Trunk Based Development

The high-throughput source control foundation of the most lauded software development organizations

Paul Hammant & Steve Smith

Trunk Based Development

- [Site Source on Github](#)

Book transformation of TrunkBasedDevelopment.com
Copyright © 2017: Paul Hammant and Steve Smith

Context

Related Concepts

Focus Area, with overlap noted
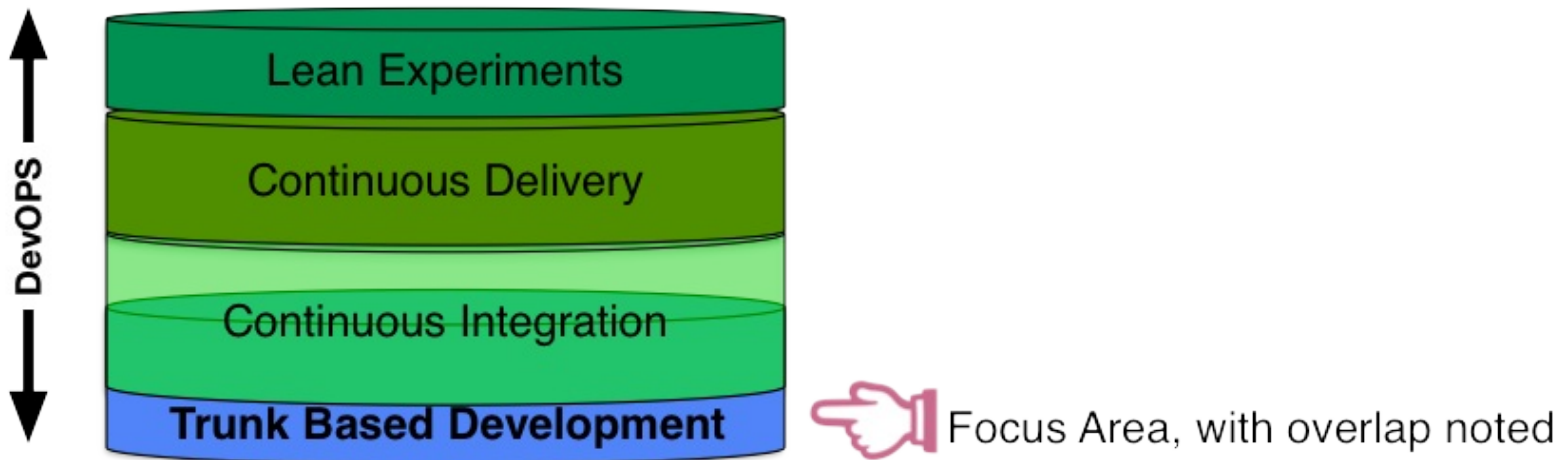
Continuous Integration (CI) has been in practice since the mid-nineties in its modern incarnation (integrating to a shared code line frequently, and testing that).

Continuous Delivery (CD) is a layer on top of that, has been practiced since the mid-2000's, and documented in Jez Humble and Dave Farley's book of the same name in 2010. This site gives a 5% summary of the practice. The reader should dive into the Book and associated site, without delay.

Lean Experiments can happen in any development team on any project but

work **best** on foundations that are solid. Specifically, the solid foundations of Trunk Based Development, CI and CD. This site does not touch on Lean Experiments at all, but the reader should strive to understand that field of science when the lower layers of the stylized cake are solid.

Importantly, the reader should understand that there is a large overlap between Trunk Based Development and Continuous Integration, as defined by its definers and documenters. Whereas Trunk Based Development focuses on a pure source-control workflow and an individual contributor's obligations to that, Continuous Integration focuses equally on that and the need to have machines issue early warnings breakages and incompatibilities.

DevOps is encompassing too. At the very least, the expansion of developers habits into operations heartlands giving Infrastructure as Code, continual improvement experiments, a focus on time through the machine, the scripting of previously manual operations things, goes further than we can represent in a stylized layer cake diagram, and indeed on this site.

Distance

Branches create distance between developers and we do not want that

— Frank Compagner, Guerrilla Games

Assuming any network-accessible source control, physical distance is mitigated by AV technologies including screen sharing. So we will not worry about that so much these days.

Frank's 'distance' is about the distance to the integration of code from multiple components/modules/sub-teams for a binary that could be deployed or shipped. The problematic distance is to code not yet in the single shared branch, that might:

- break something unexpected once merged
- be difficult to merge in.
- not show that work was duplicated until it is merged
- not show problems of incompatibility/undesirability that does not break the build

Trunk Based Development is a branching model that reduces the distance to the max.

What it is

Notes

- Use of "Developers" throughout this site, means "QA-automators" for the same buildable thing, too.
- When we say 'the trunk' on this site, it is just a branch in a single repository that developers in a team are focusing on for development. It may be called 'master'. That hints at the fact that the branch in question may literally not be called 'trunk' at all.

There are many deciding factors, before a development team settles on Trunk Based Development, but here is a short overview of the practices if they do:

Releasability of work in progress

Trunk Based Development will always be **release ready**

If an executive manager visited the development team and commanded "Competitor X has launched feature Y, go live now with what we have", the worst response would be "give us one hour". The development team might have been very busy with tricky or even time-consuming tasks (therefore partially complete), but in an hour, they are able to go live with something just stabilized from the trunk. Perhaps they can do it in less than, an hour. The rule though, is never break the build, and always be ready for that CIO-commanded disruption to plans.

Where releases happen

A key facilitating rule here is that Trunk Based Development teams exclusively **either** release directly from the trunk - see [release from trunk](#), **or** they make a branch from the trunk for the specifically for the actually

releasing. See [Branch for release](). Teams with a higher release cadence do the former, and those with a lower release cadence to the latter.

Checking out / cloning

All developers in a team that work on a application/service, clone and checkout from the trunk. They will update/pull/sync from that branch a many times a day, knowing that the build within it works perfectly. Their fast source-control system means that their delays are a matter of a few seconds for this operation. They are now integrating their team-mates commits on an hour by hour basis.

Committing

Similarly, developers completing a piece of development work (changes to source code), that provably does not break the build, will commit it back to the trunk. The granularity of that commit (how many a developer would implicitly do a day) can vary and is learned through experience, but commits are typically small.

The developer needs to run the build, to prove that they did not break anything with the commit **before** the commit is pushed anywhere. They might have to do a update/pull/sync before they commit/push the changes back to the team's version control server, and additional builds too. There's a risk a race condition there, but let us assume that is not going to happen for most teams.

Code Reviews

The developer needs to get the commit reviewed. Some teams will count the fact that the code was 'pair programmed' as an automatic review. Others team will follow a conventional design where the commit is marshaled for review before landing in the trunk. In modern portal

solutions, marshaled nearly always means a branch/fork (Pull Request) that is visible to the team.

branch: add_hair_color_to_person_page-story-2109
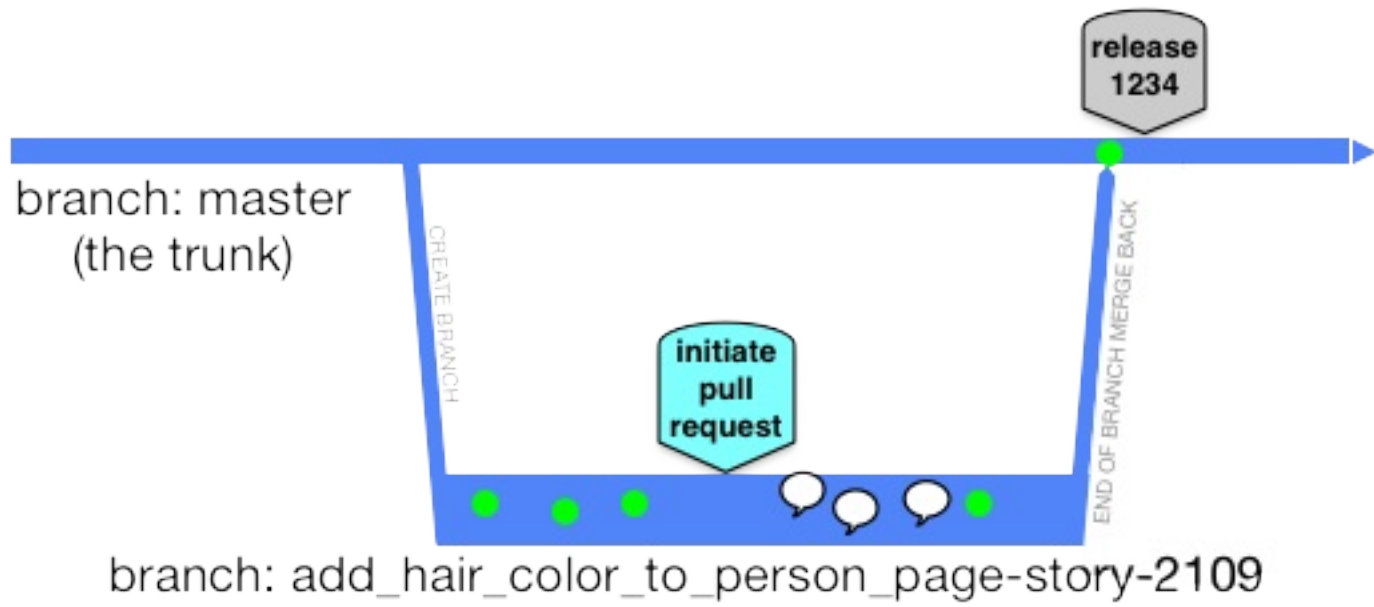
^ the speech bubbles are stylized code review comments

Code review branches can (and should) be deleted after the code review is complete, and be very short-lived. This is tricky for teams new to Trunk Based Development.

Note: You want to keep the commentary/approval/rejection that is part of the review for historical and auditing purposes, but you do not want to keep the branch. Specifically, you do not want to developers to focus on the branch after the code review, and merge back to the trunk.

A safety net

[Continuous Integration](...) (CI) daemons are setup to watch the trunk (and the short-lived feature branches used in review), and as quickly and completely as possible loudly/visibly inform the team that the trunk is broken. Some teams will lock the trunk, and roll-back changes. Others will allow the CI server to do that automatically.
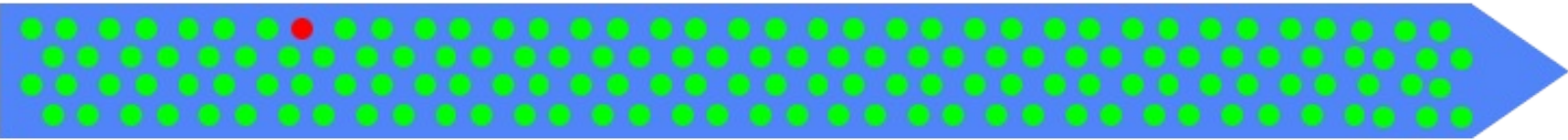
The high bar is verifying the commit before it lands in the trunk. Short-lived Pull Request branches are the modern place for that.

Developer team commitments

As stated, developers are pledging to be rigorous and not break the build. They're also going to need to consider the impact of their potentially larger commits, especially where renames or moves were wholesale, and adopt techniques to allow those changes to be more easily consumed by teammates.

Drilling into 'Distance'

Problematic 'distance' has a few tangible examples:

- Late merges of development that happened more than a couple of days ago.
  - Difficult merges in particularly
- A breaking build that lowers development team throughput, and diverts resources while it is being fixed

References elsewhere

show references

Date	Type	Article
03 Sep	MartinFowler.com	FeatureBranch

| 2009 | article | |
| 16 Jun 2015 | Blog Entry | [Organization Pattern: Trunk Based Development](#) |

Deciding factors

Release cadence

Iteration length

Different Agile teams focus on different iteration lengths. Some teams work at three-week iterations, some two, and some one. Some teams do not have an iteration at all - particularly teams doing Continuous Delivery.

If you are on a four week, or more iteration length, and each of those four weeks varies with proximity to the release and cannot change that you may be in a bind. You may be able to follow the tenants of Trunk Based Development, benefit from a Continuous Integration daemon (as all branching models can), but you are not doing to be able to get all the way to Continuous Delivery (or Continuous Deployment).

Waterfall

This one is easy. If you are doing waterfall, you are not close at all to the "do not break the build" mantra required to do Trunk Based Development. Consider a short-iteration Agile methodology.

Story size

Trunk Based Development needs you to have small stories/tasks. You

starting work on a task, should only be a matter of hours before completing and pushing it forward for code review. Longer than that, and there is going to be pressure to group a bunch of developers on a non-trunk branch. Or worse have developers make branches/forks from your branch, or worse still take intermediate merges from your branch, despite your task being incomplete.

Generally speaking, the whole development team should do whatever it can do break stories/tasks into smaller stories/tasks. In Agile, there is an INVEST mnemonic⬈ that aids in the splitting up or stories.

SCM Technology Choice

You're SCM/VCS/source-control technology choice should facilitate update/pull/sync from the team's trunk many times a day. The elapsed time for the update/pull/sync should be less than three seconds for the situation where you already had latest of everything. It should be no more than fifteen seconds the case of the shared trunk being ahead of you.

Older versions of ClearCase and PVCS Dimensions would be 30 minutes for the former and 45 minutes for the latter. Double that if two team-mates were simultaneously trying to do the update/pull/sync operation. In that configuration, it was completely impossible for teams to choose to do trunk based development.

Binaries in the Repo?

Depending on how many and how often they update, some SCM/VCS/source-control technologies are better than others. Perforce can handle terabytes of binaries and textual source. Subversion aims to. Git can only do so if configured in Git-LFS mode.

Repo size?

Git and Mercurial want to constrain repository history (ignoring Git-LFS) to 1GB. There are field reports of clones being many times bigger than that and still working, but the development team suggests 1GB as the top limit. In order to use Git and push through that ceiling yearly, you might be in a situation where you have to keep archiving a repository, and starting a new one with no history to have more head room. Archiving might look like renaming the repository in Github, and turning it read-only so that all the history, issues, and code review comments are intact.

Peak commit frequency

In Git if a colleague beat you to a commit/push on a branch (their code-review and automated CI passed), when you thought you were going to push, Git will informed you that you have to pull first. You pull, your resolve merge clashes (hopefully none), and your push again. You might struggle to find a window open long enough to push without encountering the same problem. Forks make that easier, and robots keeping pull-request branches abreast of origin:master helps a lot. Thus for developers wanting to land their commitsGit increasing commit frequency to the shared repo means contention and an artificial serialization. Microsoft acknowelged this as one of the motivations to their Git Virtual File System (~~GitVFS~~ GVFS).

Git has critical serialization points that will cause a queue to back up badly

> — Brian Harry
> [More on GVFS](#)

We're sure that within a few years, Git will be able to hangle huge scale too. Whether with the Microsoft technologies, or something else.

Conways Law

The org making applications and services that reflect the organization's own structure. If your organization feels like this, and a Monorepo does not feel right, then MicroServices could the direction for you.

Database migrations

In order get into the Trunk Based Development way, you'll need to find a way to handle table-shape changes, and even population of existing rows where new/changed columns happen. Pramod Sadlage and Scott Amber's book "Refactoring Databases: Evolutionary Database Design" goes into that much more, as does the [Continuous Delivery](#) book.

Shared code

Trunk Based Development teams typically have common code ownership rules around contributions to different parts of the source tree. If they do not have a full egalitarian system, they have objecting rules for the tree that focus on standards and come with a promise of a prioritized and fair code review. Trunk Based Development teams might have fine-grained write permissions for directories within the trunk, but **never** have any impediment to reading files in th trunk - everyone can see everything.

Build times

Keeping build times short is key.

Version Control System (VCS) - choices

The importance of integrated code review

Commercial VCS technologies and platforms were disrupted with the advent of gated code reviews that were coupled to a mechanism to quickly consume (merge) the contribution. Code review for "committers" would have been disruptive enough, but when it arrived it arrived for unknown (to the dev team) contributors by way of "forks".

All VCS technologies and platforms are measured by their adherence to forks, pull requests, integrated code review and possibly hooks into CI servers.

Read more in [Game Changes Review - Google's Mondrian](#) and [Game Changers - Github's Pull Requests](#).

Git and Mercurial

[Git website](#) and [Mercurial website](#)

Git and Mercurial have been popular DVCS technologies for many years. Portals like Github make Git the default choice for SCM/SVC/source-control. While the Linux Kernel is maintained with Git, and definitely

takes advantage of the D-Distributed aspect of the DVCS of Git (in that many divergent versions of kernel can exist over long periods of time), most enterprises are still going to count a single repository as the principal one, and within that a single branch as the long-term "most valuable" code line.

It is perfectly possible to do Trunk Based Development in a Git repository. By convention 'master' is the long term most valuable branch, and once cloned to your local workstation, the repository gains a nickname of 'origin'.

Forks

An effective Trunk Based Development strategy, for Git, depends on the developer maintaining a fork of the origin (and of master within), and Pull-Requests being the place that ready to merge commits are code reviewed, **before** being consumed back into origin:master. Other branching models use the same Pull-Request process for code-reviews too - it is the normal way of working with Git since GitHub rolled out the feature.

Size Limits

Historically, Git and Mercurial were not great at maintaining a zipped history size greater that 1GB. Many teams have reported that they have a repository size larger than that, so opinions differ. One way that you can reach that 1GB ceiling quickly is with larger binaries. As Git keeps history in the zipped repository, even a single larger binary that changes frequently can push the total use above 1GB.

With the likes of correctly configured Git-LFS extension to Git, though, the 1GB limit can be avoided or delayed many years.

Git also has Submodules⊠ and Subtrees⊠ to allow large federations of modules, within one cloneable set. For their Android initiative, Google

made Git-repo too.

Root level branches

It'll be clear later why we mention this, but Git and Mercurial maintain branches from the root folder of the checkout clone, and maintains a single permission for a user in respect of read and/or write on the branch and/or repository.

Future development

There is a suggestion that Mercurial is receiving contributions that will allow it to push into the very repository territory the likes of Google needs.

Git and Mercurial don't have branch or directory permissions, but some of the platforms that bundle them, add branch permissions.

Linux Torvalds presenting Git to Googlers

Back in 2007, Linus Torvalds presented his Bitkeeper inspired Git to Googlers in their Mountain View office:

Tech Talk: Linus Torvalds on git

Git
(silly names is what we do best)

Source code control the way it was meant to be!

Linus Torvalds
torvalds@linux-foundation.org

Video Available at https://youtu.be/4XpnKHJAok8

He had started making it two years before, and it is now the #1 VCS choice. Google had been running their Monorepo style Trunk for a few years at this point, without regret. Some Googlers would later extend their Perforce (see below) setup to allow Git operation of local branches on developer workstations.

Platform Software Choices

- Github - Git, cloud
- Github Enterprise - Git in Github's on-premises edition
- Gitlab - Git, cloud and on-premises install
- Atlassian's Bitbucket server - Git and Mercurial
- RhodeCode - Git, Mercurial
- Various Collabnet products and services for Git
- Microsoft's Team Foundation Server - git, on-premises install

Perforce

Website

Perforce is a closed-source, industrial strength VCS. Pixar store everything needed to make a movie in it, and Addidas store all their designs in it. Until 2012, Google had their Trunk and many tens of terrabytes of history in it. They moved off it to an in-house solution as they outgrew it. Perforce is peculiar in that its 'p4d' (a single server-side executable binary file) is the whole server and does not need to be installed - just executed.

Perforce is the last VCS technology that ordinarily maintains the read-only bit on the developer workstation. You definitely need a plugin for your IDE to handle the wire operations with the server, so you are not confronted with the fact that source files are read-only. Because the Perforce (p4) client having to involve the server for the flipping of read only bits in respect of editing source files, it requires a permanent connection to the server. What that facilitates is speed of operation for very large sets of files on the client. The Perforce server already knows what files need to have updated in your working copy, ahead of you doing 'p4 sync' operation. It negates the need for a directory traversal looking for locally changed files, and it means the sync operation can be limited to a second or two.

Historically Perforce was not able to **locally** show the history of the files within it. It needed that server connection again for history operations. A number of DVCS capabilities in newer versions of Perforce (see below) allow local history now though.

Perforce allows branches to be set up at any sub-directory not just the root one. It also allows read and/or write permissions to be specified at any directory (or branch) within large and small source trees.

No Code Review

Perforce does not have code-review features integrated into its server daemon. By customizing a GitSwarm (Gitlab) 'side install', Perforce now has a code review capability. It also has it with an alternate side-install called just Swarm (an slightly older product), that doesn't not offer the Git capability of GitSwarm.

