Optimizing Turnaround Times In A Continuous Integration Flow Using A Scheduler Based Implementation

Robert Strong
Samsung Austin R&D Center
r.strong@samsung.com

I. INTRODUCTION

Continuous Integration (CI) flows have become an integral part of chip design. We had implemented a "Parallel Gated Checkin" model from previous DVCON papers[1] for past projects. The use model of this flow required every user change set submission to the code repository (repo) to pass a test case that would ensure it would still compile and run all essential tests before being allowed into the repo. When many user change sets were submitted it was essential that they can also pass this test case when combined with each other. Originally our "Parallel Gated Checkin" model had the basic logic to combine edits, run test cases, remove failing changesets and push passing changesets to the repo.

Although this proved very valuable to the development process, the first iteration of this logic did not scale well with a large user base and high volume of submissions. The flow could slow down dramatically with a large number of submissions and a high failure rate as it tried to determine which changesets were causing failures. As a result we encountered high turnaround times between when a user requested their changeset to be pushed to the repo, and when it was accepted or rejected. With a testing time of 1 hour and 45 minutes we were seeing average turnaround times around 8 hours. In addition to recording high average turnaround times we observed turnaround times that were extremely variable.

The feedback from this was that the CI flow became viewed as a burden or hurdle by users rather than a useful flow that saved the development process from constant headaches. Users could not count on a change being accepted by a certain time for a nightly regression. Users could not count on a morning submission being accepted for other users to pull in by the afternoon. Users could not count on a failure being returned to them during the workday to debug and resubmit before the end of day.

This paper will review the flow and logic changes we have implemented in Jenkins to reduce the turnaround times of user submissions using a scheduler based flow.It will also demonstrate how a scheduler flow can scale into the future and keep turnaround times at or near the testing time of a changeset if an organization is willing to allocate the resources.

II. IMPROVEMENTS

To reduce overall turnaround times we identified 3 areas of focus

- Minimizing the cost of running any individual test case & eliminating unnecessary overhead in the flow
- Increasing the number of testcases run in parallel to find passing & failing testcases more quickly
- Migrating the flow to be event driven. The flow should push changesets, fail changesets, kill testcases and launch testcases as soon as the information is available.

The first two areas of focus were worked on in parallel. We implemented incremental changes in production throughout the project. The last area of improvement, migrating to an event driven flow, was developed and tested on the side. It was rolled out overnight in a large migration. We will focus on the first two sets of improvements before addressing the large event driven migration

III. MINIMIZING THE COST OF TESTING ANY CHANGESETS

We focused the push on our flow to prioritize turnaround time over compute, disk, license, and other resources. In this push, wasteful practices in our flow became magnified. We knew we wanted to push the flow to lower turnaround times using a larger of testcases. To achieve this, we needed to treat each testcase as a blackbox. The

blackbox accepts any number of changesets, combines them, returns a yes or no to the flow, and then cleans up after itself while using as few resources as possible.

In our push towards this "black box" idea, we identified four areas of focus.

1. Exiting immediately on failures to reduce unnecessary compute, and to give the flow information as quickly as possible.
2. Identifying false fails due to the batch system efficiently to avoid disruptions and false information to the flow.
3. Auditing user changes coming into the repo to catch external dependencies that can introduce random fails.
4. Recycling workspaces and taring failure logs in a separate area to save overhead time, and I/O churn.

Prioritize exiting quickly on fails

The first push to reduce the cost of testing a changeset was to push failures to propagate up through our flow as quickly as possible. From the individual test level to the test suite level, we wanted fails to exit as soon as they were detected within our flow. The continuous integration flow is not responsible for generating a complete lists of fails for users. The costs of doing this is not worth the delay it adds to other users. Once one fail is detected, the submission is removed from all testcases and sent back to the user to resolve all issues.

Strong Requeue System & False Fail Detection

The second push to reduce the cost of testing was to minimize the impact of random fails. Random fails create issues from two different aspects of our flow. Firstly, if a false fail is believed the flow will take overly aggressive steps to remove a changeset submission aborting testcases that would eventually pass. Secondly, if a false fail is identified and is restarted completely the flow will take twice as many resources as it originally needed.

Removing all false fails is nearly impossible in an environment with numerous external dependencies. However we worked to catch, remove, and lessen the impact of the most common cases. In our case, we relied on a large batch pool. Given a large number of testcases, the odds of a lone job from our test suite landing on a host with issues became significant. We added a requeue mechanism to our test suite that monitored each job as it completed and parsed for exit status' and text signatures that signified false fails. In these cases, we were able to restart individual jobs within our test suite rather than restarting the entire testcase. This also took changes at the job level to be able to clean the previous job and avoid collision when restarting on a batch host. The last job of a testcase could previously land on a bad host and fail it completely. This would waste large amounts of time and resources. Now it would simply retry the individual job and gloss over the hiccup from one bad batch host. In addition to our efforts to mitigate the effects of faulty hosts, our IT team also improved the detection and and removal of hosts to add stability.

