



ONTAP Continuous Integration/Testing

How a change becomes a product

Phil Ezolt
Netapp MTS 6 AERO/DevOps
University of Pittsburgh (7/20/20)



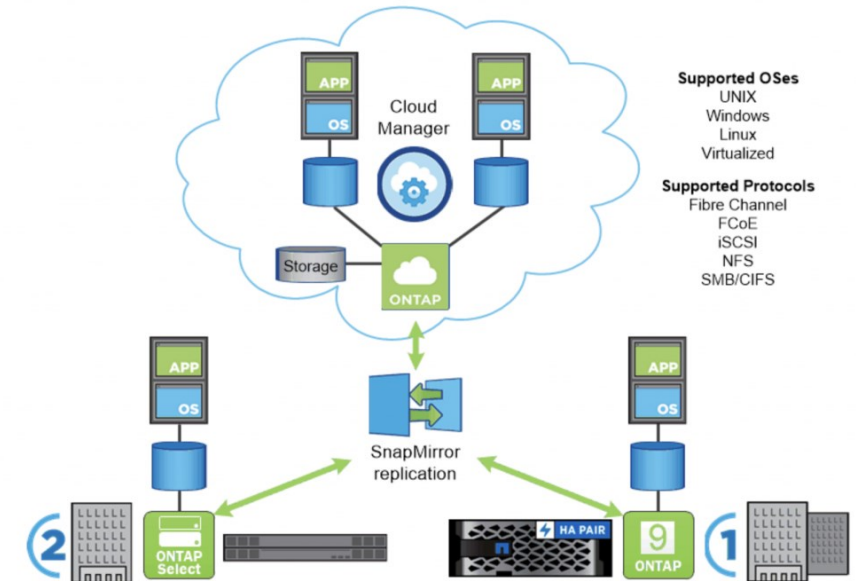
ONTAP

■ What is ONTAP?

- Data Management Software: Fast & reliable access to data
- Built-in storage efficiencies: snapshots, dedup
- Access your data: NFS/SAN/CIFS/more.
- Manage your data: GUI or CLI or Zapi (XML) or REST
- Protect your data: replication & encryption
- Runs on clustered Netapp filers, in VMs, or in the Cloud

■ ONTAP feature set is huge:

- Netapp has been making ONTAP for 20+ years
- <https://www.netapp.com/us/products/data-management-software/ontap.aspx>



Across the NetApp data fabric, you can count on a common set of features and fast, efficient replication across platforms. You can use the same interface and the same data management tools.

Why is shipping ONTAP hard?

Challenging development environment

■ Diverse codebase

- >10 millions lines of executable code (Not counting some 3rd party code)
- Kernel & User code running in FreeBSD
- C/C++ for product, python/perl for test code.
- Significant 3rd-party/opensource footprint



freeBSD



python™

THE
C
PROGRAMMING
LANGUAGE

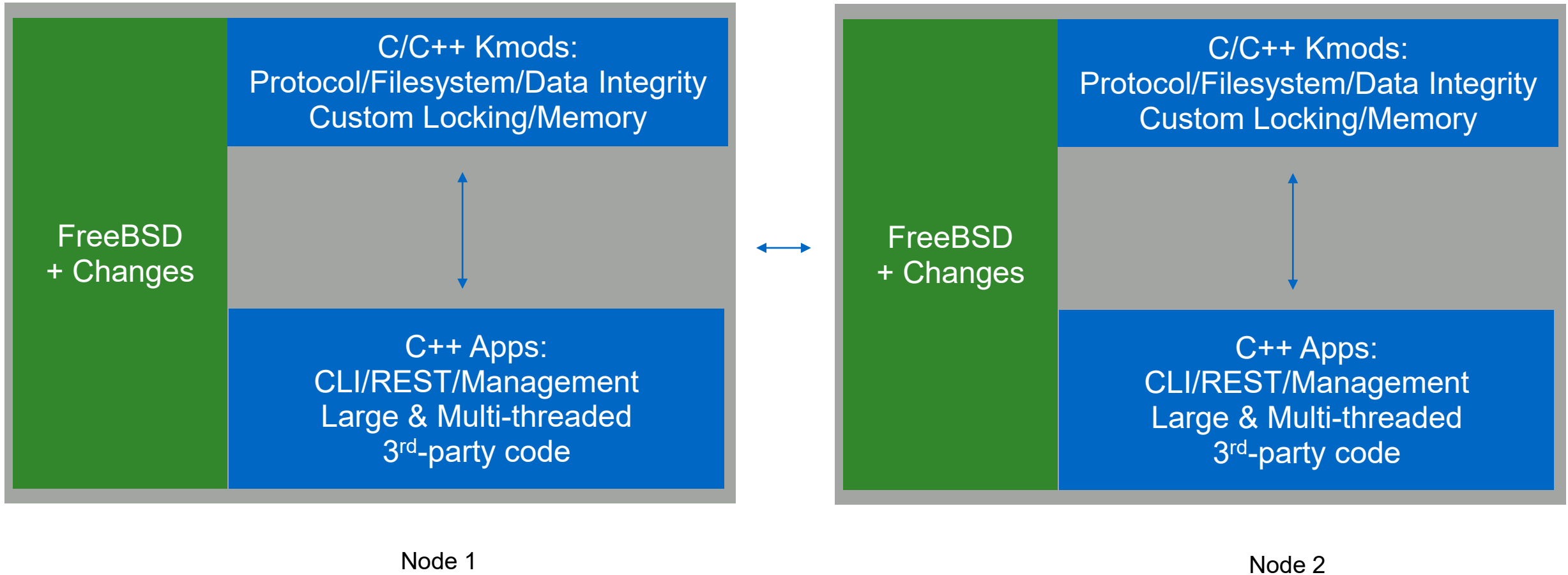


■ Constant change

- 20+ year-old code base
- >1000 developers
- High churn -> 38k changes submitted in 2018
- Subtle interactions -> Changes to Feature A can break Feature B.



ONTAP complexity: Many Moving Parts



ONTAP Engineering (Before)

- ONTAP in 2012: quality/cadence needed to improve
 - Customers needed:
 - New features faster than every 3 years
 - New features to work on the first release
 - Old features to keep working
 - Engineers needed:
 - A codeline this isn't always broken
 - An escape from the end-of-release "test-to-health"
 - A reliable way to test whether pending changes work and didn't regress other features.
 - A way to efficiently collaborate on features across groups and sites

