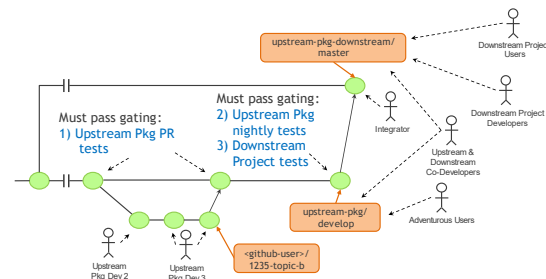


Some Challenges and Opportunities in Software Engineering in Computational Science and Engineering

Roscoe A. Bartlett
Department 1424
Software Engineering and Research

April 12, 2023



Dr. Roscoe A. Bartlett: Career Overview

1996 – 2001 : Ph.D. Chemical Engineering, CMU

- Large-scale optimization algorithms research
- Successive Quadratic Programming

2001 - 2011: Sandia National Laboratories (SNL)

- Continued Optimization R&D
- Sensitivities and Inverse problems
- Abstract numerical algorithms (Thyra)
- Build and test systems (TriBITS)
- Almost continuous integration: Charon, Sierra
- Lean/Agile advocate

(<https://bartlettroscoe.github.io>, 2008+)

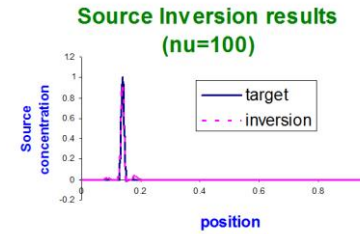
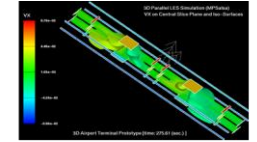
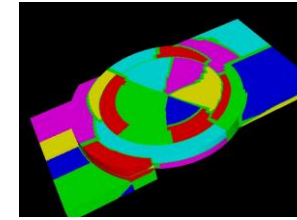
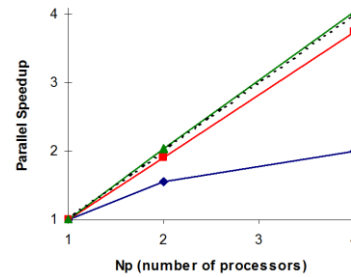
2011 – 2016: Oak Ridge National Laboratory (ORNL)

- CASL (Nuclear Reactor Modeling DOE HUB)
 - SE processes and infrastructure

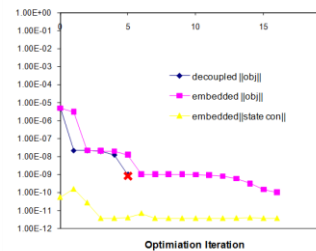
2016 – Present: Sandia National Laboratories (SNL)

- Computational Tools and Development Environments
- Productivity and Sustainability for the Exascale Computing Project (ECP)
- Large-scale build and test systems for CSE package ecosystems (TriBITS)

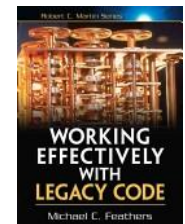
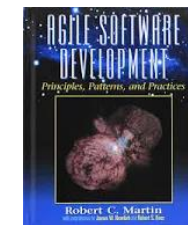
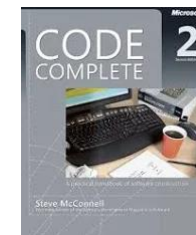
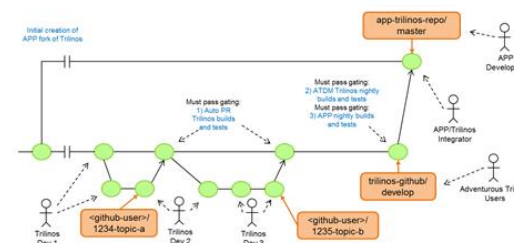
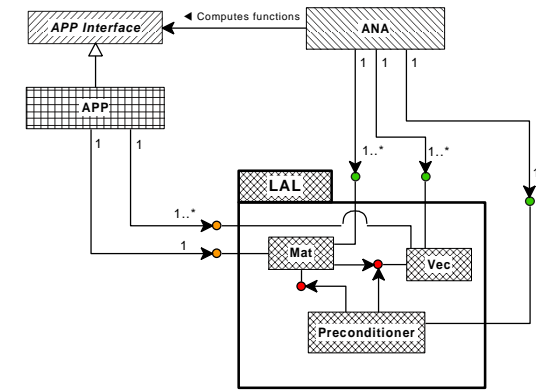
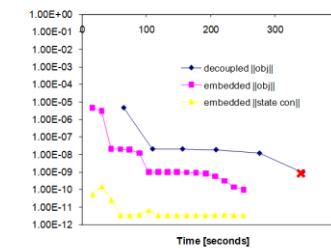
Parallel Scalability of MOOCHO



Decoupled Finite Diff. vs. Embedded Finite Diff.



Decoupled Finite Diff. vs. Embedded Finite Diff.



Software Engineering Reading List

This page lists some books about various aspects of software development and software engineering that I (Roscoe Bartlett) have read over the years, especially during a particularly intense period in the 2007-2008 time frame. I provide a brief overview of the content of each covered book and why I think it is useful (or essential) to read the book and what to look for in each book. These books (along with the personal application in the following years of the principles and practices presented in them) represent my basic education in software engineering. Hopefully these books will help others the way they helped me.

Outline:

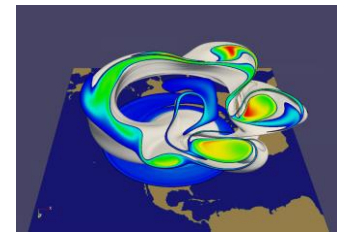
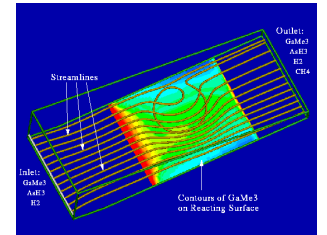
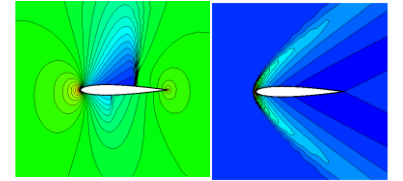
- Background
- A) General Software Engineering Books
 - A.1) Most Recommended Software Engineering Books
 - A.2) Recommended Software Engineering Books
- B) C++ Books
 - B.1) Most Recommended C++ Books
 - B.2) Recommended Books on the Fine Details of C++
 - B.3) Recommended Books on the History of C++ and why it has its current form

Site: <https://bartlettroscoe.github.io/reading-list/>

Computational Science and Engineering (CSE) Overview

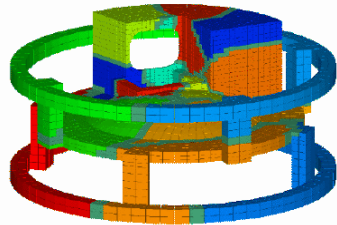
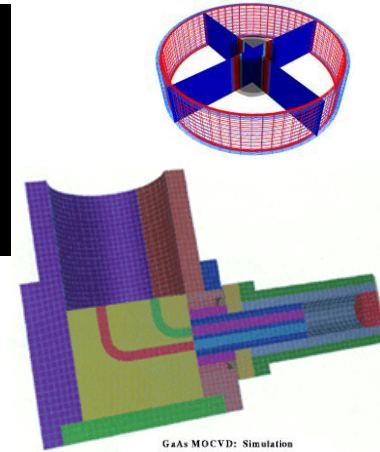
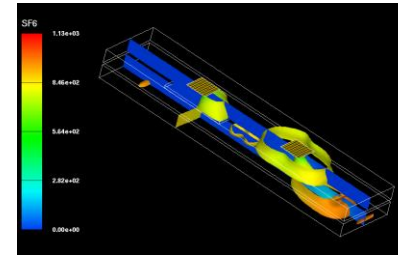
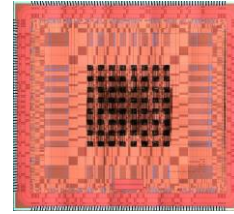
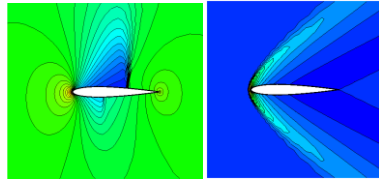
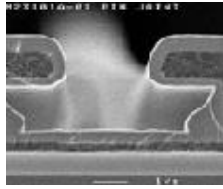


- **Computational science and engineering (CSE)** is an interdisciplinary field that uses computational methods to solve scientific and engineering problems.
- CSE draws on knowledge from mathematics, computer science, physics, chemistry, engineering, climate and other fields.
- CSE is used to solve a wide range of problems, including:
 - Modeling the behavior of complex (physical) systems
 - Designing new products and processes
 - Simulating and predicting physical phenomena
 - Improving the efficiency of existing systems
 - Making predictions about the future (e.g. climate)
- CSE is essential for solving many of the world's most pressing problems, such as climate change, energy security, and healthcare.



EXASCALE
COMPUTING
PROJECT

Computational Sciences: PDEs and More ...



Chemically reacting flows

Materials modeling

Climate modeling

MEMS modeling

Combustion

Seismic imaging

Compressible flows

Shock and multiphysics

Computational biology

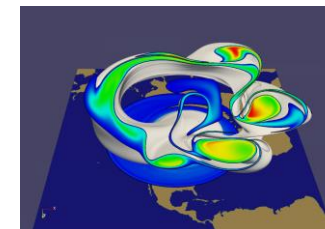
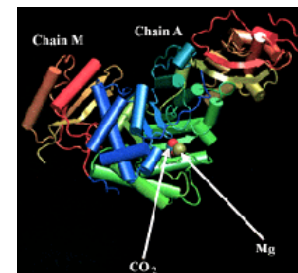
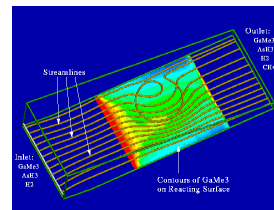
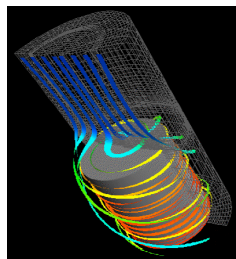
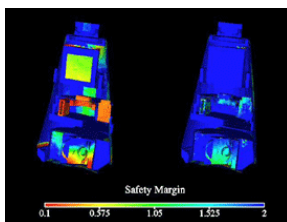
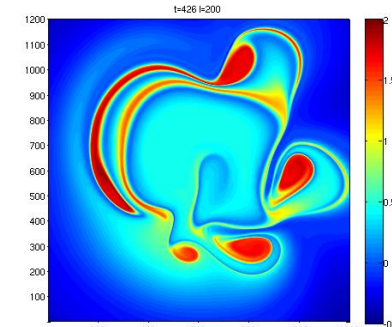
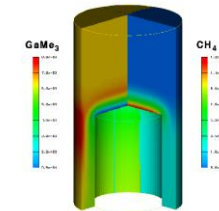
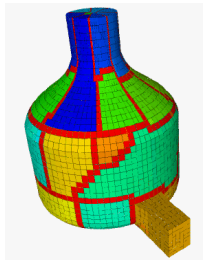
Structural dynamics

Circuit modeling

Heat transfer

Inhomogeneous fluids

Network modeling



High Performance Computing (HPC) Overview

- High performance computing (HPC) is the use of powerful computers to solve complex computing problems that are beyond the capabilities of ordinary personal computers.
- HPC systems are typically made up of hundreds or even thousands of individual nodes that are linked together with a high bandwidth interconnect.
- HPC systems are very expensive to build and maintain.
- HPC systems require a high level of expertise to operate and manage.
- HPC systems generate a lot of heat, which can be a challenge to manage.
- Current thrust in HPC is reaching exascale computations (i.e. 10^{18} floating-point operations per second)
- Modern exascale HPC systems use smaller numbers of beefier hybrid computing nodes involving a CPU and a number of accelerator units (i.e. NVIDIA GPUs, AMD GPUs, Intel Max GPUs)

Frontier

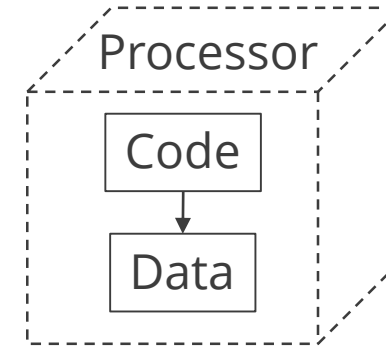


Active	Deployment: Sep. 2021 Completion: May 2022
Operators	Oak Ridge National Laboratory and U.S. Department of Energy
Location	Oak Ridge Leadership Computing Facility
Power	21 MW
Operating system	HPE Cray OS
Space	680 m ² (7,300 sq ft)
Speed	1.102 exaFLOPS (Rmax) / 1.685 exaFLOPS (Rpeak) ^[1]
Cost	US\$600 million (estimated cost)
Purpose	Scientific research and development
Website	www.olcf.ornl.gov/frontier/

Classic Environments for CSE/HPC

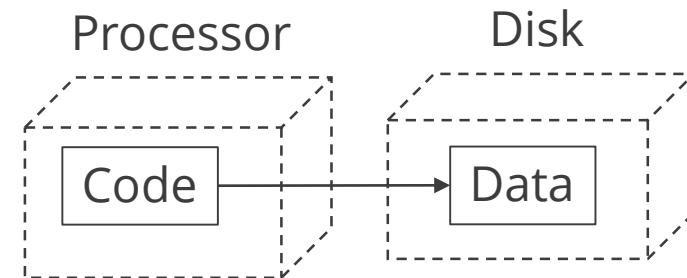
Serial / SMP (symmetric multi-processor)

