

Git Branching – the battle of the ages

What's better?
Extensive branching or
trunk-based development?

Jasmin Fluri
Gianni Ceresa





Disclaimer

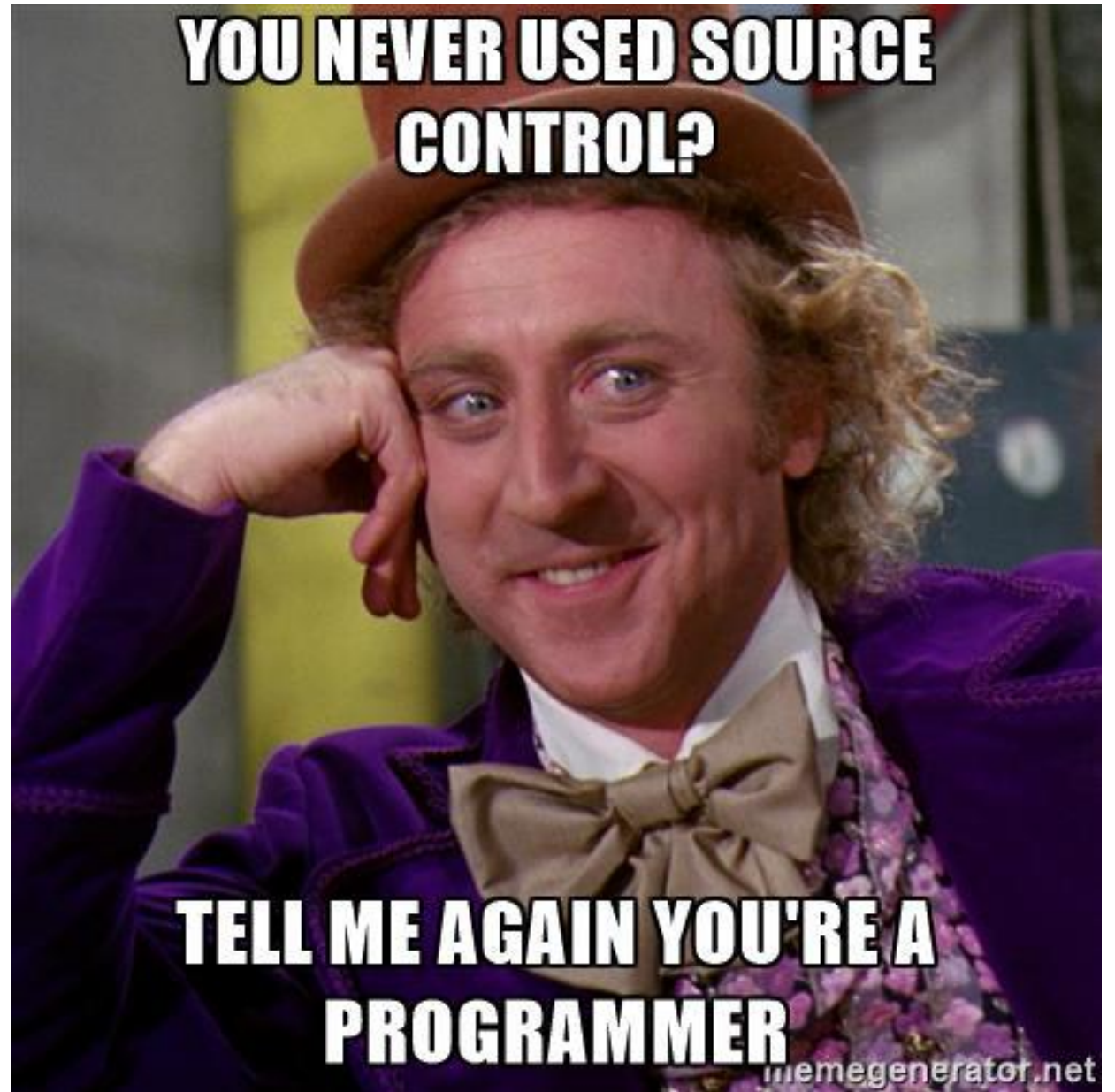
This presentation is intended for educational purposes only and does not replace independent professional judgment.

Statements of fact and opinions expressed are those of the presenters and are subject for bias.

The background features a series of concentric, semi-transparent circles in shades of light blue and green, creating a layered effect. The overall color palette transitions from a light blue on the left to a light green on the right.

Why do we need a
branching strategy?

Why you
absolutely
need a
defined git
strategy!



... not
having a git
strategy
means...

some people push their changes
directly to main...

... others create branches,

some create merge requests and
expect a review...

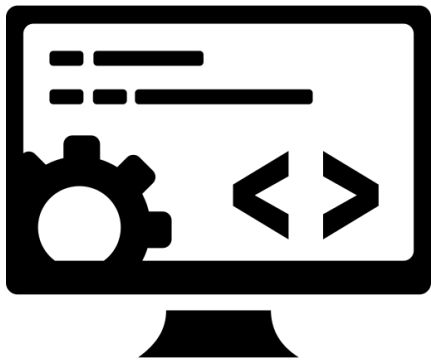
tags are sometimes created,
sometimes not...

... it's a mess.

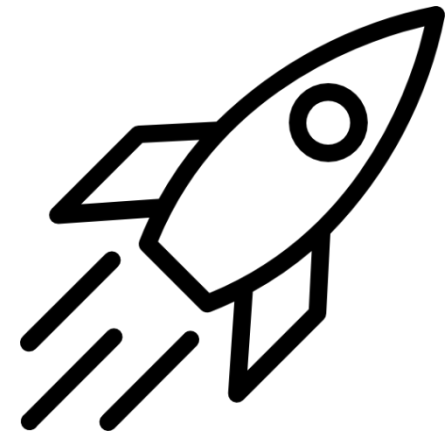
Without a defined git strategy...



... we need to define what happens ...
a branching strategy covers the FULL lifecycle



From when we
start implementing
a feature...



... until it is installed
into production!

2 different git strategies

Gianni Ceresa

DATAlysis LLC

I first started with CVS back in ... can't remember...

 @G_Ceresa



I'm supposed to play the old man here...

Git-Flow
GitLab Flow
GitHub Flow
OneFlow

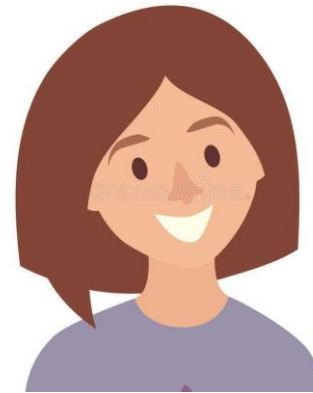
Branch-based development

Trunk-based development

I'm supposed to play the young gun here...

