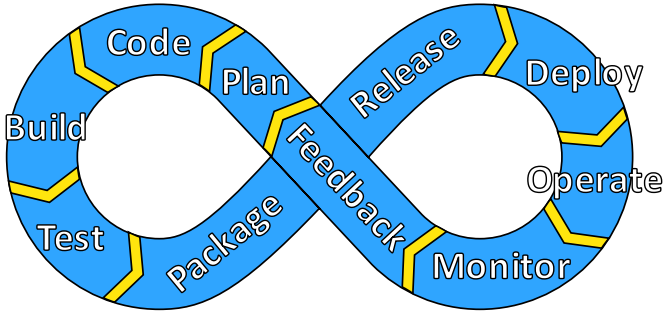
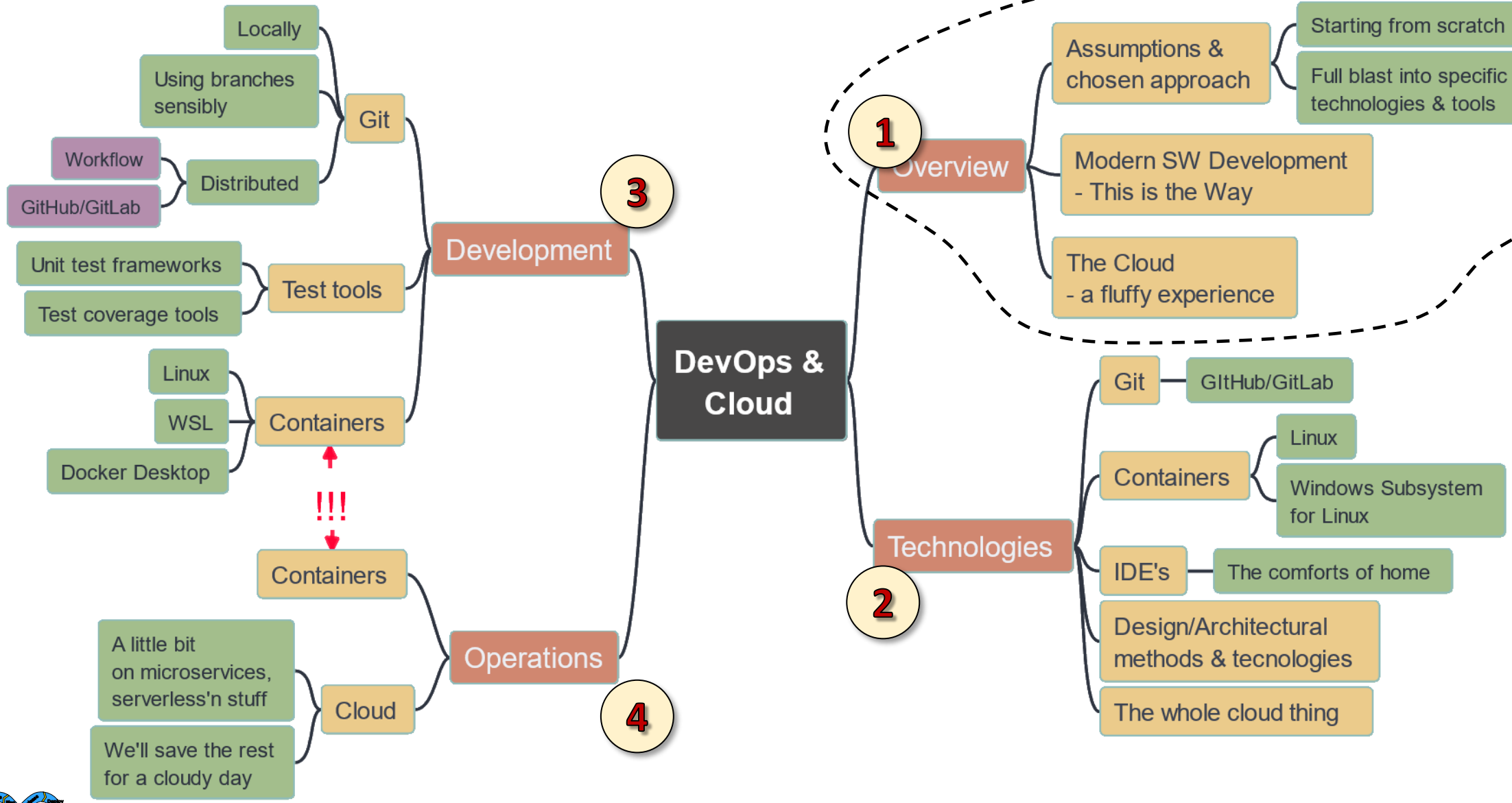


DevOps & Cloud

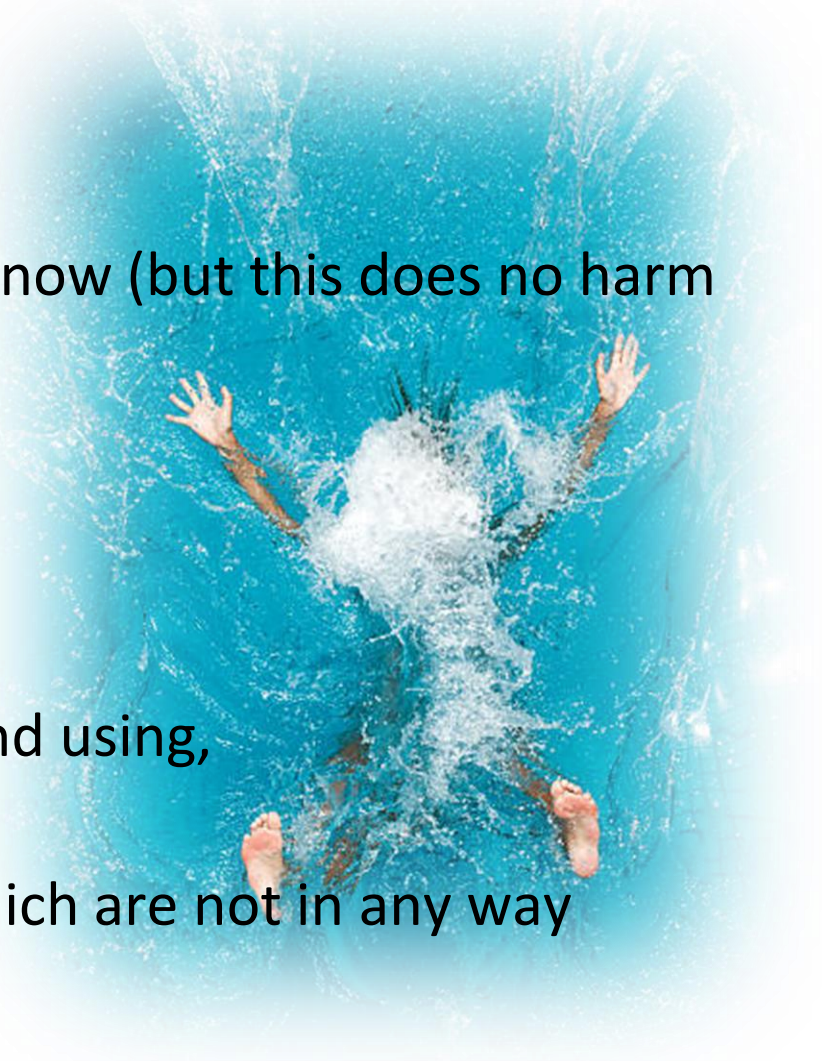
- an introduction

Overview – mind the map

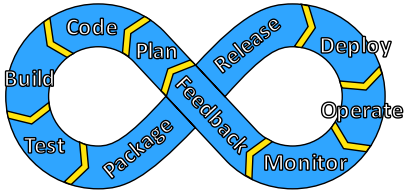


About this mini-course

- Assumptions on my part
  - None – so I may regurgitate some stuff you already know (but this does no harm and it's important anyway, so there ....)
- Approach
  - Some very general introductions
  - Then headfirst into the tools of the trade
  - To get you up to speed thinking and talking about, and using, these tools and technologies as fast as possible
  - To allow you to navigate the terms and concepts (which are not in any way difficult but somewhat fuzzy at times)



DevOps – what is it? (1)



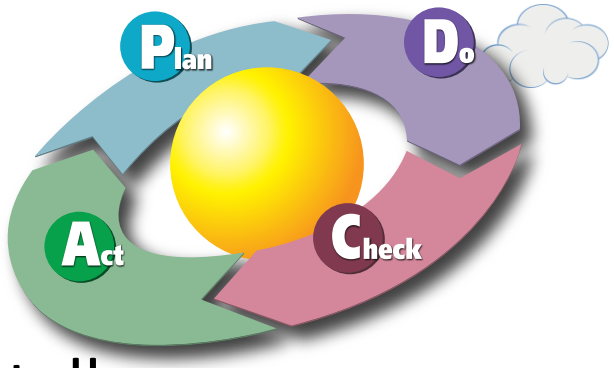
- The short answer: That's the way modern sw development is done
- The slightly longer answer:
  - Modern sw development (SWD) recognizes that SWD is much more than simply writing (good code)
  - We need to look at the full SWD lifecycle, from conception, over development and operations, to the day the sw is scrapped – good sw development processes must facilitate all phases
  - So, the "word" DevOps = **Dev**elopment And **O**peration**s** indicates methods and technologies that cover the full lifecycle:
    - Development = Specify, design, implement, test, package and release software
    - Operations = Putting the sw into operation, maintain, configure, update and monitor software
  - The "DevOps loop" indicates that the parts (Dev & Ops) feed off and into each other

DevOps – what is it? (2)

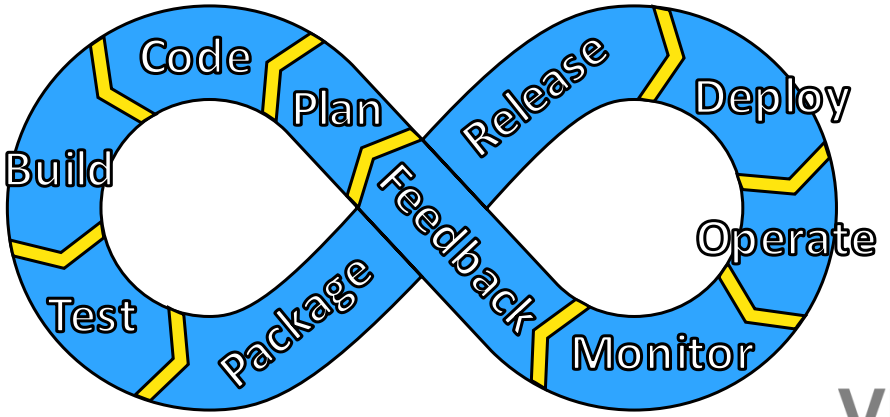
- The whole Ops part obviously depends on the context and type of sw – here's two different scenarios:
  1. Sw that is fully developed and operated internally in a company. The sw is put into immediate use each time it is updated.
  2. Sw that it is developed as an off-the-shelf product. The sw gets updated regularly and official releases made once in while.
- Scenario 1 has the closest coupling between dev & ops, and may benefit the most from DevOps methods and technologies. The ops part provides direct feedback to developers about the quality and usability of the se
- But even the developers in scenario 2 should think about (some of) the operations, especially the update part (nobody likes sw that nukes everything because it gets updated) but also the feedback from users!

