Modern Git for Game Development

AAA Workflows on a Budget

#GDC2024

Hey there and welcome to "Modern Git for Game Development" – AAA Workflows on a Budget

The goal of this talk is to teach small to medium sized companies how to do (most of) what the large companies do, with regards to Version Control and Continuous Integration, at a fraction of the cost.

It's me!

Indie Dev

Uni Lecturer

EA Melbourne

PlaySide

...Independent!

MARCH 18-22, 2024 #GDC2024

But first, a little about me – Vikram!

1. I started my games career as an engine and tools programmer at a small indie studio in Melbourne

2. Taught at a university for a couple of years, getting better at communication and transferring knowledge

3. Worked on engine tech at EA for about 5 years

4. Since then, I moved to PlaySide, I was Senior Lead Engineer and 2IC of Engineering, working on improving the Engineering Department as a whole, especially processes such as version control

5. And now, I’m getting ready to start my next role!

Overview

1. Set a Target
2. Contextualise
3. Path to Improvement
4. Rolling it out

MARCH 18-22, 2024 #GDC2024

For context, this talk is going to be an anonymised amalgamation of places that I've worked at and projects I've worked on.

Now, given that the goal of the talk is to teach how to achieve AAA workflows on a budget

I've split this talk up into four parts:

1. Firstly, we're going to have a look

at what the best in the world do and use that to set a target for what we want to achieve

2. Secondly, I'll introduce a hypothetical studio – a studio that is rapidly growing and hasn't settled on best practices yet – it might sound familiar to places that you've been, or currently are at!

3. Then, I'm going to tell a story of how that studio improved its workflows - achieving something that I think is quite close to the target we set

4. And finally, I'm going to tell you what I would do differently if I was to do it again (including on my personal projects at home)!

And before get stuck in, I'm going to state that the **scale** that I'm talking about is *tried and tested* at a team size of 30-40 developers working on an Unreal Engine game.

But with a few more tweaks that I'll talk

about before the end, my aim is to at least raise that by an order of magnitude.

So hopefully you’re in the right room! Let’s get started →

Setting a Target

What does the infrastructure look like the best companies in the world?

#GDC2024

So, what **does** version control look like in the AAA sphere?

Huge Repositories

AAA companies have huge repos.

They regularly have many Terabytes of data for single projects, and they might all be stored in a single shared repository.

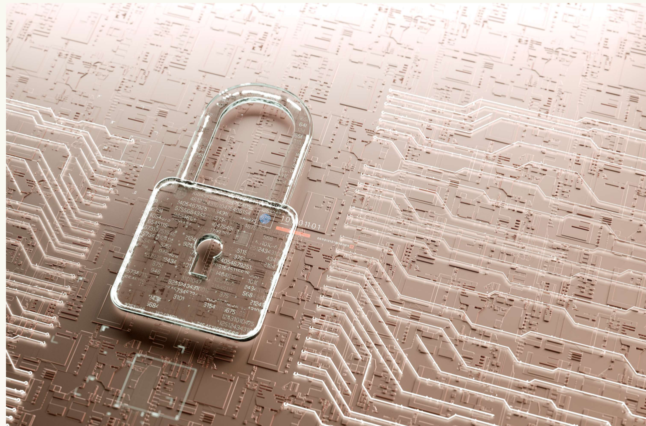
It's not impossible for a depot to get to a Petabyte or larger.

And - these repos are globally synchronised to allow for around the

clock development, so downtime can’t occur

Because of the size of the repositories – individual contributors need the ability to pull only select elements to do their work

Binary File Management



These repos are full of binary files!

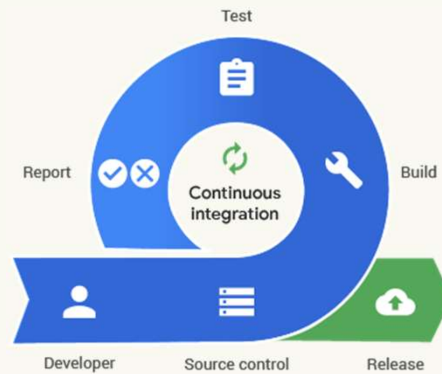
Naturally – game development deals with these more than any other kind of software engineering.

These files, by their nature, can’t be merged.

To prevent work being lost, these assets must be lockable – once someone has checked out a file, nobody else can edit it.

This is, of course, in addition to proper asset breakdown and separation of concerns.

Continuous Integration



MARCH 18-22, 2024 #GDC2024

Of course, all the best studios have some form of Continuous Integration!

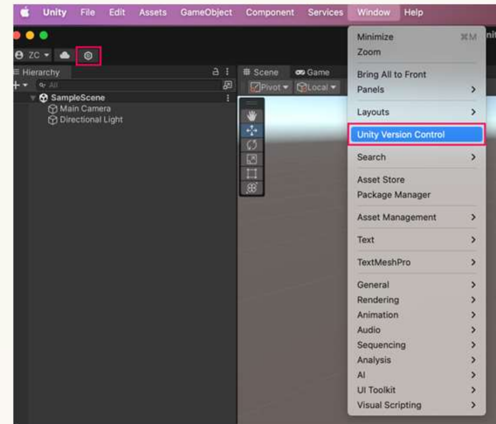
Each developer's changes are pre-integrated, tested, and validated at the maximal frequency possible - continuously

This can involve branching, isolating changes through, for example, feature toggles, merge automation, detecting merge conflicts or test

failure, and finally, ensuring that all valid changes are integrated – available for other developers to build atop of

And then for all such integrated changes to go into a release – via continuous delivery, if not continuous deployment – especially if you’re looking at a live service title

Seamless Tooling



MARCH 18-22, 2024 #GDC2024

And finally, wherever possible, studios integrate their tooling with their source control.

You want to maximise the time that each developer spends in their creation software, as opposed to their administration software – so AAA studios ensure each individual contributor has the information and interactions they need within their content creation software, whether

that’s in-engine or another DCC.

At the very least – you need to show what files are locked by other users, and to automatically lock files which the user is attempting to edit.

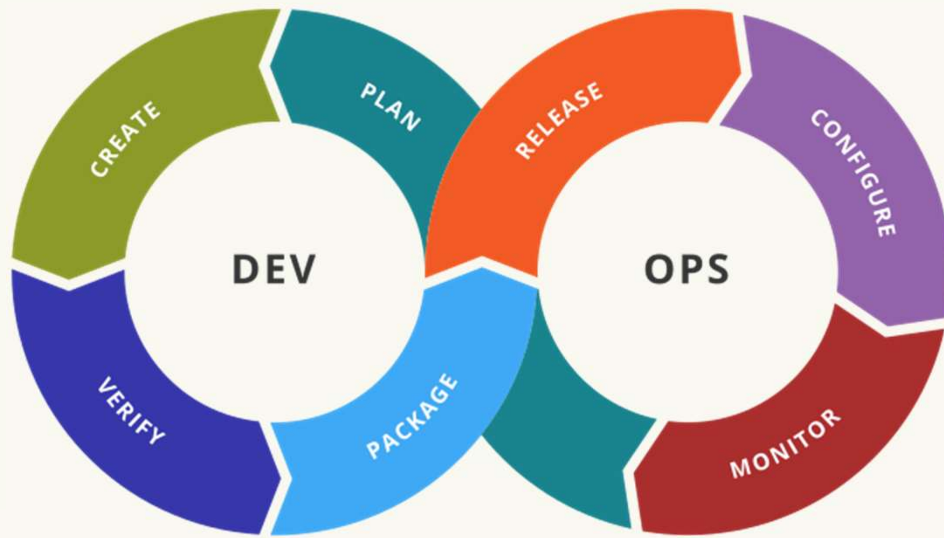
PERFORCE

MARCH 18-22, 2024 #GDC2024

Most AAA studios that I know of use Perforce for Source Control.

If you look at Epic Games, they develop Unreal Engine using Perforce – and it shows! To get tools and integrations such as Unreal Game Sync, it is perforce *only*

And if you're a small studio looking at the **cost** of setting up perforce… well… one thing to consider is that:



The AAA studios have entire teams dedicated to DevOps!