For another level of stability we have a parser that looks for signatures that were not caught at the job level. This is a simple parser to catch issues caused outside the batch system. If the flow catches these it does have to restart the entire testcase, but it prevents the flow from killing other testcase from a false fail.

Finally, we have trackers watching the health of the entire queue. Once the conditions of certain failure rates or consecutive fails are met, it will automatically freeze the queue and text the admins for manual intervention. This helps prevent needless resource churn. It will stop submissions from being falsely removed from the queue during major issues.

External Dependencies & Line by Line Checking

Our next effort to remove instability from our flows was to eliminate external pointers from the repo. These could change outside of our version control and introduce failures. Either by accident or apathy it became a common occurrence for our test suite to accept a dependency on a user's local area. The result was users performing routine spring cleaning in their areas could break the entire CI flow. This would occur every few weeks, and would cause large disruptions. To fix this our first step was to root out all existing pointers manually. After the repo was clean of these dependencies we added a line by line text parser to our submission request script. We check every line of code for unapproved release areas of collateral. Adding this layer of checks eliminated these disruptions going forward.

Recycled Workspaces & Workpools

Our last effort to reduce the cost of running a testcase was to implement pools of workspaces. Originally, we wanted users to be able to debug failures in the testcase workarea to ensure all relevant logs and files were retained.

This proved to be overly cautious and very costly to our I/O resources and the amount of time spent on overhead. We worked to ensure we gathered and tarred the correct logs for failures to debug outside of the workarea. Then we created pools of workspaces that were being recycled after every testcase (non-blocking). Before each run, they are

checked to ensure they are still valid. If the number of testcases grows larger the the pool it will create more spaces on demand.

IV. Number of test Cases & Elimination Logic

In parallel to our efforts to reduce the costs of a testcase, we also pushed our flow to find passing and failing testcases more quickly. We increased the number of changesets being used to find out what would be accepted and what would be rejected. The original flow created fewer testcases to prioritize minimizing disk & compute resources usage over user turnaround time. Rather than using multiple possibly redundant testcases, it would use an optimistically low number of testcases to separate passing and failing changesets.

In the previous paper, the idea is established that "Given enough resources (i.e. machines and licenses), we could simply regress all possible combinations of pending releases in parallel."[1]. After examining the fails encountered when testing changesets, it became apparent that the majority of fails were caused by individual user changesets. The combination of two passing changesets creating a failing testcase was rare. Due to this, we decided to prioritize more machines and licenses to test each changeset in isolation. We also decided to change our combined changesets to run incrementally. If A,B,C, and D were submitted, we would run A, A+B, A+B+C, and A+B+C+D rather than A, A+B, and A+B+C+D. This incremental combination would give the flow more information every time a testcase completed rather than leaving the flow uncertain.

Increase The Number of Combined Changesets & Remove Fails Aggressively

Our first fix was to fix inefficiencies in the flows logic for combined changesets. The previous flow would only reject submissions tested in isolation at the front of the queue in integrate. In this example from the previous paper A, and A+B pass, but A+B+C fails. The flow should be able to determine C as the cause of the fail and remove it from consideration in one step. However, previously the flow would kick off a new run with C at the head of integrate. It required C to fail again before eliminating it using unnecessary time and resources.

These changes provided marginally better results in the flow, but the flow still saw very slow turnaround times. When it encountered a case with 3 submissions of A (Broken), B (Broken), C (Passing) it would run A, A+B, and A+B+C. With a 1.75 hour testing time, it would need 3 serial iterations before C would be accepted, resulting in a 5.25 hour turnaround time. As we scaled to up to 60 submissions a day around deadlines, this was untenable.

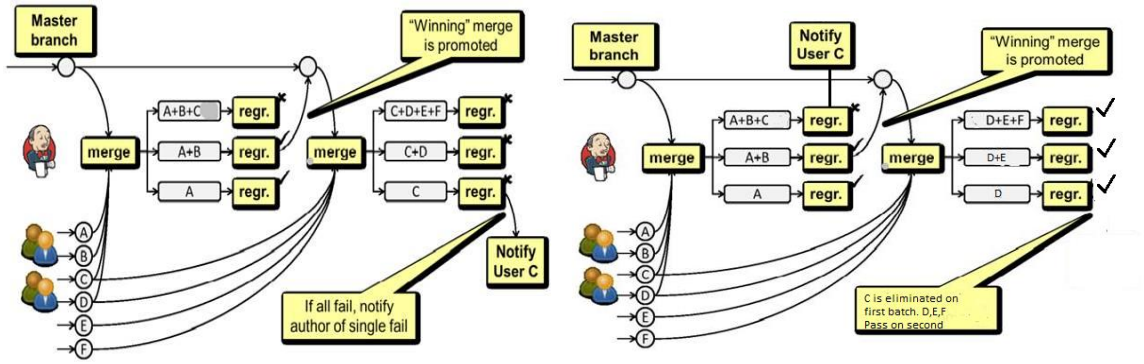

Figure 1. Changes to eliminate fails sooner in the flow. Original (left) vs. Changed (right)

Run Changsets In Isolation Before Combining Them

Our next step to resolve this was to run each changeset in isolation, and have it pass before allowing it to test in combination with others. This helped us achieve less variability in turnaround times. However, at a minimum any changeset needed to be tested twice serially taking at least 3.5 hours. This minimum also proved to be rarely seen in practice.

