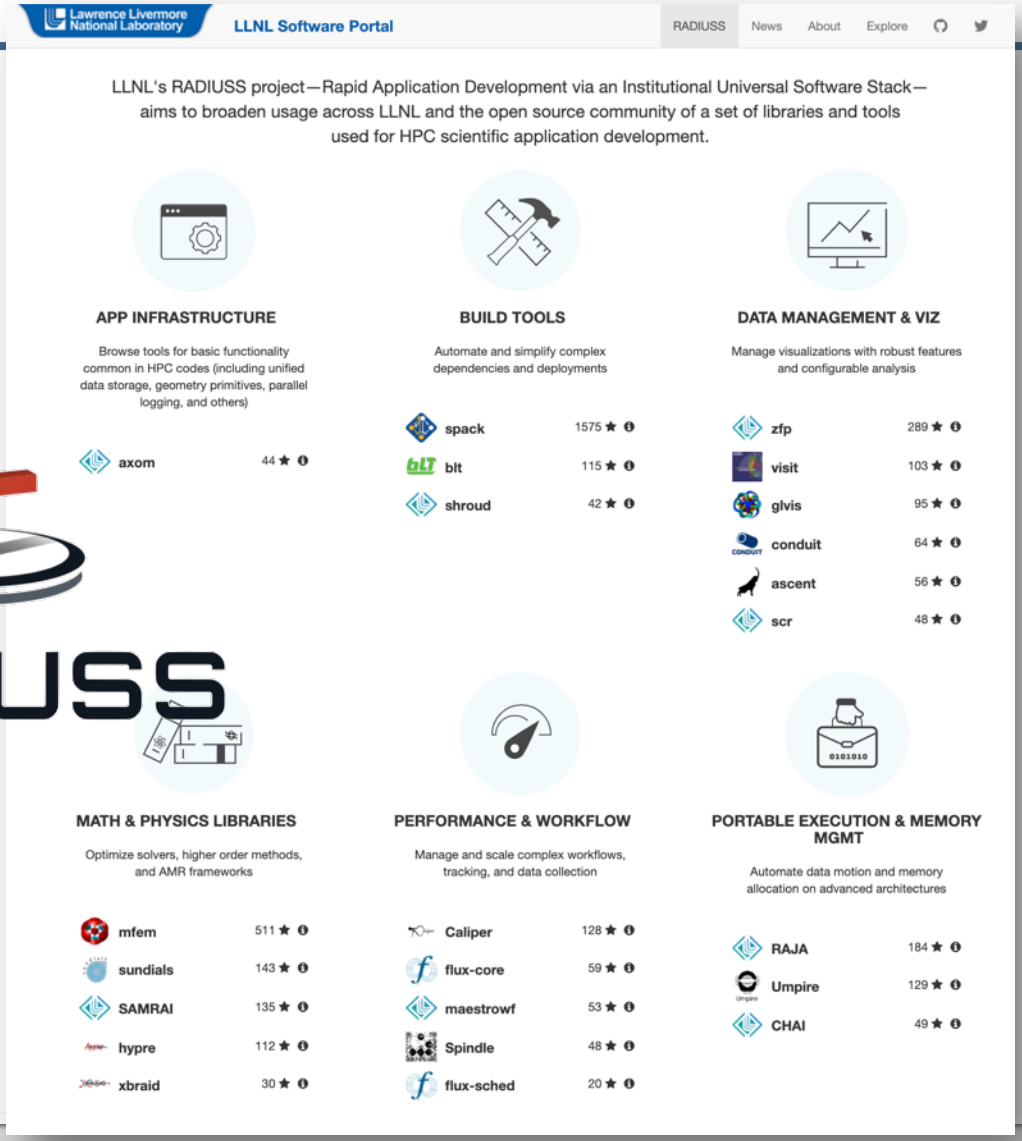
RADIUSS is integrating software components from across the lab into a universal software stack

"… Scientific software is increasingly becoming core infrastructure for the lab, and must be treated as such…"

"Develop strategy to deploy a common base of foundational scientific software with opt-in adoption from lab applications"

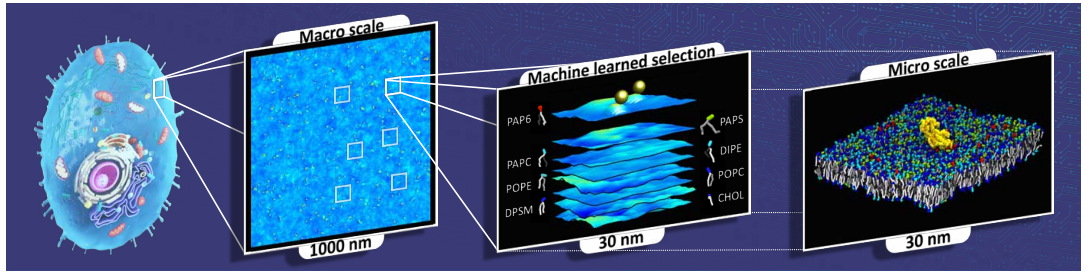
"We're leveraging years of major ASC (and other) investments"