Their job is to enable everything we've just spoken about, and more

- These teams keep P4 servers online and administrated 24/7
- They managing edge server de-syncs,
- Failed build rollbacks,

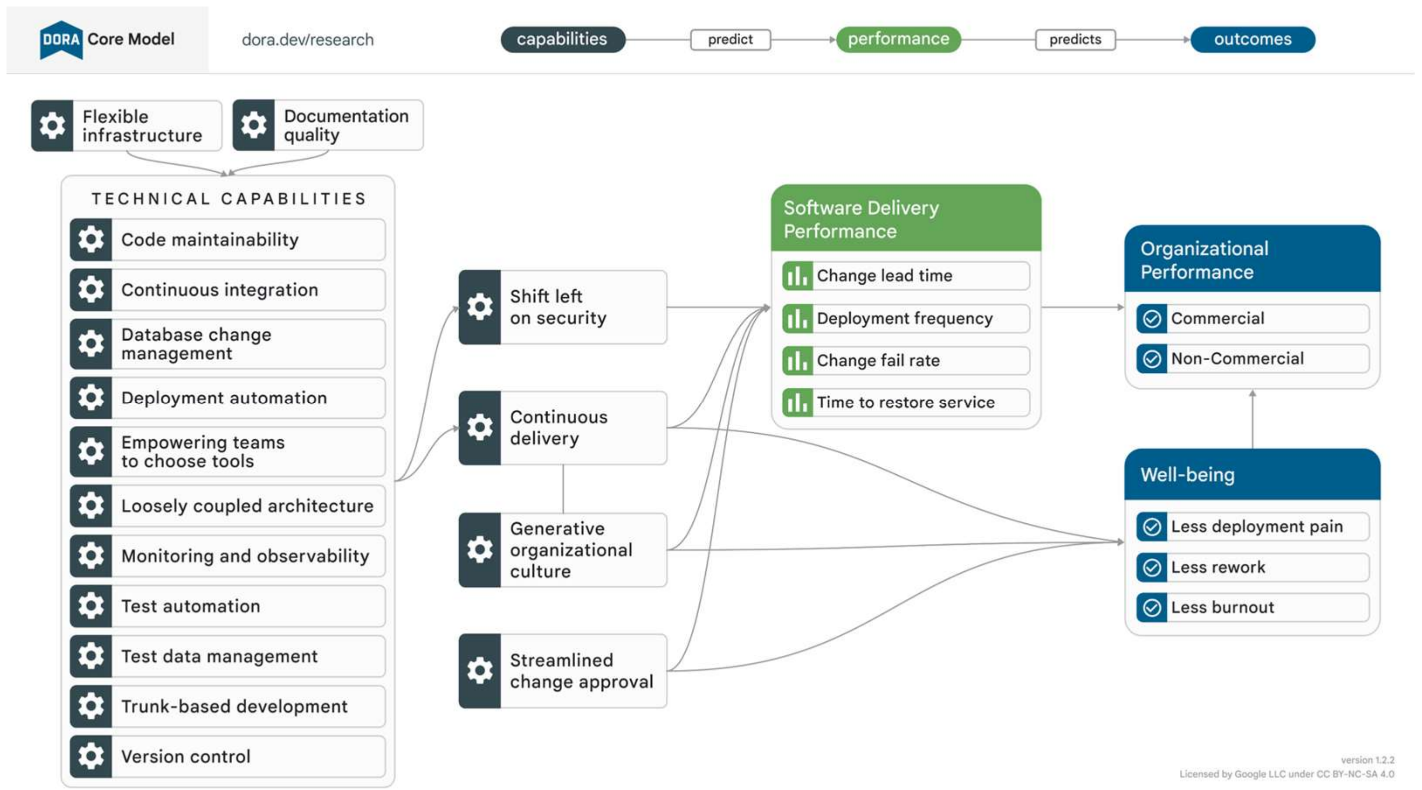
- Supporting testing infrastructure, building integrations and tooling, etc.

Even if you only need a local time zone coverage, this can be quite expensive to maintain, these people are specialists

But it’s something you can’t afford to **not** have – once your company starts to scale up!

And arguably, the return on investment is so great, that it’s worth having even if you’re a 2-person indie!

That’s a lofty goal!



I'd be remiss to neglect what's outside game development.

I'm only going to **very** briefly mention DORA, Google's Dev Ops Research and Assessment program, which you can check out at dora.dev

Almost everything that I'm talking about today is just fragments from the DORA Core Model, applied to game development

And they have reams of research on how this can benefit individuals' well-

being as well as organisational performance!

MARCH 18-22, 2024
SAN FRANCISCO, CA

Providing Context

Introducing “Tiny Turtle Studios”

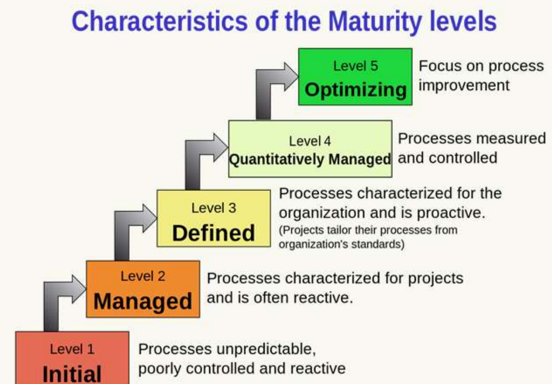
#GDC2024

Let’s talk about our hypothetical game studio – before we swoop in and fix things up.

Thanks to letsmakeagame.net for their video game company name generator – we’re going to call our studio “Tiny Turtle”

Tiny Turtle Studios

- Several projects
- Each working differently
- Using git differently
- Some Jenkins setup
- A real *scale up* problem



MARCH 18-22, 2024 #GDC2024

Tiny Turtle Studios is *scaling up*!

It’s just signed a couple of lucrative contracts and is scaling up its workforce to meet those requirements, as well as invest some of that money into its own IP!

1. Right now, Tiny Turtle has a handful of projects

Mostly mobile projects, some work for hire, and starting work on its own new IP – a AA PC game.

Most projects are small, but we'll focus on the PC game with around 30 developers.

2. Each project is set up differently to suit its own needs

Mobile projects are using Unity, 3rd party engines when doing work for hire, and unreal engine for its flagship title.

They all have different processes and pipelines!

3. …including version control!

There is no standard methodology – although most of the projects are hosted on an in-house git server.

Some are using push to trunk, some are using feature branches, some are using git-flow

4. And continuous integration is… in its infancy

No Infrastructure-As-Code – no continuous delivery, and release builds are manually triggered.

5. These are classic growth issues.

At this point of your business (**Capability Maturity Model Integration** level 1 or 2), you **must** drive standardisation

Hyper Lawnmower on the High Seas

- Cool custom tech
- Early Access
- Company's largest project
- Struggling to deliver

Let's talk about that flagship project!

1) Let's call it "Hyper Lawnmower on the High Seas" (once again, thanks to letsmakeagame.net)

2) It's an Unreal Engine title, and the Tiny Turtle team is doing some very cool things with it.

3) They've got some brilliant team members who have twisted the

engine in ways it wasn't designed – and created something unique!

4) At this point, it's in Early Access – and generating a lot of praise!

5) This is the largest project the company has ever tried, and it's got the team to match, about 30 developers altogether, including over a dozen engineers.

6) However, the team has promised a lot of features on its roadmap, and its struggling to deliver…

Let's look at that…

HLHS – Development Issues

- Two major features in development
- Intermittent merges between two dev branches
- Releases cut, and then developed on for weeks
- Custom binary data format
- Engineers constantly breaking tools for Artists/Designers

MARCH 18-22, 2024 #GDC2024

As we join the team,

1. The Hyper Lawnmower team is working on two major features. Let’s call these features UGC (Custom Maps) and Twitch Plays integration.

These features kept breaking each other, so currently they are working in two development lines.

2. They need to ensure they don’t

diverge too much, so they merge between the two branches.

They usually merge from Twitch into UGC, fix up any conflicts, and then merge back.

Of course, the merge is painful, only being done every few weeks, and often issues are missed.

3. When preparing for a release, they branch off of UGC, and submit any fixes or final content changes into that branch.

The release takes weeks to finalise, and when it's finished there is another painful merge – once into UGC and another into Twitch.

And let's make things a bit more complicated – because of the cool custom tech we mentioned before…

4. we're going to use a custom data format, with custom tooling, for the

custom maps feature.

5. And naturally, whenever you develop your own tools, they tend to be a bit more fragile.

Sadly, the constant merging between UGC and Twitch Plays keeps breaking the tooling in subtle ways that engineers don’t immediately notice.

However, the artists are definitely noticing.

Immediately, we need to fix this. Two major features, being developed like this, cannot function effectively.

So, let’s assume we have the ability to focus the team on just one feature at a time, in order to *ship*.

We’re going to get the team to work on UGC first, even though both features have been partially implemented so far.