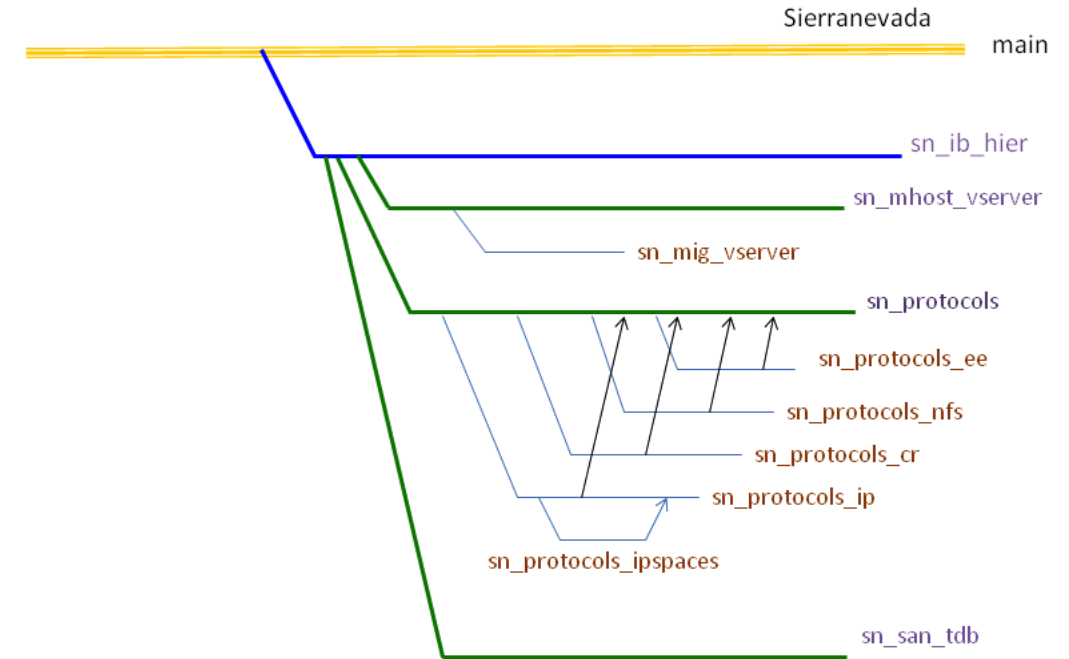
ONTAP quality/cadence needed to improve

■ ONTAP in 2012:

- Test code and product in different P4 Depots
- Many branches and sub-branches
 - Teams own branches...
 - developed independently
 - pushed features/bug fixes up every 6-8 week.
 - Fix set of tests run on merge integration point
- Main took months to stabilize (when code came together)


■ Problems:

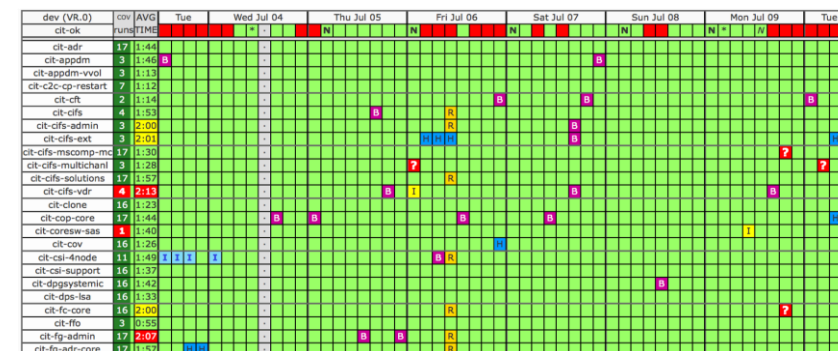
- Bug fixes 8 weeks from branch -> branch (Burt reports multiplied)
- Test code must handle every branches... or must wait until all code comes together
- High people cost synching/merging many branches



ONTAP Engineering (After) : DevOps for the win

Simplification and Continuous Testing

- Pick one way and do it well:
 - One Codeline, One dev process, One operations team, One test pipeline, One codeline health report
 - 1000 developers: trunk-based development in a Monorepo  P E R F O R M A N C E
- Give developers good builds/make change testing easy!
 - FlexClones provide a fully built client in ~1 min.
 - Run unit-test during build based on pending change.
 - Run most appropriate CI test based on pending change. (Coverage + Machine Learning to pick tests)
- Run regression tests continually
 - Unit-tests (>38k testcases) -> Every 10 minutes
 - Continuous Integration Tests (100x 2-hour tests) -> Every 3-hours
 - Continuous Integration Tests (500x 2-hour tests) -> Every Night
 - Continuous Integration Tests (250x 2-hour tests) -> Every Week
 - Autoheal: Automation detects and reverts regression-causing changes



How do we keep ONTAP working today?

■ ONTAP uses Continuous Integration

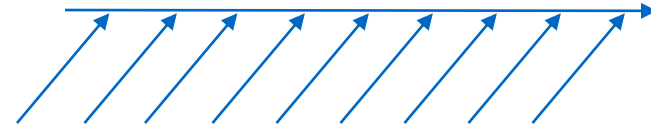
- All dev submits to a single perforce depot:

- DOT:dev -> Master development branch
- No feature branches
- Submit risky content disabled (dark)

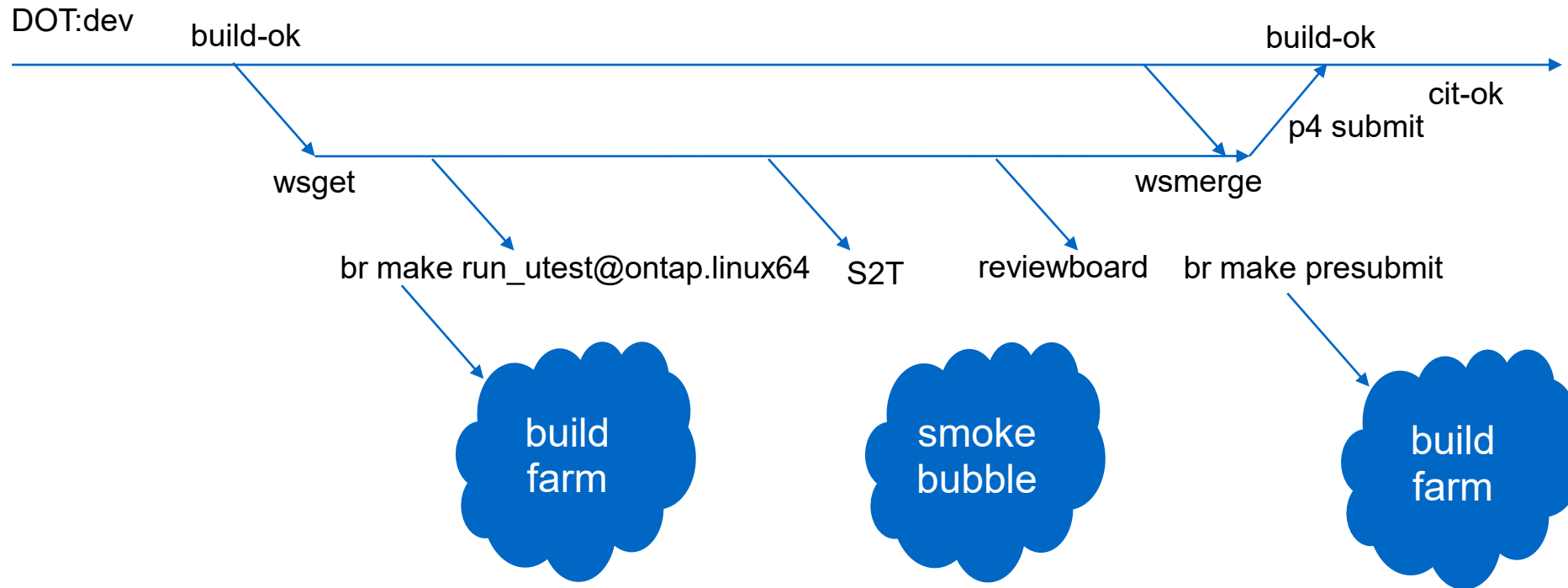
- monorepo (ish)

- Contains 3rd-party code/FreeBSD/ONTAP code/unit-tests/build-scripts/test-code/test-tools
- Can build from scratch, but most devs use incremental builds.
- Managed by internal build system (bedrock)... One command can build everything.
- Fully built workspaces snapshotted and available as a flexclone (~1 min for a full client)
- Every file has an owner

DOT:dev



DOT:dev - Life of a Change



How do we keep it working?