radiuss

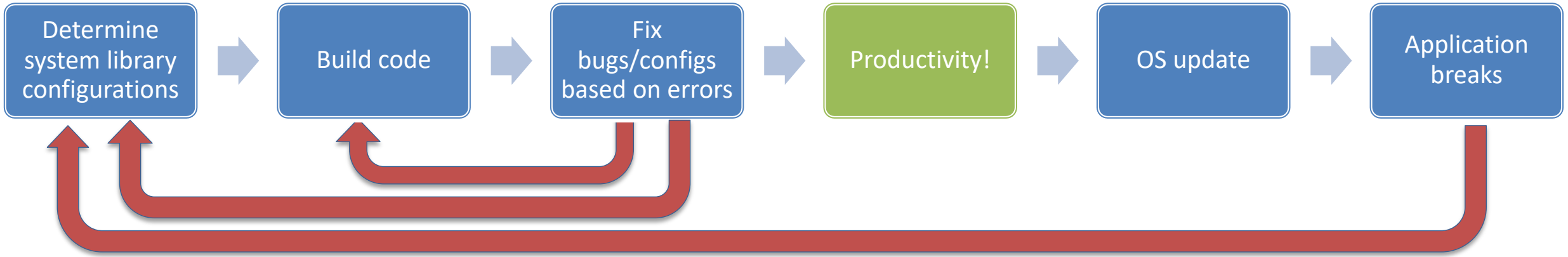


- Component usage is on the rise!
  - Projects like Lido (ENG) are adopting Axom (with some integration difficulties)

Build integration complexity has caused massive delays in the MuMMI developer workflow



- LLNL's MuMMI code is a used to model drugs, including cancers and, more recently, COVID-19
- When standing up Sierra, the team was at the mercy of constant system updates



- We use system dependencies (MPI, compilers) to:
  - get the best performance from the machine.
  - avoid long build times

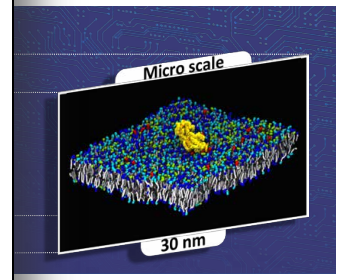
Di Natale et al. *A Massively Parallel Infrastructure for Adaptive Multiscale Simulations: Modeling RAS Initiation Pathway for Cancer*. In **Supercomputing 2019** (SC '19). 2019. **Best paper.**

Build integratio … he MuMMI devel

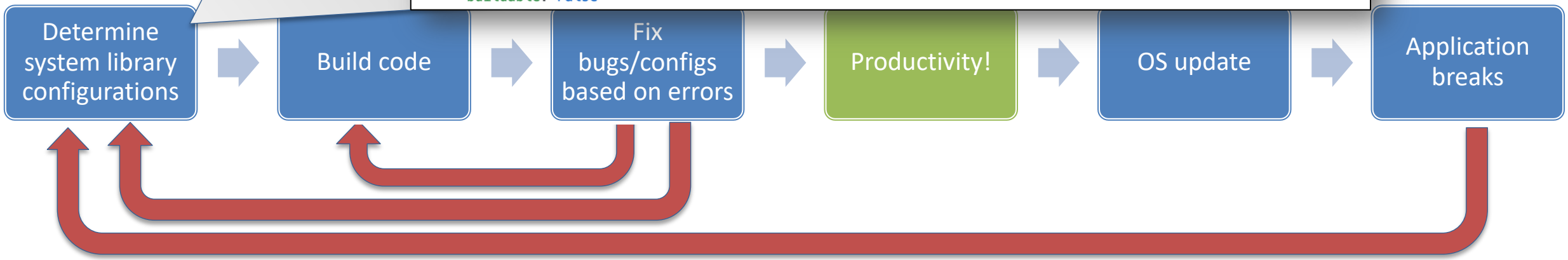
```
opengl:
  paths:
    opengl@1.7.0: /usr
  buildable: False
openglu:
  paths:
    openglu@1.3.1: /usr
  buildable: False

# Lock down which MPI we are using
mvapich2:
  paths:
    # clang mvapich2
    mvapich2@2.3%clang@9.0.0 arch=linux-rhel7-ivybridge: /usr/tce/packages/mvapich2/mvapich2-2.3-clang-9.0.0
    # gcc mvapich2
    mvapich2@2.3%gcc@8.1.0 arch=linux-rhel7-ivybridge: /usr/tce/packages/mvapich2/mvapich2-2.3-gcc-8.1.0
    # intel mvapich2
    mvapich2@2.3%intel@19.0.4 arch=linux-rhel7-ivybridge: /usr/tce/packages/mvapich2/mvapich2-2.3-intel-19.0.4
  buildable: False
```