That frees us up to look at the processes being used.

HLHS – git Setup

- git-flow…ish
- Poor review culture
- Basic CI setup
- Not using LFS
- Completely the opposite of our target!

MARCH 18-22, 2024 #GDC2024

The way the Hyper Lawnmower team is using git right now is pretty rudimentary.

1. The branching strategy is a mess, even after removing the dual-dev line problem

Designers, artists, and some engineers commit straight to trunk

Other engineers are using feature branches, but they are often developed for weeks before merging

The release branch is created in a git-flow way, with releases being developed against and hardened before going out

2. If a feature branch is used, it’s merged through a pull request

But review approval is optional.

And large features are too hard to review in detail, so reviewers often just rubber stamp them.

3. There’s a basic CI setup which pushes release builds to steam – but it’s triggered manually

QA is currently reliant on a release branch to be set up, and built, before they get a testable build

4. And, arguably worst of all – the team isn’t using git LFS – so the .git folder history is growing massively!

5. So, suffice to say, this is all completely the opposite of our target workflow!

It’s no wonder that development is grinding to a halt.

The Tiny Turtle Studios `git` Team

- Bring the Leads together, set a mission
- Research the current state git workflows
- Plan what **new** projects will look like
- But we need to test it practically, beforehand… Right?

MARCH 18-22, 2024 #GDC2024

So, if you've just joined this company, you're in a senior position, you're relatively new and have a chance to make a difference, what are you going to do?

Rather than just fixing one project, let's *standardise*.

At this point, you *could* consider switching to Perforce – but let's say that Tiny Turtle's budget is… a bit constricted right now, and leadership explicitly stated the desire to

minimise new costs.

1. Grab some of the experienced people in the company who are achieving good things on other projects, put them in a room, and set a mission to standardise the version control workflow across the company.

2. But! Don’t just assume you know what’s best. Do some research. Ask what each team is doing, what are their pain points – and then look outside of Tiny Turtle, and even outside of Games, to see what people are doing

3. Once you’ve gathered your information, you can form a plan for what an ideal workflow should look like – create a standard, write up some documentation, and make a template repository with a dummy project in it, with all of the git settings and pre-

commit hooks set up.

4. But of course, we should test this before committing to it… right? So, let’s test it on our largest project – which has the most to gain – Hyper Lawnmower on the High Seas!

Now, if you’ve been away from git for a while, or don’t keep up to date with modern git practices, you might be interested to learn that →

By Vincent Driessen
on Tuesday, January 05, 2010

Note of reflection (March 5, 2020)

This model was conceived in 2010, now more than 10 years ago, and not very long after Git itself came into being. In those 10 years, git-flow (the branching model laid out in this article) has become hugely popular in many a software team to the point where people have started treating it like a standard of sorts — but unfortunately also as a dogma or panacea.

During those 10 years, Git itself has taken the world by a storm, and the most popular type of software that is being developed with Git is shifting more towards web apps — at least in my filter bubble. Web apps are typically continuously delivered, not rolled back, and you don't have to support multiple versions of the software running in the wild.

This is not the class of software that I had in mind when I wrote the blog post 10 years ago. If your team is doing continuous delivery of software, I would suggest to adopt a much simpler workflow (like GitHub flow) instead of trying to shoehorn git-flow into your team.

If, however, you are building software that is explicitly versioned, or if you need to support multiple versions of your software in the wild, then git-flow may still be as good of a fit to your team as it has been to people in the last 10 years. In that case, please read on.

To conclude, always remember that panaceas don't exist. Consider your own context. Don't be hating. Decide for yourself.

MARCH 18-22, 2024 #GDC2024

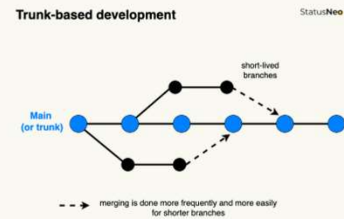
git-flow – the git workflow that was previously the top dog, was no longer king of the hill!

Even the original author of gitflow has since reflected on it and said it’s too complicated for how we need to make software nowadays

---

➔ https://nvie.com/posts/a-successful-git-branching-model/

Trunk-Based Development



Trunk-based development is a version control management practice where developers merge small, frequent updates to a core "trunk" or main branch. It's a common practice among DevOps teams and part of the DevOps lifecycle since it streamlines merging and integration phases.

In fact, trunk-based development is a required practice of CI/CD.

https://www.atlassian.com/continuous-delivery/continuous-integration/trunk-based-development

1. Instead, the modern thought in the area is to use "Trunk Based Development".
2. This is a workflow that is used **extensively** outside of the games industry, which predominately uses git in this way – including companies such as Google, Facebook, and Amazon – so we know that it is battle tested and scales well

3. There are a few flavours of it, and

my personal preference for a team like the Hyper Lawnmower one, the flavour we'll be discussing today, is called "trunk-based development *at scale*".

Of course, if you're considering trunk-based development, have a look at what flavours are out there!

There's a website that provides a great primer on this methodology and what differentiates good and bad implementations at trunkbaseddevelopment.com

So, given the state of Tiny Turtle Studios, and Hyper Lawnmower on the High Seas' issues… how do we upgrade an existing project, mid project, to use a completely new workflow? →

The Path to Improvement

An incremental journey

#GDC2024

Well, there are some obvious pain points - we clearly have a lot to improve

The first step, of course, is identifying potential solutions and getting alignment with the wider team, as we've done so far.

But then →

The Path to Improvement

1. Git LFS
2. File Locking
3. Rebase-Centric workflow
4. Managing Change
5. Branching Strategy

The next steps, or at least the next *potential* steps, of how one might go about getting better, might look something like this:

1. Firstly, I’m going to talk about one of the most loved and hated features of git for game development – Git LFS
2. Then, I’m going to introduce git’s “new” killer feature, file locking (originally experimental in 2016,

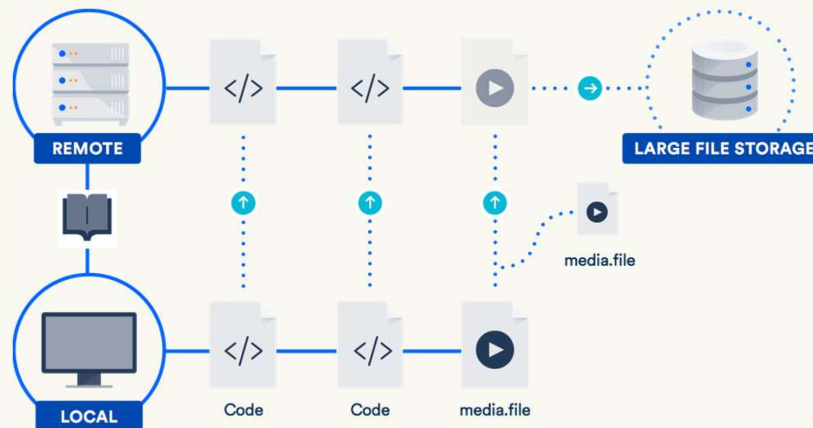
released in 2020), and cover how it works

3. But, of course, that itself will introduce a few problems that **force** a workflow change, so we’ll cover why and how you can switch to a rebase model

4. And the moment you hear rebase, it’s scares people, so we’ll talk about how to manage that kind of change with your team

5. And finally, we’ll review what kind of branching strategy you end up with, including a bunch of benefits to release management

1. Git LFS – How it works



MARCH 18-22, 2024 #GDC2024

Let’s start with Git LFS, and why and how it’s set up

Traditionally, git stores the state of every single file in every single commit, from the first commit in your repository until the current HEAD.

For text files, like code, it’s just a set of cumulative diffs – that works great!

However, binary files don’t have diffs. This causes git to store a **copy** of every single **version** of the binary file.

On your local machine… ouch.

Git LFS is a system, whereby, instead you store *pointers* of large files – moving the *contents* of the large files into its own storage on your git server – and those pointers are dereferenced (aka “*smudged”*) only for whatever large file pointers you have in your current working copy.

This used to be a somewhat manual painful process, and a lot of people have told me about how their team have run into smudging issues in the past, however in recent years, this has improved greatly, especially with modern git clients, and you shouldn’t have to care about how git lfs works.

1. Git LFS - Migration

MARCH 18-22, 2024 #GDC2024

HOWEVER, if you have a large project which is already using git but not LFS…

You need to migrate your repository. This is a **headache**.

That means going back to every commit which edited a (now) binary file and rewrite history, so that the binary “diff” was instead a pointer to LFS.