Jasmin Fluri

Schaltstelle GmbH



Born in the Git century – never used SVN or CVS

 @jasminfluri

Trunk based development

4 Steps in a Project Lifecycle

1

Start project

2

Review code

3

**Build an
integration
pipeline**

4

Ship feature

4 Steps in a Project Lifecycle

1

Start project

2

Review code

3

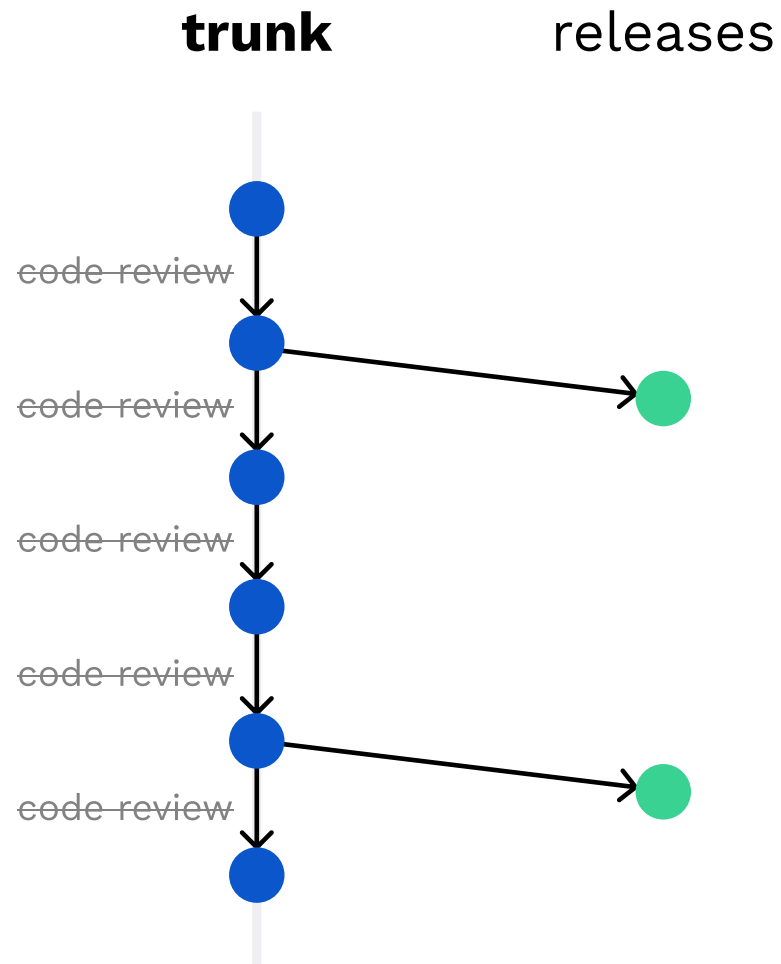
**Build an
integration
pipeline**

4

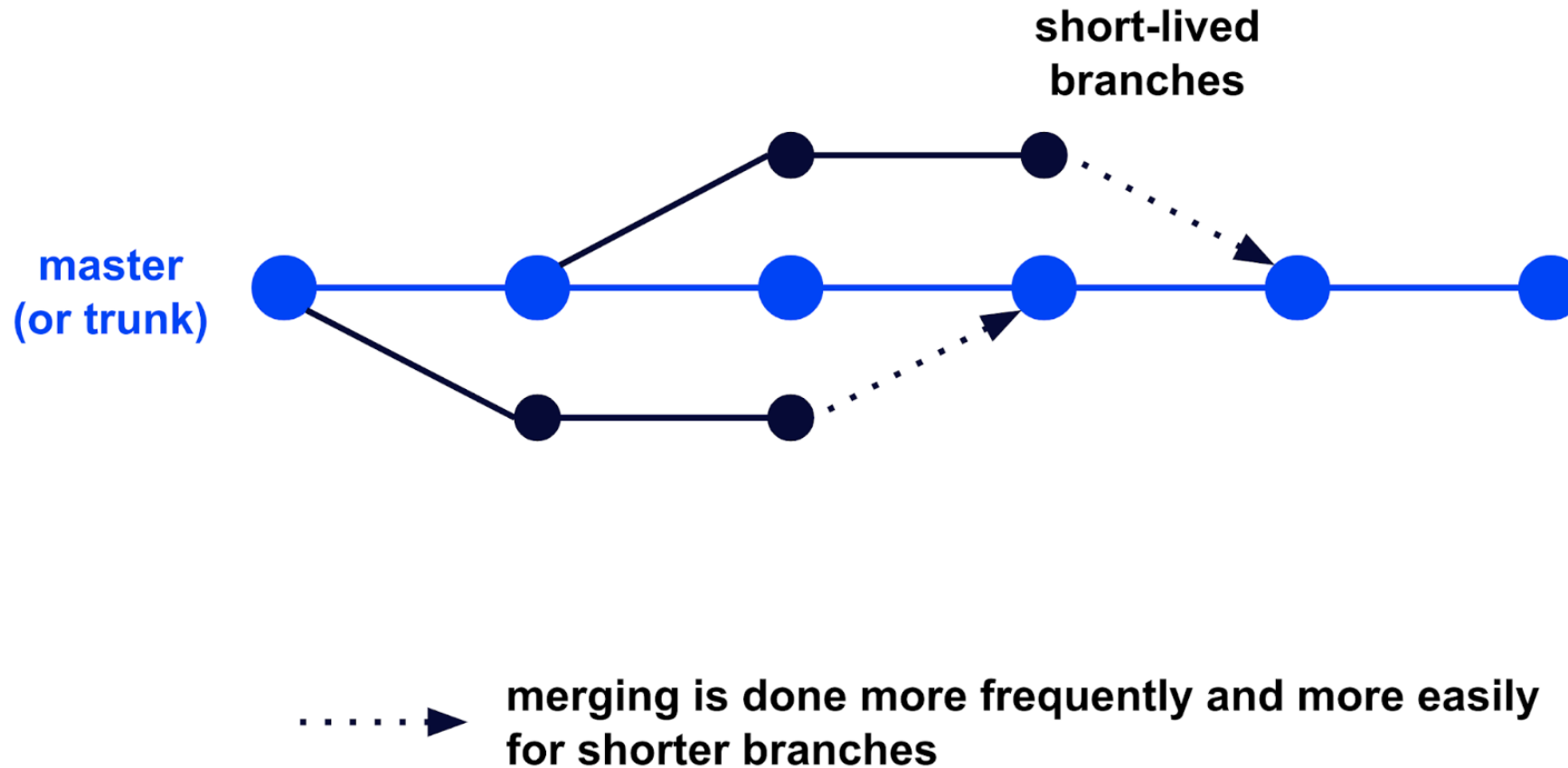
Ship feature

Starting a project ...

Trunk based development is easy!

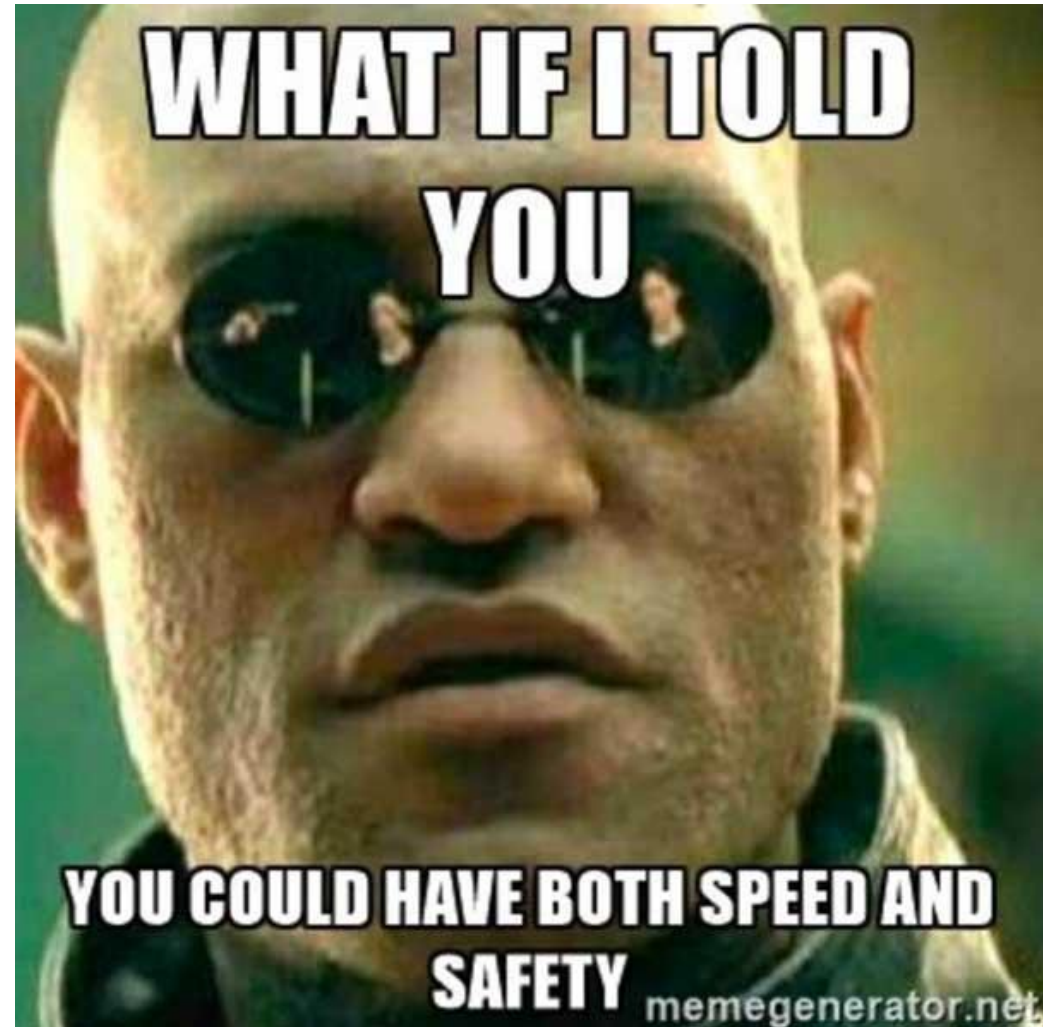


Trunk based does not mean you have no branches!



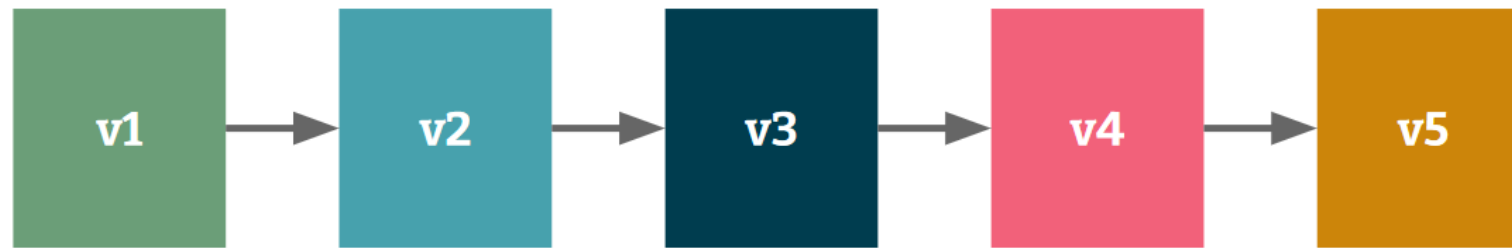
Trunk-based development combines both speed (bringing changes into production fast)

and safety (automated testing in the pipeline).

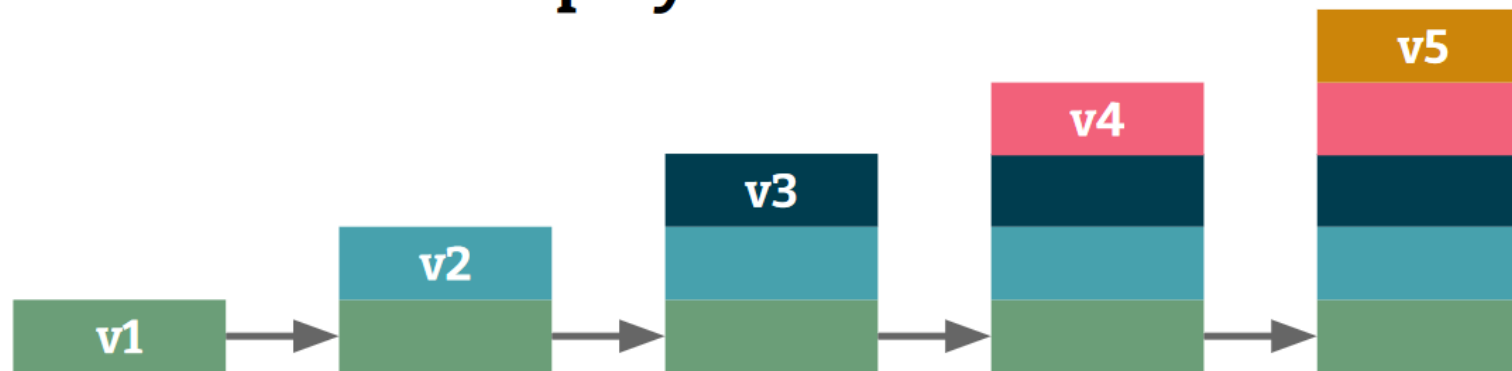


Trunk based development and infrastructure code!

