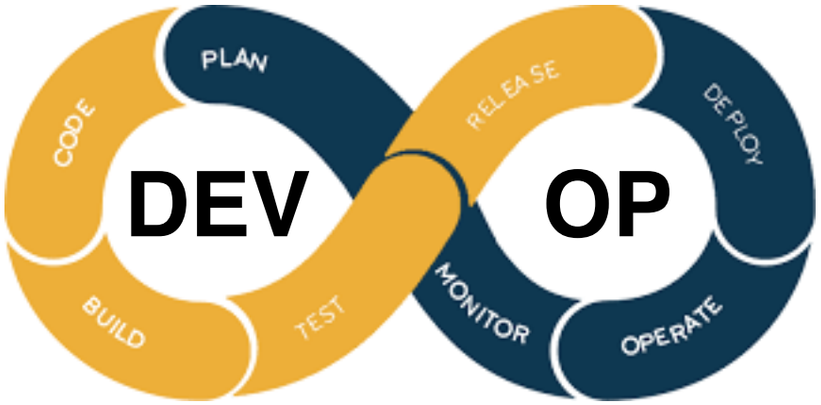
UNIVERSITAT DE BARCELONA

Continuous…



Design
Testing
Integration
Delivery/Deployment

by Eloi Puertas

What is **Continuous Design**

- Create and Modify the design of a system as it is developed.

- Continuous Design and Development implies:

- **TDD** (Test Driven Development)

- **Refactoring**

What is **Continuous Testing**

- Process of executing **automated tests** as part of the Agile sofware development process. (Self-Testing Code)

- Provides fast and continuous feedback

- Continuous Testing includes the validation of both

- **functional requirements:** unit Test, API testing, Integration Testing, System Testing…

- **non-functional requirements:** static code analysis, security testing, performance testing…