Git Fusion

There's a VM appliance from the Perforce people, that can sit in your infrastructure and mediate between the perforce server, and your wish to use Git in an idiomatic way on your development workstation.

With a Git-fusion clone from a Perforce repository, and client spec was specified, you get the subsetted representation of the source tree, complete with history. That's a neat feature. Things checked out through Git-Fusion also are not encumbered by the read-only bit feature.

GitSwarm kinda replaces this.

p4-git and p4-dvcs

P4-git is very similar to the Git fusion technology but is not made by the Perforce people themselves. It also does not require the launching of second server appliance (as Git Fusion does).

In 2015, the perforce technologies were extended to include custom DVCS features. All the features of P4-git but without the Git compatibility.

As for Git-fusion, things checked out through p4-git and p4-dvcs are not encumbered by the read-only bit control of p4d.

Subversion

Subversion (Svn) has been in development for 16 years and was a sorely needed open-source replacement for CVS. It chases some of the features of Perforce, but is developed quite slowly. Nobody has pushed Subversion to the Perforce usage levels, but that is claimed as a possibility.

Note also the Subversion team themselves, do not do trunk based development, despite Subversion have default root directories of 'trunk', 'tags' and 'branches' for newly-created repositories.

Subversion, like Perforce, has read and write permissions down to the directory and branch.

Interestingly there is a "Subversion vs Git" website. It does not have a feedback/contact mechanism in order suggest updates (some claims are out of date).

No Code Review

Note that Subversion has no local branching capability, and to get code review you need to install third-party servers along side it or (better choice) use a platform that integrates code review like those below.

Git-Svn

There is an extension to Git that allows it to deal with a Subversion backend. A Git-subversion clone has all the local history, local-branching possibilities of Git. That clone from subversion can be many tens of times slower (for the same history set), than the equivalent clone from Git. The local branching possibilities afforded by this mode of operation are very

handy, and it should work easily with whatever Svn hosting platform you installed.

Platform Software Choices

- [RhodeCode](#) - installable on-premises
- Various [Collabnet](#) products and services.
- [ProjectLocker](#) - cloud
- [Deveo](#) - cloud
- [RiouxSvn](#) - cloud
- [SilkSvn](#) - cloud
- [Assembla](#) - cloud and installable on-premises
- [XP-dev](#) - cloud
- [Codeplex](#) - cloud

Team Foundation Server - TFS

[Website](#)

Microsoft launched TFS in the mid-2000's with a **custom VCS technology** "TFVC". It is said that they have an internal 'SourceDepot' tool that is a special version of Perforce compiled for them in the nineties, and that TFS reflects some of the ways of working of that technology. It has grown to be a multifaceted server platform. Perhaps even a one-stop shop for the whole enterprise's needs for application lifecycle management. It is perfectly compatible with a Trunk Based Development usage.

PlasticSCM

[Website](#)

PlasticSCM is a modern DVCS like Git and Mercurial, but closed-source.

It is compatible with Trunk Based Development and quite self-contained (has integrated code review, etc). Plastic is very good with bigger binaries and comes with an intuitive "Branch Explorer" to see the evolution of branches, view diffs, execute merges, etc. For sizes of individual repos, multiple terrabytes is not unheard of. A least for some of the ganes-industry customers.

It is also the first modern VCS to have semantic merge - it understands select programming languages and the refactorings developers perform on them. For example "move method", where that method is 50 lines long, isn't 50 lines added and 50 deleted in one commit, it is a much more *exact* and terse diff representation.

Plastic even calmly handles a situation where one developer moves a method within a source, and another simultaneously changes the contents of the method in its former location. Plastic does not consider that a clash at all, and just does the merge quietly - the method moves and is changed in its new location.

Feature flags

Feature Flags are a time-honored way to control the capabilities of an application or service in a large decisive way.

An Example

Say you have an application or service that launches from the command line that has a `main` method or function. Your feature flag could be `--withOneClickPurchase` passed in as a command line argument. That could turn on code in the app to do with Amazon's patented one-click purchasing experience. Without that command line argument, the application would run with a shopping cart component. At least that's the way the developers coded that application. The 'One Click Purchase' and 'Shopping Cart' alternates are probably also the same language that the business people associated with the project use. It gets complicated in that flags need not be implicitly a/b or new/old, they could be additive. In our case here, there could also be a `--allowUsersToUserShoppingCartInsteadOfOneClick` capability. Flags can be additive, you see.

Flags Are Toggles

Industry Luminary, Martin Fowler, calls these Feature Toggles, and wrote a foundational definition (see refs below). Feature Flags is in wider use by the industry, though, so we're going with that.

Granularity

If could be that the flag controls something large like the us of a component. In our case above we could say that `OneClickPurchasing` and `ShoppingCart` are the names of a components. It could be that the granularity of the flag is much smaller - Say Americans want to see temperatures in degrees Fahrenheit and other nationalities would prefer degrees Centigrade/Celsius. We could have a flag `--temp=F` and `--temp=C`. For fun, the developers also added `--temp=K` (Kelvins).

Implementation

For the `OneClickPurchasing` and `ShoppingCart` alternates, it could be that a `PurchasingCompleting` abstraction was created. Then at the most primordial boot place that's code controlled, the `--withOneClickPurchase` flag is acted upon:

Java, by hand:

```
if args.contains("--withOneClickPurchase") {
  purchasingCompleting = new OneClickPurchasing();
}
```

Java Dependency Injection via config:

```
bootContainer.addComponent(classFromName(config.get("purchasing
```

There are much more ways of passing flag intentions (or any config) to a runtime. If you at all can, you want to avoid if/else conditions in the code where a path choice would be made. Hence our emphasis on an abstraction.

Continuous Integration pipelines

It is important to have CI guard your reasonable expected permutations of flag. That means tests that happen on an application or service after launching it, should also be adaptable and test what is meaningful for those flag permutations. It also means that in terms of CI pipelines there is a fan-out **after** unit tests, for each meaningful flag permutation. A crude equivalent is to run the whole CI pipeline in parallel for each meaningful flag permutation. That would mean that each commit in the trunk kicks off more that one build - hopefully from elastic infrastructure.

Runtime switchable

Sometimes Flags/Toggles set at app launch time is not enough. Say you are an Airline selling tickets for flights online. You might also rental cars in conjunction with a partner - say 'Really Cool Rental Cars' (RCRC). The connection to any partner or their up/down status is outside your control, so you might want a switch in the software that works without relaunch, to turn "RCRC partner bookings" on or off, and allow the 24/7 support team to flip it if certain 'Runbook' conditions have been met. In this case, the end users may not notice if Hertz, Avis, Enterprise, etc are all still amongst the offerings for that airport at the flight arrival time.

Key for Runtime switchable flags is the need for the state to persist. A restart of the application or service should not set that flag choice back to default - it should retain the previous choice. It gets complicated when you think about the need for the flag to permeate multiple nodes in a cluster of horizontally scaled sibling processes. For that last, then holding the flag state in Consul⊞, Etcd⊞ (or equivalent) is the modern way.

Build Flags

Build flags affect the application or service as it is being built. With respect to the OneClickPurchase flag again, the application would be incapable at runtime of having that capability if the build were not invoked

with the suitable flag somehow.

A/B testing and betas

Pushing code that's turned off into production, allows you to turn it on for ephemeral reasons - you want a subset of users to knowingly or unknowingly try it out. A/B testing (driven by marketing) are possible with runtime flags. So is having beta versions of functionality/features available to groups.

Tech Debt - pitfall

Flags get put in to codebases over time, and often get forgotten as development teams pivot towards new business deliverables. Of course, you want to wait a while until it is certain that you are fixed on a toggle state, and that's where the problem lies - the application works just fine with the toggle left in place, and the business only really cares about new priorities. The only saving grace is the fact that you had unit tests for everything, even for code that is effectively turned off in production. Try to get the business to allow the remediation of flags (and the code they apply to) a month after the release. Maybe add them to the project's readme with a "review for delete" date.

History

Some historical predecessors of feature toggles/flags as we know it today:

- Unified Versioning through Feature Logic (Andreas Zeller and Gregor Snelting, 1996) - white paper.
- Configuration Management with Version Sets: A Unified Software Versioning Model and its Applications (Andreas Zeller's, 1997) - Ph.D. thesis.

There's a warning too:

- "#ifdef considered harmful" (Henry Spencer and Geoff Collyer, 1992) - white paper.

Brad Appleton says:

> " The thing I do not like about feature-toggles/flags is when they end up NOT being short-lived as intended, and we end up having to revisit Spencer and Collyer's famous paper. The funny thing is feature-branches started out the same way. When they were first introduced it was for feature-teams using very large features, and the purpose of the separate branches was because too many people were trying to commit at the same time to the same branch. So the idea was use separate branches (for scale) and teams would integrate to their team-branch daily or more often WITH at least nightly integration across all feature-branches [sigh].

References elsewhere

[show references](#)

Date	Type	Article
29 Oct 2010	MartinFowler.com article	[Feature Toggle](#)
30 May 2011	TechCrunch article	[The Next 6 Months Worth Of Features Are In Facebook's Code Right Now, But We Can't See](#)
19 Jun		[Branching Strategies: Feature Branches vs](#)

2013	Slides from a talk	[Branch by Abstraction](#)
10 Oct 2014	Conference Talk	[Trunk Based Development in the Enterprise - Its Relevance and Economics](#)
08 Feb 2016	MartinFowler.com article	[Feature Toggles](#)

Branch by Abstraction

Branch by Abstraction is a set-piece technique to effect a 'longer to complete' change in the trunk. Say a developer (or a pair of developers), has a change that is going to take five days to complete. There could easily be pressure to create a branch for this - somewhere that can be unstable for a period of time before it completes (and get's merged back somewhere).

There may be some aspect of repetition to the coding activities that makes it a longer to complete. No matter, the change was predicted as being time-consuming, complex, destabilizing/disruptive to every else in the development team.

Rules:

1. There's also a lot of developers already depending on the code that is subject of the 'longer to complete' change, and we do not want them to be slowed down in any way.
2. No commit pushed to the shared repository should jeopardize the ability to go live.

Ideal steps

For simplicity's sake, let us say there is code that is 'to be replaced', code 'to be introduced'.

1. Introduce an abstraction around the code that is to be replaced, and

commit that for all to see. If needed, this can take multiple commits. None of those are allowed to break the build, and all of them could be pushed to the shared repository in order, and as done.

2. Write a second implementation of the abstraction for the to be introduced code, and commit that, but maybe as 'turned off' within the trunk so that other developers are not depending on it yet. If needed, this can take multiple commits as above. The abstraction from #1 may also be occasionally tweaked, but must follow the same rule - do not break the build.
3. Flip the software 'off' switch to 'on' for the rest of the team, and commit/push that.
4. Remove the to be replaced implementation
5. Remove the abstraction

Hopefully, your team IDE that can perform complex refactorings on sets on checkouts in a way that running the build after each is an uneventful validation of the refactorings.

Contrived example

Let's talk about a car having its wheels upgraded. We should never forget that software engineering is nothing like conventional construction, and we want to ram that home. At least, it is nothing like conventional construction where we are not talking about a production line.
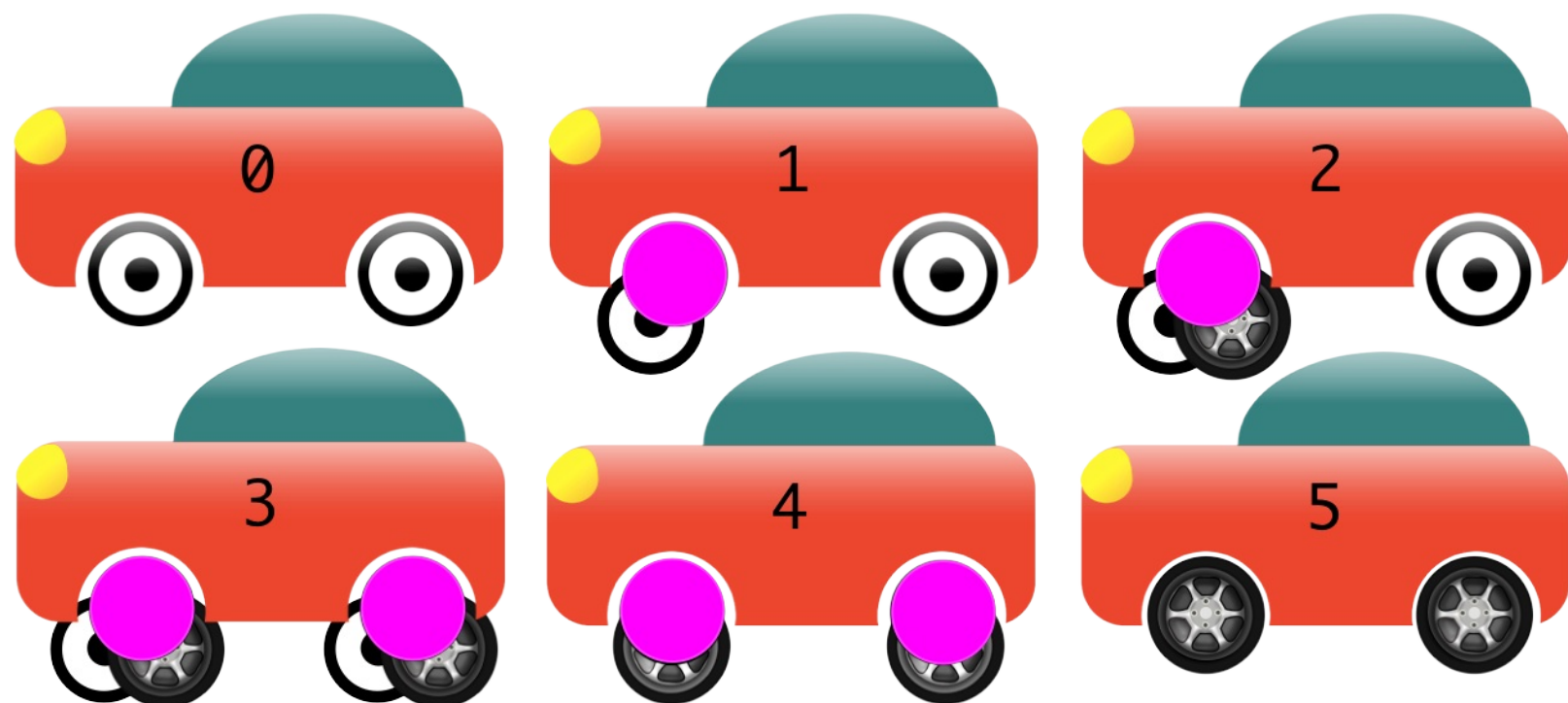
Rules

1. Mechanics must be able to simultaneously work on the upholstery, engine, etc.
2. The car must be drivable after every change.

Steps

All steps are efficiently performed raised up on car jacks/lifts/ramps, before lowering the car to the ground again.

1. One wheel is removed, put in a container that looks pretty much like a wheel (rotates around an axis, can bear weight) and replaced on the car. If driven this wheel functions exactly like the other three wheels.
2. The wheel-like container gains a second better/desired/alternate wheel, within exactly the same physical space (magically). A switch is added inside the car to allow the choice of wheel to be switched conveniently
3. perhaps only before the engine is started, though.
4. The same two operations (#1 and #2) are performed on the other three wheels. Or maybe #1 is done four times, followed by #2 four times. The Mechanics experience will guide which is most efficient.
5. After determining that the new wheels are better, the old wheels are removed from the wheel-like containers and are send for recycling.
6. The wheel-like containers are also removed from the new wheels, either one by one or all four simultaneously.

At any stage, when lowered from the jacks/lift/ramps, the car could have been driven (a 'ready to go-live' metaphor).

We said 'jacks' above, because that's what mechanics use in real life. Software, however, does not follow the rules of gravity, or many of the costs of actual construction. With an IDE for a glove, a single finger could reposition the car in 3D space to allow easy replacement of the wheels.

Software example

A documented case is ThoughtWorks' Go CI-daemon. They changed an Object-Relational mapping library (for persistence), while not slowing down teammates development activities (rule 1), and not jeopardizing the ability to go live (rule 2).

Going from "iBatis" to "Hibernate" for a bunch of reasons, was their plan.

They:

1. Introduced an abstraction around the classes/components using iBatis directly, and ensured that all classes/components indirectly referring to iBatis were changed to refer to the abstraction instead.
2. Wrote a second implementation of the abstraction, introducing Hibernate to the codebase, perhaps tweaking the abstraction where needed.
3. Did a tiny commit that turned on Hibernate for all teammates.
4. Removed iBatis, then the abstraction and the on/off old/new switch.

As it happens you could leave the abstraction in place, if your unit tests are able to benefit because of the possibility of another seam that can be mocked.

Secondary benefits

Cheaply pause and resume 'migrations'

The migration from old to new can be paused and resumed later casually. This is because the build guards the second, incomplete, implementation. It does so merely because of a compile stage that turns the abstraction and somewhere between 1 to 2 implementation into object code. If there are unit tests for the two alternates, then even more so.

If on a real branch, the casual restart of the paused initiative is missing. There's possibly an exponential cost of restart given the elapsed time since the initiative was paused.

Pause and resume is much more likely in an enterprise development organization that does not have limitless coffers.

Cancellation of a project is still cheap

In the case of abandonment, deleting a real long running feature branch is cheaper, but deletion of a branch by abstraction *thing* is only incrementally more expensive.

History

Teams employed Branch by Abstraction many years before it got its name (Stacy Curl named it in 2007), but it is unknown when the first implementation was. Before the adoption of BbA, teams **had to** make a branch for the big lengthy disruptive change, or do it with an incredible amount of choreography: "hey everyone, take a week of vacation now".

With the Branch by Abstraction technique, Trunk Based Development was less likely to be temporarily or permanently abandoned for a multi-branch model.

References elsewhere

| Date	Type	Article
26 Apr 2007	Blog entry	[Introducing Branch by Abstraction](#)
05 May 2011	ContinuousDelivery.com article	[Make Large Scale Changes Incrementally with Branch By Abstraction](#)
21 Jun 2013	Blog entry	[Branching Strategies: Feature Branches vs Branch by Abstraction](#)
14 Oct 2013	Blog entry	[Application Pattern: Verify Branch By Abstraction](#)
07 Jan 2014	MartinFowler.com article	[BranchByAbstraction](#)

Branch for release

Branch: only when necessary, on incompatible policy, late, and instead of freeze

— Laura Wingerd & Christopher Seiwald
(1998's High-Level SCM Best Practices white paper from Perforce)

If a team is pushing production releases monthly, then they are also going to have to push bug-fix releases between planned releases. To facilitate that, it is common for Trunk Based Development Teams to make a release branch on a just in time basis - say a few days before the release. That becomes a stable place, given the developers are still streaming their commits into the trunk at full speed.

The incompatible policy (ref Wingerd & Seiwald above), that the release branch "should not receive continued development work".
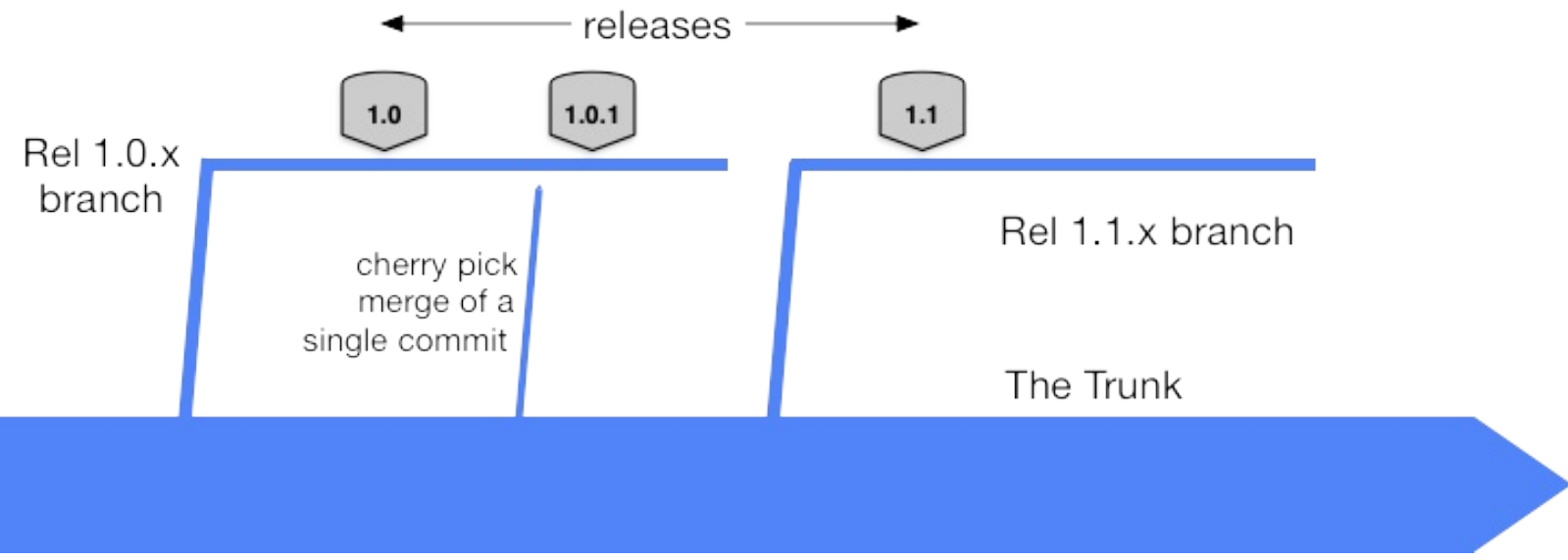
^ Trunk, two release branches, three releases, and a bug-fix

CD teams do not do release branches

High throughput, Continuous Delivery teams can ignore this - if they had a lemon in production, they choose a roll-forward strategy for solving it, meaning the fix for a bug is in the trunk, and the release to production is from the trunk.