So, all sw developers should think about DevOps

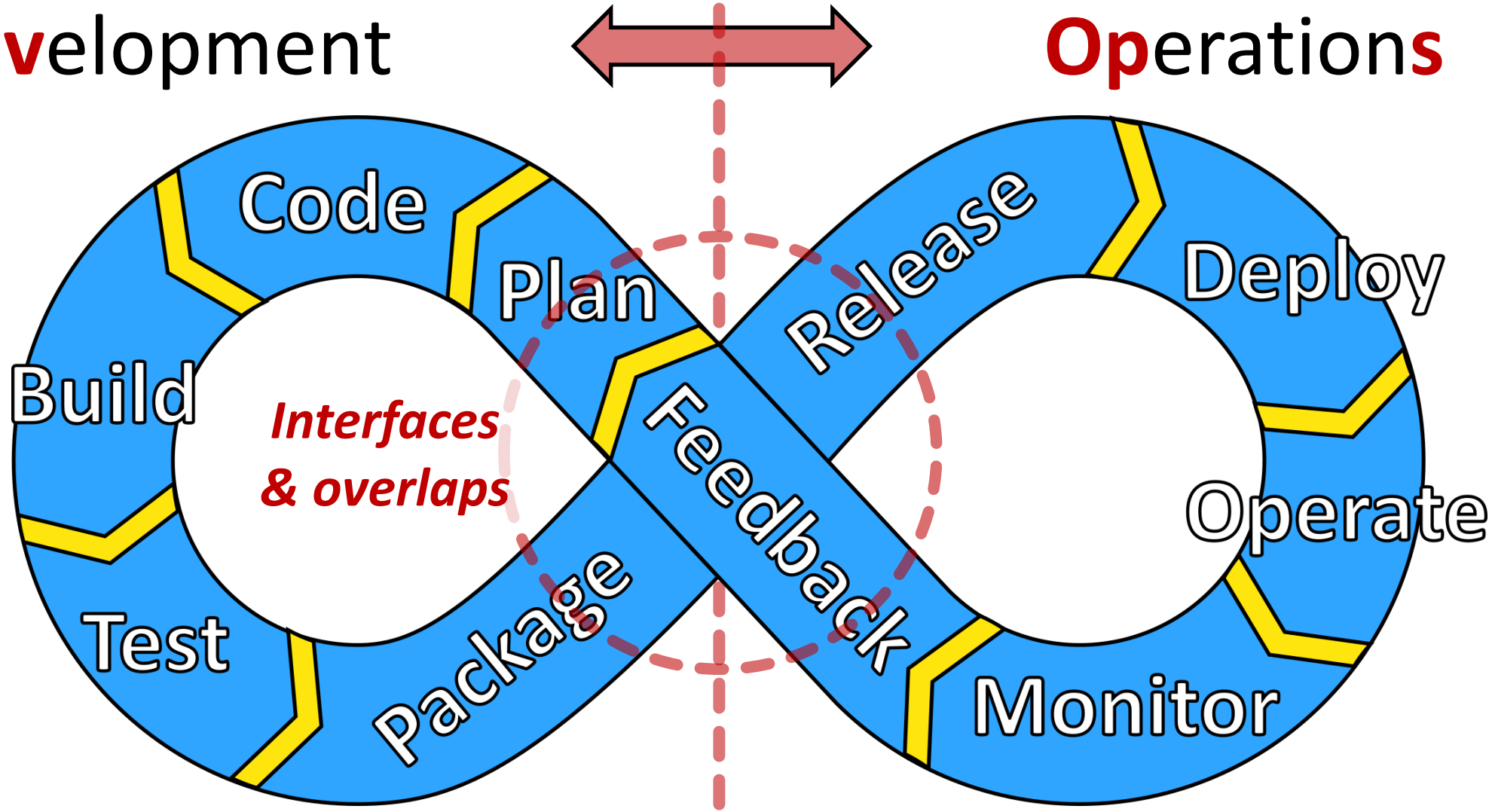
DevOps – what is it? (3)



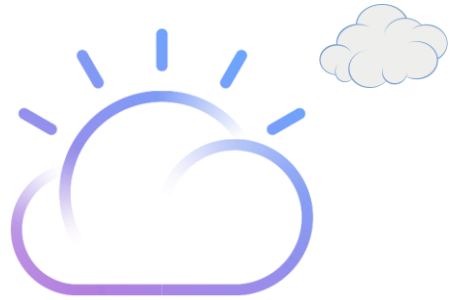
- Modern development processes promotes agile and iterative sw development where the sw is built incrementally in many small specify-design-implement/test-release cycles (aka p-d-c-a)
- We want to continuously integrate new features in the code (**CI** = continuous integration) and make the result immediately available for use (**CD** = continuous delivery/deployment)
- So, we need to be efficient at doing all this specify/design/ implement/ test/release/package stuff
- To do that we have to think about the whole process and not only about writing some code:



The DevOps loop



Cloud – why's that relevant? (1)



- Cloud is just a common designation for a set of technologies
- Cloud is mostly associated with the Ops-part of DevOps (deploying, running and monitoring applications), but many newer development tools also makes good use of cloud technologies
- The "cloud" thing indicates that these technologies are commonly realized on computing resources that are owned by another party, and that a company pay the cloud owner for using those resources based on how much you use them (on-demand, pay-per-use)
- But there's absolutely nothing that prevents a company from using "cloud" technologies on computers bought and owned by the company itself

Cloud – why's that relevant? (2)

- Using a "real cloud" relieves you from buying, maintaining and operating computer hardware which can be costly (money, man-hours, power, cooling etc.)
- But after the initial cloud hype there's a been a trend the last few years of companies moving back to self-owned hardware because it may actually be cheaper, even though you then have to pay for hw, salaries and power
- A mix of on-premise resources and real cloud (aka hybrid cloud) is also becoming common
  - You can use the cloud to quickly prototype new setups and then move it internally when things work
  - You can use your own stack and use the cloud to help handle peak loads

Cloud – why's that relevant? (3)

- Stuff like GDPR also muddies the water when it comes to using the "real" cloud – it may prevent certain customers from using a company's product if the customers have no control of the physical whereabouts of data
- So, all in all, the notion of "cloud" incorporates several ways of using these technologies
  1. Full-on real cloud from providers like Amazon, Azure and Google
  2. Hybrid cloud combining self-owned resources with external "real cloud"
  3. Fully internal "cloud" running 100% on self-owned resources
- But that does not diminish the value of the technologies involved, such as virtualization and network-based storage - the cloud thing is really about making efficient use of computer resources by sharing them between many applications in a flexible and dynamic way

What is DevOps, here's 3 takes - a bit fuzzy, right?

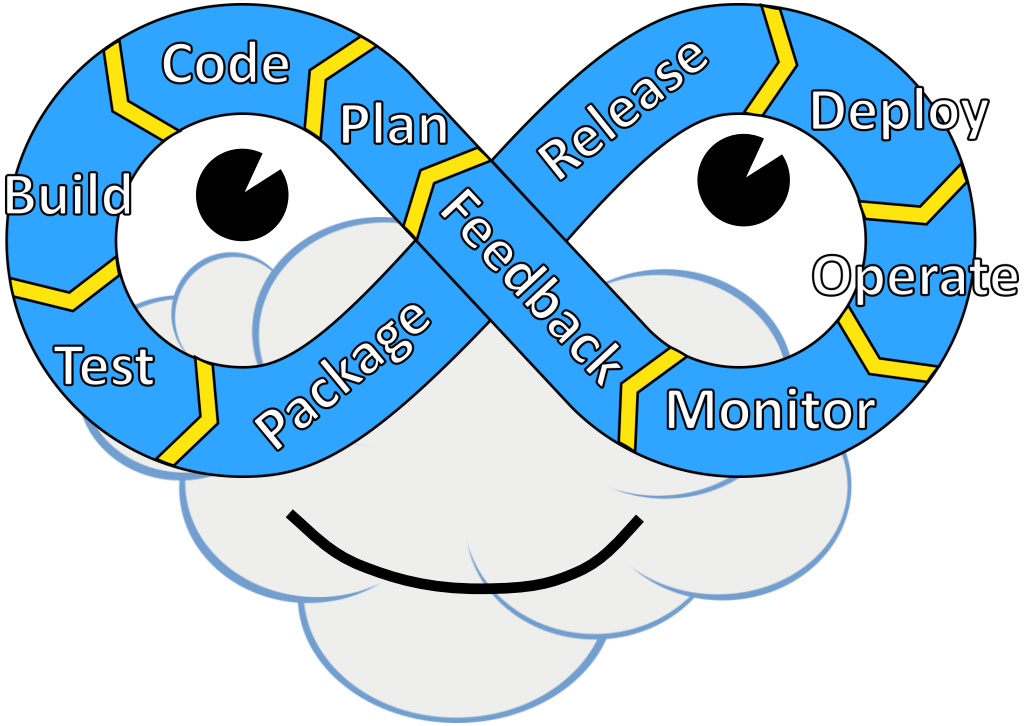
1. *Used as a set of practices and tools, **DevOps** integrates and automates the work of software development (Dev) and IT operations (Ops) as a means for improving and shortening the systems development life cycle.*
2. *Other than it being a cross-functional combination (and a portmanteau) of the terms and concepts for "development" and "operations", academics and practitioners have not developed a universal definition for the term "**DevOps**". Most often, **DevOps** is characterized by key principles: shared ownership, workflow automation, and rapid feedback.*
3. *"a set of practices intended to reduce the time between committing a change to a system and the change being placed into normal production, while ensuring high quality"*