- Known good points:
 - build-ok -> Change builds most variants & passes in-build unit-tests
 - Every 20 minutes
 - In-build unit-tests -> 38k CxxTest based ONTAP test-cases
 - cit-ok -> Change successfully passes all ~100 Continuous Integration Tests (CITs)
 - Every 3-hours
 - CITs -> Run for 2-hours, typically end-to-end ONTAP testing on VMs (vsims)
- Workspace (Client) pre-submission requirements
 - 'br make presubmit' -> build most variants, run all unit-tests
 - Source-2-test (S2T) -> run 6 CITs based on pending changes
 - Reviewboard



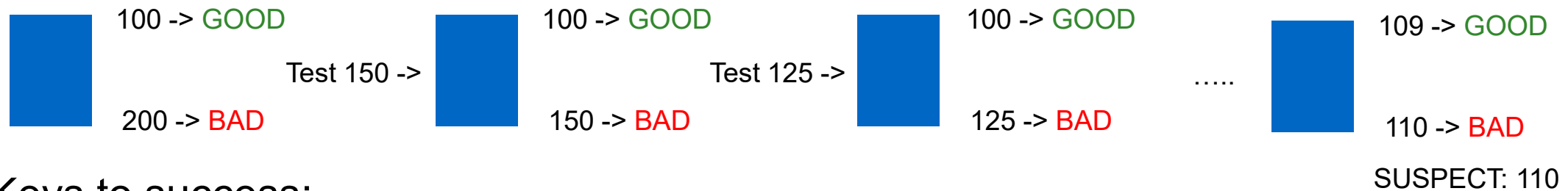
When things go wrong: Bisect & Autoheal

- Key Technologies: Bisect & Autoheal
 - Automatically find bad changes, and revert them from the line.
 - Bisect -> Find first change that broke build or CIT
 - Autoheal -> Apply 'p4 undo' to bad change, validate, submit
- Autoheal fundamental to maintaining + improving quality
 - Protected areas called 'autoheal-layer'
 - Autoheal-layer enables quality ratcheting
 - Add tool/test/sanitizer to autoheal layer, autoheal keeps it clean

Bisect (details)

■ Bisect:

1. Run specific build/CIT on cadence.
2. When cadence fails, record the last known good change & first failure change
3. Pick a change in-between... see if it passes.
4. Update last-good/first-bad. Go to 3 until we've identified the SUSPECT change that causes a failure.



■ Keys to success:

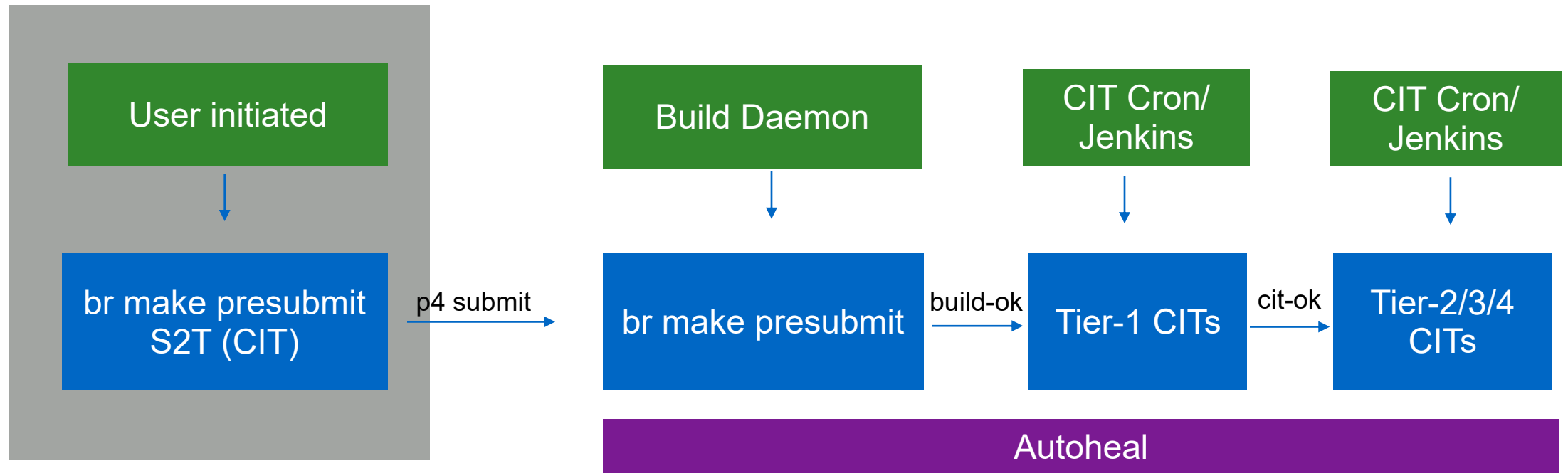
- Minimize external dependencies... Or version them by a p4 change.
 - The same change should fail today and next week
- Run multiple tests in parallel.
- Premake clients at important changes. (before bisect needs them)

Autoheal (details)

- Autoheal

- Validation – Is bisect right?:
 - Re-run at the first failed change, make sure it fails.
 - Re-run at head-of-line with SUSPECT change reverted, make sure it passes.
 - Validate: All changes before suspect change MUST pass, and all runs after suspect MUST fail.
- If yes... Submit the revert, and email the user & manager:
 - Change that was reverted and test that failed
 - Instructions to recreate the client, how to run the test.
- If no... Send message to Build/CIT team warning of intermittent error

Regression Protection layers



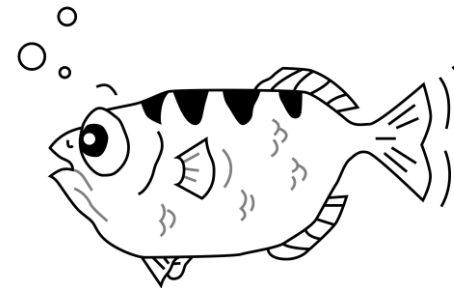
'br make presubmit' -> **Build** and much more

- wsget -> get a flex-cloned client <1 minute
- br make presubmit (~10 minutes)
 - Enforce coding standard/static analysis: (fail if violated)
 - clang-format: require code in Netapp coding standard
 - include-what-you-use: remove unneeded includes
 - clang-tidy: validate C/C++ code
 - Python (pep8): passes clean
 - Man pages: missing commands?
 - Gdb macros: still work?
 - ...
 - Compilation: (fail on warning)
 - Compile w/ aggressive Clang warnings

```
MAN(1)                                Manual pager utils                                MAN(1)
NAME
    man - an interface to the on-line reference manuals
SYNOPSIS
    man [-C file] [-d] [-D] [--warnings=warnings] [-R encoding] [-L locale] [-m
    system[...]] [-W path] [-S list] [-e extension] [-i-i] [--regex=regex]
    [--names-only] [-a] [-u] [--no-subpages] [-P pager] [-r prompt] [-7] [-E encod-
    ing] [--no-hyphenation] [--no-justification] [-p string] [-t] [-T[device]]
    [-H[browser]] [-X[doll]] [-Z] [[section] page ...] ...
    man -k [apropos options] regexp ...
    man -K [-w-W] [-S list] [-i-i] [-l] [--regex] [section] term ...
    man -f [whatis options] page ...
    man -l [-C file] [-d] [-D] [--warnings=warnings] [-R encoding] [-L locale]
    [-P pager] [-r prompt] [-7] [-E encoding] [-p string] [-t] [-T[device]]
    [-H[browser]] [-X[doll]] [-Z] file ...
    man -w[-W] [-C file] [-d] [-D] page ...
    man -c [-C file] [-d] [-D] page ...
    man [-?V]
DESCRIPTION
    man is the system's manual pager. Each page argument given to man is normally
    the name of a program, utility or function. The manual page associated with
    each of these arguments is then found and displayed. A section, if provided,
    will direct man to look only in that section of the manual. The default action
    is to search in all of the available sections following a pre-defined order ("1
    n l 8 3 0 2 5 4 9 6 7" by default, unless overridden by the SECTION directive
    in /etc/man.db.conf), and to show only the first page found, even if page
    exists in several sections.

    The table below shows the section numbers of the manual followed by the types
    of pages they contain.

    1 Executable programs or shell commands
    Manual page man(1) line 1 (press h for help or q to quit)
```