Of course, this re-writes your repo’s

history, so your whole team will need to nuke and re-clone.

This process is a pain, and if you MUST do it, there *are* guides online.

It’s better to avoid this, I strongly recommend that you set up .gitattributes properly at the start of the project.

There are a million and one templates you can look up online as a base for your projects in Unity, Unreal, C++, or anything else.

You can also set up pre-commit hooks to catch any files that are committed above a given file size!

This can help catch any potential LFS-able files that you missed in your .gitattributes and prevent this headache as well.

1. Git LFS - Scale

But in the end, if you have git lfs, everyone's local repository is now down to a manageable size, and the server can be the one place to handle a larger scale.

I've seen this solution be self-hosted on a relatively simple NAS and handle a very large number of active projects – over 20 – including archival of several more, older projects.

This includes a game repo and an art source repo per project – with some art source repos exceeding 1tb in active size, with more in LFS storage.

As far as I know, all git hosting providers support LFS with a reasonable limit, but for larger games (or unreal engine games), you'll probably want to self-host a git server using gitlab, or look into azure devops (which has unlimited LFS storage).

But ultimately, with git lfs, you should be able to scale your git repository to as large as you need!

The folks at Anchorpoint just published a blog about this exact topic, where they show repo speed with a 1tb demo repo.

(https://www.anchorpoint.app/blog/scaling-git-to-1tb-of-files-with-gitlab-and-anchorpoint-using-git-lfs)

As a side note: I’ve been asked a few times if teams should use submodules to manage larger git repos. My current thoughts are *generally* no.

Some duplication is usually a reasonable price to pay to avoid the complexity overhead.

2. File Locking – Why?

Okay, so let's discuss File Locking!

As discussed before, Hyper Lawnmower on the High Seas is now focussing on developing their UGC system to enable Custom Maps.

This, of course, heavily leans into using the custom binary data format

As we're shipping a few example maps, it has a lot of **contention** between our artists and designers who are trying to do their job.

What is more, the custom data format doesn’t have any way to create prefabs or blueprints, or any other nice collaboration features.

We can’t afford for conflicts to cause work to get lost.

So, fundamentally, we need artists and designers to know when it’s safe to work on a particular map file, or when it’s not!

Right now, designers and artists are using a google sheet, manually handled by a producer, to know who is working on which file and when.

This hurts the engineer inside me, and I’m sure you feel the same, so let’s introduce file locking to automate this!

2. File Locking – How it works

- Add the lockable attribute
- View locks: git lfs locks
- Check out: git lfs lock <file>
- Check in: git lfs unlock <file> [--force]

https://vikram.codes/git-file-locking

MARCH 18-22, 2024 #GDC2024

If you haven’t heard of git lfs file locking, you’re not alone!

Hands up if you **have** heard about it before this presentation?

Now, git lfs locking docs are sorely lacking, so I’ve put some docs together myself and put them up on vikram.codes/git-file-locking

But the high-level, command line

focussed summary is this:

1) Firstly, you need to add the lockable attribute to your .gitattributes, this tells LFS which files we want to be lockable.

You can select only certain file types, or folders. For Hyper Lawnmower let’s enable this for our custom binary format only.

From there, file locking is somewhat like perforce – lockable files are readonly by default!

And it only has 3 commands that you need to know.

2) The first is to see who has what files locked – using git lfs locks

3) The second is to acquire a lock, setting the file to be editable on that user’s machine, using git lfs lock <file>

The user works on the file as normal,

finishing with the file being merged or integrated into the main development line

4) And finally, you unlock the file, when you’re done with your changes (of course, if you have elevated privileges, you can –force the unlock in case someone accidentally left a file locked).

Sounds pretty great, right?

There is one big difference to perforce though…

2. File Locking - Global

MARCH 18-22, 2024 #GDC2024 GDC

Git lfs file locks are **global** by file path

This means, if you are using multiple development lines, feature branches, release branches, or any kind of branching structure, a file can only be edited on one branch at a time.

And the implications of **that** is that you **cannot merge a locked file** into any other branch.

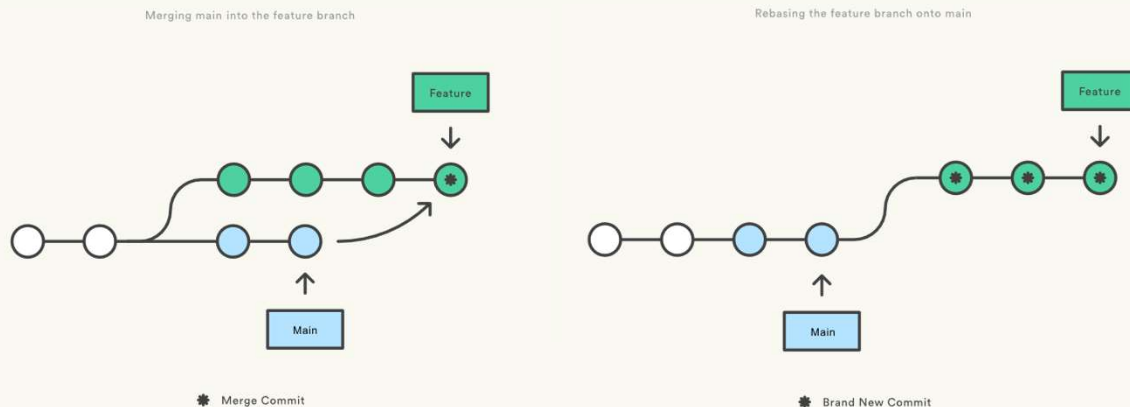
So, if your git workflow includes using *git merge* – as Hyper Lawnmower on the High Seas does – you'll quickly run into problems.

Let's say that we have an update to a map file in the main branch, and we want to get that change into our feature branch – but another developer has locked the map to edit it on *their* feature branch… well, git won't let us merge the latest changes into our branch!

So now you can't get the latest version of a file onto a branch, if someone has that file locked.

That's a problem… →

3. Rebase - Updating Feature Branches



MARCH 18-22, 2024 #GDC2024

So, we need to avoid merges… well, fortunately, git has a model for this – **Rebase**.

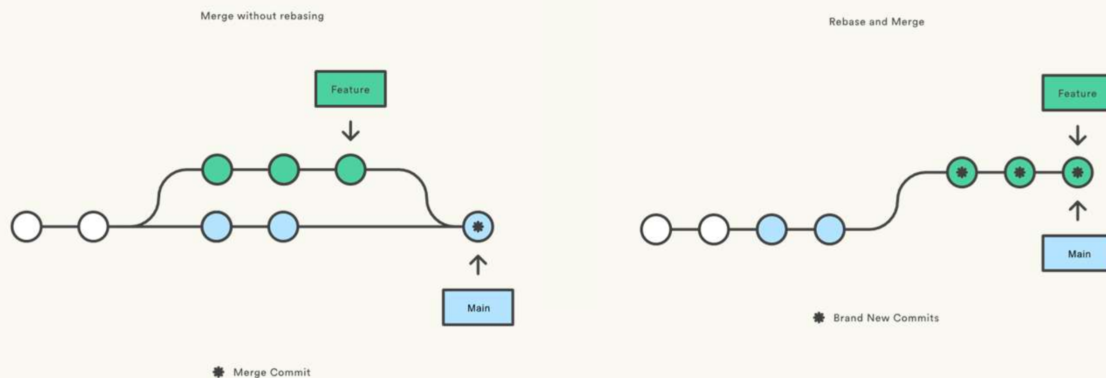
As an overly simplistic recap, rebase creates new commits that replay the history of one branch on top of another branch.

So instead of updating feature branches by creating a new merge commit, like on the left, you can “move” your feature branch commits

with a rebase, like on the right.

This means that whichever feature branch you’re working on has the latest changes that everyone else has made, and you deal with any conflicts on your end, just as before, you just need to use a different command to do so – and because we are not touching the files which were updated in main (i.e. with a merge commit) – we don’t run into the file lock issue!

3. Rebase - Integrating into Trunk



MARCH 18-22, 2024 #GDC2024

Likewise, we need to ensure that when we are integrating a feature branch into the development line, we use a similar mechanism.