- All data stored in RAM in a single local process



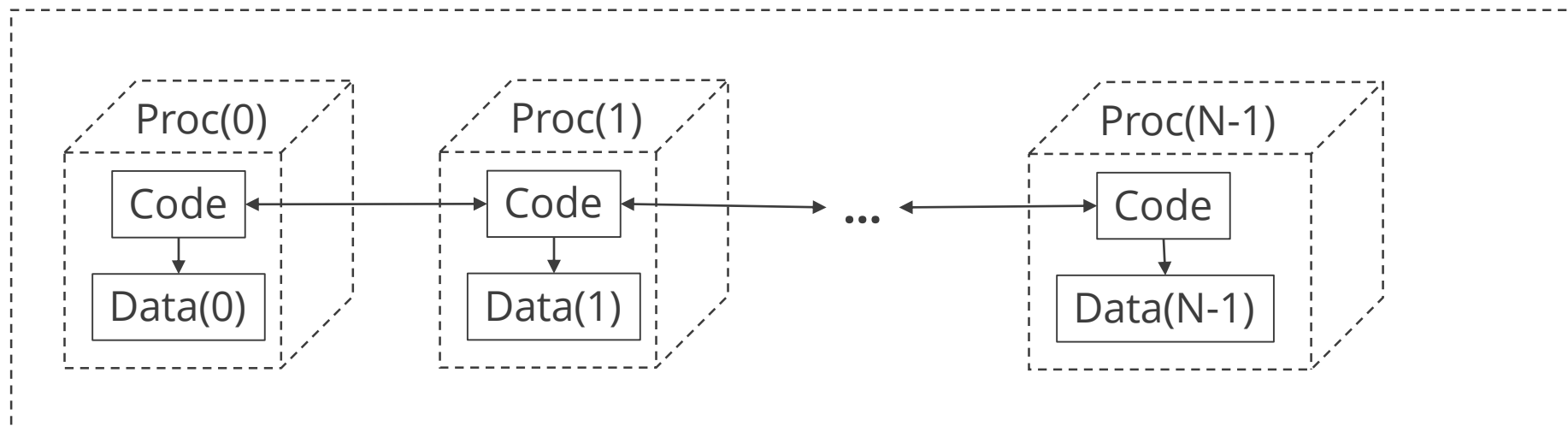
Out of Core

- Data stored in file(s) (too big to fit in RAM)



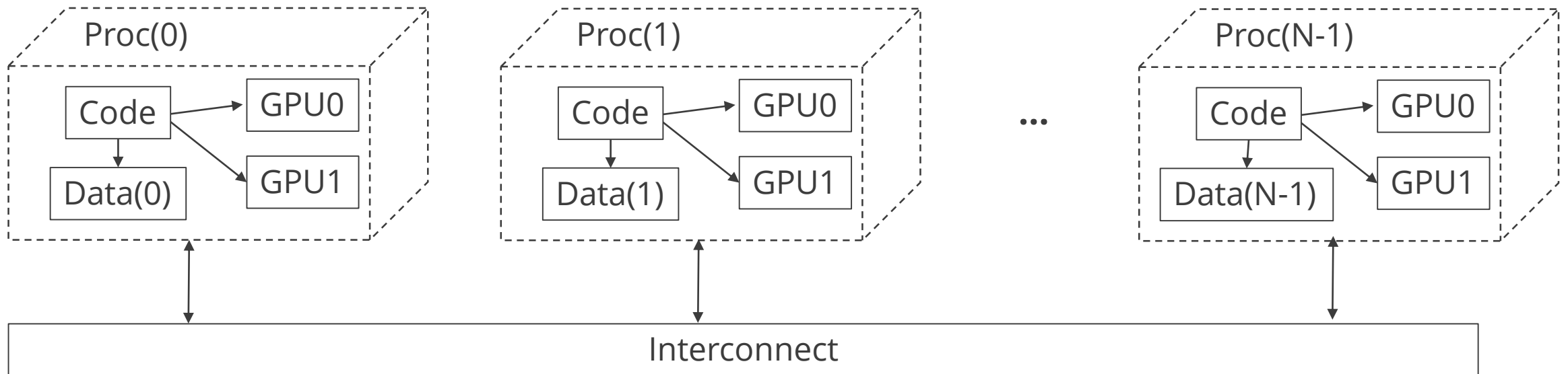
SPMD (Single program multiple data)

- Same code on each processor but different data: **Message Passing Interface (MPI)**



MPI/X

- **Course-grained parallelism:** MPI
- **Fine-grained on-node/GPU parallelism:** CUDA (NVIDIA GPUs), HIP (AMD GPUs)



Complex multi-level parallel algorithms and software

=>

Lots of complex bugs leading to random errors



Software Engineering (SE) Challenges in CSE



- Increasing complexity of hardware and programming HPC environments.
- Needing to develop and integrate increasingly more complex algorithms from specialized domains to continue progress solving CSE problems
- CSE developers are domain experts (classic engineering, applied math, physics, etc.) and lack basic SE knowledge and skills
- Needing to update CSE/HPC software for new algorithms and new programming environments
 - => Often giving up and starting from scratch (i.e. “Things you should never do: Part 1”)
- Difficultly hiring and retaining skilled software engineers to aid CSE domain expert developers
 - => Agile best practices
 - => Software modeling and design (e.g. UML & Design Patterns)



Some of the Challenges in Modern Exascale CSE/HPC Systems

- Running on bleeding-edge hardware and system software
- Defects in system software (e.g. compilers, MPI) slipping through system testing and instead being detected in downstream testing of CSE/HPC Codes
- Fairly frequent systems upgrades to newer bleeding-edge system software
- Setting up builds on new platforms requires painful debugging of failing automated tests to distinguish latent Code defects from system hardware/software defects

Advancing scientific productivity through better scientific software

Science through computing is only as good as the software that produces it.



- 1 Customize and curate methodologies**
 - Target scientific software productivity and sustainability
 - Use workflow for best practices content development

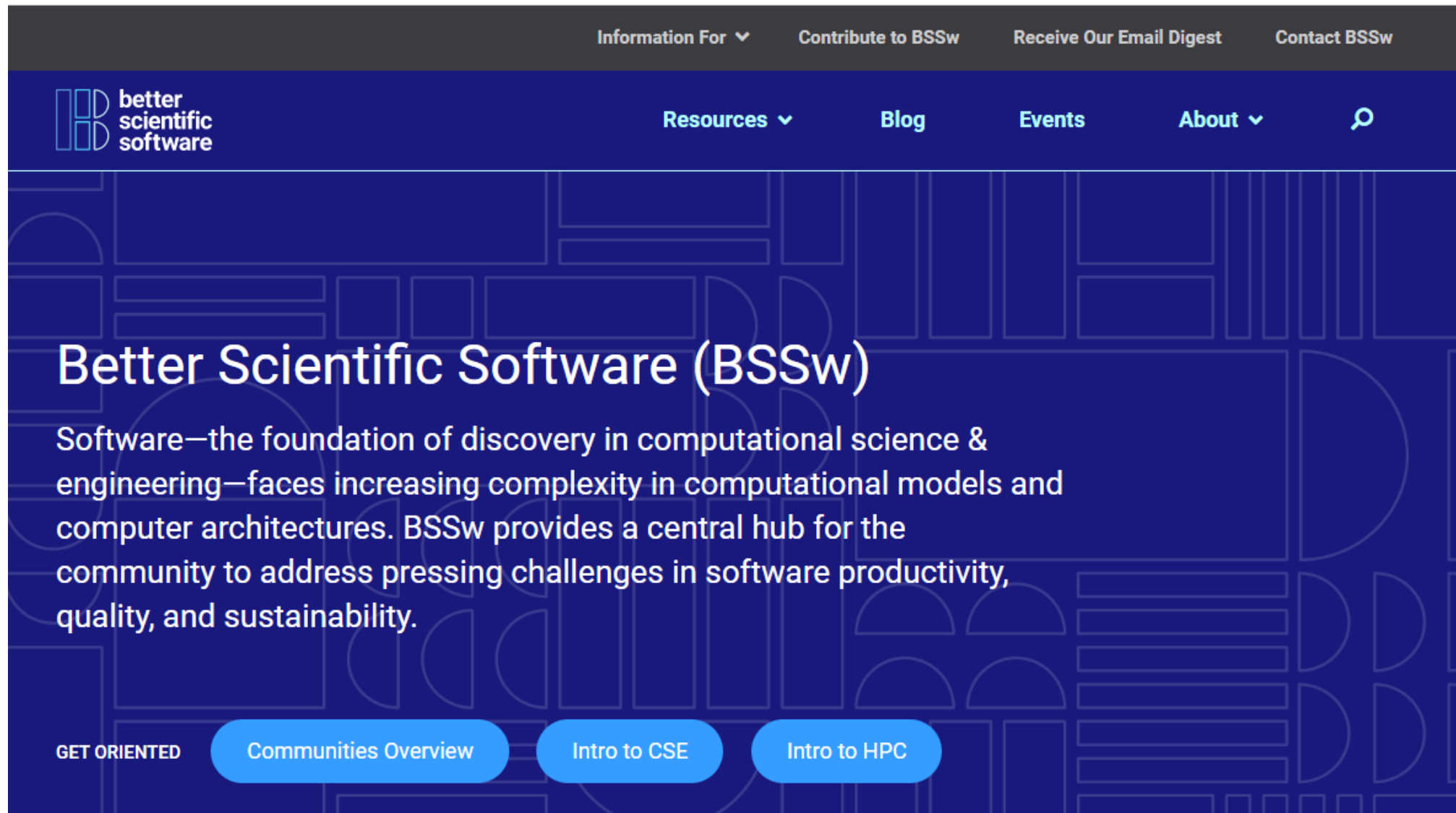
- 2 Incrementally and iteratively improve software practices**
 - Determine high-priority topics for improvement and track progress
 - *Productivity and Sustainability Improvement Planning (PSIP)*



- 3 Establish software communities**
 - Determine community policies to improve software quality and compatibility
 - Create Software Development Kits (SDKs) to facilitate the combined use of complementary libraries and tools

- 4 Engage in community outreach**
 - Broad community partnerships
 - Collaboration with computing facilities
 - Webinars, tutorials, events
 - *WhatIs* and *HowTo* docs
 - Better Scientific Software site (<https://bssw.io>)

Better Scientific Software (<https://bssw.io>)



The screenshot shows the homepage of the Better Scientific Software (BSSw) website. The page has a dark blue background with a pattern of white geometric shapes. At the top, there is a navigation bar with the following items: "Information For" with a dropdown arrow, "Contribute to BSSw", "Receive Our Email Digest", and "Contact BSSw". Below this is a secondary navigation bar with the BSSw logo on the left, followed by "Resources" with a dropdown arrow, "Blog", "Events", "About" with a dropdown arrow, and a search icon. The main content area features a large white heading "Better Scientific Software (BSSw)" and a paragraph of text: "Software—the foundation of discovery in computational science & engineering—faces increasing complexity in computational models and computer architectures. BSSw provides a central hub for the community to address pressing challenges in software productivity, quality, and sustainability." At the bottom, there is a "GET ORIENTED" section with three blue buttons: "Communities Overview", "Intro to CSE", and "Intro to HPC".



Better Scientific Software (Bssw.io): Resources

<p>Better Planning</p> <ul style="list-style-type: none">› Software Process Improvement› Software Engineering› Requirements› Design› Software Interoperability› Software Sustainability	<p>Better Development</p> <ul style="list-style-type: none">› Documentation› Configuration and Builds› Revision Control› Release and Deployment› Issue Tracking› Programming Languages› Development Tools› Refactoring	<p>Better Performance</p> <ul style="list-style-type: none">› High-Performance Computing (HPC)› Performance at Leadership Computing Facilities› Performance Portability› Cloud Computing› Big Data
<p>Better Reliability</p> <ul style="list-style-type: none">› Peer Code Review› Testing› Continuous Integration Testing› Reproducibility› Debugging	<p>Better Collaboration</p> <ul style="list-style-type: none">› Projects and Organizations› Strategies for More Effective Teams› Inclusivity› Funding Sources and Programs› Software Publishing and Citation› Licensing› Discussion and Question Sites› Conferences and Workshops	<p>Better Skills</p> <ul style="list-style-type: none">› Online Learning› In-Person Learning› Personal Productivity and Sustainability





Agile Principles and Technical Practices

Agile: Defined



- **Agile Software Engineering Methods:**

- Agile Manifesto (2001) (Capital 'A' in Agile)
- Founded on long standing wisdom in SE community (55+ years)
- Push back against heavy plan-driven methods in 1980s & 1990s (e.g. CMM(I))
- Focus on incremental design, development, and delivery (i.e. software life-cycle)
- Close customer focus and interaction and constant feedback
- Example methods: SCRUM, XP (extreme programming)
- Has become a dominate software engineering approach (example [IBM](#))

References: <https://bartlettroscoe.github.io/reading-list/>

Principles for Agile Technical Practices



- **Agile Design:** Reusable software is best designed and developed by incrementally attempting to reuse it with new clients and incrementally redesigning and refactoring the software as needed keeping it simple.
 - Technical debt in the code is managed through continuous incremental (re)design and refactoring.
- **Agile Quality:** High quality defect-free software is most effectively developed by not putting defects into the software in the first place.
 - High quality software is best developed collaboratively (e.g. pair programming and code reviews).
 - Software is fully verified before it is even written (i.e. Test Driven Development (TDD)).
 - High quality software is developed in small increments and with sufficient testing in between sets of changes.
- **Agile Integration:** Software needs to be integrated early and often
- **Agile Delivery:** Software should be delivered to real (or as real as we can make them) customers in short (fixed) intervals.

References: <https://bartlettroscoe.github.io/reading-list/>

Key Agile Technical Practices

Unit Testing

- Re-build fast and run fast
- Localize errors
- Well supports continuous integration, TDD, etc.

Integration and System-Level Testing

- Tests on full system or larger integrated pieces
- Slower to build and run
- Generally does not well support CI or TDD.