DevOps in summary – our take

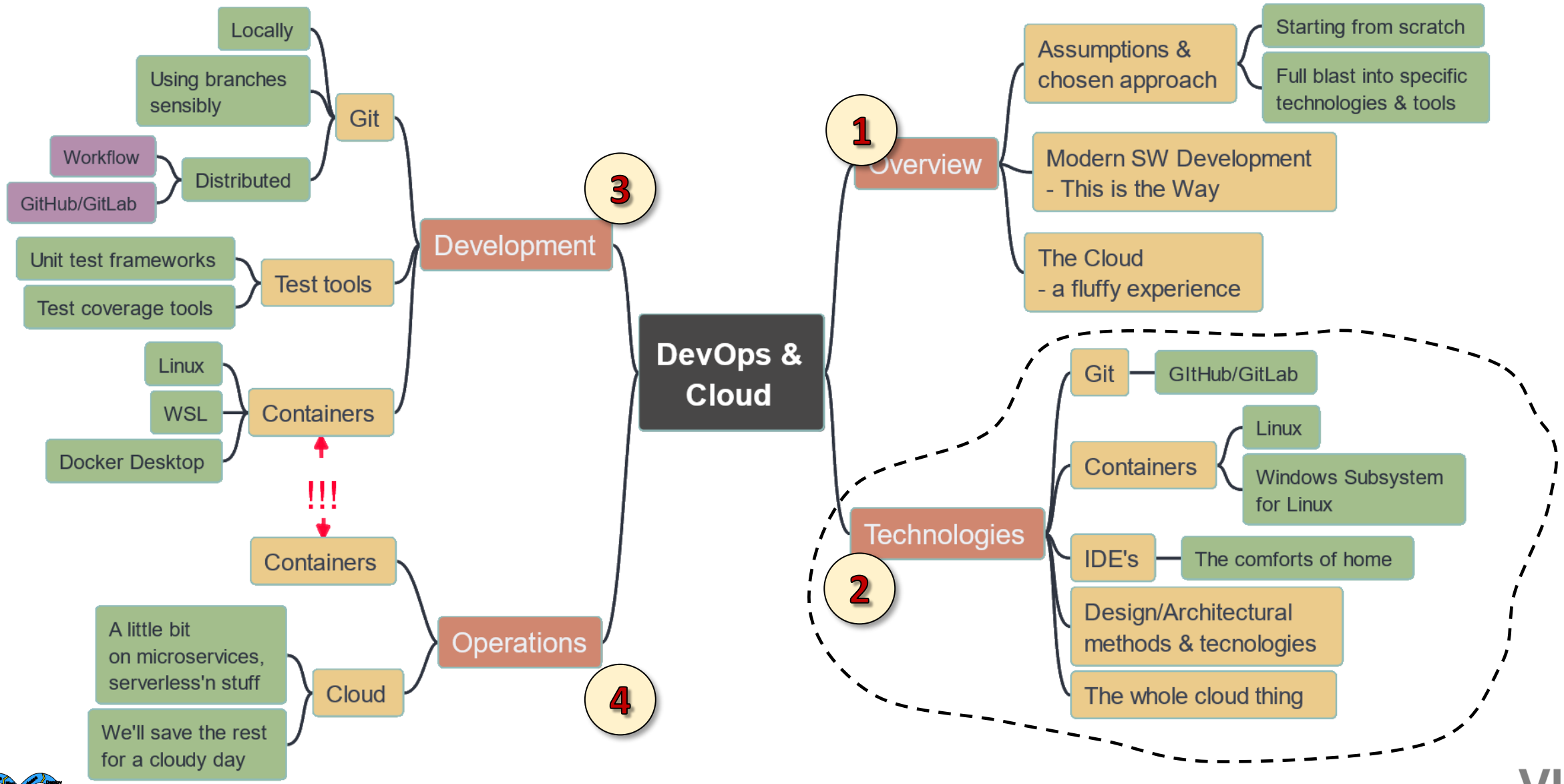
1) A *way of working*, creating *smart workflows*, that *reduces the time* that goes from deciding and planning a feature (or bugfix) until
   - That feature is implemented, fully tested and *integrated in our code base*
   - A new version of our system is *in normal operation*
2) Any *tool* or *method* that helps us achieving the above by
   - *automating* development and operational workflows, across *the full sw development life-cycle*
   - *supporting the simultaneous work* of multiple developers working on several lines of development
   - *bridging the technical differences* between the development environment and the operational environment
   - *facilitates deployment* of new releases of software

DevOps and Cloud - BFF

Using the definition from the previous slide, we can see why DevOps and Cloud concepts are BFF – Cloud technology can really help us achieve the DevOps goals of minimizing the systems development life cycle time



Overview – mind this map



Technologies – quick appetizer …. (1)

- ***Git*** (VCS - version control system) – because that's the stuff that ties things together and forms the core of our workflow
- ***Linux*** (often in the form of WSL (Windows Subsystem for Linux)) – because Linux is ubiquitous and (for DevOps/Cloud) has one special feature that is reeeeeaaaally helpful, namely
- ***Containers*** (a form of virtualization) – because this tech makes it possible for us to fully control the runtime environment of our applications in a cheap, neatly wrapped packet

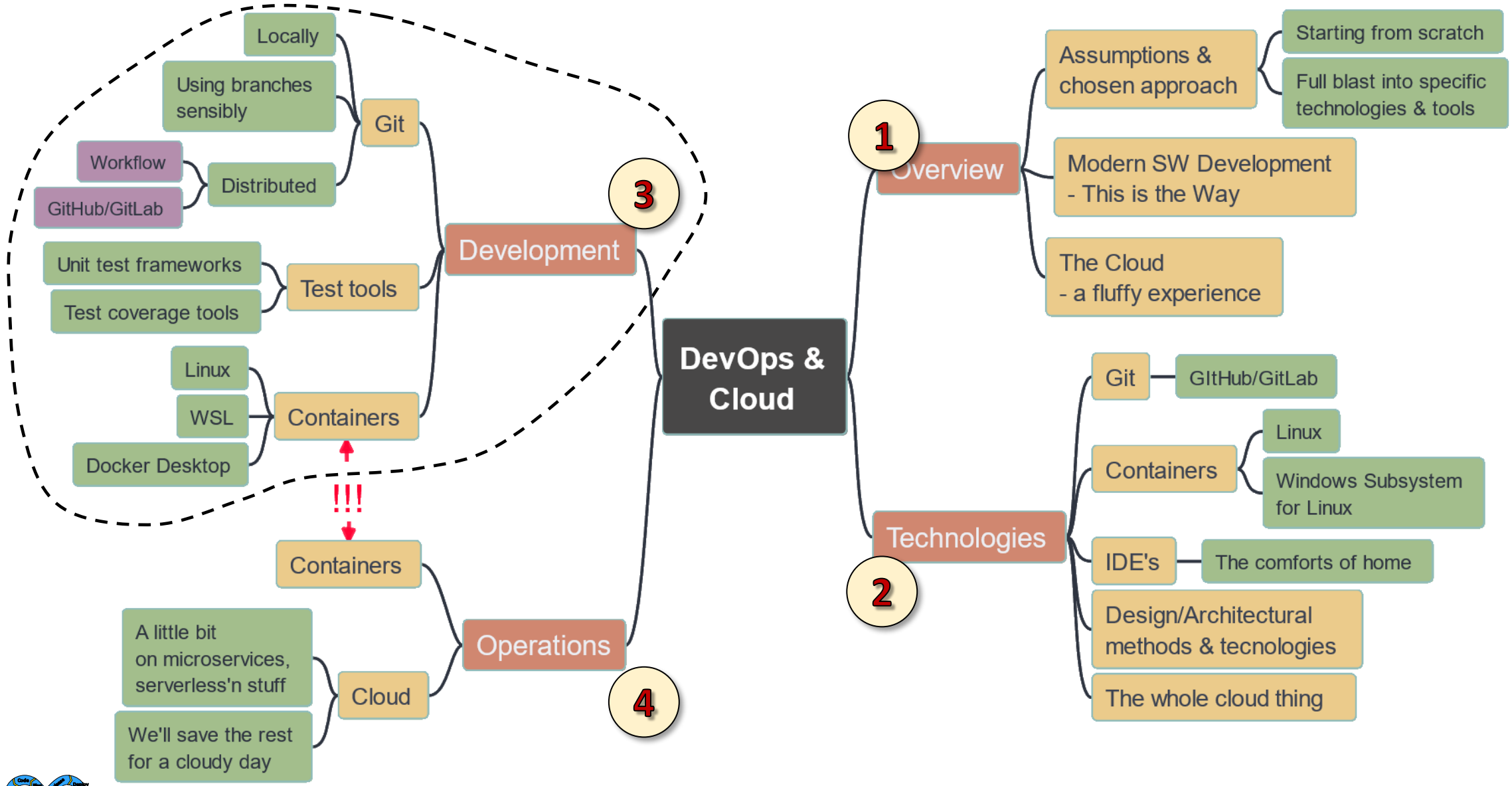
Technologies – quick appetizer …. (2)

- ***Design/architectural*** methods – because they make it possible to make sw that is both scalable, robust and easy to move around (even while an application is running). Examples are:
  - Microservices
  - Serverless
- ***IDE's*** (such as IntelliJ and Visual Studio Code) – because they are equipped with features that works with containers which make it possible to do all sorts of neat tricks
- ***Github/Gitlab*** – because they facilitate both distributed development and automation of builds, testing, packaging and releases

Technologies – quick appetizer …. (3)

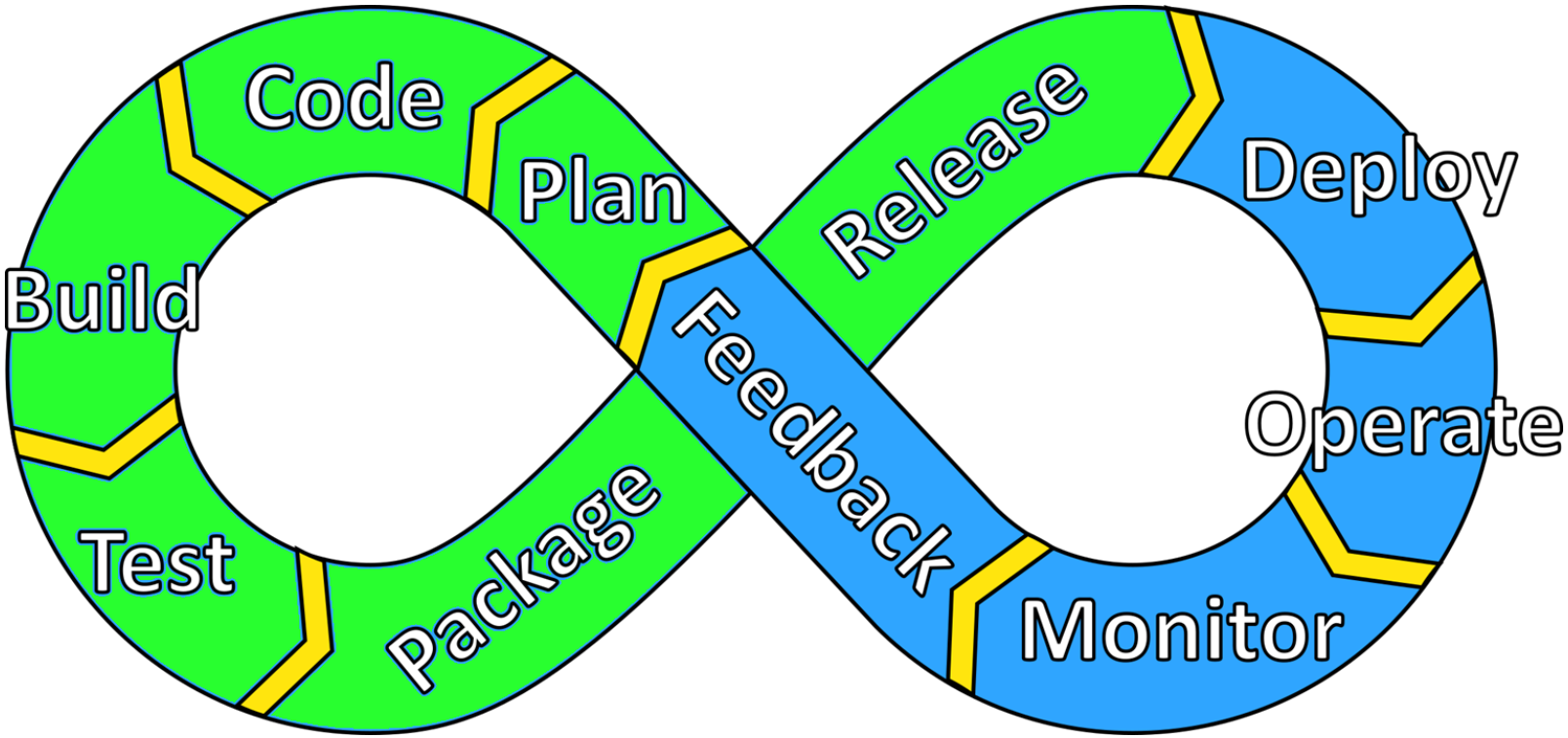
- This whole Cloud thing – Amazon, Google, Azure, OpenStack, etc.
- The Cloud providers more or less provide the same services and resources
  - CPU cycles (virtual machines running, Linux or Windows, commissioned in seconds) and computer RAM
  - Persisten ttorage (in several categories, getting more expensive the faster you want)
  - Network (bandwidth, DNS mappings, and what have you not)
  - Lots of “premade” appliances (databases (SQL or NoSql), webservers etc.) for us to use or customize

