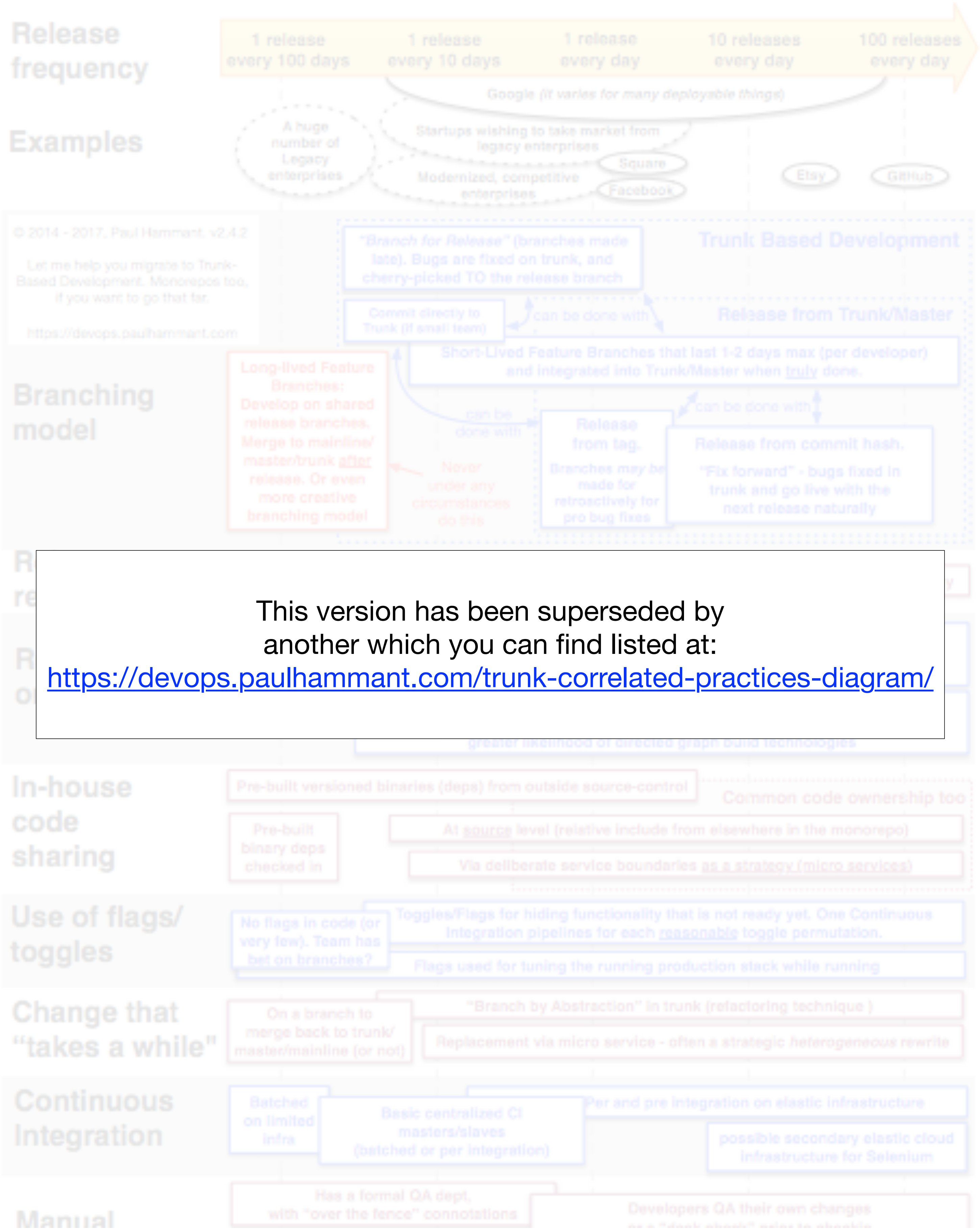

Release frequency

1 release every 100 days

1 release every 10 days

1 release every day

10 releases every day

100 releases every day

Google (it varies for many deployable things)

