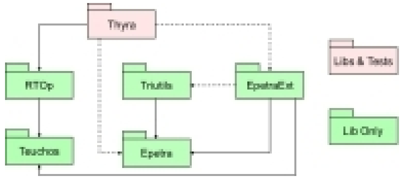
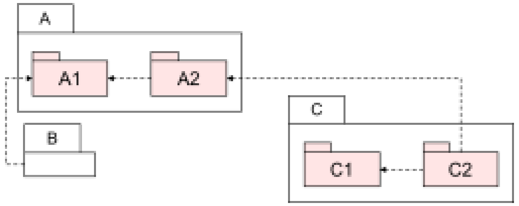
SAND2022-2272C

Sandia National Laboratories

Challenges and Suggested Solutions to Sustainable Build, Test, and Integration Processes in CSE Software Ecosystems





Roscoe A. Bartlett
Department 1424
Software Engineering and Research

February 25, 2022

SIAM Parallel Computing Conference, 2022

https://bartlettroscoe.github.io/

U.S. DEPARTMENT OF ENERGY
NNSA

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

Bartlett: Build, Test, and Integration Efforts in CSE Software

- **2001-2010**: Trilinos numerical algorithms development (https://trilinos.org)
  - Contributed to many Trilinos packages (5K+ commits)
  - Build and integrations with research and production application codes and Trilinos
- **2005:** Co-developed of Makefile.export.<package> system for Trilinos autotools build system
  - Scalable package-based architecture for Makefile-based build systems
- **2007**: Lead ASC Vertical Integration milestone (pioneered APP+Trilinos integration processes)
- **2008-present**: Took over transition and maintenance of Trilinos build system to CMake
  - Scalable architecture for large CMake projects 8 years before well supported by modern CMake 3.7
- **2008-present:** Lead numerous contracts with Kitware to extend CMake, CTest and CDash
  - CMake: e.g.: Ninja support for Fortran, improved parallel build performance for Ninja
  - CTest: e.g. Parallel running of tests, test resource allocation control (for GPUs)
  - CTest, CDash: e.g. Asynchronous submits and processing, subproject support, improved query filters
- **2010-2016**: Infrastructure team lead and integration architect for DOE CASL project (Consortium for the Advanced Simulation of Light-water reactors) (https://casl.gov/)
  - Multi-institution, multi-team, multi-repository development and integration workflows
- **2015**: Initial co-author of Exascale Computing Project xSDK Community Package Policies
- **2016-2020:** Tool and Development Environment Lead for SNL ATDM project
  - Lead stabilization of Trilinos on pre-exascale platforms (largest in Trilinos history)
  - Developed integration plans for Trilinos and ATDM application codes
- **2019:** Developed prototype for tools and third-party library installation system using Spack => SNL SEMS Spack-CM
- **2021-present**: Refactoring Trilinos/CMake build system to modern CMake
  - Initial goal: Maintain backward compatibility for thousands CMakeLists.txt files and thousands of user configure scripts in all environments!