'br make presubmit' -> Build **and much more**

- br make presubmit (~10 minutes)
 - Unit-test execution:
 - Run ~38k CxxTest-based linux unit-tests (<5 minute execution)
 - Address+Undefined+Memory sanitizer for all unit-tests
 - Valgrind for a subset of unit-tests
 - Thread sanitizer for a subset of unit-tests
 - Linux-based simulator testing (<5 min)
 - Execute workflow tests on a pared-down version of ONTAP
 - Libfuzzer corpus execution (<5 min)
 - Run checked-in corpus w/address sanitizer.
 - Code coverage (<5 min)
 - Generate UT code coverage information (including coverage of pending change)



LCOV - code coverage report


Current view: top level

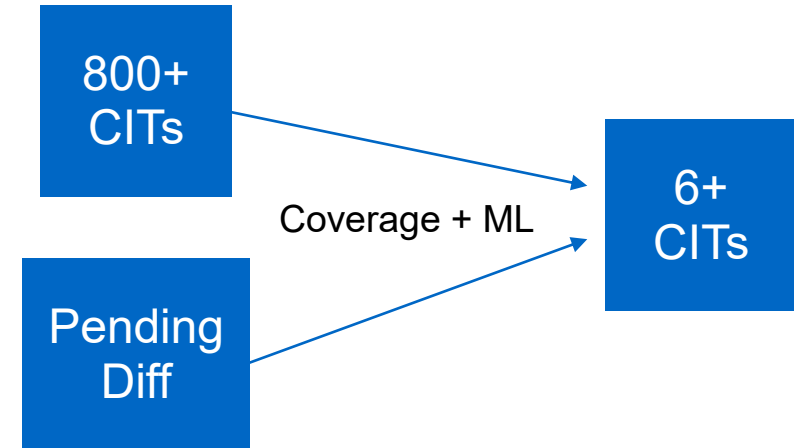
Test: coverage.info	HR	Total	Coverage
Date: 2011-11-20	Lines: 93372	154764	60.3 %
	Functions: 4538	8656	52.4 %
	Branches: 56337	140803	40.1 %

Directory	Line Coverage	Functions	Branches
/home/melanson/libav	56.0 % 1475 / 2635	59.0 % 62 / 105	48.6 % 1029 / 2117
libav/include	80.8 % 2 / 24	-	50.0 % 4 / 8
libav/include/bits	54.1 % 29 / 54	-	18.0 % 7 / 39
libav/include/ogg	100.0 % 2 / 2	-	0 / 0
libavcodec	65.3 % 14049 / 21511	55.3 % 2627 / 4741	41.3 % 40833 / 98725
libavcodec/x86	64.8 % 1522 / 2347	36.1 % 410 / 1137	32.1 % 343 / 1070
libavdevice	1.4 % 12 / 854	5.5 % 3 / 51	0.2 % 1 / 464
libavfilter	34.1 % 1290 / 3782	37.3 % 112 / 300	29.6 % 590 / 2002
libavfilter/ast	0.0 % 0 / 28	0.0 % 0 / 0	0.0 % 0 / 24
libavformat	52.5 % 20519 / 39102	59.0 % 1025 / 1730	39.4 % 9413 / 23899
libavutil	71.7 % 1937 / 2701	74.4 % 160 / 215	44.5 % 1315 / 2957
libavutil/x86	69.4 % 43 / 62	100.0 % 1 / 1	37.9 % 22 / 58
libpostproc	0.0 % 4 / 482	7.4 % 2 / 27	0.0 % 0 / 358
libswscale	47.4 % 1613 / 3401	38.4 % 117 / 305	31.7 % 2187 / 6902
libswscale/x86	22.9 % 239 / 1043	11.9 % 18 / 151	34.6 % 404 / 1169
tools	80.0 % 36 / 45	100.0 % 1 / 1	79.2 % 19 / 24

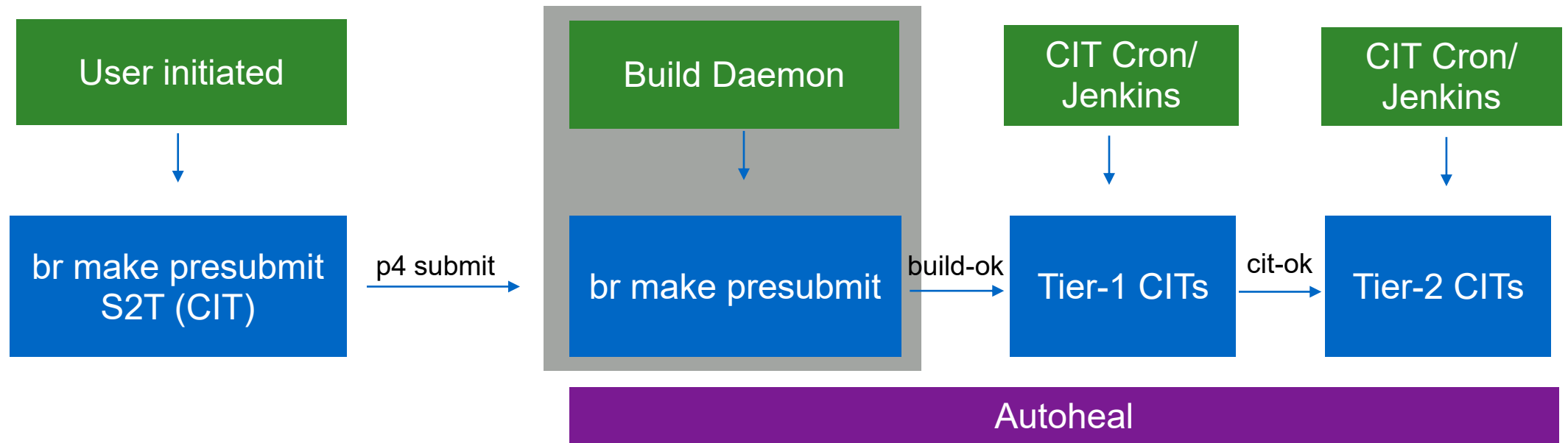
Generated by: LCOV version 1.9

Get Ready for Submission

- source-2-test (S2T)
 - Client diff + CIT coverage data -> pick 6+ CITs to run
 - Coverage analysis algorithm + machine-learning
- Submit review to reviewboard 
- p4 submit (w/Netapp additions)
 - Validates you've built the pending changes ✓
 - Validates that S2T has passed ✓
 - Checks for pending conflicting changes ✓