Application deployment



Infrastructure deployment



© 2021 Thoughtworks



git merge

master

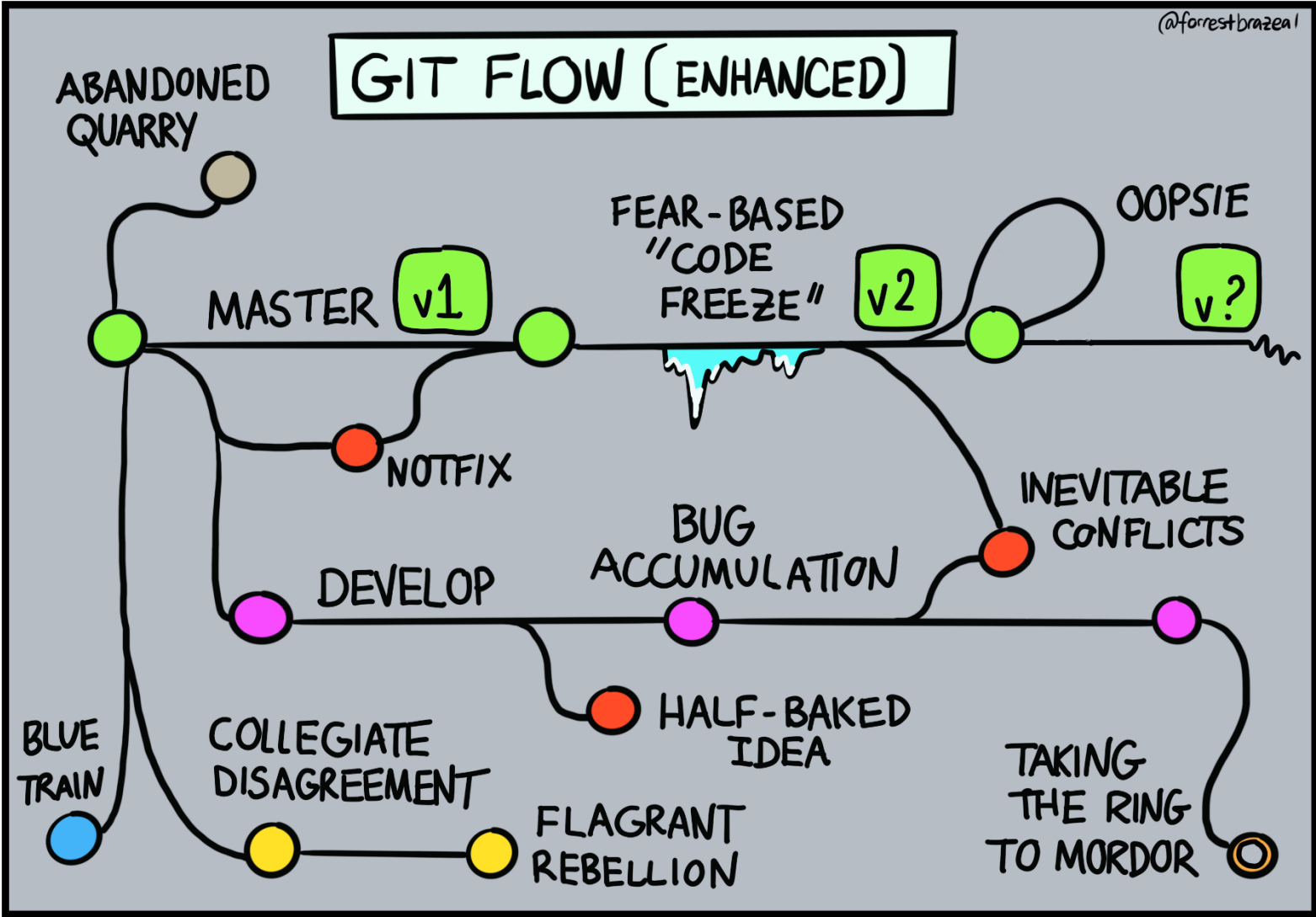
origin

Branch based development is

Also easy!

Until Jasmin finds a
fun picture of it...

Branch based development is



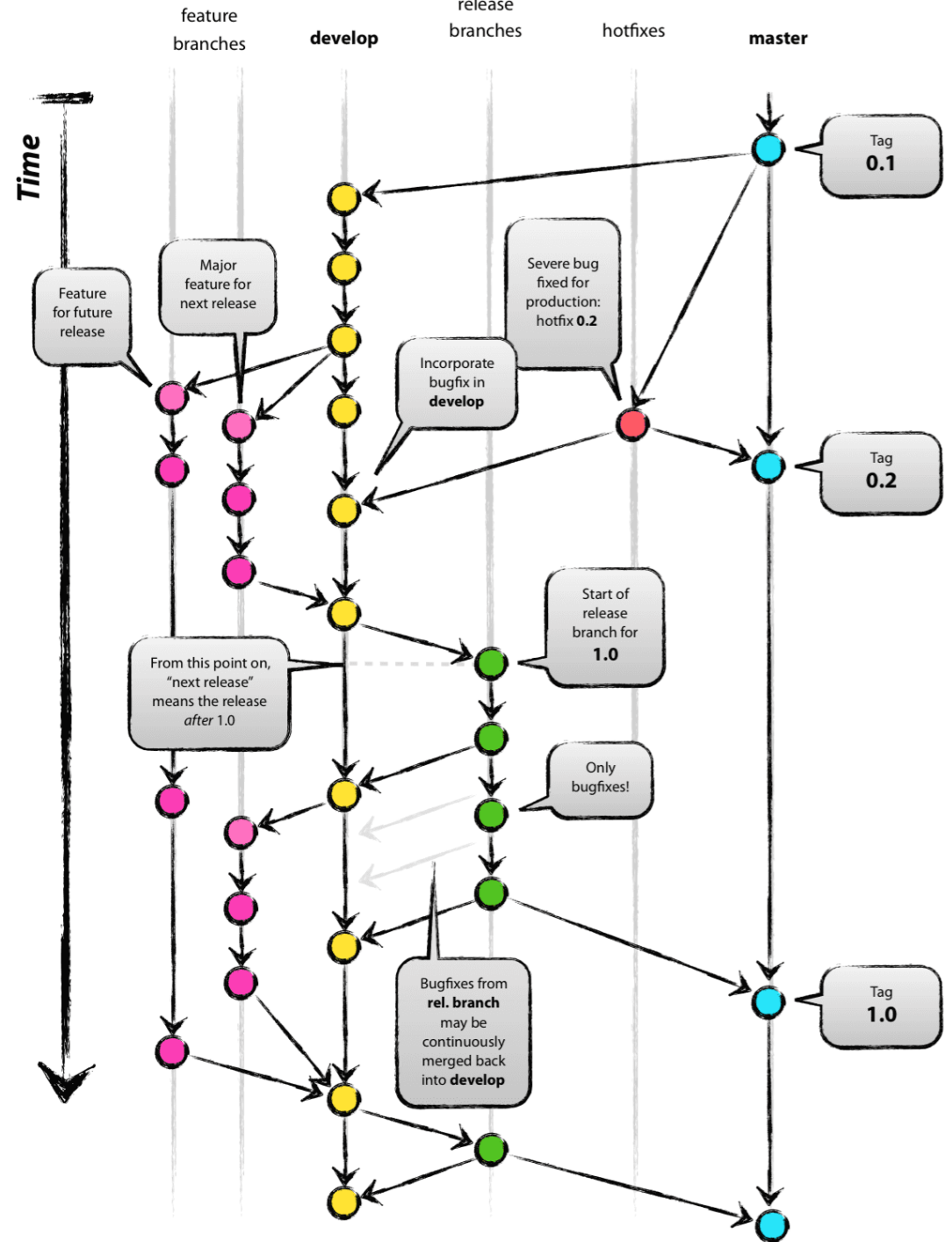
GitFlow

Everything can be a branch

Everything “needs” a branch

Once you start using branches,
no reason to stop!

And then you merge, merge and
merge branches...