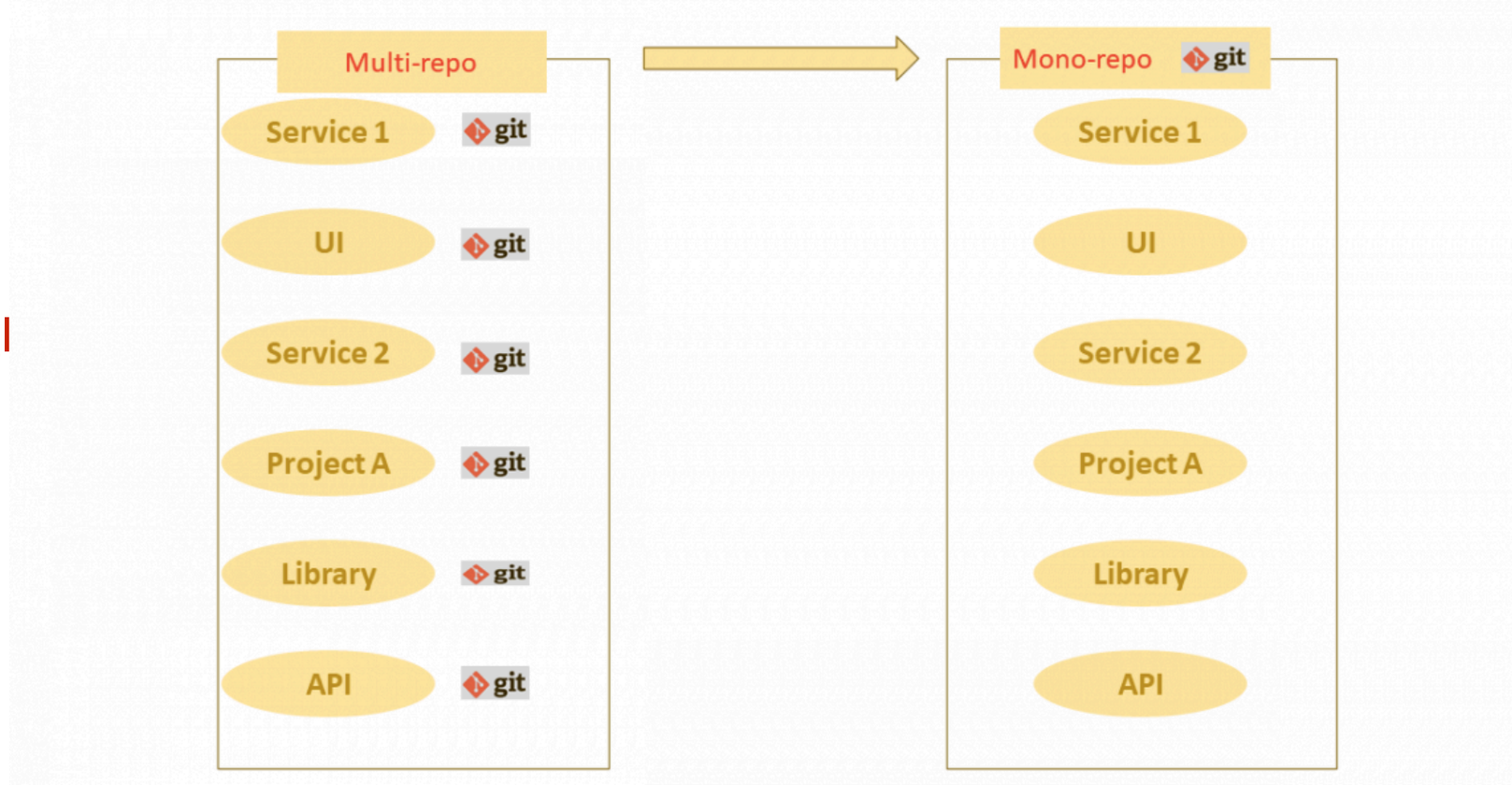
Repo Strategy

- Mono-repo

Companies like **Facebook**, **Googl**
and **Dropbox** use **mono-repo**.

- Multi-repo

Companies like Netflix
and **Amazon** use **multi-repo**.



Mono-repo favors consistency, whereas multi-repo focuses on decoupling

https://geekflare.com/code-repository-strategies/

Mono-Repo

+ A single place to store all the project code, and can be accessed by everyone on the team
+ Easy to reuse and share code, collaborate with the team
+ Easy to understand the impact of your change on the entire project
+ Best option for code-refactoring and large changes to code
+ Team members can get an overall view of the entire project
+ Easy to manage dependencies
- Performance: operations might become slow

Multi-Repo

+ Each service and library have its own versioning
+ Code check-outs and pulls are small and separate, thus there are no performance issues even if the project size grows
+ Teams can work independently and need not have access to the entire codebase
+ Faster development and flexibility
+ Each service can be released separately and have its deployment cycle, thus making CI and CD easier to implement
+ Better access control. All teams don't need to have full access to all the libraries, but they can get read access if they need
- The dependencies and libraries used across services and projects have to be regularly synced to get the latest version
- Encourages a siloed culture at some point, leading to duplicate code and individual teams trying to resolve the same problem
- Each team may follow a different set of best practices for their code causing difficulties in following common best practices
- Running end-to-end tests can be hard

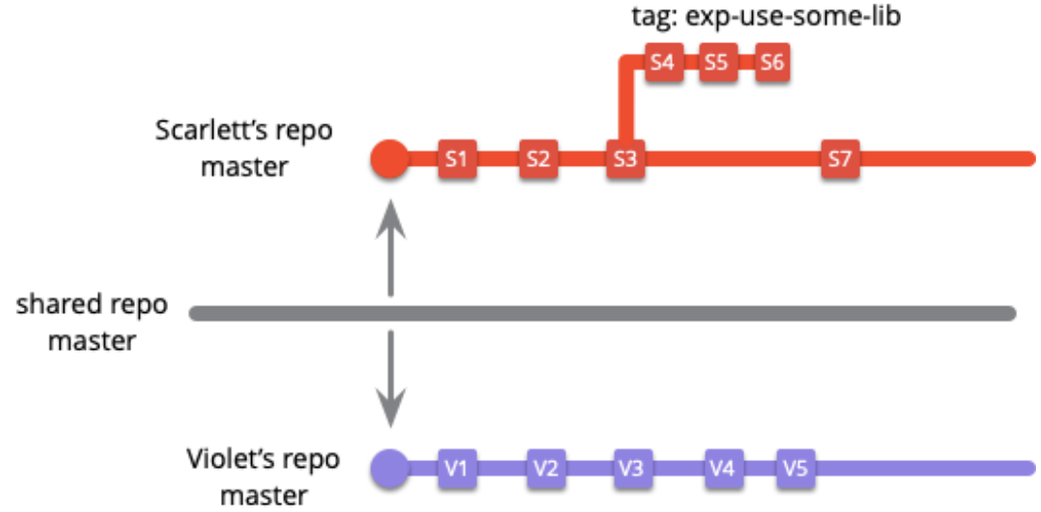
meta: A tool for managing multi-project

Branching and Code Integration

- *Source Branching:* Create a copy and record all changes to that copy. Branch per person, to avoid conflicts.

