Advanced Development Techniques

08: DEVOPS

Software development in the old times

Servers in the cloud

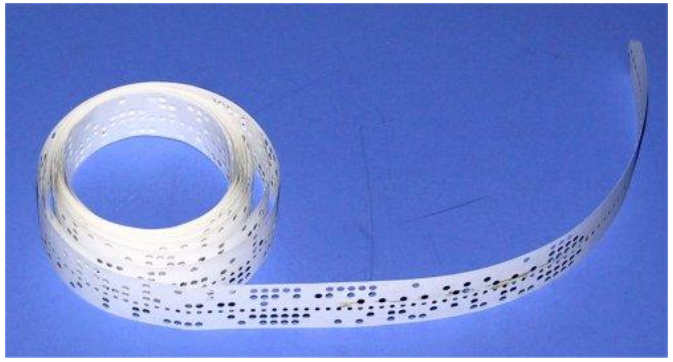
CI/CD

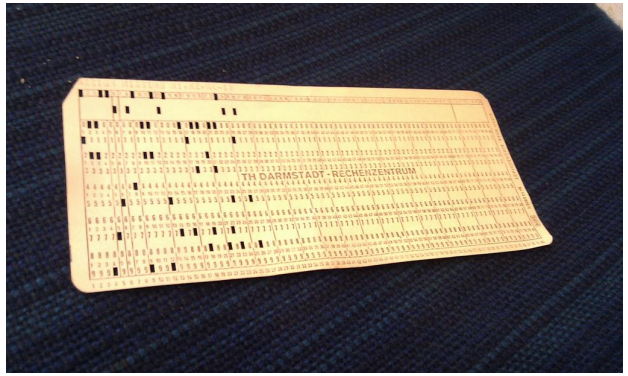
Container technologies

Advanced Development Techniques

Software development in the old times

- Developer works on the code at home...
- Takes it to the work place...
- Test it, run it...
- But it doesn’t work...
- Notes the problems and bugs and goes back home
- ...Repeat

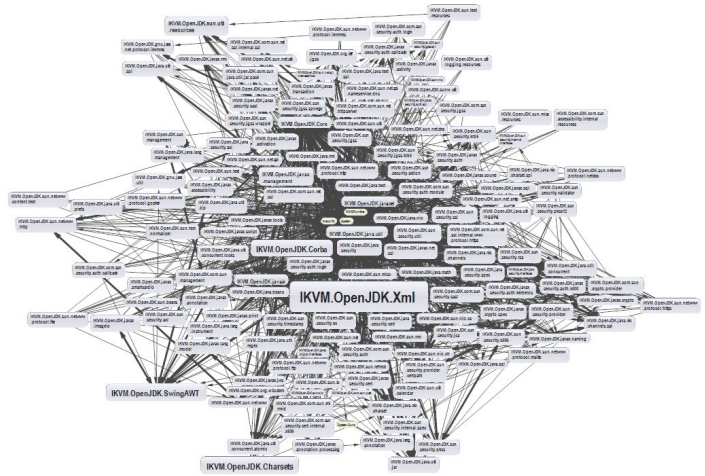




Deployment in the early days

- At a given company a server has been bought after a long procedure
  - it is oversized (“if we buy one, buy something which exceeds our needs”)
  - it can be broken after some time of usage
  - it had multiple responsibilities (mailing, web, storage etc.)
  - there is no staging server, or at least it is rare to have
- Codes were uploaded via FTP
  - who uploaded what and when?
  - there is no rollback
  - this is called manual deployment
  - there is no configuration management tools (manual install)
- Outcome: it needed a lot of patience, it was chaotic, the system admin was the god
- Time period: in the 90s

Version upgrades / Dependency hell



- On 1 computer 1 OS is running
- On 1 OS there is 1 software version
  - Eg. Java development JDK4 version is installed
  - It was a barrier to the development (*development* as progression!)
    - It was impossible to move to JDK5 for example because there was no capacity to rewrite the already running and live JDK4 programs
    - Because of this the developers lost the newer versions' (eg. JDK5) advantages and new features
    - This resulted softwares with old versions (and all the disadvantages eg. security issues etc.)
- Who makes the decision? → System Architect
  - Usually SA is a better developer
  - Or somebody who has no idea how he got there… :)
  - But at least everybody is "happy"