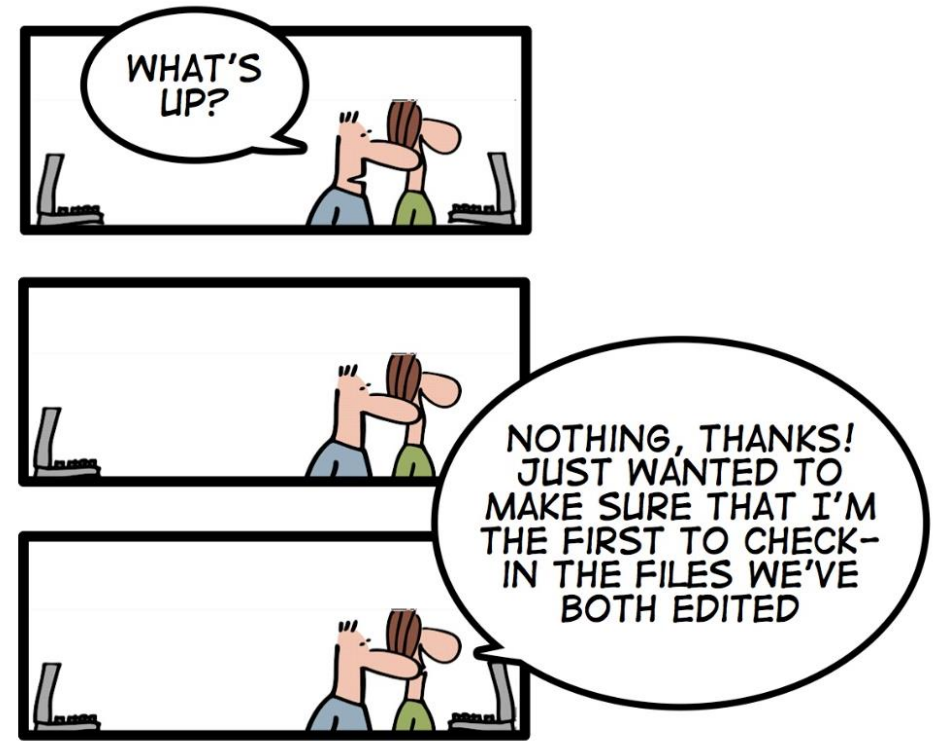
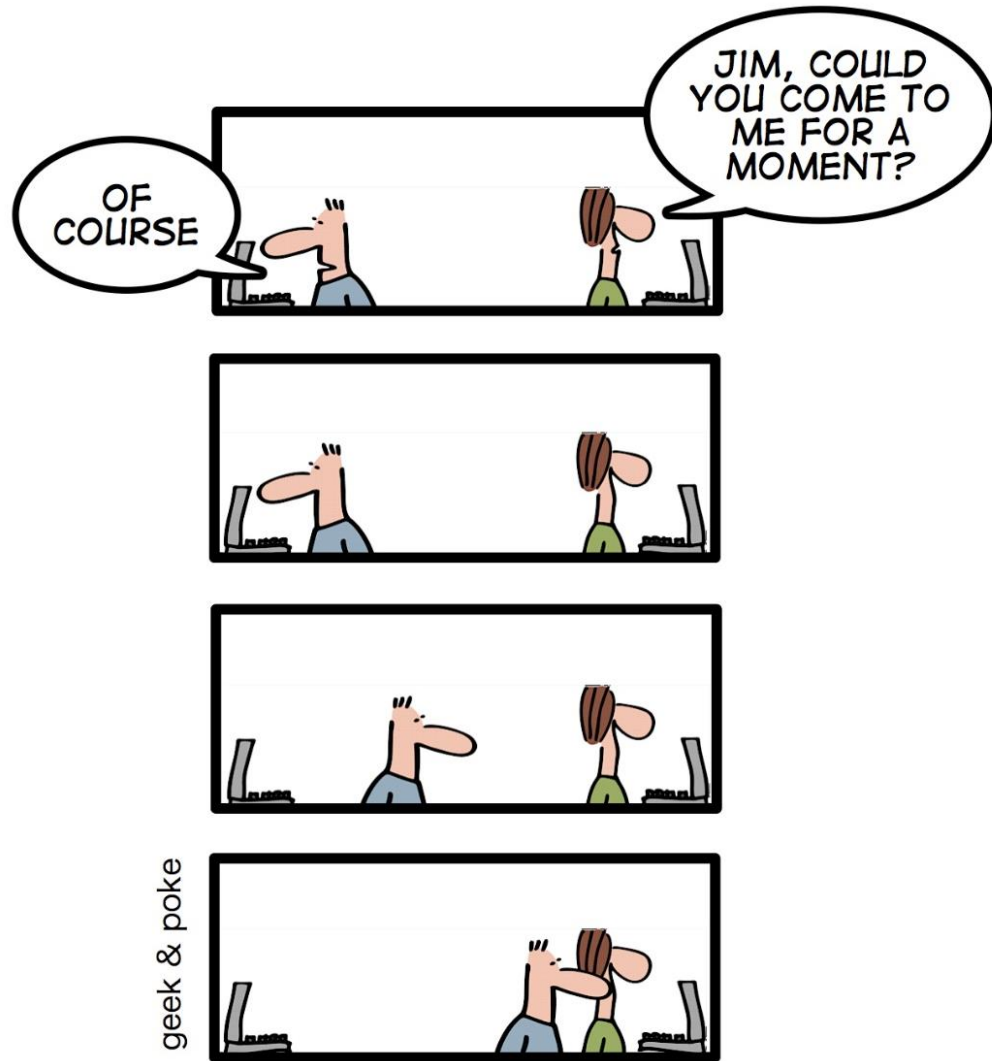


**WHEN JASMIN SEES A MERGE
REQUEST**



**AND DISCOVER IT'S ASSIGNED
TO HER**

BEING A CODER MADE EASY



CHAPTER 1: HOW TO
AVOID ~~MERGE~~ COMMIT
CONFLICTS

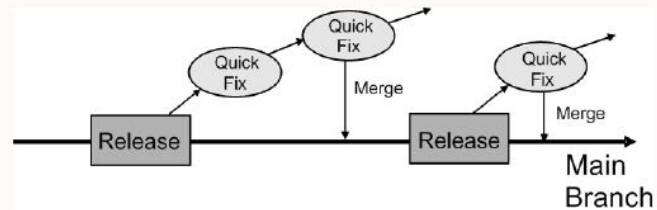
GitFlow ...



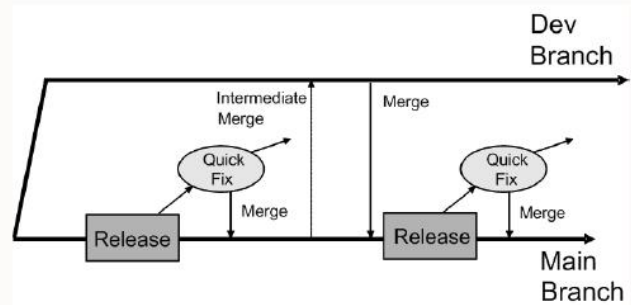
Branch based development is... suggested by Oracle

Multi-User development - Branching

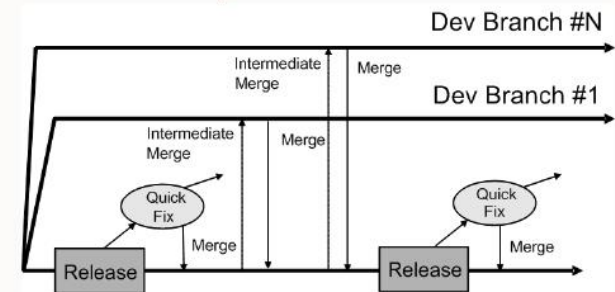
1. Simple Development Model



2. Small Team Development Model



3. Multi-Team, Multi-Release Model



Copyright © 2022 Oracle and/or its affiliates.

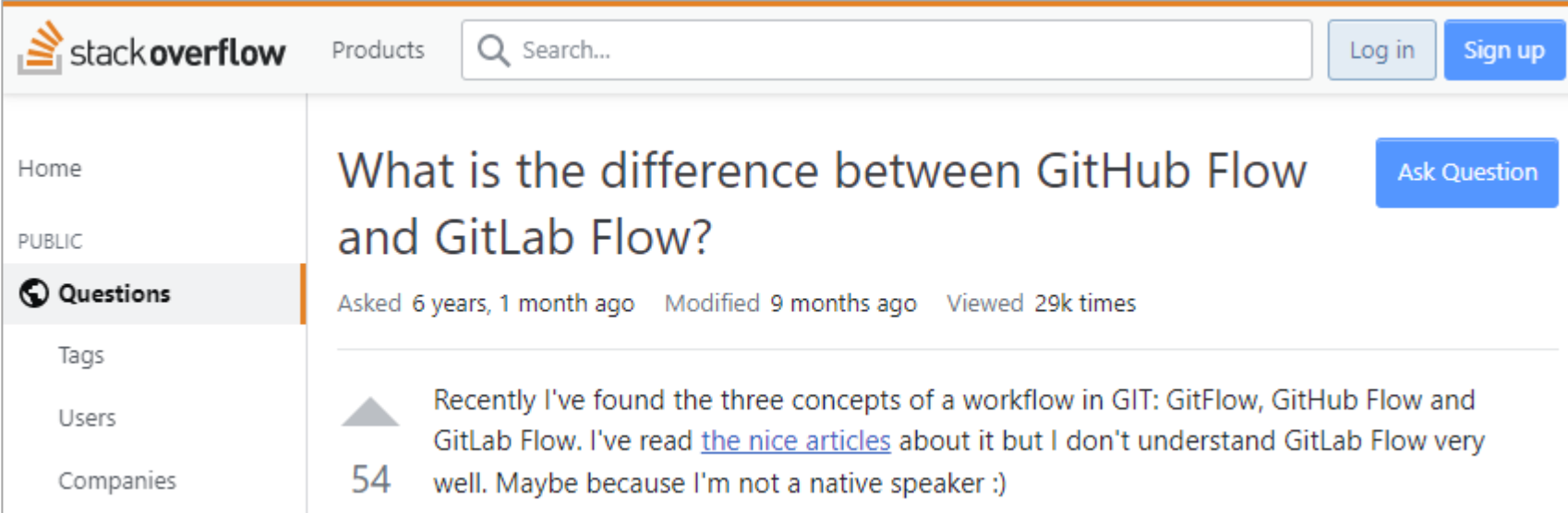
Branch based development: many ways

GitFlow has been the first, presented in 2010

- In development age this is a loooooong time ago!

Many more followed:

- GitHubFlow
- GitLabFlow
- OneFlow
- **whatever*-flow*



The screenshot shows the Stack Overflow website interface. At the top, there is the Stack Overflow logo, a search bar, and buttons for 'Log in' and 'Sign up'. The main content area features a question titled 'What is the difference between GitHub Flow and GitLab Flow?' with an 'Ask Question' button. Below the title, it indicates the question was asked 6 years, 1 month ago, modified 9 months ago, and viewed 29k times. The question text reads: 'Recently I've found the three concepts of a workflow in GIT: GitFlow, GitHub Flow and GitLab Flow. I've read [the nice articles](#) about it but I don't understand GitLab Flow very well. Maybe because I'm not a native speaker :)'. The number '54' is visible below the question text, likely representing the number of answers.

