Advanced Usage Models for Continuous Integration in Verification Environments

John Dickol
Samsung Austin R&D Center
Austin, TX
j.dickol@samsung.com

Abstract -Continuous Integration (CI) tools such as Jenkins have become popular in design projects for automating the monitoring and maintenance of design health - through running regressions, publishing results, etc.

This paper will describe the evolution of CI usage models for a large, multi-site IP development project. Implementation details as well as pros and cons for each usage model will be covered

INTRODUCTION

Continuous Integration (CI) is software development process which encourages developers to check in and integrate their changes frequently. Automated testing of the checked-in changes provides continuous feedback about the health of the project to the entire development team. Although originating in the software development world, CI is increasingly being applied to hardware development as well.

A wide variety of proprietary and open-source CI tools are available to automate the CI process. These tools communicate with the project's revision control system (CVS, Perforce, etc.) and can detect new checked-in changes, automatically run regression, publish results etc. Previous DVCon papers[1][2] have described applying Jenkins[3], a popular open source CI tool, to verification projects.

This paper will describe the evolution of CI usage models using Jenkins for a large, multi-site IP development project. Implementation details as well as pros and cons for each usage model will be covered.

PROJECT OVERVIEW

The concepts described in this paper were developed for a CPU IP project which was developed at multiple worldwide Samsung sites. Revision control for the RTL and Design Verification (DV) code was managed with git[4], an open source distributed version control system. The verification environment uses a mixture of unit-level, subsystem-level and full-IP-level testbenches. Users are expected to run a "smoke regression" before checking in any changes. This smoke regression compiles all testbenches and runs a minimal set of sanity check tests on each testbench. Early in the project, this regression completed in 5-10 minutes. As the design matured, more RTL and DV code was added so compile and simulation times increased. By the end of the project, smoke regression time was about 1 hour. In addition to the smoke regression for check-ins, we ran more extensive daily regressions for bug finding, coverage collection etc.

FIRST STEPS

Early in the project, the author saw a DVCon presentation about continuous integration [1]. Our project did not yet have any regression automation in place, so Jenkins seemed worth a try.

The flow described in [1] implements a "**Clean HEAD**" usage model. This model provides continuous feedback about the health of the design. Jenkins monitors the source code repository and runs a regression if it detects updates to the HEAD or "master" branch of the design. If the regression fails, Jenkins sends email to all users who made

check-ins since the last passing regression. Regression results are also available through the Jenkins web interface which summarizes information about regressions: start time, duration, design changes (modified files and check-in comments), pass/fail status, etc.

It was surprisingly easy to get started, following the recommendations in [1]. We already had a regression script which provided a single command to run the smoke regression on our batch farm. This script would launch jobs to compile all testbenches, run simulations, wait for all jobs to complete and summarize the results. We only had to tell Jenkins where our source code repository was and to run this regression script.

We also used the Jenkins Email-ext plugin[5] to customize the email indicating the status (pass or fail) of each regression to each person who checked-in changes since the last passing regression. If a regression failed, the names of the failing tests and corresponding error messages were also included in the mail.

Refinements

Initially, Jenkins was configured to poll the git repository periodically – every 5-10 minutes. We eventually refined this to have git check-ins automatically trigger Jenkins. Jenkins has a powerful web API which allows many types actions to be initiated by reading certain predefined URLs on the Jenkins web server. One of these available actions is triggering the polling of the revision control system. Using a command line tool such "curl" or "wget", it's possible to have a script trigger Jenkins after each check-in.

Git defines several "hooks" which are optional user-definable scripts which are called after certain pre-defined git actions. The "post-receive" hook is run whenever the git server receives a set of changes (a "commit" in git-speak) pushed to the server by a user. So, to have a git check-in trigger Jenkins, we just need to create a one-line post-receive script which reads the magic URL which will trigger polling. With this in place, if Jenkins is idle, it will immediately start the regression of a new release without waiting for the polling interval to expire. If a regression is already running, it will wait until the regression is done before starting the next regression.

```
curl http://yourserver/jenkins/git/notifyCommit?url=<URL of the Git repository>
```

Figure 1. git post-receive hook to trigger Jenkins

GROWING PAINS

The initial reaction from the team was positive. In most cases, users would react to the Jenkins-generated emails and fix problems with their check-ins without needing to be nagged about it. Over time, though, both the team and design grew in size and it became harder to keep this model healthy. A broken model made it difficult for users to update unrelated portions of the design, because their smoke regression would fail due to the other errors.

We eventually decided that we wanted a more robust flow to meet these requirements:

- Maintain a "known-good" version of the design. If a user checks out the master branch of the design, it should always compile and simulate successfully.
- Don't let a bad check-in from one user break the model for everyone else.
- One bad check-in shouldn't prevent other users from checking in their changes.