Who's committing where?

Developers are committing (green dots) at the highest throughput rate to the trunk, and don't slow up around a branch-cut.
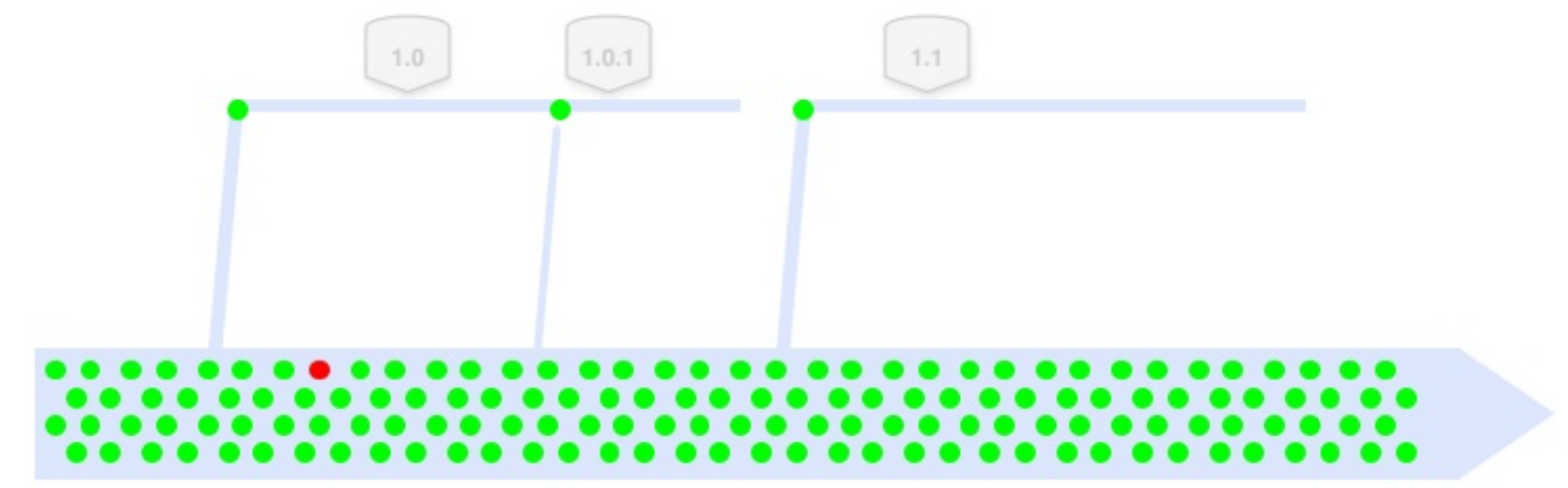
The branch cut itself is a commit. Subversion and Perforce would technically have a bigger commit here, but all VCS systems in use today would count the commit as lightweight in terms of it's impact on the history/storage.

That red dot is an accidental build break that was fixed (somehow) soon after.

Fix production bugs on Trunk

The best practice for Trunk Based Development teams is to reproduce the bug on the trunk, fix it there with a test, watch that be verified by the CI server, then cherry-pick that to the release branch and wait for a CI server focusing on the release branch to verify it there too. Yes, the CI pipeline that guards the trunk is going to be duplicated to guard active release branches too.

Cherry-pick is not a regular merge

A cherry-pick merge takes a specific commit (or commits) and merges that to the destination branch. It skips one or more commits that happened before it, but after the branch was cut.

Late branch creation

Some teams [release from a tag on the trunk](#) and do not create a branch at that time. That in itself is **an alternate practice to this one, "branch for release"**.

Those teams wait for a bug that needs fixing for a Released, before creating a branch from the release tag (if they are not going to just issue another release from the trunk).

Brad Appleton points out that many do not realize that branches can be created **retroactively**. That is taken advantage of here in the case of bugs after "release from tag", or even changes for point releases.

Directionality of cherry-pick

This one is controversial even within teams practicing everything else about Trunk Based Development: you should not fix bugs on the release in the expectation of cherry-picking them back to the trunk, in case you forget to do that. Forgetting means a regression in production some weeks later (and someone getting fired). It can happen if things are being fixed in the night by a tired develop who wants to get back to bed.

Of course, sometimes you cannot reproduce the bug on trunk so you have to do it the other way round.

Release branch deletion

You really should delete release branches when releases from succeeding release branches have gone to prod. This is a harmless tidying activity - branches can be undeleted again quite easily.

References elsewhere

show references

Date	Type	Article
1998	White Paper	High-level Best Practices in Software Configuration Management

Release from trunk

Teams with a very high release cadence do not need (and cannot use) release branches at all. They have to release from the trunk.


It is most likely that such teams do not use a Dewey-decimal release numbering scheme, and instead have something referent to the commit number or date and time. They probably also choose to roll forward and fix the bug on the trunk as if it were a feature, albeit as quickly as possible.

Here's what stylized commits look like:
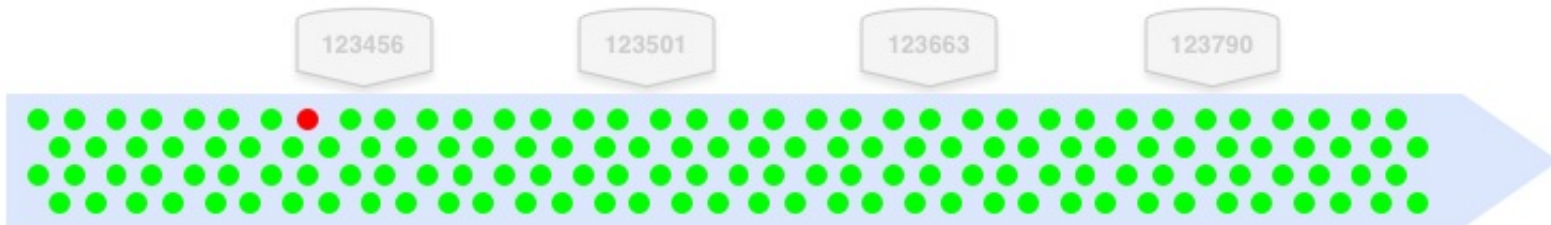
No slow down around a release, and bug fixes inline.

Teams with one release a day (or less) **might** still make a branch, to cherry-pick the bug-fix to and release from:
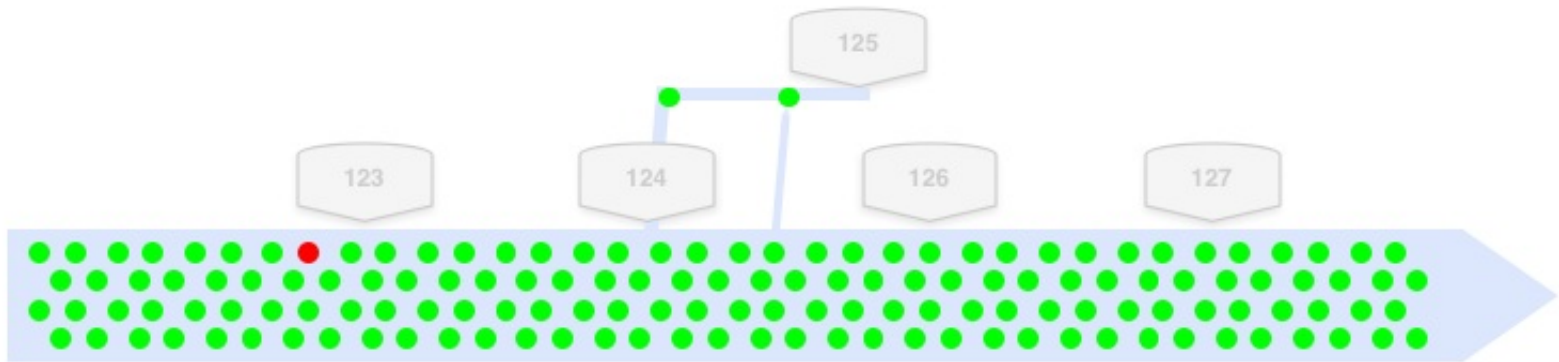
Branches can be made retroactively

Newbies to source-control systems often forget that you don't have to make a branch because you think you might need it in the future. For any source-control technology made today, you can choose the revision in the past to branch from. The outcome is exactly the same as if you had made it at the time.

Continuous Integration (CI)

individuals practice Trunk Based Development, and teams practice CI

— *Agile* Steve Smith

Continuous Integration - as defined

For many years CI has been accepted by a portion of software development community to mean a daemon process that is watching the source-control repository for changes and verifying that they are correct, **regardless of branching model**.

However, the original intention was to focus on the verification **single integration point** for the developer team. And do it daily if not more. The idea was for developers themselves to develop habits to ensure everything going to that shared place many times a day was of high enough quality, and for the CI server to merely verify that quality, nightly.

CI as we know it today, was championed by Kent Beck, as one of the practices he included in "Extreme Programming"⬀ in the mid-nineties. Certainly in 1996, on the famous 'Chrysler Comprehensive Compensation System' (C3) project⬀ Kent had all developers experiencing and enjoying the methodology - including the continuous integration aspect. The language for that project was Smalltalk and the single integration point

was a Smalltalk image (a technology more advanced than a "mere" source-control systems that rule today).

Thus, the intention of CI, as defined, was pretty much exactly the same as Trunk Based Development, that emerged elsewhere. Trunk Based Development did not say anything about Continuous Integration daemons directly or indirectly, but there is an overlap today - the safety net around a mere branching model (and a bunch of techniques) is greatly valued.

Martin Fowler (with Matt Foemmel) called out Continuous Integration in an article in 2000 ⊡, (rewritten in 2006⊡), and ThoughtWorks colleagues went on to build the then-dominant "Cruise Control"⊡ in early 2001. Cruise Control co-located the CI configuration on the branch being built next to the build script, as it should be.

CI daemons performing verifications

Every development team bigger than, say, three people needs a CI daemon to guard the codebase against bad commits and mistakes of timing. Teams have engineered build scripts that do their thing quickly. Hopefully all the way from compile through functional tests (perhaps leveraging mocking at several levels) and packaging. There is no guarantee that a developer ran the build though before committing, though. The CI daemon fills that gap and verifies commits are good once they land in the trunk. Enterprises have either built a larger scaled capability around their CI technology so that it can keep up with the commits/pushes of the whole team, or by the batching of commits and using less computing poer to track and verify work.

Humans and daemons do the same build

It is important to note that the build script that developers run prior to checking, is **the same one** that the CI process follows. Because of a need

for succinct communication to development teams, the build is broken into gated steps. The classic steps would be compile, test-compile, unit test invocation, integration test invocation, functional test invocation.

Radiators

A popular radiator-style visual indication of progress would be those shown as a left-to-right series of Green (passing) or Red (failing) icons/buttons with a suitably short legend:

| #4321 | Compile | Unit | Integration | Functional |
| #4322 | Compile | Unit | Integration | Functional |
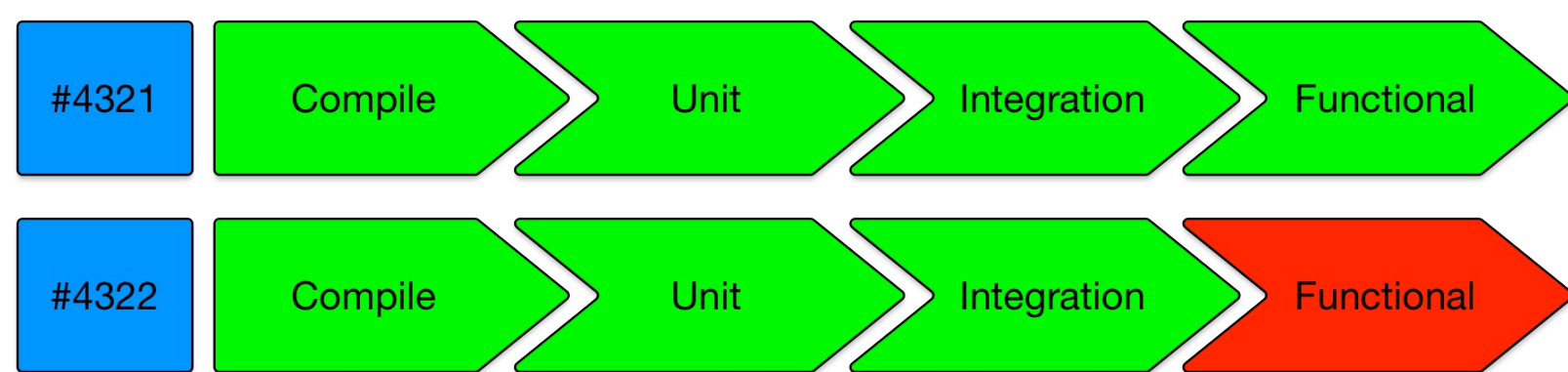
This should go up on TVs if developers are co-located. It should also be a click through from notification emails.

Quick build news

The elapsed time between the commit and the "this commit broke the build" notification, is key. That is because the cost to repair things in the case of a build breakage goes up when additional commits have been pushed to the branch. One of the facets of the 'distance' that we want to reduce (refer 5 minute overview) is the distance to break.

Pipelines - further reading

Note: Continuous Integration Pipelines are better described in the bestselling Publications - Continuous Delivery book. So are dozens of nuanced, lean inspired concepts for teams pushing that far.

Advanced CI topics

CI per commit or batching?

Commiting/pushing directly to the shared trunk may be fine for teams with only a few commits a day. Fine too for teams that only have a few developers who trust each other to be rigorous on their workstation before committing (as it was for everyone in the 90's).

Setups having the CI server single threading on builds and the notification cycle around pass/fail will occasion lead to the **batching** in a CI job. This is not a big problem for small teams. Batching is where one build is verifying two or more commits in one go. It is not going to be hard to pick apart a batch of two or three to know which one caused the failure. You can believe that with confidence because of the high probability the two commits were in different sections of the code base and are almost never entangled.

If teams are bigger, though, with more commits a day, pushing something incorrect/broken to trunk could be disruptive to the team. Having the CI daemon deal with **every commit** separately is desirable. If the CI daemon is single-threading "jobs" there is a risk that the thing could fall behind.

Master / Slave CI infrastructure

More advanced CI Server configurations have a master and many slaves setup so that build jobs can be parallelized. That's more of an investment than the basic setup, and but is getting easier and easier in the modern era with evolved CI technologies and services.

The likes of Docker means that teams can have build environments that are perfectly small representations of prod infra for the purposes of testing.

Tests are never green incorrectly.

Well written tests, that is - *there are fables of suites of 100% passing unit tests with no assertions in the early 2000's.*

Some teams focus 99.9% of their QA automation on functional correctness. You might note that for a parallelized CI-driven *ephemeral* testing infrastructure, that response times for pages are around 500ms, where the target for production is actually 100ms. Your functional tests are not going to catch that and fail - they're going to pass. If you had non-functional tests too (that 0.1% case) then you might catch that as a problem. Maybe it is best to move such non-functional automated tests to a later pipeline phase, and be pleased that so many functional tests can run through so quickly and cheaply and often, on elastic (albeit ephemeral) CI infrastructure.

Here is a claim: Tests are never green (passing) incorrectly. The inverse - a test failing when the prod code it is verifying actually good - is common. QA automators are eternally fixing (somehow) smaller numbers of flakey tests.

A CI build that is red/failing often, or an overnight CI job that tests everything that happened in the day - and is always a percentage failing is of greatly reduced value.

A regular CI build that by some definition is comprehensive, well written and **always green** unless there's a genuine issue is extremely valuable.

Once you get to that trusted always green state, it is natural to run it as often as you can.

CI Pre or Post Commit?

In terms of breakages, whether incorrect (say 'badly formatted'), or

genuinely broken, finding that out **after the commit** is undesirable. Fixing things while the rest of the team watches or waits, is a team-throughput lowering practice.

Pushing straight to trunk to initiate CI
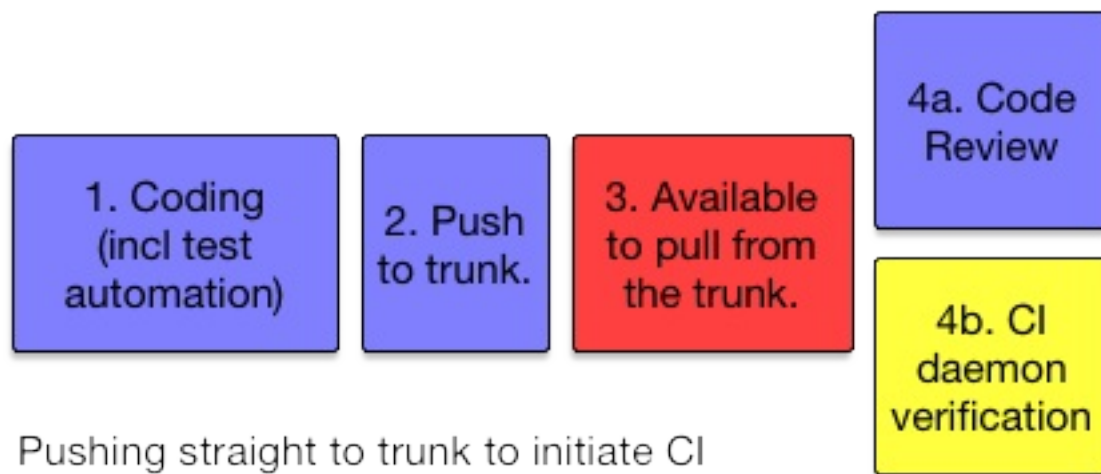
Yellow = automated steps, Red = a potential to break build for everyone

Note: for committing/pushing straight to the shared trunk, code review and CI verification can happen in parallel. Most likely though is reviews happening after the CI daemon has cast its vote on the quality of the commit(s).

Better setups have code-review and CI verification before the commit lands in the trunk for all to see:

Code-Review and CI before code lands in trunk for all to see

It is best to have a human agreement (the code review), and machine agreement (CI verification) before the commit lands in the trunk. There is still room for error based on timing, so CI needs to happen a second time **after** the push to the shared trunk, but the chance of the second build failing so small that an automated revert/roll-back is probably the best way

to handle it (and a notification).

The high bar, today

Highest throughput teams have CI server setups that prevent breakage of the trunk. That means that commits are verified before they land in the trunk to the extent where teammates can update/sync/pull.

The problem this solves is when the rate of commit into the trunk would be too high to have an auto-rollback on build failure. In Google one commit lands in the trunk every 30 seconds. Few CI technologies (and source control systems) can keep up with that in a way that isn't batching (to some degree of interpretation). You'd be stopping the line too often for humans to make sense of a train wreck of red builds, where only one two were actual breakages rather than just bad timing.

It would not be computationally hard to recreate a last-green-plus-this-commit contrived HEAD to verify a commit in isolation to the other 20 that arrived in the last ten minutes, but that would be a crazy amount of computing power to do so. Better to marshal the pending commit in a place where it looks like it is immediately following a previously known green (passing) commit and is not yet on the shared trunk.

That place has a name - a branch (or a branch of a fork the Github way). It is a perfect place to CI verify the commit before auto-merging it to the shared trunk (if you want to auto-merge after code review approvals).

The new problem is how do you prevent that short-lived feature branch from sleepwalking into a long-lived feature branch with half a dozen developers keeping it from being 'complete' (somehow) and merged back. You can't with tools today, but it would be cool if you could have a ticking clock / count down on those branches at creation to enforce its 'temporary' intention.