Overview – mind this map

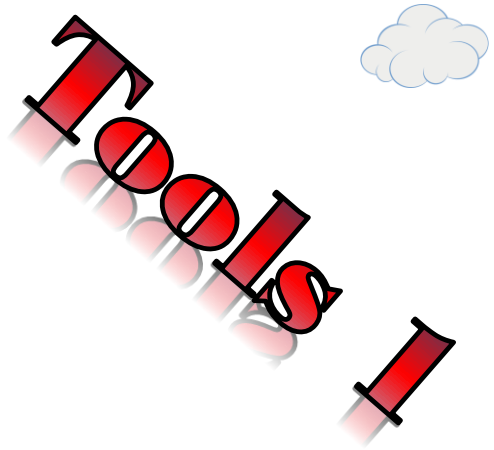


The Dev-part of DevOps – tools & technologies

- So let's start out with a look at (some of) the tools/technologies we can use for DevOps'ing the Dev part ....



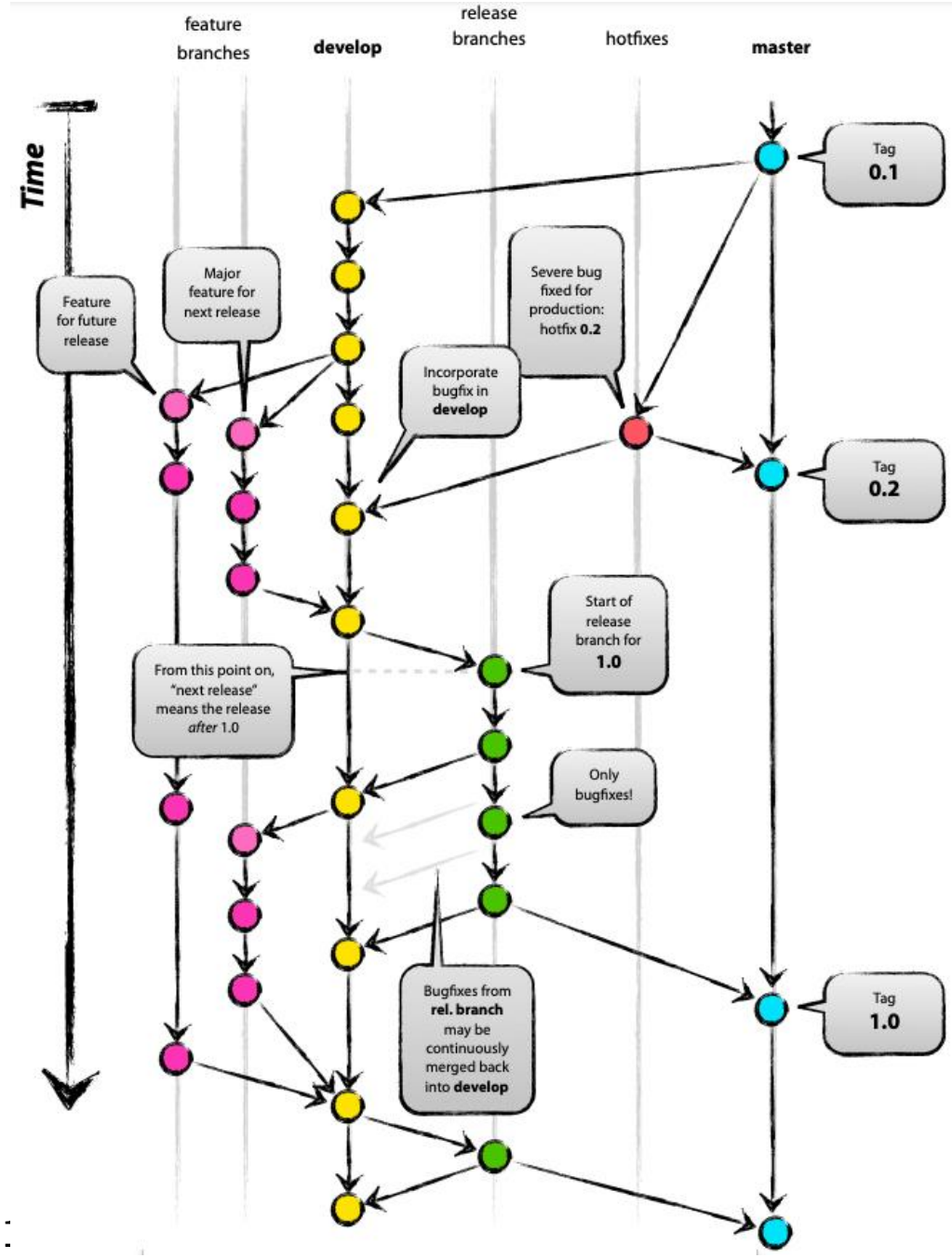
Git basics – using Git sensibly



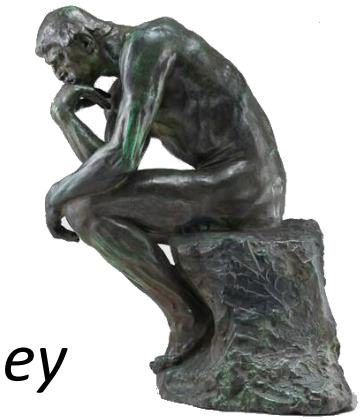
- We have to think about Git in two (but related) contexts:
  1) ***Locally*** (and that's all we discussed so far)
  2) ***Distributed*** (we'll get into more details on that later)
- Local use of Git
  - Something you do from a command prompt with git commands
  - Or something you do from from your IDE of choice
  - We need to be able to work locally with branches, tags etc. before we start sharing code
- Distributed use of Git
  - This is where GitHub/GitLab etc. comes into play
  - Here we must work smart with branches, pull requests, branch protection rules etc.

GitFlow – overkill!

- Googling around, many stumbles across GitFlow
- A model for using Git branches in a large project with many participants and parallel ongoing lines of work (features, bug fixes etc.)
- ***Don't use that for your project!***
- GitFlow has its raison d'etre and a role to play in some circumstances, but ***it's not*** for everybody



Thinking branches “globally” and locally …



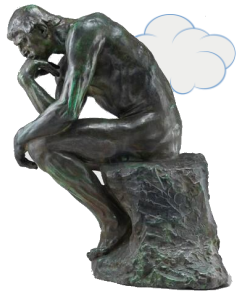
- First a few observations:
  1. We can do whatever we want locally, in our own copy of a central repo.
     So: *as long as we don’t try to push our local branches to the repo, they are invisible to everybody else using (cloned copies of) the central repo.*
  2. Branches in a central repo (i.e. GitHub, GitLab etc.) are visible to everybody that clones that repo.
     So: *all users of a shared repo needs to agree to a set of rules about how the “public” branches are used.*
  3. One fixed and crucial rule for branches in a central repo is *that all changes (commits/merges) to the main development branch must be done via pull requests* in the central repo.

To ***push*** local changes, we need to do a ***pull*** request

To push a change on a local branch to a global (= shared) branch we must perform three steps:

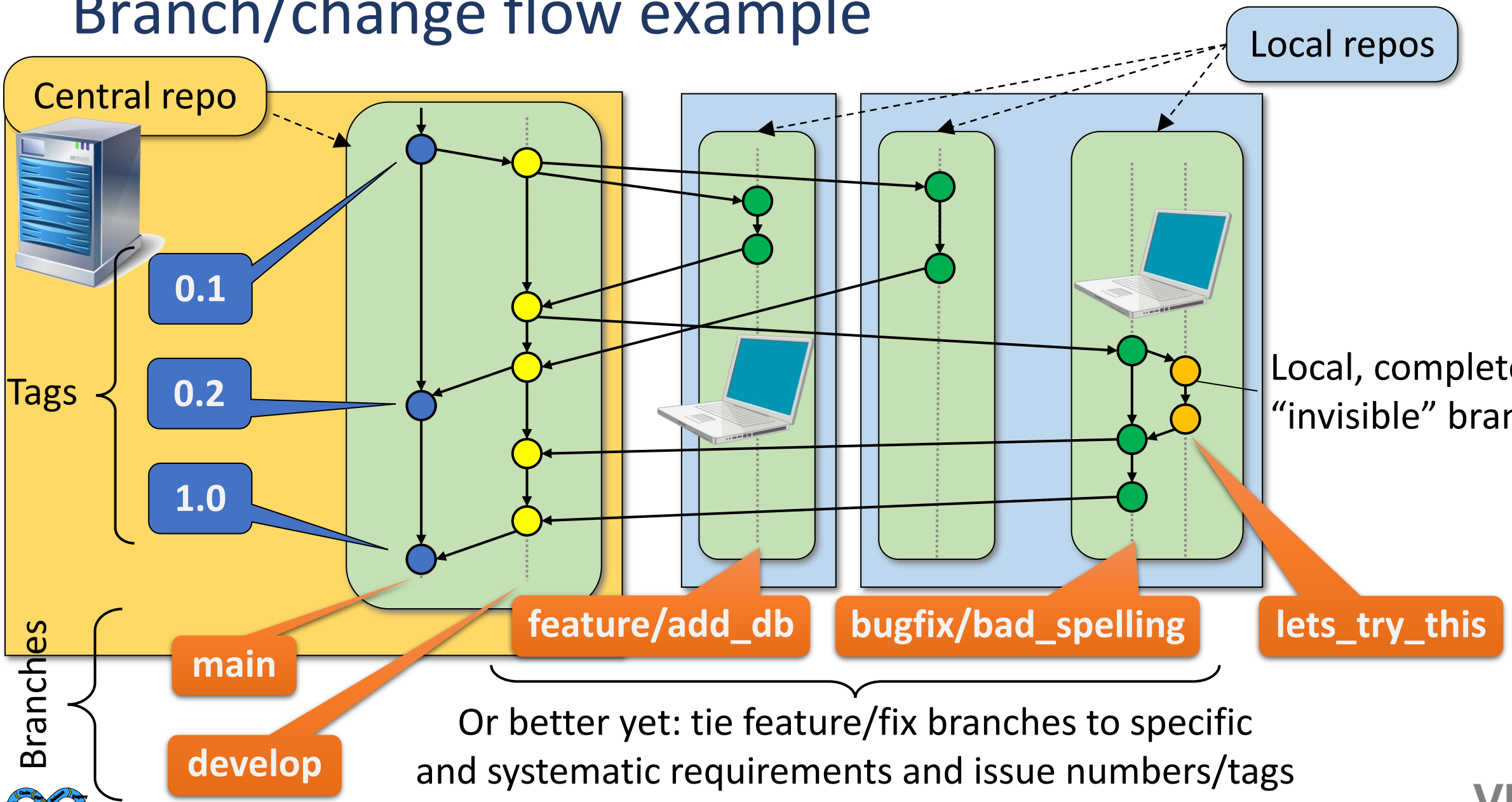
- **First push the local branch, warts and all, to the central repo (Github).**
  That just creates/updates a copy of the local branch on Github – it has no effect on any other branches
- **Then create a pull request from the shared branch where you want the updates to end up.**
  The shared branch should be setup disallow direct pushes and require approval of pull requests – so this step does not change the shared either
- **Finally, when the pull request has been reviewed and approved, merge the pull request into the shared branch**
  First at this point do the changes hit the shared branch and become visible to others