Run Changsets In Isolation In Parallel With Combined Testcases

The next iteration to improve turnaround times was to run every changeset in isolation while also combining the changeset with others to be tested in parallel. This started to make impacts in our turnaround time. We could root out failures quickly while also having the potential to pass in successes quickly. We divided our testcases into "trusted" and "untrusted" bundles. The unstrusted bundles were always supersets of the trusted bundles to ensure a changeset that had passed in isolation could not be delayed by a changeset that had not passed in isolation. These changes brought more consistency to our turnaround times. Fails would always be eliminated from consideration within one testing time.

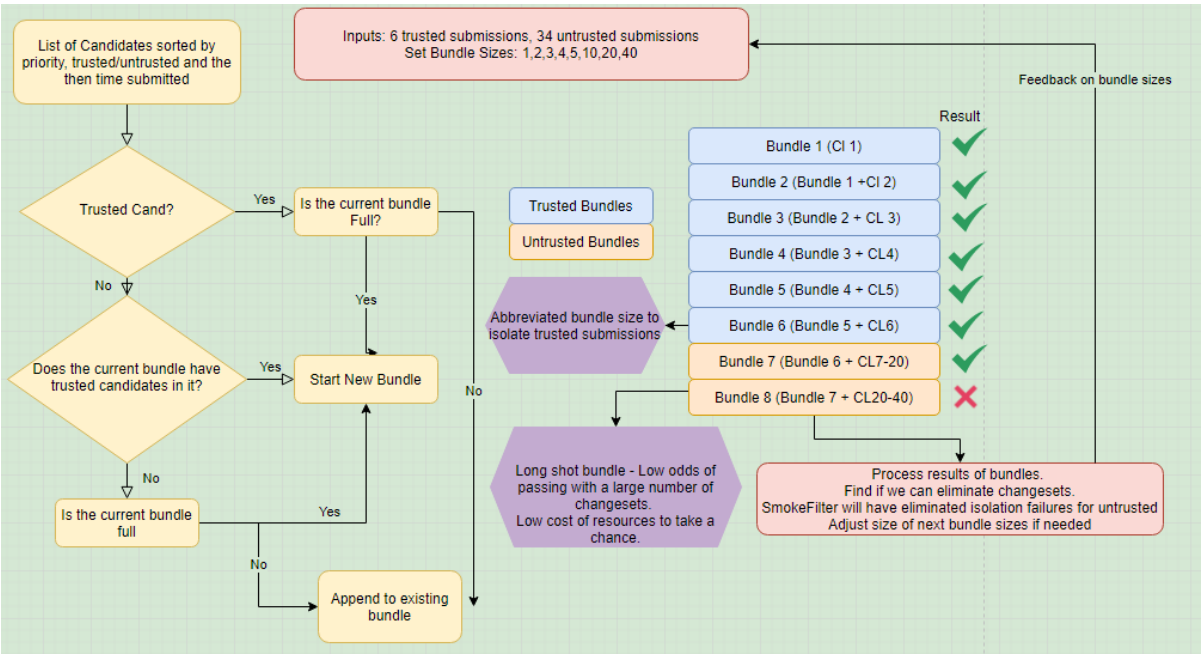

Figure 2. Running testcases in isolation & in combination in parallel bundle construction.