Examples

A huge number of Legacy enterprises

Startups wishing to take market from legacy enterprises

Square

Modernized, competitive enterprises

Facebook

Etsy

GitHub

© 2014 - 2017, Paul Hammant. v2.4.2

Let me help you migrate to Trunk-Based Development. Monorepos too, if you want to go that far.

https://devops.paulhammant.com

Trunk Based Development

"Branch for Release" (branches made late). Bugs are fixed on trunk, and cherry-picked TO the release branch

Commit directly to Trunk (if small team)

can be done with

Release from Trunk/Master

Branching model

Long-lived Feature Branches: Develop on shared release branches. Merge to mainline/master/trunk after release. Or even more creative branching model

Short-Lived Feature Branches that last 1-2 days max (per developer) and integrated into Trunk/Master when truly done.

can be done with

can be done with

Release from tag. Branches may be made for retroactively for prod bug fixes

Release from commit hash. "Fix forward" - bugs fixed in trunk and go live with the next release naturally

Never under any circumstances do this

This version has been superseded by another which you can find listed at:
https://devops.paulhammant.com/trunk-correlated-practices-diagram/

greater likelihood of directed graph build technologies

In-house code sharing

Pre-built versioned binaries (deps) from outside source-control

Common code ownership too

Pre-built binary deps checked in

At source level (relative include from elsewhere in the monorepo)

Via deliberate service boundaries as a strategy (micro services)

Use of flags/toggles

No flags in code (or very few). Team has bet on branches?

Toggles/Flags for hiding functionality that is not ready yet. One Continuous Integration pipelines for each reasonable toggle permutation.

Flags used for tuning the running production stack while running

Change that "takes a while"

On a branch to merge back to trunk/master/mainline (or not)

"Branch by Abstraction" in trunk (refactoring technique)

Replacement via micro service - often a strategic heterogeneous rewrite

Continuous Integration

Batched on limited infra

Basic centralized CI masters/slaves (batched or per integration)

Per and pre integration on elastic infrastructure

possible secondary elastic cloud infrastructure for Selenium

Manual

Has a formal QA dept, with "over the fence" connotations

Developers QA their own changes

Release frequency

1 release every 100 days
1 release every 10 days
1 release every day
10 releases every day
100 releases every day

Examples

A huge number of Legacy enterprises
Google *(it varies for many deployable things)*
Startups wishing to take market from legacy enterprises
Modernized, competitive enterprises
Square
Facebook
Etsy
GitHub

© 2014 - 2017, Paul Hammant. v2.5

Let me help you migrate to Trunk-Based Development. Monorepos too, if you want to go that far.

https://devops.paulhammant.com

Branching model

Trunk Based Development

"Branch for Release" (branches made late). Bugs are fixed on trunk, and cherry-picked TO the release branch

Release from Trunk/Master

Commit directly to Trunk (if small team)

can be done with

Short-Lived Feature Branches (SLFBs) that last 1-2 days max (per developer) and integrated into Trunk/Master when truly done. *GitHub Flow is v. close*

can be done with

can be done with

Release from tag. Branches *may be* made for retroactively for pro bug fixes

Release from commit hash. "Fix forward" - bugs fixed in trunk and go live with the next release naturally

Long-lived Feature Branches: Develop on shared branches and merge to mainline/ master/trunk after release. Or even more creative branching model. Also: Code Freezes!

Never under any circumstances do this

No code freeze!