Thinking branches “globally” and locally …



- As long as we keep these things in mind, we can do pretty much whatever we want
- Here’s a few ***SUGGESTIONS*** …. (kind of a minimum viable rule set)
  - Globally (in GitHub/Gitlab)
    - Use a single branch to track your “releases” (could be **main**)
    - Use a designated branch for integration of features and fixes (let’s call it **develop**)
    - Snapshot **develop** to **main** everytime you want to release (and add a suitable tag)
  - Locally (in your cloned copy of the central repo)
    - Create and use a new branch, branching of from **develop**, for every new feature or bug fix you want to make
    - Use well-defined names for feature branches (feature/<description>) and bug fix branches (bugfix/<description>
    - Once the local feature/fix is done, push the branch to GitHub/GitLab and create a pull req.

Branch/change flow example



Or better yet: tie feature/fix branches to specific and systematic requirements and issue numbers/tags

Git – good git manners (how best to use Git)

To make the most of Git (for yourself and your team members), use these rules:

1) Use branches all the time, use them sensibly and name them well(example: https://dev.to/varbsan/a-simplified-convention-for-naming-branches-and-commits-in-git-il4
2) Make many frequent and small commits, rather than a seldom and few large commits – use selective staging to break down larger changes into smaller commits
3) Make each commit be about one thing, and 1 thing only and use sensible rules for the commit messages (example: https://cbea.ms/git-commit/)
4) Do NOT check in derived/generated files (such as compiled stuff) – EVER!
5) Do NOT check in config files that change screen colors, keyboard shortcuts or any other personal preferences – your teammates will (rightly) kill you!
6) Setup, use and maintain one or more **.gitignore** files from the very start

Good house keeping …

- Delete branches when they are no longer needed. You can always recreate the branch from the last commit if needed
- This is a must in the central repo – otherwise the old branches with completed features and fixed bugs will seep into local repos over time and pollute the whole thing
- You can do what you want locally, but your local repo can easily drown in a mountain of lint in the shape of old, unused and stale branches – so cleanup

Test tools

- Good test tools and test habits are critical for keeping the code base in good shape
- Using good test tools (especially for unit testing), and using them right and all the time, is very much a part of a DevOps flow
- Automating tests not only saves time writing and maintaining the tests but saves TONS of time downstream
- Adding a test coverage tool adds a fantastic objective view into the state of the code base
  - Which parts have been tested, and which parts not?
  - How does test coverage evolve over time? Is it steady/increasing/decreasing?

How to enforce workflow rules on GitHub

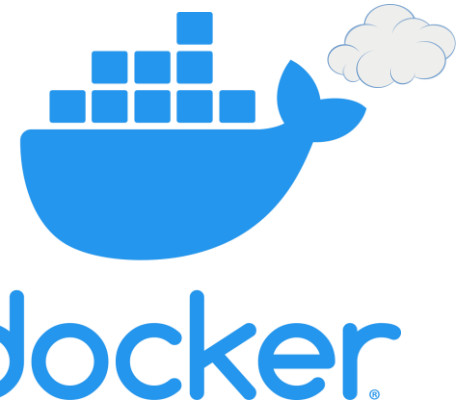
- We have 2 essential tools in GitHub that we can use to enforce workflow rules that help us keep our code base in good shape:
  - Always buildable
  - All tests running
  - Always runnable

  } = code that's in good shape

- These 2 tools are:
  - Branch protection rules
  - GitHub actions – which really should be called GitHub workflows
- A small example https://github.com/DaFessor/devops-demo
- Branch protection, workflows, test and coverage

Containers & Docker

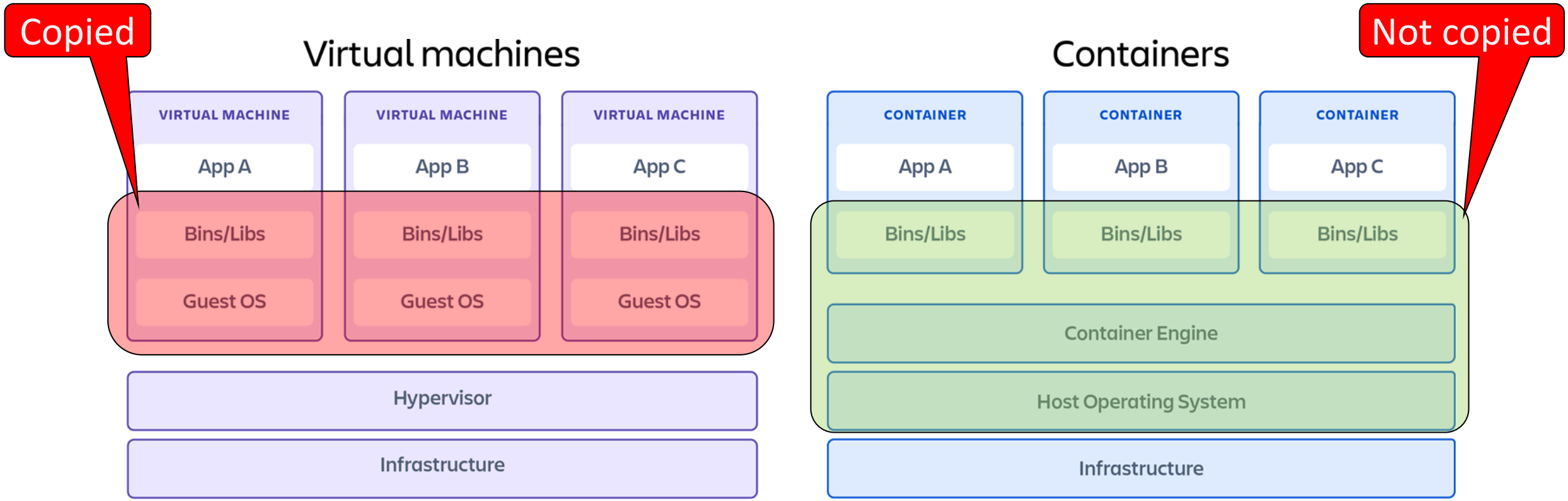
Tools 3



- Docker is the company that popularized Linux containers
- A container is a kind of “virtual machine”, in the sense that apps running inside a container sees their own “private” machine
  - They have their own file system, nobody from outside the container can mess with files
  - No other processes are running
  - They see their own network (devices and network addresses)
  - All needed libraries have the exact right, well-defined version
  - An app may run with root/superuser rights
- Seen from the outside the container is a fully closed and sandboxed jail
  - Apps running inside the container cannot see or modify files, processes or access the network outside the box

Containers vs traditional VM's

- Containers start & stop really quick since they don't have to simulate a full machine w. hw and everything, VM's have to do a full boot/shutdown cycle
- Real VM's duplicate all stuff used across different VM's, containers don't

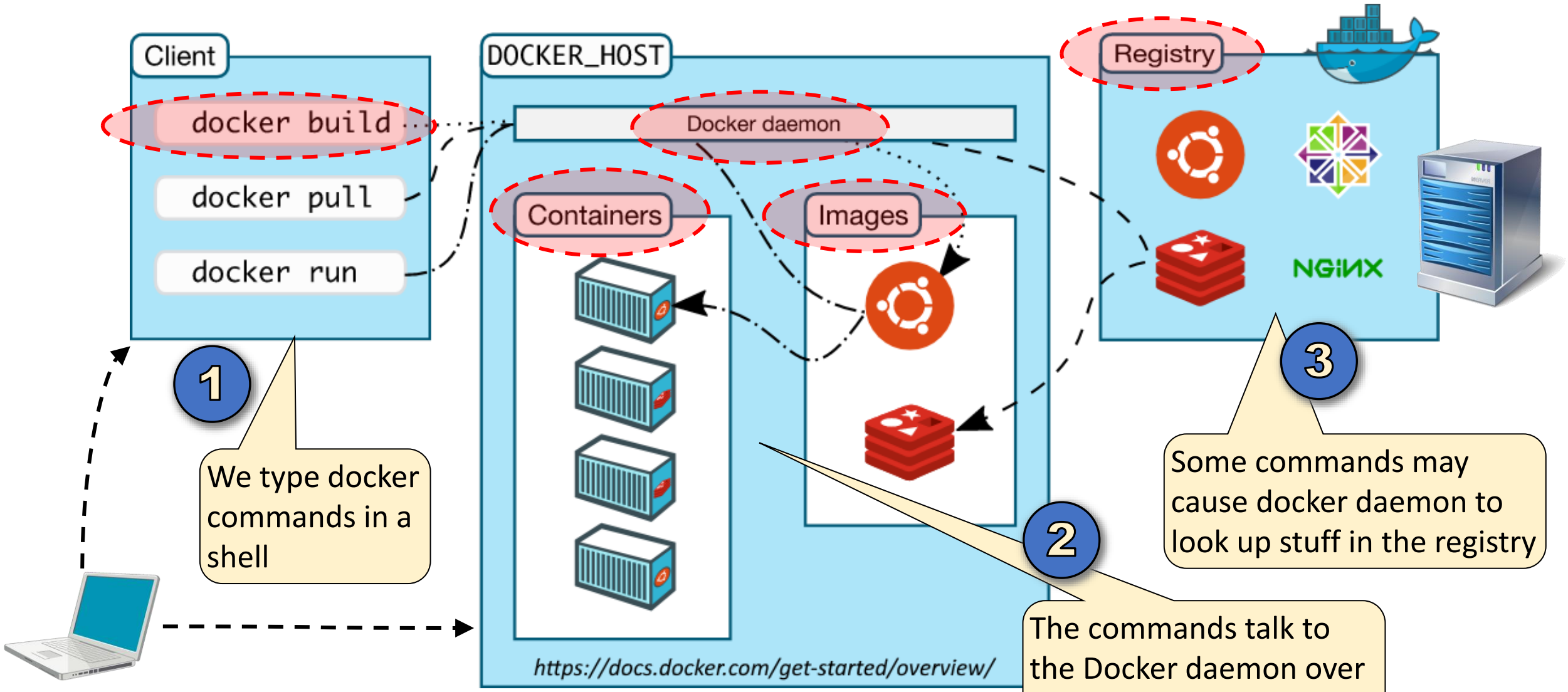


https://www.atlassian.com/microservices/cloud-computing/containers-vs-vms

Docker schmocker, Podman and containers

- Docker (the company) popularized containers, and we often use the two, Docker and containers, as if they are one and the same
- They are not – there are other providers of container technology on Linux, such as Redhat Linux that makes Podman
- So, informally Docker and Podman are two examples of *container management "engines"* allowing us to build, run and manage containers
- Container tech is standardized to an actually useful level, via the OCI (Open Container Initiative)
- Not all containers are the same type , but what we learn by working w. Docker works very well to other container types (e.g Podman)
- But henceforth we'll only talk about Docker containers

How Docker works



Docker images, containers and such …

- ***Client:*** An application that wants to do Docker stuff. Example: A shell running in a terminal, where we manually execute Docker commands.
- ***Dockerd:*** The actual container management engine that runs as a service which the clients can talk to over a (usually local) network connection (i.e. client and docker host is often the same machine)
- ***Registry:*** One or more registries ("library") of premade docker images ready to be downloaded and used
- ***Image:*** A read-only template with instructions for creating a Docker container, often based on another image with some additional customization.
- ***Container:*** A runnable instance of an image.
- ***Volume:*** A persistent "file system" that can be attached to a container.

