Software Integration and Development Tools in CMS

David J Lange
Lawrence Livermore National Laboratory

Representing the CMS collaboration

March 21, 2009

CMSSW: The CMS offline software distribution framework

Core framework:
- Event model
- Processing model
- I/O
- Conditions

Reconstruction
- Algorithms
- RECO/AOD data

Calibration
- Algorithms
- Conditions data

Simulation
- Detector model
- Digitization

Analysis
- Algorithms
- Event selection

Physics object definitions

G4

Fast

Tool kit

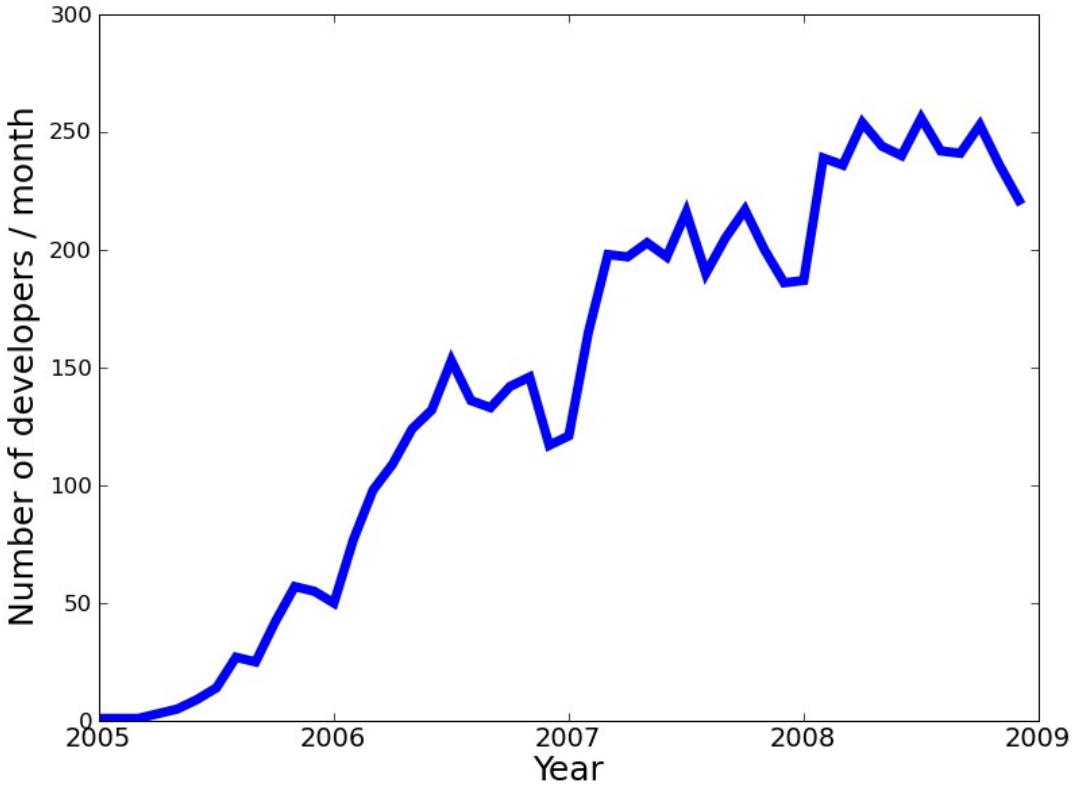
Higgs

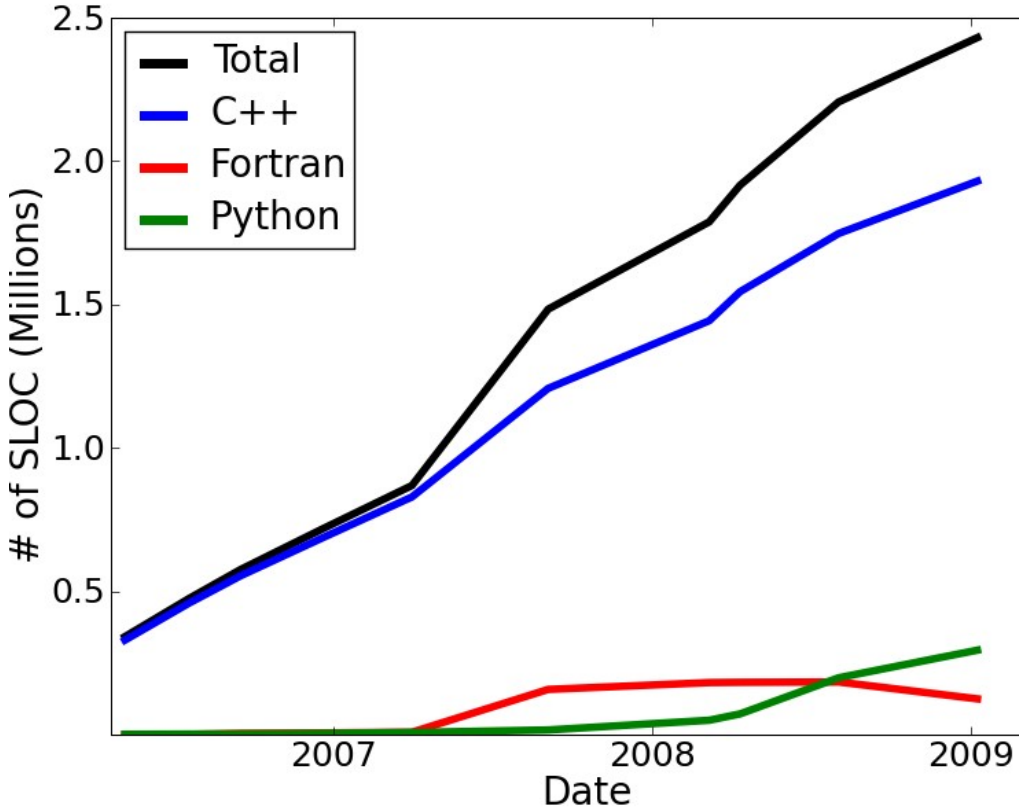
....

Data quality / Code validation

- CMSSW supports a variety of use cases and is the singular repository for offline algorithms in CMS.

We continue to see a consistent increase CMSSW code and user base