Instead of merging a feature into Main, we can rebase Main onto the Feature, aka fast forward main

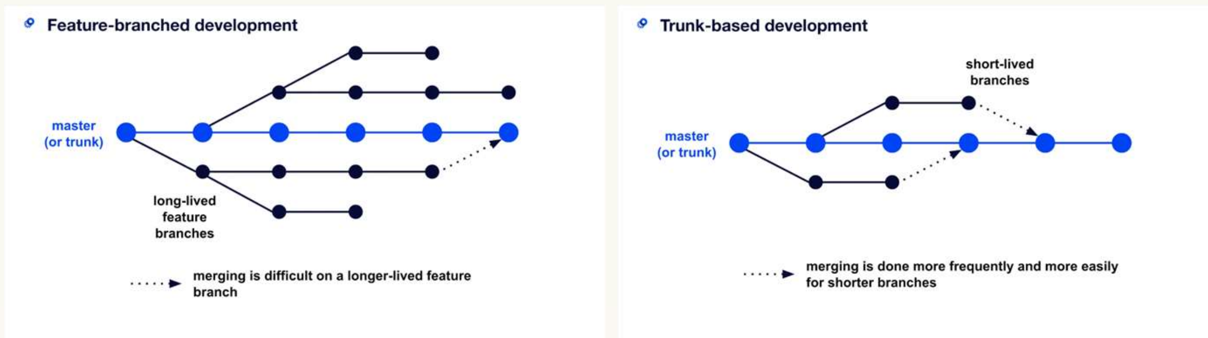
We get all the lovely rebase benefits that git experts love to rant on about: clean linear history, the ability to reword or squash commits to clean

up history, a better ability to hunt down issues using git bisect – but these are all tangential and have been discussed by git experts elsewhere – so we're not going into the advanced use cases and benefits, we only need the basic rebase to solve our problem with locks.

But many of you are likely wondering – how on earth do we get our artists and designers to do this?!!

Well, it involves the 3 Rs… Reduce, Reuse, and Recycle →

3. Rebase - Reduce



ttps://graphite.dev/blog/the-ideal-pr-is-50-lines-long

MARCH 18-22, 2024 #GDC2024

The first thing to do is **reduce** the number of times we *require* an integration (i.e. rebase) from the user!

Ideally, we get the number of user-initiated rebases to ZERO!

Which is completely plausible, and something that most of designers and artists should be able to achieve.

To do so, we have made a push for short lived feature branches.

The longer a branch is alive, the more likely there is a conflict or integration issue, the more painful the integration.

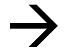
We encourage that every single developer uses a new feature branch for each task, and each task should be doable in roughly a single day.

1) For engineers, my guideline is that each feature branch should contain either around 50 lines of changes, or a single unanimous refactor (no matter how large).

This, therefore, **reduces** the amount of rebases you are forced to do:

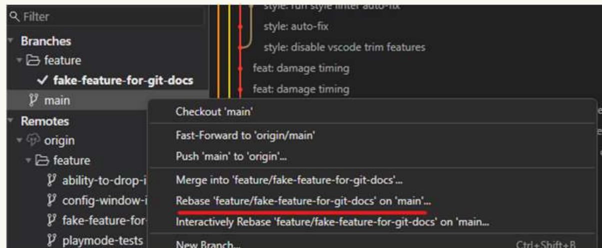
- If your branch is less than a day old, you don’t need to update it to test the latest changes
- And if it has no conflicts because of file locking, you don’t need to resolve

any conflicts.



3. Rebase - Reuse

- Good GUI tooling
- Visual Documentation
- Helpdesk



MARCH 18-22, 2024 #GDC2024

The second situation is when you **must** rebase.

You can't avoid a conflict because you're working on something like code, or a shared subtitle asset file.

Our goal is to make it as easy as possible!

1) Of course, most importantly it is all based on good GUI tooling

2) I *highly* recommend git-Fork – it's an excellent bit of software for a once-off payment per seat, and I seriously think the developer should charge more for what they're offering.

For rebase, Fork avoids using "theirs" and "mine" labels, which is one of the main causes of confusion, so conflicts are easy enough to solve even as a less-technical user!

3) You need *very* visual step-by-step documentation, in an easily accessible and bookmarked location

I recommend writing it *in collaboration with your less-technical end users*

Then, this documentation can be **reused** every single time this situation comes up, until there's no longer any fear.

If there are no conflicts in the rebase, this is a process that only takes a couple of clicks and is just done.

4) And, if there is a conflict, we need a method to help users. I recommend a helpdesk git channel in slack or teams.

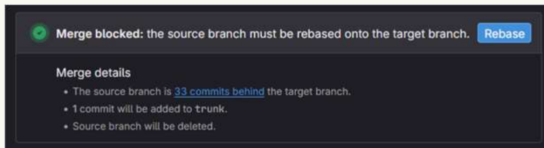
Any developer can ask for help in the channel, and git lovers the company over can reply, going straight to a desk or screensharing via zoom to solve the problem.

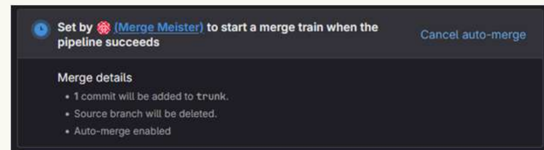
So now we have a system for when a feature branch needs to be rebased, whether due to being stale or due to a conflict, and a mechanism in place to help less technical users resolve conflicts.

Note for readers: Anchorpoint is also a game-centric option, although they use their own implementations for some

things instead of base git, but they also include additional features your artists may like.

3. Rebase - Recycle





And finally, for the integration with main, I prefer using a fast-forward history

(i.e. your branch must be rebased on top of main before integrating)

If you do this as a manual process, this can get really frustrating

Especially because game CI takes a while, and, as we're about to discuss, CI should pass before a pull request can be integrated

Rather than making it a manual process, we use automation to **recycle** as many build results as possible…

Okay, well, technically it's not recycling, but also reducing, but I really wanted to make the 3 Rs work!

1) I've personally had experience with using GitLab Merge Trains, which do quite well, as well as the open-source Marge Bot for GitLab.

I know that GitHub has Merge Queues or a range of 3rd party options like Graphite, and JetBrains The Space has The Space Flow – so there's a version of this for everyone.

All the tool needs to do is handle the rebase/integrate loop for currently merge-able requests.

Specifically, the tool will heuristically

decide on **one** branch to rebase, wait for CI, and integrate.

3. Rebase - Green Builds

Automatically maintain a repository of code that always passes all the tests

-Graydon Hoare

MARCH 18-22, 2024 #GDC2024 GDC

Note! So far, I've been saying that CI must pass, or a pull request will fail.

You should enable this in your git host, and your tool should respect this.

This lets us follow the "not rocket science rule of software engineering" by Graydon Hoare, creator of Rust:

<u>1. Always maintain a repository of code that always passes all the tests.</u>

This means that the test pass *after integration* – not “before a merge”.

By using the method I’ve described, our automated tooling will:

- Selects one of the ready to merge pull requests - i.e. the pull request have been reviewed and approved
- Rebase the selected branch on top of trunk, and wait for CI to run on the integration of the feature and latest-trunk
- Assuming it passes, fast-forward trunk directly to the tip of the branch (and optionally squash the feature)

Guaranteeing that all commits in trunk pass your pre-merge CI tests.

You might have longer running tests limited to be nightly only, but the more you put into pre-merge the better.

Reader’s note: This can be either fast-forward all commits, squash, or merge, all work, it depends on your team’s ability to write good commit messages.

3. Rebase – Merge Meisters

- Rotating responsibility
- Accountable for pull requests
- Monitor for issues
- Assist resolution
- Integrate

https://trunkbaseddevelopment.com/branch-for-release/#merge-meister-role

To keep things moving smoothly, I recommend using the concept of a "Merge Meister"

This is *my* version of a merge meister – not exactly what the literature suggests.

1) Specifically, I like to have a *rotating roster* of Merge Meisters - every day one junior and one senior engineer are assigned →

2) To be Accountable for all pull

requests moving smoothly.

Specifically, I set a KPI that all pull requests must be *looked at* within one hour.

If a code review is needed, the merge meister can either do it themselves, or assign it to another engineer.

3) If an issue *does* occur, like a CI failure or failed code review, we should notify the author.

This is typically automatic – a slack bot or similar – and the merge meister pays attention to ensure the author has received the message.

The author can then rebase their feature branch, fix the integration, and re-submit the request.

4) Of course, merge meisters are then able to help the author resolve the issue, utilising the documentation and

helpdesk, or escalate it further if needed

5) And finally, the merge meister ensures that the changes “ready to integrate” by whatever heuristic our automated tool uses, usually by approving the request!

3. Rebase – Typical Task Flow

- Start a branch
- Work on it
- Submit for pull request
- Done!

And that's it!