~ 100 lines, 20 pinned system dependencies configured per machine



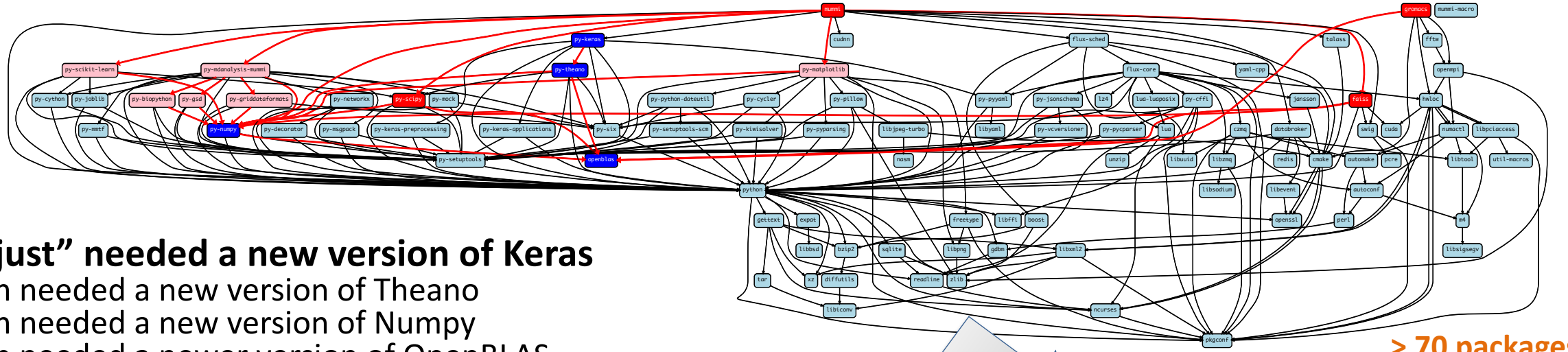
- LLNL's MuMMI c … including cance …
- When standi …



- We use system dependencies (MPI, compilers) to:
  - get the best performance from the machine.
  - avoid long build times

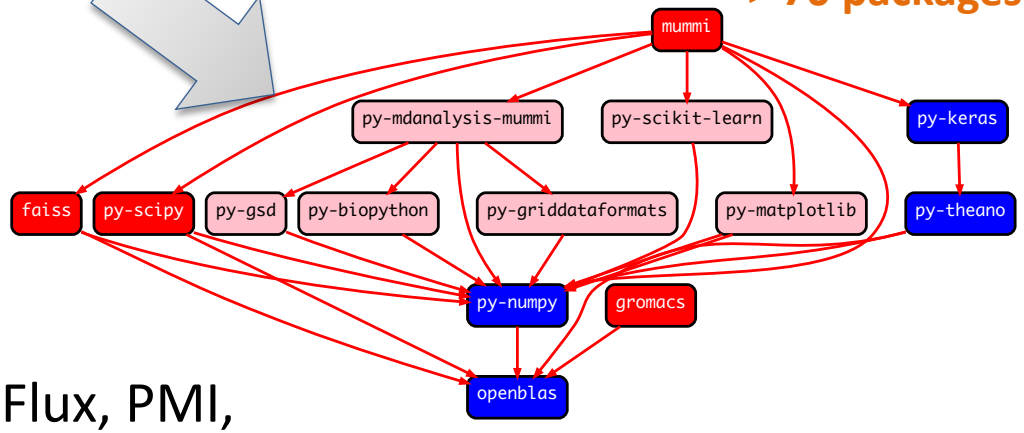
Di Natale et al. *A Massively Parallel Infrastructure for Adaptive Multiscale Simulations: Modeling RAS Initiation Pathway for Cancer*. In **Supercomputing 2019** (SC '19). 2019. **Best paper.**

Transitive dependency requirements can cause cascading issues



- **Team "just" needed a new version of Keras**
  - which needed a new version of Theano
  - which needed a new version of Numpy
  - which needed a newer version of OpenBLAS
  - But the team was using the system OpenBLAS, which was too old
  - Team had to build several versions of OpenBLAS before they found one compatible with all other packages in the DAG
  - Then had to rebuild the entire stack for ABI compatibility

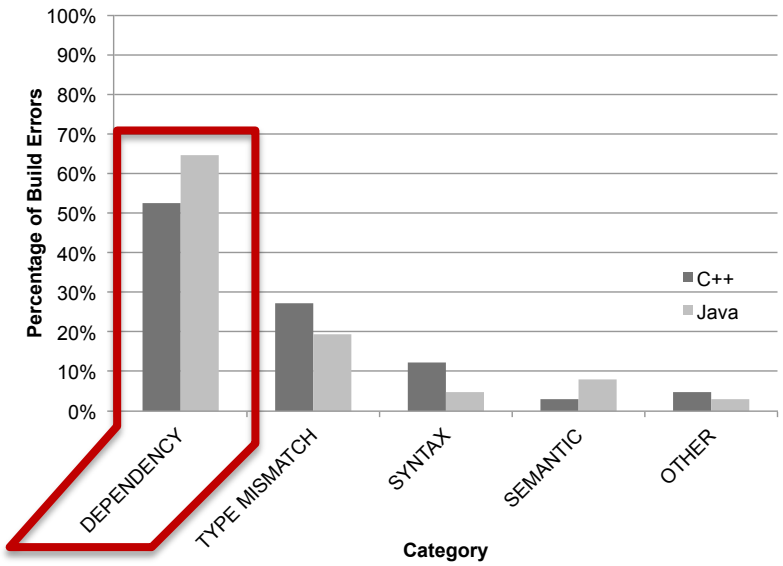
> 70 packages



- This particular issue consumed **36 person hours**
- **Frequent OS updates causing ABI incompatibilities** between Flux, PMI, and the system MPI cost **hundreds of person hours**

Even outside of HPC, dependencies are the most frequent cause of build errors and software release delays