Good luck with Trunk based!



Or relax with Branch based!

Alone

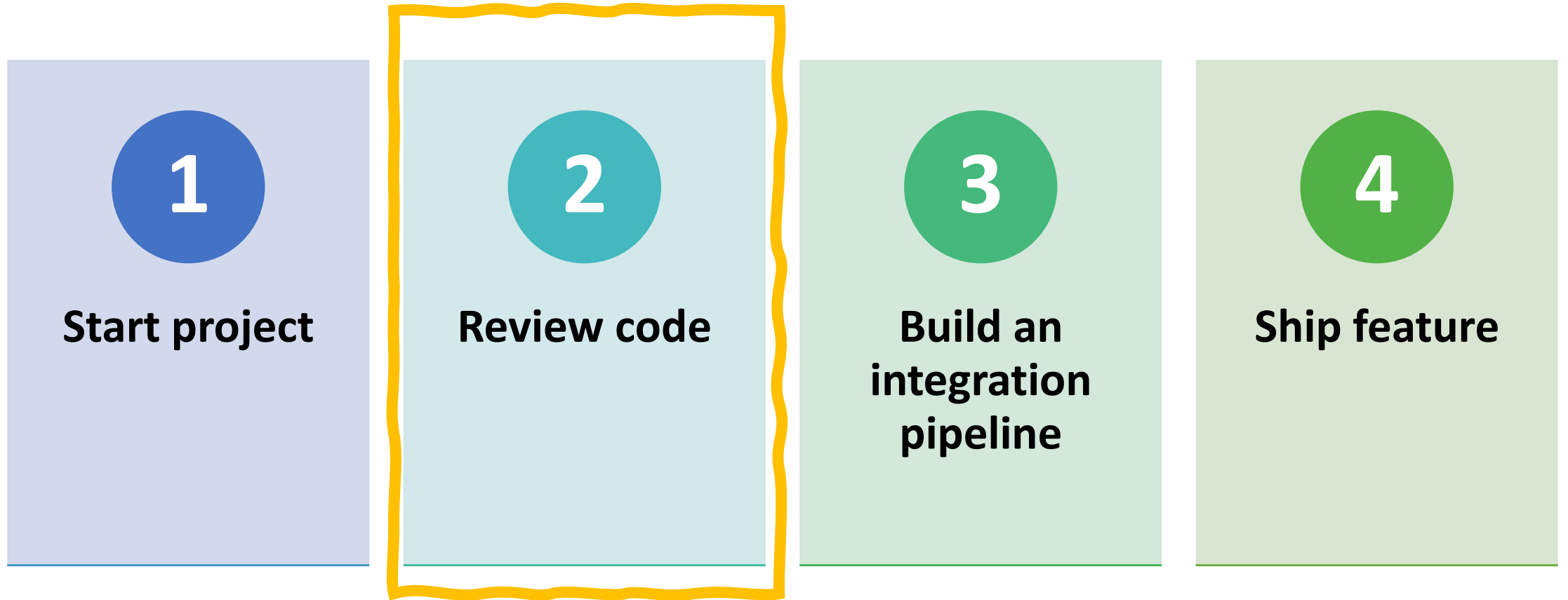


Or relax with Branch based!

Or in a team

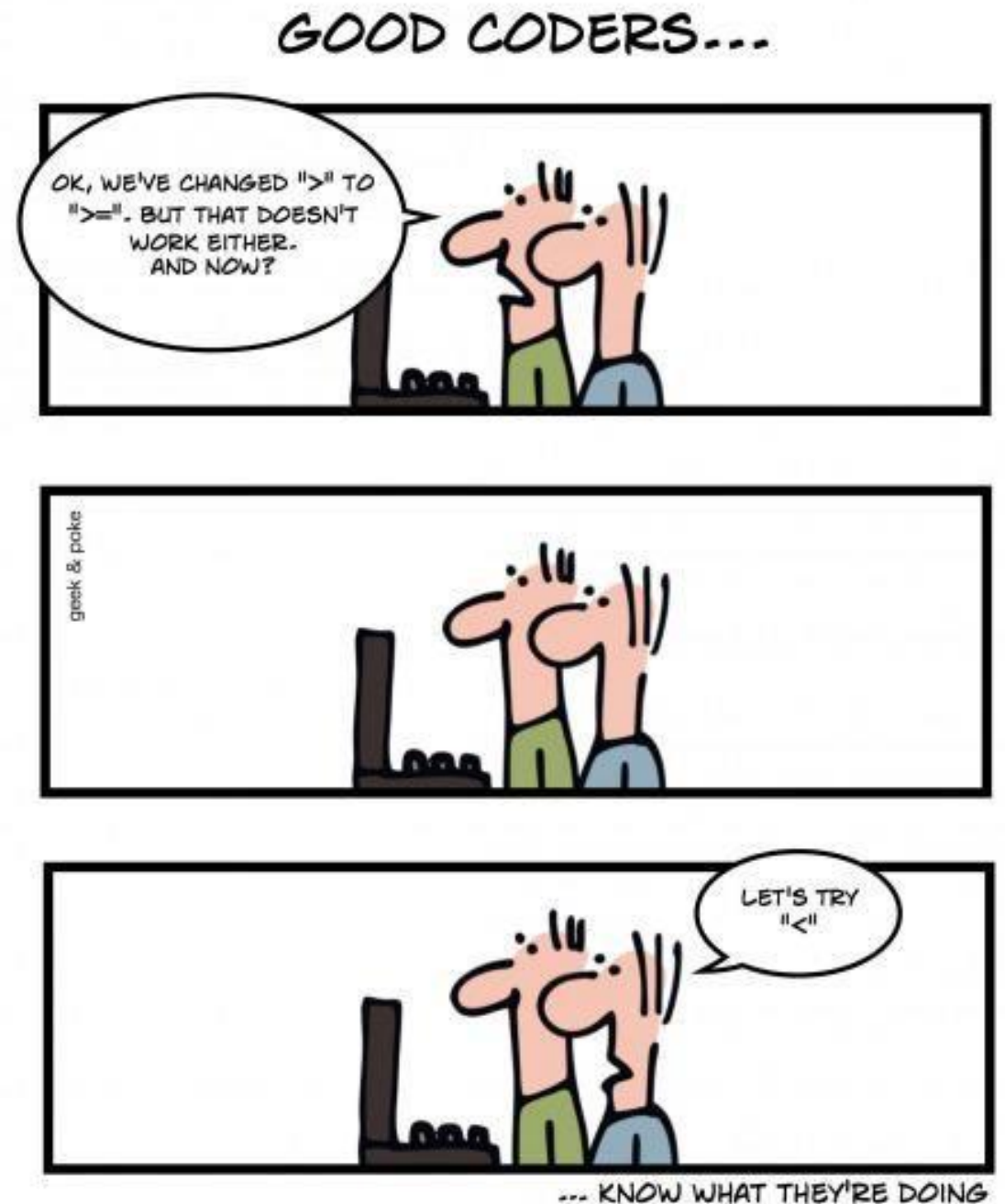


4 Steps in a Project Lifecycle



Code Reviews in Trunk-based Development

- Either happens directly during pair programming
- ... or code reviews are very small due to very small merge requests!
- But we want to avoid merging...



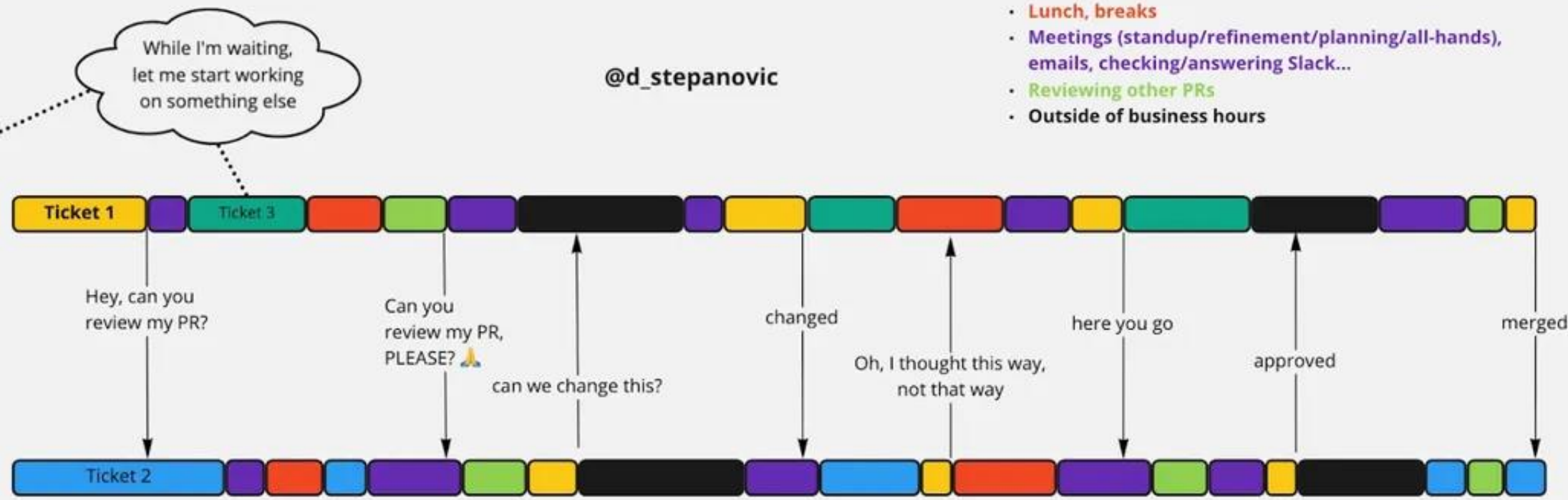
Handovers in software development



Jasmin

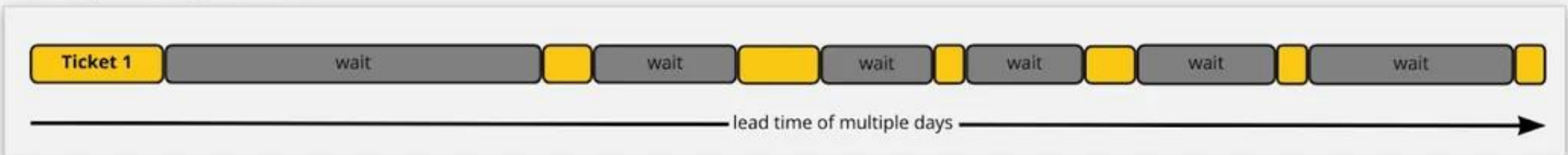


Gianni



- Lunch, breaks
- Meetings (standup/refinement/planning/all-hands), emails, checking/answering Slack...
- Reviewing other PRs
- Outside of business hours

wait to processing time ratio



Merging is bad for you!