Refer to [Game Changers - Google Home Grown CI and Tooling](#) for more information on the high commit rate CI stuff. Note too that they do not have a temp branch set up to facilitate that.

Industry CI daemon confusion

ThoughtWorks commissioned a survey - "No One Agrees How to Define CI or CD"⊡.
That the hypothesis of Continuous Integration being thought of as compatible with branching models other than Trunk Based Development was, unfortunately, shown to be true. Their chief scientist, Martin Fowler, writes about the general effect in his "Semantic Diffusion" article⊡.

Martin also wrote specifically on the lamentable *pat on the back* that multi-active-branch teams give themselves when they set up a CI server/daemon for one or all of those branches: "Continuous Integration Certification" ⊡ and within that *a great coin* "Daemonic Continuous Integration" for this effect.

This site's use of CI and Trunk Based Development

Given other popular branching models (that are not Trunk Based Development) **also** benefit from CI servers watching for and verifying commits, this site is going to refer to the commit to a *enforced single shared source-control branch* practice as Trunk Based Development.

There are many CI technologies and services available for teams to use. Some are free, and some are open source. Some store the configuration for a pipeline in VCS, and some store it somewhere else. In order to more smoothly support [branch for release](#), the best CI solutions co-locate the configuration for a pipeline in the same branch too.

Server/daemon implementations

- [Jenkins](#) commercial service, for [Jenkins Open Source](#) - on-premises installable
- [Travis-CI](#) - cloud
- ThoughtWorks' [Snap-CI](#) - cloud
- [Circle-CI](#) - cloud
- ThoughtWorks' [Go CD](#) - cloud and on-premises install
- [Codeship](#) - cloud
- Atlassian's [Bamboo](#) - on-premises install
- JetBrains' [TeamCity](#) - on-premises install
- Microsoft's [TFS platform](#) - on-premises install (built into larger platform)

Note, for Jenkins, you should use it with GroupOn's [DotCI](#) to co-locate the config with the thing being built/verified in source-control.

References elsewhere

[show references](#)

Date	Type	Article
10 Sep 2000	MartinFowler.com Article	[Continuous Integration - original version](#)
18 May 2006	MartinFowler.com Article	[Continuous Integration](#)
18 May 2015	Hangout Debate	[Branching strategies and continuous delivery](#)
02 Sep 2015	Conference Presentation	[The Death of Continuous Integration](#)

Continuous Code Review

The high bar today

Continuous Code Review is where the team commits to processing team-mates proposed commits to trunk speedily. The idea is that a system (the code portal probably) allows developers to package up commits for code review and get that in front of peers quickly. That peer developers make a commitment to do code reviews objectively and quickly. There is a cost to multi-tasking, so maybe someone in the dev team who is between work items at that moment should focus on the review before they start new work. With a continuous review ethos, it is critical that code reviews are not allowed to back up.

Pull Requests

The pull-request (PR) model introduced by Github is the dominant code review model today. It is a branch, but a personal one and short-lived. It may suffer many commits before the developer feels the task is complete, and initiates a PR which triggers code review (and the CI daemon to wake up and build/verify the branch). The temporary branch may have received many commits before the developer initiated the pull request. Some developers will squash (rebase) the changes into a single commit before starting code review. Some teams have a policy in favor of or against squash/rebase.

Common Code Owners

Commits being reviewed are never rejected for "Only I am allowed to change source in this package" reasons. Rejections must be for objective reasons.

Enterprise code review - as it was

In enterprises, if code review was done at all prior to 2008, it was done in a batch, and probably a group activity. It was often abhorred as it gave a lead developer/architect a moment to set an agenda, round on a large portion of the attendees and make sure that their own code flubs were not discussed at all.

Historically, open source teams never had the luxury of procrastinating about code review. They either did code reviews as they went (perhaps days were the review cadence, not hours or minutes), or they didn't bother at all.

See also

See [Game Changes - Google's Mondrian](#) and [Game Changes - Github's Pull Requests](#) for the industry impact of continuous code review.
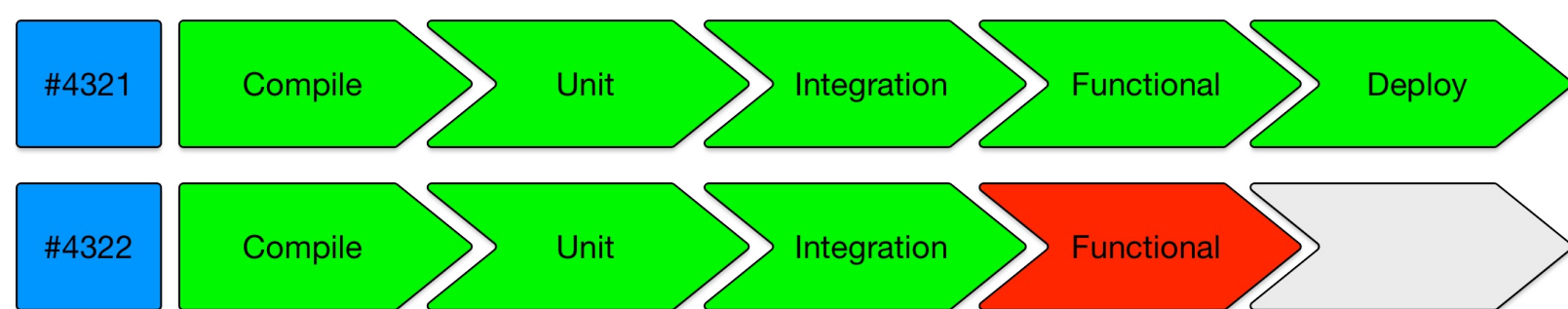
References elsewhere

[show references](#)

Date	Type	Article
13 Nov 2014	Blog Article	[Code Review - the unit of work should be a single commit](#)

Continuous Delivery (CD)

Continuous Delivery is the practice expanding your Continuous Integration (CI) usage to automatically re-deploy a proven build to a QA or UAT environment. If the bounce time for a deployment is quick enough, then it could be that you are doing that for every commit that lands in the shared trunk. The Radiator's pipeline view would become:

| #4321 | Compile | Unit | Integration | Functional | Deploy |

| #4322 | Compile | Unit | Integration | Functional | |

The [bestselling book of the same name](#) by Jez Humble and Dave Farley, details the 'marching orders' for many companies, where there is whole dev-team improvement agenda.

A Layer above Trunk Based Development

Continuous Delivery is a broad multifaceted subject, that sits on top of Trunk Based Development as a practice. This website, and this page in particular is not going to give it justice. Head on over to `ContinuousDelivery.com` and understand too that "lean experiments" are the part of CD and not so much the concern of Trunk Based Development.

Continuous Deployment

An automatic push all the way into production; Maybe every commit

This is an extension of 'Continuous Delivery', but the deployment is to production. Certain types of startups like Netflix, Etsy and Github deploy their major application to production with each commit. Companies that have applications/services where clients/customers could lose money are much less likely to *firehose into production*.

References elsewhere

show references

Date	Type	Article
2010	Jez Humble's Continuous Delivery portal	ContinuousDelivery.com
5 Jan 2015	TheGuardian newspaper on their CD	Delivering Continuous Delivery, continuously

Concurrent development of consecutive releases

Your company wants a stream of major functionality to arrive in the application you are pushing live at a regular cadence. Because you are good Extreme Programmers, you know that consecutive development of consecutive releases is best. However, the effort and length of time needed to complete each of the major pieces of functionality is large enough to require different project teams cooperating towards that plan. Some of those teams will be within the same codebase. Some may dependent building services that the application will call. Not everything is equal effort it seems, yet the business wants a specific rollout, including dates and can plan that even 18 months ahead. They are so specific because there is an impact on the user community (staff, clients, customers or members of the public).

What you've got is the perfect setup for disaster born from the random bigger pieces bad news that can happen in software development.

One thing was underestimated by 50% and that is determined later rather than sooner. Should all of the following releases slip too, assuming the company did not attempt to throw bodies at it in an attempt to solve it?

One compelling answer is to change the order of releases. To the business, that could be a relief even if it requires re-planning and problems around marketing/education given the impacted staff, clients, customers or

members of the public.

The trouble is that the development teams might have to face a selective unmerge or commenting-out frenzy to support that, depending on what had merged already. Different branching models have different merge impacts and are either early or late in terms of keenness for the act of merging. That in itself is disruptive to the business, as they fear and probably witness additional delays because of the retooling and new found nerves.

If your team has institutionalized Trunk Based Development, [Feature Flags](#) and (to a lesser degree) [Branch by Abstraction](#), it is in a perfect position to reorder releases, and only have a small impact on development team throughput. Choosing Trunk Based Development, Feature Flags and branch by Abstraction could be said to be a **hedging strategy** against the costs of larger scheduling changes.

Consecutive development of consecutive releases is by far superior!

Every high throughput Extreme Programming team will tell you that finishing and shipping a release before starting work as a team on the next releasable slice of work is much better than attempting to do concurrent development of consecutive releases. Sure, some teammates (PM, BA, tech leads) are looking a couple of weeks ahead to make sure that eveything is ready for development and QA automation on a just in time basis, but the majority of the dev team will only pick up new release work as the previous one has been pushed into production.

References elsewhere

[show references](#)

Date	Type	Article

Strangler Applications

Strangulation is a mechanism by which a very large disruptive change is made in an application or service that, does not disrupt its ability to go live, even while partially complete. Martin Fowler named this practice (see references below) after the strangler vines that creep up existing trees, in order to steal sunlight at canopy level of a jungle.

The trick is to have a mechanism to route invocations of logic between the old and new solutions for the same. Say you are an Airline, and you had written your first online purchasing experience in Perl. You're now wanting to do 'Elixir' and its web framework 'Phoenix'.

Strangulation is where you would use the Apache server that you doubtless had fronting Perl, to **conditionally** route HTTP requests to Erlang/Elixir/Phoenix. Say your first completed milestone was 'Loyalty Account View/Edit' you would route based on the URLs the browser was seeking pages for. Obviously agreeing on URLs (and cookies) is key for the old Perl and new Elixr app. So is deployment in lockstep.

At some point in the strangulation, you might put Elixir in front Apache/Perl and have traffic drop through to it instead. That is the residual situation before you delete the last lines of code of Perl and snip that delegation when the strangulation is complete.

This relates a little to [Branch by Abstraction](#). Strangulation is a strategy for incompatible languages (they are not in the same process), whereas Branch

by Abstraction is where the 'from' and 'to' languages are the same (say Java -> Java), or compatible (Java -> Scala).

References elsewhere

show references

Date	Type	Article
29 Jun 2004	MartinFowler.com article	Application Strangulation
17 Jan 2006	Blog Entry	Great Architects Are Also Stranglers
14 Jul 2013	Blog Entry	Legacy Application Strangulation : Case Studies

Observed habits

No Code Freeze

Developers living in a Trunk Based Development reality, mostly do not experience variance in their days or weeks on the trunk. In particular, there is no "we're close to a release so let's freeze code", and generally there is no indication of a slowdown in proximity to a release. Sure, a couple of developers out a team might be assigned to bug-fixing closer to the release but everyone else is going to work at full speed.

Generally speaking the trunk is a place to firehose commits into, and the habits of the developers are such that everything is ready to go live. If a team is doing 12 releases a year, then a release branch that is cut on the just in time basis and is the one that is observed to be 'frozen' because of the absence of developers. Refer [branch for release](#).

Every Day is the same

Ignoring meetings, developers commit/push at the same rate regardless of the day of the week or the week of the month. This is a reinforcement of the No Code Freeze suggestion above.

Quick Reviews

Teams doing trunk based development, know that their commits/pushes

will be scrutinized by others, as soon as they have landed on the shared trunk. They are keen on bring that forward, not delaying it, so they may prefer to pair-program on code changes. Or they may ask colleagues for a code review at the time the change is submitted to be merged into the trunk.

Chasing HEAD

Trunk based development teams update/pull/sync from the shared trunk often. Many times a day in fact.

Running the build locally

Developers practicing Trunk Based Development run the build before a commit/push in order to not break the build. This one practice, for very small teams, allows them to not setup a CI server until later. If they can't push their commits to the shared trunk because someone else beat them to it, they have to do another update/sync/pull then another build then the push of the revised commit(s). "It worked on my machine" says the developer that does not want to confess to breaking the build (assuming quick reliable idempotent builds).

Powering through broken builds

OK, so because of that lazy developer, or the flaky build, or pure accident of timing (Google has a commit every 30 seconds into their monorepo - there must be quantum entangled commits on a 0.0001% basis), the trunk will be observed to be broken occasionally. There could be an automatic rollback that's about to happen, or a good old fashioned "lock the trunk" while the build-cop sorts it out. That last is particularly true in situations where batching of commits in CI builds is the reality.

A developer wanting to update/pull/sync from the shared trunk often runs the risk of encountering that statistically improbable broken build. They do not want to have the commits that broke the trunk, on their workstation if they are developing. So what they do is update/pull/sync to the last known good commit, and only go further ahead when the trunk build is officially repaired. This way they know they can stay 'green' on their workstation.

Shared Nothing

Developers, on their developer workstations, rely on a 'microcosm' setup for the application or service they are developing. They can:

- bring up the application on their workstation and play with it.
- run all unit, integration and functional tests against it locally

Shared nothing requires significant discipline to achieve. It generally means that no TCP-IP leaves the developers box, and being able to prove that by running those operations while disconnected from the network. The implementing of the wire mocking (service virtualization) of dependent tiers outside the team, is a given. The highest accomplished Trunk Based Development teams employ mocking of tiers within the same application, in order to make tests fast and stable. Technologies such as Mountebank⇗ make programming working with wire mocking easy. Tiers refers to a layer-cake view of an applications construction, of course.

With a Microcosm strategy which delivers shared nothing for a developer workstation, it is acknowledged that non-functional consistency with production has been thrown out of the window and that only functional correctness is being honored. This is only really any good for the act of development on a workstation, and the verification of that (per commit) by a Continuous Integration daemon.

Your team will need many named QA environments, and many named UAT environments. Each of those with different rules about the frequency

of deployment, and even perhaps even a temporarily reservation for different reasons. Those environments pull together **real** dependent services and integrated applications. As much as possible those environs should not have shared services.

Companies often make a classic mistake when buying software in that they (say) buy one license for prod, and another for all dev, QA and UAT, meaning the DevOps team had configured it as shared for all those environments, with a wide-ranging negative impact on productivity and quality for innumerable and sometimes subtle psychological reasons.

Common code ownership

Committing to the trunk many times a day requires a broad sense of ownership to code, and a willingness to allow developers to contribute changes to sections of an application or service that they have not previously be involved with. Allow does come with responsibilities and checks. The former is to standards, and the checks are by the CI server, and by humans who should honor to do a speedy code review. That last, for the highest performing teams, means as soon as the proposed commit is ready.

You're doing It Wrong

Merely naming a branch trunk.

Say you are using Subversion, and you accepted its default directory design, when you made a new repository. That will give you 'trunk', 'tags' and 'branches' as directory names. The mere fact that you have a branch called trunk does not mean you are doing trunk based development. "We merge branches back to trunk often" can be heard a lot in the industry, and if you are grouping multiple developers on those branches of they not deleted after a couple of days, then it is not the trunk based development branching model.

Direction of Cherry Pick on release branches

All your developers are using a trunk and they're doing the right thing re not breaking the build. Your release cadence is infrequent enough to allow you to cut a release branch on a just in time basis, and harden that in the run up to the actual release.

If you are fixing the bug on the release branch and merging it down to the trunk you are doing it wrong - although there is debate about this. There is a chance you might forget to merge it down, and then there is going to be a regression at the next release moment (fresh branch cut from trunk).

Bugs should be reproduced and fixed on the trunk, and then **cherry-**

picked to the release branch. A build should happen from that, a second confirmation that the issue has been remediate, and that deployment should go live (perhaps a point release). If you can't reproduce on the trunk (truly rare), then you've permission to go ahead and reproduce it on the release branch, fix it there, and merge back.

Merging rather than cherry-pick to/from a release branch

The developers cut a release branch because their release cadence is low, and they're hardening and certifying the release there. BUT in the days that lead up to the release, they are also doing general merges up to the release branch from the trunk. That is not right - it is like they cut the branch on the wrong day. Maybe the business people on the team are pushing too hard to make a date.

Cherry-picking every commit since the branch-cut to the branch from the trunk is the same thing of course.

Duration of 'short-lived' feature branches

The short-lived feature branch should only last a day or two, and never diverge from the trunk enough so that a merge back is problematic. After the seal of approval from code reviewers and CI daemons it should be merged back into the trunk. It should be deleted, as proof of convergence. The developer in question may then go ahead and make the next short-lived feature branch for the next story/task they're doing.

Numbers of developers on 'short-lived' feature branches

If there is more that one developer (and the developer's pairing partner) on the same short-lived feature branch, then that branch is at risk of not being short-lived. It is at risk of being more and more like a release branch under active development, and not short at all.

There is a risk too, that a developer may choose to pull changes to their workstation **from a short-lived feature branch** rather from trunk. They may thing that the code review for that short-lived feature branch is going to take too long, or they need the changes before they are ready. Unfortunately there is no way that the current generation of code portals can prevent people pulling changes from non-trunk branches.
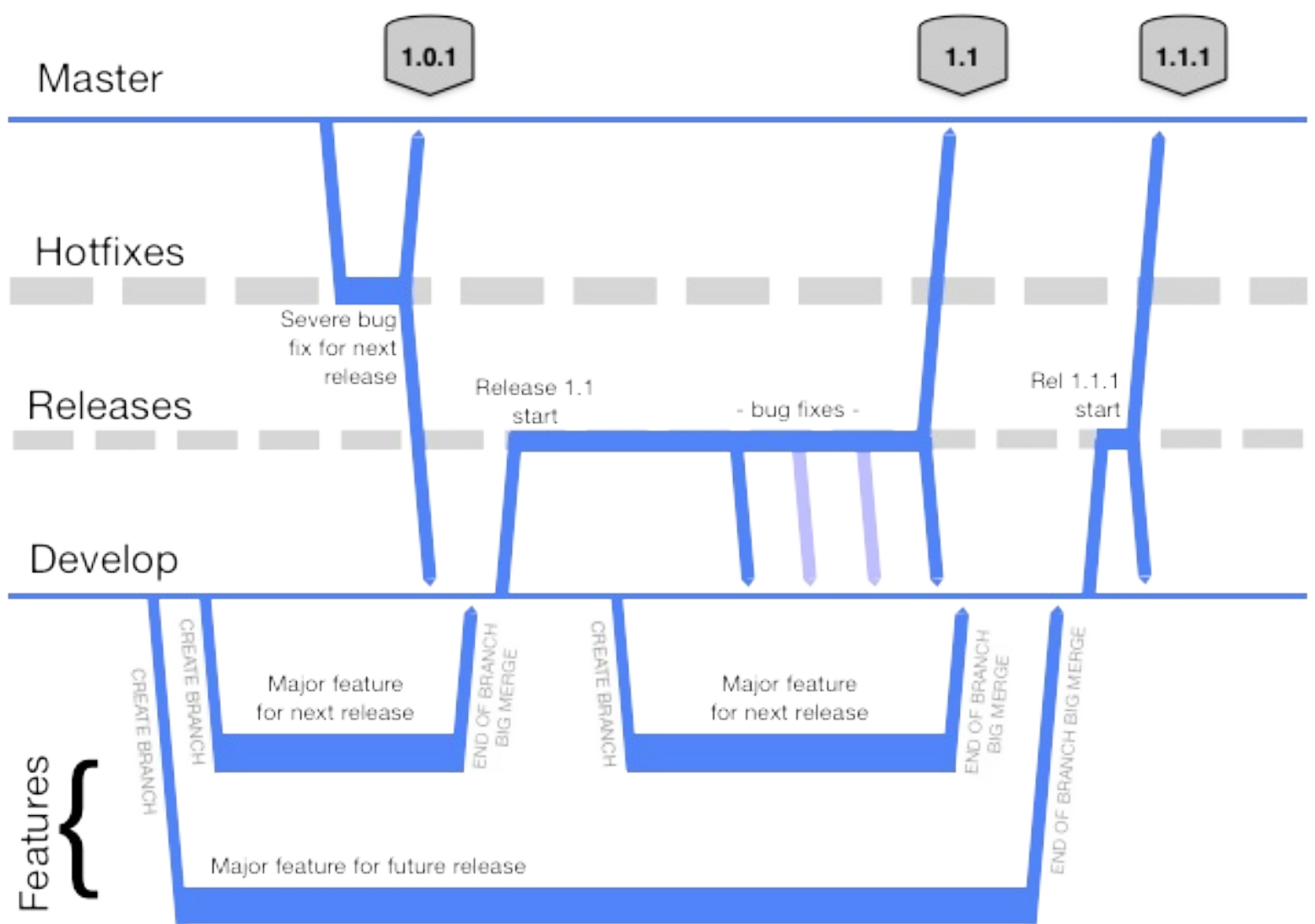
Alternative Branching Models

Branch: only when necessary, on incompatible policy, late, and instead of freeze

— Laura Wingerd & Christopher Seiwald
  (1998's High-Level SCM Best Practices white paper from Perforce)

Modern claimed high-throughput branching models

GitFlow and similar

There are plenty in the modern age that swear by this model, and feel it has plenty of room to scale with few downsides. It is a branching model that has **groups** of developers active concurrently in more than one branch (or fork).