Fortunately, the Jenkins Git Plugin[6] had the answer. The plugin's "pre-build branch merging" feature can be used to implement a "Gated Check-in" flow. In this flow, users check their changes onto a unique "release" branch instead of the master branch. Git uses a lightweight branching model which makes this much less painful than it would be with other revision control systems. When Jenkins detects an update on one of these release branches, it merges that branch onto a local copy of the master. If the merge is successful and if the integration regression passes, the merge result is pushed to the source repository and becomes the new master. If either the merge or the regression fails, the offending user is notified and Jenkins moves on to the next user's release. As a result, the master branch is always clean. A bad check-in from one user will not prevent subsequent users from checking in their changes.

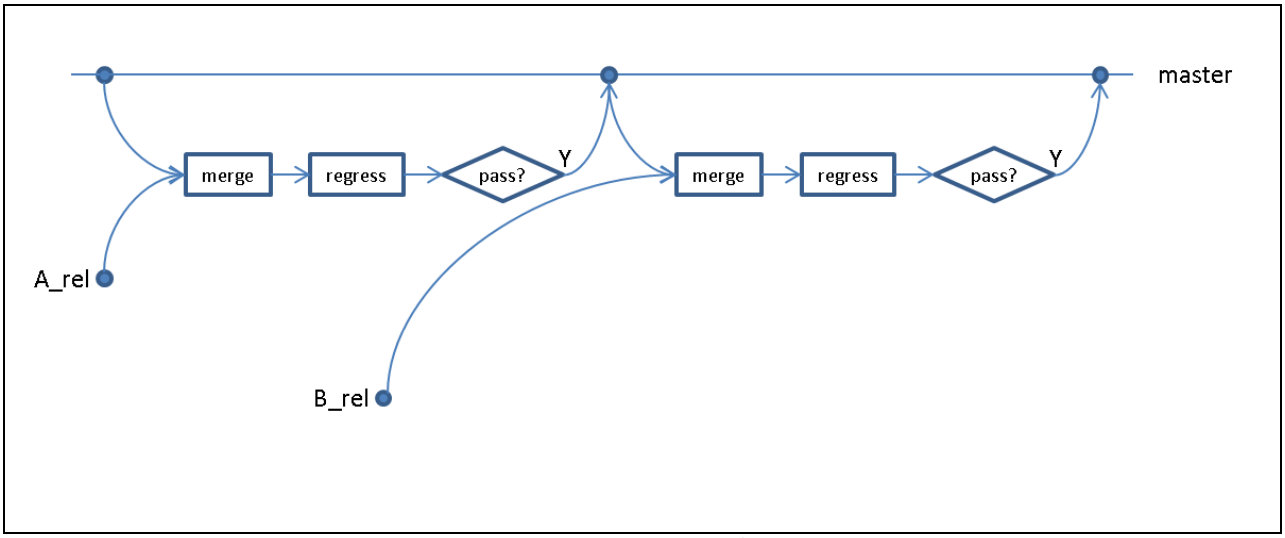


Figure 2. Gated check-in flow

Gated Check-in Flow Implementation

This flow is easy to set up using standard Jenkins plugins and requires only a few tweaks to the existing Jenkins job for the Clean HEAD model.

1. Change the "Branches to build" configuration from "origin/master" to "origin/**_rel". This will build any branch whose name ends in "_rel".
2. Enable the "merge before build" option and specify the "Branch to merge to" as "master". In our version of the git plugin, this option is hidden in the Advanced tab.
3. In Post-Build Actions, Git Publisher, enable both "push only if build succeeds" and "Merge result"
4. Note: We're using an older version of the git plugin (1.4.0). Later versions (2.x) changed the user interface, so these features may be hidden in a new location.

Setting up Jenkins was the easy part. Actually doing the switchover and re-training the users took a little more effort. We needed to prevent ordinary users from pushing changes to the master branch. This was easily done on our git server by configuring the permissions to allow only the dedicated userid running the Jenkins server from pushing changes to the master branch. If a user did try to push to master, they would get an error message. Next we

needed to get users to push to their release branch. To simplify this, we created a git alias - a new git command – "git release" which pushes the changes to a branch name constructed from the user's username.

```
# create the alias (added to project environment setup script)
git config alias.release '!git push origin HEAD:refs/heads/${USER}_rel'

# Use this command to push committed changes to personal release branch:
git release
```

