# DevOps

## Introduction

### What is DevOps?

DevOps, as a word, is the combination of *development* and *operations*, where were historically the people responsible for building the software and the IT operations team(s) were responsible for deployment and infrastructure management. Sometimes there is also a further separation of testing and quality assurance (QA) teams. Basically, DevOps is a new paradigm of software development based on the idea that you cannot and should not separate development and IT operations if you want to be competitive because such a separation causes major issues, such as bad deployments that break previously working software.

### Why DevOps?

DevOps is the next phase in the evolution of the software industry beyond Agile. Like Agile, DevOps is concerned with increasing the velocity of deploying valuable and working code to customers, and getting feedback from the market as quickly as possible. Overall, DevOps is a complementary set of principles, attitudes, practices, processes and tools. The highest goal for any software technology organisation: **evolve and improve your product as quickly as possible through real customer feedback**. All the tools and methods in DevOps serve this highest goal by removing obstacles to the team achieving high iteration velocity. In my opinion, all modern software development teams that do not implement DevOps will - because of market pressure - either (1) eventually implement DevOps or (2) be beaten by their competitors and fail.

### The Basics

**Cultural philosophies**: some of the many

- **Communication**: developers need to communicate as much as possible and as effectively as possible.
- **Visibility**: we can see each other's work and help each other out if there are problems.
- **Error correction**: assume code will break and (1) practice defensive programming through TDD to minimise it and (2) make it easy to undo breaking code, rollback the broken version, then identify and fix the issue.
- **Continuous feedback**: by releasing working code into the production (or production-like) environment, we can get real and meaningful feedback quickly and adjust course as we go.

**Practices**:

- **Version control**: we all use version control across all our artefacts (code, design, documentation) so that we can rollback if there is a breaking change.
- **Peer review**: code is checked by one of your team mates before it is integrated (i.e. pull requests need to be checked by a team mate) and we make time to do code reviews.
- **CI/CD**: continuous integration and continuous deployment. In a nutshell, this means that individuals should push code every single day (that they are developing), getting it to pass all the automated tests and integrating it into `**master**`. Don't let your code build up and do a big push. Smaller pushes means that if the code doesn't pass it is easier to figure out why and then fix it.
- **Env sync**: environments are managed and synced so that everyone is using the same environment for development, and staging is as close to production as possible.
- **Testing culture**: testing is done from the beginning of development. Everyone writes tests for their own code.
- **Automate as much as possible**: unit and integration tests are automated so that we can push code quickly and know that it is not regressing the quality of the code base. Deployment is automated as much as possible, except in special cases where the "gate" needs to manually monitored.

**Tools**:

- **Standardised tooling**: everyone uses standard tools and software versions so that there is as little risk of clashes and mismatches in our workflow.
- **Automation server(s)**: testing and deployment is managed by a specialised server (or service), such as Jenkins (or Github Actions).
- **Configuration**: configuration management between the dev, staging and production environments need to be carefully managed.

## Our setup

**Mapping teams to architecture**: our architecture setup has three main parts (at the time of writing the specifics of the architecture are not decided so languages are just examples).

