Keeping trunk releasable

When your goal is to have stable releases on a defined basis (releases don't slip) you need a place to aggregate all the code for each release that you believe is ready to ship. You can't realistically create stable releases that include code that isn't (stable && validated).

You can create release branches and attempt to stabilize after the fact, but you lose a lot of locality and context (both for individuals, and across the team) around the work being done and the validation necessary to prove it is ready to release.

If we release off trunk it is pretty much necessary for trunk to be in a releasable state all the time. Allowing trunk to become unstable means you always have to schedule in time to stabilize and it means you aren't including the cost of testing into estimates of release dates and active work.

That means at no point can you merge code into trunk that isn't "done" a.k.a. ready to release. By drawing a line in the sand around merging to trunk we can enforce a team wide definition of what it means for new work to be done. This is critical because if we compromise on the definition of done then we are pretty much guaranteed to start shipping changes that don't work or be caught in a situation where we want to ship and have to assemble a new branch of effectively untested or less tested code.

This is a definition that we can tune over time as we find out what works and what doesn't, but I want to start by outlining what I think we could use as an initial definition of done.

Testing tools

So what are the tools we have?

WIth unit tests we can create fast tests that are relatively easy to write and can be run on every commit for every developer. They can be tightly coupled with implementation and can have direct visibility into execution. What unit tests don't capture is interactions between units and even when they do it is rarely in a realistic or complete way.

So when is a unit test enough? If you are working on something that isn't user visible then a unit test is going to be the only way to reach the code. Hypothetically there is already a functional test exercising the dependencies that are user visible. More on missing tests later.

With dtests we can test things end to end from the users perspective (functional testing), but it isn't real, and as a harness it doesn't attempt to explore the configuration space. We could probably grow dtests to serve as a complete harness for functional testing. However we might end up regretting trying to stuff the entire functional testing harness, cluster, and clients onto one node if we find we start running into timeouts or configurations that can't be expressed as a

local cluster. Running dtests as a special case of a more flexible harness might be one way to go.

The biggest missing piece right now is a functional testing harness to test realistically, to test the interactions between features, and to explore the configuration space. Such a harness would have a client validating the behavior of the database while the full feature set is being exercised. I think the harness and Jenkins aspect of this is something we can ask QA for, but it will involve us because we are going to be the users of the harness along with QA. When you go to implement a feature and need a functional test you will be faced with doing it yourself or asking QA to do it and suffer the additional latency before merging.

I am pushing to have QA run the unit tests and dtests on every commit in our forks on github and give each of us a page with our branches. You should never have to run the entire test suite again because Jenkins should run it faster and it only matters if it passes in Jenkins anyways. By the time you are done with code review and ready to merge you should already have an answer as to what will happen once you merge. If someone sends you a patch you can (and should) make a branch for it to make sure Jenkin accepts it.

Functional tests tend to be longer running and running them on every commit isn't feasible. Once we have a useful set of functional tests the goal will be to have those run 24/7 in rotation so that each tests runs frequently enough that we discover regressions soon enough to tie them to specific commits. The plan is that Michael Shuler is going review the output of these and keep on us for issues as they come up.

Our ability to write functional tests is still limited at this point. I would say that we are blocked by the lack of a harness, and I am pushing to have QA provide us with that. An initial implementation would simply be a Jenkins' job running stress against a cluster with an eye towards being able to plug in additional manipulations of the cluster as well as additional validation into stress. I am working with QA to get the seed planted at which point we can incorporate functional testing into the validation workflow and testing requirements of each task.

What testing constitutes done

If you have been down the agile rabbit hole you have probably heard about "user stories" which describe bugs, features and enhancements, and refactorings. Stories are the end goal and shouldn't restrict how the implementation or validation occurs.

A story might be "As a user I want to delete data associated with partitions that are no longer assigned to a given node." We are good at refining the story into a design and implementation, but not good at ensuring the story is done.