Most users, most of the time, simply:

1. Start a branch
2. Work on the task
3. Submit it
4. Take their hands off the wheel!

The merge meisters and tooling take over from there.

Users only need to get involved if

there’s a conflict or a code review request.

For a vast majority of tickets, artists and designers never have to even think about rebase.

So, how do we get the users to be *happy*? →

4. Managing Change

1. Acknowledge
2. Support
3. Document
4. Improve
5. Showcase

There are a million and one books and articles on change management!

This is just what has worked for me, and for teams that already used git.

If you are trying to sell a perforce-using art & design team on git… it's a tricky one, you will **have** to sell them, *in advance*, on what **DevOps** means and why it helps **them**.

But, overall, once you start rolling out a process change like this, it can

get spicy. I follow this loop:

1. Listen to what the developers have to say
2. Support the developer **through** their problem
3. Add to or update the documentation to include the solution
4. Start working on a better way, immediately, and loop that developer in, and once they’re happy with the solution…
5. Showcase the new benefits to all developers! Close the loop from the pain point that started this process.

Support must be easily accessible to all – via helpdesk or DM’ing their favourite engineer

Which, of course, means that all your engineers who might help, need to understand both git, and support

techniques!

4. Managing Change - Acknowledge

- Actively listen
- Empathise and confirm
- Identify root cause

As things change, you might be fielding **multiple complaints**, but it's up to you, as the change manager, to handle all of them simultaneously.

1. Remember that the first, key thing, to be doing is to **listen**. We're ultimately doing this to improve developer lives, as well as product delivery, so ensure that people *feel* heard.

2. We do that through **Empathy**. *Understand* the frustrations of the developer and confirm your understanding by speaking it back to them.

3. Then, find the root cause. They might start by explaining a solution they want, rather than a problem they have. So, ensure that you get to the *problem*.

4. Managing Change - Support

- Work through
- Work around, manually

The second thing, is to support your team to resolve the issue.

1) If possible, the support person walks them through the steps that would have avoided the problem.

Either you have documentation, and it was unclear

Or you don’t have documentation, and you’ll need to right it

But either way, the support person

works through the problem with the user and finds out what is unclear.

2) However, if it *wasn’t* as simple as a user or documentation mistake, your **tools** may be broken or missing something!

This happens awfully frequently when rolling out a new system

To resolve this, allow people to break the process – but only in limited ways!

If you let the user use the “old way” of doing things, you’ll struggle to get buy in, so it’s important to push through!

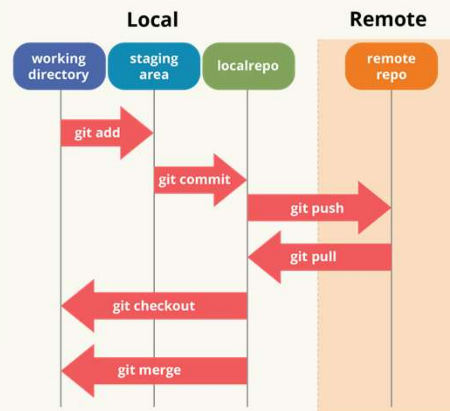
Instead enable power-user workarounds within the new system

For example, if the rebase management bot isn’t working, rather than letting someone merge things in the old way, I would recommend stepping in and

manually taking care of it for the user.

4. Managing Change - Document

- Initial Visual Guides
- Git Presentation
- Technical Writing skills
  - Concepts, Tasks, and Troubleshooting



Next, let's talk about documentation's role in the change management process!

You could start the change in process by building all the docs your users would ever need.

But nobody would ever try and do that… right? (guilty look)

Instead, focus on the core workflow.

1) I would recommend starting with

3 simple confluence pages, step by step visual guides for:

- how to start a new task
- how to work on a task
- how to finish a task

2) Then, it really helps if your support team has a solid understanding of git, I would recommend making a short but thorough visual presentation which you can guide the engineers through.

I’ve previously built up a Miro presentation to teach them *how* git works, and **why** rebase solves the problem with locks

You don’t need to teach a *lot* – but having your engineers understand the basics of git’s data model will help greatly with the quality of support they can offer the rest of your team!

3) In order to help us Improve the

process, the next step in change management, each support person needs some basic technical writing skills.

There are a lot of technical writing guides online that you can lean on.

4) For example, the GitLab technical writing pages talk about splitting up documents by objective types – such as teaching a concept, showing how to complete a task, or troubleshooting.

Additionally, I would heartily recommend that every single engineer completes google’s short and free technical writing course, as they say – every engineer is also a writer!

4. Managing Change - Improve

- Add or Update Docs
- Developer Experience

So obviously, we need to improve the process as a direct result of the user's feedback

1. The first part of that, is to add or update any documentation relevant to the user's problem

Every time your support team helps a user with the problem, they need to reflect upon the docs.

Did the user get confused by them? Fix it.

Is there something missing? Add it.

Once the docs are updated, get the user to review the changes and ensure that their confusion would not have occured

Over time this will turn your docs into a comprehensive solution

2. Then, you want to see if there is any further Developer Experience improvements that you can do to resolve the underlying issue

For example, artists and designers might complain about “pull requests take too long to merge”

If you dig deeper, you’ll realise this is often about subsequent tasks

As a DX improvement, you can create a dashboard that shows expected integration time for the queue of pull requests

4. Managing Change - Showcase

- Tool Stability
- Release Stability
- Fewer Known Defects
- Increased Velocity

MARCH 18-22, 2024 #GDC2024

And don’t forget – the whole point of change is to improve quality – of both the build but also the lives of developers!

To close the change management loop, you need to show off the improvements you’ve consequently made!

Let’s fast forward a couple of months, after rolling out all these processes to the Hyper Lawnmower team.

Now, every single commit in trunk passes the (very tiny suite of) unit tests – including validating the custom tooling for the custom data format.

It's possible that after a couple of months of trying the new processes, your designers or artists might get frustrated.

They'll maybe say something like: "I understand that it's good *for the engineers*, but why do **I** have to deal with branches and things?"

So, showcase what you've achieved:

1) Because pre-merge CI is enforced for all developers, the tools that are needed every single day are available every single day.

There's no more searching for the "last known good" build, or warnings not to

update at the start of the day.

2) More stable feature development processes leads to more stable releases

The last time I implemented this workflow, the live crash rate was reduced significantly with each new release.

The product became **more** stable with *less* effort!

3) Fewer defects get into trunk, so QA will have less work and is able to spend more time providing better details in their bug tickets.

It helps to build up a bit of pride in the quality of work as well – people **want** to ship quality – they're willing to accept a little daily friction for a noticeable improvement in quality!

4) Overall, this process leads to being

able to focus more and build up a faster velocity of work getting completed

The Hyper Lawnmower team is smart. They *get it*. There’s a friction in their work week that no longer exists, they *used to* suffer from an unstable build.

Just because the process is **a little** more involved than before, doesn’t mean they aren’t reaping a **net** benefit!

5. Branching Strategy - Questions

- What about larger or riskier changes?
- And what about releases?

Okay let's wrap up the final set of changes – we know that file locking leads to some difficulties.

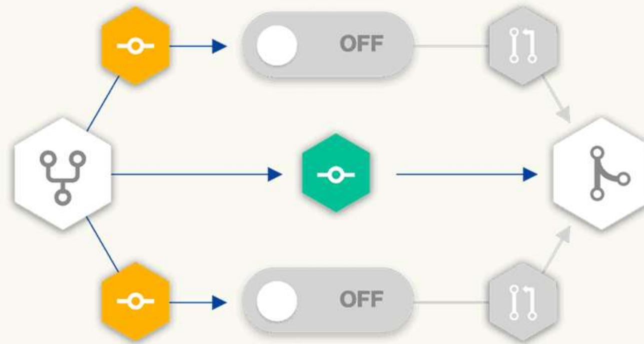
And we covered the immediate short-term ramifications – making short term feature branches the go-to method

We *like* the fact that every single commit in trunk is guaranteed to pass all tests, which leads to improvements in stability and all that jazz.

But given that we can’t merge changes, there are two big branching questions.

1. What about larger or riskier changes like refactors? Do they work in short lived feature branches?
2. And how does this all impact making and hardening release branches?

5. Branching Strategy - Feature Flags



MARCH 18-22, 2024 #GDC2024 GDC

Okay, so the first answer is more of a development method than a version control strategy, feature flags!

If you don’t know what a feature flag is, it’s merely a runtime or compile time flag which can be queried to alternate between code paths.

I’ll focus here on integrating feature flags, for features that might take a week or longer, with the branching strategy we’ve introduced

Very simply, you want to use feature flags to wrap features which are **being developed**.