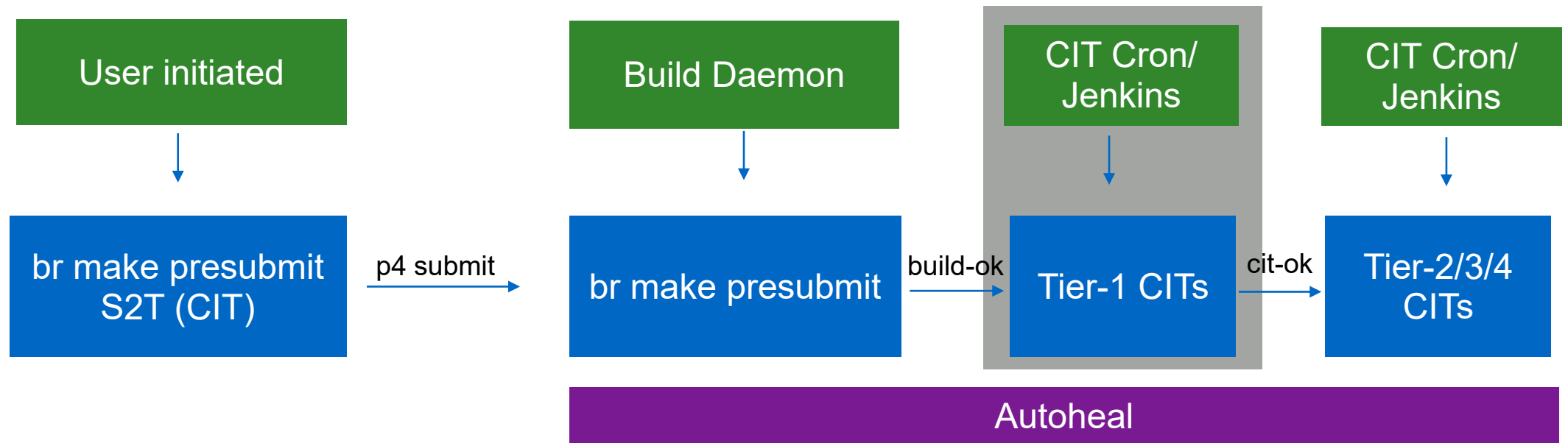
Regression Protection layers



Post-submission: build-ok

- Bammbamm daemons wake up and build change. (every 20 min)
 - If 'br make presubmit' passes create new ws* snapshots, and stamp the change 'build-ok'.
 - If it fails, start bisect.
- Autoheal:
 - Automation bisects to find the change that broke the build.
 - Once guilt is verified, automatically revert change from the line. (ie. Submit an inverse of the bad change)
 - Email user about revert and how to reapply.
- Bammbamm daemon will sync forward to try again
 - If build @change passes, stamp change as 'build-ok'
 - Implications: wsget clients (which use build-ok) will always build AND in-build unit-tests will always pass.

Regression Protection layers



Post-submission: CITs (tier-1)

- Continuous integration tests (CITs)
 - ~100 2-hour tests -> tests ONTAP and OFFTAP in the smoke bubbles.
 - Mainly VSIM, with some HW.
 - Run every 3-hours on latest build-ok.
 - If all tier-1 CITs pass on a given change, the change is stamped 'cit-ok'
- Autoheal for CITs
 - If any CITs fail, the offending change is bisected, and autohealed out of the line.
- CITs
 - Have strict requirements on intermittent failure rates. (<5%)
 - Require a dedicated sheriff, who must triage all failures. (+ mailing list named after cit)
 - 24-hour operational support across multiple Netapp sites.
 - If cit-ok isn't stamped within 24-hours, line is locked and fixed.

CIT: Week at a glance (WAAG)

dev (VR.0)	cov	AVG	Tue	Wed Jul 04	Thu Jul 05	Fri Jul 06	Sat Jul 07	Sun Jul 08	Mon Jul 09	Tue
cit-ok	runs	TIME		* .	N	N	N	N	N *	N
cit-adr	17	1:44								
cit-appdm	3	1:46	B							
cit-appdm-vvol	3	1:13								
cit-c2c-cp-restart	7	1:12								
cit-cft	2	1:14					B			B
cit-cifs	4	1:53			B		R			
cit-cifs-admin	3	2:00					R			
cit-cifs-ext	3	2:01				H H H				H
cit-cifs-mscomp-mc	17	1:30								?
cit-cifs-multichanl	3	1:28				?				?
cit-cifs-solutions	17	1:57					R			
cit-cifs-vdr	4	2:13			B	I				B
cit-clone	16	1:23								
cit-cop-core	17	1:44			B		B			H
cit-coresw-sas	1	1:40							I	
cit-cov	16	1:26					H			
cit-csi-4node	11	1:49	I I I	I			B R			
cit-csi-support	16	1:37								
cit-dpgsystemic	16	1:42						B		
cit-dps-lsa	16	1:33								
cit-fc-core	16	2:00					R			?
cit-ffo	3	0:55								
cit-fg-admin	17	2:07			B	B	R			
cit-fg-adr-core	17	1:57	H H				R			

Your change hit cit-ok (email)

Hello user,

The [CIT-OK](#) marker on DOT:dev has moved from [4954128](#) to [4954794](#), and these recent change(s) of yours are now CIT-OK:

Change Number	Change Description	Burt Associated
4954359	1) Create a kernel version of <code>ems_helpers</code> . (Since almost all of the code is the same, I just recompile the same...	1172664

This is not an absolute guarantee that your change(s) will not be reverted, but it is a good indication that it has not caused any serious issues.

Please consider using `wstakechange` for propagating your changes to other codelines.

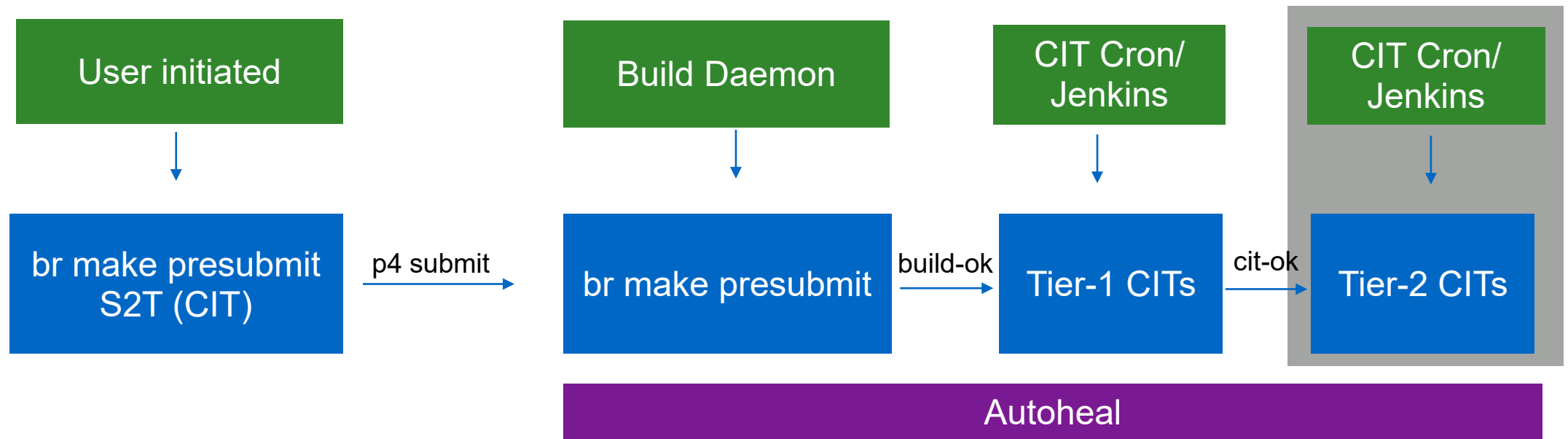
E.g. To propagate change #11111 to DOT:Rfullsteam and run build/smoke tests for verification:
`wstakechange -c 11111 -d DOT:Rfullsteam -t build,smoke`

Alternatively, you can use "[p4 take_change](#) -state auto -c new changenum" to bring these changes into applicable prior releases.