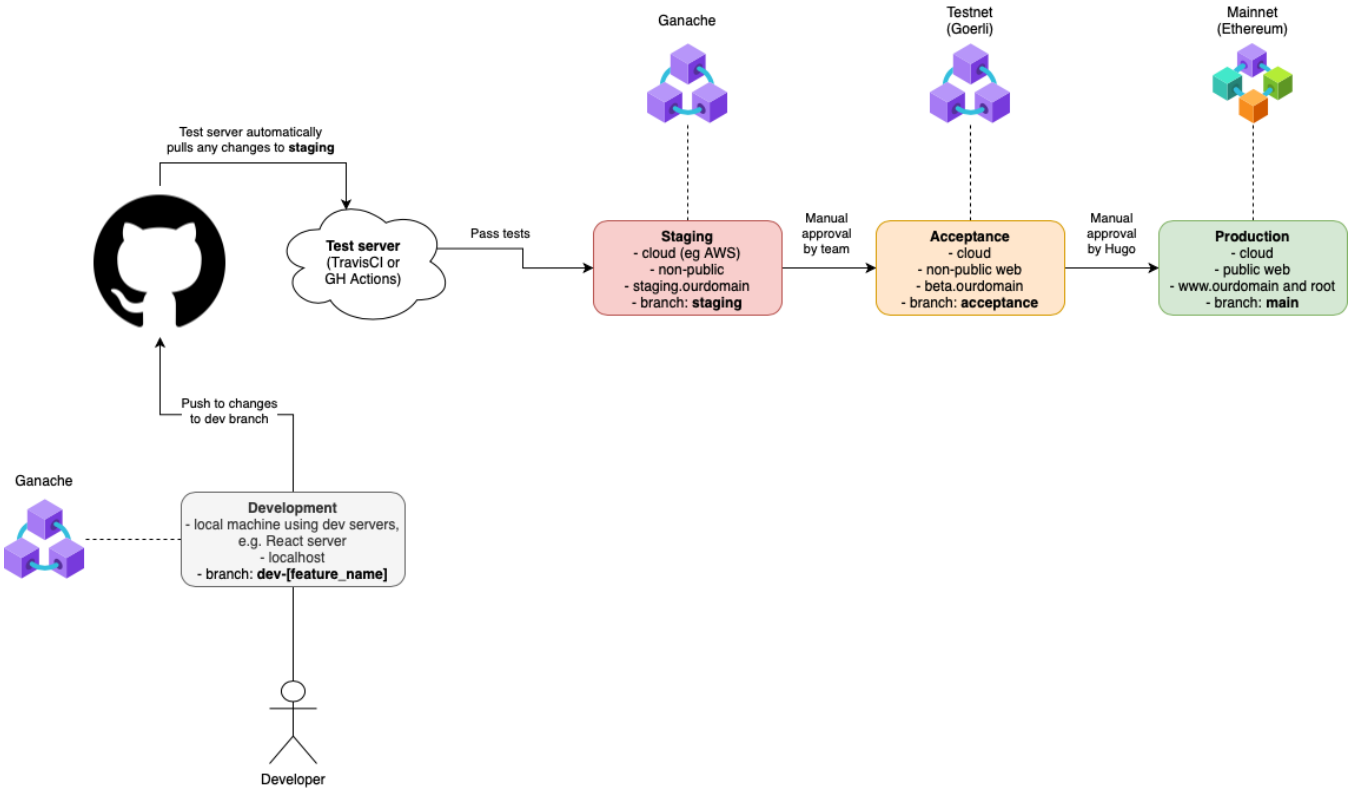
1. Smart Contract (Solidity)
2. Frontend for the client UI (Vue or React)
3. Backend for admin dashboard, API, contract oracle (Python Django or Node Express)

**How many Github repos?** We will have one (1) Github repo. The first highest level division of directories will be: `frontend`, `smart_contract`, and `backend`.

```
root
 |- frontend
 |- backend
 |- smart_contract
 |- git
 |- .gitignore
```

Thus, we will have three sub-teams (that can change throughout the year) working on a single git repo, but the repo will be divided at the highest level into a directory for each team. This will mean that as teams push code, they will all push to the same **master** branch.

### Diagram 1: high-level deployment pipeline




**Notes**:

- Development environment is essentially just your local computer
- Pushing to your feature branch will not trigger the test server. However, you should merge to the Staging whenever you finish developing for a day to make sure you're leaving your code in a working state.
    - Merging a branch into staging will trigger the test server to run the automated test suite before letting it merge. If this doesn't pass then you should fix it before going AFK for the day.
- We can play around with the release in Staging
- Staging and Acceptance will be clones of Production (as much as possible).
- Updating Acceptance environment will require manual approval from team.
- Acceptance environment is for Hugo to review our work himself
- Hugo will manually approve the app in the Acceptance to go to the Production environment
- Only the app in the Production environment will have (a) any real user data and (b) connection to the Ethereum mainnet

## Key technologies

Please start looking at how to use the following links:

1. Ganache: https://www.trufflesuite.com/ganache
2. Docker: https://www.docker.com
3. Test-Driven Development: https://technologyconversations.com/2013/12/20/test-driven-development-tdd-example-walkthrough/