The first pull request that a developer submits should very simply be the creation of a new feature flag

Then, locally, they turn on the feature flag and start working on incremental tasks.

They merge in each part of the completed system as they work on it – even if the system as a whole isn't working yet

Remember, 50 lines of code of the perfect pull request size!

And finally, once the feature is complete, we may choose to change the feature flag to turn the feature *on* by default.

But we'll come back to *when and how* to do that

There's a swathe of benefits related to

the code review process.

You get additional, earlier, and faster code reviews

Therefore, they're more thorough than a single review of a completed system

And they can help catch bad architectural decisions before they get baked in and you lose work to remake something!

In Hyper Lawnmower's case, what we can do is wrap all the existing *Twitch Plays* code behind a feature flag.

This flag remains off, by default, but for any work that needs to consider impact on the input or command system, or anything relating to Twitch Plays, the developer can manually test it.

Also, we can automatically test it in nightly builds with the flag turned on to make sure we're not breaking the Twitch Plays while working on UGC!

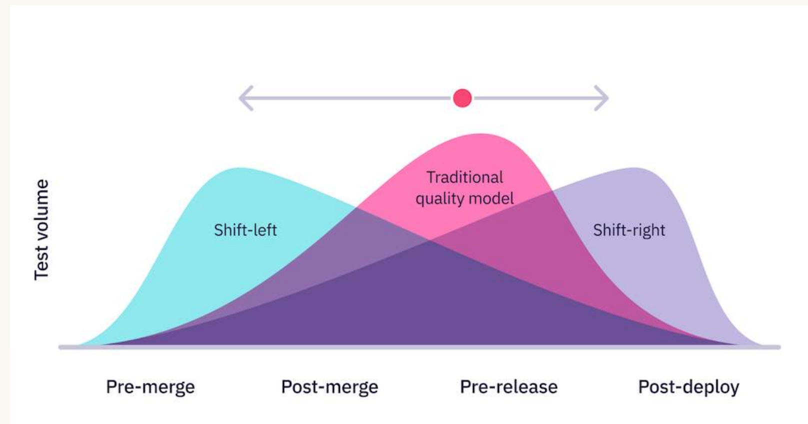
And in between releases, perhaps once a week, we can ask QA to do a functional pass over the Twitch Plays feature to make sure all the prior work isn’t breaking.

Note: feature flags can be compile-time or runtime – if you do them compile time you can use it to prevent packaging assets you don’t want to ship yet – and if you do them runtime you can use it for live A/B testing.

Of course, use what’s most appropriate for your game, and for your feature.

Note: you can also use “branch by abstraction” – which is a similar enough topic that I’m not going to cover it here.

5. Branching Strategy - Riskier Changes



Well, what if we're working on something that doesn't work well behind a feature flag?

What if it's something that is large and risky like a core API refactor?

1) The first thing to understand is that, in general, what we are attempting to do is "shift-left" our testing and quality assurance, move more testing earlier in development.

We want to enable as much testing

before the merge as possible to achieve the “not rocket science” goal – all commits in trunk being green.

So, let’s enable the Hyper Lawnmower team to be able to do just that, even with larger changes!

If a feature *absolutely cannot go behind a feature flag* – then we allow for a developer, or a pair of devs, to have a longer-lived feature branch – in the old style.

Obviously when the branch is about to merge in, our already set up tooling will catch any CI test failures.

However, for some changes, we know that automated tests are unlikely to be sufficient to catch all potential issues.

Bugs may come in edge-cases or are otherwise not covered by regular CI.

We can solve this problem by allowing our developers to map a git feature branch onto a (hidden, beta) steam branch through CI.

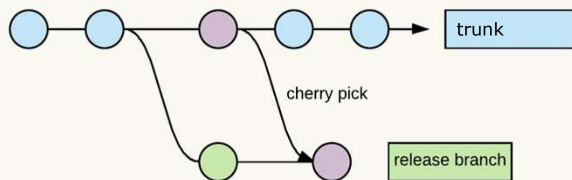
When they’re ready for additional manual testing, they can request it from QA

Because your long-lived git feature branch is automatically built to a steam branch by CI, your Functional QA then have an easy way to do some targeted testing.

And once QA signs off on the feature, we merge it in!

With a lot more peace of mind that the change is safe because we left-shifted Functional QA.

5. Branching Strategy - Releases



- Code lock → git branch
- Must fix → git cherry-pick
- Verified → git tag
- Also: build RC nightly!

MARCH 18-22, 2024 #GDC2024
GDC

Now, Hyper Lawnmower on the High Seas is gearing up for its next early access release.

Considering merges don't work anymore, how do we deal with having a release branch now?

Fortunately for us trunk-based development has an answer for us – and it works out of the box with file locking!

In fact, trunk-based development

proposes *several* possible solutions, and the one I’m describing here is merely my favourite

1. First, when we hit code lock for a particular release and are sending it to QA – we create a git branch in a release folder.

The act of creating this branch triggers our CI and sets up the steam branch for our QA to start looking at.

2. Then, if any fixes are required before a release can go out, we develop it using our usual feature branch workflow **on top of trunk**, and cherry pick the merge

3. Finally, once QA has approved a build, we tag it with a git tag, which triggers off the final steam upload to a staging branch, which can then be scheduled to publish by our release managers.

There are some great benefits to this!

Our release branches are smaller – we cut them later in the cycle, and we no longer develop fixes against the release branch

So, our developer workflows are simpler (they **always** merge into trunk)

And QA is easier (because we can verify the fix *works* in trunk before cherry picking to release)

And it also means we don't have issues with merging a release back into trunk!

However – compared to our older method of branching and hardening against the release (i.e. gitflow) – we now branch much later into a release.

This means QA doesn't get a release candidate until much later – they don't have a few weeks of hardening – so let's flip how we think about RC's on its head:

4. Build the candidate for the next

release every single night – even when we are months away!

5. Branching Strategy - Flags & Versions

- Centralised Control
- Development Features
- Version Locked Features
- Multiple Nightly Builds

So now we have all the pieces – let's go back and look at feature flags to add some sugar on top

1. One thing about Feature Flags that I strongly recommend – is having them all controlled from one location.

There are a lot of ways to achieve this – you can use your Build.cs files in Unreal Engine, or a 3rd party tool like Unleash for almost any engine

2. My preferred starter feature flag is a "*development"* feature flag, which is off by default – and can be opted into by individuals

These are the flags that are used for longer running, or riskier features, before they are ready to be turned into →

3. "version locked feature flags" – a feature flag that simply says "from this version onwards, this flag is set"

This is automatically on for all builds above a given version number – with all developers always building the highest version locally by default

This also give us the benefit of moving the release of specific features to be a **business** decision, not just a **software** decision, as to when you release them.

Product Owners can play with the "set" of completed features, and release them incrementally at their own whims, based

on their user-readiness.

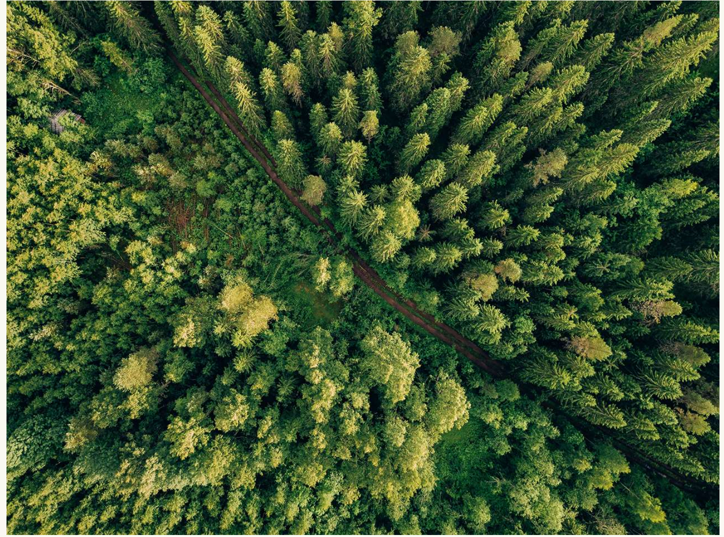
4. And to tie this into our releases – we now enable **multiple** nightly builds. One for the next patch version increment, one for the next minor version increment, and one for the next major version increment.

This generally covers all possible permutations of features being on and off, and lets QA test builds well in advance of a release

Shifting more and more testing “to the left”.