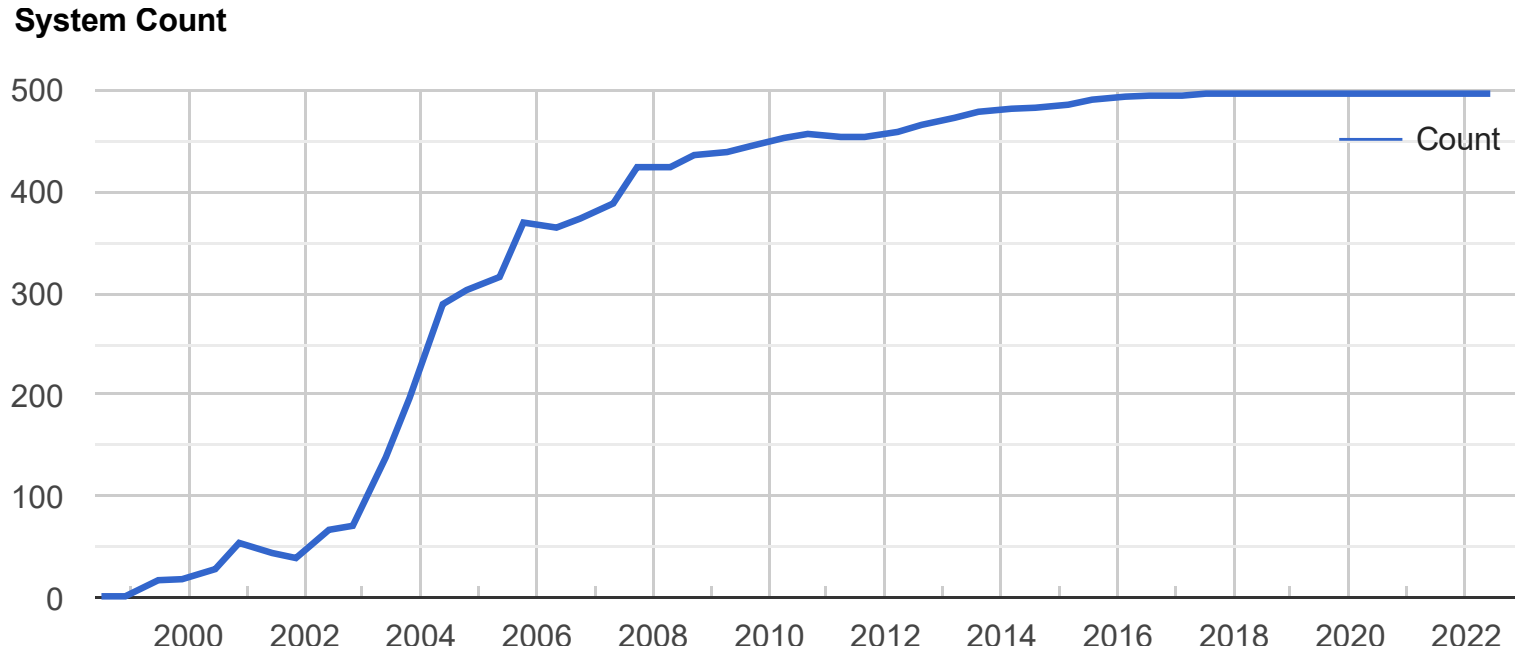
To work w. (Docker) containers we need Linux

- .... or a very recent version of macOS
- Docker Desktop also runs on Windows (handy) but it needs Windows Subsystem for Linux (WSL) to do handle the actual containerization
- WSL v2 runs a real Microsoft customized Linux kernel inside Windows
- Docker needs WSL2 to work

Linux – from humble origins to today

- In ***1991*** Linus Torvalds, as a student of computer science at Helsinki University, wrote in an Internet news group:
  Hello everybody out there using minix – I'm doing a (free) operating system (just a hobby, won't be big and professional like gnu) for 386(486) AT clones. This has been brewing since april, and is starting to get ready

- A graph of how many of the most powerful 500 computers in the world that used Linux in november ***2022***



Linux usage

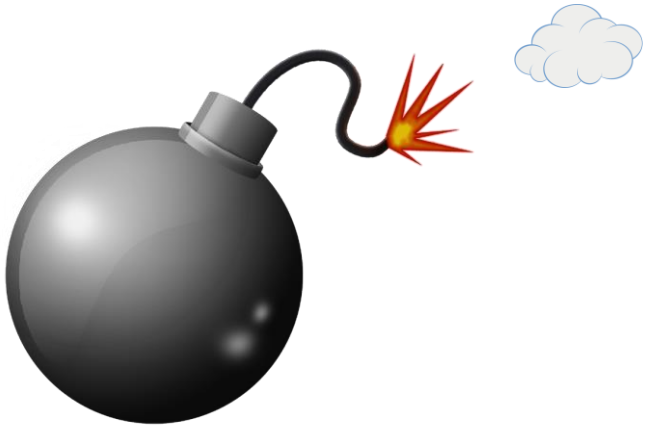
- In oct 2021 appr. 71% of all smart phones in the world used Android – which has Linux running under the surface
- On top of that comes all the routers, switches and other network devices, uses in automotive applications, smart home applications and so on ….
- In short, Linux is a VERY WIDELY USED open source operating system, that runs on all sorts of devices from usb-stick sized embedded devices to the largest computers in the world with literally thousands of cores.
  The frontrunner as of today('ish) has 8,730,112 cores, and delivers 7.94 EFlop/s (that's 10^{18} =7.94 x 1 million x 1 million x 1 million 64-bit floating points ops per second – not shabby for a hobby project, eh? You can encode a LOT of cat videos VERY fast with that kind of power!

Why has containers become so popular so fast?

- They are very use to manage using tools like Docker, Docker Compose and Kubernetes
- They are very fast to start up (≈a few milliseconds on my humble laptop)
- They make it easy to run/deploy software together with all necessary dependencies as a single “runnable” entity
- They can be used everywhere we might want to run, test or build or our software:
  - On our development machines
  - In the pipelines of GitHub/GitLab etc.
  - On the operational target machine(s)/systems

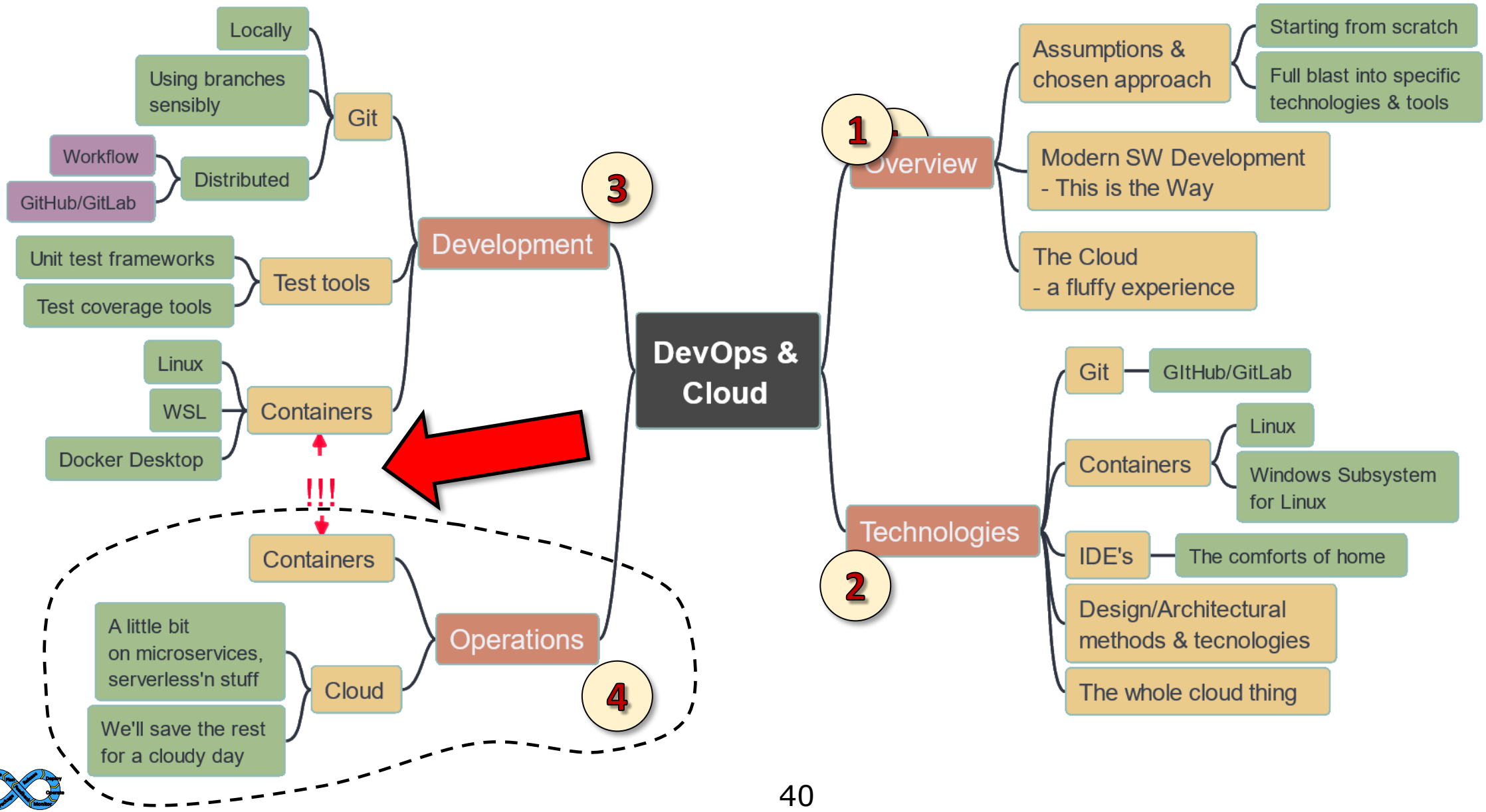
We can use the same container in all 3 situations! That has far-ranging implications for making things easier .....