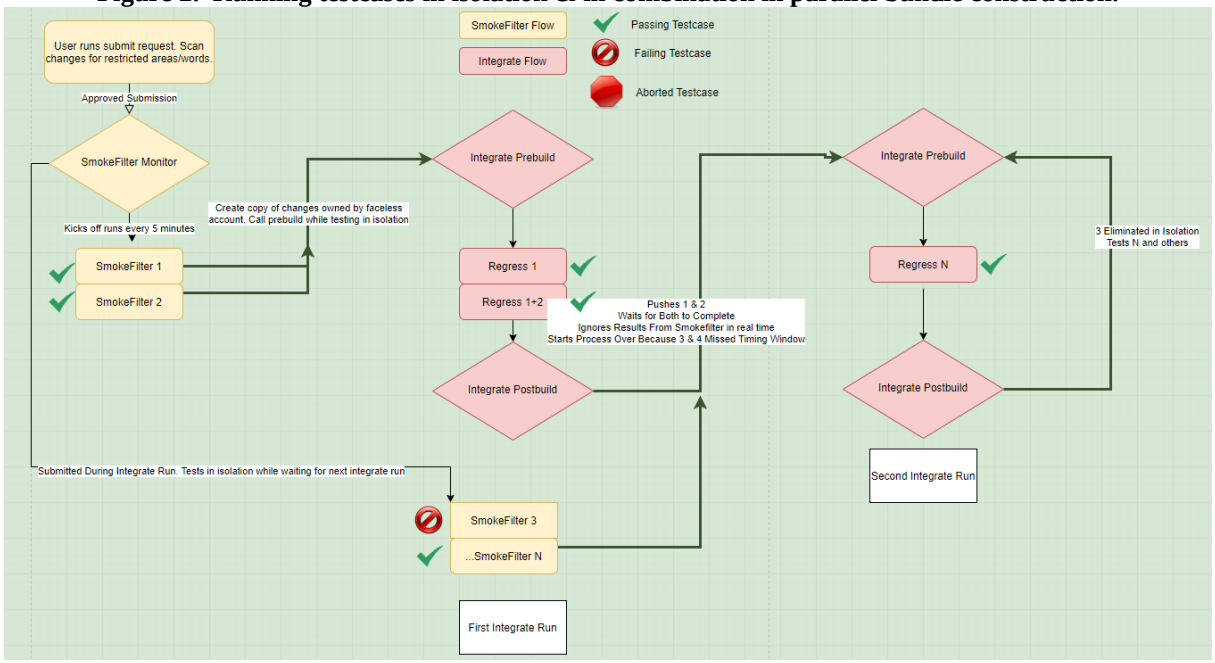

Figure 3. Running testcases in isolation & in combination in parallel workflow.

V. Event Driven

After making these changes, we still were seeing turnaround times above our expectations. The next improvement redesigend how the flow reacted when a changeset was submitted, failed, or passed. The flow originally was broken up into distinct start and stop points. It would need to come to a stop to re-evaluate what combination of changesets to test. Now it evaluates what to do every time an event occurs and any changeset changes state.

Previously, we would run "batches" of potential combinations of changesets while they ran in isolation. If A, B, and C were running in isolation, the batches would also fire off A+B, and an A+B+C. If D was submitted shortly after, it would run in isolation, but would wait for the batch of A+B and A+B+C to complete. Once complete, then it would kick off another HEAD + D, assuming A+B+C passed and was pushed to the repo.

This issue was causing large amounts of downtime in our flow. Changesets could see very different turnaround time outcomes depending on the timing windows they were submitted. Our flow was changed to react when any

testcase changed state. As soon as any changeset is submitted, fails, or passes, the state machine needs to decide to push changesets, fail changesets, or create new combinations of changesets to test.

To achieve this, we changed the structure of our flow to be a "scheduler", a "regress", and a "processor". Using Jenkins triggers and API calls, the flow calls the scheduler every time a changeset changes state, decides what testcases need to be run, and then launches off the number of regress jobs needed. Every time a regress job completes, it calls the processor to decide what to push, what to fail, and what testcases to abort. Once the processor changes the states of changesets it triggers the scheduler and the loop continues.

These changes made a dramatic difference in turnaround times. Average and median turnaround times stabilized near our testing time plus a small amount of overhead. The flow is able to process large amounts of submissions, and recover from external issues quickly.

This also allows the flow to run "prospective" combinations. In the example of A, B, & C, we could potentially run all of these: A, B, C, A+B, B+C, A+B+C, A+C

Where the number of user changesets proposed = n, then the number of testing changesets launched = (2^n) -1

This would guarantee the turnaround time of submissions always remained stable at the test-time plus a static amount of overhead.

Currently, we do not run this aggressively because our pass rate is high enough that it would not make a substantial difference. However for a pipeline with a high failure rate, this could be configured for low volumes of submissions. The scheduler also enables the possibility of scaling this up with more combinations to hedge the odds of a failure without going all the way to (2^n)-1. With high volumes of submissions, long testing times, and high testing resources this would become too costly very quickly.

VI. Implementation Of Event Driven Flow

To implement this event based flow, we have divided it into 2 distinct halves. Both are completely run by the same faceless account. We wanted the ability to manage environments, shelves, and workspaces through the entire flow. The first half, smokefilter is responsible for testing changesets individually. The second half, integrate is responsible for testing changesets in combination.

SmokeFilter

The first half of the flow, Smokefilter, is responsible for querying requested chageset submissions, landing the run on a valid workspace from our smokefilter pool, merging the changeset with the current head, creating a new changeset owned by the faceless account, and launching the test suite against it.