Figure 3. git alias to create new "git release" command

The switchover was surprisingly uneventful. We pre-warned the team several days before the change. At switchover time, we stopped the old Jenkins job and let the current regression finish. Next, we changed the git server permissions and reconfigured Jenkins to implement the new flow. Within an hour we able to re-enable Jenkins and begin running with the new flow. The new flow mostly worked OK out of the box, but there were a few of minor rough spots:

1. There was no good built-in way to rerun false fails (e.g. if a regression failed due a license or file system error.) Jenkins keeps track of what branches/commits it has built, so if a build failed, it wasn't easy to convince Jenkins to retry it. The typical solution for this was to re-release the changes with a trivial change (e.g. comment or whitespace change). Jenkins would see this as a new, untried release and would attempt to rebuild it.
2. If a user's release could not be regressed due to a merge conflict, Jenkins would email the fail info to the Jenkins administrators, but not to the author of the failed release. We eventually worked around this with some custom scripting to update the mail list for failed merges. Ideally, the git plugin would handle this.

MORE GROWING PAINS

The Gated Check-in flow worked well for several months. Eventually, the smoke regression runtime and number of check-ins per day both increased to the point where changes checked in mid-afternoon were not integrated until late at night – well after the nightly regressions had started. Figure 4 shows the total integration time for each check-in in a typical week. This integration time is the length of time from the user checking in changes until Jenkins has finished the regression for those changes. Check-ins made during off-peak times (mornings & weekends) were typically integrated within 20 minutes. Check-ins made during peak times (afternoons) could wait up to 8 hours to get integrated.

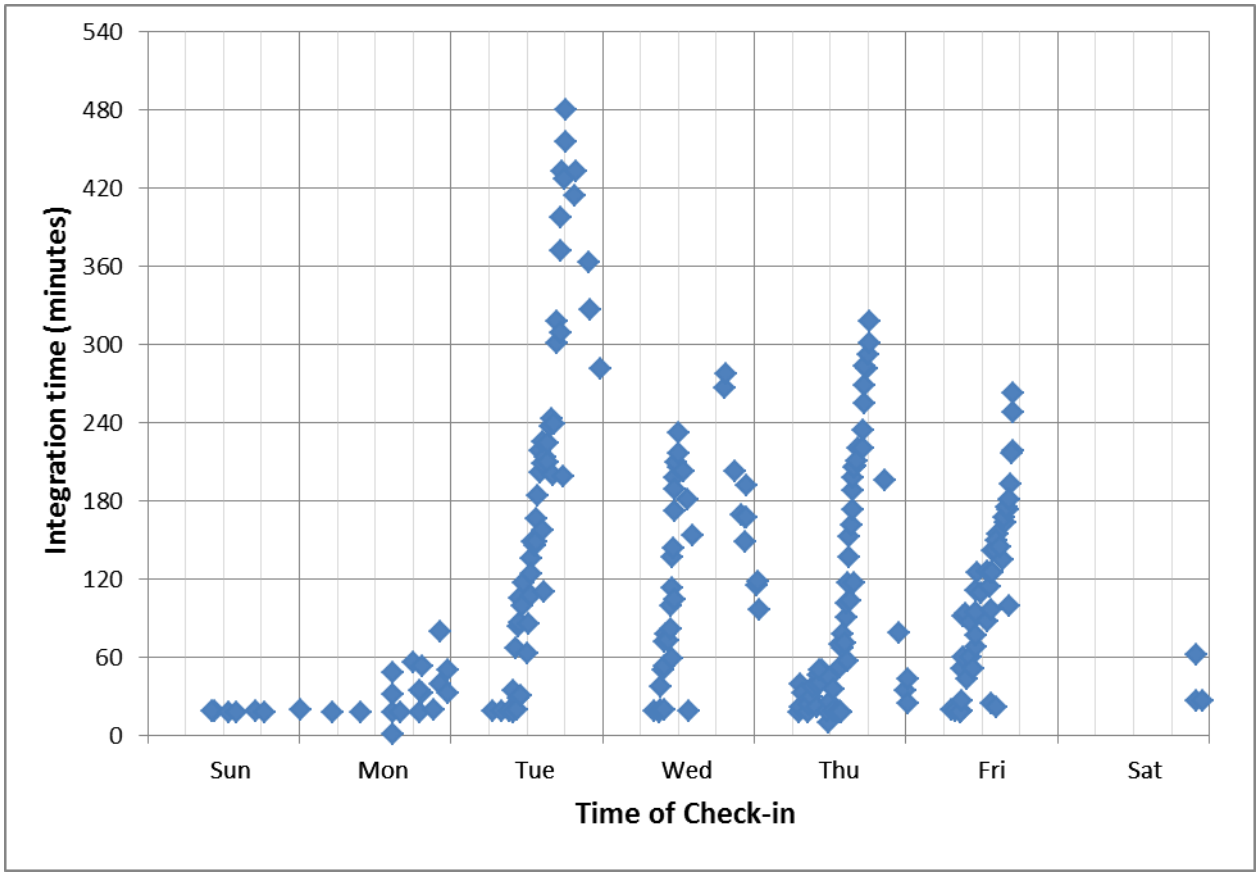


Figure 4. Gated check-in flow integration time (queue time + regression time) vs. time of check-in