Virtual machines

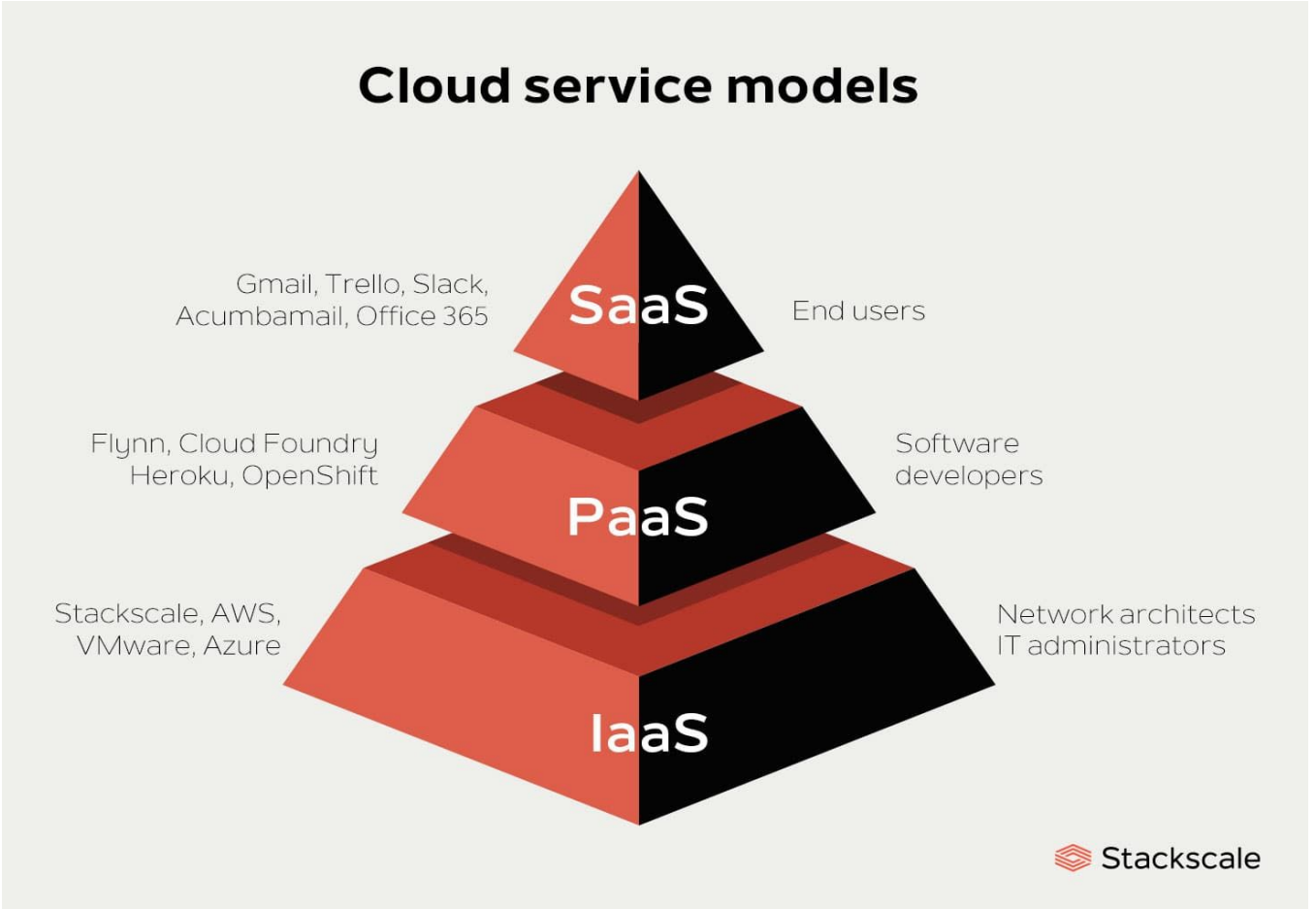
- Back in the 1960s it was already a thing at IBM
- **Essence**: On a physical machine multiple OS can be run
- Advantages:
  - Better resource management
  - Dependency problems solved
  - Can be copied, moved etc.
- Disadvantages:
  - Less resource is available for the softwares
  - Competition to the host OS's resources
- Hypervisors: Vmware, Virtualbox, Xen, QEMU, Hyper-V
- Typically no one used configuration management at this time