- Algorithms written in C++ except for handful of legacy codes (e.g., fortran generator packages)
- Python selected as new job configuration language
- 1500 packages. Interface dependencies between packages comes from both C++ interfaces and python

Frequent CMSSW code release cycles matched to production needs

- Cycles overlap to support a range of production, development and analysis use cases

Pre releases:
1month -3months

- Core development
- Low level code finalized: Basic data and conditions formats
- Integration of new algorithm developments

Production releases:
2months – 18 months

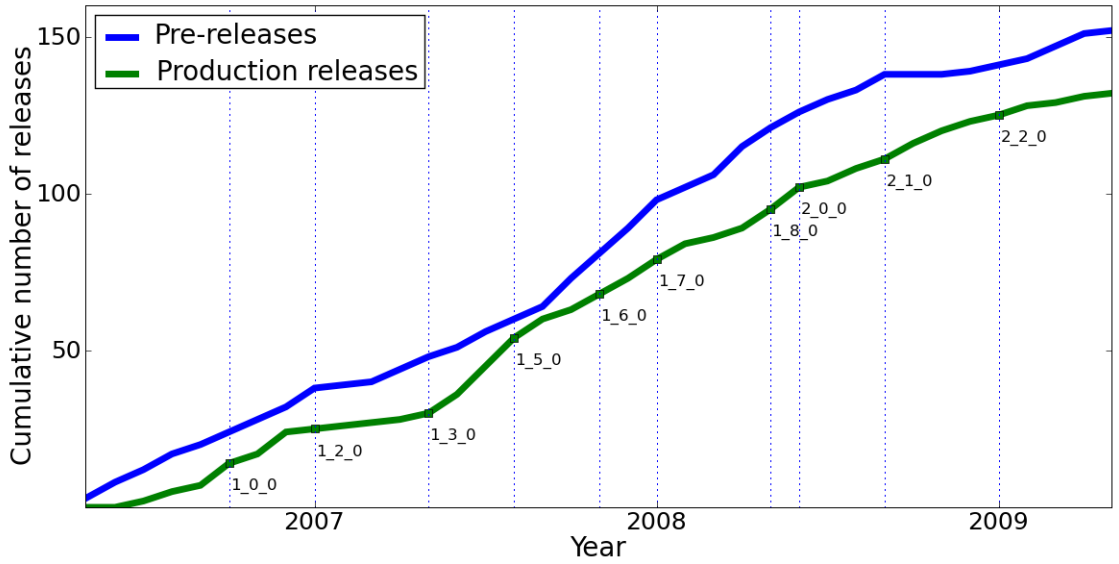
- First production release targeted at specific data taking period or simulation production activity
- Bug fixes and needed functionality extensions added for follow-on data taking/production
- Integration of development to support corresponding analysis work