- Merging is hard work
- Merging is a change to the codebase
 - requires testing before and after the merge
- No guarantee that integration works both before and after the merge
- No branches == no merging
- Small batches == short test cycles

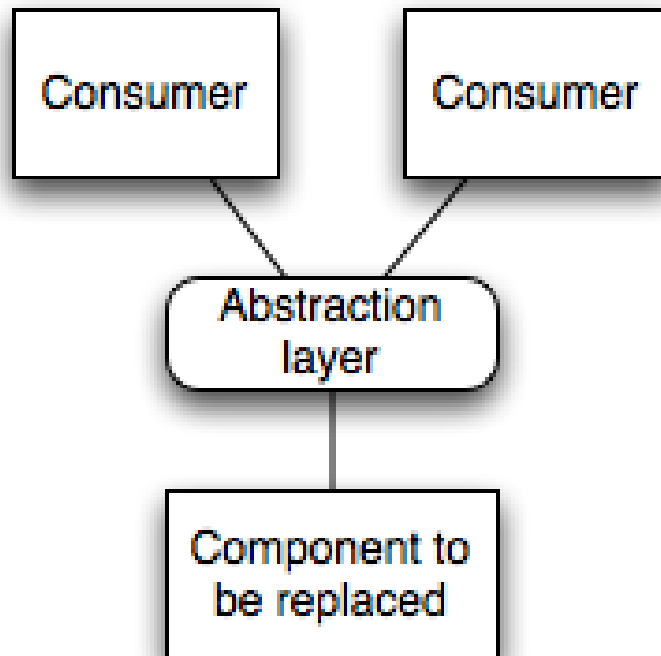


But we need a way to introduce changes step by step!

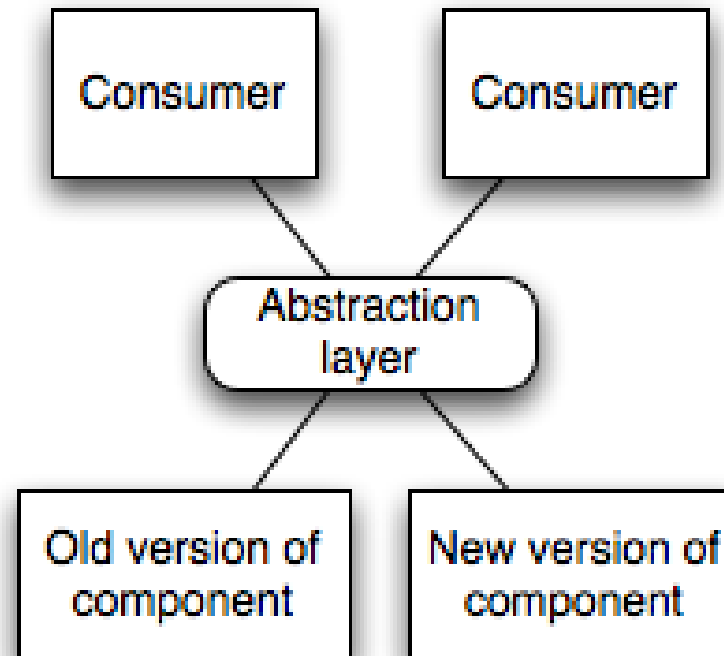


Branching by Abstraction is your Friend!

Steps 1 and 2

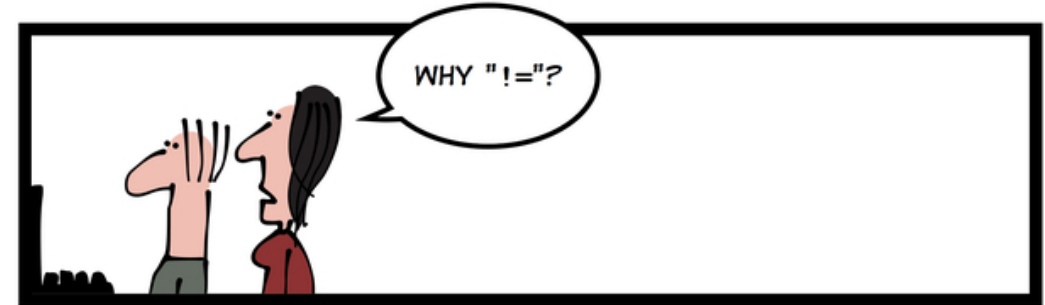


Steps 3 and 4



Code Reviews in Branch-based Development

- Either happens directly during pair programming
- ... or code reviews happen once when the development is over before merging!
- And we aren't afraid of merging...



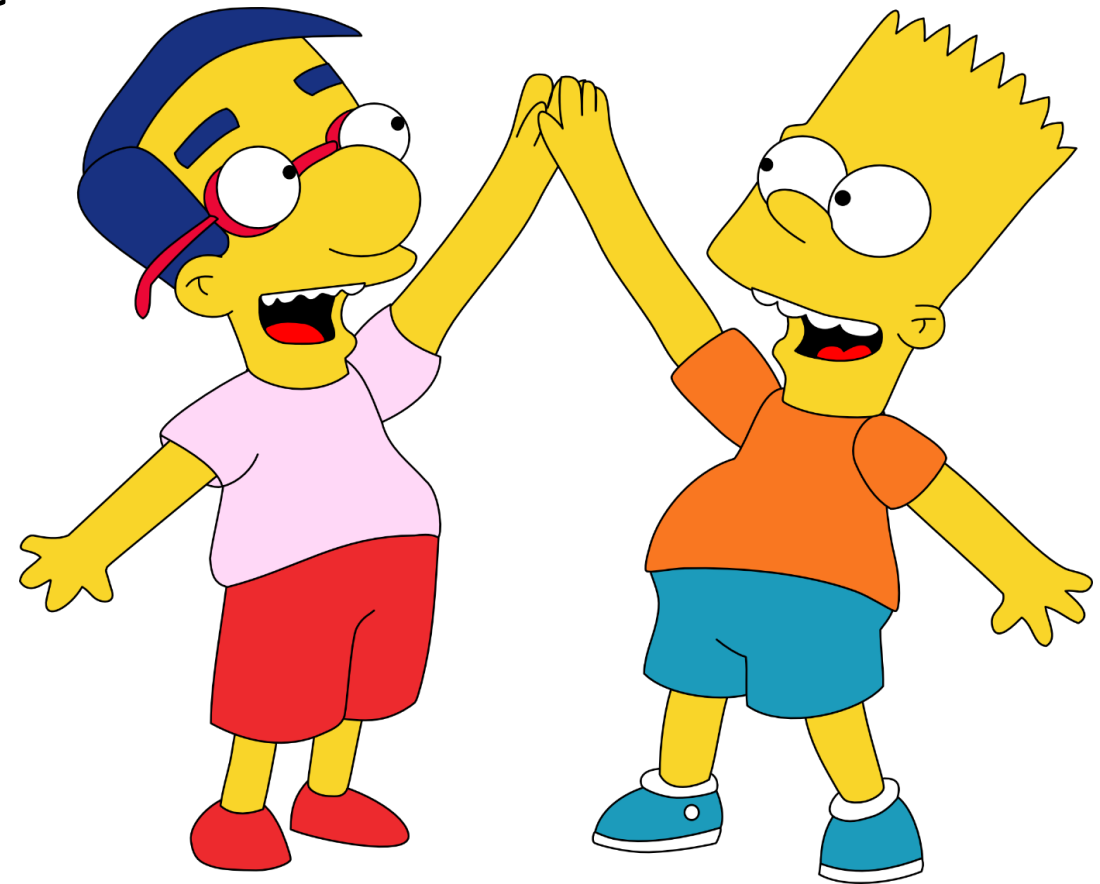
Code Reviews in Branch-based

- There isn't one unique universal way for this, it depends on what you are versioning
- Regression testing is your friend
 - If you have a full set of regression tests covering your code base, no regression = no issues and you can safely merge
- If you aren't happy with the quality of the code, no issues: the branch is independent, nothing will be impacted by keeping it there and work on it again
- Somebody else can easily take over a feature and keep working on the branch

```
mirror_mod = modifier_ob.  
set mirror object to mirror  
mirror_mod.mirror_object  
operation == "MIRROR_X":  
mirror_mod.use_x = True  
mirror_mod.use_y = False  
mirror_mod.use_z = False  
operation == "MIRROR_Y":  
mirror_mod.use_x = False  
mirror_mod.use_y = True  
mirror_mod.use_z = False  
operation == "MIRROR_Z":  
mirror_mod.use_x = False  
mirror_mod.use_y = False  
mirror_mod.use_z = True  
selection at the end -add  
mirror_ob.select= 1  
modifier_ob.select=1  
context.scene.objects.active  
("Selected" + str(modifier_ob.  
mirror_ob.select = 0  
= bpy.context.selected_object  
data.objects[one.name].select  
print("please select exactly  
-- OPERATOR CLASSES ----  
types.Operator):  
X mirror to the selected  
object.mirror_mirror_x"  
mirror X"  
context):  
context.active_object is not
```

Merging can be your friend!