- *Mainline:* A single, shared, branch that acts as the current state of the product

- *Healthy Branch:* On each commit, perform automated checks, usually building and running tests, to ensure there are no defects on the branch

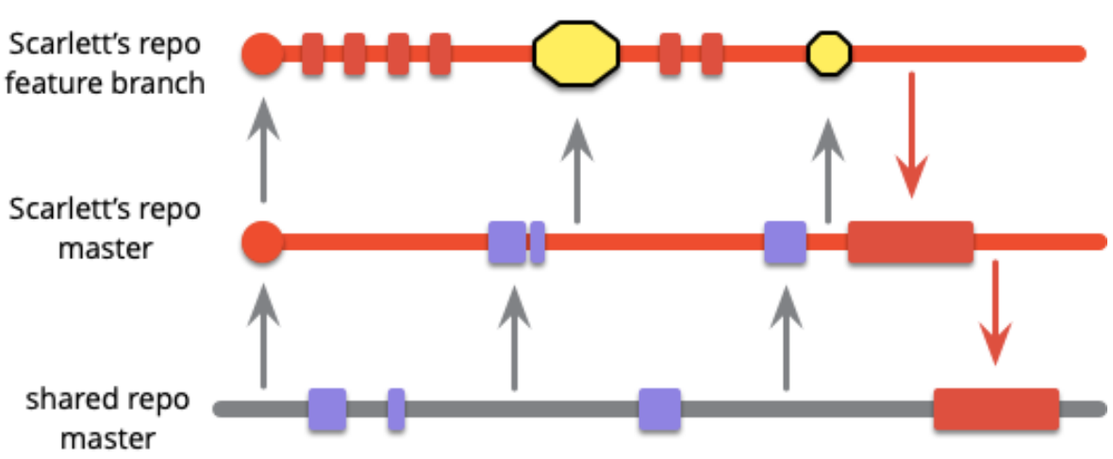


branching is easy, merging is harder.