Cloud systems

- Amazon story:
  - We have a dozen of servers which are **not in use** during most of the time of the year
  - Lease them (anybody can rent them for some purpose)
  - The leasing/rental should be as flexible as possible
  - Support it with a great admin UI for the renter
- **This is exactly the birth of cloud computing**
- From the developers POV
  - My data is stored somewhere (on the servers of Amazon)
  - Somebody makes all the system updates and maintenance
  - We pay some amount based on the usage
- The cloud hides the technical details which occur as everyday problems on a high abstraction level

Service levels

- **IAAS (Infrastructure-as-a-service)**
  - virtual machines
  - software defined networking (SDN)
  - firewalls
- **PAAS (Platform-as-a-service)**
  - databases (eg. Microsoft SQL Server)
  - file servers (eg. Azure Blob Storage)
  - application servers (eg. IIS)
- **SAAS (Software-as-a-service)**
  - data storage (eg. Google Drive)
  - mailing (eg. Office365)



Cloud providers

- Microsoft Azure
- Amazon Web Services
- Google Cloud Platform

Disposable infrastructures

- Old days: we have the server and it's running
- Virtualization: we have the virtual server and it's running
- **Why would anybody throw it away?**
  - Because it is no longer needed
  - Because it is no longer needed in any given X number of copies
  - Because we only want to try out something (eg. as a prototype)
  - Because we only want to test something
- Testing something = all day long PC installing?
  - No! Infrastructure As Code
  - We write down in a text file what we need and it will be created
  - For virtual machines: Vagrant
  - For cloud: YAML files, GUI clicking

Vagrant


```
teszt = "teszt15"

default_disktype = "Standard"
default_box = "generic/ubuntu1804"

config.vm.box_check_update = true

machines=[
  {
    :hostname => "#{teszt}-server",
    :ip => "10.0.200.10",
    :disk => 2,
    :disksize => 420,
    :disktype => "#{default_disktype}",
    :memory => 1024,
    :cpu => 2,
    :script => "provision.sh",
  },
  {
    :hostname => "#{teszt}-client",
    :ip => "10.0.200.200",
    :box => "centos/7",
    :memory => 512,
    :cpu => 1,
    :shell => "yum -y update",
  }
]
```

C# application types

- **Console Application / WPF Application / Windows Forms Application**
  - Client side application
  - .exe build → has to be downloaded to the client machine and use it
  - Uses the resources of the user's machine
- **ASP.NET Core MVC / ASP.NET Webapi**
  - Server side application
  - Replies back to client application's HTTP requests through the internet
    - Previously the reply was generated HTML but nowadays it is mostly JSON packages (see API lecture)
  - What can be client software?
    - Web browser / desktop app (eg. WPF) / smartphone app
  - What will be the build? → DLL
  - What will host this DLL? → Web server software
  - Uses the resources of the company's servers

Web server software / application server

- **The building of a PHP runtime environment**
  - Apache or Nginx web server software installment
  - During requesting a .php file, reach out to the php interpreter
  - Easy because PHP is an interpreted language (interpreted / processed line by line)
- **The building of a ASP.NET Core runtime environment**
  - During the building it is possible to build the web server itself to the code (Kestrel)
  - Thus this, we will receive a console application
  - After launching it, it will listen on the 80/443 port and replies back to the HTTP requests
  - OR
  - We build a DLL which will be hosted by Microsoft IIS

Releasing a new version

- **Desktop/Console/Mobil App**
- Delivery
  - After some time the user realizes that there is a newer version on the website
  - Application marketplaces handles updates (Google Play / App Store)
  - Before starting the application there is a subrutin which checks for available updates
    - “There is a newer version of application X, would you like to install it now?”
    - No. :)