Master   1.0.1   1.1   1.1.1

Hotfixes

Severe bug
fix for next
release

Releases   Release 1.1
start   - bug fixes -   Rel 1.1.1
start

Develop

Features {

CREATE BRANCH
CREATE BRANCH
CREATE BRANCH

Major feature
for next release

END OF BRANCH
BIG MERGE

CREATE BRANCH

Major feature
for next release

END OF BRANCH
BIG MERGE

END OF BRANCH BIG MERGE

Major feature for future release

- Diagram copied from Vincent Driessen's 2010 article on GitFlow: "A successful Git branching model"
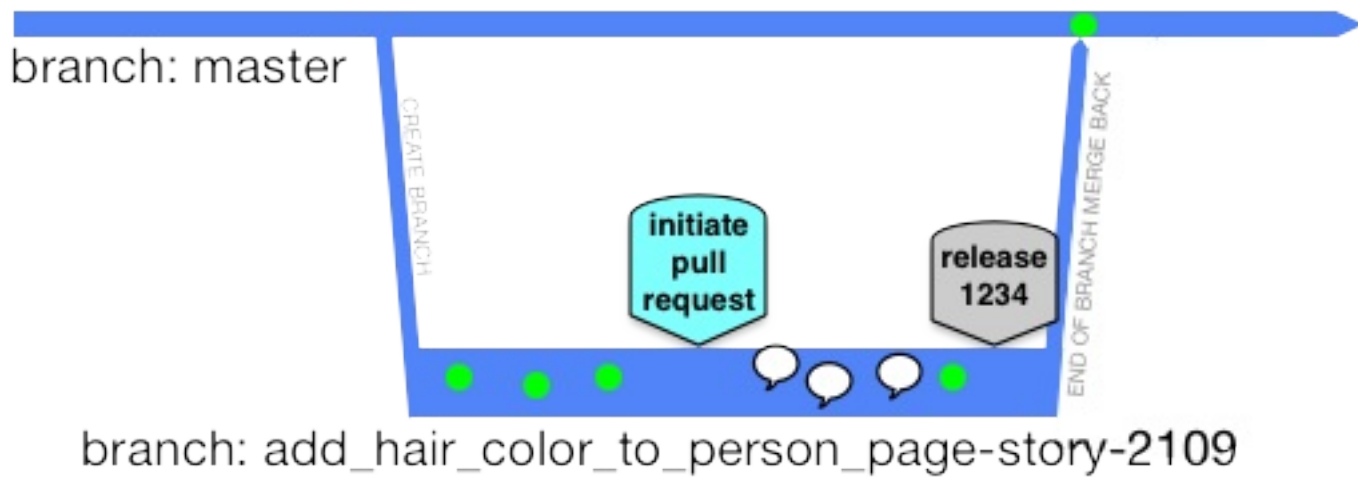
I looks like you'll not be able to do concurrent development of consecutive releases with this branching model, or the hedging that Feature Flags and Branch by Abstraction enable.
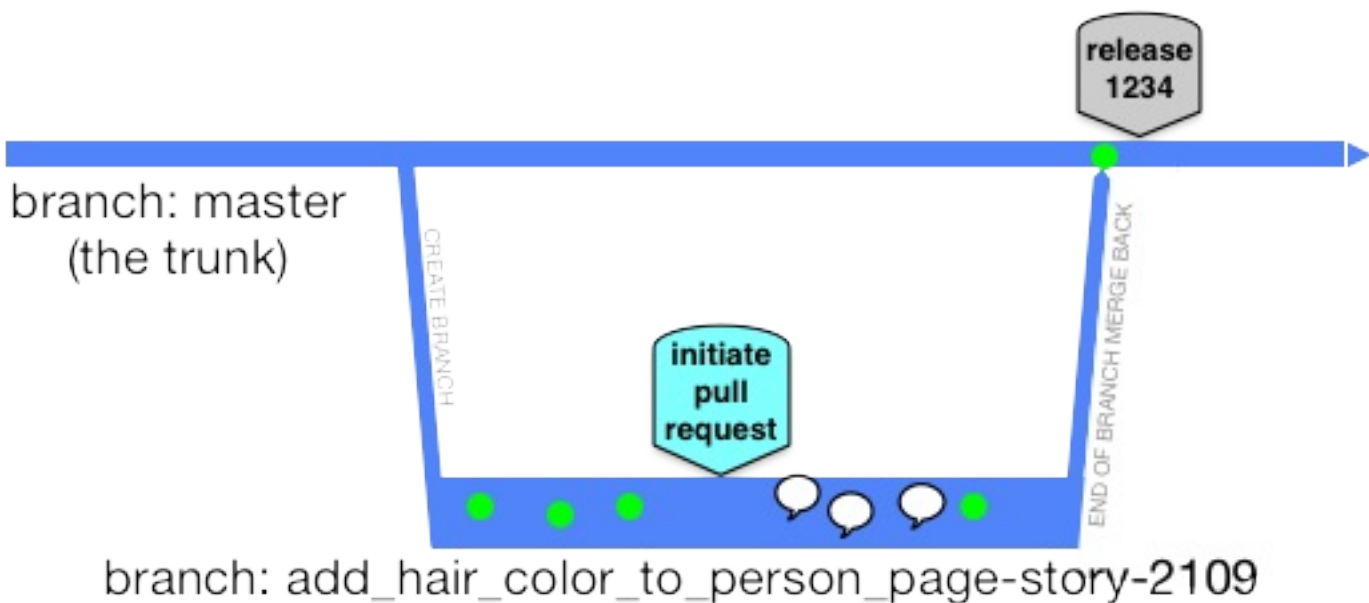
Github flow

This is sooooo close to PR-centric Trunk Based Development. Why? Because, it is a branching model that has individual developers active concurrently in more than one (short-lived) branch (or fork). Or developer pairs, rather than individuals.

The crucial difference is where the release is performed from. Whereas for

their release-from-branch step:

branch: master

initiate pull request

release 1234

branch: add_hair_color_to_person_page-story-2109

As the Github documentation portrays, review comments are part of the process. Of course they are, they are the speech bubbles in timeline above followed by another commit (presumably 132 columns end of line versus 80 prevailed). How Trunk Based Development modifies the Github Flow model:

release 1234

branch: master (the trunk)

initiate pull request

branch: add_hair_color_to_person_page-story-2109

After the dust has settled, and the short-lived feature branch has been deleted, the commits are not smushed together in a bigger one (as would be the case Subversion and Perforce), the instead zip into their respective places in the commit history, which is not as linear as we present here:

branch: master
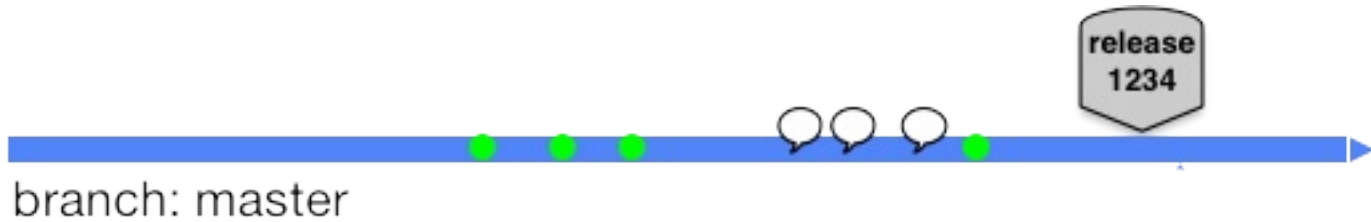
Of course if you rebase/squash your series of commits, they could land in the trunk as a single commit. Also note that the review commentary is still available after the branch is deleted, as it should be.

See the Github Flow landing page for more⧉

Legacy branching models

Mainline

Mainline is a branching model that was promoted for ClearCase implementations. It is the principal branching model that Trunk Based Development opposes. Mainline is a branch that will last forever✳. Off that, branches are formed for teams to do development work on. When that work is complete, a release may happen from that branch, and there is a **big** merge down to the mainline. On the way to the release, the branch may be frozen.
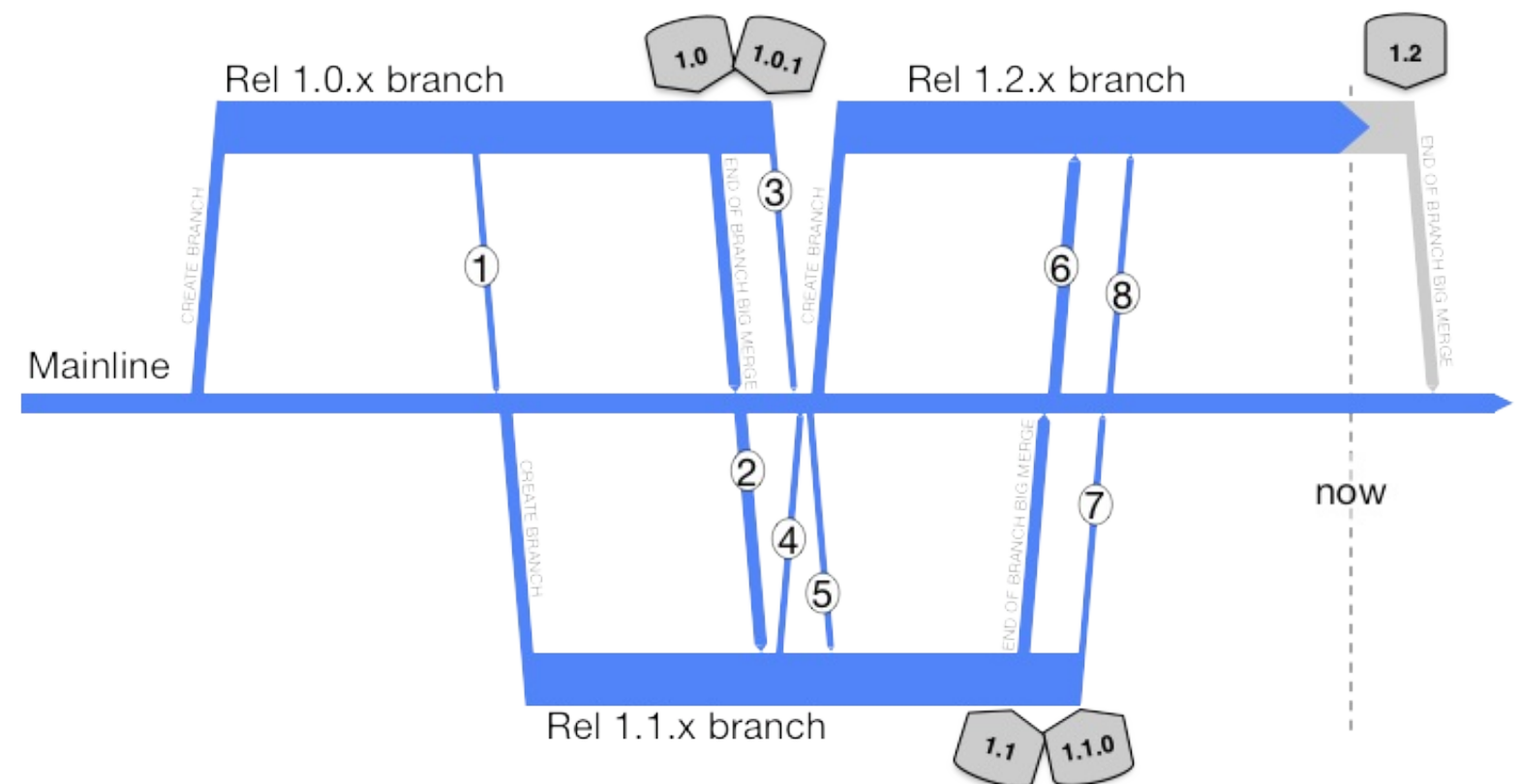
So here is the intention, with Mainline:
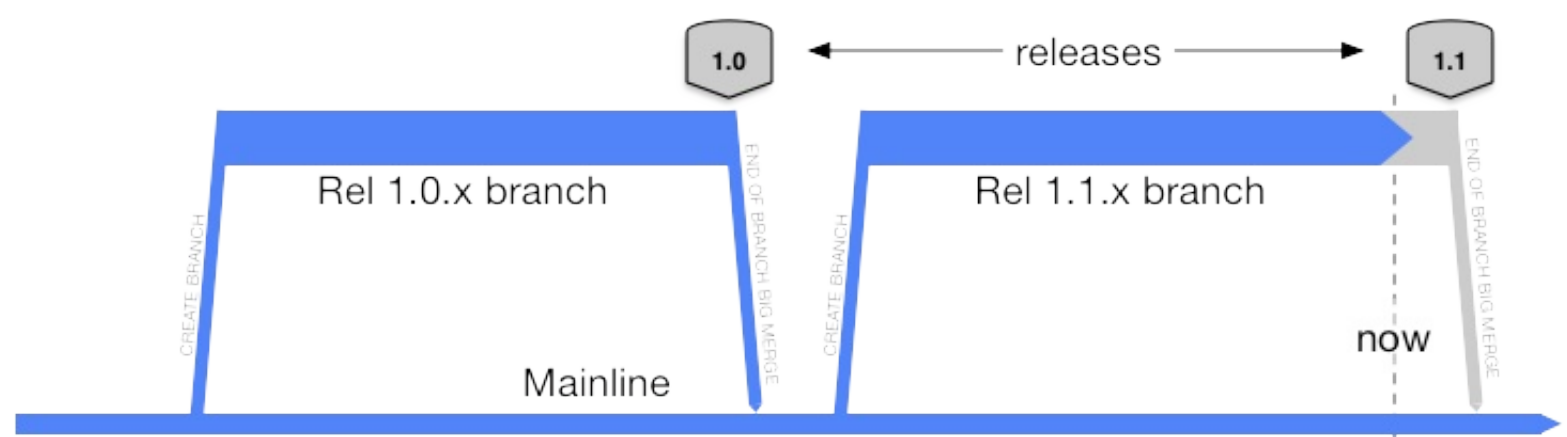
When bugs inevitably happen:
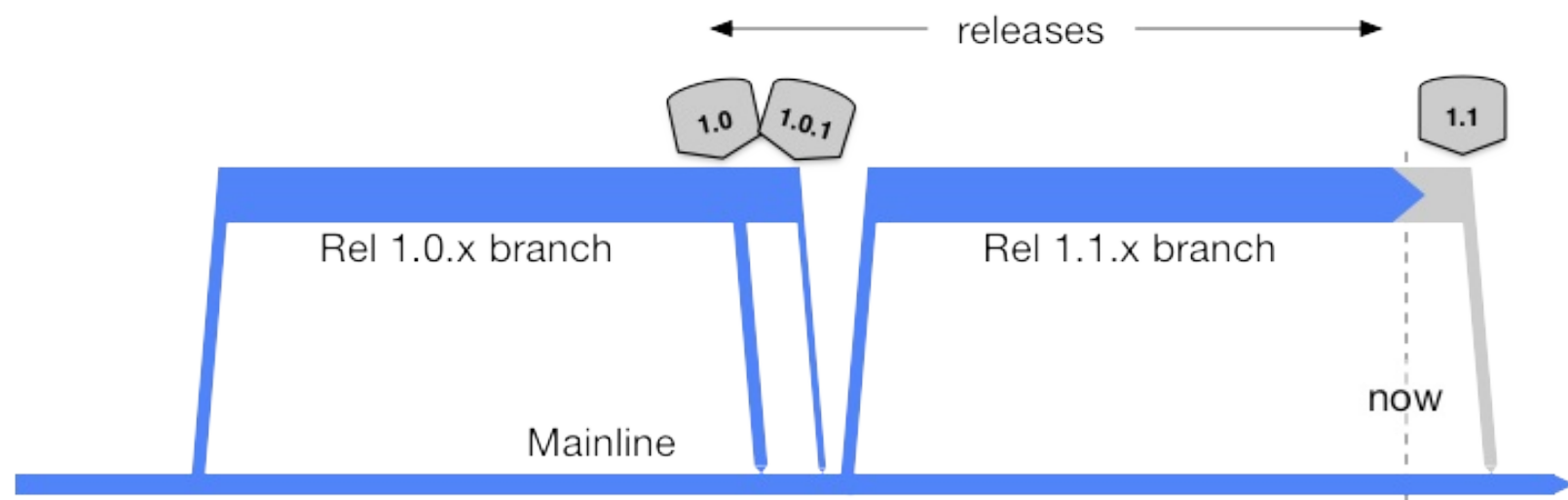
Whenever there is a bug fix there has to be a merge down to the mainline afterwards. There's no 'wrong' in this modified branch diagram, but you should be able to guess what the worst case branching/merging scenario is. In case you cannot:

Merges for the above

1. Release 1.1 team persuades the release 1.0 team to bring something

back to Mainline early (and incomplete) before they cut their branch
2. Release 1.1 team merges the release 1.0 work upon apparent completion
3. Release 1.0 team merges post-release bug fixes back to Mainline, and cross their fingers that the 1.0 branch can truly die now
4. Release 1.2 team persuades the release 1.1 team to bring something back to Mainline early (and incomplete) before they cut their branch
5. Release 1.1 team merges from Mainline, to pick up #3
6. Release 1.2 team merges the release 1.1 work upon apparent completion
7. Release 1.1 team merges post-release bug fixes back to Mainline, and cross their fingers that the 1.0 branch can truly die now
8. Release 1.2 team merges from Mainline, to pick up #7

All of these compromises versus the planned "consecutive development of consecutive releases". In many cases it is worse, particular when the numbers of developers goes up.

One key thing to note, versus Trunk Based Development, teams doing the Mainline branching model, almost never do cherry pick merges for any reason. Instead they're doing a "merge everything which is not merged already" kind of merge. Minimalistically the VCS they are using should have "merge point tracking". At the high end, that should include "record only" merges, and normal merges even after that.