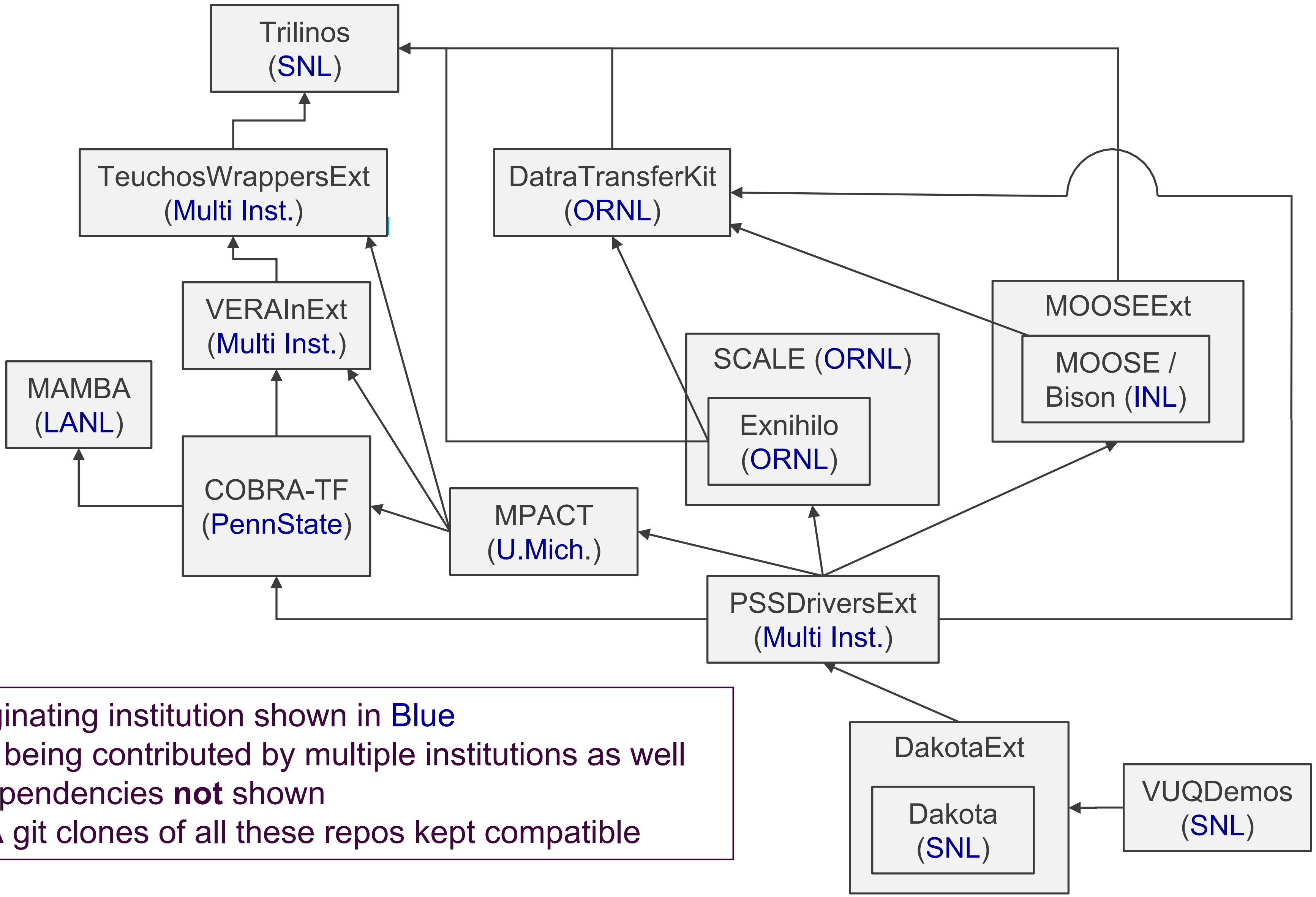
Overview of CASL

- **CASL: C**onsortium for the **A**dvanced **S**imulation of **L**ightwater reactors
- DOE Innovation Hub including DOE labs, universities, and industry partners
- Goals:
  - Advance modeling and simulation of lightwater nuclear reactors
  - Produce a set of simulation tools to model lightwater nuclear reactor cores to provide to the nuclear industry: **VERA: Virtual Environment for Reactor Applications**.
- Phase 1: July 2010 – June 2015
- Phase 2: July 2015 – June 2020
- Organization and management:
  - ORNL is the hub of the Hub
  - Milestone driven (6 month plan-of-records (PoRs))
  - Focus areas: **Physics Integration (PHI)**, Thermal Hydraulic Methods (THM), Radiation Transport Methods (RTM), Advanced Modeling Applications (AMA), Materials Performance and Optimization (MPO), Validation and Uncertainty Quantification (VUQ)

CASL VERA Development Overview (2016)