Test Driven Development (TDD)

- Write a compiling but failing test and verify that it fails
- Add/change minimal code until the test passes (keeping all other tests passing)
- Refactor code to make more clear and remove duplication
- Repeat (in many back-to-back cycles)

Continuous Integration

- Integrating software in small batches of changes frequently
- Most development on primary 'master' branch

Incremental Structured Refactoring

- Make changes to restructure code without changing behavior (or performance, usually)
- Separate refactoring changes from changes to change behavior

Agile-Emergent Design

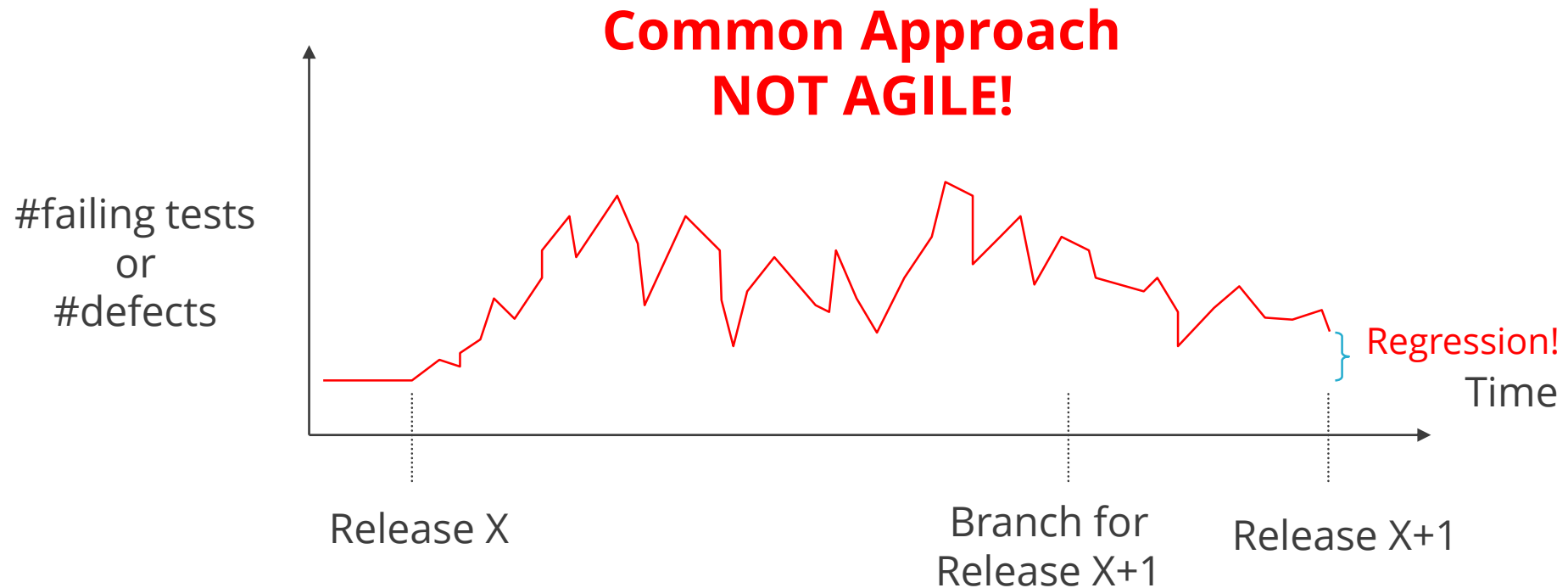
- Keep the design simple and obvious for the current set of features (not some imagined set of future features)
- Continuously refactor code as design changes to match current feature set

Integration Tests

Courser-grained “Integration tests” can be relatively fast to write but take slightly longer to rebuild and run than pure “unit tests” but cover behavior fairly well but don’t localize errors as well as “unit tests”.

These are real skills
that take time and
practice to acquire!

Common Approach: Development Instability



Problems

- Cost of fixing defects increases the longer they exist in the code
- Difficult to sustain development productivity
- Broken code begets broken code (i.e. broken window phenomenon)
- Long time between branch and release
 - Difficult to merge changes back into main development branch
 - Temptation to add "features" to the release branch before a release
- Nearly impossible to consider more frequent development integration models
- High risk of creating a regression

Agile Approach: Development Stability



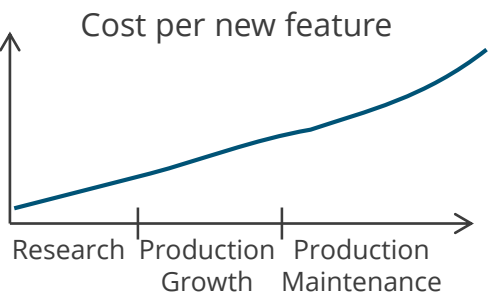
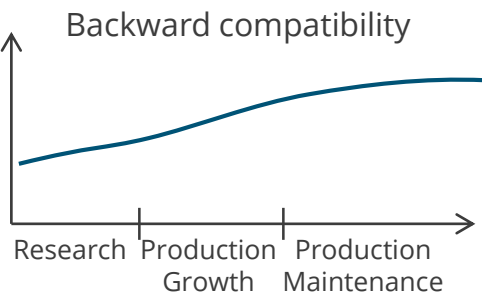
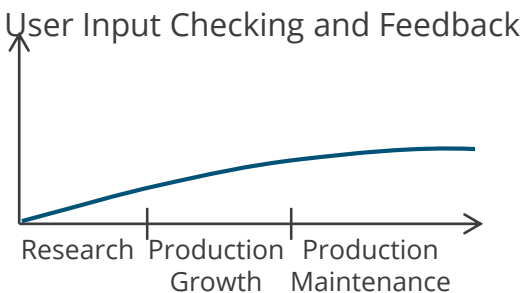
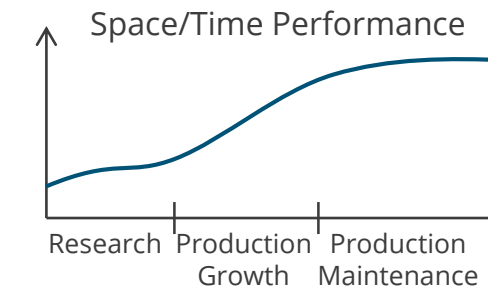
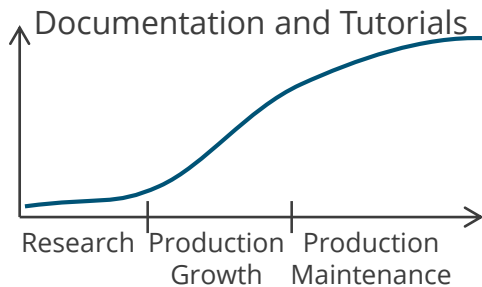
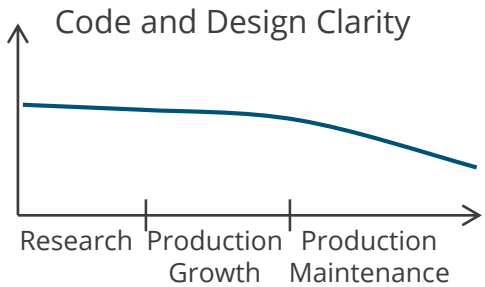
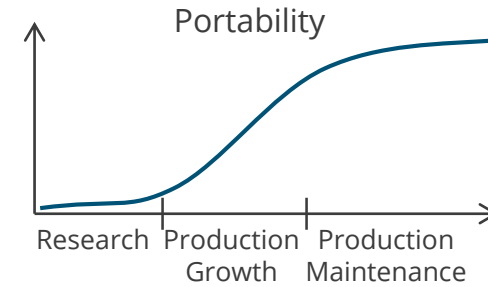
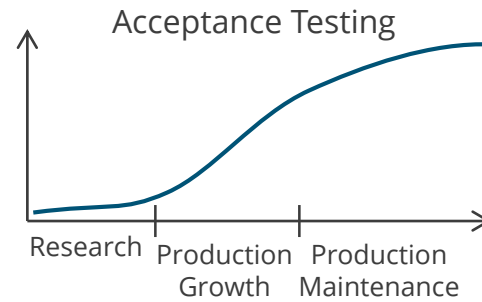
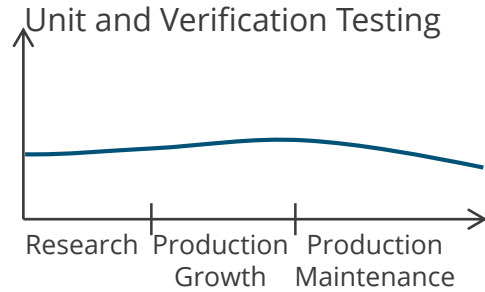
The Agile way!



Advantages

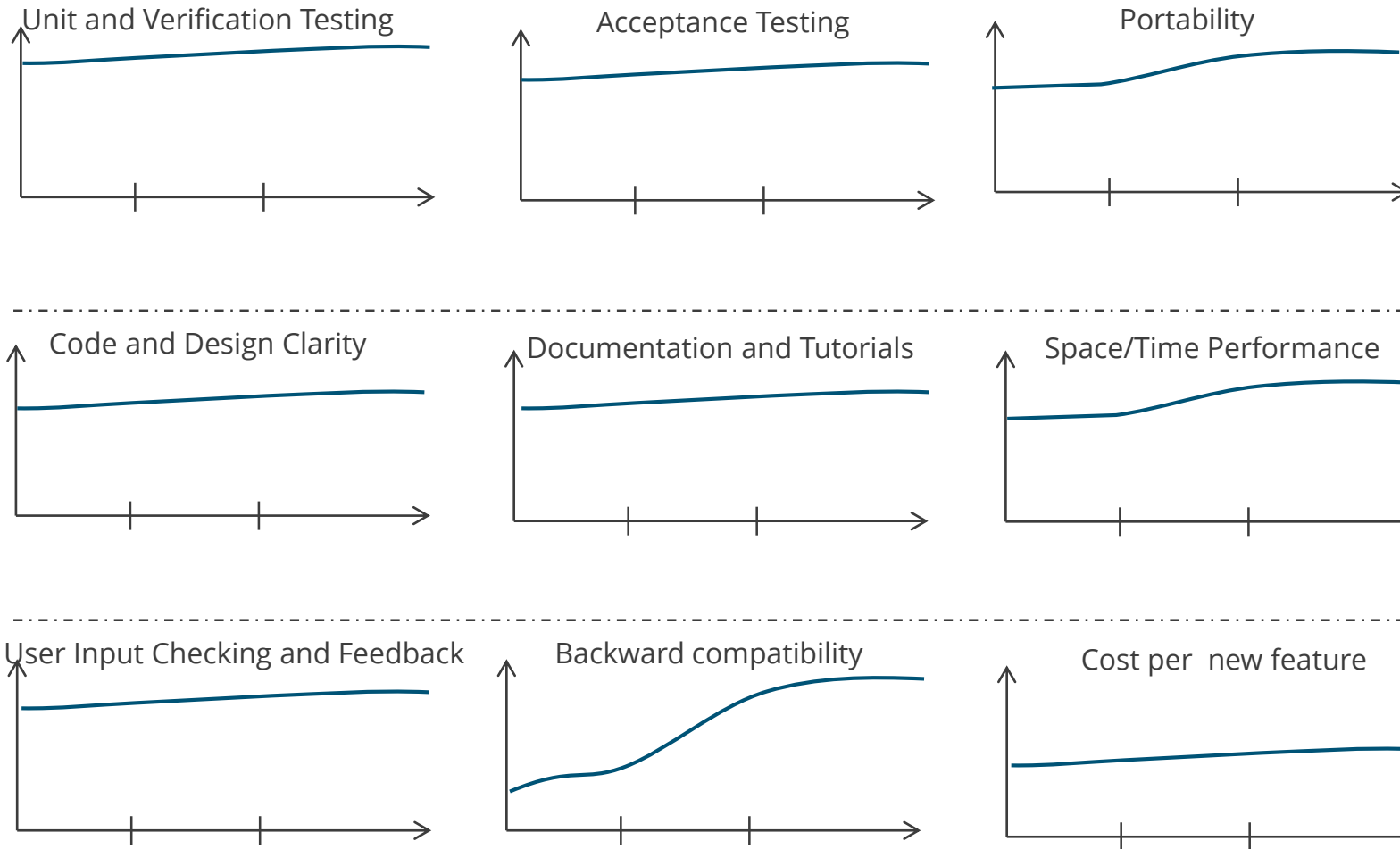
- Defects are kept out of the code in the first place
- Code is kept in a near releasable state at all times
- Shorten time needed to put out a release
- Allow for more frequent releases
- Reduce risk of creating regressions
- Decrease overall development cost (Fundamental Principle of Software Quality)
- Allows many options in how to do development integration models

Typical (i.e. non-Lean/Agile) CSE Lifecycle



Time

Pure Lean/Agile Lifecycle: "Done Done"



Time

Why Software Practices Matter for Research Software?



- Even research software (only written by and run by a researcher) needs to have a base level of SE practices/quality to support basic research!
- **Example:** A simple software defect in protein configuration/folding research code [1]
- **The researcher:** A respected Presidential Early Career winner
- **The defect:**
 - An array index failure (i.e. verification failure)
 - leading to incorrect protein folding (validation failure)
- **Direct impact:**
 - Several papers were published with incorrect results & conclusions
- **Indirect impact:**
 - Papers from other competing researchers with different results were rejected
 - Proposals from other researchers with different results were turned down
- **Final outcome:** Defect was finally found and author retracted five papers!
 - **But: Damage to the research community not completely erased!**
- [1] Miller, G. "A Scientist's Worst Nightmare: Software Problem Leads to Five Retractions", Science, vol 314, number 5807, Dec. 2006, pages 1856-1857

And Software Development
Productivity Matters CSE
Researchers Too!