Regards,

Build Team

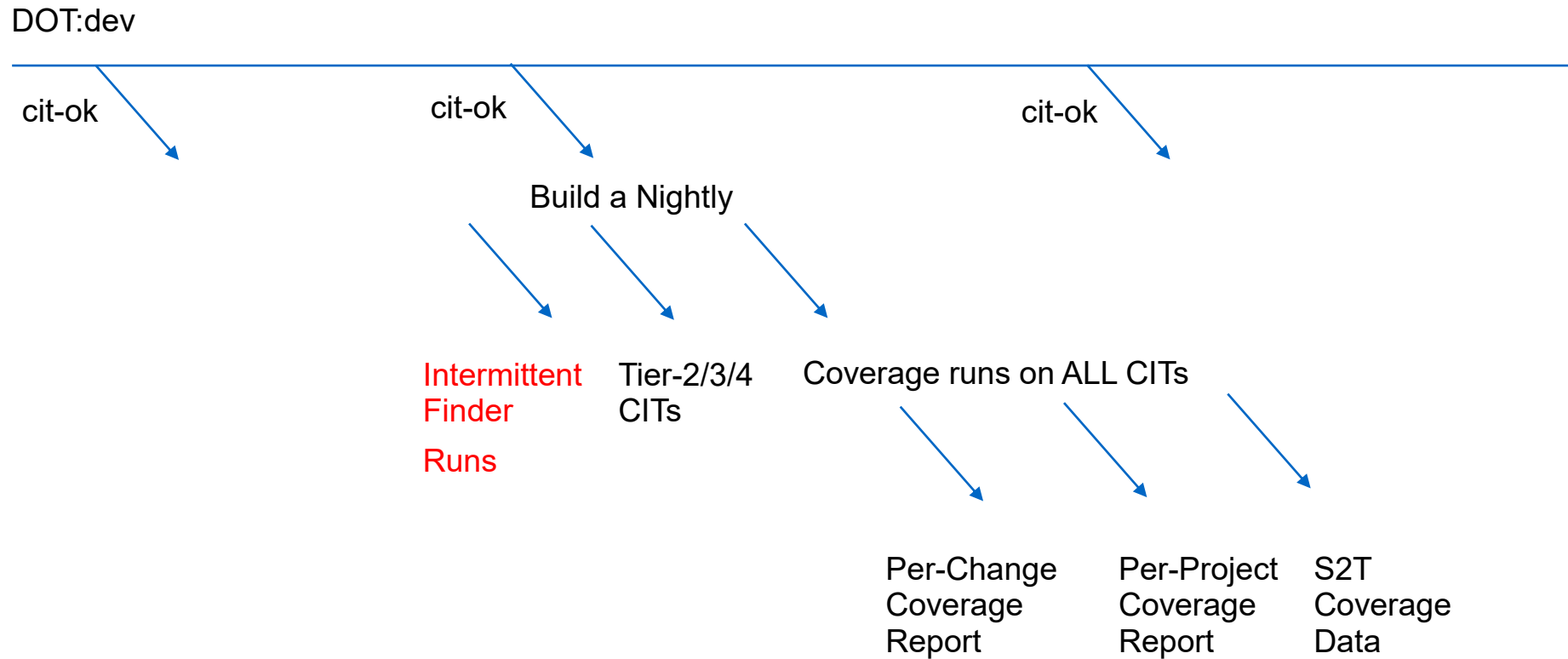
Regression Protection layers



Post-submission: CITs (tier-2/3/4)

- Tier-2/3/4 CITs run at lower cadence
 - ~500 tier-2/3 CITs, ~250 tier-3
 - Follows all the requirements of CITs
 - Typically 'lower-risk' CITs. (higher-coverage tests are pushed to tier-1)
 - Tier-2/3 -> Runs daily on a cit-ok build.
 - Tier-4 -> Run weeks on a nightly build.
- Failures are autohealed out of the line.
 - Bigger change range to bisect over, but will eventually be reverted. (a few days rather than hours)
 - Does NOT block cit-ok... so errors may linger longer and can be present in a cit-ok build.

DOT:dev – Beyond cit-ok








Driving out intermittent errors

- Weekly: Run all tier-1 CITs 50X times on a cit-ok change.
 - This CIT passed at the given change, so....
 - Failures must be due to intermittent issues in infra, product or test code.
 - Regular runs help identify WHEN issues started to occur.

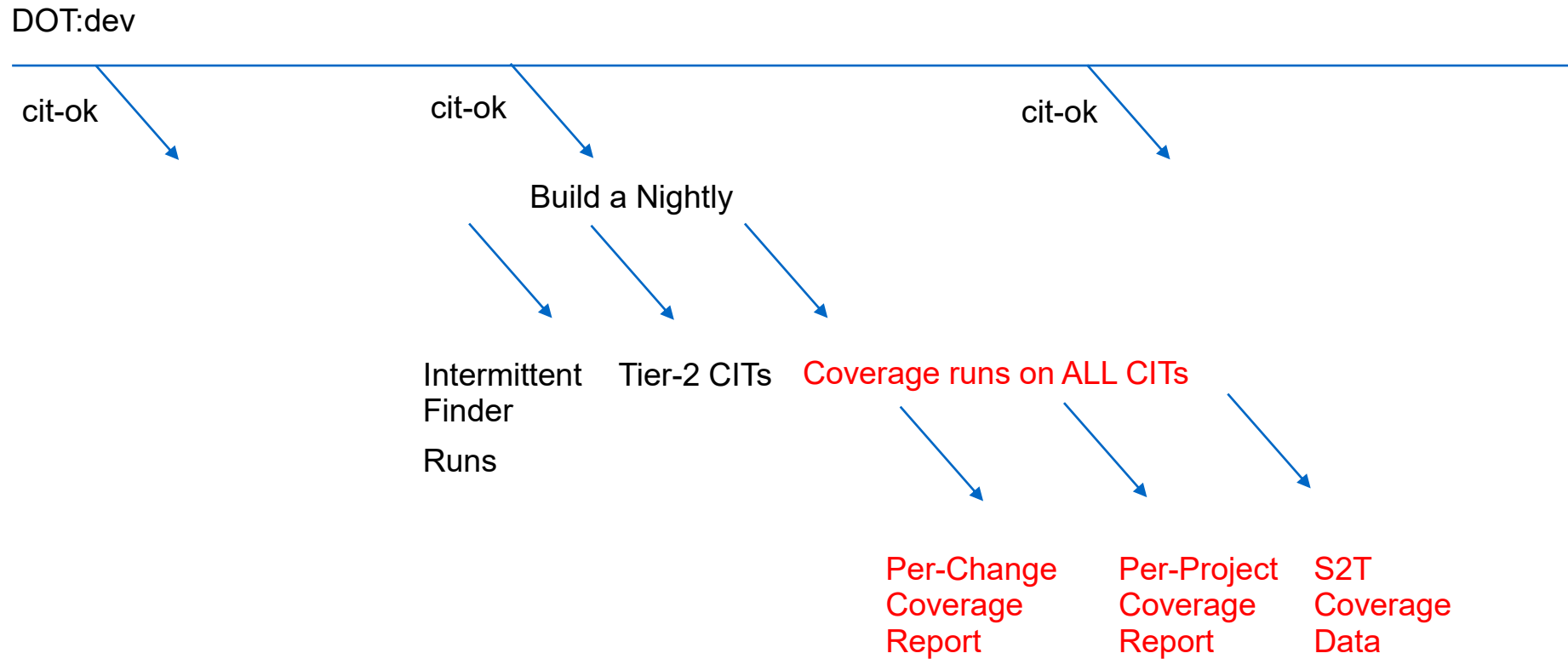
- Status tracked in summary page:
 - All failures must be triaged and driven out.