- VERA development was complicated in almost every way ☹
- VERA composed of:
  - 21 different Git repositories (clones of other repos)
  - Different access lists for each Git repository (NDAs, Export Control, IP, etc.)
  - Integrating development efforts from many teams from 9+ institutions
- Single large CMake build system using TriBITS CMake Framework:
  - Very large full source code base:
    - 55K source and script files
    - 12M lines of code (not comments)
    - 2,700 CMakeLists.txt files
  - 229 packages + subpackages enabled (out of 496 total) ≈ **46% of full code base**
  - Most CMake developer reconfigures take place in less than 30 seconds!
- VERA Software Development Process:
  - VERA integration maintained by continuous and nightly testing:
    - Pre-push CI testing: checkin-test-vera.sh, cloned VERA git repos
    - Post-push CI testing: CTest/CDash, all VERA git repos
    - Nightly testing: MPI and Serial builds, Debug and Release builds, …
    - Main 100% passing builds and tests most days!
  - Many internal and external repository integrations on daily basis
  - VERA releases are taken off of stable 'master' branches on casl-dev git repos.
  - Very low maintenance cost of the infrastructure

Dependences Between Selected CASL VERA Repositories (2016)



- Primary/originating institution shown in Blue
- Most codes being contributed by multiple institutions as well
- All direct dependencies **not** shown
- Local VERA git clones of all these repos kept compatible

Layers of Build Tools and Build Systems

Package dependency handling and build/install/test orchestration

- Listing of different packages (each with their own meta-build system tool) and dependencies and version information.
- Acquires (and patches) source code for each package for the correct version, consistent configure, builds, and installs
- Examples (popular) tools/approaches: Spack, CMake External Project, SNL CApp, home grown scripts
- Arguably the most popular approach: Home-grown scripts

Meta-build system / Build file generator

- Uses higher-level description of the build targets in platform independent way
- Rules for each compiler and platform to generate detailed compiler and linker command-line options
- Automatically computes dependencies specification in generated Makefiles, Ninja files, or other tools
- Example (popular) tools/approaches: CMake, GNU Autotools, home-grown Makefiles (with thing configure scripts)
- Arguably the most popular approach: CMake

Build driver with low-level dependencies

- Given dependencies between different input and output files, runs low -level compile commands
- Only builds output targets that are out of date given the time stamps of their upstream dependencies (files).
- Example (popular) tools: (GNU) Makefiles, Google Ninja
- Arguably the most popular approach: GNU Makefiles

Raw compile and link commands

```
$ g++ -I<dir1> -I<dir2> … -isystem <dirn> … -fopenmp -O3 -DNDEBUG -fPIC -std=c++14 -MD -MT … -o
  <object_file>.o -c <source_file>.cpp
$ g++ … <object1>.o <object2>.o … -Wl,-rpath,<dir1>:<dir2> … -L<dir3> … -l<lib1> -l<lib2> … -o <exec>
```

Why CMake?

Open-source tools maintained and used by a large community and supported by a profession software development company (Kitware).

CMake:

- Simplified build system, easier maintenance
- Improved mechanism for extending capabilities (CMake language)
- Support for all major C, C++, and Fortran compilers.
- Automatic full dependency tracking (headers, src, mod, obj, libs, exec)
- Good Fortran support (parallel builds with modules with src => mod => object tracking, C/Fortran interoperability, etc.)
- Shared libraries on all platforms and compilers (support for RPATH)
- Faster configure times (e.g. > 10x faster than autotools)
- Generates different backend builds: Makefiles, Google **Ninja**, Visual Studio, Eclipse, XCode, …
- Portable support for cross-compiling

CTest:

- Parallel running and scheduling of tests and test time-outs, resource (GPU) allocation
- Memory testing (Valgrind)
- Line coverage testing (GCC GCOV)
- Better integration between test system and build system

CDash:

- Web server for display, query, and archive of build, test, memory, and coverage results
- Flexible query and filtering of build and test results
- REST API to extra data to develop various tools

Recent news:

- There has been significant growth in CMake adoption, maturation and feature development in recent years.
- CMake is now most popular build system for C++ code in the world
- Improved documentation: Book "Professional CMake"

Major obstacles to build, test, and integration of CSE Software

Package dependency handling and build/install/test orchestration
e.g. Spack, homegrown scripts

↓

Meta-build system / Build file generator
e.g. CMake, GNU Autotools, homegrown Makefiles