Release prep & readiness

Only on the day that was originally planned
Sign-offs (maybe with audit trails) required. *Hello UAT*
Human coordination/planning likely
One hour notice is doable
No notice req'd, its happening anyway
No planning/coordination, software owns the release process

Repository organization

One Repo with many trunk/branches roots within it (Svn, Perforce)
Many smaller repos (Git, Mercurial); separate buildable modules; separate OR coordinated releases; nearly always recursive build technologies
OR
One trunk (Monorepo style); with separately buildable modules; with independent OR lock-step releases of those pieces; and a greater likelihood of directed graph build technologies vs recursive

In-house code sharing

Pre-built versioned binaries (deps) from outside source-control
Pre-built binary deps checked in
Common code ownership too
At source level (relative include from elsewhere in the monorepo)
Via deliberate service boundaries as a strategy (micro services)

Use of flags/ toggles

No flags in code (or very few). Team has bet on branches?
Toggles/Flags for hiding functionality that is not ready yet. One Continuous Integration pipelines for each reasonable toggle permutation.
Flags used for tuning the running production stack while running

Change that "takes a while"

On a branch to merge back to trunk/ master/mainline (or not)
"Branch by Abstraction" in trunk (refactoring technique)
Replacement via micro service - often a strategic *heterogeneous* rewrite

Continuous Integration

Batched on limited infra. Red most of the time
Basic centralized CI masters/slaves (batched or per integration)
Per and pre integration on elastic infrastructure
possible secondary elastic cloud infrastructure for Selenium
Build Cop!
Auto revert of commits of broken builds (not SLFBs)

Manual QA

Has a formal QA dept, with "over the fence" connotations
Thorough
Expedited
Release certification
Developers QA their own changes or a "desk check" prior to checkin
'Eat Your Own Dogfood' usage of app, if applicable

QA automation

some manual running of 'good' UI automation
In denial about value of "tests" not actually in source-control
Automated functional UI tests. Run sequentially from CI. Separate source repo?
Thorough yet appropriate functional UI tests that bypass login. Parallelization speeds everything up. Same source repo as prod code. Mocks/stubs and resettable data, all figure. 'Test pyramid' obeyed.

Per- Developer Environments

n/a
Dev workstation can instantiate some pieces, but shares other downstack services
Microcosm of prod stack on workstation poss. incl. *wire-mocking* techniques
Can also leverage CI infra *during builds*
…….. Infrastructure as Code …….

Pre-Prod environments

(other than the per-dev'r ones)

Handmade, slowly. Specialist knowledge and approvals req'd, costly, dreaded, intertwined shared down-stack svcs that are busy & contested
Quick, complete recreation is a habit (via IaC scripts). Ephemeral QA envs are a possibility
Few / Snowflakes
Limited?
Numerous
Truly Elastic
No more UAT or humans, bots only

Code review

One at a time, pre-integrate (Pull-Request), AKA "Continuous Review"
Some time after integration in batches
One commit a time after integrate - most likely teams committing direct to trunk
No code review
Claims to do these effectively quite often do not match reality
Note: almost universally Pair-Programming is counted as part of a review

DB rollbacks

(in case of a regretted release)

Pre-prepared delta scripts for DB rollbacks, and much practice
Case by case (un prepared), DBAs leading the effort
DB is forwards & backwards compatible by design. No downtime needed for changes

DB changes

(during deployment)

DBAs doing manual changes
Delta scripts …
… that need downtime
… that can't support multiple versions

App config per Environment

Co-located with app source
Outside of *all* management
Config fully managed and separate to binaries & app source

Talent retention

Talent leaves and recruiters do not see qualified candidates. Also: *'B players hire C players'*
Talent comes and goes
Talent accumulates, helps 'lift the bar', and maybe some become speakers or bloggers. *A players hire A players.*

Developer activity change with proximity to release

Many in dev team change activities the closer they get to a release, including getting very busy merging. Also: heroism rewarded
Small percentage of the developers change activities towards a release, including excessive cherry-pick merges
Very little change at all for any developer
No changes at all for any developer

Methodology

Acknow - ledged Waterfall
Claimed Agile, but really: *Fragile, Watergile or Mini-waterfall*
Scrum or other Agile
Agile and Continuous *Delivery* into QA/UAT only and to prod less frequently
two different CDs, remember?
On a DevOps foundation
Kanban or 'flow centric' Agile as well as Continuous *Deployment* into Prod

Bots in charge after when?

After decision to go ahead with release, and the related button press in CI
After commits are *integrated* into trunk/master (merge of pull request). All that follows is unattended by humans