- Intermittent bisect:
 - Given a failure rate, a good & bad change, a CIT + test case,
 - We can track down which change introduced an intermittent error (within a given confidence level)

Intermittent Runs: Display results of last round(s)

CIT Test	4949120 (NA)			Average Run Time for 4949120
cit-adr	v64d 	v64nd 	0%	01:32:10
cit-appdm	v64d 		2%	01:40:22
cit-appdm-vvol	v64d 		0%	01:14:47
cit-c2c-cp-restart	v64d 		2%	01:14:10

DOT:dev – Beyond cit-ok



Generate coverage data

- Coverage variants of every (850+) CIT are run on the latest nightly.
- Data is gathered from the filer, combined with the in-build unit-test coverage data
 - Post processed -> human readable. (~18+ hour process)
 - Post-processed -> machine readable for quick source-to-test (S2T) analysis.
- "Coverage in the autoheal layer" -> In-build UT + CIT tier-1 + CIT tier-2
 - Projects don't ship if they don't reach autoheal coverage targets.

Per-Change/Per-Project Coverage Report

- Per-Change: Send developers reports on autoheal coverage of every submitted change.
- Per-Project: Aggregate coverage of all change for an ONTAP project into one report.
 - Each project has UT and Autoheal coverage goals.. Don't ship until hit.
 - Project reports are recalculated nightly with fresh code-coverage data:


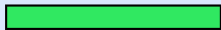



LCOV - code coverage report

Current view: **top level**

Test: **/x/eng/bbrtp-nightly/builds/DOT/devNightly/devN_180708_0746(autoheal)**

Date: **2018-07-09 15:36:53**

	Hit	Total	Coverage
Lines:	3369	5948	56.6 %
Functions:	0	0	-

Directory	Line Coverage ↕			Functions ↕	
apps/lib/libfiji/src		72.7 %	8 / 11	-	0 / 0
apps/lib/libtimed_threadpool/src		100.0 %	1 / 1	-	0 / 0
cro_proxy/cro_proxy_mgwd/src		64.4 %	58 / 90	-	0 / 0
cro_proxy/cro_proxy_mgwd/src/tables		68.8 %	22 / 32	-	0 / 0
cro_proxy/cro_proxyd/src		42.3 %	721 / 1705	-	0 / 0

Using Netapp Solutions to make DevOps work #1

Go Faster -> Start Developing

- Problem: To submit code developers need a workspace
 - They want to go fast:
 - Quick access to a new workspace
 - Fully built: Enables incremental build of JUST their changes
 - From a known good point: must build and pass tests -> Don't waste time dealing with a bad change.
- The solution: ONTAP Snapshots & FlexClones
 - Continually build codeline: Sync to head-of-line, attempt build, and take a snapshot.
 - If builds pass, stamp that snapshot 'build-ok'
 - If tests pass, stamp that snapshot 'CIT-ok'
 - When developer needs a workspace, FlexClone the latest build-ok or cit-ok
- Does it work?
 - Quick access to a new workspace -> 1 minute for 10+ million lines of code. ✓
 - Fully built: Enables incremental build of JUST their changes -> 300G client ✓
 - From a known good point -> Pick 'build-ok' or 'cit-ok' ✓



ONTAP Engineering (After)

- ONTAP in 2020: Did quality/cadence improve?
 - Customers needed:
 - New features to be delivered faster than every 3 years : ✓ New releases come out every 6-months
 - New features to work on the first release: ✓ All features must be protected by Autoheal to ship
 - Old features to keep working : ✓ UT/CIT kick-out changes which cause regressions
 - Engineers needed:
 - A codeline this isn't always broken ✓ Developing against cit-ok, means you have a good client.
 - An escape from the end-of-release "test-to-health" ✓ Testing + Autoheal means we never regress
 - A reliable way to test whether their changes work and didn't regress other features. ✓ Devs can run UT/CIT
 - A way to efficiently collaborate on features across groups and sites ✓ We're all in one branch.
 - Each subsequent ONTAP release becomes the highest quality ONTAP release
 - Disruption/Node count drops every release
 - Continuous Delivery: We use cit-ok builds to run our CIT infrastructure



Thank You

CIT: Triage/Operation -> Jenkins

- Jenkins (stuck in the middle)

- Clearing-house for CIT results.



- Blends into preexisting infrastructure

- Preexisting processes -> trigger Jenkins jobs -> trigger other Preexisting processes.
- Can spin up Jenkins instances/slaves in different test/compute environments.

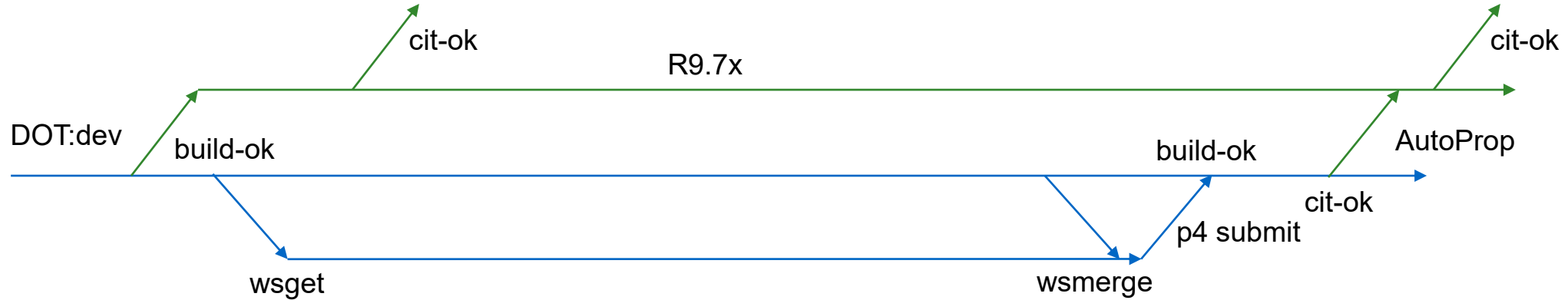
- Jenkins gathers results, and allows for triage of each failure

- Homegrown tools wrapped around Jenkins to make common triage easier.
- Tooling created to automatically add new CITs

What about shipping releases?

- Release branches hang off of DOT:dev
 - Release testing: focus on DOT:dev as long as possible
 - Release fixes submitted to DOT:dev first
 - Put high risk in the development branch.
 - DOT:dev is often more strict (because quality gates show up there first.)
 - Changes that pass in DOT:dev are pulled back.
 - Every change: May request propagation to release branches.
 - Hit cit-ok -> individual changes are automatically propagated back (auto-prop)
 - Any future reverts of those changes are ALSO auto-proped back.
- Release branches run CITs as well, but at a reduced cadence.