Working Effectively with Legacy Software

Where the rubber meets the road with
“Things you should never do: Part 1”

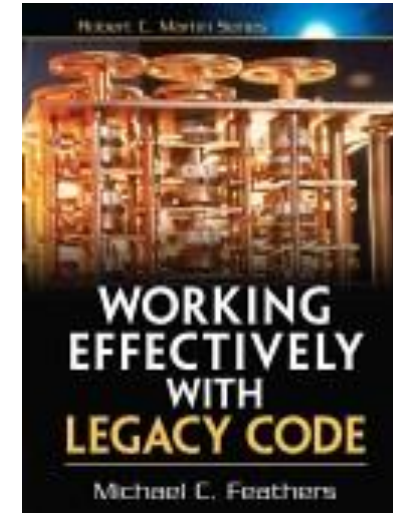
Definition of Legacy Code and Changes



Legacy Code = Code Without Tests

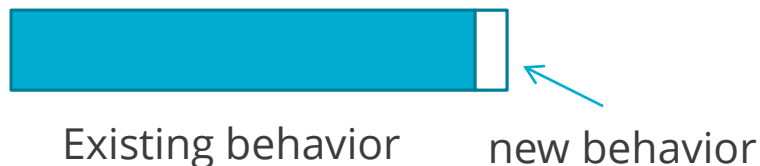
“Code without tests is bad code. It does not matter how well written it is; it doesn’t matter how pretty or object-oriented or well-encapsulated it is. With tests, we can change the behavior of our code quickly and verifiably. Without them, we really don’t know if our code is getting better or worse.”

Source: M. Feathers. Preface of “Working Effectively with Legacy Code”



Reasons to change code:

- Adding a Feature
- Fixing a Bug
- Improving the Design (i.e. Refactoring)
- Optimizing Resource Usage



Preserving behavior under change:

“Behavior is the most important thing about software. It is what users depend on. Users like it when we add behavior (provided it is what they really wanted), but if we change or remove behavior they depend on (introduce bugs), they stop trusting us.”

Source: M. Feathers. Chapter 1 of “Working Effectively with Legacy Code”

Legacy Software Change Algorithm: Details



Abbreviated Legacy Software Change Algorithm:

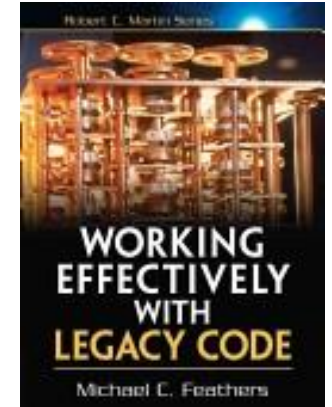
1. Cover code to be changed with tests to protect existing behavior
2. Change code and add new tests to define and protect new behavior
3. Refactor and clean up code to well match current functionality

Legacy Code Change Algorithm (Chapter 2 “Working Effectively with Legacy Code”)

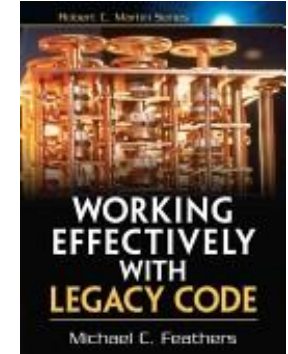
1. Identify Change Points
2. Find Test Points
3. Break Dependencies (without unit tests)
4. Cover Code with Verification or No-change/Characterization Unit or Integration Tests
5. Add New Functionality with Test Driven Development (TDD)
6. Refactor to remove duplication, clean up, etc.

Covering Existing Code with Tests: Details

- Identify Change Points: Find out the code you want to change, or add to
- Find Test Points: Find out where in the code you can sense variables, or call functions, etc. such that you can detect the behavior of the code you want to change.
- Break Dependencies: Do minimal refactorings with safer hipper-sensitive editing to allow code to be instantiated and run in a test harness. Can be at unit or integration test levels (consider tradeoffs).
- Cover Legacy Code with Unit Tests: If you have the specification for how to code is supposed to work, write tests to that specification (i.e. verification tests). Otherwise, write no-change or “Characterization Tests” to see what the code actually does under different input scenarios.



Legacy Software Tools, Tricks, Strategies



Reasons to Break Dependencies:

- Sensing: Sense the behavior of the code that we can't otherwise see
- Separation: Allow the code to be run in a test harness outside of production setting

Faking Collaborators:

- Fake Objects: Impersonates a collaborator to allow sensing and control
- Mock Objects: Extended Fake object that asserts expected behavior

Seams: Ways to inserting test-related code or putting code into a test harness.

- Preprocessing Seams: Preprocessor macros to replace functions, replace header files, etc.
- Link Seams: Replace implementation functions (program or system) to define behavior or sense changes.
- Object Seams: Define interfaces and replace production objects with mock or fake objects in test harness.
- NOTE: Prefer Object Seams to Link or Preprocessing Seams!

Unit Test Harness Support:

- C++: Teuchos Unit Testing Tools, Gunit, Boost?
- Python: pyunit ???
- CMake: ???
- Other: Make up your own quick and dirty unit test harness or support tools as needed!

Refactoring and testing strategies ... See the book ...

Two Ways to Change Software: An Example



The Goal: Refactor five functions on a few interface classes and update all subclass implementations and client calling code. Total change will involve changing about 30 functions on a dozen C++ classes and about 300 lines of C++ client code.

Option A: Change all the code at one time testing only at the end

- Change all the code rebuilding several times and documentation in one sitting [6 hours]
- Build and run the tests (which fail) [10 minutes]
- Try to debug the code to find and fix the defects [**1.5 days**]
- [Optional] Abandon all of the changes because you can't fix the defects

Option B: Design and execute an incremental and safe refactoring plan

- Design a refactoring plan involving several intermediate steps where functions can be changed one at a time [1 hour]
- Execute the refactoring in 30 or so smaller steps, rebuilding and rerunning the tests each refactoring iteration [15 minutes per average iteration, 7.5 hours total]
- Perform final simple cleanup, documentation updates, etc. [2 hour]

Are these scenarios realistic?

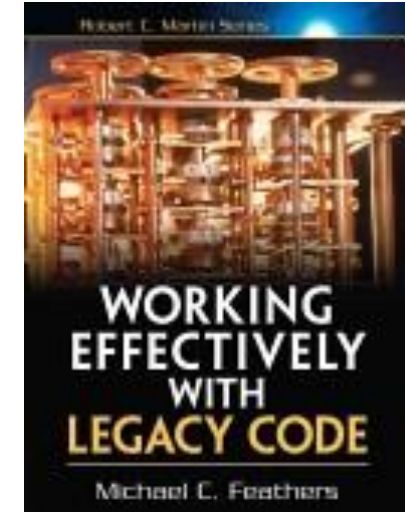
=> This is exactly what happened to me in a refactoring several years ago!

Legacy Software: Turning the ship around

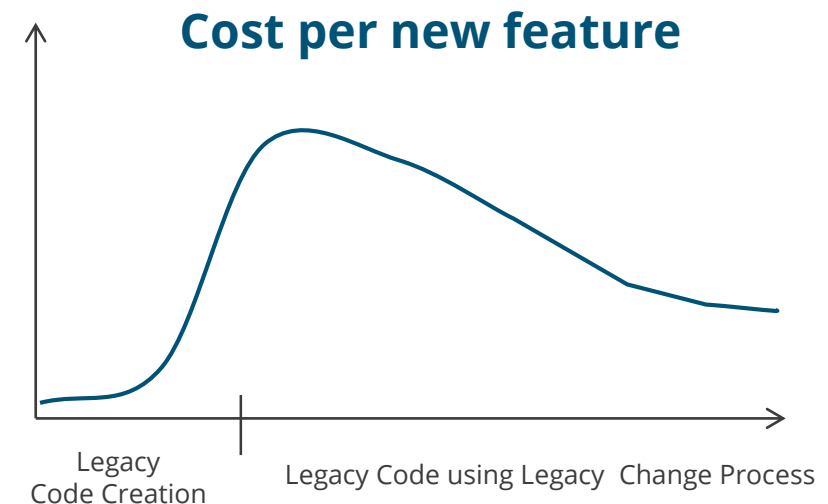


Agile Legacy Software Change Algorithm:

1. Identify Change Points
2. Break Dependencies
3. Cover with Unit Tests
4. Add New Functionality with Test Driven Development (TDD)
5. Refactor to removed duplication, clean up, etc.



NOTE: After enough iterations of the Legacy Software Change Algorithm the software may approach Agile-compliant sustainable software!

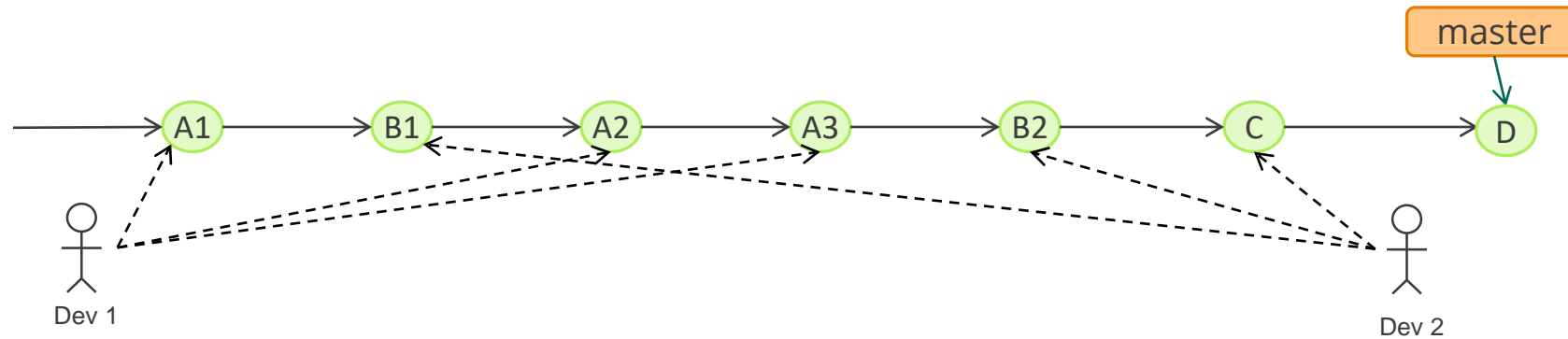


Any short questions or clarifications?



Basic Agile Development Workflows

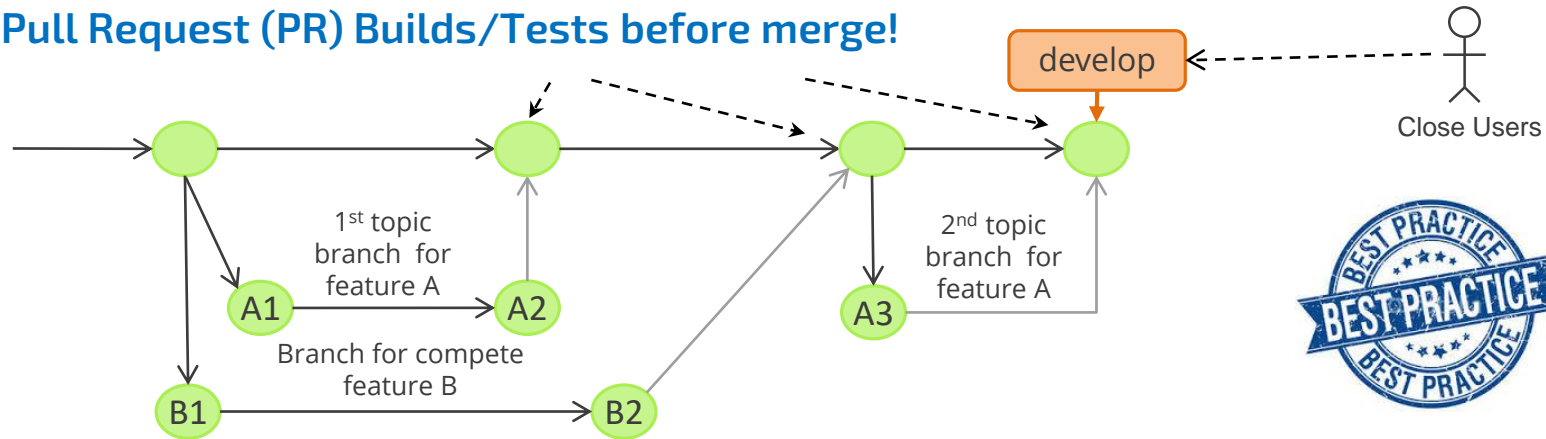
Simple Centralized CI Workflow



- Features implemented in commits intermingled on 'master' branch
 - Feature "A": Commits "A1", "A2", "A3"
 - Feature "B": Commits "B1", "B2"
 - Feature "C": Commit "C"
- **Pros and Cons** (w.r.t. other more sophisticated workflows):
 - **Pro:** Simplest workflow with fewest Git commands, no distributed VC concepts (i.e. SVN-like).
 - **Pro:** Requires least knowledge of Git .
 - **Pro:** Minimizes merge conflicts (frequent pushes to and pulls from 'master').
 - **Con:** Difficult to perform pre-merge code reviews.
 - **Con:** Difficult to collaborate with other developers with partial changes (can't push broken code to 'master' to share with others).
 - **Con:** Difficult to back out bad feature sets.
 - **Con:** Difficult to maintain 100% passing tests for all Nightly Builds.

Addition of Topic Branches and Pre-Merge/PR Testing

Must pass Pull Request (PR) Builds/Tests before merge!