↓

Build driver with low-level dependencies
e.g.: Makefiles, Google Ninja

↓

Raw compile and link commands

```
$  g++ … -o <object_file>.o -c
   <source_file>.cpp
$  g++ <object1>.o … -o <exec>
```

- **Heterogeneity in the build file generators (CMake, Autotools, home-grown tools that generate Makefiles, etc.)**
- Inconsistent and incompatible usages of CMake (mix of old and modern CMake approaches with different packages)
- Inability to tweak generation of low-level compile and link lines to address difficult cases on some builds and platforms (i.e. **CMake**)
- Compatible upgrades of dependent packages (i.e. "Dependency hell")
- Lack of robust portable test suites to drive packages integration processes (e.g. fragile randomly failing tests).
- Difficulties debugging through all of the different layers down to low-level compiler and linker command-lines (i.e. Spack => CMake => Makefiles/Nina => Raw compile and linke commands)

Suggested solutions to major obstacles to build, test, and integration

Package dependency handling and build/install/test orchestration
e.g. Spack, homegrown scripts

↓

Meta-build system / Build file generator
e.g. CMake, GNU Autotools, homegrown Makefiles

↓

Build driver with low-level dependencies
e.g.: Makefiles, Google Ninja

↓

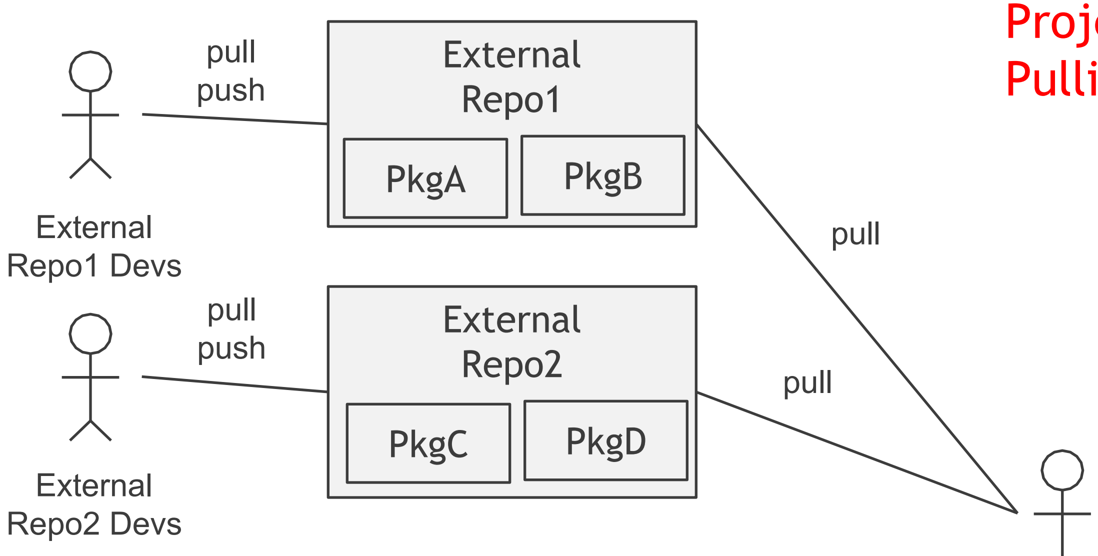
Raw compile and link commands

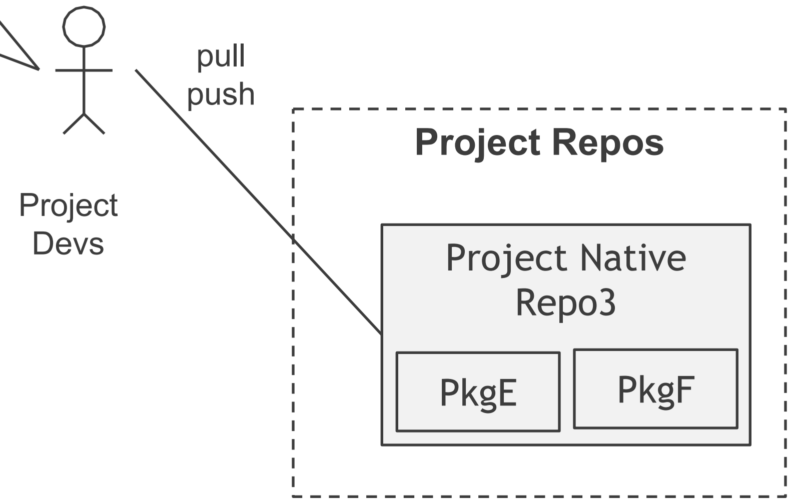
```
$ g++ … -o <object_file>.o -c
  <source_file>.cpp
$ g++ <object1>.o … -o <exec>
```