DOT:dev - Life of a Change (Release)



Your change hit cit-ok

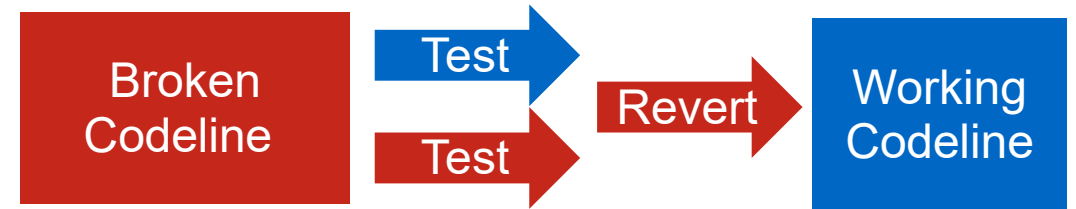
- Autoprop starts
 - Requested changes are applied to release branch client.
 - If it can be applied and builds, it is submitted.
 - If not, user-gets an email with details and manual instructions about how to take it.

Using Netapp Solutions to make DevOps work #2

Go Faster -> Eject Bad Changes

- Problem: Remove bad changes as fast as possible

- Autoheal automation needs to run test against changes between last good and first fail.
 - Have instant access to all prebuilt changes in good/bad range.
 - May need to build missing variants



- The solution: ONTAP Snapshots & FlexClones

- Continually build codeline: Sync to every new change, make build, and take a snapshot.
- All changes on the codeline are prebuilt for testing.
- When autoheal needs to test a change, it flexclones the snapshot at the given spot, and starts testing.

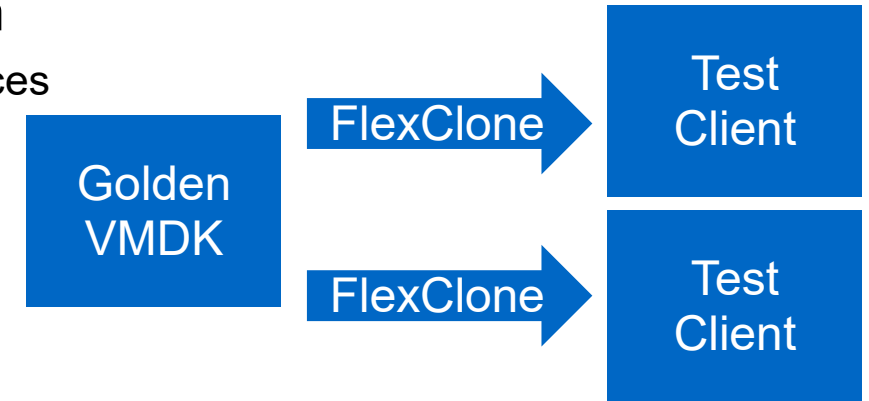
- Does it work?

- Have instant access prebuilt changes in good/bad range -> All changes are available after ~1 minute clone ✓
- May need to build missing variants -> Flexclones can diverge after creation, so additional variants are built as needed ✓

Using Netapp Solutions to make DevOps work #3

Go Faster -> Provide Consistent Test Environment

- Problem: VMs used for testing need to be consistent and clean
 - Testers spin up and use Virtual Linux/Win Clients and virtual ONTAP instances
 - Bring hosts to known good state
 - Client configuration identical for all test runs
 - Testers can modify local instance if needed
- The solution: NFS, ONTAP Snapshots & FlexClones
 - Host VMDKs on cluster over NFS
 - Create golden VM with minimally viable packages
 - Allocate Flexcloned VM image when test starts
 - Allow user to modify VM, but 'snapshot restore' to known good state when test done.
- Does it work?
 - Bring hosts to known good state -> Snap restore erases any changes a test may make ✓
 - Client configuration identical for all test runs -> Consistent configuration deployed from golden ✓
 - Testers can modify local instance if needed -> It is safe since all changes are erased at test end. ✓

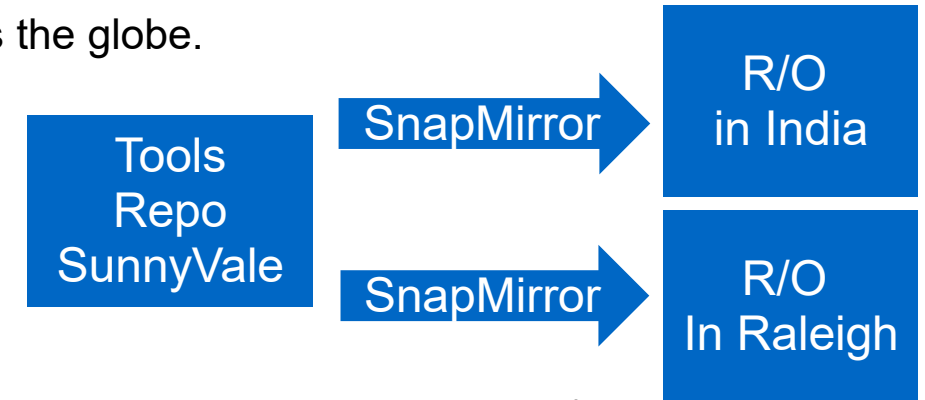


Using Netapp Solutions to make DevOps work #4

Go Faster -> Provide Consistent Tools across the globe

■ Problem: Consistent version of tools needed for multiple development sites.

- A consistent tool chain needs to be deployed across multiple sites across the globe.
 - All sites need a local copy of tools.
 - Changes must be deployed quickly.
 - If things don't go well, quickly roll back to previous versions must be possible.
 - Engineers can't modify deployed tools.



■ The solution: ONTAP Snapmirror and Snapshots

- Source contains snapshotted version of tools to deploy. (Some deployed weekly, some every 5 minutes)
- Tools are read-only snapmirrored to sites around the global (by default latest version is in production)
- Local admins can quickly rollback to previous versions if necessary

■ Does it work?

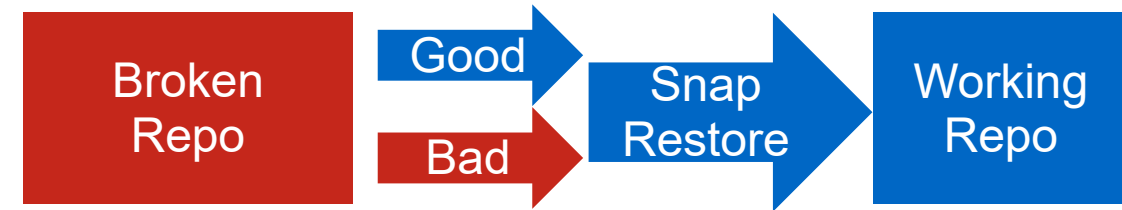
- All sites need a local copy of tools -> Snapmirror deploys changes to each sites ✓
- Changes must be deployed quickly -> Snapmirror only pushes the changes. ✓
- If things don't go well, quickly roll back to previous versions must be possible -> Easy to rollback to previous versions. ✓
- Engineers can't modify deployed tools -> Snapmirrors are read-only. ✓

Using Netapp Solutions to make DevOps work #5

Go Faster -> Undo the disaster

- Problem: Rogue scripts/users may accidentally corrupt infrastructure

- Labs are fairly open environments, but user error can impact infrastructure
 - Give generous permissions to tools/tests.
 - In case of disaster recover quickly.



- The solution: ONTAP Snapshots

- Put important configuration on volumes with (read-only) snapshots.
- In case of corruption/deletion, move infrastructure back to known working point.

- Does it work?

- Give generous permissions to tools/tests -> Although the latest copy can be changed, snapshots remain read-only. ✓
- In case of disaster recover quickly -> In case of corruption, snapshots mined or rolled back to working versions. ✓