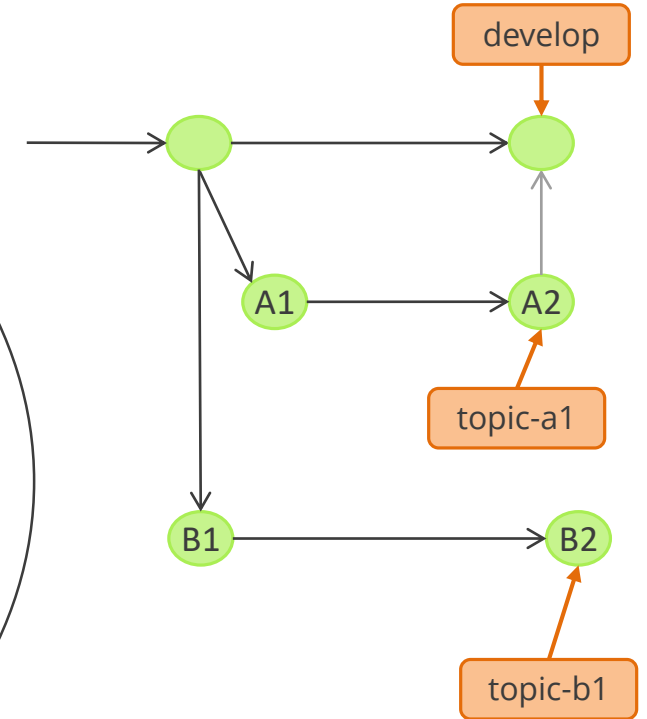
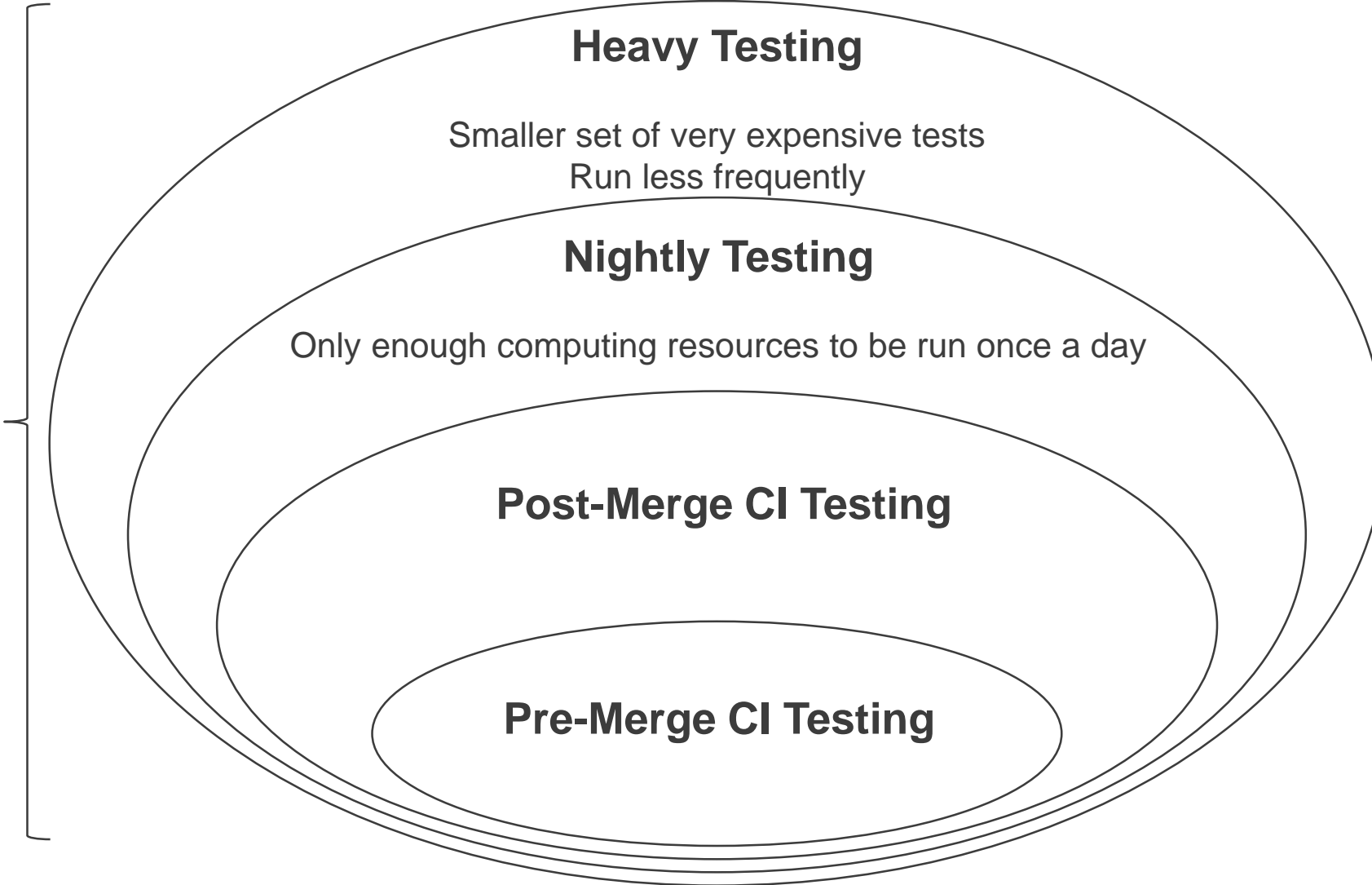


- **Introduce usage of temporary short-lived topic branches:**
 - Developers implement features in one or more topic branches and merge to 'develop'. E.g.:
 - Feature "A": 1st topic branch (commits "A1", "A2"), 2nd topic branch (commit "A3")
 - Feature "B": Single topic branch (commits "B1", "B2")
 - Topic branches pass **PR Builds** and merged into 'develop' about once/day or 4-6 hours of work (ideal)
 - **NOTE: Usage of topic branches does not degrade CI at all! Does not lead to more merge conflicts!**
 - **NOTE: Not typically long-lived "feature branches" that are hard to merge back!**
- **Pros and Cons (w.r.t. single branch workflow):**
 - **Pro:** Allow changes to be easily backed out if something goes wrong
 - **Pro:** Allow switching between different topic branches quickly
 - **Pro:** Allow easy sharing for quick collaboration with other devs before merging to 'develop'
 - **Pro:** Allow quick code reviews (pull-requests) on the topic branch before merging to 'develop'.
 - **Pro: Maps to GitHub Flow with Pull Requests including automated testing before merges!**
 - **Con:** Requires knowing how to use multiple branches and merges with Git

Where does testing fit in to development and integration processes?



Correctness Testing



Coverage Testing?

Memory Testing (e.g. Valgrind, Clang Sanitizers)?

Development & Integration: Speed vs Stability



Primary Goal: Provide stable yet frequent updates

Development and Integration Challenges:

- Balancing stability vs. speed of updates
 - **Stability:** Maintaining portability on wide range of platforms
 - **Stability:** Avoid getting significant defects

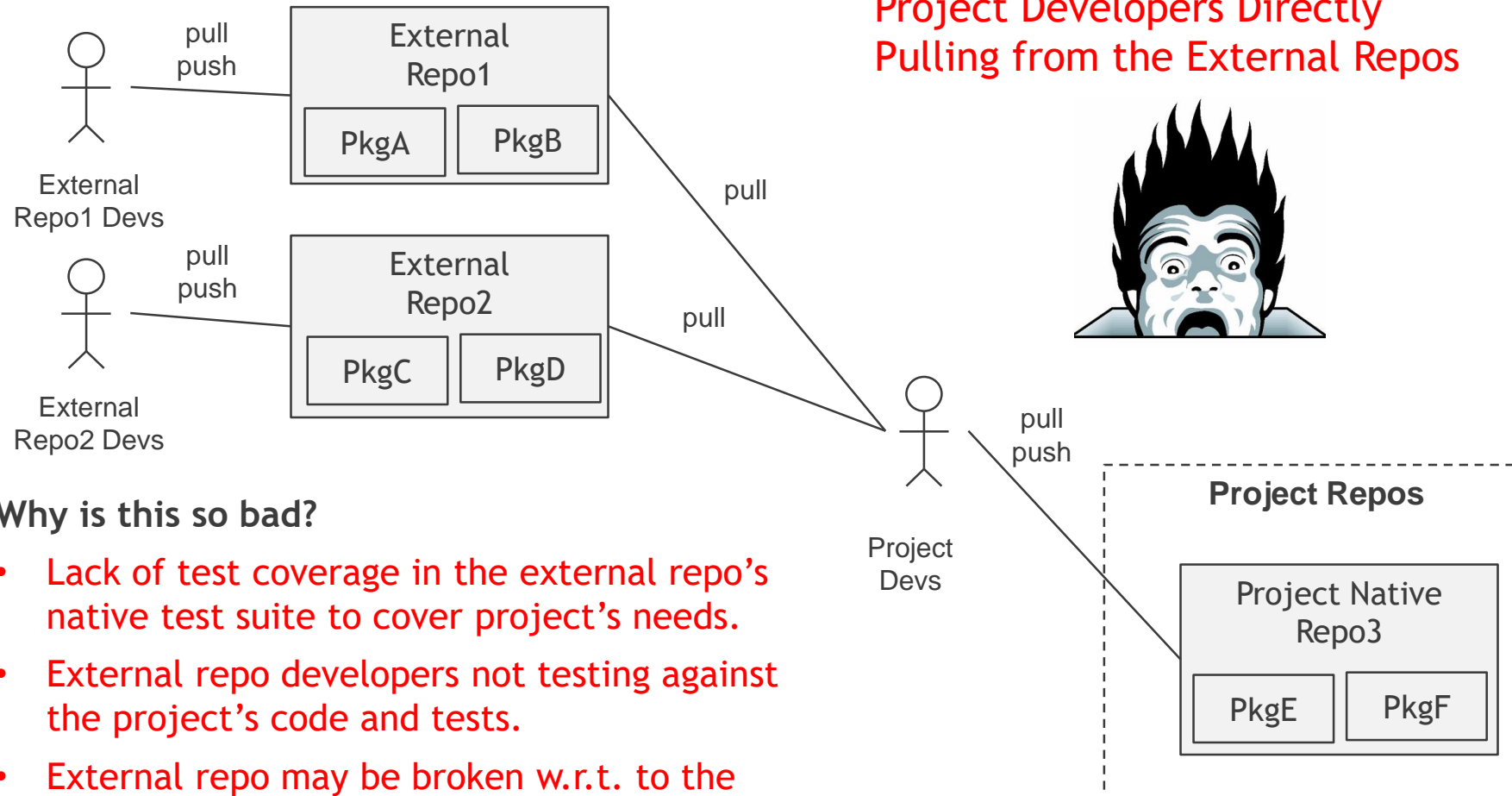
vs:

- **Speed:** Frequent updates to get new features to drive progress
- **Speed:** Avoid merge conflicts
- **Speed:** Co-development upstream and downstream packages requires frequent upgrades



Multi-Team Multi-Repository Integration

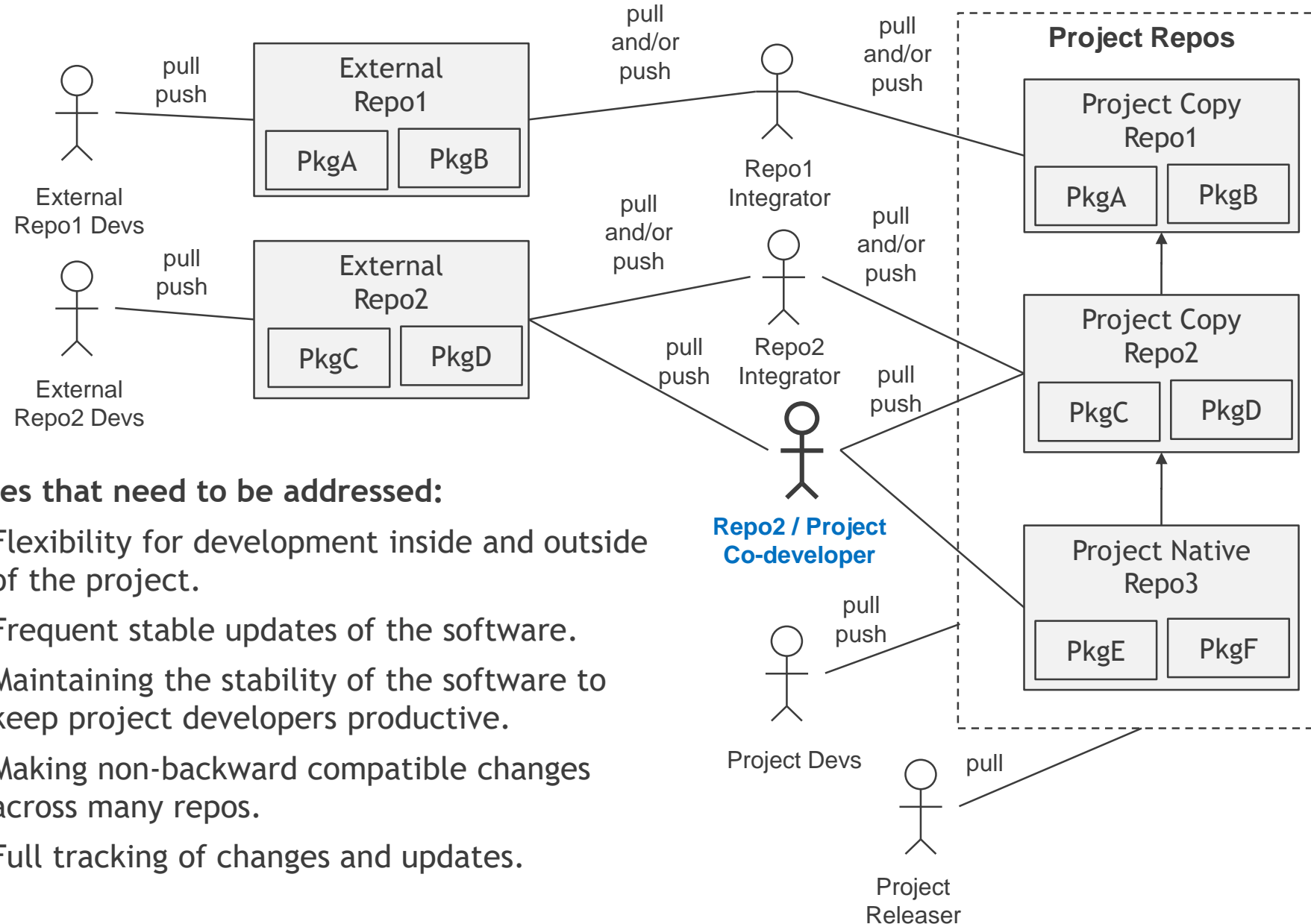
What Not to Do



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.