CMSSW release integration

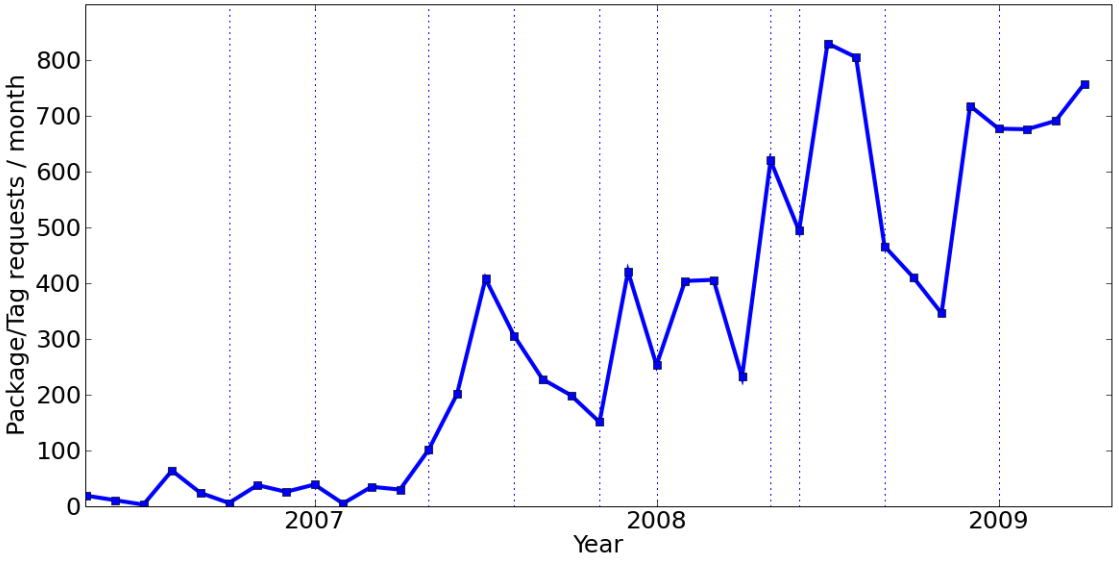
- During both the pre-release and production release cycle, integration builds run daily to provide the primary development platform
  - Available only at CERN and FNAL.
  - Pre-releases are a periodic snapshot of these builds that can be distributed as necessary

- "By approval" system for package integration, even for pre-releases
  - SQL DB w/ CGI "tag collector" package is used to track package+tag requests for each release cycle
  - Software managers for each area take package requests from developers, follow up on any interface changes and pass the request up to the release manager.
  - After several release cycles, we found that we did not have enough people (or energy) to maintain working open integration builds. Developers do not worry enough about packages that depend on their interfaces.

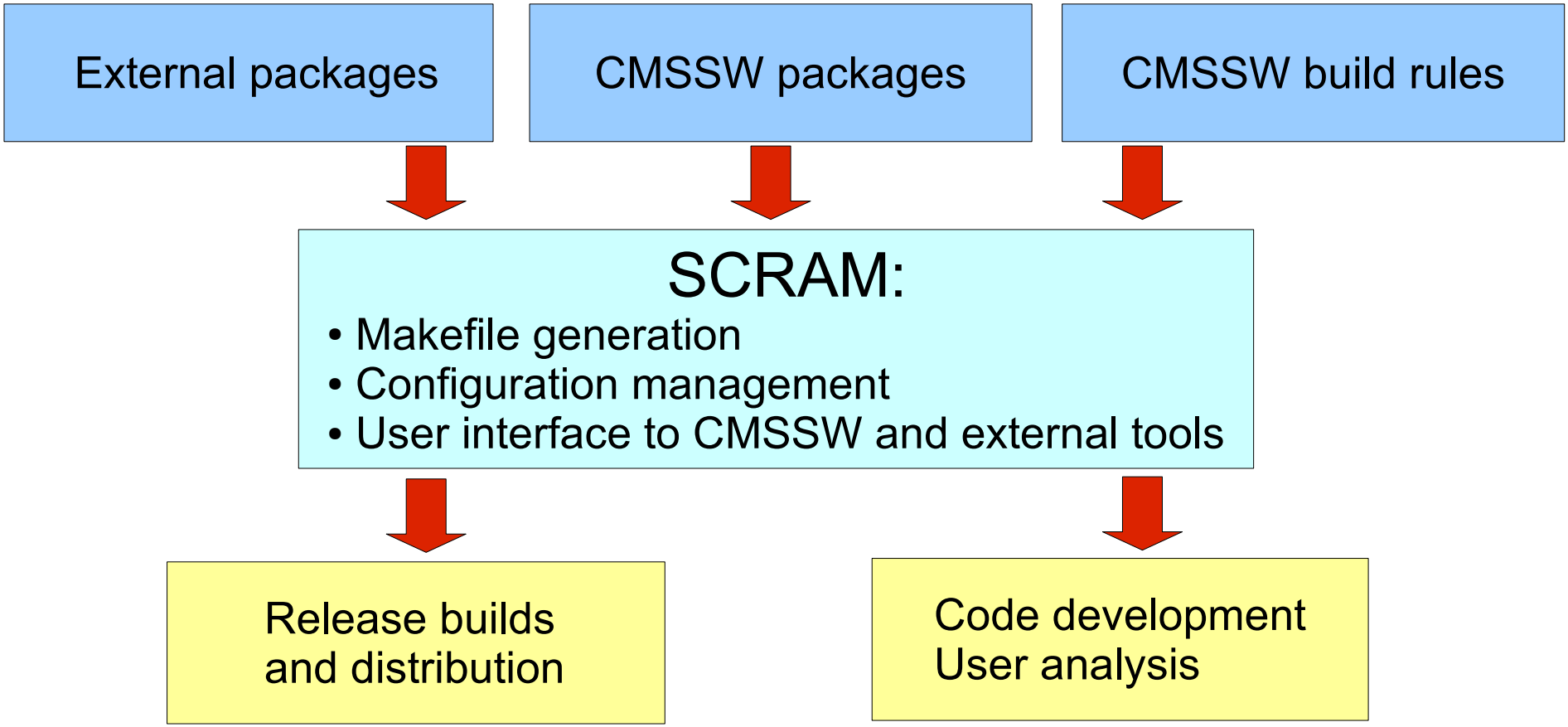
We produce 4 pre-releases and 4 production releases per month on average