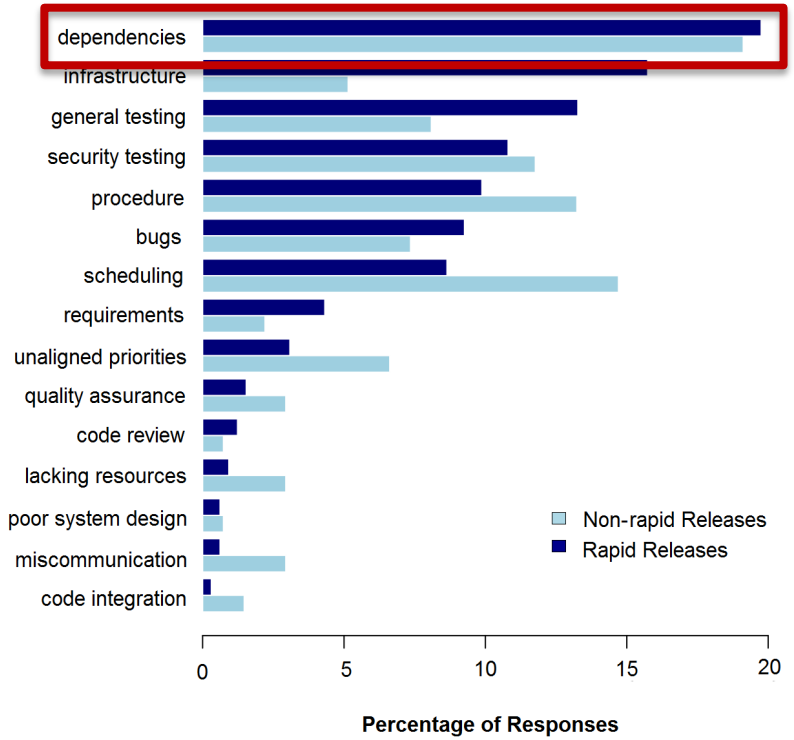
Types of errors in 26.6 million builds at Google



"our study clearly shows that better tools to resolve dependency errors have the greatest potential payoff"

Survey of 26.6M builds by 18K developers at Google. Seo et al., *Programmers' Build Errors: A Case Study (at Google).* ICSE 2014.

Factors perceived to cause release delays among 491 developers at ING



Survey of 491 individuals from 691 teams at ING. Kula et al., *Releasing fast and slow: an exploratory case study at ING.* ESEC/SIGSOFT FSE 2019.

- Developers avoid updates to avoid problems with dependencies, leading to:
  - Security vulnerabilities
  - Lack of performance
  - Stagnation as upgrades become harder and harder over time

Axom, Serac, xSDK, and E4S teams have similar issues with managing configuration complexity

- Teams *really* like to lock versions down for testing:
  - **Axom** team tests on several systems and pins about 20 package versions to consistent (at the time) values
  - **Serac** team pins more system versions than this
  - **xSDK** and **E4S** stacks from ECP pin specific versions for each package

- Incompatibilities arise and builds fail in one or both of two ways:
  1. Spack upgrade leads to failure because new versions and options enter the Spack repository that are incompatible
  2. OS upgrades at a local site change local versions underneath a package

- Inevitably, this version locking effort is spent over and over again for subsequent releases

```
depends_on('dealii +trilinos', when='+trilinos +dealii')
depends_on('dealii ~trilinos', when='~trilinos +dealii')
depends_on('dealii@develop~assimp~python~doc~gmsh+petsc+slepc+mpi~int64+hdf5~netcdf+metis~sundials~ginkgo~symengine', when='@develop +dealii')
depends_on('dealii@9.1.1~assimp~python~doc~gmsh+petsc+slepc+mpi~int64+hdf5~netcdf+metis~sundials~ginkgo~symengine', when='@9.5.0 +dealii')
depends_on('dealii@9.0.1~assimp~python~doc~gmsh+petsc~slepc+mpi~int64+hdf5~netcdf+metis~ginkgo~symengine', when='@9.4.0 +dealii')

depends_on('pflotran@develop', when='@develop')
depends_on('pflotran@xsdk-0.5.0', when='@0.5.0')
depends_on('pflotran@xsdk-0.4.0', when='@0.4.0')
depends_on('pflotran@xsdk-0.3.0', when='@0.3.0')
depends_on('pflotran@xsdk-0.2.0', when='@xsdk-0.2.0')

depends_on('alquimia@develop', when='@develop')
depends_on('alquimia@xsdk-0.5.0', when='@0.5.0')
depends_on('alquimia@xsdk-0.4.0', when='@0.4.0')
depends_on('alquimia@xsdk-0.3.0', when='@0.3.0')
depends_on('alquimia@xsdk-0.2.0', when='@xsdk-0.2.0')

depends_on('sundials+superlu-dist', when='@0.5.0: %gcc@6.1:')
depends_on('sundials@develop~int64+hypre+petsc', when='@develop')
depends_on('sundials@5.0.0~int64+hypre+petsc', when='@0.5.0')
depends_on('sundials@3.2.1~int64+hypre', when='@0.4.0')
depends_on('sundials@3.1.0~int64+hypre', when='@0.3.0')

depends_on('plasma@19.8.1:', when='@develop %gcc@6.0:')
depends_on('plasma@19.8.1:', when='@0.5.0 %gcc@6.0:')
depends_on('plasma@18.11.1:', when='@0.4.0 %gcc@6.0:')

depends_on('magma@2.5.1', when='@develop +cuda')
depends_on('magma@2.5.1', when='@0.5.0 +cuda')
depends_on('magma@2.4.0', when='@0.4.0 +cuda')
depends_on('magma@2.2.0', when='@0.3.0 +cuda')

depends_on('amrex@develop', when='@develop %intel')
depends_on('amrex@develop', when='@develop %gcc')
depends_on('amrex@19.08', when='@0.5.0 %intel')
depends_on('amrex@19.08', when='@0.5.0 %gcc')
depends_on('amrex@18.10.1', when='@0.4.0 %intel')
depends_on('amrex@18.10.1', when='@0.4.0 %gcc')

depends_on('slepc@develop', when='@develop')
depends_on('slepc@3.12.0', when='@0.5.0')
depends_on('slepc@3.10.1', when='@0.4.0')

depends_on('omega-h +trilinos', when='+trilinos +omega-h')
depends_on('omega-h ~trilinos', when='~trilinos +omega-h')
depends_on('omega-h@develop', when='@develop +omega-h')
depends_on('omega-h@9.29.0', when='@0.5.0 +omega-h')
depends_on('omega-h@9.19.1', when='@0.4.0 +omega-h')

depends_on('strumpack@master', when='@develop +strumpack')
depends_on('strumpack@3.3.0', when='@0.5.0 +strumpack')
depends_on('strumpack@3.1.1', when='@0.4.0 +strumpack')

depends_on('pumi@develop', when='@develop')
depends_on('pumi@2.2.1', when='@0.5.0')
depends_on('pumi@2.2.0', when='@0.4.0')
```