The Path to Improvement

1. Git LFS
2. File Locking
3. Rebase-Centric workflow
4. Managing Change
5. Branching Strategy



MARCH 18-22, 2024 #GDC2024

Let’s recap the story of what we used to help Hyper Lawnmower on the High Seas get better.

1) Firstly, liberal usage of git lfs

2) Secondly, we enabled lfs file locking for highly contentious and unmergeable files

3) Thirdly, this **forced** us to go towards a rebase centric workflow

4) Which, obviously, can be a bit of a

struggle to switchover to

5) And finally, the struggle eased off and we started reaping lots of benefits when we adjust our branching strategy to fully utilise trunk-based development at scale

Especially focusing on smaller, daily, feature branches

Rolling it out

Doing it better

#GDC2024

Alright, now that we've talked about the highly hypothetical rollout at Tiny Turtle Studios, let's take a step back.

I want to talk about things that I would do differently, if I was doing this again, rolling it out at another studio, or on my own projects.

And if you are going to take this home, I really hope that you do better than I did!

Even though my previous implementations were worth it, they

were more of a struggle than they needed to be!

Costs to Consider

- Merge Automation Tooling
- Hardware and Administration
- Documentation and Training
- CI Costs
- Friction and overhead

MARCH 18-22, 2024 #GDC2024

Firstly, let’s talk costs – I did say we were going to achieve AAA quality on a budget!

1. The first and possibly most central cost to consider is the merge queuing or stacking tool.

Most of the available tools in this area come under premium offerings from git hosting providers

GitLab, GitHub, and Jetbrains the Space all have an option that should work – although I've only tested GitLab's fast-forward merge queues

Or an open-source alternative is Marge Bot, which unfortunately doesn't handle file locking properly yet.

If someone wants to add that – reach out and we can collaborate! It would be great to enable this workflow for free!

Overall, if you compare the premium git hosting providers costs, you'll see that it's not cheap, but it is still "budget" compared to Perforce.

I can't share exact numbers here, but if you already have a business relationship with any of these companies, they'll happily lay out a costing appropriate for your studio.

2. Hardware and Administration – overall roughly the same cost whether you go

with self-hosted git or perforce – but if you go with an online git provider you may end up spending more over time and having less control

3. Documentation and Training for the team is something that takes time, which is money, to develop.

If you try and do it up front, it’s really expensive – but I’ve given you a roadmap of how to do it cheaper!

4. CI Costs – this will be identical between Perforce and git so I’ll skip over it – you can use whatever system you like with either

Although, I would **strongly** recommend that you use infrastructure-as-code, which most git hosting providers have a solution for, if you don’t want a third party one like TeamCity.

5. Friction and overhead – the day-to-day friction of creating a branch and managing pull requests.

Realistically, this is noticeable at the start, but within a couple of months, max, this should become negligible.

Regardless, as shown so far – I'm sure you can see the benefits outweigh the costs!

So, in the end… will this be cheaper than Perforce?

Based on numbers I can't share; the answer is **yes** – even at a scale of several hundred people.

For *your* studio? It's up to you to do the costings, but please reach out to me if you need a hand with figuring out the details!

As soon as possible

- Git Attributes
- pre-commit
- Infrastructure as Code (IaC)
- Pull request requirements

Things that I would recommend you do as soon as possible:

1. Set up git attributes for git lfs and locking as the very first commit in a repo
2. pre-commit integrations

This can be file size checks to catch un-LFSd files, commit message validation, clang-tidy, or whatever else you can get going

3. Then, ensure your CI is set up using infrastructure as code

And ensure all of your engineers understand the bare bones basics of how it works

4. And lastly, enable your pull request requirements

Approval requirements, CODEOWNERS files, CI must pass, and the tooling we've discussed

I would only ever start development after setting up all of these, and as many of the things I mention in this talk as possible, but if you're here you likely already plan to.

Do Better

- Rollout Strategy
- Feature Flags
- Engine Integration
- pre-commit hooks

Things I want to do better next time:

1. I wish we rolled out the system in a more cohesive way!

When I've done this in the past, it's been quite reactionary, dealing with problems as they arose.

If I had a stronger vision from the start, and got more buy in, then I think change management would have been a lot easier!

The aim of this talk is to give you all the tools to do just that.

2. Feature Flags

I would like to use them more liberally in our code – but also see if we can integrate them better into asset management!

3. Better engine integration

Both Unity and Unreal have git plugins.

The main Unreal git plugin I'd recommend is by Project Borealis, which includes a very good file locking integration.

However, that the current state of Unity git plugins isn't ideal.

Fortunately, I know for fact it can take a competent developer just a couple of weeks to fix that – but it's not yet open sourced or easily available.

4. Finally, pre-commit hooks

This is another thing I think could be open sourced for the good of the game dev community – a good set of pre-commit presets for Unreal and Unity.

In prior roles, I've added file size checks, commit message validation, automated code formatting, and some static analysers, which help shift your quality checks further to the left.

But I suspect there's even more we can do here that's game specific – like checking that textures are power-of-2 if they're being checked in, and things like that.

An Eye to the Future

- Research and Development:
  - git scalar and sparse checkout
  - Alternative to UGS
- Tooling Needs to Improve:
  - Batch to reduce CI thrashing
  - marge-bot supporting locking

And finally, here are some of the next steps.

1. I’m doing a bit of research, and potentially development, into

2. In the past, Microsoft developed a system called scalar, which was a scaling wrapper to git.

However, almost everything from scalar has now been merged into git

itself, and git has added a few features as well.

Specifically, I want to look into are shallow clone (skipping LFS files), partial clone (skipping history), and sparse checkout (checking out specific LFS folders only)

If we can make this artist friendly, I think this might be the last step to fully replacing perforce!

Note: none of the git GUI programs support this in its entirety. Only Anchorpoint, as far as I’m aware, supports **some** of this

(https://www.anchorpoint.app/blog/scaling-git-to-1tb-of-files-with-gitlab-and-anchorpoint-using-git-lfs#sparse-checkout)

3. Additionally, I want to figure out an alternative solution to Unreal Game Sync to sync custom engine binaries – maybe by forking it to add git support over p4?

I know that ProjectBorealis actually has a solution for this as well – PBSync – but “official” support via UGS would be ideal.

4. And there a few things I think need to improve in the ecosystems to make this a lot easier to deal with at even larger scales.

5. Tools like gitlab, github, etc. that support this workflow don’t have a good batching solution to reduce CI thrashing! If you are merging multiple PRs at once, you should be able to just run CI on the combination of all of them, to get a green light to merge all of them.

6. marge-bot HAS batching – but it doesn’t support dealing with file locking – it would be good if someone can fix that for the opensource community!

I know that this system can work for a single project with a team of 30-40, and a repo of up to a terabyte in size.

With these additional features, I think we can raise that by an order of magnitude.

GDC MARCH 18-22, 2024 SAN FRANCISCO, CA

Wrapping it up

1. Aim for the Stars
2. Tiny Turtle Studios
3. Hyper Lawnmower
4. Doing it again

#GDC2024

So, to very briefly summarise what we’ve covered.

1. If we’re going to use git, we’re going to aim for the stars

This means attempting to achieve everything that the biggest software companies to in modern DevOps

But balancing it with everything that our artists and designers need to succeed.

2. Including the artists and designers at the hypothetical Tiny Turtle studios.

I’m sure that most of us have worked in studios not unlike Tiny Turtle.

3. Next, we dove into the nitty gritty details of how we can take their flagship IP and improve its processes.

This section should hopefully have given you enough of a framework that you could implement this yourself.

4. Finally, we covered what’s next – for me, for this workflow, and hopefully for you!

Bonus – git tricks

- rerere
- git rebase --update-refs

Now as a super quick bonus slide, a few git power user commands or config options to try out:

1. rerere - reuse recorded resolution – helps if you need to rebase multiple times – can be used as a command or a config
2. git rebase with update refs – lets you rebase a set of stacked pull requests at the same time

Get in touch

contact@vikram.codes

Check out the locks write up:

- vikram.codes/git-file-locking

Resources and References:

- trunkbaseddevelopment.com
- dora.dev

I've been Vikram, thank you all for going with me on this journey

Feel free to get in touch, ask me questions at any level of detail, or even to get me to consult with setting this up yourself!

I've put up a blog post on git file locking, which you can check out on my website

And if you’re only just hearing about trunk-based development, or have seen it only done badly, please check out trunkbaseddevelopment.com

Likewise, for devops, I can’t speak highly enough about DORA, the DevOps Research and Assessment program by Google – their information is excellent.

And with that, any questions?