https://martinfowler.com/articles/branching-patterns.html#integration-patterns

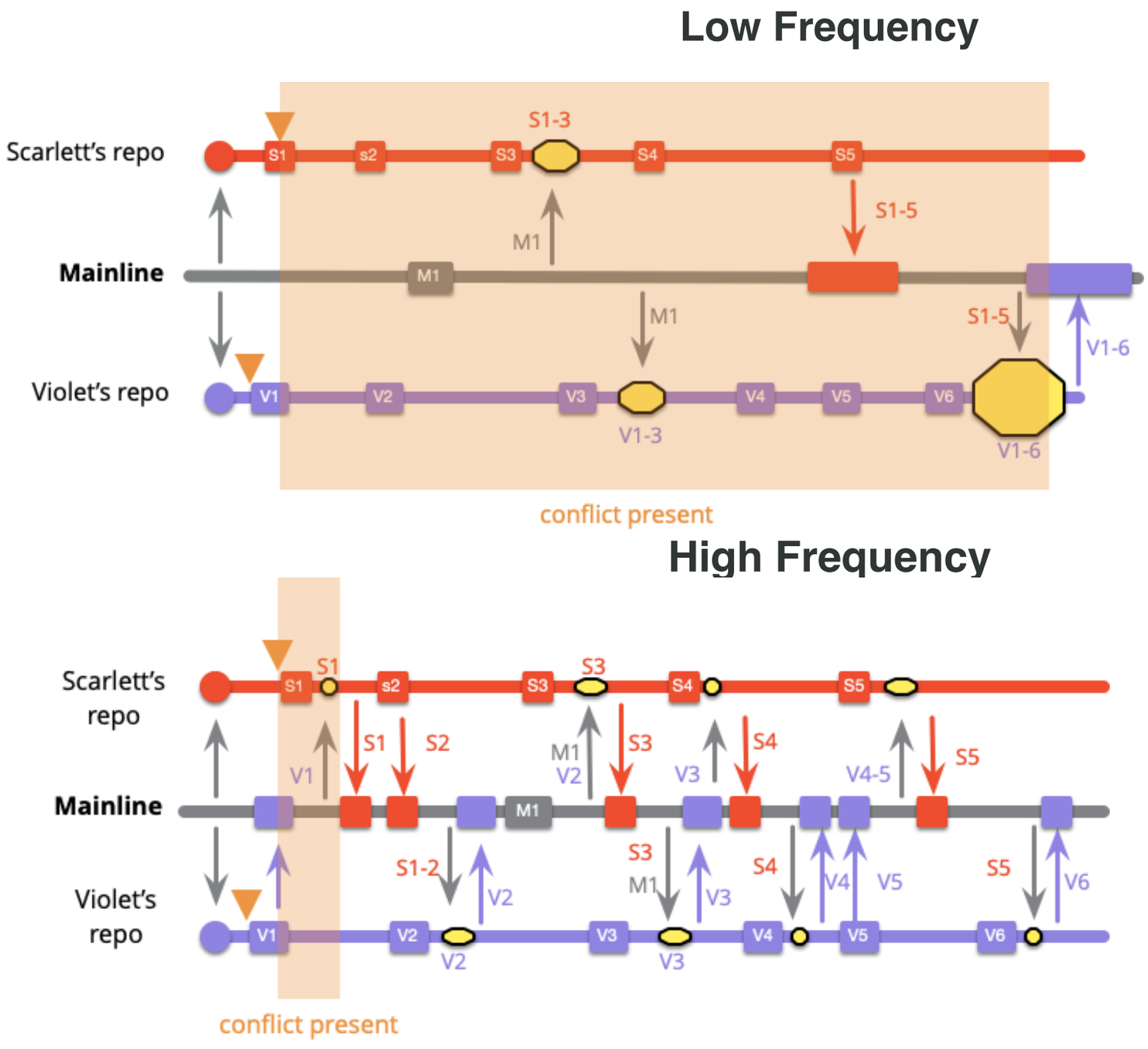
Feature Branching

- *Feature Branching*. Put all work for a feature on its branch, and integrate it into the mainline when the feature is complete.



https://martinfowler.com/articles/branching-patterns.html#integration-patterns

Integration Frequency



- Frequent integration increases the frequency of merges but reduces their complexity and risk.

https://martinfowler.com/articles/branching-patterns.html#integration-patterns

What is **Continuous Integration (CI)**

- Developers do mainline integration as soon as they have a healthy commit they can share, usually less than a day's work, reaching frequent integration points with a partially built feature.
- To hide partially built feature we can use *feature flags*
- The difference between **feature branching** and **continuous integration** isn't whether or not there's a feature branch, but when developers integrate with the mainline.

Feature Branching vs Continous Integration

Feature Branching (pull request)