What we needed was a method of regressing several users' releases at once. But if that multi-release regression failed, we also needed a way to quickly determine which of those releases were causing the fail. Git has a handy "bisect" command which, given two commits (e.g. the last known-good master and the failing multi-release merge) can do a binary search to find the culprit by running a user-specified command (e.g. the smoke regression) on each of the intermediate points during the search. The problem with this is the search is a serial process. Finding the failing release could take several iterations, causing even more backlog in the integration pipeline.

What we needed was something like a parallel git bisect. Given enough resources (i.e. machines and licenses), we could simply regress all possible combinations of pending releases in parallel. At that time, we didn't have that many resources to spare, so we implemented a Parallel Gated Check-in flow that would run 2 parallel regressions.

Parallel Gated Check-in

The "**Parallel Gated Check-in**" flow launches multiple regressions in parallel. Each regression contains one or more merged pending user releases. The "winning" regression, i.e. the one which successfully integrated the largest number of pending releases is promoted to master.

For example, given a queue of releases to be integrated:

A B C D E F ...

Jenkins will launch 2 parallel regression jobs and wait for both to complete (similar to Verilog fork/join):

```
master+A
master+A+B+C+D
```

If candidate 2 (master+A+B+C+D) passes, it's promoted. If only candidate 1 passes, that one is promoted. If neither passes, release A is marked as failure and the author notified via email (same process as before). Jenkins then repeats the process with candidates B-E:

```
master+B
master+B+C+D+E
```

The number of releases per candidate can be adjusted. We initially set it to 4 (i.e. at most 4 releases would be merged and regressed.) If necessary, the number of candidates can be increased (e.g. run 3 parallel regressions: m+A, m+A+B+C, m+A+B+C+D+E+F+G). This would give faster determination of a failing release, but at the cost of additional resources (server slots & licenses). Given infinite resources, we could try all combinations of pending releases in parallel (A, A+B, A+B+C, A+B+C+D …). This would maximize throughput, but could potentially tie up too many resources needed for other tasks.

With this flow, a failing commit needs to propagate to the "A" slot before we can unambiguously determine that it is the cause of the multi-release regression failure. (i.e. the A+B+…) slot. This causes some debate over the optimal number of releases to merge in the 2nd slot. Too few, and the releases are not processed fast enough. Too many, and it can take more iterations for a "bad" release to be identified. The "correct" answer depends on how frequently bad code gets released, how big is the backlog, etc.

Even if there is a fail in the 2nd slot, we can still make progress in the 1stslot. So, the throughput of this flow is no worse than the single Gated Check-in flow.