Once the new changeset owned by the faceless account is created, the flow triggers an event to the integrate flow. The integrate flow can then begin merging this changeset in combination with others to test while it is tested in isolation in parallel. If the test suite fails, smokefilter will mark the candidate as failed in our database. After this, it will find any integrate test case that contains the changeset and abort it knowing that we do not need to wait on the results. This does remove the possibility of two changesets being submitted that would pass in combination and fail independently. We accept this small caveat and expect users to combine edits locally in this situation. Finally for passes and fails, the flow will clean the workspace, reset it to the current head, and return it to the pool. For failing test cases, we compress relevant logs and send them to the users. This part of the flow is where the vast majority of changesets are failed in.

This part of the flow is almost completely separate from Jenkins and is written as perl scripts from scratch. Jenkins only executes a cron job to monitor for new request tokens. Once it finds these tokens, it launches free standing scripts into our batch system. These scripts handle all scenarios for individual changesets.

Integrate

The second half of our flow, integrate, is responsible for constructing combinations of changesets to test, landing the cominations on clean workareas from our integrate pool, kicking off test suites for the combinations, maintaining the correct state of every run, and deciding how to act once every combination finishes. To achieve this, we broke this part of the flow into three scripts. All three of these scripts are configured as Jenkins jobs that run custom scripts.

The first is our scheduler script. We run a curl command to kick it off every time a new changeset is introduced, failed, or passed. We also run this on a 5 minute interval to ensure we don't miss a trigger. The flow maintains a very simple database that contains every submissions CL# (previously sha for git implementations), a submission timestamp, a failed/passed/active switch, a priority, and a retry switch. When this scheduler script is run, it first queries this database to see what active changeset submissions exist. After this it uses simple file system data to determine what testcases exists. With this information, the flow decides if it needs to launch more testcases. Depending on the testcases needed, the scheduler combines changesets, lands them on workspaces from the pool, launches a regress job for each, and then exits.

This is a major departure from the setup of the original flow. Previously, this setup job would run, kick off regress jobs, and then sit idle waiting for the results. This is one of the major issues that caused delays. If a submission came shortly after a run had started, the submission had to wait the entire test case time to be combined with others and tested. Now this job combines submissions as they arrive, launches testcases immediately, exits and awaits the next state change in less than a minute. Normal execution is roughly 10 seconds, but can grow larger depending on how long the merges take.

Our second script is the regress job. The scheduler script can spin up infinite (limited by resources) regress jobs in parallel. It is very simple. It lands on the workspace designated by the scheduler that already has the correct changes merged in and runs our test case. Once the testscase is done, it triggers a processor job with the exit status along with other identifying environment variables.

The final script is the processor job. Its role is to receive the exit status of testcases and determine what actions to take. It can only run serially. We do not want the multiple processor jobs running in parallel trying to push changes, fail changes, and modify states. If multiple sets finish within a very short window it will run the decisions one after the other. Run times are similar to the scheduler with most taking 10 seconds and the high end taking around one minute.

For passing testcases, the processor determines which changesets need to be pushed. It also determines if any running testscases are subsets of passing testcases. If a testcase of A+B+C+D has passed and a testcase of A+B is still running due to variance in our batch system, the flow will abort the regress job.

For failing testcases the processor determines if it can determine one of the changesets is the cause, and if it can be marked as failed. This is rare because by this point all the individual tests of changesets should have finished and removed themselves. However, even if it cannot determine the cause, it will kill supersets of the failing changesets. If A+B has failed, the flow will abort the regress jobs for A+B+C, A+B+C+D etc. We do toggle this feature off because in the case of false fails it can cause the flow to become overzealous in killing off testcases.

In addition to handling passes and fails, the processor also modifies the filesystem structures that reflect what test cases are running. This ensures that the next scheduler run will correctly identify the state of running testcases.

Finally the processor will clean workspaces, tar relevant failing logs, send emails to users for passes & rejections, and recycle the workspace into the pool. When a regress job is aborted, it calls an abbreviated version of the processor that can run serially to only handle the filesystem and workspace changes. The aborted regress job assumes that all states changes have been handled from whatever script called the aborting curl.