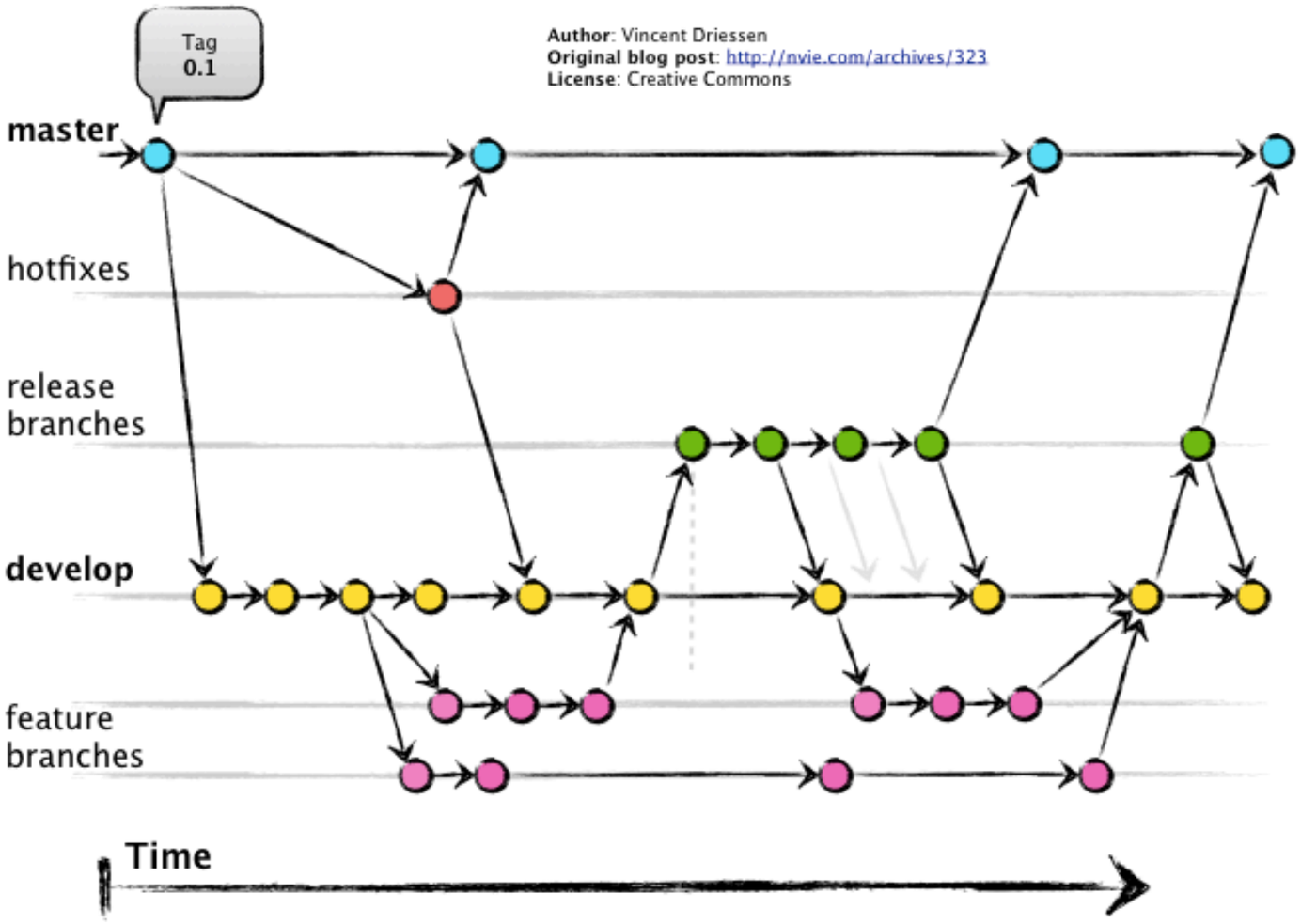
- All the code in a feature can be assessed for quality as a unit
- Feature code only added to product when feature is complete
- Less frequent merges

Continuous Integration (github actions)

- Supports higher frequency integration than feature length
- Reduced time to find conflicts
- Smaller merges
- Encourages refactoring
- Requires commitment to healthy branches (and thus self-testing code)
- Scientific evidence that it contributes to higher software delivery performance