- Again ... It all depends on what you are versioning and your processes
 - Not everybody is versioning just “some code”
- The better your tests, the more you can “blindly” merge
 - Of course it requires testing before and after, but is this a bad thing?
 - If you have full coverage with your tests, tests are successful, there is little reason to believe something will break in an unexpected way



Branching for
real is better
than
“virtually”

- Why branching by abstraction instead of a real branch?
 - No need of any “_v2” or other weird name when replacing something
 - Who do not love a Big Bang release?
- Having a business sign off on a branch is easier than on ... nothing



4 Steps in a Project Lifecycle

1

Start project

2

Review code

3


**Build an
integration
pipeline**

4

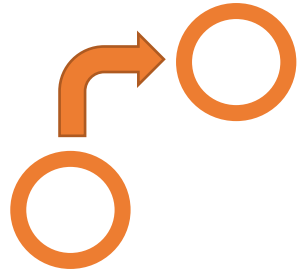
Ship feature

A large orange circle on the left side of the slide, partially cut off by the edge.

continuous integration best practices

- Don't commit broken code
 - Don't commit untested code
 - If the build is broken, fix it before you continue developing
 - Don't go home if the build is broken
 - Commit frequently (at least once a day)
 - Every commit must trigger the integration
 - Keep the build fast (to get fast feedback)
- 
- A decorative yellow dashed line in the bottom right corner, consisting of several curved segments.


Building an integration pipeline trunk-based!



On Merge Request



On Commit Main

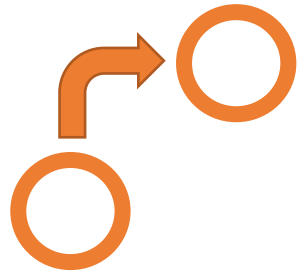
The slide features decorative curved lines in shades of green and blue. One set of lines is in the top-left corner, curving downwards and to the right. Another set is in the bottom-right corner, curving upwards and to the left. The text is centered in the middle of the slide.

But ... if you have a high test coverage, it means your code quality is good, right?

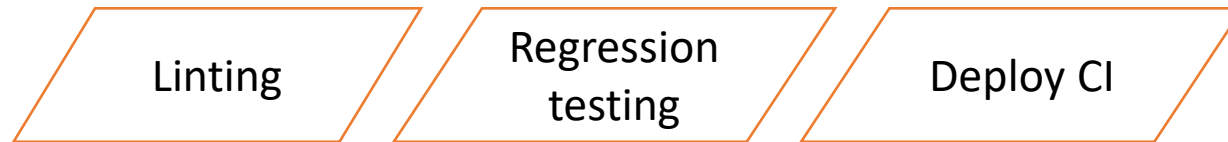
... only because you have a high test coverage,
it doesn't mean your tests are any good....



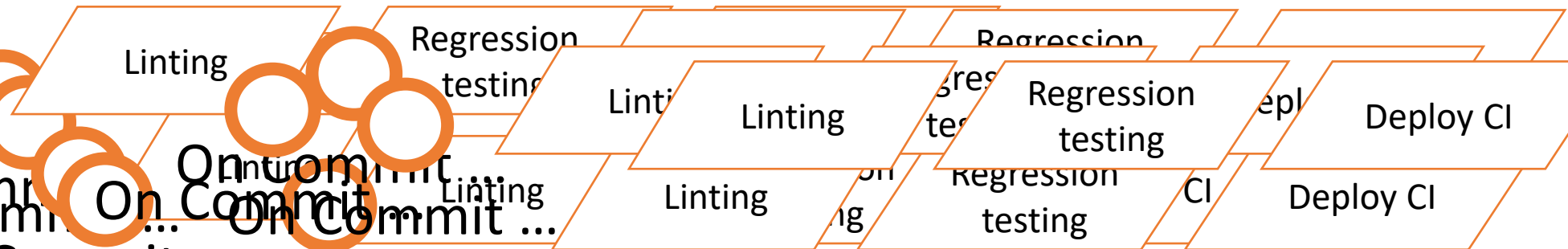
Building an integration pipeline branch-based!



On Merge Request



On Commit Main



On Commit ...

On Commit ...

On Commit ...

On Commit ...

On Commit ...

On Commit ...

On Commit ...

On Commit ...

On Commit ...

On Commit ...

On Commit ...

On Commit ...

On Commit ...

On Commit ...

On Commit ...

On Commit ...

On Commit ...

On Commit ...

On Commit ...

On Commit ...

Building an integration pipeline branch- based!

A bit more seriously...

- Having various kinds of branches requires different/multiple integration pipelines
- A solid naming convention allows to make it extremely simple
 - A pipeline can be linked to a branch by a prefix (like “feature-”, “release-” etc.)
- Integration pipelines can be custom tailored for some special needs applying only to a given branch
 - Warning: More customization == more work to implement it and maintain it

Building an integration pipeline branch-based!

An example of a GitLab CI/CD pipeline definition

- A condition link a job only with branches matching a name condition

```
70 validate_rpd:
71   stage: test
72   tags:
73     - docker
74   image: 192.168.120.80:4567/datalysis/obiee-docker-images:12.2.1.2.0_slim_conf
75   script:
76     # run RPD validation
77     - BI_CONFIG_DOMAINE_NAME=bi
78     - export BI_CONFIG_DOMAINE_NAME
79     - $DOMAIN_HOME/$BI_CONFIG_DOMAINE_NAME/bitools/bin/validaterpd.sh -P Admin123 -R $CI_PROJECT_DIR/rpd/repository.rpd -O $CI_PROJECT_DIR/validate_rpd.txt
80   artifacts:
81     paths:
82       - validate_rpd.txt
83   dependencies: []
84   only:
85     - main
86     - /^dev.*$/
```

regular expression matching branch name

4 Steps in a Project Lifecycle

1

Start project

2

Review code

3

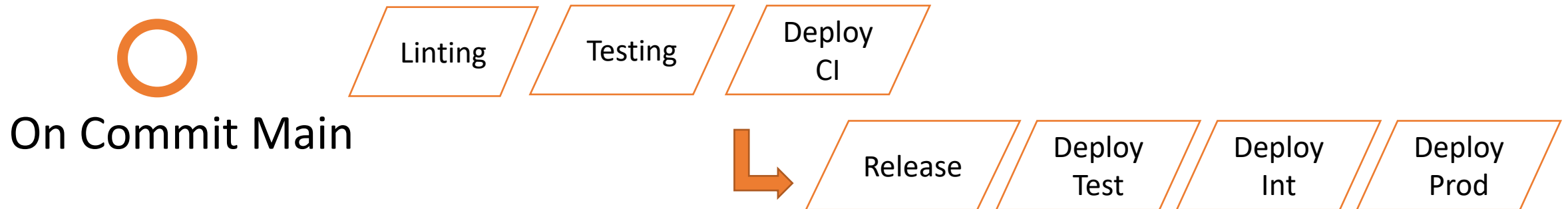
**Build an
integration
pipeline**

4

Ship feature

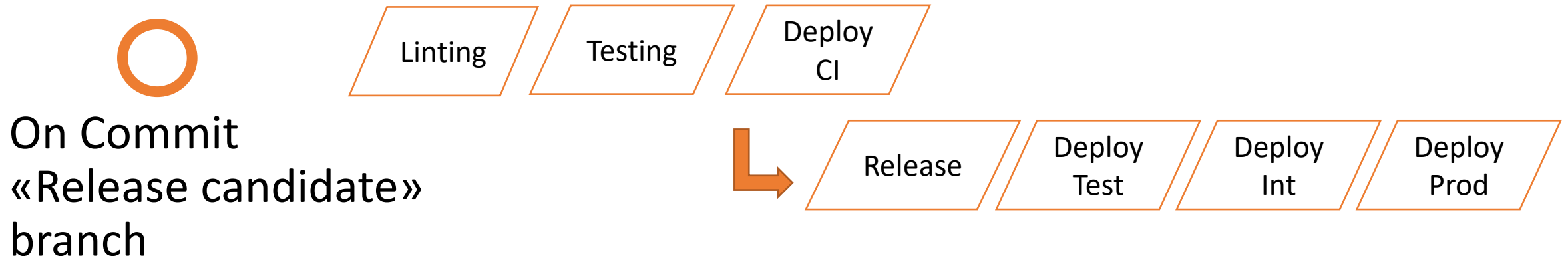
Shipping features into prod trunk-based!

- The integration pipeline is extended with deployment steps
- Those steps can be triggered manually (so not every commit is automatically pushed into prod).