xSDK

~21 core libraries
~70 total packages

Pinned versions and options from xSDK

Three main ways to deal with dependencies have emerged in the past 10-20 years

	Bundled Distribution	Semantic Versioning	Live at Head
Examples	Linux distributions (Red Hat, Debian) E4S, xSDK, Anaconda Spack with locked versions	Spack NPM, Cargo, Go Most language dependency managers	Google, Facebook, Twitter
Idea	Curate a large set of mutually compatible dependencies	Use uniform version convention, Solve for compatible set	Everything in one repository, Developers test changes with all *dependents*
Pros	Stability (if software is included)	Frequent updates Only relies on local information Works in theory	Frequent updates Stability, consistency All changes tested
Cons	**Infrequent updates** **High packaging/curation effort** **Lack of flexibility**	**Versions are coarse** **Developers over-constrain/over-promise** **Errors start to dominate at scale**	**Doesn't scale beyond a single organization** **High computational cost of testing** **Lack of flexibility (typically just one target env.)**

- All of the approaches have serious drawbacks
- Need a way to guarantee stability, frequent updates, and version/config flexibility

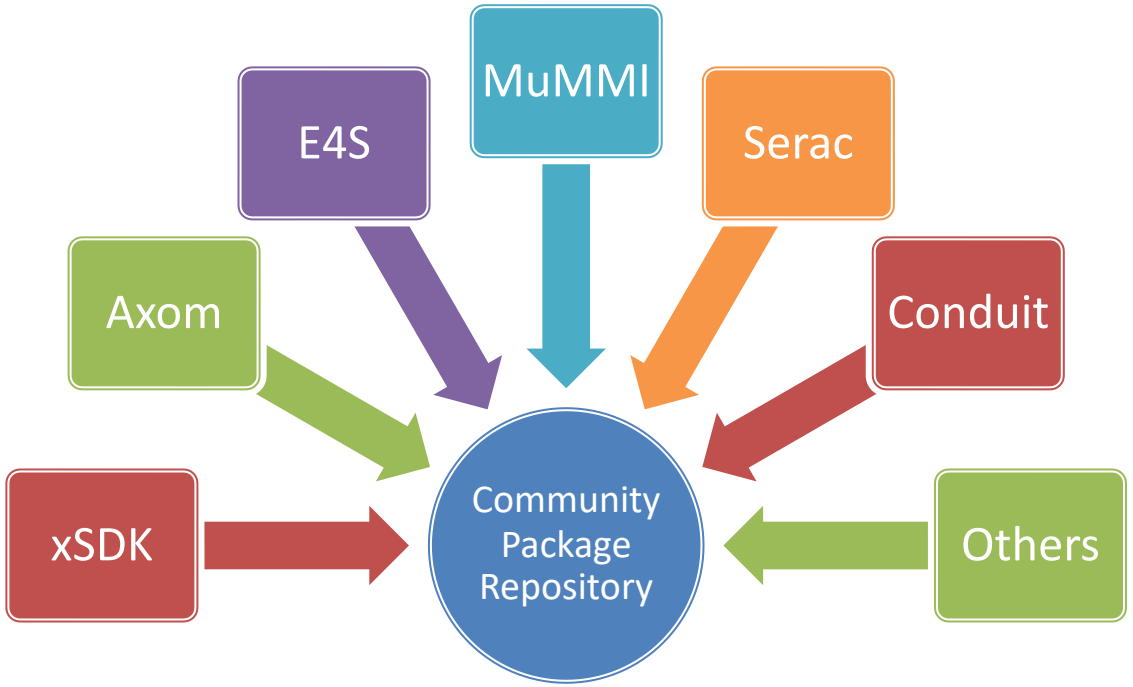
Taxonomy c/o T. Winters et al. *Software Engineering at Google.* 2020.