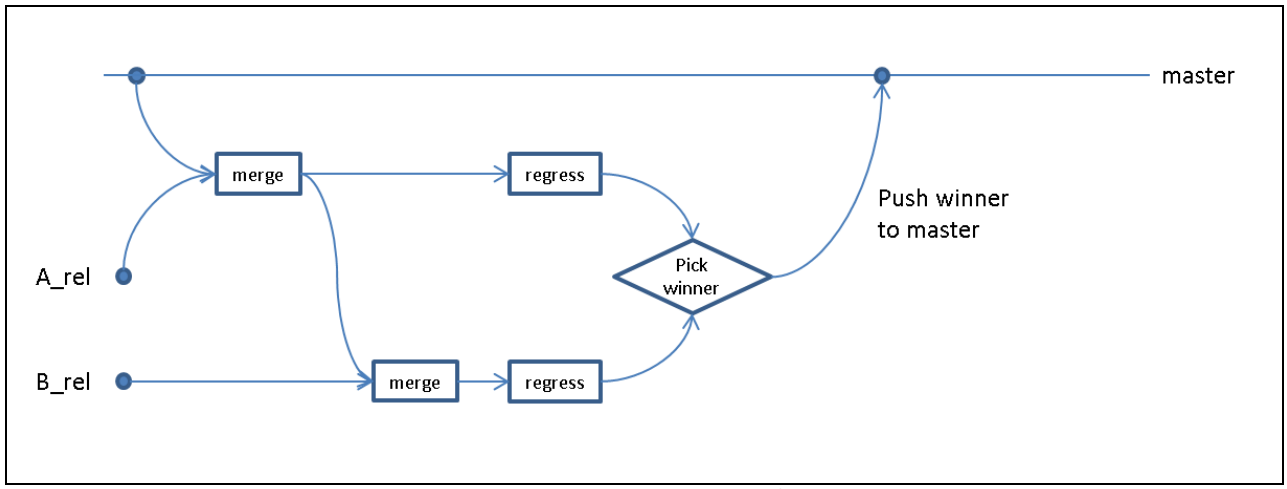


Figure 5. Parallel gated check-in flow

Parallel Gated Flow Implementation

To implement this flow, we needed to go beyond what could be done with existing Jenkins plugins. We considered modifying the git plugin or writing our own. We rejected this idea because we didn't have the necessary expertise on the team. We opted instead to extend the existing flow with some custom scripting to implement a parallel regression flow.

We split the existing integration job into two jobs: one to select the releases (i.e. git commits) to build, the other to run the regressions. The launcher job uses Jenkins' existing algorithm to select a commit to build. Jenkins picks the oldest commit which is not already merged onto the master branch and which has not already been tried by Jenkins. A custom script then queries git to get an ordered list of new, unmerged commits newer than the first one selected by Jenkins. The script merges the first N new commits onto the initial commit. The result is two new commits ("merge candidates"): one containing the Jenkins-selected commit, the other containing the Jenkins commit plus N additional commits. (N is configurable).

Next, we launch a regression job for each of the merge candidates. The Parameterized Trigger plugin [7] makes this easy. It enables a build step in one Jenkins job to trigger builds of another Jenkins job. The plugin reads one or more "property files" which specify job parameters for the new builds. In our flow, a custom script creates a property file for each of the merge candidates. This file specifies the commit to build, run directory, etc. for each of the candidates. We configure the Parameterized Trigger plugin to read all property files matching a file pattern "merge_candidate*.prop". The plugin launches one job for each property file and waits for them to complete (similar to Verilog fork-join). Figure 6 shows sample property files for two merge candidates.

merge_candidate_1.prop:

```
COMMIT_TO_BUILD=merge_candidate_1
LAUNCHER_GIT_REPO=/jenkins/workspace/proj_integrate/.git
CANDIDATE_JOBNUM_FILE=/jenkins/workspace/proj_integrate/merge_candidate_1.job_num.txt
CANDIDATE_NUM=1
WS_SUFFIX=
CANDIDATE_BRANCHES=origin/A_rel
```

merge_candidate_2.prop:

```
COMMIT_TO_BUILD=merge_candidate_2
LAUNCHER_GIT_REPO=/jenkins/workspace/proj_integrate/.git
CANDIDATE_JOBNUM_FILE=/jenkins/workspace/proj_integrate/merge_candidate_2.job_num.txt
CANDIDATE_NUM=2
WS_SUFFIX=@2
CANDIDATE_BRANCHES=origin/A_rel origin/B_rel
```

Figure 6. Sample Jenkins job launch prop files

After the parallel regressions finish, the launcher job runs another script to determine the winner. The Parameterized Trigger plugin sets environment variables with the final status (SUCCESS/FAILURE) of each launched job. Because Jenkins doesn't necessarily run the regression jobs in the same order as we launch them, we need a way to map each merge candidate to the corresponding Jenkins regression build number. We do this by passing a file name parameter to each regression job and have each regression job write its job number to that file. The "winner" is the merge candidate with the largest number of releases which passed its regression. If there is a winner, that merge candidate is pushed to the server as the new master.