- **Heterogeneity in the build file generators (CMake, Autotools, home-grown tools that generate Makefiles, etc.)**
  - ⇒ **Use CMake as the meta-build system for all packages!**
- Inconsistent and incompatible usages of CMake (mix of old and modern CMake approaches with different packages)
  - ⇒ Develop and adopt minimal standards for the usage of CMake and the interoperability of modern CMake-based packages.
- Inability to tweak generation of low-level compile and link lines to address difficult cases on some builds and platforms (i.e. **CMake**)
  - ⇒ Continue to develop CMake to allow finer-grained control when needed of the include directory handling and other options.
- Compatible upgrades of dependent packages (i.e. "Dependency hell")
  - ⇒ Adopt Semantic Versioning Standard (semver.org) and policies for transitioning package ecosystems for breaks in compatibility.
- Lack of robust portable test suites to drive packages integration processes (e.g. fragile randomly failing tests).
  - ⇒ Invest in the development and maintenance of test suites!
- Difficulties debugging through all of the different layers down to low-level compiler and linker command-lines
  - ⇒ Standardize on usage of Spack and simplify/streamline usage
  - ⇒ But, support configuration and development with just CMake

Package/Repository Integration: What Not to Do

Project Developers Directly Pulling from the External Repos

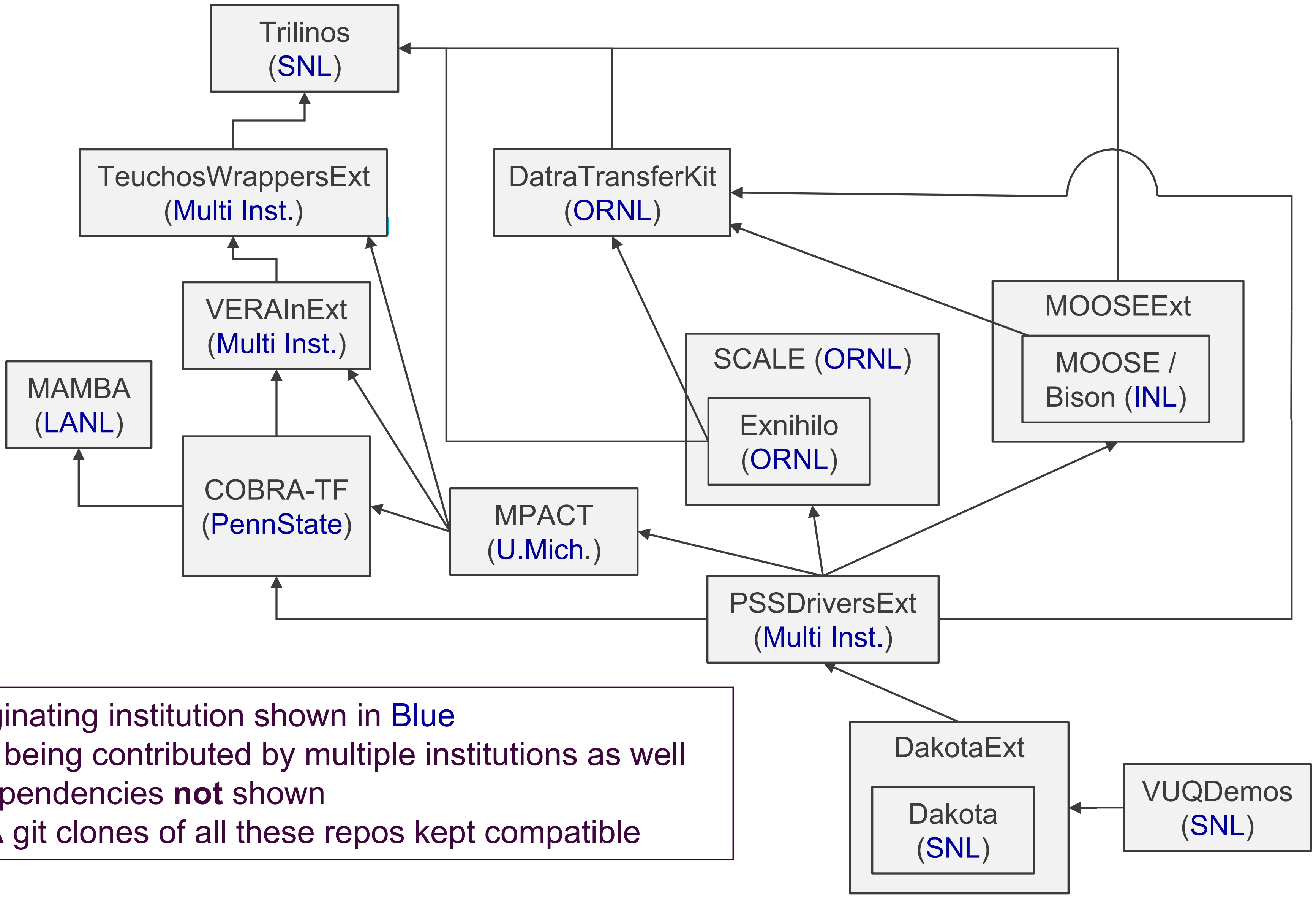




Why is this so bad?

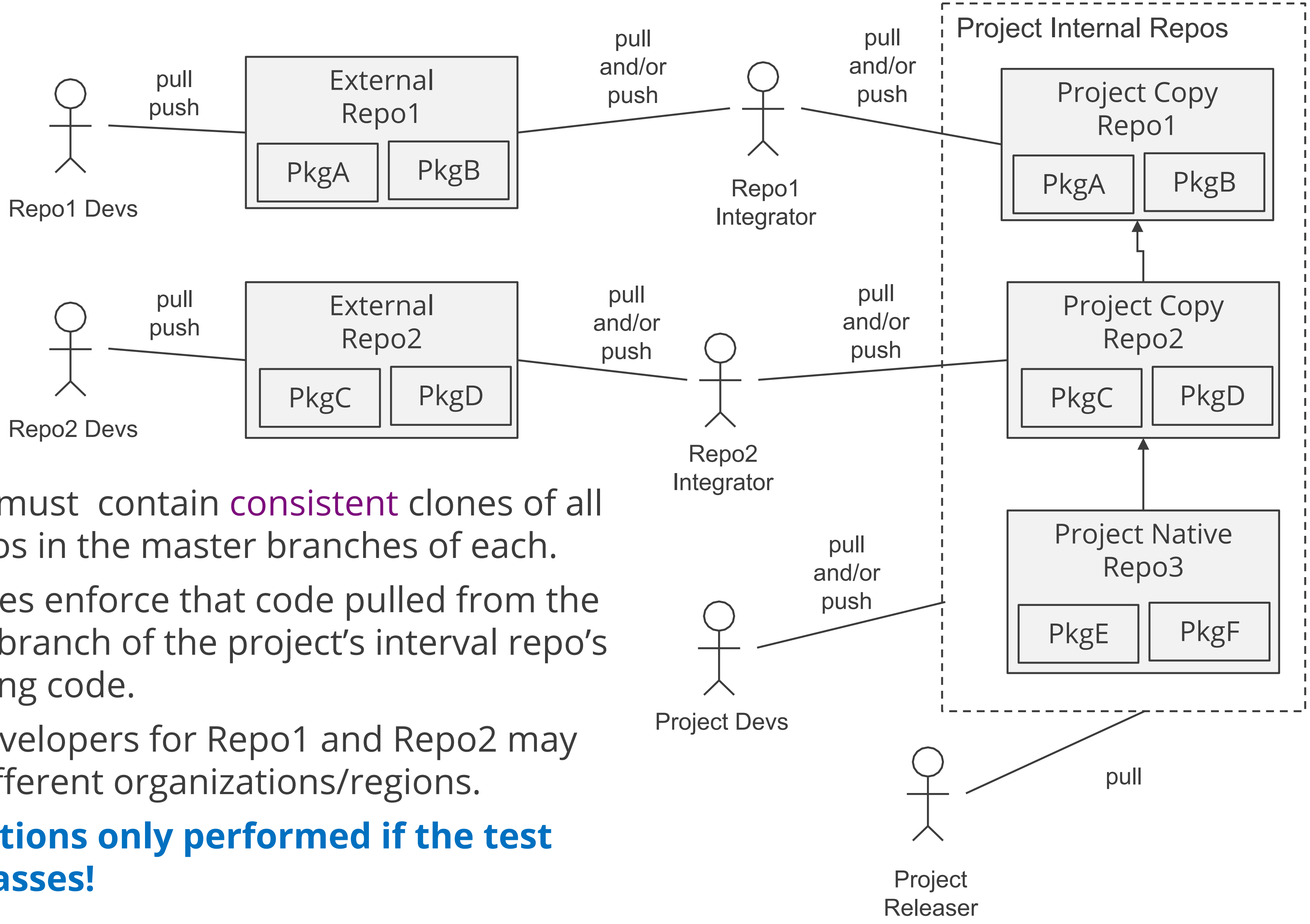
- Lack of test coverage in the external repo's native test suite to cover project's needs.
- External repo developers not testing against the project's code and tests.
- External repo may be broken w.r.t. to the project for long periods of time.
- Project developers frequently pull code that does not even configure or build.
- Broken code frequently interrupting the work of project developers.