Managing Internal and External Development & Integration



Issues that need to be addressed:

- Flexibility for development inside and outside of the project.
- Frequent stable updates of the software.
- Maintaining the stability of the software to keep project developers productive.
- Making non-backward compatible changes across many repos.
- Full tracking of changes and updates.

Basic Parts to Development & Integration Processes



- **Git Workflows:**
 - How git repositories and branches are set up, how merges occur, what git commands are run, etc.
 - Different git workflows used for external repo developers, Project developers, and repo/project co-developers.
- **Testing gates for workflows:**
 - Gating test suites can/should be run before each “merge” in the workflow.
 - Gating tests can be run manually or automated, daily or “every-so-often”.
 - Important test suites:
 - **Upstream Package build & tests:** Gates updating the main Upstream Package ‘develop’ branch.
 - **APP builds & tests:** Gates all updates of the APPs’s repos.
- **Detection, triage and fixing of new failing builds and tests:**
 - Detection and notification of new failures.
 - Triage failures.
 - Address failures.
 - Manage & follow-up.

Development & Integration: Speed vs Stability



Primary Goal: Provide stable yet frequent updates

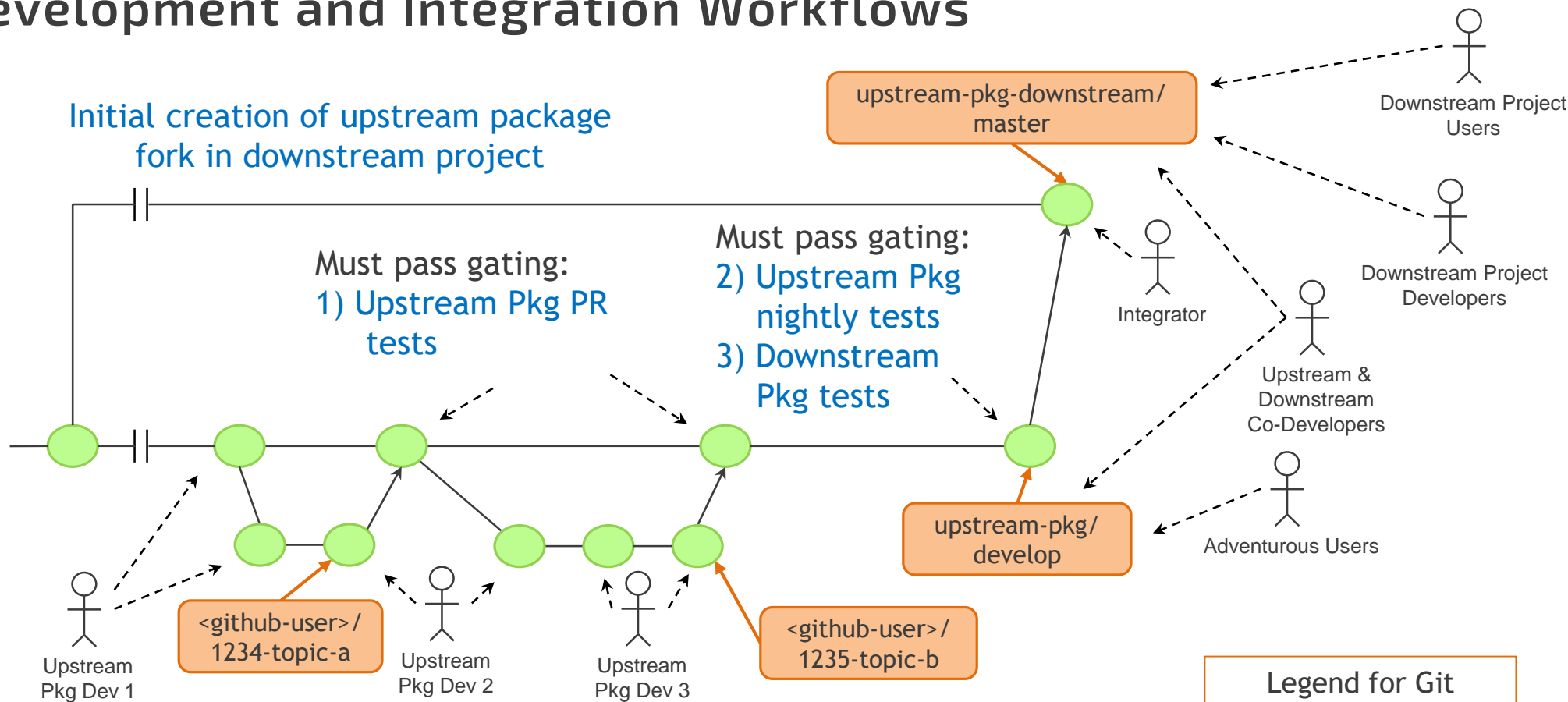
Development and Integration Challenges:

- Balancing stability vs. speed of updates
 - **Stability:** Maintaining portability on wide range of platforms
 - **Stability:** Avoid getting significant defects

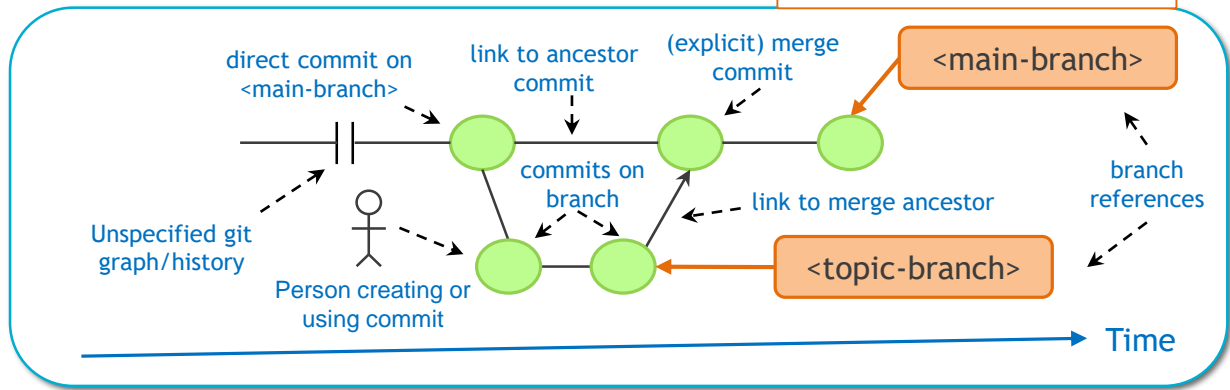
vs:

- **Speed:** Frequent updates to get new features to drive progress
- **Speed:** Avoid merge conflicts
- **Speed:** Co-development upstream and downstream packages requires frequent upgrades

Development and Integration Workflows



Legend for Git Workflow Diagrams



Testing and Defect Detection in Challenging CSE Environments

2) Upstream Package Nightly Builds & Tests (CDash, 2018+)



ATDM										38 builds	
Site	Build Name ▲	Update		Configure		Build		Test		Start Time	Labels
		Revision	Error	Warn	Error	Warn	Not Run	Fail	Pass		
cee-rhel8	Trilinos-atdm-cee-rhel8_clang-5.0.1_openmpi-1.10.2_serial_static_opt	e8df0f	0	1	0	50	0	0	2121	May 22, 2019 - 04:20 UTC	(27 labels)
cee-rhel8	Trilinos-atdm-cee-rhel8_gnu-7.2.0_openmpi-1.10.2_serial_shared_opt	e8df0f	0	1	0	50	0	0	2123	May 22, 2019 - 05:24 UTC	(27 labels)
cee-rhel8	Trilinos-atdm-cee-rhel8_intel-17.0.1_intelmpi-5.1.2_serial_static_opt	e8df0f	0	1	0	50	0	0	2120	May 22, 2019 - 06:20 UTC	(27 labels)
cee-rhel8	Trilinos-atdm-cee-rhel8_intel-18.0.2_mpic2-3.2_openmp_static_opt	e8df0f	0	1	0	50	0	0	2348	May 22, 2019 - 07:55 UTC	(27 labels)
chama	Trilinos-atdm-chama-intel-debug-openmp	e8df0f	0	20	0	0				May 22, 2019 - 11:54 UTC	(25 labels)
chama	Trilinos-atdm-chama-intel-opt-openmp	e8df0f	0	20	0	0				May 22, 2019 - 11:54 UTC	(25 labels)
hansen	Trilinos-atdm-hansen-shiller-gnu-debug-openmp	e8df0f	0	20	0	50	0	0	2050	May 22, 2019 - 12:55 UTC	(25 labels)
hansen	Trilinos-atdm-hansen-shiller-gnu-debug-serial	e8df0f	0	20	0	50	0	0	1940	May 22, 2019 - 08:12 UTC	(25 labels)
hansen	Trilinos-atdm-hansen-shiller-gnu-opt-openmp	e8df0f	0	20	0	50	0	0	2051	May 22, 2019 - 23:05 UTC	(25 labels)
hansen	Trilinos-atdm-hansen-shiller-gnu-opt-serial	e8df0f	0	20	0	50	0	0	1941	May 22, 2019 - 08:19 UTC	(25 labels)
hansen	Trilinos-atdm-hansen-shiller-intel-debug-openmp	e8df0f	0	20	0	50	0	0	2050	May 22, 2019 - 17:24 UTC	(25 labels)
hansen	Trilinos-atdm-hansen-shiller-intel-debug-serial	e8df0f	0	20	0	50	0	0	1942	May 22, 2019 - 10:16 UTC	(25 labels)
hansen	Trilinos-atdm-hansen-shiller-intel-opt-openmp	e8df0f	0	20	0	50	0	0	2051	May 22, 2019 - 20:18 UTC	(25 labels)
hansen	Trilinos-atdm-hansen-shiller-intel-opt-serial	e8df0f	0	20	0	50	0	0	1945	May 22, 2019 - 14:49 UTC	(25 labels)
mutrino	Trilinos-atdm-mutrino-intel-opt-openmp-HSW	e8df0f	0	20	0	0				May 22, 2019 - 13:58 UTC	(25 labels)
mutrino	Trilinos-atdm-mutrino-intel-opt-openmp-KNL-panzer	e8df0f	0	2	0	50	0	1	188	May 22, 2019 - 09:04 UTC	Panzer
sems-rhel8	Trilinos-atdm-sems-rhel8-gnu-7.2.0-openmp-complex-shared-release-debug	e8df0f	0	20	0	50	0	0	2097	May 22, 2019 - 12:39 UTC	(25 labels)
sems-rhel8	Trilinos-atdm-sems-rhel8-gnu-7.2.0-openmp-debug	e8df0f	0	20	0	50	0	0	2050	May 22, 2019 - 13:49 UTC	(25 labels)
sems-rhel8	Trilinos-atdm-sems-rhel8-gnu-7.2.0-openmp-release	e8df0f	0	20	0	50	0	0	2051	May 22, 2019 - 10:21 UTC	(25 labels)
sems-rhel8	Trilinos-atdm-sems-rhel8-gnu-7.2.0-openmp-release-debug	e8df0f	0	20	0	50	0	0	2050	May 22, 2019 - 11:05 UTC	(25 labels)
sems-rhel8	Trilinos-atdm-sems-rhel8-intel-17.0.1-openmp-release	e8df0f	0	20	0	50	0	0	2051	May 22, 2019 - 12:20 UTC	(25 labels)
sems-rhel7	Trilinos-atdm-sems-rhel7-clang-3.9.0-openmp-complex-shared-release-debug	e8df0f	0	20	0	50	0	0	2097	May 22, 2019 - 15:00 UTC	(25 labels)
sems-rhel7	Trilinos-atdm-sems-rhel7-cuda-9.2-Volta70-complex-static-release-debug	e8df0f	0	23	0	100	0	0	1987	May 22, 2019 - 11:51 UTC	(24 labels)
sems-rhel7	Trilinos-atdm-sems-rhel7-gnu-7.2.0-openmp-complex-shared-release-debug	e8df0f	0	20	0	50	0	0	2097	May 22, 2019 - 15:52 UTC	(25 labels)
sems-rhel7	Trilinos-atdm-sems-rhel7-intel-17.0.1-openmp-complex-shared-release-debug	e8df0f	0	20	0	50	0	0	2097	May 22, 2019 - 15:04 UTC	(25 labels)
serrano	Trilinos-atdm-serrano-intel-debug-openmp	e8df0f	0	20	0	0				May 22, 2019 - 11:57 UTC	(25 labels)
serrano	Trilinos-atdm-serrano-intel-opt-openmp	e8df0f	0	20	0	50	0	0	2050	May 22, 2019 - 11:57 UTC	(25 labels)
waterman	Trilinos-atdm-waterman-cuda-9.2-debug	e8df0f	0	8	0	0				May 22, 2019 - 14:27 UTC	(27 labels)
waterman	Trilinos-atdm-waterman-cuda-9.2-rtc-release-debug	e8df0f	0	8	0	50	0	0	2215	May 22, 2019 - 09:05 UTC	(27 labels)
waterman	Trilinos-atdm-waterman_cuda-9.2_fp16_static_opt	e8df0f	0	8	0	50	0	4	2219	May 22, 2019 - 12:24 UTC	(27 labels)
waterman	Trilinos-atdm-waterman_cuda-9.2_shared_opt	e8df0f	0	8	0	50	0	0	2223	May 22, 2019 - 09:06 UTC	(27 labels)
ride	Trilinos-atdm-white-ride-cuda-9.2-gnu-7.2.0-debug	e8df0f	0	21	0	50	0	0	2031	May 22, 2019 - 10:10 UTC	(25 labels)
ride	Trilinos-atdm-white-ride-cuda-9.2-gnu-7.2.0-rtc-release-debug	e8df0f	0	21	0	50	0	1	2059	May 22, 2019 - 10:08 UTC	(25 labels)
ride	Trilinos-atdm-white-ride-cuda-9.2-gnu-7.2.0-rtc-release-debug-pt	e8df0f	0	10	0	0				May 22, 2019 - 11:48 UTC	(53 labels)
ride	Trilinos-atdm-white-ride-cuda-9.2-gnu-7.2.0-release	e8df0f	0	21	0	50	0	0	2059	May 22, 2019 - 11:50 UTC	(25 labels)
ride	Trilinos-atdm-white-ride-cuda-9.2-gnu-7.2.0-release-debug	e8df0f	0	21	0	50	0	0	2060	May 22, 2019 - 10:11 UTC	(25 labels)
ride	Trilinos-atdm-white-ride-gnu-7.2.0-openmp-debug	e8df0f	0	20	0	50	0	0	2048	May 22, 2019 - 10:09 UTC	(25 labels)
ride	Trilinos-atdm-white-ride-gnu-7.2.0-openmp-release	e8df0f	0	20	0	50	0	0	2051	May 22, 2019 - 10:05 UTC	(25 labels)

- Build and run native Trilinos test suite on all customer platforms.
- Catch defects in upstream package and system software customer platforms.
- Builds are too expensive to run more than one set per 24-hour day.
- Build and test results often go missing (e.g. should be 40 not 38 builds and missing test results in 6 builds).
- Frequent random system failures make detection of new code-related failures difficult.