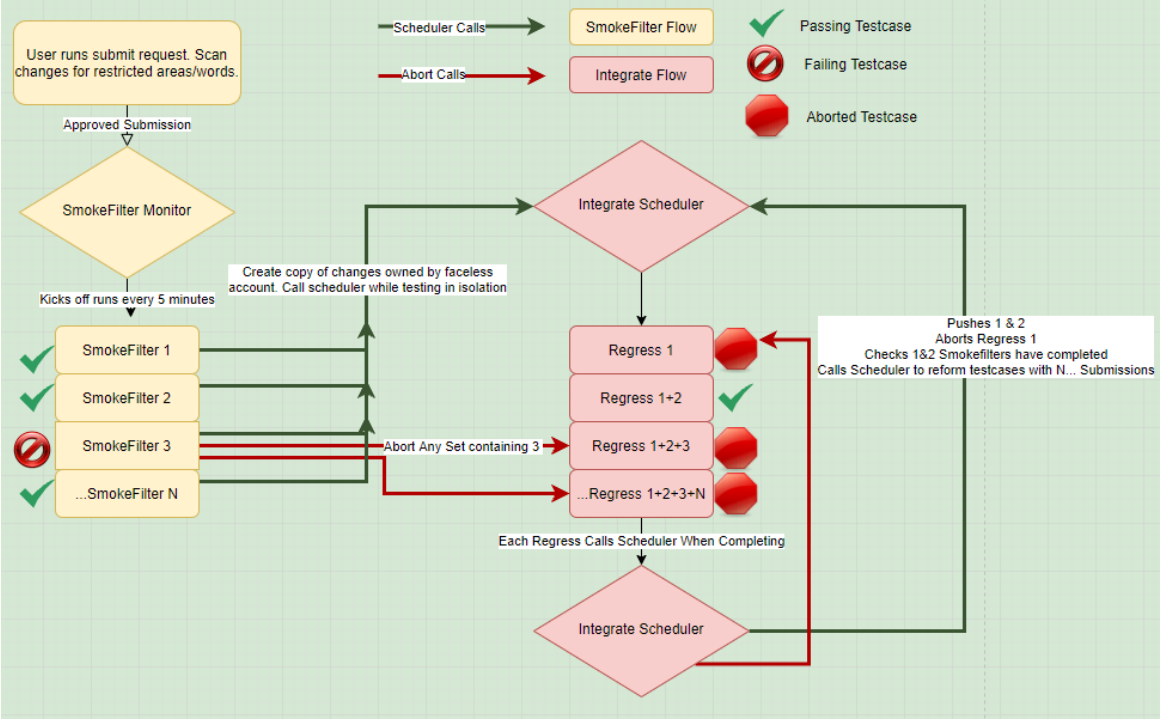

Figure 4. Smokefilter & Integrate Scheduler base flow without time delays in user submissions

Details

Every scheduler job that launches a regress job creates a directory with a prop file for each testcase that is run. Every time a regress job is aborted, or completes and is sent to the processor job it cleans up this prop file. If it is the last prop file remaining in the scheduler job's directory it cleans up the directory as well.

This creates a very simple filesystem based implementation for the scheduler job to load in active testcases. It does not need a database structure to track the state of the flow. It only needs to find all existing prop files and look at their contents to determine what is running.

Below is a quick example of the integrate and smokefilter halves of the flow working to quickly find passing and failing combinations of changesets. In this example 6 users submit every 15 minutes starting at 11:55 am. User 2s changeset will fail after 25 minutes, and user 4s changeset will fail after 20 minutes.

User	Run Window	Pass/Fail	Log File (Debug)
USER6	Start: Oct_10_13_10_00	Running	Run Log
USER5	Start: Oct_10_12_55_00 Finish: Oct_10_14_25_00	PASS	Pass Log
USER4	Start: Oct_10_12_45_00 Finish: Oct_10_13_15_00	FAIL	Fail Log
USER3	Start: Oct_10_12_25_00 Finish: Oct_10_13_55_00	PASS	Pass Log
USER2	Start: Oct_10_12_10_00 Finish: Oct_10_12_50_00	FAIL	Fail Log
USER1	Start: Oct_10_11_55_00 Finish: Oct_10_12_00_00	DuplicateKill	Duplicate Kill Log

Figure 5. SmokeFilter Example With Times

7007	2021-10-00 01:15:00	USER1_AT_samsung.com_00001 USER3_AT_samsung.com_00003 USER5_AT_samsung.com_00005 USER6_AT_samsung.com_00006 Status: Running (Gantt Chart)	NA
7006	2021-10-00 01:05:00	USER1_AT_samsung.com_00001 USER3_AT_samsung.com_00003 USER5_AT_samsung.com_00005 Status: Passed Run: 1:05 to 2:35 (Gantt Chart)	NA
7005	2021-10-00 01:00:00	USER1_AT_samsung.com_00001 USER3_AT_samsung.com_00003 USER4_AT_samsung.com_00004 USER5_AT_samsung.com_00005 Status: Failed (Subset Fail Aborted by Processor Job From Regress #2 @7004) Run: 1:00 to 1:05 (Gantt Chart)	NA
7004	2021-10-00 12:50:00	USER1_AT_samsung.com_00001 USER3_AT_samsung.com_00003 Status : Aborted (Superset Pass Long Run Aborted by Processor Job From Regress#1 @ 7006) Run: 12:50 to 2:35 (Gantt Chart)	USER1_AT_samsung.com_00001 USER3_AT_samsung.com_00003 USER3_AT_samsung.com_00004 Status: Failed (SmokeFilter Fail) Run: 12:50 to 1:15 (Gantt Chart)
7003	2021-10-00 12:45:00	USER1_AT_samsung.com_00001 USER2_AT_samsung.com_00002 USER3_AT_sa msung.com_00003 USER4_AT_samsung.com_00004 Status: Aborted (Subset Fail Aborted by Processor Job From Regress #1 @7001) Run: 12:45 to 12:50 (Gantt Chart)	NA
7002	2021-10-00 12:30:00	USER1_AT_samsung.com_00001 USER2_AT_samsung.com_00002 USER3_AT_samsung.com_00003 Status: Aborted (Subset Fail Aborted by Processor Job From Regress #1 @7001) Run: 12:30 to 12:50 (Gantt Chart)	NA
7001	2021-10-00 12:15:00	USER1_AT_samsung.com_00001 USER2_AT_samsung.com_00002 Status: Failed (SmokeFilter Fail) Run: 12:15 to 12:50 (Gantt Chart)	NA
7000	2021-10-00 12:00:00	USER1_AT_samsung.com_00001 Status: Passed Run: 12:00 to 1:30 (Gantt Chart)	NA