- Even so, with closed release cycles and requirement for tag approvals, the release manager job of external package management, tag integration, and release builds is only a part time commitment.



SCRAM is the CMSSW configuration and user interface



We have recently made big improvements to the user experience of using SCRAM.

Recent SCRAM development has improved performance in several areas

Clear scaling in SCRAM were identified early in the development of CMSSW. Instead of starting from scratch, forcing a big change in user interface, we made dramatic improvements in SCRAM performance behind the scenes:

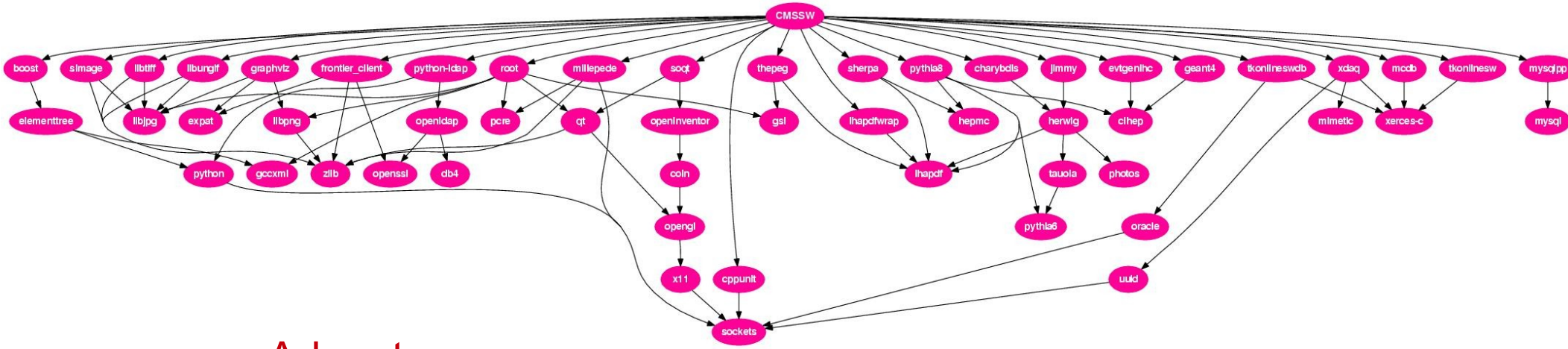
- Time to prepare for compilation:
  - Package dependencies resolved via |make| instead of in perl.
  - Overhead for full release build is reduced from 250 to 35 seconds
- Memory usage: 800 MB reduced to 100 MB
- Cache disk usage: >100 MB reduced to 600 kB
- Portability: Eliminate binary dependencies of SCRAM code

- New features supported:
  - make -j for multicore machines
  - Ability to combine build products into large libraries

Details: See Shahzad Muzaffar's contribution

CMSSW relies on software packages spanning the core OS to HEP developed codes

- We build and distribute the entire set of external packages
  - gcc+binutils up to root/pool/coral and event generators
  - Only a few exceptions: glibc, X11, perl, bash...