Common Problem: Existing Failures Hiding New Failures



Example: Customer update of upstream package with key defect:

- APP customer had many failing tests on dashboard every day
- Customer **updated upstream package with a major defect** which **triggered a new failing APP test**. (**Did not notice new failing test due to other existing failing tests!**)
- Major upstream package defect only noticed months later.
- Detailed & expensive Git bisection study performed over months of upstream package commit history discovered the defect.

=> Discovered that an APP test had actually caught this!

Lessons Learned:

- **Don't let existing failing tests hide the emergence of new failing tests!**
- **Must examine every failing test to look for possible new defects!**

Testing Challenges for CSE/HPC software



Testing Challenges:

- **Random system-related failures create noise on results dashboard.**
- *Random build failures due to system issues:*
 - Disk and network I/O issues
 - Machine overloading
 - Compiler license server problems (e.g. Intel license server).
- *Random test failures due to system issues:*
 - Disk and network I/O issues
 - MPI stat-up problems
 - Non-deterministic (rare) bugs in bleeding-edge system software (e.g. OpenMPI+CUDA CPU/GPU data management).
- *(Randomly) failing tests due to defects in CSE code:*
 - Defects in features not used by real customers
 - Defects in tests (but not the code itself) for features used by real customers
 - **Defects in functional software used by customers (5-10% of failing tests)**
- *Missing build and test results on dashboard:*
 - Machine overloading (e.g. queues are full)
 - Machines go down for maintenance in middle of build/testing.
 - Communication problems with dashboard server to upload results.

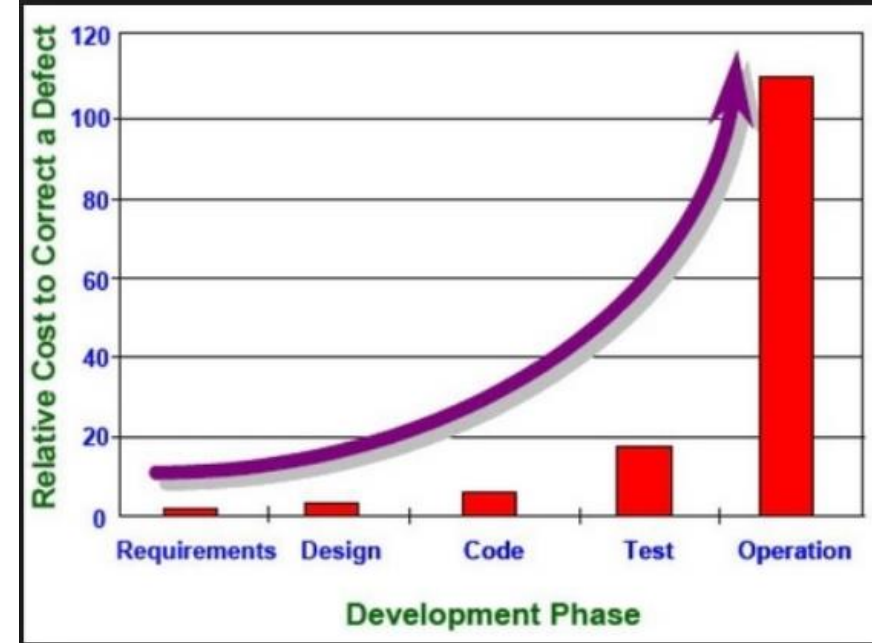
Can't judge health of a CSE/HPC code from just the amount of red or green on results dashboard!

Reducing Time to Detect, Triage, and Address New Failures

General SE Principles for Defects



- Cost of a defect goes up (significantly) the longer it takes to detect and correct a defect.



- **Lean/Agile SE Practices for dealing with defects:**
 - Strong automated testing (have tests help new detect defects)
 - Continuous testing (reduce the time to detect new defects caught by tests)
 - Continuous integration (reduce time to detect conflicts)
 - **STOP THE LINE** when a new defect gets into the main development branch
 - Fixing defects in previously working software is higher priority than developing new features!

Where to Catch Defects in Upstream Package?



Upstream package native test suite running in customer configurations/environments

- **Best place to catch an upstream package's defect!**
- Upstream package developers can triage and fix a defect before customer/package Integrators need to dig in to triage customer failures caused by these defects

Downstream Package native test site

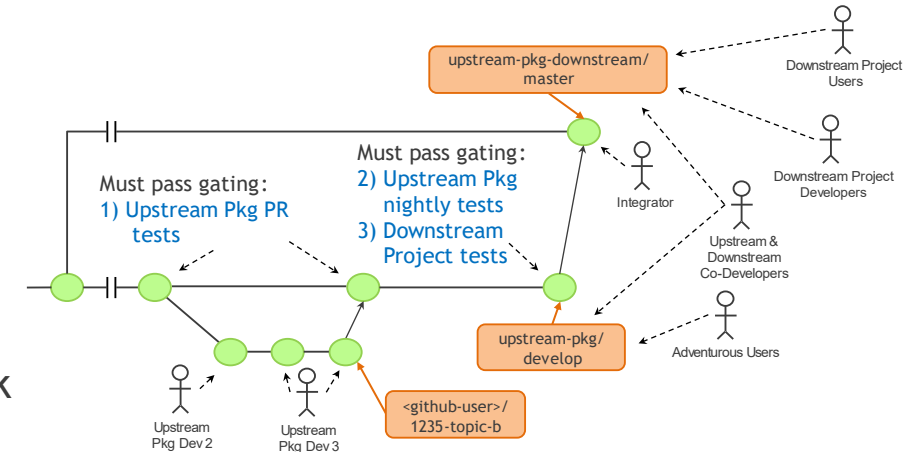
- Less than best place to catch a upstream package defect
- Requires Downstream/Upstream Package Integrator and Downstream Developers to triage problems and communicate back to Upstream Package developers

Downstream developer or user when running downstream code

- **The worst place to catch an upstream package defect!**
- Customer has to report problems back to Developers who have to triage the failure and then report back to upstream package developers

Example:

- SEACAS update <https://github.com/trilinos/Trilinos/issues/2650>
 - Broke upstream Trilinos/SEACAS CUDA test suite
 - Did **NOT** break the downstream EMPIRE test suite
 - Broke usage of downstream EMPIRE by EMPIRE users!



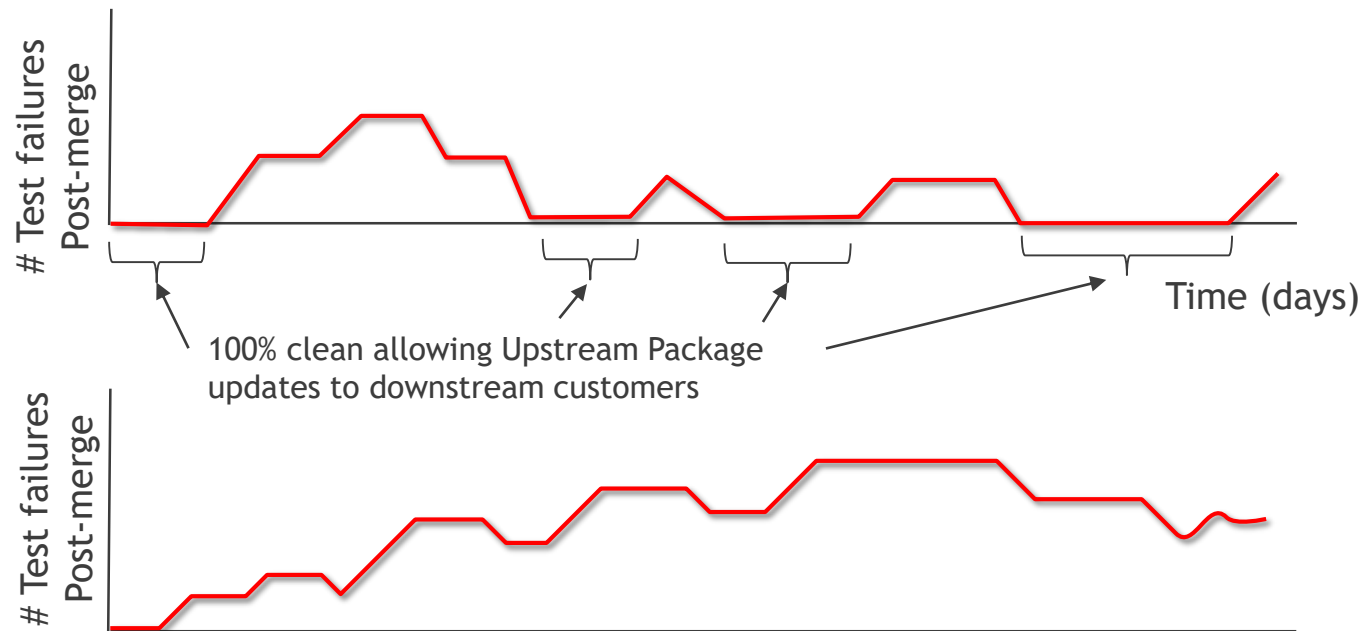
If update of Trilinos/SEACAS was gated by 100% passing SEACAS tests, then downstream EMPIRE developers and users may have never seen these defects!

Injecting New Failures and Fixing Failures: A Race!

- **Mean-time to fail:** Average time (in days) for when a new failure shows up in 'develop' branch.
- **Mean-time to fix:** Average time (in days) to discover, triage and fix a failure on the Upstream Package 'develop' branch.
- **The core problem:** If “mean-time to fail” is less than “mean-time to fix”, then the Upstream Package builds on 'develop' on average will **ALWAYS be broken** (and therefore block updates of Upstream Package to downstream customers)!