Dependences Between Selected CASL VERA Repositories (2016)



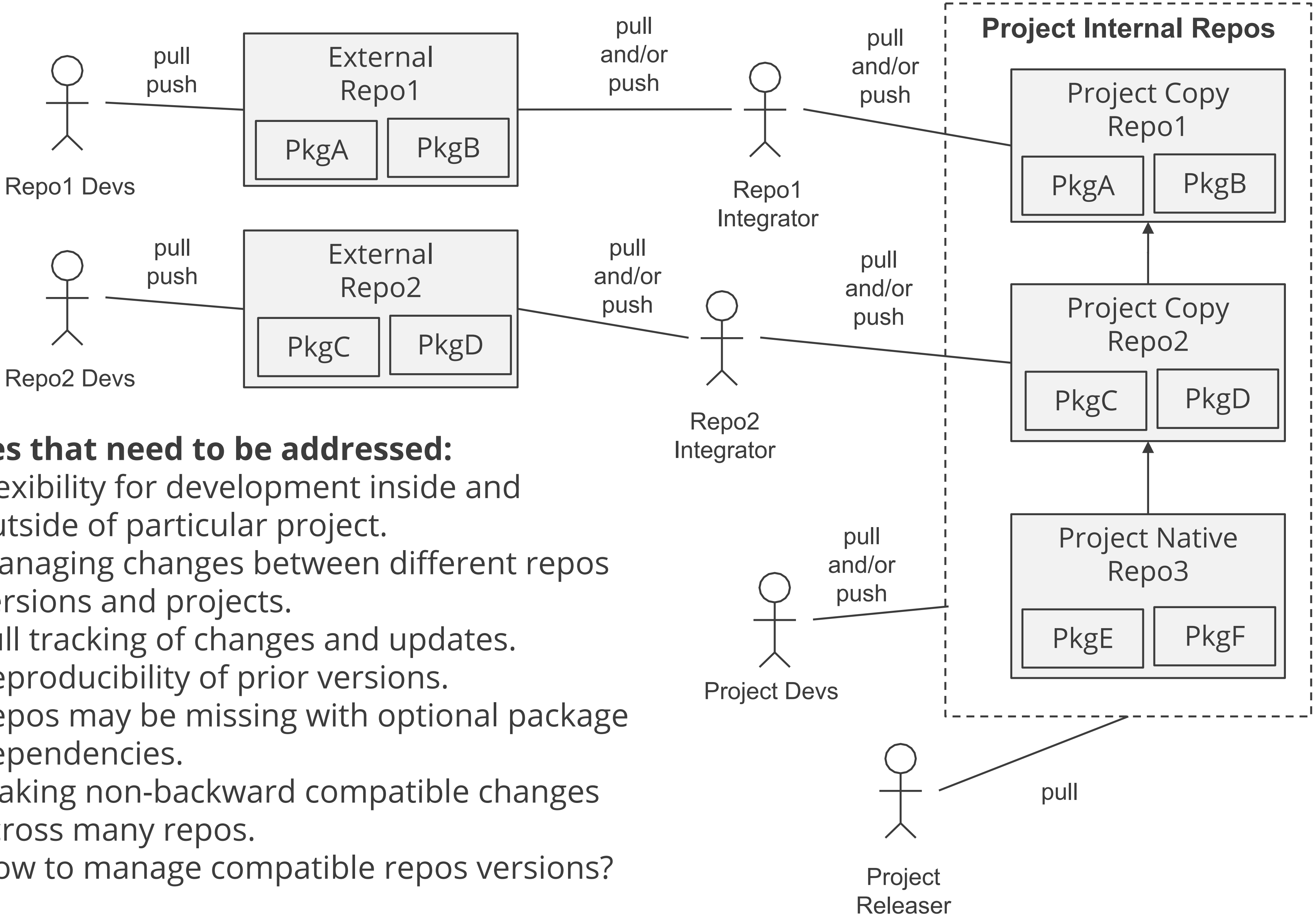
- Primary/originating institution shown in Blue
- Most codes being contributed by multiple institutions as well
- All direct dependencies **not** shown
- Local VERA git clones of all these repos kept compatible