- **WebApp (MVC vagy API)**
- Deployment
  - After some time the user realizes that there is a new feature on the web application the there is a new feature
  - The server has everything
    - the logic
    - the HTML code which should be sent to the client
    - etc.
  - Can be changed at any time
    - Fast user (customer) request fulfillment
    - Green background → yellow background

Continuous Integration

- **What is CI?**
  - We define versions
  - We have the aim predefined, eg. in the 3.0 version we need features X, Y and Z
  - These are mostly managed together with Git
    - Development is made on the development branch, we create feature branches off of that
    - Merging to master (or production) can mean a new version (but not necessarily!!!)
    - Merge commit can be labelled eg. v3.0 or tags can be added as well
  - Apps running on the client (desktop or mobil) can be downloaded directly eg. from GitHub
  - But server apps need a pipeline system

Continuous Deployment

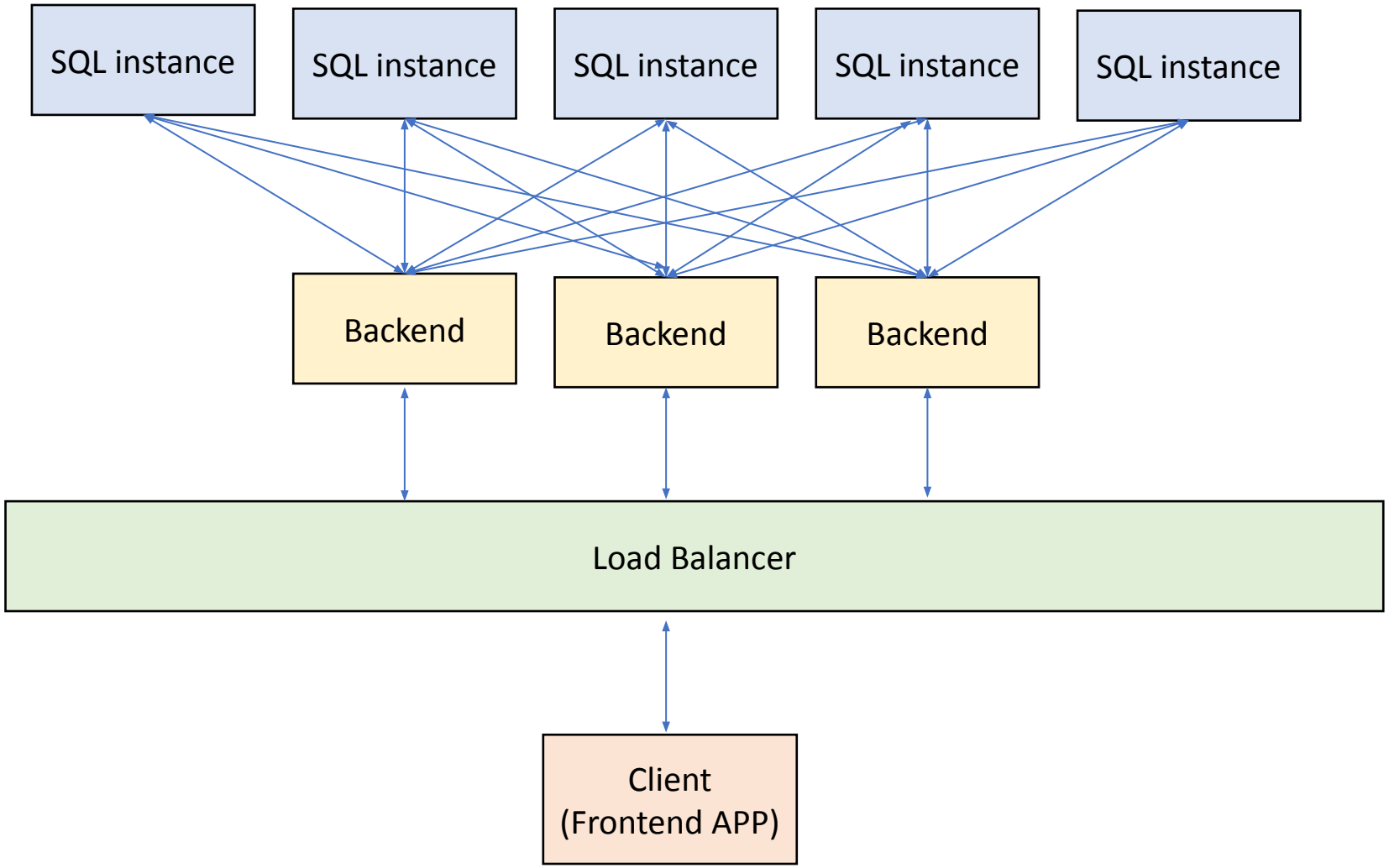
- **What is CD?**
  - Git has git hooks (event-like thing)
    - based on that GitHub (and others) have webhooks
  - We can subscribe for example to a merge event at master branch
  - After the merge the webhook will use a *git pull* on the build server
  - After the git pull the build server will execute some commands like *dotnet build*
    - The test server runs the unit tests, integration tests etc.
  - The buildelt binary then is sent to the application server
  - The app. server stops the currently running version, dispose it and start hosting the new build version
  - The user can see the new version (which can contain new features etc.) after a page refresh