Demo time



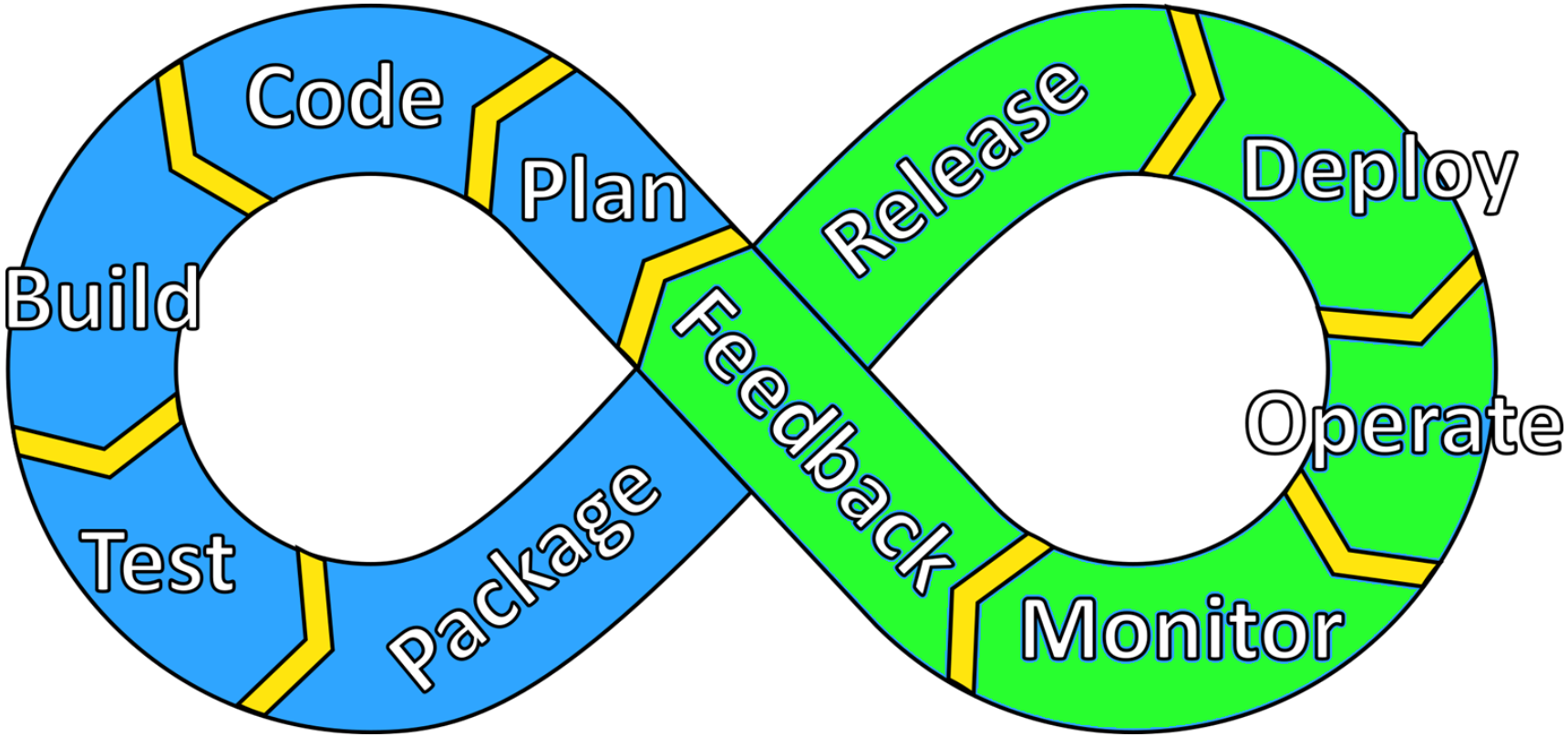
- WSL – bashing around .....
- Docker Desktop
- Working with a project using a *development* container
  https://github.com/DaFessor/devops-demo
  - Demo full development environment w. Java & Maven
  - Run application: mvn spring-boot:run
  - Run tests: mvn test
  - Show result of doing test coverage: https://dafessor.github.io/devops-demo/
- VSCode really rocks when it comes to container support

Overview – mind this map



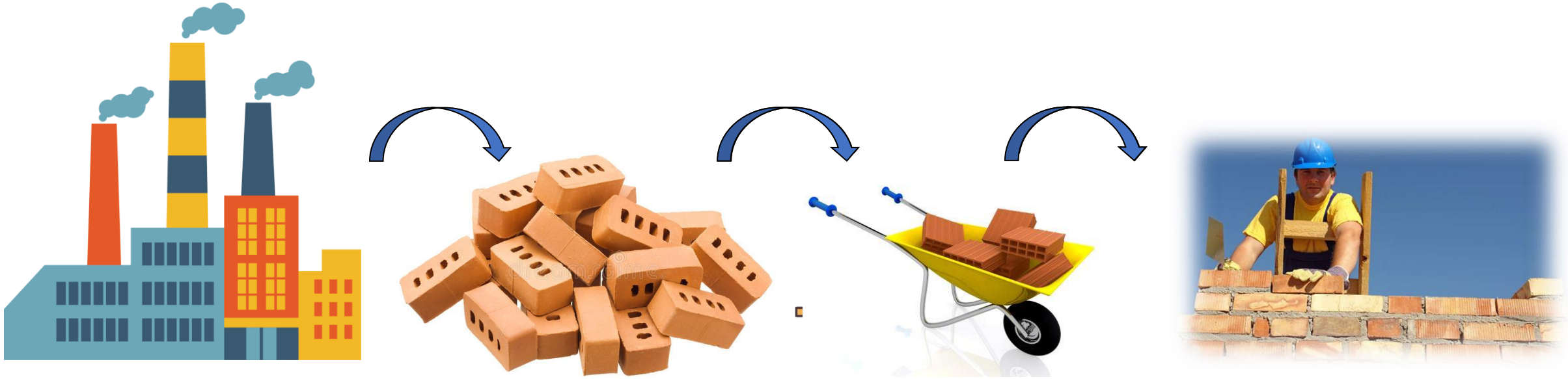
The Ops-part of DevOps – tools & technologies

- Now let's take a look at (some of) the tools/technologies we can use for DevOps'ing the Ops part ....



So, we get Dev-ilish fast at releasing new versions ...

- ... but that's not much help if we can't put the new software to use



- One of the corner stones of the DevOps "mind set" is that we have to think about the full sw lifecycle – so we're not done just because we throw a new release out the door ....

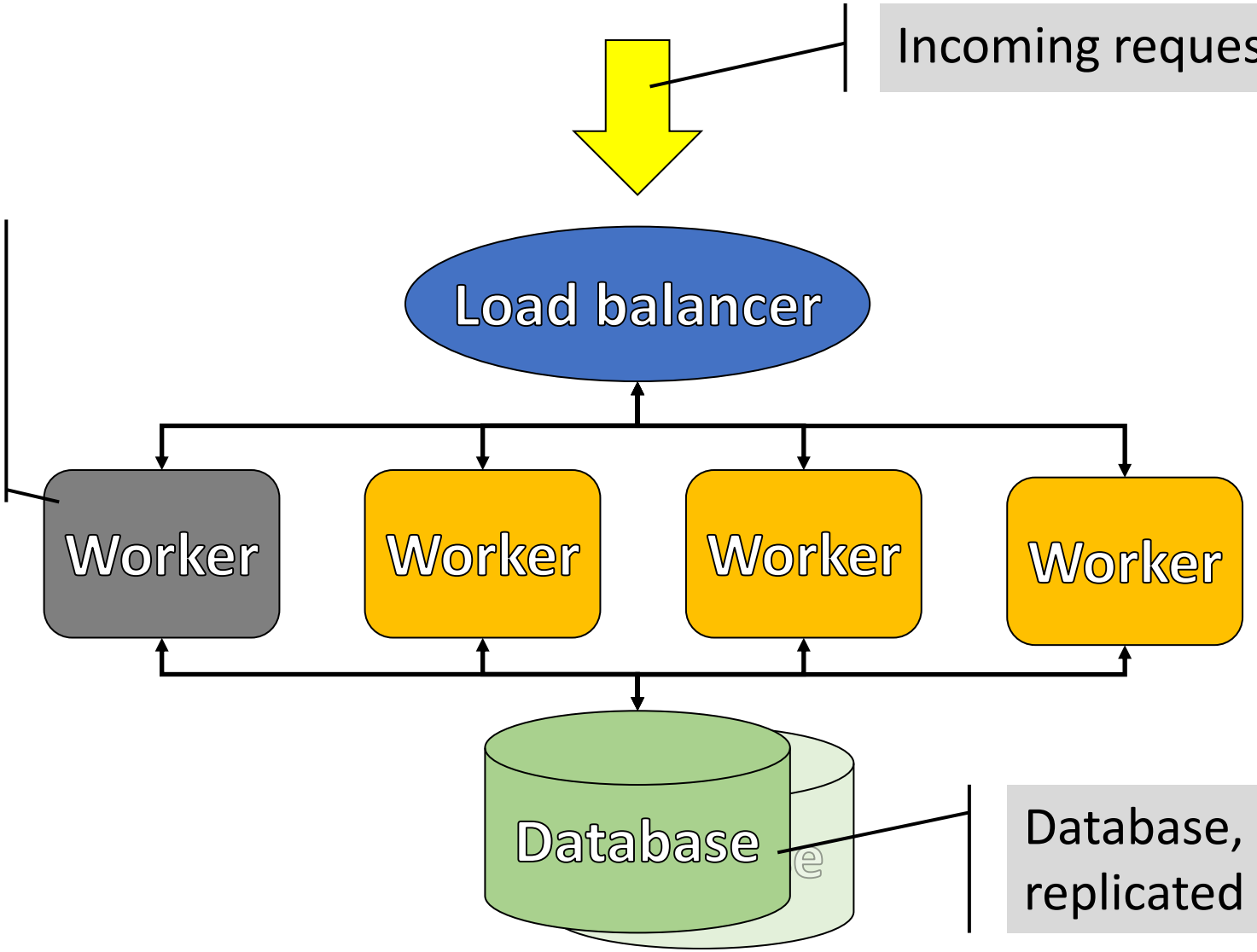
Facilitating the Ops-part of the sw lifecycle

- How can we make it easy to deploy and operate our software?
- Let's start with the easy/obvious thing - we need to deliver our software in an easy-to-deploy "format", but what is that?
  - executable/jar-file(s)?
  - container(s)? <- This has become increasingly popular
- We also need to take operational demands into account
  - Is a stop-update-restart cycle permissible or do we have a 100% uptime requirement (think Netflix)?
- If we are not allowed any downtime for updates, we must design the software from the ground up to facilitate a "running" update – we need to think about how we design the basic architecture