Figure 6. Integrate Example With Times

The far left column represents every time the scheduler job runs with timestamps of the start time as the next column. The next columns represent regress jobs launched by the scheduler.

In parallel to these combinations running in integrate, smokeFilter is running each of these changesets in isolation to identify failures as quick as possible and send signals to the integrate flow to remove them. In this example, we have 3 passing changesets that complete: The first in 90 minutes (test time); The second completing in 105 minutes with a delay due to being re-grouped into a new changeset and landing on a long running host; The third completing in 90 minutes. This results in a 95 minute average turnaround time for these three submissions, despite only a 60% pass rate on the completed submissions.

VII. Rollout And Results

Throughout this process, we made many incremental changes, and faced unreliable conditions from machines, environment, licenses, etc. Due to this noise, the effects of certain changes were not always immediately apparent. Other improvements like scanning lines of code for outside user directories had long lagging indicators. Blocking an improper release area could take months before the benefit of increased stability was actually realized. Once all the changes were in place, we went from seeing average and median turnaround times between 4-6 hours to consistently seeing average and median turnaround times near 2 hours per submission.

At the beginning of the project, we saw turnaround times at nearly exactly two times the testing time (3.75 hours). At this point we had already implemented the changes to run individual and combined changesets in parallel. These times were a result of very low submission volumes, making it rare for changesets to be waiting. The queue would have 1 or 2 changesets testing in parallel, making it easy for the flow to not create a backlog.

As the volume of submissions increased approaching the end of 2020, we saw large spikes in turnaround times as the increase in the volume of submissions created a strain on our infrastructure, more random fails, and backlogs in the flows logic.

We were able implement our fixes to reduce the costs of running changesets and dampen the effects of false fails by workweek 6 of 2021. After this, the average and median turnaround times somewhat stabilized around 4 hours. In workweek 2,4 we rolled out the event based logic and saw a drastic reduction in turnaround times. The times were roughly halved to stabilize at 2 hours. We do see occasional issues that causes averages to spike. These spikes are caused by large datapoints from paused and delayed submissions during infrastructure issues. However the median has remained stable, reflecting that excluding the delays that are outside of the flows control it continues to run smoothly. The flow continues to produce reliable turnaround times even with large volumes of submissions, and recovers well from external issues. We did see a decline in submission volume as this project tapered, but we have continued to see stability in our turnaround times as our next projects ramp up.