＊ Companies that choose 'Mainline' wither and die, we claim, so there is no forever.
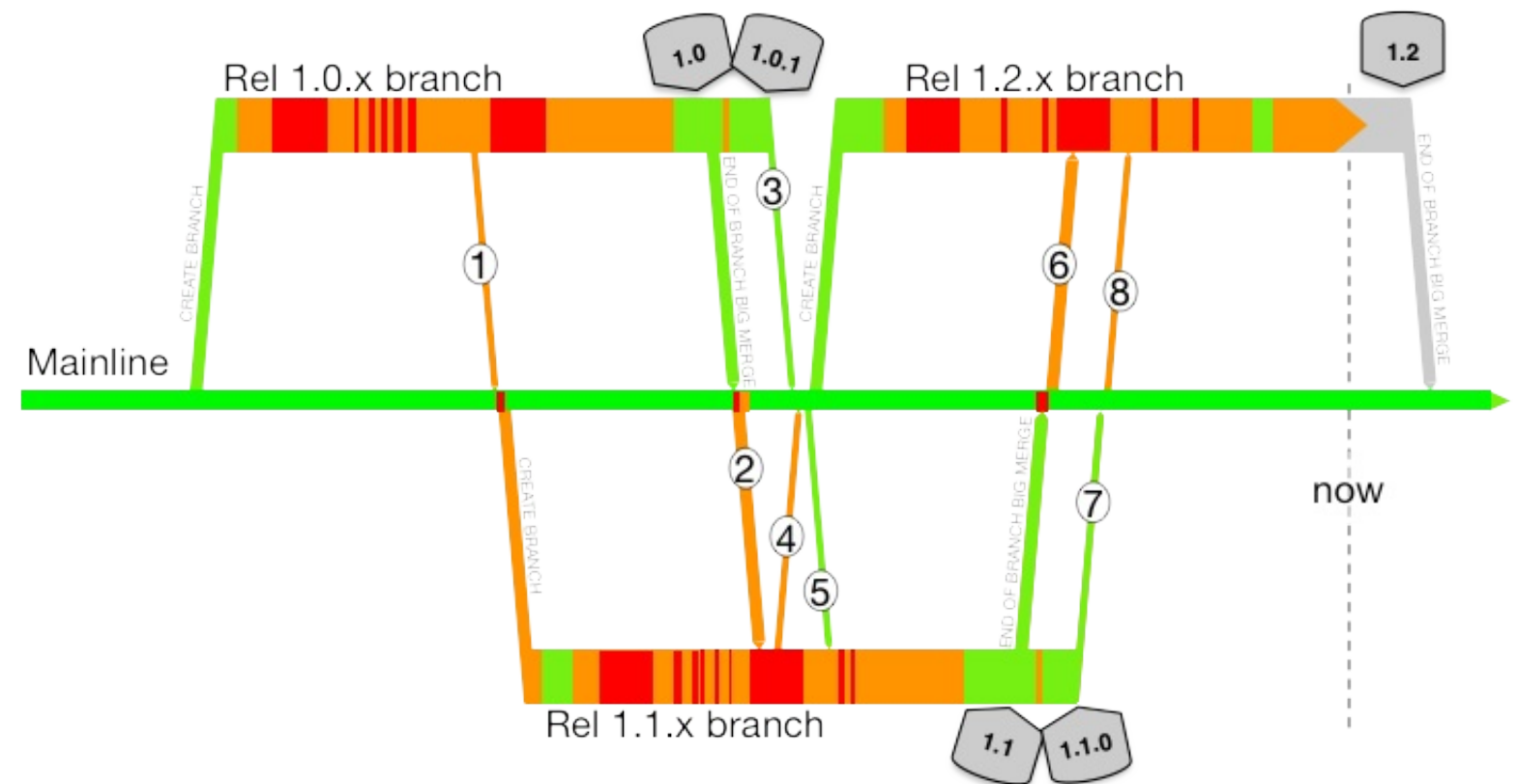
Merges

After the release the code will be merged back en masse to the mainline. Those merges may be hard and lengthy. It could be that the team **took merges from** mainline part way through the project. It could also be that the team **pushed merges to** mainline part way through the project.

How many branches?

We've just described a two branch model - the mainline and a project branch. It could be that the application in question has more that one project in flight at any one time. That would mean more than one project branch, and that creates pressure for more intermediate merges, and consequentially greater merge difficulty.
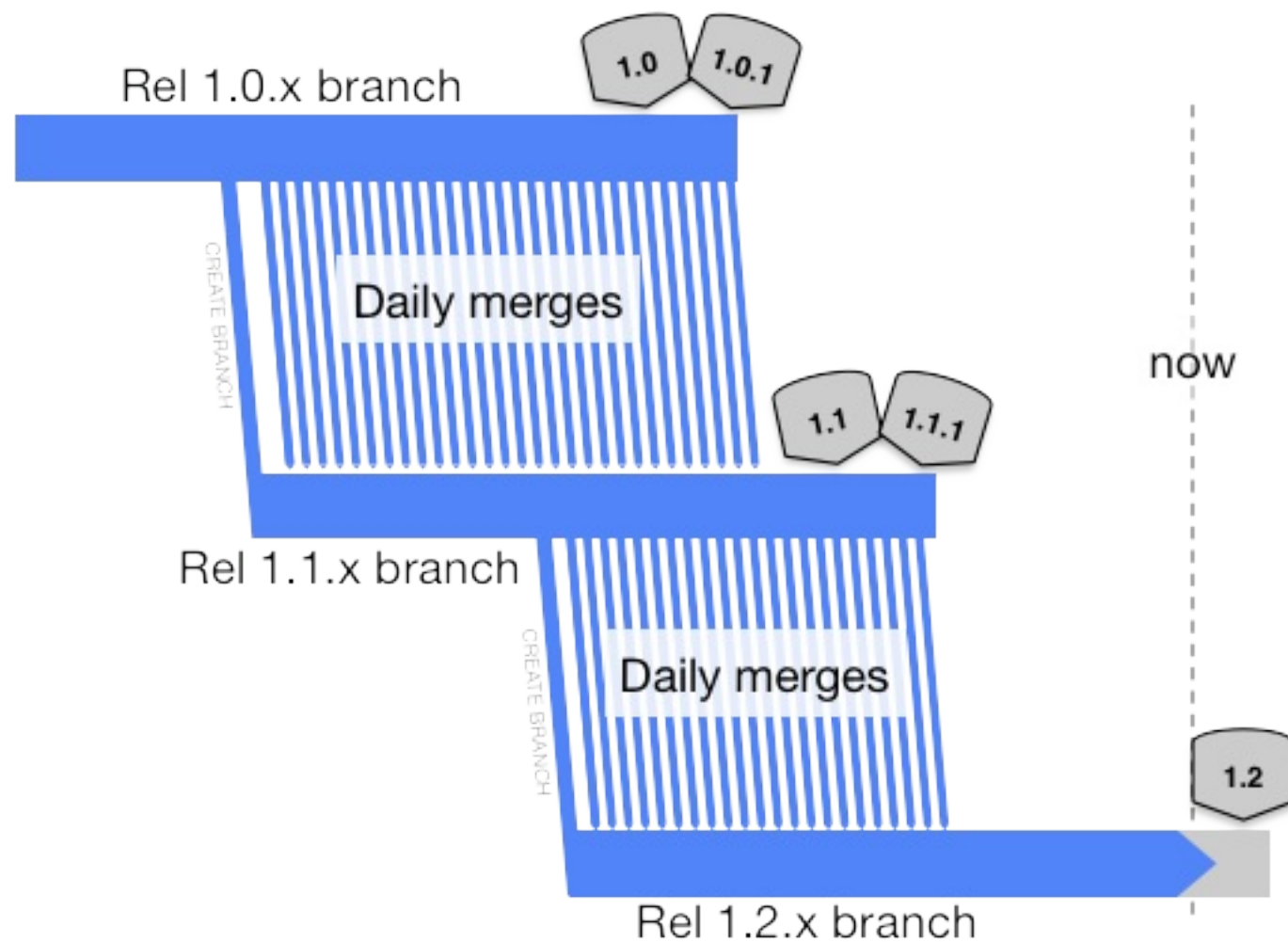
Always release ready?

Not on your life! Planned work needs to complete, with estimates guiding when that will be. Defects need to be eliminated, formal testing phases need to kick in. Here we take the first branch diagram, and overlay red and orange and green to show known build-breaks, build passes missing automated tests will not catch hidden defects, and green for could go live. At least for the worst performing with missing or ineffectual automated testing run in the CI pipelines:


Cascade

The idea is that each release has its own branch, and that each release team merges from the 'upstream' branch daily. They only do so if the CI server says the build is green for the upstream, of course.

Rel 1.0.x branch

1.0   1.0.1

CREATE BRANCH

Daily merges

now

1.1   1.1.1

Rel 1.1.x branch

CREATE BRANCH

Daily merges

1.2

Rel 1.2.x branch

Problems compound with this model, the more releases being juggled concurrently there are. An upstream butterfly, is a downstream Tsunami of unmergability. Downstream merged begin to get skipped, or abandoned. Or the merge works, but the code is not right so there is some in-branch fixing, which is not applicable to upstream. Here's the reality (breakages overlaid again):
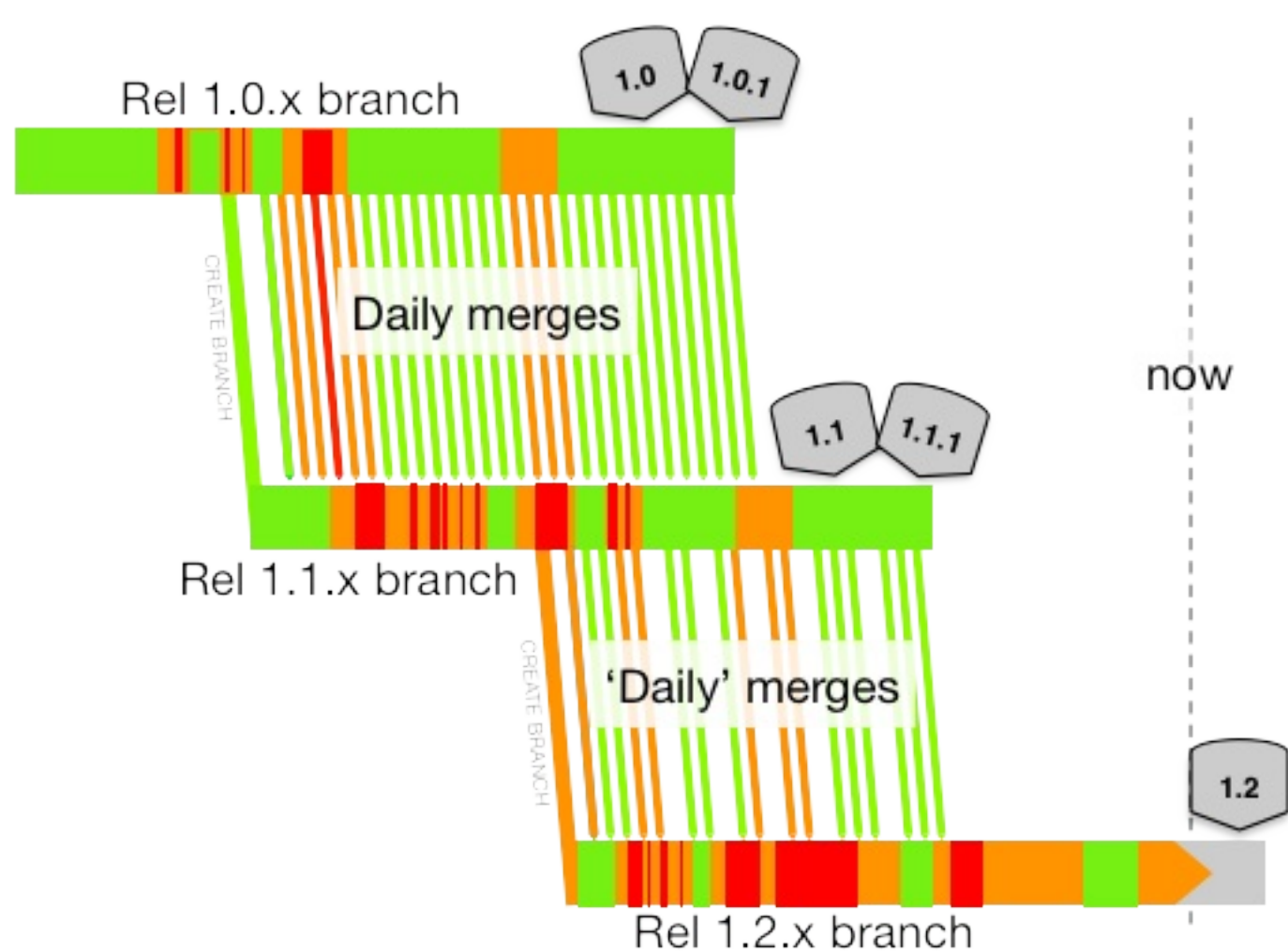
Remember, the merges are never cherry-picks in this model - they are sweeps of everything not merged yet (or upto an chosen commit number in order to make it more bite sized).

Of course only larger organizations have to worry about concurrent development of consecutive releases, and many would argue that the application is too large anyway (and that microservices is the solution).

CI (dis)proof of your branching model

Here's an idea. Configure your CI Server to focus on every branch, regardless of branching model. Specifically to do per-commit builds, and do that speculative merge analysis described in game changers.

If everything is green everywhere, then you're in a position to always be release ready, but very few teams are going to see green instead of red for

this CI server enthusiasm

References elsewhere

[show references](#)

Date	Type	Article
04 Dec 2013	Blog Entry	[What is Your Branching Model?](#)
05 Apr 2013	Blog Entry	[What is Trunk Based Development?](#)
19 Mar 2013	Blog Entry	[The Cost of Unmerge](#)
15 Oct 2015	InfoQ Interview	[More Feature Branching Means Less Continuous Integration](#)
03 May 2015	Blog Entry	[GitFlow considered harmful](#)

Monorepos

A Monorepo, is a specific trunk based development implementation where the organization in question puts its source for all applications/services/libraries/frameworks into one trunk, and forces developers to commit together in that trunk - atomically.

Google has the most famous monorepo, and they do the above AND force teams to **share code at source level instead of linking in previously built binaries**. Specifically, they have no version numbers for their own dependencies, just an implicit 'HEAD'. Third party libraries (like JUnit) will be checked into the repo with a specific version number (like 4.11), and all teams will use that version if they use it at all.

The deployment and/or release cadences for each application/service/library/frameworks will probably be different as will the team's structures, methodologies, priorities, story backlogs etc.

The name 'monorepo' is a newer name for a previously unnamed practice that is more than a decade old.

Monorepo implementations deliver a couple of principal goals:

- Acquire as many third-party and in-house dependencies as possible for a build from the **same** source-control repository/branch, and in the same update/pull/sync operation.
- Keep all teams in agreement on the versions of third-party and in-

house dependencies via lock-step upgrades.

And some secondary goals:

- Allow changes to common dependencies to via atomic commits.
- Allow the extraction of new common dependencies (from existing code) to be achieved in atomic commits.
- Force all developers to focus on the HEAD revisions of files in the trunk

Google and Facebook are the most famous organizations that rest development on a single company-wide trunk, that fits the monorepo design.

Risk of chaotic directory layout

Google's co-mingled applications and services site within highly structured and uniform source trees. A Java developer from one project team instantly recognizes the directory structure for another team's application or service. That goes across languages too. The design for the directory layout needs to be enforced globally. You can see that in the way that Buck and Bazel layout trees for production and test code. If you cannot overhaul the directory structure of your entire repository, you should not entertain a monorepo.

Third party dependencies

With the monorepo model, there is a strong desire to have third-party binaries in source-control too. You might think that it would be unmanageable for reasons of size. In terms of history, Perforce and Subversion do not mind a terabyte of history of binary files (or more), and Git performed much better when Git-LFS was created. You could still feel that the HEAD revision of thousands of fine-grained dependencies is too

much for a workstation, but that can be managed via an [expanding and contracting monorepo](#).

Note: Python, Java, C++ and other SDKs are installed the regular way on the developer workstation, and not acquired from the source-control repository/branch.

In-house dependencies

It could be that your application team depends on something that is made by colleagues from a different team. An example could be an Object Relational Mapping (ORM) library. For Monorepo teams there is a strong wish to depend on the source of that ORM technology and not a binary. There are multiple reasons for that, but the principal one is that source control update/pull/sync is the most efficient way for you to keep up with the HEAD of a library on a minute by minute basis. Thus `MyTeamsApplication` and `TheORMweDepOn` should be in your source tree in your IDE at the same time. Similarly, another team that depends on `TheORMweDepOn` should have it and `TheirApplication` checked out at the same time.

Directed graph build systems

To facilitate Monorepos, it is important to have a build system that can omit otherwise buildable things/steps that are not required for the individual developer's **current** build intention.

The general directory structure for directed graph build systems is like so:

```
root/
    prod_code/
        build_file.xml
        (source files)
        a_directory/
```

```
            build_file
            (source files)
            another_directory/
                build_file.xml
                (source files)
        yet_another_directory/
            build_file.xml
    test_code/
        build_file.xml
        (source files)
        a_directory/
            build_file
            (source files)
            another_directory/
                build_file.xml
                (source files)
        yet_another_directory/
            build_file.xml
            (source files)
```

Obviously, YAML, JSON, TOML or custom grammars are alternatives to XML, for build files.

Contrived example

Two examples:

- I want to run impacted tests locally, relating to the hair-color field I just added to the person page of `MyTeamsApplication`
- I want to run bring up `MyTeamsApplication` locally, so I can play with the hair-color field I just added to the person page

Not only do you want to omit unnecessary directories/files from your build's activities, you probably also want to omit them from your IDE.

Facebook's Buck and Google's Bazel

Google has Blaze internally. Ex-Googlers at Facebook (with newfound

friends) missed that, wrote Buck⧉ and then open-sourced it. Google then open-sourced a cut-down Blaze as Bazel⧉. These are the two (three including Blaze) are directed graph build systems that allow a large tree of sources to be speedily subset in a compile/test/make-a-binary way.

The omitting of unnecessary compile/test actions achieved by Buck and Bazel works equally well on developer workstations and in the CI infrastructure.

There is also the ability to depend on recently compiled object code of colleagues. The recently compiled object code for provable permutations of sources/dependencies, that is, and plucked from the ether (think of a LRU cache available to all machines in the TCP/IP subnet). That is in place to shorten compile times for prod and test code.

Recursive build systems

Java's Apache-Maven is the most widely used example. It's predecessor, Ant, is another. Maven more than Ant, pulls third-party binaries from 'binary repositories', caching them locally. Maven also traverses its tree in a strict depth first (then breadth) manner. Most recursive build systems can be configured to pull third party dependencies from a relative directory in the monorepo. A binary dependency cache outside of the VCS controlled working copy, is more normal.

The general directory structure for recursive build systems is like so:

```
root/
    build_file.xml
    module_one/
        build_file.xml
        src/
            (prod source directory tree)
            (test source directory tree)
        module_two/
```

```
            build_file.xml
            src/
                (prod source directory tree)
                (test source directory tree)
    module_three/
        build_file.xml
        src/
            (prod source directory tree)
            (test source directory tree)
    src/
        (prod source directory tree)
        (test source directory tree)
```

Again, YAML, JSON, TOML and custom grammars are alternatives to XML for build files.

Recursive build systems mostly have the ability to choose a type of build. For example 'mvn test' to just run tests, and not make a binary for distribution.

The diamond dependency problem

What happens when two apps need a different version of a dependency?

For in-house dependencies, where the source is in the same monorepo, then you will not have this situation, as the team that first wanted the increased functionality, performed it for all teams, keeping everyone at HEAD revision of it. The concept of version number disappears in this model.

Third party dependencies

For third-party dependencies, the same rule applies, everyone upgrades in lock-step. Problems can ensure, of course, if there are real reasons for team B to not upgrade and team A was insistent. Broken backward compatibility

is one problem.

In 2007, Google tried to upgrade their JUnit from 3.8.x to 4.x and struggled as there was a subtle backward incompatibility in a small percentage of their useages of it. The changeset became very large, and struggled to keep up with the rate developers were adding tests.

Because you are doing lock-step upgrades, you only secondarily note the version of the third-party dependencies, as you check them into source control without version numbers in the filename. I.e. JUnit goes in as `third_party/java_testing/junit.jar`.

Clash of ideologies

Above we contrasted **directed graph** and **recursive** build systems. The former are naturally compatible with expandable/contractible checkout technologies. The latter not necessarily so.

Maven

Recursive build systems like maven, have a forward declaration of modules that should be built, like so:

```
<modules>
    <module>moduleone</module>
    <module>moduletwo</module>
</modules>
```

Presently though, these build technologies do not have the ability to follow a changeable checkout that the likes of gcheckout can control.

Directories `moduleone` and `moduletwo` have to exist in order for the build to work. The idea of expandable/contractible monorepos, is that trees of buildable things are **calculated or computed** not **explicitly declared**. In

order to deliver that, you would need a feature to be added Maven like so:

```
<modules>
  <calculate/> <!--or--> <search/>
</modules>
```

Or you could "hack it" and rewrite your pom.xml files after every expansion or contraction⧉.

If you decide you do not want to do a Monorepo

Then repository separation should be **no more fine grained** than things that have separate deployment cadence.

With micro services you traditionally get exactly that: a deployable micro service in its own repository. There is no reason why hundreds of microservices could not be in the same monorepo, but the microservices community has promoted the one repo per microservice for a while now.

References elsewhere

[show references](#)

Date	Type	Article
09 Apr 2013	Blog entry	[Scaling Trunk Based Development](#)
06 May 2013	Blog entry	[Google's Scaled Trunk Based Development](#)
06 Jan 2014	Blog entry	[Googlers Subset their Trunk](#)
08 Jan 2014	Blog entry	[Google's vs Facebook's Trunk Based Development](#)
10 Apr		

2014	Blog entry	[Continuous Delivery: The price of admission..](#)
10 Oct 2014	Conference Talk	[Trunk Based Development in the Enterprise - Its Relevance and Economics](#)
18 May 2015	Blog entry	[Advantages of monolithic version control](#)
20 May 2015	Blog entry	[Turning Bazel back into Blaze for monorepo nirvana](#)
27 Jan 2017	Blog entry	[Maven In A Google Style Monorepo](#)

Monorepos

Expandable and Contractible Checkouts

As some point with a Monorepo approach to source control (especially with binary dependencies in the source tree), your checkouts could be bigger than your local workstation's hard drive. Or even if the checkout is not too big for your hard drive, then it might be too much for your IDE, and you do not want to have to abandon it for Vim/Emacs. Or maybe it isn't IDE that chokes is is something about the build that's too much locally, despite command line arguments to attempt to pare it down for a shorter elapsed time.