The fundamental problem in integration workflows is lack of compatibility information

In workflows we've seen so far:

1. No information on how system libs were built
2. Build repeatedly to find compatible libraries
3. Hard constraints (pinned versions, etc.) hide information and limit choice

Each team ends up curating its own configuration with baked-in, incompatible assumptions

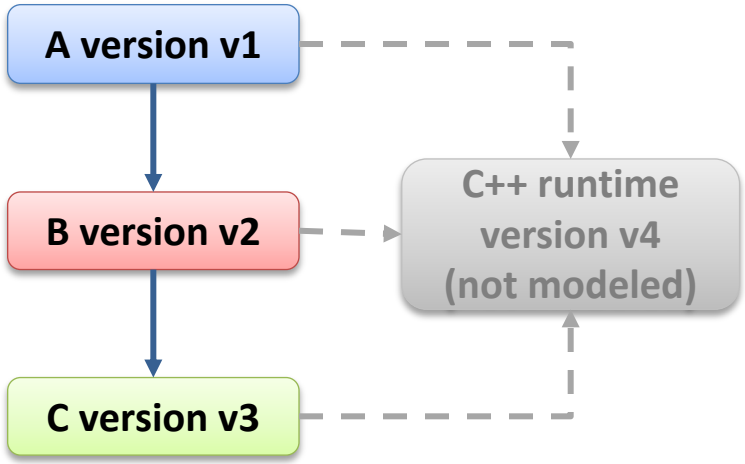


If all projects add restricted versions, conflicts will eventually arise that prevent all packages from building.

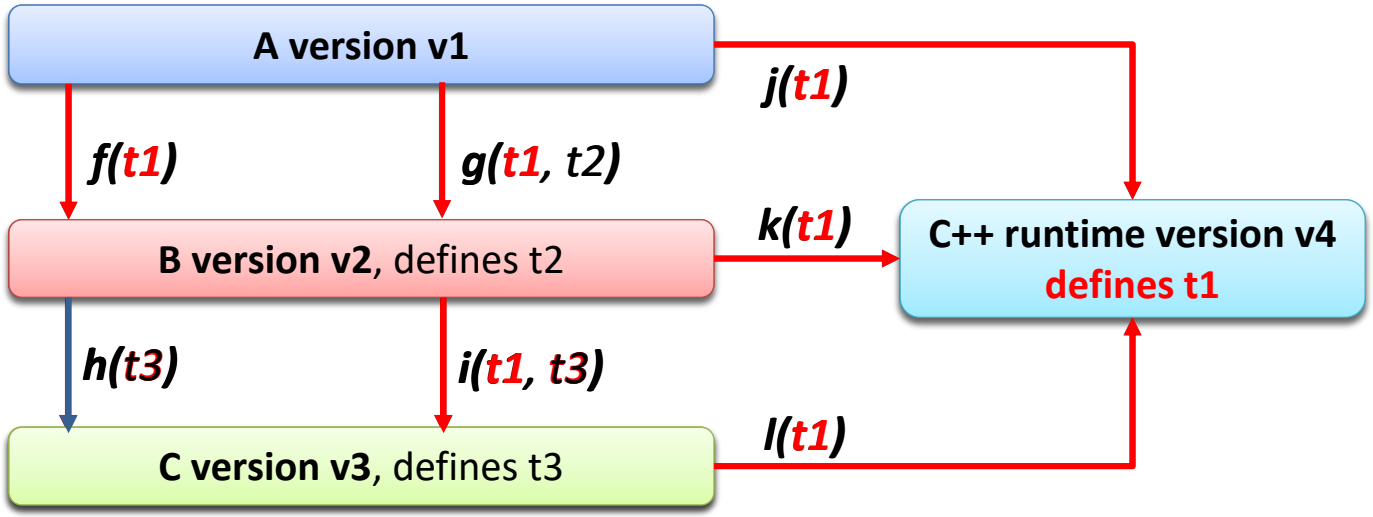
We need a way to reuse package builds among different communities of developers