At some point the assignee and a reviewer have to decide whether the change and it's associated tests are done and ready to merge. That means the assignee and reviewer are persuaded that the automated tests (unit, dtest, blackbox system tests) demonstrate the feature

works when interacting with all the other features and configuration space on a real cluster. Very few things meet this criteria at this point because it is rare to explore the configuration space much less validate the interactions between features.

As an assignee you are judged by how many bugs fixes and features you can get done. You have every incentive to get the testing for your work done as early as possible so your code is merged saving you the trouble of constantly rebasing and allowing you to move on. If you are going to enlist help from QA in implementing some of the dependent tests you will want to figure out what new tricks the tooling is going to have to learn and incorporate that into your plans.

If what you are working on/reviewing is a documented way of interacting with the database then it needs to be included in a functional test and not just a unit test. If there is a likelihood //of interacting with other features/functionality/implementation then that functional test needs to be in a harness that is going to test that interaction. You may want to skip the dtests as they stand anyways because they are still unit tests and don't have a built-in fuzzing provided by the harness test client.

The end goal when shoring up functional tests is to avoid scenarios where users exercise the database normally, but discover bugs and interactions that could have been tested for before release. For every bug and regression we should be asking why it wasn't caught. For some it will be because it was genuinely complicated or hard to reproduce and it is unlikely we would have ever emitted a test to catch it. For others it will be because we didn't test something that we claimed to support.

It's the things we should have caught that we want to focus on, and to prevent it from happening again we can refine our definition of done until encompasses all the things we think we can address before release.

Missing tests

In the brave new world of doneness we need a way to iteratively progress towards having coverage for everything while still making forward progress on new development. The campfire rule only goes so far and we need to agree on when touching something requires adding missing coverage.

Existing code is assumed to work more or less because we shipped it so it is ok to leave it be. It's not ok to change existing code without coverage and assume it continues to work just because you added a unit test. If you add dependencies to existed untested code it isn't quite as bad, but still problematic because your new code may exercise it differently.

I would say that missing unit test coverage is less bad than missing functional test coverage. You can't can't can't change existing code that doesn't have functional test coverage. That's taking a shot in the dark. Unit tests are there to make our lives easier not prove the product works as intended. You do have to pretty much prove the product works to get reliable releases.

Role of QA

Our relationship with QA has two facets. One is that we are clients of QA and users of their products. The other is that they have a different perspective, priorities/incentives, and experience from us and are equipped to test differently and we need to be transparent enough to allow that.

We are their clients in the sense that they manage Jenkins (it's a big job when you start trying to do a lot with it and make it responsive), building a harness to make plugging in new functional tests easier, provisioning and managing hardware for tests to run on, running performance jobs and visualizing the results, triaging test output.

We need to communicate what we need, and we need to do it early enough that we can meet our own scheduling goals. If you are implementing a new feature and have functional test needs (harness needs to learn a new trick) you need to let them know as soon as possible so it can be scheduled or be prepared to do it yourself. Since we are releasing frequently it shouldn't matter if something slips.

They are also responsible for doing their own validation, but we should be presenting them with functionality that we are already convinced works and wait to be proven wrong. We still need to ship our own functional tests because the white box knowledge we have when we implement is an important starting point for writing functional tests. The flip side is that coming at the problem without white box knowledge also changes how testing is done.

QA may also come back with requests for JIRA process improvements.

Retrospectives

I think we should do a monthly or every other month email retrospective where dev and QA chime in on

- What worked
- What didn't
- What we are going to change (based on previous two)

You don't need to say anything if you don't have anything to say, but you should read it.

This is a forum for discussing process not individual bugs or features. You would only bring up an individual bug or feature if you had something to say about how our process worked/didn't work in that instance.

I volunteer to facilitate the retrospectives..

TL;DR

- Jenkins must approve branches before merging

- We need to agree on and religiously enforce done-ness criteria as assignees and reviewers
- Done-ness needs to include richer functional tests and QA will help with our support
- Things that aren’t done don’t get merged to trunk