There is a way to intelligently expand or contract the checkout on you developer workstation, to alleviate all of the above.

Gcheckout.sh

Google's in-house DevOps uses some simple scripting to modify the checkout on the developer's workstation to omit the source files/packages that are not needed for the current intentions of the developer. This Blaze related technology is a shell command called 'gcheckout'. It can modify the mappings between the multi-gigabyte HEAD revision of company-wide trunk (monorepo) and developer's own workstation. Thus the source-control tools maintain the **smallest possible subset** of the monorepo on the developer's workstation, for them to perform their daily work. Google and

the industry refer to the general feature as 'sparse checkout'.

You can run gcheckout at any time to modify your sparse checkout to be bigger or smaller (or wholly different) for different reasons. All of those are operations on your local representation of a larger trunk.

Contrived example of use

We detailed two intentions for directed graph build systems above, using a contrived application. Here is one more:

- I now want to change `TheORMweDepOn`, because a change to `MyTeamsApplication` needs me to do that.

In Google, rather than feed into the backlog of the team that maintains `TheORMweDepOn` (which may exist as a part-time committee rather than a team), the developer in question would make the change themselves. Perhaps they had made it in the same commit as the first usage of it for `MyTeamsApplication`. In the code review cycle (Google do common code ownership), the approvers for the `TheORMweDepOn` would see all the changes together. The larger change is all accepted or rejected (to be remediated) atomically.

So our developer was working on `MyTeamsApplication`, which depended on `TheORMweDepOn` (which probably transitively depended on other things). Now that developer is going to change `TheORMweDepOn` and that impacts `TheirApplication` too. The Blaze related checkout-modifying technology performs an expansion to bring in `TheirApplication` to the developer's checkout. From that moment on, the developer doing update/pull/sync will bring down minute by minute changes to those three modules. For free, the build expands to make sure that the `TheORMweDepOn` changes do not break either of `MyTeamsApplication` or `TheirApplication`.

Contrived example of use #2

We used 'change the wheel on a car', on the [Branch By Abstraction](#) page for its contrived example. It will serve us again here. The wheel is what we want to change. The other team using 'Wheel(s)' is making a Segway thing (two wheels and self-balancing via high-torque and very responsive motors). Here's the procedure:
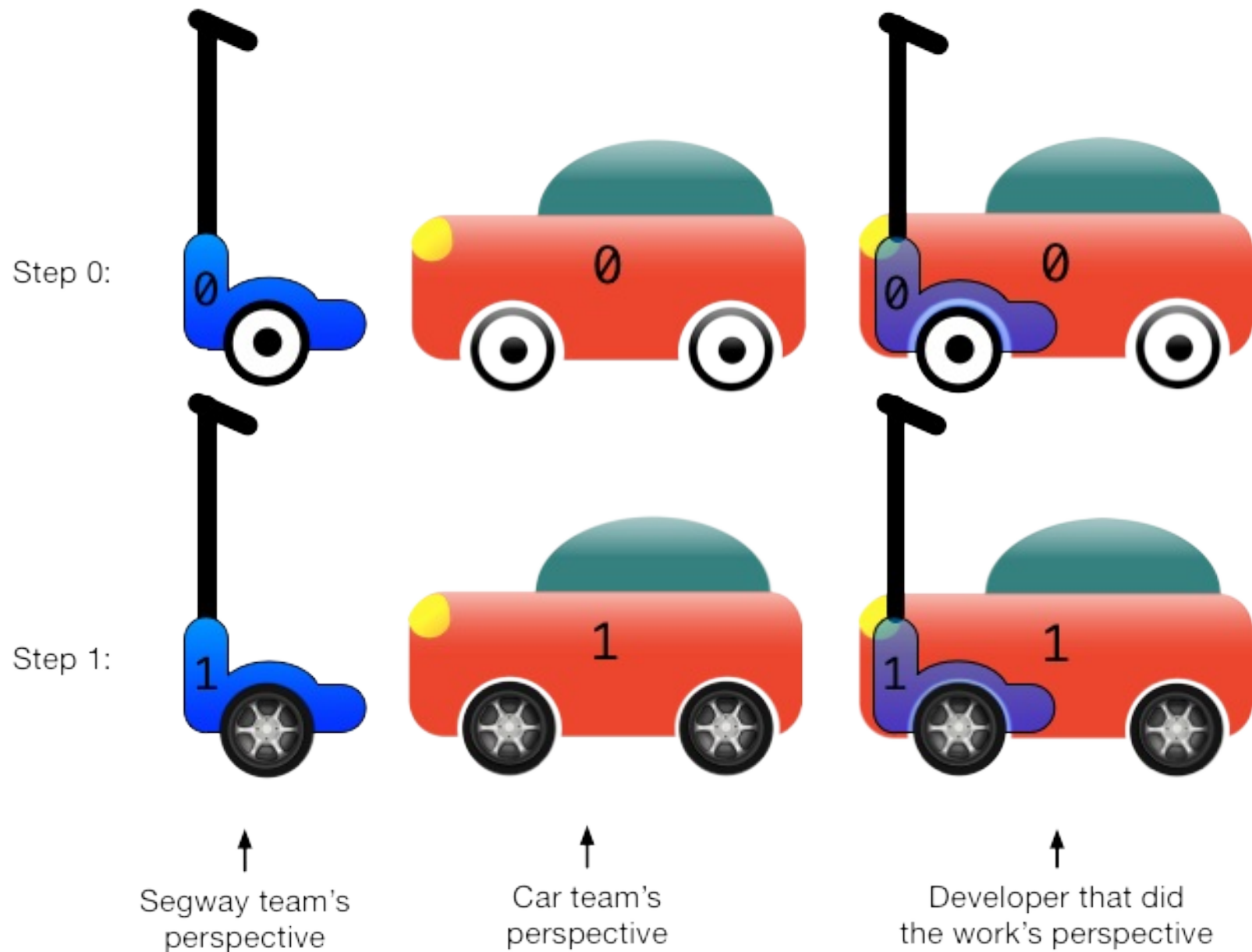
The starting position is two teams working separately, using 'Wheel' (4 for car, 2 for Segway). Without any commits happening the engineer changing 'Wheel' for everyone, runs gcheckout (or its equivalent) to modify the source in the IDE to the union of Car and Segway (and in-house dependencies). That is marked as step 0. Lets say the change is quick/easy this time (not requiring Branch By Abstraction) step 1 shows the single commit that changes wheel for everyone. After the commit/push, running

again shows the application focused team checkout - either 'Car' or 'Segway'.

Git's Sparse checkouts

Git has a 'sparse checkout' capability, which exactly facilitates this sort of thing. Subversion and Mercurial do too.
Perforce has a 'client spec' capability that is more or less the same. A team wanting to have their own gcheckout equivalent would have some scripting around sparse checkouts (or equivalent).

Using Git this way today

If you're willing to go a 'split history' maneuver on your monorepo once or twice a year, Git can do the expandable and contractible monorepo setup today.

Perforce's client-specs

Perforce has a 'client spec' (alternatively 'view') that is accessed via the client command or UI. Amongst other things, it allows a checkout to be a subset of the directories/files available within the branch. A list of globbed includes and excludes is the format. You would script this (as Google did until 2012) to have a directed graph driven expandable/contractible checkout.

PlasticSCM's cloadked.conf

As Perforce, but via 'cloaked.conf' file.

Subversion's sparse-checkouts

Subversion has a 'sparse checkout' capability. You do a series of checkout

operations at various directory levels in order to create the mapping, so is less atomic or centrally configured than the others.

Game Changers

Since the early 80's a number of things have pushed best practices **towards** Trunk Based Development, or **away** from it.

The language in use to describe such things has changed over time. Software Configuration Management (SCM) is used less today than Version Control Systems (VCS) is. A simpler still term -"Source Control" - seems to be used more recently, too.

Similarly, 'trunk' and 'branch', have not always been used as terms for controlled code lines that have a common ancestor, and are eminently (and repeatably) mergeable.

Revision Control System - RCS (1982)

Impact:

status quo

reinforces Trunk Based Development a little

RCS was a simple but 'early days' version control technology, by Walter F. Tichy.

In Tichy's 1985 paper "RCS - A System for Version Control", a trunk focused mode of use is described as a "slender branch". In section 3.1. "When are branches needed?", he says that you step away from the trunk for four reasons:

" A young revision tree is slender: It consists of only one branch, called the trunk.

As the tree ages, side branches may form. Branches are needed in the following 4 situations.

Temporary fixes, Distributed development and customer modifications, Parallel development, and Conflicting updates.

Two of those, Tichy suggests, are temporary branches and would come back to the trunk at the earliest opportunity.

Superficially, RCS allowed multi-branch parallel development, but some teams were very careful and stuck to a 'slender', or Trunk Based Development mode of use.

Note: Over time all version control systems would adopt this branch/merge language.

Concurrent_Versions_System - CVS (1990)

Impact:

status quo

reinforces Trunk Based
Development a little

A handful of scripts created in 1986 by Dick Grune, were fashioned into an initial release of CVS in 1990. For the open source community, CVS was it until Subversion came along year later. The adoption of CVS in the young open source community spurred its adoption in the enterprise too. While many branching models were possible, merging was painful and Trunk Based Development was the sensible choice.

Microsoft Secrets book (1995)

Impact:

status quo — very much reinforces Trunk Based Development

Microsoft Secrets: How the World's Most Powerful Software Company Creates Technology, Shapes Markets and Manages People (Michael Cusumano & Richard Selby, 1995)

The book was translated into 14 languages, and a bestseller, and was about practices that solidified withing Microsoft for a few years before.

There's a section in *Microsoft Secrets* dealing with Microsoft's per-developer workflow using Source Library Manager (SLM) on a one-branch model (the book does not use the words trunk or branch). SLM (AKA "slime") - an Internal Microsoft tool for source-control. That daily developer workflow was:

1. checkout (update/pull/sync or checkout afresh)
2. implement feature
3. build
4. test the feature
5. sync (update/pull)
6. merge
7. build
8. test the feature
9. smoke tests
10. check in (commit/push)
11. makes a daily build from HEAD of the shared master branch

The authors note in the book, that #10 is not always an everyday thing. And the last step, #11, isn't per developer, it is for the designated "build

master" within the team, and manual.

The book also briefly mentions Test Case Manager (TCM) and "Microsoft Test". These were tools for helping developers manage and record/edit/playback application tests at their workstations. It isn't clear if all SLM-using teams also used these, but the Excel team did (as they maintained the former at least).

These are clearly practices to support teams working in a trunk model.

Notes:

1. Steve McConnell's Rapid Development (1996) also reinforces #11 - make a daily build.

2. In 2000, ex Microsoftee and early blogger Joel Spolsky would extol the virtues of #11 in his famous "The Joel Test" posting.

Mozilla's Tinderbox (1997)

Impact:

reinforces Trunk Based
Development a little

Mozilla had a service that compiled and tested bits and pieces of their open-source offerings together. That service was Tinderbox and it debuted for the public to see in 1997. Their source organization was single-branch in the trunk style managed by CVS, and allowed individual developers to checkout and keep abreast of only the pieces they wanted/needed to. Tinderbox was the safety net that ensured everything was correct across all the whole trunk. It ran until it wouldn't scale anymore in 2014.

Perforce and ClearCase (1998)

Perforce and ClearCase bit into the corporate VCS market significantly. Both, as technologies, were open to any branching model and implementing teams chose differently. In the end, though, people's newfound willingness to experiment with multiple parallel active branches won out, and we had some dark years generally for Trunk Based Development ahead.

Microsoft installed a custom build of Perforce called "Source Depot". It took over from SLM/Slime (mentioned above). We're not sure, but Microsoft may have embraced the possibility of multiple active branches (rather than Trunk Based Development) within their SourceDepot (SD) install from that moment.

By contrast, Google installed Perforce (see below) and embraced a Trunk Based Development model with it from the outset. They rose to every scaling challenge with extra tooling around it, including more than a few actual inventions of technology and technique (see below).

Perforce's High-Level SCM Best Practices white paper

Laura Wingerd and Christopher Seiwald penned this widely read paper⬚ (presented at an SCM conference in Brussels the same year).

The paper alternates between 'trunk' and 'mainline' language, but has many valuable nuggets in it that help set a foundation for the next ten years of version-control advances.

Extreme Programming's Continuous Integration (1999)

Impact:

very much reinforces Trunk Based Development

Kent Beck⬚ published "Extreme Programming Explained" in 1999. Picked out that, amongst a bunch of practices for the influential XP methodology, is "Continuous Integration" that Kent felt was "risk reducing".

He says "Integrate and build the system many times a day, every time a task is completed", and goes on to detail a reserved workstation, that a developer pair would sidle up at the appropriate moment to prove that their code contribution was integrateable, and therefore good for teammates to depend on at that moment. That last notification was often oral at that time "build passes, gang".

He calls out a requirement for "fast integration/build/test cycles". This is key. In fact, every pro Trunk Based Development game changer listed in this page was facilitated by faster builds generally (versus a predecessor technique for the team in question). And, no, faster did not mean delete or comment out automated test execution in the build. Faster meant reduce the elapsed time to "a few minutes" (Kent again).

Kent had pioneered (with many industry luminary friends) in 1996 on the famous Chrysler Comprehensive Compensation System (C3) project. The C3 project used Smalltalk as its language, and OTI's ENVY was the version control tool used. It is important to note that today's CR-delimited text file systems are blunt instruments compared to the fine-grained directed graphs with fidelity down to class/method history of each of those. It was more like a multidimensional database with cross-cutting tags representing HEAD, or someone else's important combination of those three. It was omnipresent too - a decision made to move HEAD was instantly available without 'update' action to teammates.

Continuous Integration paper on

MartinFowler.com (2000)

Impact:

very much reinforces Trunk Based Development

Martin Fowler and Matt Foemmel⊡ wrote an influential article "Continuous Integration" in 2000⊡, calling out this one part of XP. Martin greatly updated it in 2006 ⊡.

ThoughtWorks' Cruise Control (2001)

Impact:

very much reinforces Trunk Based Development

Martin's ThoughtWorks colleagues (Alden Almagro⊡, Paul Julius⊡, Jason Yip⊡) went on to build the then-dominant "Cruise Control"⊡ starting in early 2001 (for CVS, StarTeam). This was a groundbreaking technology and very accessible to companies wanting a machine to fully verify checkins. ThoughtWorks already had success the year before with the same server design on a client project, and CruiseControl was the rewrite (from scratch) in order to make it open source. CruiseControl was quite successful in the enterprise, and was and easy decision after teams had read the Continuous Integration paper above.

Early CI servers, including CruiseControl used to have a "quiet period" to make sure they had received every last element of an intended commit. To facilitate that, only one pair of developers was allowed to checkin at a time. With CVS the other developers in the team could only do their "cvs up" when CruiseControl had given the green light, automating that "build passes, gang" oral notification above. A particular non-functional feature to note for CruiseControl was that it stored its 'pipeline' configuration in source-control. In fact, that was alongside the project's source and build file - developers could tweak CI configuration in a commit.

Apache's Gump

reinforces Trunk Based
Development a little

Apache's Gump was built on a similar timeline to CruiseControl, but focused more on the binary integration hell of interdependent Apache (and other) open-source projects. It gave an early warning of integration clashes that were already or were about to be problematic, for teams. While impressive, it did not gain traction in the enterprise. This is because enterprises were able to be more buffered from open-source library hell (and the implicit diamond dependency problem), by limiting the rate at which they upgraded their third-party binary dependencies.

Gump creator, Sam Ruby remembers:

> " The original motivation for Gump wasn't so much continuous as it was integration - in particular, integration in the large. Many projects had unit tests but would routinely make changes that would break their 'contract' and nobody would notice until well after the changes were released.

Subversion's "lightweight" branching (2000 through 2001)

Impact:

enables multiple parallel
active branches a little

Karl Fogel helped start Subversion and remembers one early goal was "CVS + atomicity". **The lack of atomicity in CVS meant that teams had to coordinate as to who was checking in presently**, and whether they

would avoid accidentally breaking the build as a result. Early CI servers (as mentioned) used to have a "quiet period" to make sure they had received every last element of an intended commit, and that was no longer needed for Subversion and its atomic commits.

In comparison to the clunky CVS, Subversion had "lightweight" branching. This made it easier to consider multiple branches active in parallel and merge the team's changes back later.

Until v1.5 in June 2008, Subversion had an inadequate "merge tracking" capability. It still has edge-case merge bugs today, like this one⬚.

Git's "lightweight" branching (2005)

In comparison to the clunky Subversion, Git had "lightweight" branching. This made it easier to consider multiple branches as active (in parallel) and merged back later. Git's merge engine was very good too. It was more able than prior merge technologies, to silently process complexity.

A critical part of Git was local branching. A developer could make multiple local branches, and even map them to the same remote branch. Say one could be a feature, part complete, and another a surprise bug fix to go back first. Or the developer could be making alternate implementations of the same complicated thing, to decide later which to push back. Git does not need a centralized server repo, but enterprise teams are going to have one anyway.

Lastly, Git came with a capability to rewrite history. Although this was a general feature, it is where the history around your local HEAD is rewritten before you push it back to the shared repository, that is of interest. Say your Agile story was four tasks and there for four local

commits, you can effectively squash those into one commit before you push it back to the shared repository. There are pros and cons to that, but having the choice is cool.

Generally, Git made it much easier to consider multiple branches as a viable team setup.

Google's internal DevOps (1998 onwards)

Impact:

would have greatly reinforced Trunk Based Development has Google not kept it secret for many years
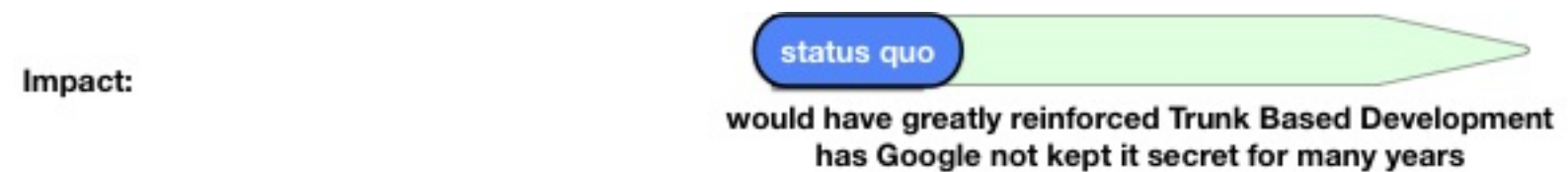
Note: Google were practicing Trunk Based Development since the beginning - Craig Silverstein (the first hire) remembers setting it up that way. Much of these were secret to Google until much later, including their recommendations for a 70:20:10 ratio for small:medium:large tests, where 'small' were sub-1ms unit tests (no threading, no I/O), 'medium' were unit tests that didn't qualify for *small* (and probably did TCP/IP headlessly to something), with 'large' being slower more costly Selenium functional tests. Pyramid like, and in the early to mid-2000's.

Home-grown CI and tooling

This was 2002 onwards, but only barely documented outside Google, this the influence is much smaller.

Google is the most famous example of using Scaled CI infrastructure to keep up with commits (one every 30 seconds on average) to a single shared trunk. Google's setup would also allow the same infrastructure to verify *proposed* commits.

Their VCS technology, at the outset, was Perforce, and it did not have an ability to effectively do CI on commits that had not yet landed in the trunk.

So Google made their own tooling for this and pending commits were plucked from developer workstations for verification (and code review - see "Mondrian" below). After its initial creation, Google's now "Google3" setup, gained a UI, Mondrian (see below) which made the results of the pre-commit CI verification very clear.

Mondrian (2006)

Impact:

very much reinforces Trunk Based Development

Tools for code-reviewers/approvers of proposed contributions to trunk were developed internally at Google in the early 2000's as a command-line tool and part of "Google 3". Things would not land in the shared trunk, until everyone agreed. Their culture was that such that reviews were speedy. Getting pending commits to the point of rejection or acceptance ("Looks Good To Me" : LGTM) was almost competitive. Some new Googlers (Nooglers) would pride themselves about taking on random code-review chores and being one of a few people that weigh into the decision moment.