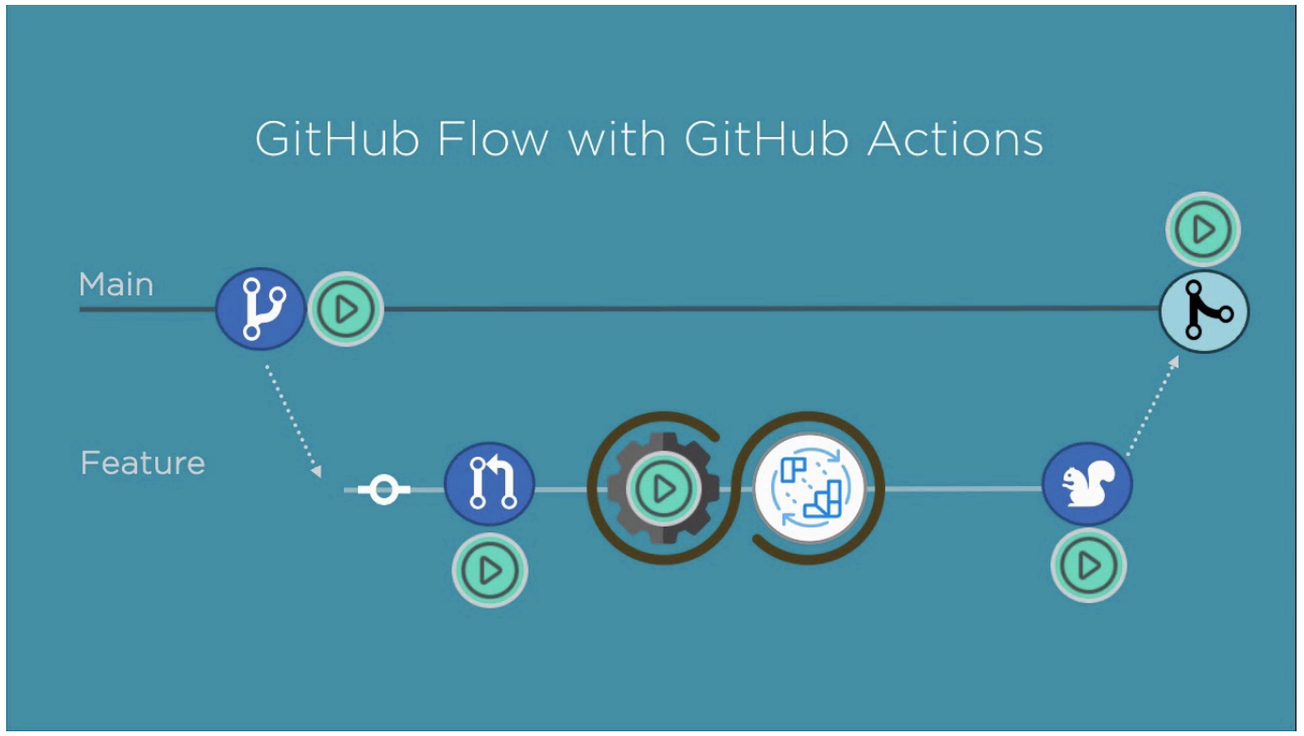
Branching Workflows and policies

GitFlow Workflow



Branching Workflows and policies

GitHub Workflow



Workflow:

- Run tests locally: run and pass all **unit tests**
- Compile code in CI: After every commit compile in the build server
- Run tests in CI: Run test units, integration tests, static analyses, profile performance… in the builder server
- Ready for Deploy an artifact from CI: the code in the builder server is ready for CD

What is **Continuous Delivery/Deployment (CD)**

- **Continuous Delivery**: Makes sure the software checked in on the **“dev”** line is always in a state that can be deployed.

- **Continuous Deployment:** Makes the deployment process fully automated.

Release Frequency

Including a deployment stage before release that utilizes a realistic testing environment

- **Blue/green deployment:** Set up two identical environments, with only one going live at a given time. Initially roll out new releases to the offline environment, and, if successful, switch to the new environment, and the original production environment becomes idle. The first environment provides a backup, allowing you to switch back if there is an issue with the new release.

- **Canary Release:** Release a new update to a subset of users to test it in a limited, real-world setting. If successful, you can roll out the deployment to a wider user base. If unsuccessful, you can roll back the release.

- **Deployment with A/B testing.** Deploys different versions of a feature to verify performance and usability. It is primarily used to review the effectiveness of a change and how the market reacts to the change.

CI/CD Best Practices

- Maintain one code repository
- Automate the build
- Make the build self-testing
- Everyone integrates to the baseline every day
- Every merge (to baseline) should be built
- Every bug-fix commit should come with a test case
- Keep the build fast
- Test in a clone of the production environment
- Make it easy to get the latest deliverables
- Everyone can see the results of the latest buid
- Implement Continuous Delivery first. After, to automate deployment
- Implement progressive delivery (feature flags)

Cloud Services for CI/CD

SaaS: Software as a Service. Web Application

PaaS: Platform as a Service. Web App for Deployment.

- (**dynamic website**): Heroku, Dokku (**static website**): Netlify, Vercel.

IaaS: Infrastructure as a Service. Web App for Infrastructure Management.

- AWS, Azure, Google Cloud, DigitalOcean

CI/CD: CI: Github Actions / CD: Azure

GitHub & Azure tutorials

- Understanding GitHub Actions

- GitHub Actions for Continous Integration

  Build process: Compile & Linter & tests

- GitHub Actions for Continous Deployment on Azure

  Deployment process: Dockerize App and publish

- GitHub Actions for managing Board flows