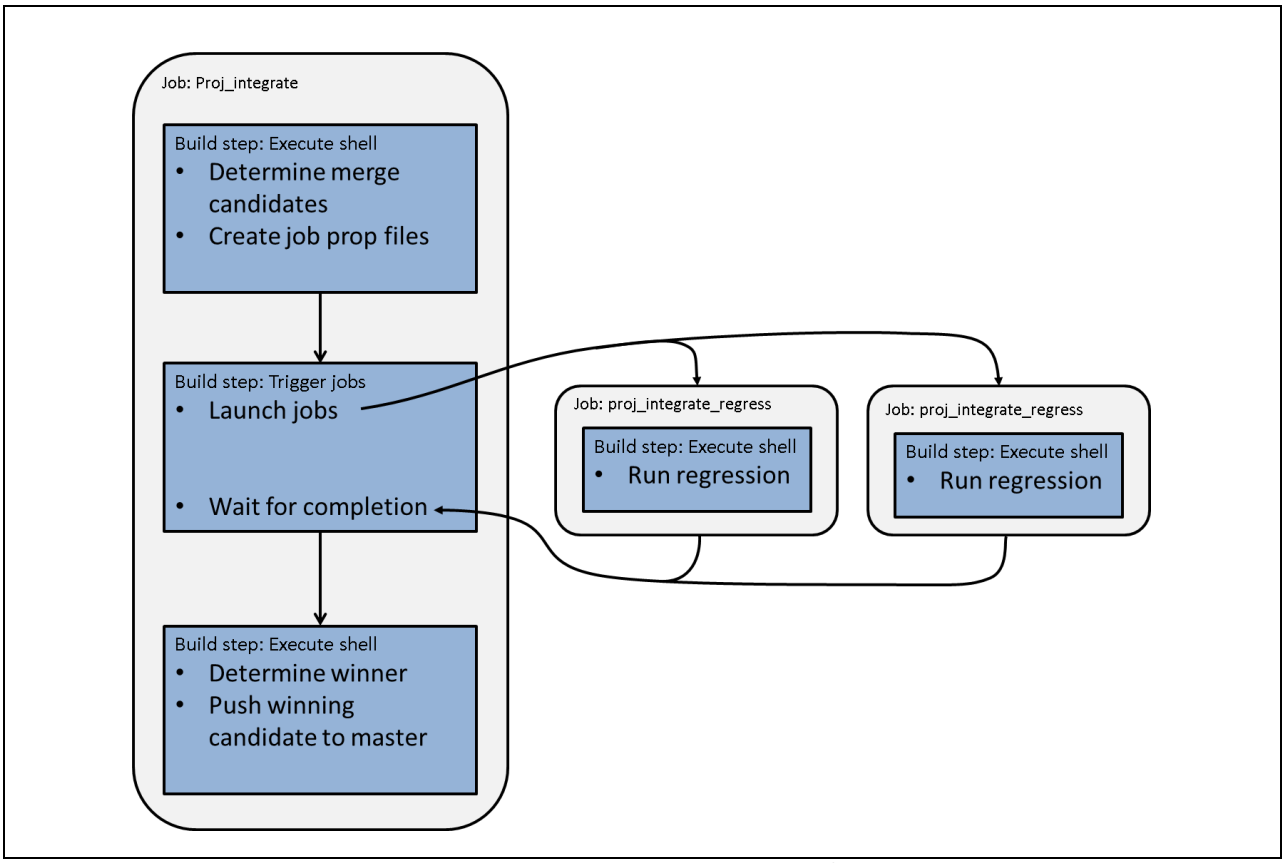


Figure 7. Parallel gated check-in Jenkins job flow

Results

After implementing the 2-way parallel flow, we saw a significant reduction in integration time as shown in Figure 8. Aside from a few outliers caused by batch or license system failures, integration time was typically around 1 hour with occasional peaks of 3 hours. Note: the data in Figure 8 is slightly misleading. Due to limitations of the data collection methods used, each point on the chart represents 1 Jenkins regression which, in the parallel gated flow, may contain multiple user check-ins. So, the number of check-ins is actually greater than (2-3x) the number of points. Even with that limitation, the chart is still useful for indicating the regression wait time.