CMSSW External Tools Dependencies

Advantages

1. Complete control: Can patch problems at any level quickly
2. Fully consistent software stack
3. Support for non-SLC4 linux
4. Integrated build and distribution
5. Reproducibility

Disadvantages

1. Must define and maintain build rules for each software component

We redeveloped and simplified our release build+distribution system

Requirements:

- Packaging tool: **RPM**
- Software distribution tool: **apt-get**

Rely on standard tools when available

- Helper scripts to resolve dependency tree, fetch sources and launch the build: **PKGTOOLS**

Ability to patch urgent problems and undesired features

- Build rules for each external package (and for CMSSW): **CMSDIST**

Provides full control over what goes into the software release

Combined build/distribution system identifies hidden dependency problems. Installation problems identified at build time via test installation.

CHEP 2007 coincided with the end of the infrastructure development for PKGTOOLS

G. Eulisse: CHEP 2007

Today it is reality:

Fully deployed Dec07

Done

Done

Not used

ia32/amd64 and gcc345/gcc412/gcc432

~6 hours/8cpus

Future

- Next release of PKGTOOLS coming soon:
  - Focused to make the release integrator happy
  - will simplify / integrate / fortify "configuration management" phase.
  - parallel build of non dependent packages
  - distcc
  - more platforms
- Early tests (with a quad core Xeon) show that we can seriously aim to build everything from scratch to CMSSW, in less than 2 hours (yes, with **ALL** the externals).

Lawrence Livermore National Laboratory

CHEP09: cmsBuild system fully deployed

What have we learned?

- External configuration management and the mechanics of building releases are now very easy.
  - Single source of external configuration greatly simplified process of deploying external package updates
  - We are past the bootstrapping period of developing external package build recipes and working bugs out of the configuration
- Parallel builds on single 8-core machine very successful.
  Tag integration process dominates release build time

- Distribution to non-Scientific Linux 4 users: Discovery of remaining OS dependencies is not automated. Once per linux platform effort needed before distribution to that OS will work
- CMSSW rpm pushes RPM limits:
  - Severe memory scaling issue limited # of installed releases [**Solved**]
  - >1 hour to build CMSSW rpm once build is completed

We also support a few subsets of the CMSSW release

- "Partial" builds each support a particular use case where distribution size is important:
  - **Online distribution for high-level trigger farm**
    - Provide algorithms used in trigger.
    - Omit components such as the simulation
  - **FWLite distribution for user analysis**
    - AOD level data formats and core framework
    - Supports root macro analysis on existing data. Serves as light weight alternative to full framework distribution.

- For each partial build, we
  1. User defined "application" set of CMSSW packages
  2. Mine dependency info to find full set of required CMSSW packages
  3. Determine corresponding required set of external dependencies
  4. Build corresponding RPM following the same procedure as for CMSSW

Details: See Natalia Ratnikova's contribution

“Patch” releases to make quick turn around repairs for problems in production apps

Even a few hours is too long to wait to fix problems that prevent data taking. Create “patch releases”, built on top of production releases for critical bug fixes.

- Patch is explicitly distinct and tracked separately from underlying release. Thus, code used for data processing is clearly identifiable.
- Reuse sources, libraries, etc from production build. Add new versions of the package(s) to be repaired.
- Mine dependency information to identify any dependencies that must be rebuilt due to interface changes.
- Resulting “release” functions just like a full production release for both running the production applications and user development/analysis.
- Time to build is limited by compilation of repaired packages. Small problems fixed, packaged into rpm and installed in ~10-20 minutes.
  - Instead limited by availability of the bug fix itself.

Status: Development complete, however production testing and verification still to be done.

Conclusions and future work

- Production release build system and code development environment in CMS has undergone a significant redevelopment. Major functionality now stable and ready for first LHC data.
- Developer and code base continue to grow
- Refocused on additional features, tools to improve CMS code quality and user code development/analysis experience

In particular:

- Raise awareness of software dependencies and performance
  - With so many small packages, it is easy to end up with a jumbled dependency tree
  - Integrate existing tools into common practice
- Minimize expert resources needed for release management as CMSSW moves into a more stable project
- Increase support for other operating systems (e.g., OS/X)
- Identify and mitigate potential scaling issues