Integration of Packages/Repositories into Project (CASL VERA)



- Project must contain consistent clones of all the repos in the master branches of each.
- Processes enforce that code pulled from the master branch of the project's interval repo's is working code.
- Core developers for Repo1 and Repo2 may be in different organizations/regions.
- **Integrations only performed if the test suite passes!**

Managing Compatible Repos and Repo Versions



Issues that need to be addressed:

- Flexibility for development inside and outside of particular project.
- Managing changes between different repos versions and projects.
- Full tracking of changes and updates.
- Reproducibility of prior versions.
- Repos may be missing with optional package dependencies.
- Making non-backward compatible changes across many repos.
- How to manage compatible repos versions?

Build, Test, and Integration: Final Recommendations

Package dependency handling and build/install/test orchestration
e.g. Spack, homegrown scripts

↓

Meta-build system / Build file generator
e.g. CMake, GNU Autotools, homegrown Makefiles

↓

Build driver with low-level dependencies
e.g.: Makefiles, Google Ninja

↓

Raw compile and link commands

```
$  g++ … -o <object_file>.o -c
   <source_file>.cpp
$  g++ <object1>.o … -o <exec>
```

- **Standardize on the usage of CMake** for generating build files (i.e. Makefiles or Ninja files, it does not matter which)
- **Extend/refine/improve CMake and the usage of CMake** to handle all portability aspects in generating low-level compile and link options
- **Develop and refine minimal standards for usage of CMake** to improve package portability and package interoperability.
- **Train package development teams on modern CMake** and accepted minimal standards for usage of CMake.
- **Fund early CMake porting work and interactions with venders** to new HPC platforms along with early test packages
- **Quickly push out binaries for new CMake releases** to all HPC and support platforms as soon as they are released.
  - Example: We need CMake 3.23 to address backward compatibility for transition of TriBITS and Trilinos to modern CMake.
- **Keep higher-level package build/orchestration software out of the low-level details** of architectures, compiler options etc. (I.e. **Let CMake handle all low-level compiler and linker details**).
  - Example: Avoid drilling low-level compiler options through compiler wrappers to underlying CMake builds.

Questions and Comments?