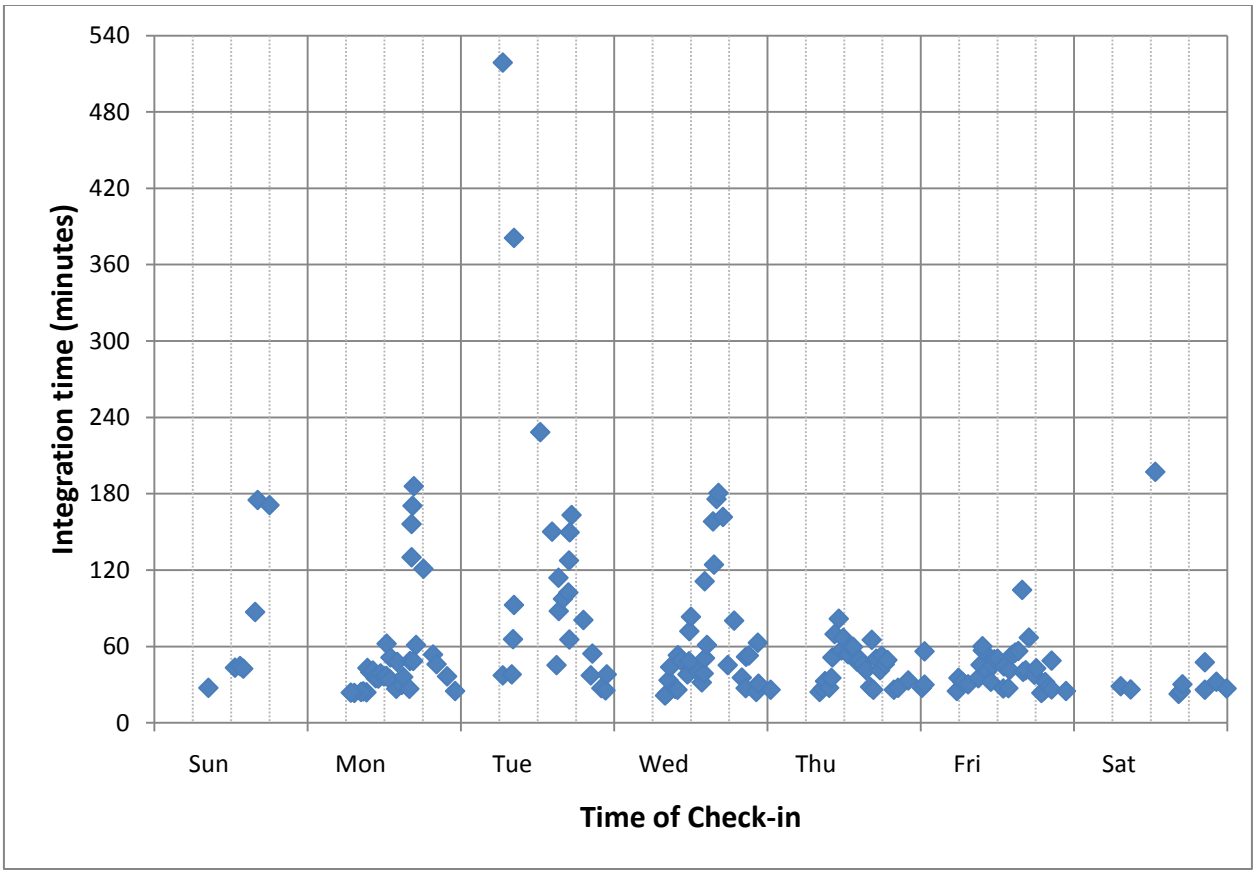


Figure 8. Parallel gated check-in flow integration time

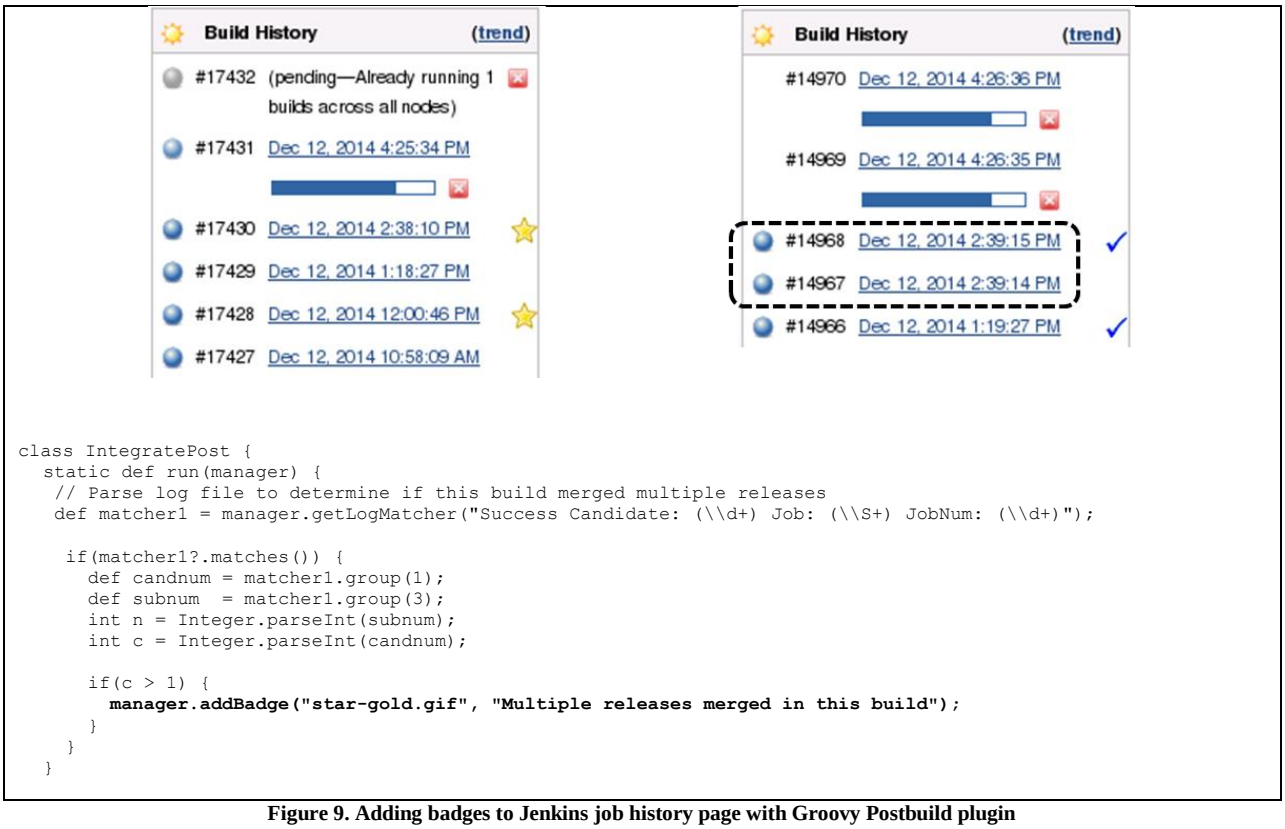
This flow worked well for about 8 months after which the increased runtime of the smoke regression started causing increased integration backlogs. The fix was simple: increase the number of parallel regressions from 2 to 3 (or even to 4 in some cases.) The custom scripts needed minor tweaks to remove some original assumptions of only two parallel branches. We used a modified power-of-2 series for the number of releases per regression (1,2,5 or 1,2,4,9). This gave us good throughput with fairly quick isolation of failing releases.