Package managers do not model software at sufficient granularity

Current model is coarse



Complete model represents *how* changes affect code

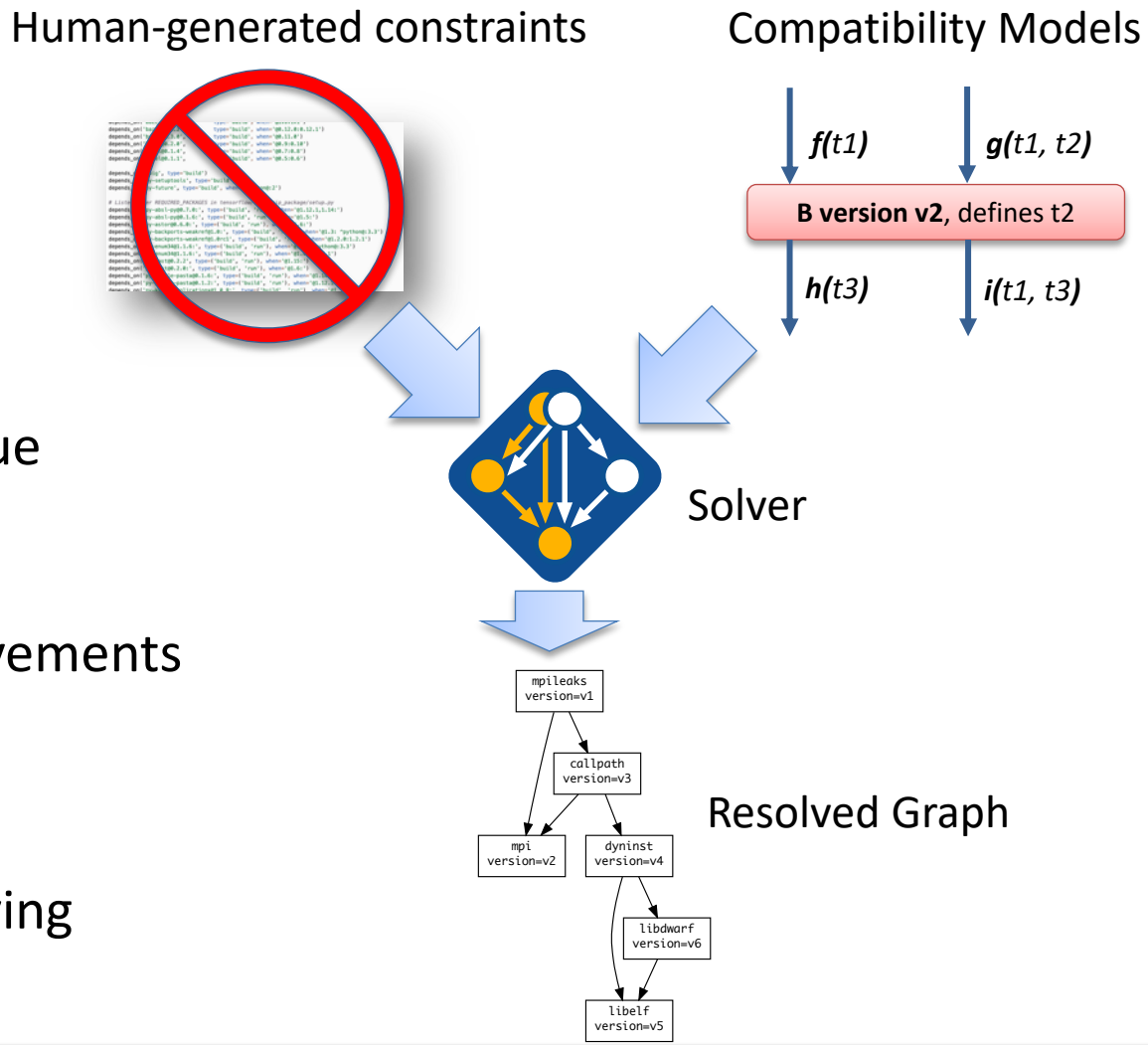


- Need to model libraries at call granularity:
  - Entry calls
  - Exit calls
  - Data type definitions & usage
- We need to model runtime libraries behind compilers
  - C++, OpenMP, glibc
  - GPU runtimes
- Need to model changes in the graph
  - "If C changes, what needs to be rebuilt?"
  - We will model semantics of interfaces
  - "If h(t3) changes, is B still correct?

This model allows us to reason about compatibility, so we can find usable packages

We need better package solvers using sound information

- Recall: package managers produce *valid* but not *sound* graphs.
  - Compatibility models give us *soundness*
- Can generate entry/exit ABI models using binary analysis (ground truth)
- Solve not on the coarse version model but on true *compatibility*
  - Solve for *what will work* instead of what humans say
- Past 10-20 years have brought enormous improvements in solver technology
  - CDCL algorithms
  - Optimizing SMT and ASP solvers
- Time is right to attack packaging with better solving



These techniques would enable much easier software reuse across project and community boundaries

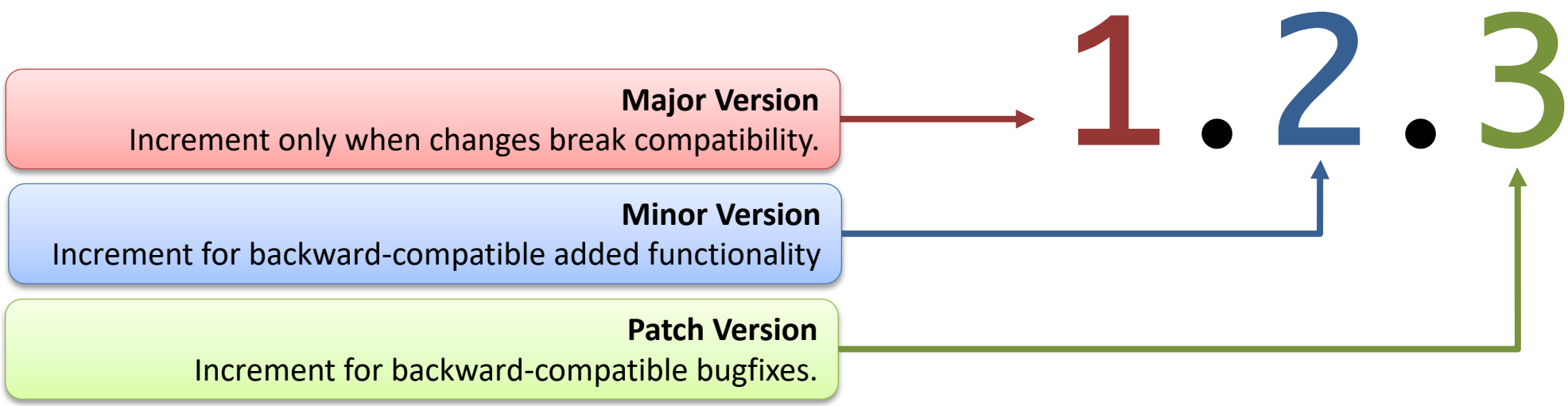
1. **When OS updates happen underneath a stack:**
   - Know what changed by examining the binaries' ABI
   - Identify what in the stack is no longer compatible
   - Rebuild compatible configurations