Mean-time to fix
<
Mean-time to fail

Mean-time to fix
>
Mean-time to fail

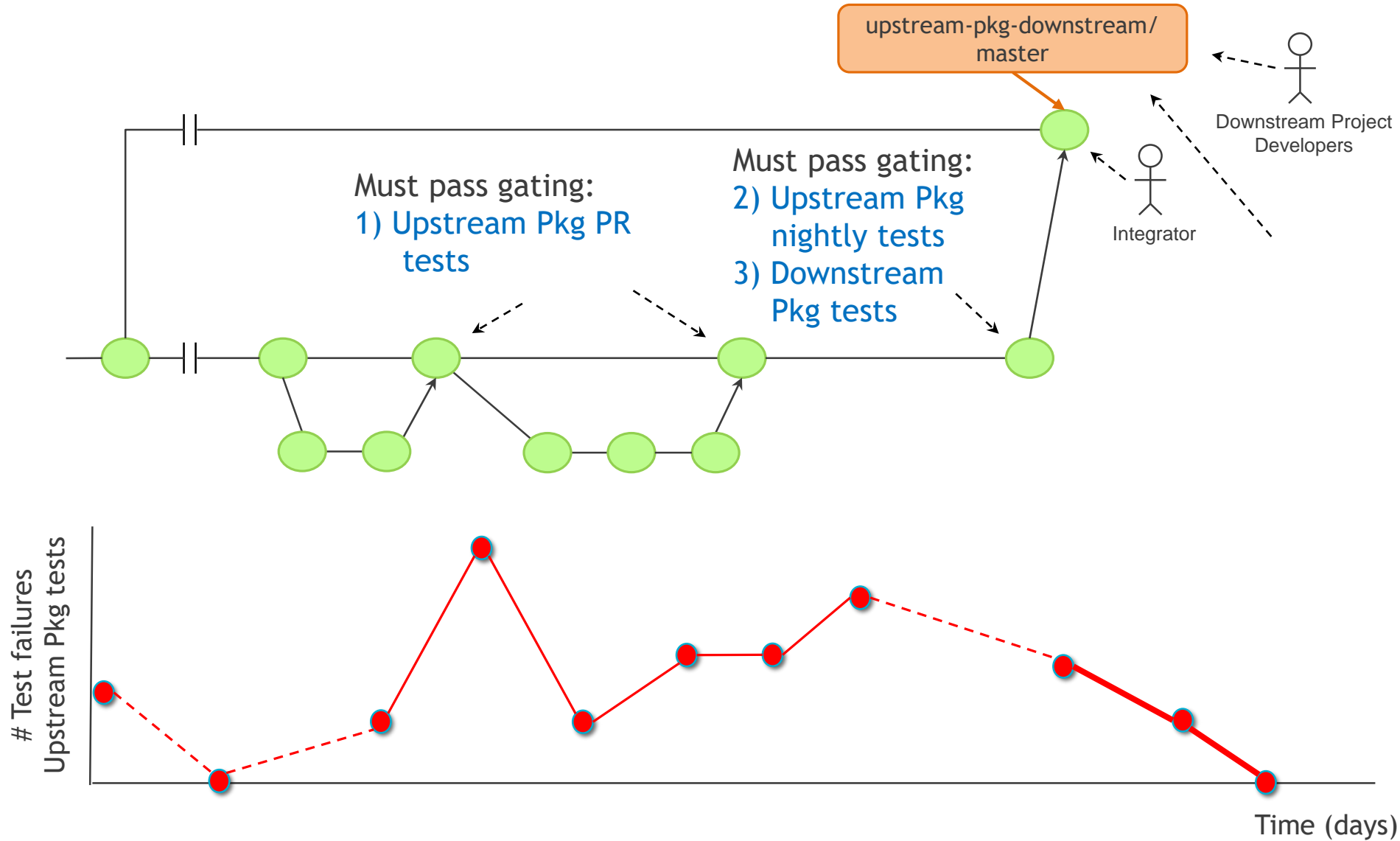


Options for updates of upstream package in downstream project



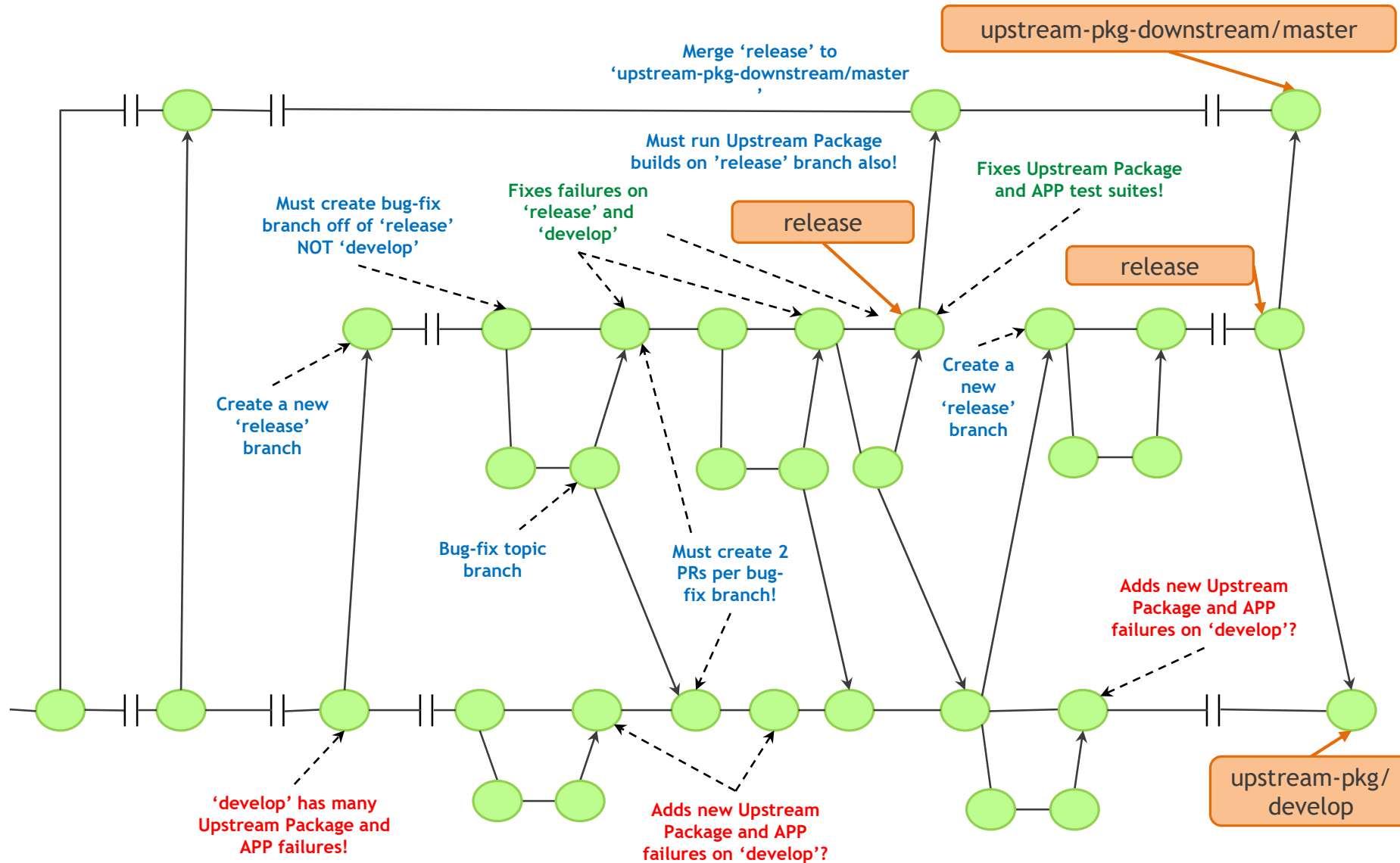
- Option-1: Make Upstream Package builds clean on ‘develop’ periodically
- Option-2: Create ‘release’ branches and clean up there

Option-1: Get clean upstream package test on 'develop'



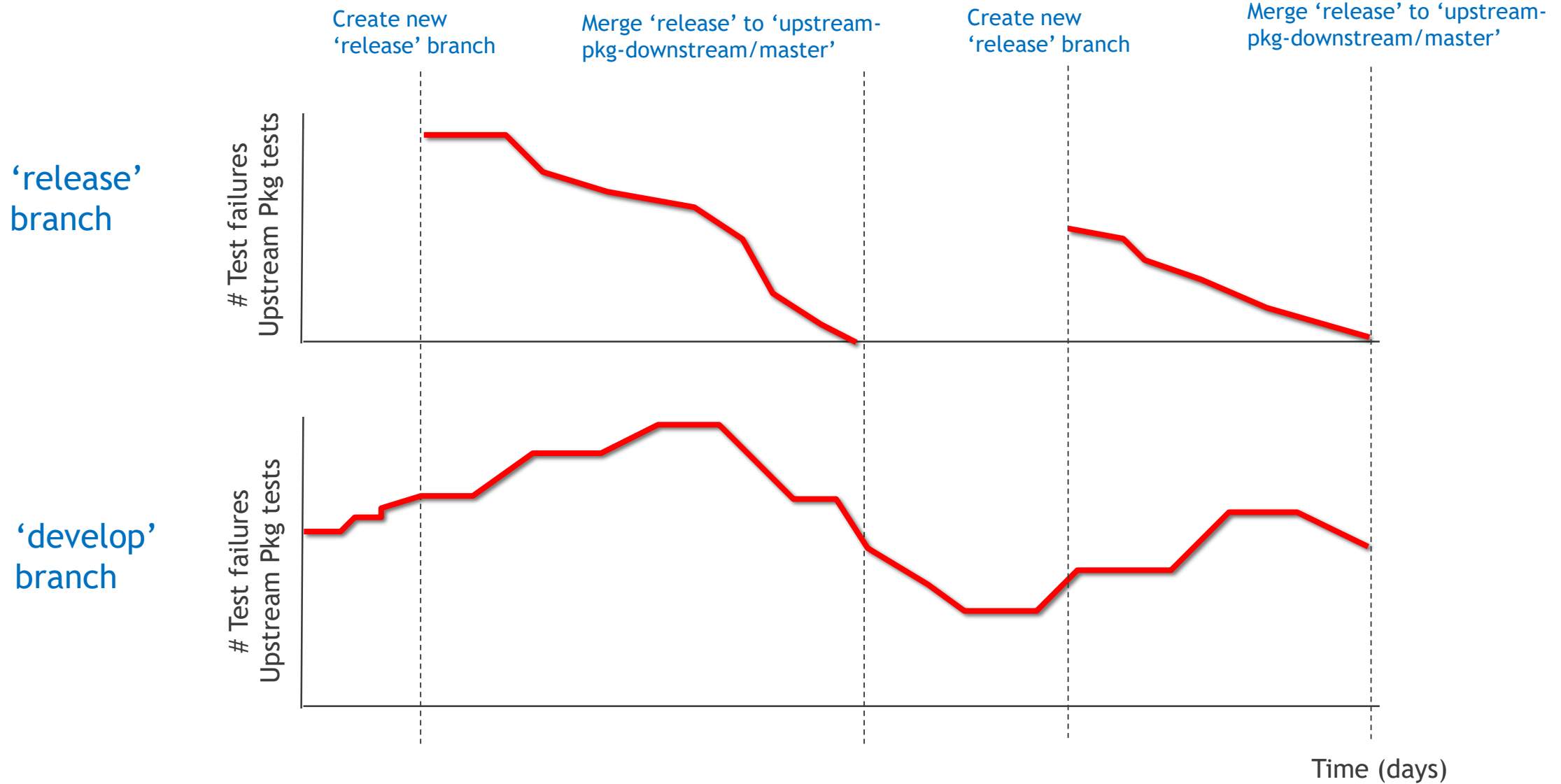
Mean-time to fix < Mean-time to fail

Option-2: Upstream package release branches: Workflow



NOTE: Note this is really just an adaptation of the [gitworkflows\(7\)](#) release 'maint' branch

Option-2: Upstream Release Branches: Failures



Options for Upstream Package APP Updates: Summary



- **Option-1: Make Upstream Package builds clean on ‘develop’ periodically**
 - **Assumes:** “Mean-time to fix” is less than the “mean-time to fail” on ‘develop’ branch.
 - **Pro:** Requires just one set of builds on the platforms.
 - **Pro:** Simpler workflow for Upstream Package developers merge bug fixes to ‘develop’ branch.
 - **Pro:** Provides quicker APP updates of Upstream Package.
 - **Pro:** Allows APPs like EMPIRE to co-develop Upstream Package and update Upstream Package ‘develop’ and get updates to the APP fairly regularly.
 - **Con:** Requires fast reaction time to detect and triage new failures and then either a) fix, b) disable, or c) revert breaking PRs so that the “mean-time to fix” is less than “mean-time to fail”.
- **Option-2: Create ‘release’ branches and clean up there**
 - **Assumes:** “Mean-time to fix” is less than the “mean-time to failure” (not true right)
 - **Pro:** More leisurely reaction time to fix defects since no race with “mean-time to fail”.
 - **Pro:** Guaranteed periodic Upstream Package updates with 100% clean Upstream Package builds.
 - **Con:** Requires double the number of builds; one on ‘develop’, one on ‘release’
 - **Con:** More complex workflow for Upstream Package developers to commit fixes to ‘release’ and then merge back to “develop” in two Upstream Package PRs per bug-fix branch!
 - **Con:** More complex workflow for APP Upstream Package co-developers involving branches, cherry-picks (e.g. EMPIRE git-git-like workflow and SPARC cherry-picking workflow).



Detecting New Failures/Missing Results: Dashboard Analysis

FAILED (bm=1, twoif=2, twip=1, twif=2): Promoted ATDM Trilinos Builds on 2019-01-04

[Builds on CDash](#) (num/expected=33/33)

[Non-passing Tests on CDash](#) (num=4)

Builds Missing: bm=1

Tests without issue trackers Failed: twoif=2

Tests with issue trackers Passed: twip=1

Tests with issue trackers Failed: twif=2

Failures in **red** may require triage!

- **Missing test results!**
- **Failing tests without issue trackers!**

Builds Missing: bm=1

Group	Site	Build Name	Missing Status
ATDM	waterman	Trilinos-atdm-waterman-cuda-9.2-release-debug	Build exists but no test results

Tests without issue trackers Failed (limited to 20): twoif=2

Site	Build Name	Test Name	Status	Details	Consecutive Non-pass Days	Non-pass Last 30 Days	Pass Last 30 Days	Issue Tracker
sems-rhel6	Trilinos-atdm-sems-rhel6-intel-opt-openmp	Belos BlockGmresPoly Epetra - File Ex 0 MPI 4	Failed	Completed (Failed)	<u>1</u>	<u>1</u>	<u>22</u>	

...

Tests with issue trackers Failed: twif=2

Site	Build Name	Test Name	Status	Details	Consecutive Non-pass Days	Non-pass Last 30 Days	Pass Last 30 Days	Issue Tracker
mutrino	Trilinos-atdm-mutrino-intel-opt-openmp-HSW	Anasazi Epetra BKS norestart - test MPI 4	Failed	Completed (Failed)	<u>21</u>	<u>21</u>	<u>3</u>	#3499

Software Engineering (SE) Challenges in CSE: Solutions?



- Needing to develop and integrate increasingly more complex algorithms from specialized domains to continue progress solving CSE problems
 - => Adopting/adapting Agile development practices
- Needing to update CSE/HPC software for new algorithms and new programming environments
 - => Legacy Software Change Process
- CSE developers are domain experts (classic engineering, applied math, physics, etc.) and lack basic SE knowledge and skills
- Difficultly hiring and retaining skilled software engineers to aid CSE domain expert developers
 - => Research Software Engineers


Research Software Engineer Role?



- **A Research Software Engineer (RSE)** is a software engineer who **works with and supports CSE researchers and developers** to develop, maintain, and extend CSE software
- RSEs have a strong understanding of both **the research domain** and **software engineering principles**, and they use this knowledge to help researchers solve problems and achieve their goals.
- What are the responsibilities of an RSE?
 - Developing and maintaining software that supports research projects
 - Working with researchers to understand their needs and requirements
 - Designing and implementing software solutions
 - Testing and debugging software
 - Documenting software
 - Collaborating with other software engineers and researchers
 - Keeping up-to-date with the latest software approaches and technologies
- RSE Organizations:
 - **RSE** (society-rse.org) Society of Research Software Engineering (UK)
 - **US-RSE** (us-rse.org): The United States Research Software Engineer Association

Better Scientific Software (<https://bssw.io>)

Information For ▾ Contribute to BSSw Receive Our Email Digest Contact BSSw

 better scientific software Resources ▾ Blog Events About ▾ 🔍

Better Scientific Software (BSSw)

Software—the foundation of discovery in computational science & engineering—faces increasing complexity in computational models and computer architectures. BSSw provides a central hub for the community to address pressing challenges in software productivity, quality, and sustainability.

GET ORIENTED [Communities Overview](#) [Intro to CSE](#) [Intro to HPC](#)



Questions?