The code review technology marshaled changes for proposed commits to the trunk, and stored them outside the VCS in question (in a database probably). To do that the tech would reach into the developer machine and the appropriate moment and make a tar.gz of the changes and the meta-data around them, and pull that back to the central system for global presentation. Anyone could review anything. A review was just on a commit (not a batch of commits). Therefore code review was continuous.

Reviewers could quickly bring the marshaled change down to their workstation to play with it, or use it as a basis for a counter proposal. They could put that back in review again.

In 2006, Guido van Rossum presented one of his bigger contributions -

"Mondrian" - to Googlers. Here is that 'tech talk' on YouTube:

Mondrian Code Review On The Web
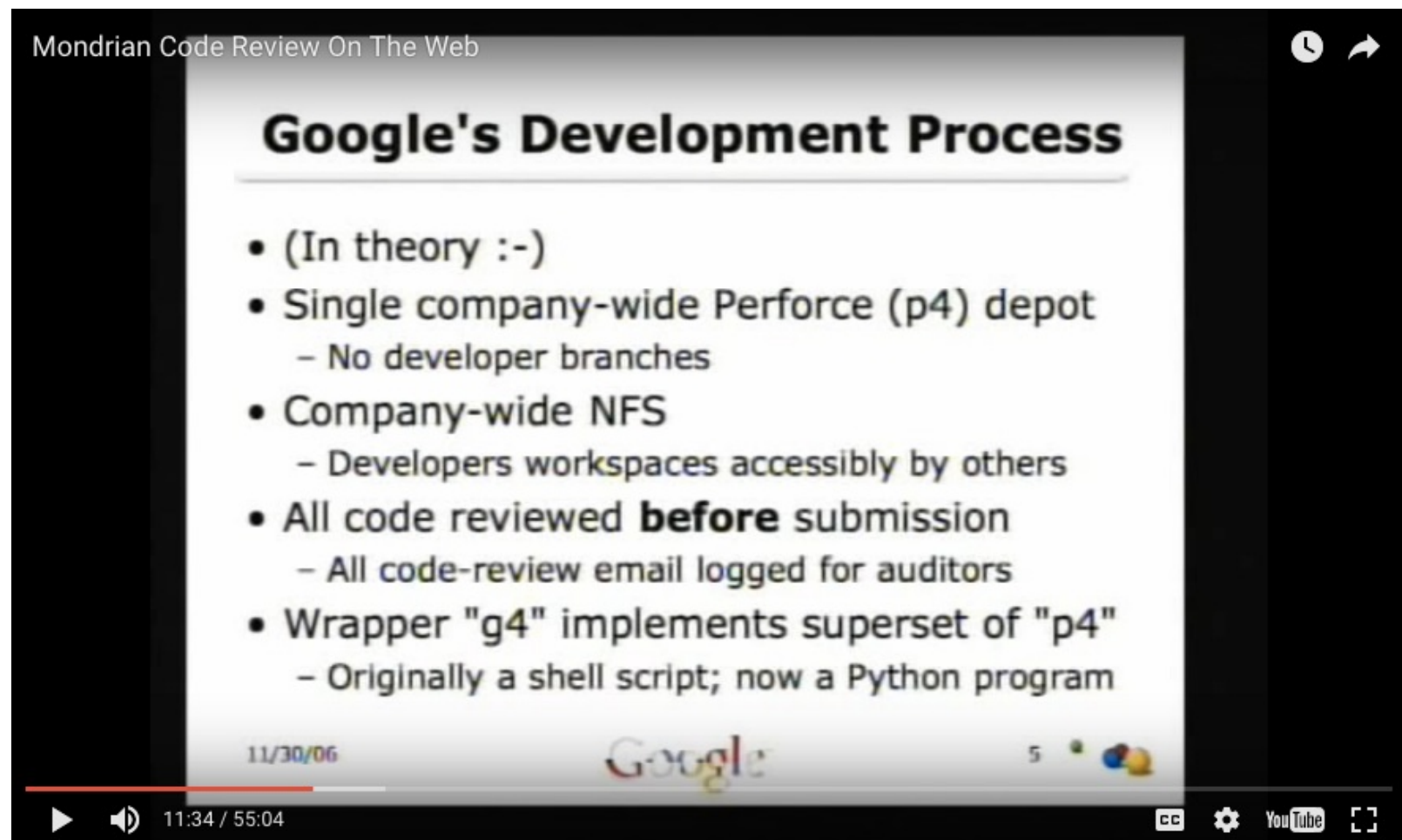
Google's Development Process

- (In theory :-)
- Single company-wide Perforce (p4) depot
  - No developer branches
- Company-wide NFS
  - Developers workspaces accessibly by others
- All code reviewed **before** submission
  - All code-review email logged for auditors
- Wrapper "g4" implements superset of "p4"
  - Originally a shell script; now a Python program

11/30/06   Google   5

11:34 / 55:04
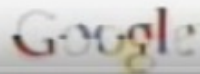
Video Available at https://youtu.be/CKjRt48rZGk

Note at the start he says XP practice "Pair-Programming" is best, and that code review helps fill the gap for situations where you cannot do it.

After Mondrian, the open source world saw Gerrit⬀ released in its image, and after that Facebookers made Phabricator⬀ and released that as open source too.

Selenium Farm (2006)

Impact:

status quo

reinforces Trunk Based Development

Google CI infrastructure was expanded to have **a second tier of elastic infrastructure**, for scaled Selenium/WebDriver testing.

This "Selenium Farm" (internal cloud) was also available to developers at their desks, who just wanted to run such tests against a stood-up version of what they were working on. Teams who had to run Firefox (etc) on their own desktop on a Friday, were able to lease one or more Firefox browsers in parallel on a Monday, and no longer lock up their developer workstations.

Other companies since have been able to deploy their own Selenium-Grid internally or leverage one of the online services for elastic Selenium testing.

Branch by Abstraction technique (2007)

Impact:

very much reinforces Trunk Based Development

Paul Hammant blogged about a 2005 ThoughtWorks client engagement in a Bank of America software development team, that used the Branch by Abstraction technique. Whereas many had previously used this technique to avoid longer version-control branches in a trunk model, this was the first time it had been detailed online and given a name (by Stacy Curl).

Github's entire platform (2008 onwards)

Impact:

enables multiple parallel
active branches
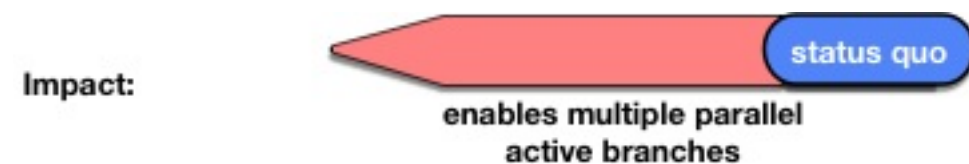
Github was launched as a portal on February 8, 2008, and features have been added steadily ever since. The initial version contained forks, which was a formal way of expressing the directionality of related DVCS repositories, and promoting a forgiveness model for unsolicited changes to source code (as opposed to the permission model that preceded it for other

portals).

Pull Requests (2008)

Github added "Pull-Requests" (PRs) on Feb 23rd, 2008⧉, while in beta, and popularized the entire practice for the industry when they came out of beta in April of that year. For source/repo platforms, and VCSs generally, this and "forking generally" was a total game changer, and commercial prospects of other companies were decided based on their ability to react to this culture change.

Code Review built in

Pull Requests came with an ability to leave code review comments for the contribution. That meant that "upstream" receivers of contributions could parry them with feedback, rather than consume them and fix them which was common previously.

No more clunky patch sets

The open-source community for one could step away from patch-sets that were donated by email (or rudimentarily). Pull-Requests changed the dynamics of open source. Now, the original creator of open source was forced to keep up with PRs because if they did not, a fork with more activity and forward momentum, might steal the community. Perhaps rightfully so.

This forced the entire VCS industry to take note, and plan equivalents. It greatly facilitated multi-branch development for teams of course.

Continuous Delivery Book (2010)

Impact:

status quo

reinforces Trunk Based Development

See [Publications - Continuous Delivery](#)

Jez Humble⬚ and Dave Farley⬚ wrote this influential book after a ThoughtWorks project in London that finished in 2007. The client was AOL - enough time has passed to share that. Specific DevOps advances were being made across the industry, but a critical aspect of this mission was that the prescribed go-live date was tight, given the known amount of work to be completed before then. Tight enough to want to compress the classic 'coding slows down, and exhaustive user acceptance testing starts' phase of a project. The team had to pull the trigger on plenty of automated steps, to allow faster feedback loops. This allowed then to have a high confidence in the quality of commits, from only minutes before. CI pipelines and delta-scripts for database table-shape migrations, in particular, were focused on.

The 2010 'Continuous Delivery' book is the bestselling result. It has been translated into three languages since, and both authors now have careers that further deliver/describe the benefits for clients. The book ties the foundational aspects of DevOps, Continuous Integration pipelines, and tight lean-inspired feedback loops together to get a broad and deep definition of how we should develop software collectively in 2010 and onwards.

Anecdotally the pipelines thinking captures a linear representation of Mike Cohn's famous "Test Pyramid" from his 2009 book, "Succeeding with Agile"⬚. See Mike's blog entry a month later too⬚, as well as Martin's recap in 2012⬚.

Dan North⧉ (Mr. BDD), Chris Read⧉ (an unsung DevOps pioneer) and Sam Newman⧉ were also key in the AOL advances. Dan North gave a deeper account of the mission at GOTO in 2014⧉ (no video sadly) and was interviewed later by InfoQ⧉.

A year or so before that mission, Sam and Dave were on a different client, UK retailer 'Dixons'. They were part of a team rolling out emergent DevOps practices, which they would get to reuse and refine on the following AOL mission. Standouts were: making the test environments have consistent behaviour with production environments (very close by not quite 'Infrastructure as Code'), QA automation technologies setup by the dev team & inducting/co-locating individual QAs with the dev team, Test Driven Development (TDD) and Acceptance Test Driven development (ATDD), a CI pipeline that included performance tests, a focus of team dynamics for high throughput.

Travis-CI's Github integration and pass/fail badges (2011)

Impact:

status quo

reinforces Trunk Based Development a little

In 2011, Travis-CI⧉ provided easy integrations into Github's platform run CI builds for Pull Requests and the general state of HEAD on any branch. This was visually indicated with "build passes" and "build fails" badges were inserted into the Github UI⧉. This made it was clear whether the proposed PR would break the build or not were it to be merged into trunk.

Microservices (2011 and 2012)

The emergence of micro-services as small buildable/deployable things that are glued together with TCP/IP (and XML/YAML/DNS configuration)

reinforced "many small repoitories" (the kinda reinforce each other really), while this can be done with any branching model, the non-trunk models probably had the mindshare. Monorepos were out completely. A possibility from monorepos, teams sharing code and source level a HEAD revision, positively laughed it. The history page of Wikipedia lists multiple people concurrently pushing the same emergent micro-service idea⬚.

Case Study: A Practical Approach To Large Scale Agile Development (2012)

Impact:

Gary Gruver, Mike Young, and Pat Fulghum wrote "A Practical Approach To Large-Scale Agile Development"⬚ to describe the multi-year transformation programme in the HP LaserJet Firmware division. In 2008, there were over 400 engineers dotted around the world working on over 10 million lines of printer firmware code in the HP LaserJet Firmware division. There were 10+ long-lived release feature branches (one for each product variant), with 1 week required for a build and 6 weeks required for manual regression testing. The engineers spent 25% of their time working on product support i.e. merging features between branches and only 5% of their time on new features.

For the next couple of years, HP committed to a huge investment in Trunk Based Development and Continuous Integration. All product variants were rearchitected as a single product on a Git master, per-variant features extracted into XML config files, all engineers worldwide were given the same virtual machine for development, and a huge multi-tier continuous build process was fully automated in-house. The results were outstanding, with build time reduced to 1 hour and manual testing replaced with a 24 hour fully automated test suite including printing test pages. 10-15 builds

could be produced a day, engineers spent 5% of their time not 25% on product support and 40% of their time not 5% of their time on new features. That is an 8x increase in productivity for 400 engineers.

PlasticSCM's semantic merge (2013)

Impact: enables multiple parallel active branches a little

Plastic's semantic diff and merge⊡ capability was launched in March 2013 ⊡. It allowed a greatly reduced diff for refactoring commits.

If merges between branches are required, and larger code changes (like refactorings) are desired, then multi-branch development is a little easier with this. However, Trunk Based Development's commits are more elegant too, because of it, and in the fullness of time, it might make on techniques like Branch by Abstraction easier, or reduce the need for it, if merge conflicts happen less often (according to source-control) for something in 2012 that would have been a definite clash.

Other source-control tools are not doing semantic diff/merge yet (2017), but they should be. Semantic merge is just as useful for trunk based development and multi-branch models. It means that there are less likely to be clash situations for commits a developer wants to do. Maybe that last vision isn't quite complete yet, but there's a direction to go in now.

Snap-CI's per-commit speculative mergeability analysis (2013)

Impact: reinforces Trunk Based Development a little

Snap-CI was the first CI service to setup pipelines for new branches in the tracked repository without a human initiating that - it did so automatically on push of the first commit into a branch. Well, at least if the branch name conforms to a given regex/prefix. That commit, and any to the branch afterwards, even preceding the Pull Request, are run through a pipeline that includes:

1. all the classic compile/unit-test/integration-test/functional-test steps of the regular build, in situ
2. a speculative merge back to the master/trunk/mainline - only into working-copy as it is for analysis only
3. step 1 **again** on that resulting merge

The speculative merge is discarded every time after #2 (if it fails to merge automatically) or after #3 (regardless) - the actual merge result is never pushed off the build server to the remote (in Git terms). It is only the "is this buildable and mergeable or not" notification that was desired from the exercise.

Although they intended this feature of Snap-CI for short-lived feature branches, it is clear now that teams should do this CI setup **regardless of branching model**. Yes, even the long-lived branching models also benefit from this, though they'll be challenged to stay 'green' the whole time, and remain eminently and automatically mergeable back to the mainline/master.

Badrinath 'Badri' Janakiraman wrote a blog entry⬚ when the feature was rolled out. The blog entry is very much worth a read, especially as Badri was product owner for Snap-CI at the time and had the epiphany to implement this feature.

Circle-CI offers the same feature now, and it is a question of time before all CI technologies do.

What is a reality in 2017 is that the high bar is every commit, **every branch**, with that speculative merge, and elastically scaled so that the notification is within seconds of pushing the commit to the shared VCS. Back in 2001 (CruiseControl) we were batching commits, we would wait a little while to allow checkins to finish (particularly for the non-atomic CVS), and humans would have to pick apart who actually broke the build.

Surely non-Trunk Based Development teams would turn on CI for every branch and soon after plan their migration to Trunk Based Development.

Google revealing their Monorepo Trunk (2016)

Impact:

very much reinforces Trunk Based Development

In none other than the Association for Computing Machinery's magazine, Googlers Rachel Potvin and Josh Levenberg share how Google arranges for 95% (25,000) of its software developers to share one trunk in "Why Google Stores Billions of Lines of Code in a Single Repository". They use a [Monorepo](#) variant of a trunk, with internal code shared at **source level**, for high-throughput, low-defect delivery of multiple applications and services. Each application/service has a release cadence chosen by the dev+biz team in question. Yes, everything works just fine.

Rachel Potvin presented on the same topic a couple of months later in "Why Google Stores Billions of Lines of Code in a Single Repository":
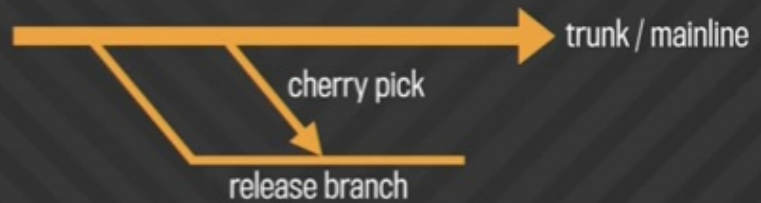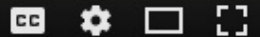
Microsoft's Git Virtual File System (2017)

Impact:

reinforces Trunk Based Development a little

This new technology deserves time and analysis before the full impact is determined. There is a suggestion from Brian Harry's blog⬈ that branch reduction was one of the motivators.

References elsewhere

show references

Date	Type	Article
13 Nov	Talk	A Practical Approach to Large Scale Agile

Publications

Books promoting Trunk Based Development

Continuous Integration (June 29, 2007)

Continuous Integration: Improving Software Quality and Reducing Risk
by Paul M. Duvall, Steve Matyas, Andrew Glover
Amazon (hardback, kindle)

Continuous Delivery (July 27, 2010)
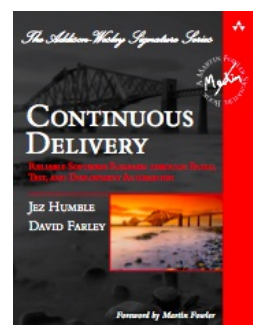
Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation
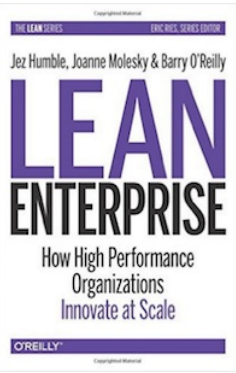by Jez Humble and Dave Farley
Amazon (hardback, kindle)
InformIT (pdf, epub, mobi)
Translations: 中文 | 日本語 | 한국말 | português

Lean Enterprise (January 3, 2015)

Lean Enterprise: How High Performance Organizations Innovate at Scale
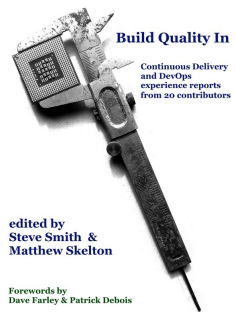by Jez Humble, Joanne Molesky and Barry O'Reilly

Amazon (hardback, kindle)
O'Reilly (pdf, epub, mobi)
Translations: 中文 | 日本語 | Deutsch | português
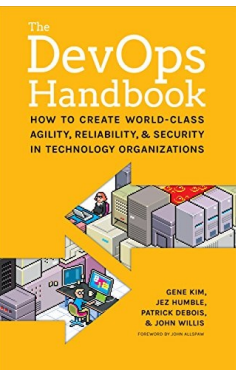
Build Quality In (February 27, 2015)

Build Quality In: Continuous Delivery and DevOps Experience Reports
by Steve Smith and Matthew Skelton
Leanpub (kindle)

DevOps Handbook (October 6, 2016)

The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations
by Gene Kim, Jez Humble, Patrick Debois, John Willis, John Allspaw
Amazon (hardback, kindle)
O'Reilly (pdf, epub, mobi)

Reports promoting Trunk Based Development

More Engineering, Less Dogma (Oct 18, 2013)

More Engineering, Less Dogma: The Path Toward Continuous Delivery Of Business Value

by Kurt Bittner and Glenn O'Donnell
[Forrester Research - link](#)

The Role of Continuous Delivery in IT and Organisational Performance (Oct 27, 2015)

The Role of Continuous Delivery in IT and Organizational Performance
by Nicole Forsgren and Jez Humble
[Proceedings of the Western Decision Sciences Institute - link](#)
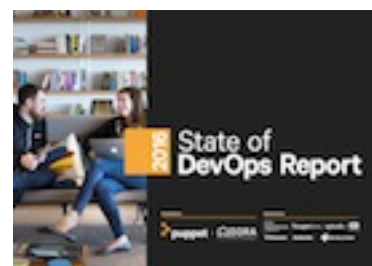
2015 State of DevOps Report

A survey of thousands of software development professionals distilled into a report that determines practices for the highest achieving organizations

"It was gratifying, though unsurprising, to find that deployment pain was predicted by whether the key continuous delivery practices had been implemented: comprehensive test and deployment automation, the use of continuous integration including trunk-based development, and version control of everything required to reproduce production environments"
by Puppet Labs

[Puppet's download form](#)

2016 State of DevOps Report

As their 2015 report, a survey of thousands of software development professionals distilled into a report that determines practices for the highest achieving organizations

"The idea that developers should work in small batches off master or trunk rather than on long-lived feature branches is still one of the most controversial ideas in the Agile canon, despite the fact it is the norm in high-performing organizations such as Google.3 Indeed, many practitioners express surprise that this practice is in fact implied by continuous integration, but it is: The clue is in the word 'integration.'"

by Puppet Labs

Puppet's download form