Updating software on the run

- This is a rather large subject, but let’s take a quick look on some of the ways we may make it possible to do “hot updates”
- We’ll look at
  - Load balancing
  - Micro services
  - Serverless

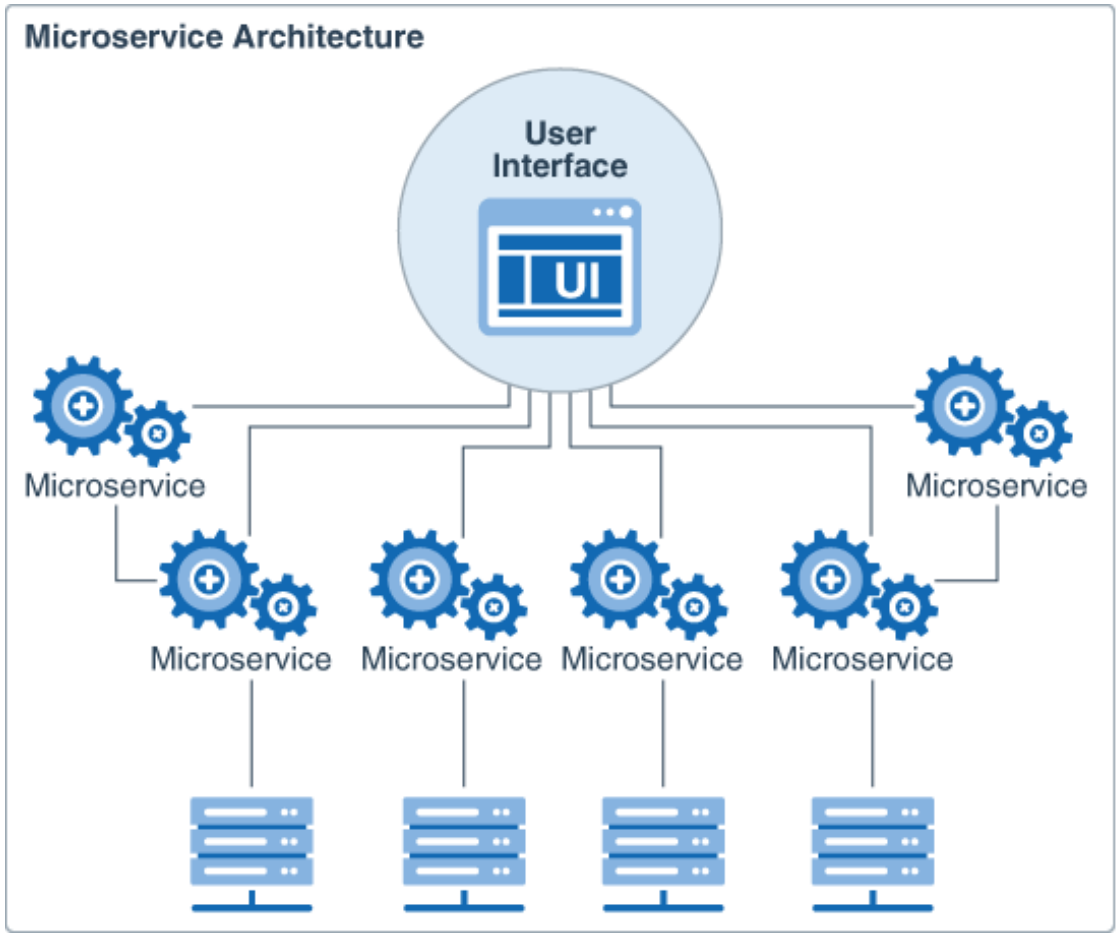
Load balancing

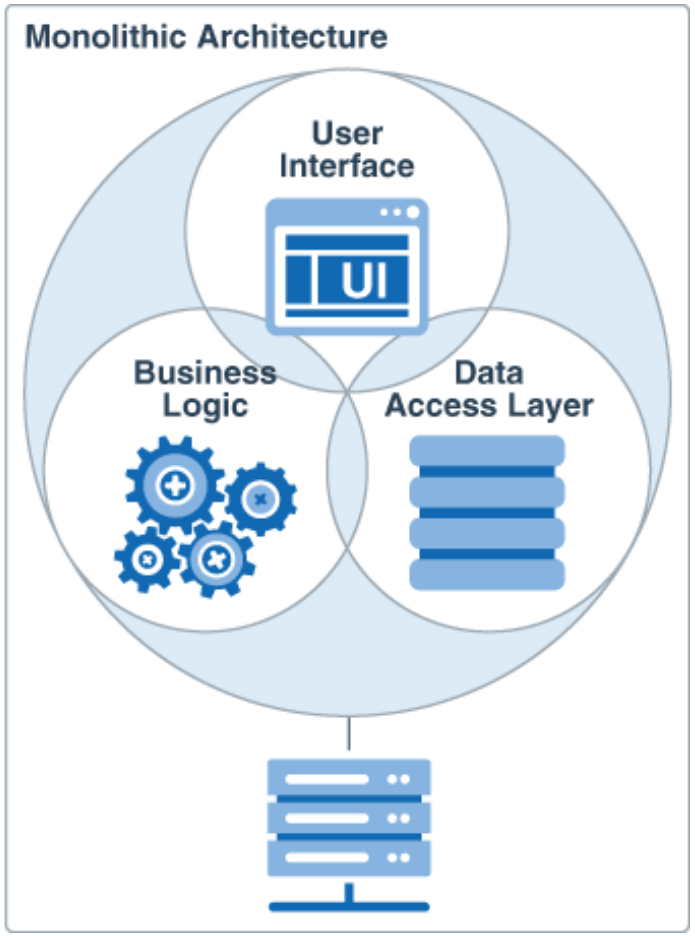


We can do a stop-update -restart cycle on 1 worker at a time w.o. interrupting normal operations

Micro services (1)

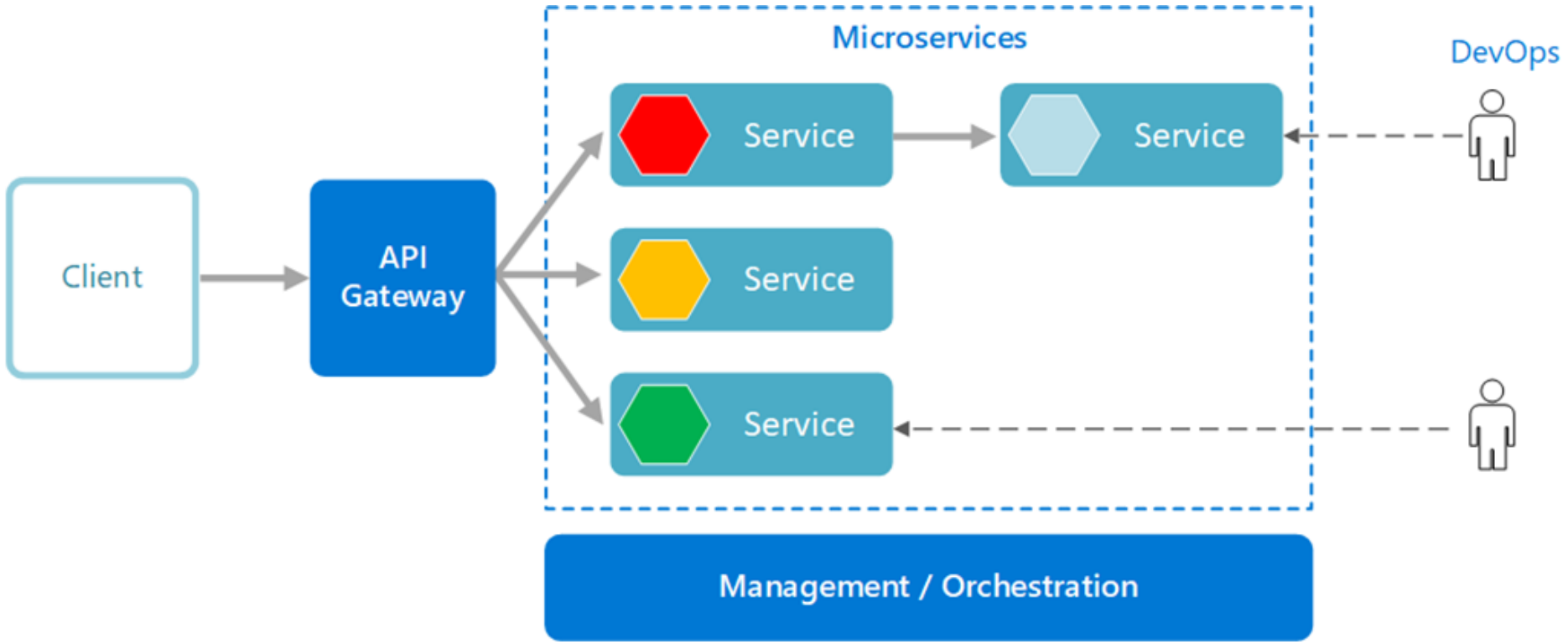
- Monolithic vs microservice architecture





Microservices (2)

- A microservice architecture opens for a very flexible DevOps setup – you can have different team/tool/DevOps setup per microservice



Microservices (3)

- Microservices frequently runs in containers in the Cloud
- But while microservices frequently is associated with running software in the Cloud, it's actually a general architectural pattern that can used without necessarily using the Cloud (or perhaps with a private on-premise cloud)
- So, in many ways microservices is architectural pattern that can be used to get rid of traditional monolithic applications with all their "bad qualities"
- A microservice can be developed, maintained and concurrently updated separately from all other parts of the system – that's smart!

Serverless

- No, Serverless is not w.o servers – there has to be some servers running, somewhere
- Serverless is cloud-native – i.e. closely coupled to Cloud service providers
- Serverless is also known as FaaS = Function-as-a-Service
- Cloud providers sells ready-to-use serverless databases, serverless storage, serverless blahblah etc.
- But what IS serverless – it's actually hard to find a single, simple and direct answer on that ....

Serverless explained (sort of)

- Imagine architecting your code as a set of functions
- Each time the code needs to do some kind of work, a specific function is called to do the job
- The code for the function is normally dormant, i.e. the code is not “active” like it would be if it was placed inside a traditional server application – so no CPU cycles are used to keep the code “warm” or ready
- Whenever the function is called, the code is spun up very fast, it gets it’s data, does what it needs, returns data and stops again
- You need need to pay for the number of times you call the code, and the amount of CPU cycles that’s consumed during the call – it does not cost anything to have the function “on standby”, you only pay for actual use

Interesting aspects of μ-services and serverless

- Some say that microservices and serverless has the potential to make the traditional Ops-work obsolete
- The idea is that developers themselves become Ops-people, if they get the responsibility of updating microservices or serverless functions directly whenever the developers make new versions available
- So, we don't anybody to babysit the operations and supervise updates of the software
- That has caused some people to declare that "normal" DevOps thinking will become obsolete, since the Dev part and Ops part fuse together
- Time will tell ...