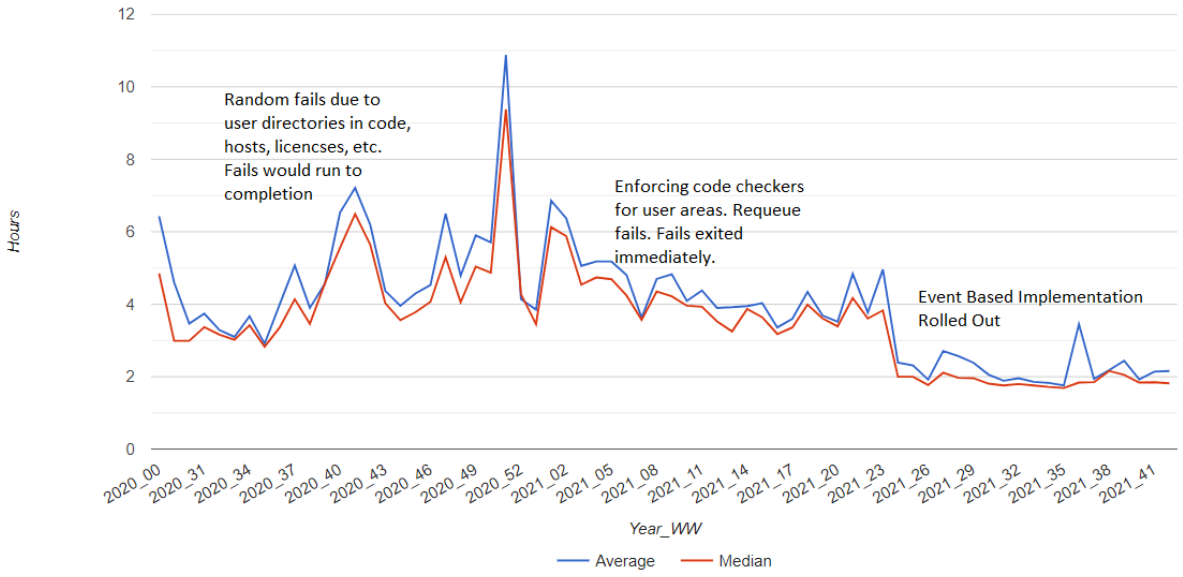

Figure 7. Weekly Average and Median Turnaround Times In Main Pipeline

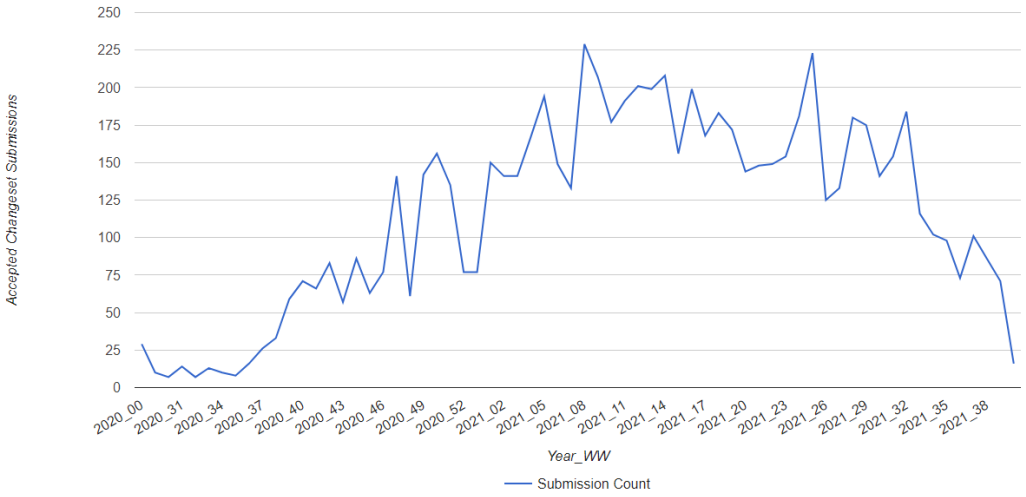

Figure 8. Weekly Total Accepted Submissions In Main Pipeline

VIII. Future Improvements

The CI flow has been very successful helping our engineers develop in a stable environment and meet deadlines/milestones. Due to this success, it will continue to be used on multiple further projects. Even marginal improvements in time and resources used by this flow will provide valuable returns due to the scale. The following are some of the areas we are exploring to continue improving this flow.

If the flow was redesigned from scratch, we would remove the distinction between smokeFilter and integrate. The distinction of having one or five changesets contained in a testcase should be irrelevant to the scheduler. The only unique purpose smokeFilter has is creating an immutable copy of the changeset that is owned by the faceless account, but this can be rolled into one flow.

In addition to removing this distinction between smokefilter and integrate, we would also consolidate the flow completely on Jenkins or on our batch system.

The last high level flow design we would change is to allow the configuration of more or less change sets to be run in parallel dynamically. Based on the amount of resources available, the number of submissions in the queue, and the projected pass rates of submissions based on files modified the flow should be able to become more or less aggressive in scheduling testcases.

Outside of these high level flow considerations, the turnaround time for our flow has nearly reached the limit of our test time. To break through this barrier we will need to find ways for some submissions to run faster than the test time. To achieve this we are exploring running only a subset of tests based on the files modified. However this is very difficult to reliably implement due to a complex dependency tree of files having the ability to affect tests.

IX. Conclusion

Implementing a scheduler based CI flow that runs changesets individually and in combination in parallel to find passing & failing combinations of changesets can significantly reduce the turnaround time of user submissions.

As a team can push the cost of running any given changeset lower, they are also capable of running more unique combinations of changesets in parallel.

Given the ability and resources to run enough unique combinations of changesets in parallel, the turnaround time for any number of submissions with any pass rate should stay at the amount of time it takes to run a test case.

Enabling this flow in Jenkins can be done with three relatively simple jobs triggering each other on events. First a scheduler to determine the existing state of submitted, running, and completed changesets, and determine what tests cases to run. Second a regress job that can be scaled for any number of testcases to launch the test suite and the exit status. Finally a processor job that runs serially looking at the results of every regress job and determines what to push, what to fail, and how to modify the states of changesets and testcases.

This scheduling capability can be very useful going forward. It can be re-created connecting to any batch or cloud compute service and does not necessarily need to be tied to Jenkins. What this means is that the turnaround time of a continuous integration scheduler based flow will only ever be resource limited.

References

[1] John Dickol, “Advanced Usage Models for Continuous Integration in Verification Environments”, DVCON 2015