Bells and Whistles

Another plugin (Goovy Postbuild[8]) provides a way to use Groovy[9] (a java-like scripting language) to modify the Jenkins job webpages. It's possible to add additional text or icons (the plugin calls these "badges") to these pages. We used this plugin in several places to provide additional information about successful multiple regressions:

- Add a gold star to indicate successful multiple regressions in the integration job build history list.
- Add a check mark to indicate the merge candidate "winner" in the integration regression job build history list.
- List all the user releases for a successful multi-release regression on the integration job summary page.

Figure 9 shows a portion of the build history page with these badges and a fragment of Groovy code used to add the gold star. Similar code is used to add the checkmark.




```
class IntegratePost {
  static def run(manager) {
   // Parse log file to determine if this build merged multiple releases
   def matcher1 = manager.getLogMatcher("Success Candidate: (\\d+) Job: (\\S+) JobNum: (\\d+)");

    if(matcher1?.matches()) {
      def candnum = matcher1.group(1);
      def subnum  = matcher1.group(3);
      int n = Integer.parseInt(subnum);
      int c = Integer.parseInt(candnum);

      if(c > 1) {
        manager.addBadge("star-gold.gif", "Multiple releases merged in this build");
      }
    }
  }
```

Figure 9. Adding badges to Jenkins job history page with Groovy Postbuild plugin

FUTURE WORK

The Parallel Gated Check-in flow works well, but the current implementation is a hodgepodge of scripts, Jenkins jobs and plugins, etc. Ideally this could be encapsulated in a single Jenkins plugin – possibly an extension of the existing git plugin.

We've only used these flows with git. Additional work would be required to adapt them to other revision control systems.

CONCLUSIONS

This paper presents three Jenkins usage models shows how they were applied to an actual project resulting in improved model stability and reduced integration times for user check-ins.

The "Clean Head" model is easy to set up and is a good fit for smaller, well-disciplined teams who are willing to tolerate some churn in the health of the design.

The "Gated Check-in" model is also easy to set up and provides a more stable, known-good model but at the expense of longer times to integrate users' changes.

The "Parallel Gated Check-in" model is more complex to set up but provides the model stability of the Gated Check-in flow with improved integration throughput.

REFERENCES

[1] JL Gray, Gordon McGregor, "A 30 Minute Project Makeover Using Continuous Integration," DVCON 2012

[2] Jason Sprott, Andre Winkelmann, Gordon McGregor, "A Guide to Using Continuous Integration within the Verification Environment", DVCON 2014

[3] Jenkins website http://jenkins-ci.org/

[4] Git website http://git-scm.com/

[5] Jenkins Email-ext plugin https://wiki.jenkins-ci.org/display/JENKINS/Email-ext+plugin
[6] Jenkins Git plugin https://wiki.jenkins-ci.org/display/JENKINS/Git+Plugin
[7] Jenkins Parameterized Trigger Plugin. https://wiki.jenkins-ci.org/display/JENKINS/Parameterized+Trigger+Plugin
[8] Jenkins Groovy Postbuild Plugin. https://wiki.jenkins-ci.org/display/JENKINS/Groovy+Postbuild+Plugin
[9] Groovy language website http://groovy.codehaus.org/