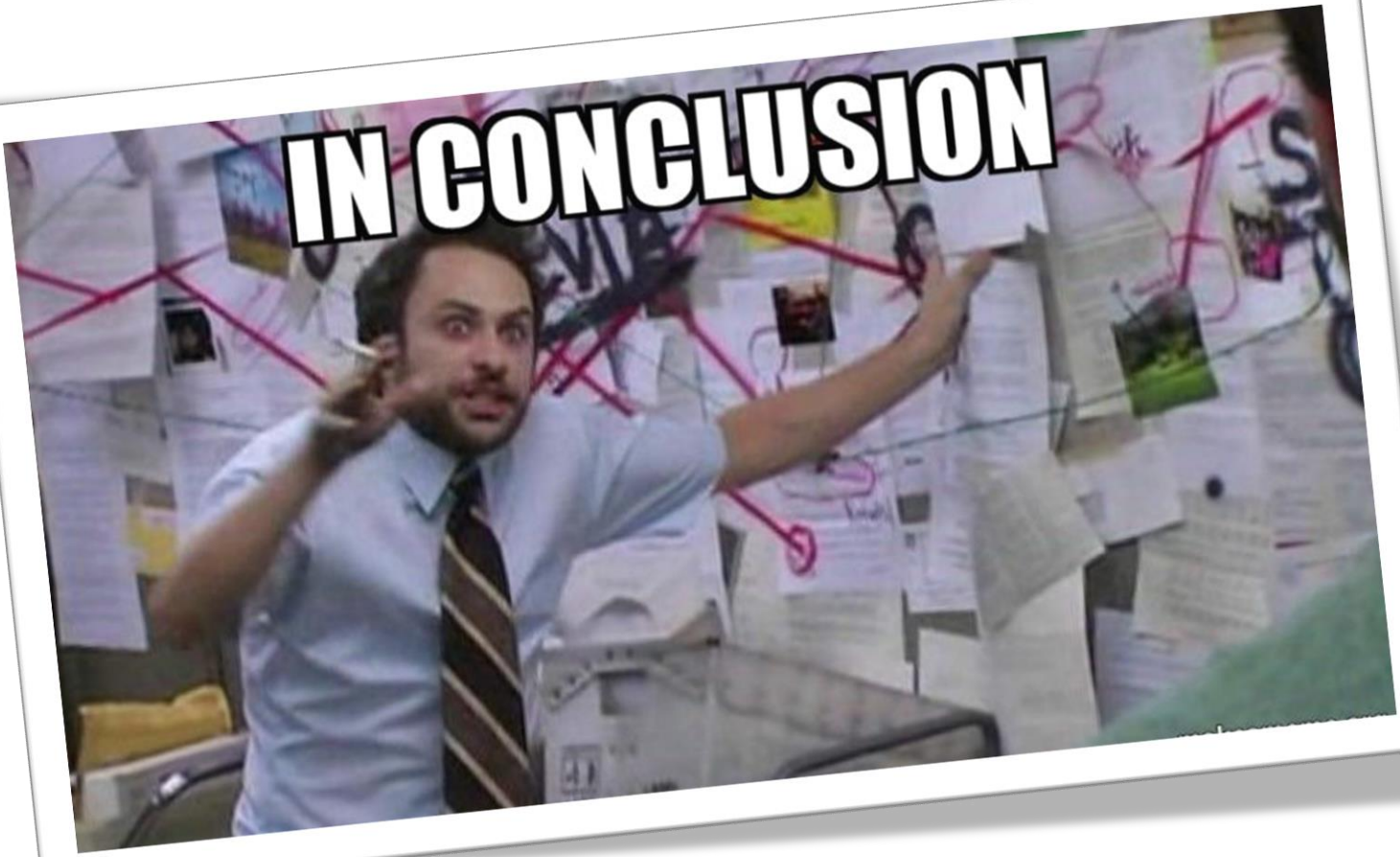


Shipping features into prod branch-based!

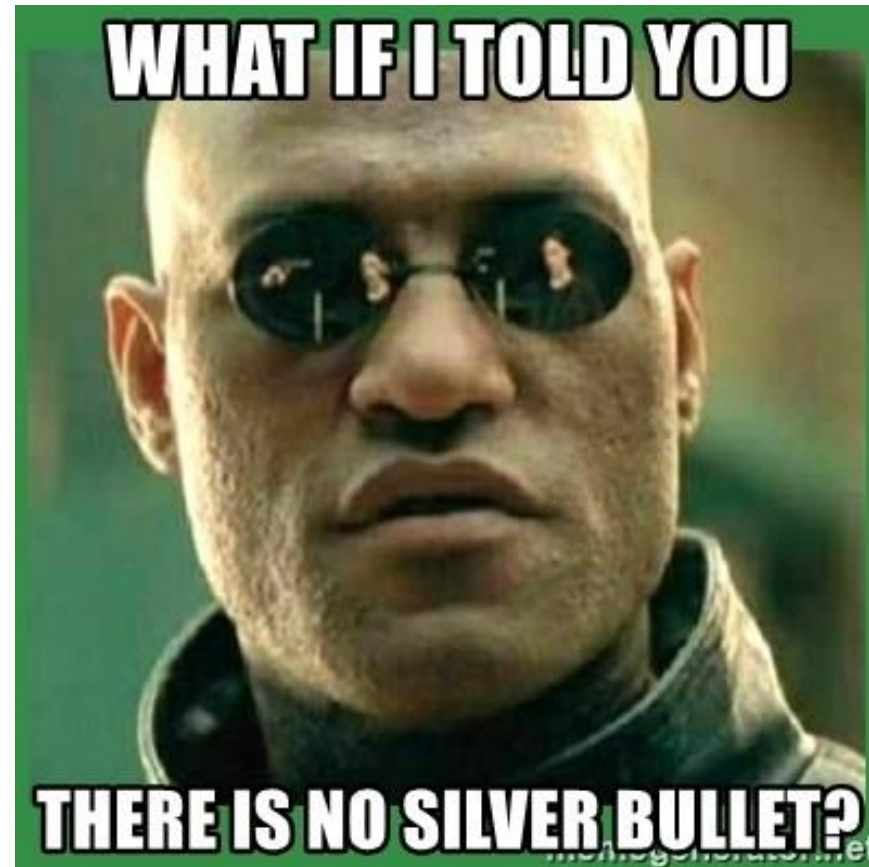
- Just what she said ...
- But applied to a branch with the release candidate, every time there is one
- Doing all the required changes in that branch, and some more cooking
- Merge back into Main the changes, if any



IN CONCLUSION



Choose the git strategy that fits your project best! → You might need to adjust one.

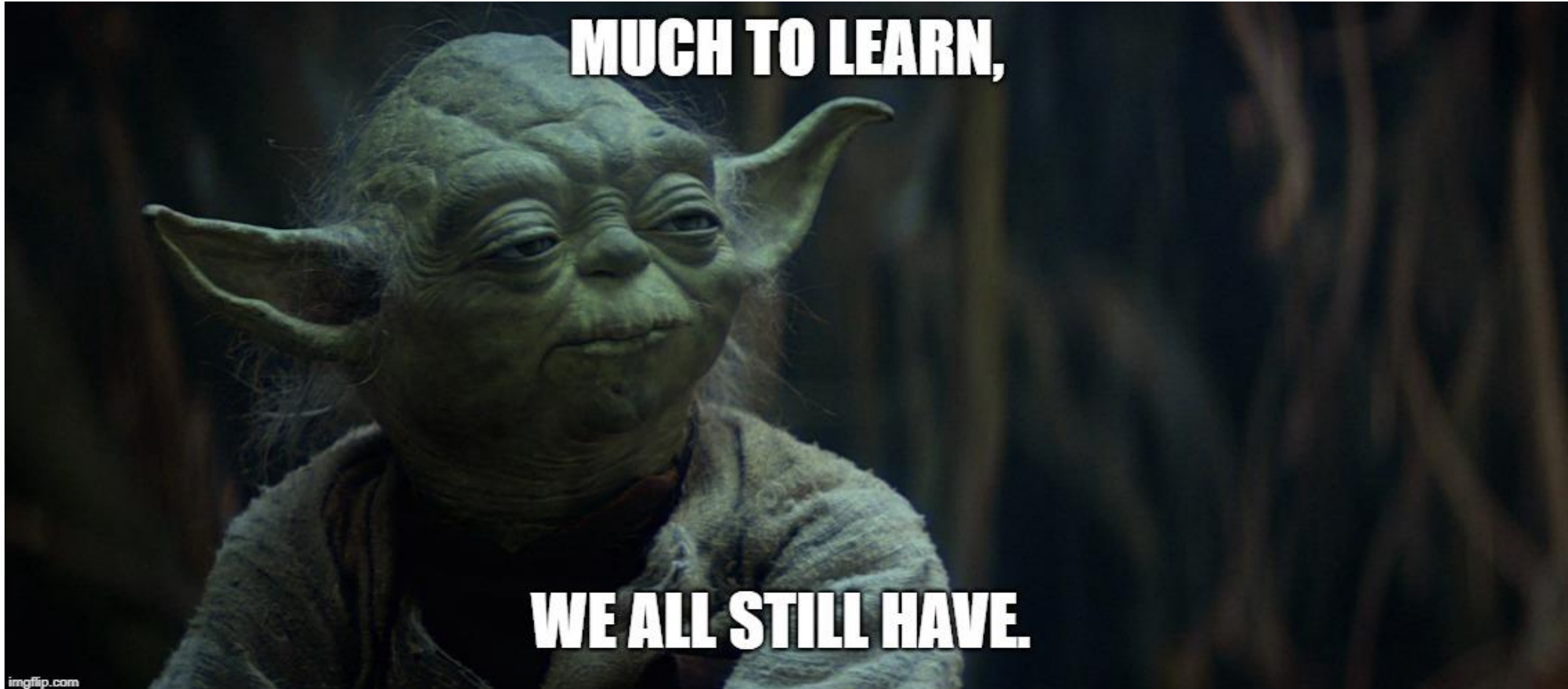




You can stay out of trouble if you plan your tasks well!

- No overlapping tasks with other team members
- Clear architecture that allows to do work in parallel
- Clear interfaces that separate components

There are many good git branching strategies available! → Know them!



Use Git and its features how they are supposed to be used! As a versioning tool!



 @jasminfluri

 @G_Ceresa

What questions do you have?

Thank you for your time!



References / Sources

- <https://trunkbaseddevelopment.com/>
- <https://nvie.com/posts/a-successful-git-branching-model/>