2. **When user requirements change (e.g., due to a new version):**
   - Know which packages need to change to meet the new requirements
   - Identify existing binaries (system or packaged) that satisfy the requirements
   - Install binaries or rebuild as necessary

3. **When information is not available:**
   - Extrapolate what and how to build based on past, similar builds

We will enable binary code reuse to reduce iteration in developer workflows

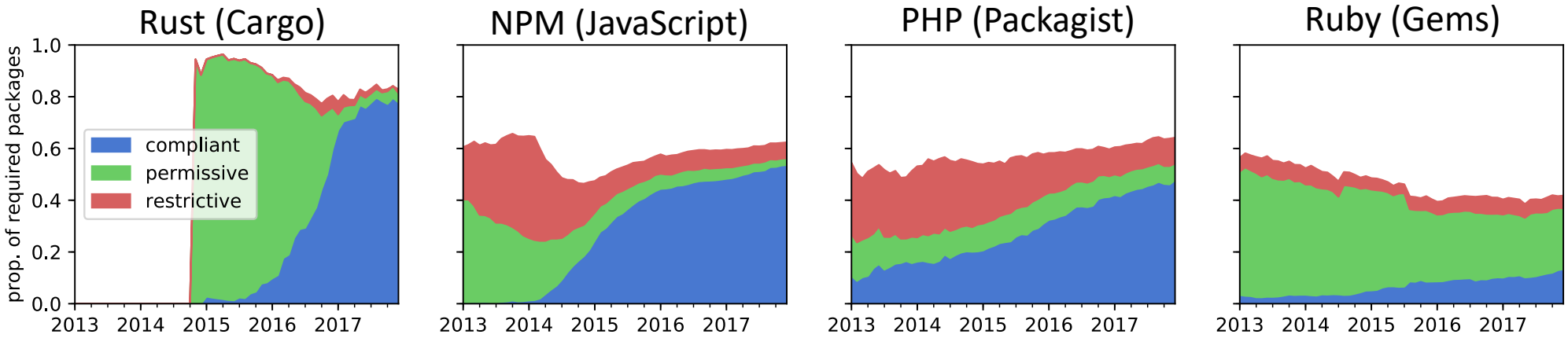
Semantic versioning is the de-facto standard for conveying compatibility information



- Pitfalls:
  - Applies to the whole package, but packages may only depend on a subset of functions
  - May over-promise: packages may break despite developers' intentions
  - May over-constrain: pinned dependency versions are common but lead to false unsatisfiable cases
- Relies on developers to specify versions correctly
  - Relies on broad community participation

https://semver.org/

Humans do not accurately specify version information



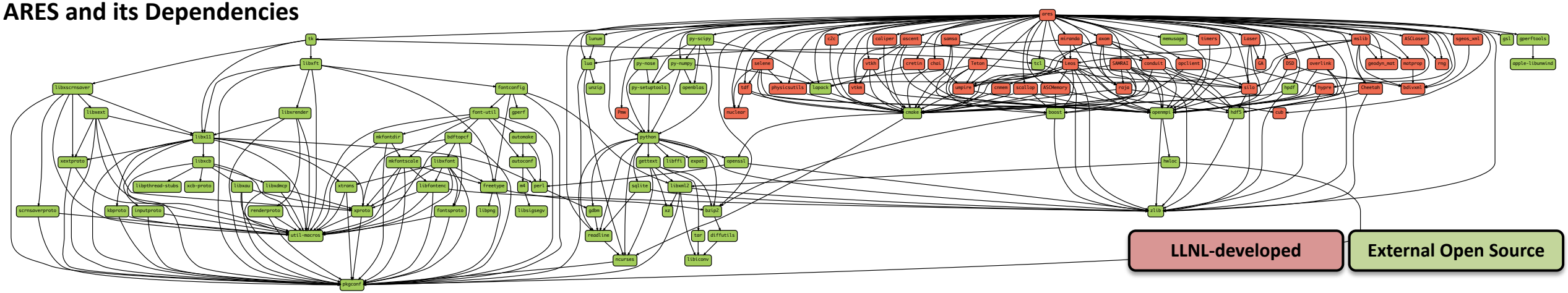
- Plots show version restrictions across 4 package ecosystems.
  - **Permissive** leaves room for lots of room for incorrect builds
  - **Restrictive** rules out builds that would work
  - Non-specialized (white) leaves everything open (no constraints)

- HPC ecosystem is most like Ruby (right) – dated, with many permissive constraints
  - Most projects don't use semantic versioning and won't switch
  - Likelihood of build errors is very high

Decan et al. *What Do Package Dependencies Tell Us About Semantic Versioning?* IEEE Trans. Software Eng. May 2019.

Even our proprietary codes rely on many open source libraries

ARES and its Dependencies



- **Since 2014, number of libraries has grown from ~60 to > 100**
  - Added complexity is from modularization and libraries to support GPUs and accelerators

- **Team constantly struggles with updates, improvements, and feature additions:**

— OS updates on LANL Trinitite system	**1 day – 1 week per update**
— OS updates at LLNL	**similar (sometimes give up on outdated machines)**
— Python and other OSS updates (many transitive upgrades required)	**days – 1 week**
— CUDA updates at LLNL	**days**
— Fortran ABI Issues with mixed compilers	**days**
— Using Numpy with XL Fortran	**weeks of LC Spack developer time (Becker)**