  - (There is many other ways to implement the same!)

Is there a need for scaling?

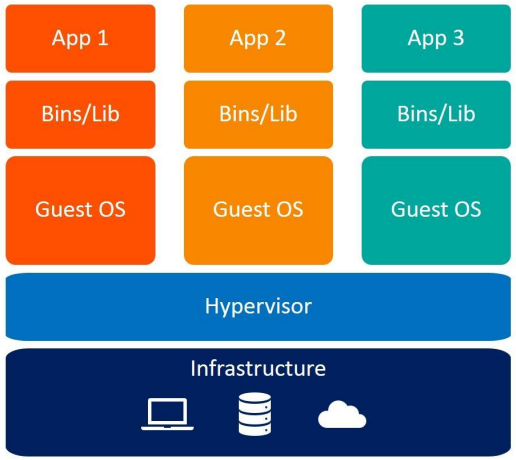
- **What are the limits of a server?**
  - Given a machine: 8 CPU cores, 16GB RAM (physical or VM does not matter)
  - How many users can be served from the server side?
    - Eg. the user requests a list with 1000 records
      - Database query: eg. 600ms (we saw during the labs that there is a small delay)
      - This can be considered as 20M clock cycle
      - Logic stores and converts: eg. 1000x2KB □ 2MB
    - How many users can this machine serve?
      - 3GHZ = (3 billion cycles * 8 core) / 20M cycles = 1.200 user / second
      - 16GB ram / 2MB = 8.000 user
    - We can assume that approximately 1.200 users can be served and each of them receives the response within 1 second
    - At 12.000 user the response time will be 10 seconds
    - At 120.000 user the service seems to be dead, as the HTTP's timeout is 30 seconds

One possible solution for scaling

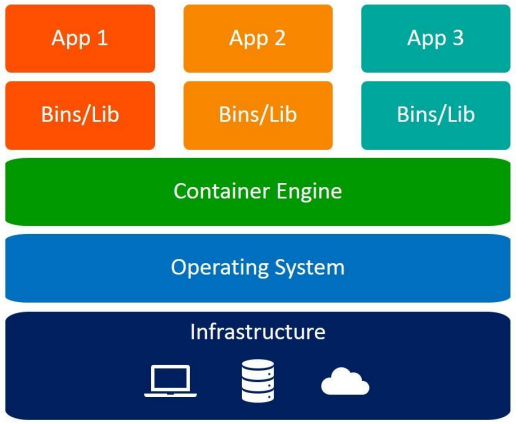


Container technology

- **What is a container?**
  - Almost like a virtual machine
  - But the OS's kernel is shared
  - What is inside a VM?
    - OS + application
  - What is inside a container?
    - **Only** the application
  - What is it good?
    - Isolated application environments
    - No wasted resources
    - Small size (containers)
  - Most well known container technology: Docker
    - Linux / MacOS / Windows all supported



Virtual Machines



Containers

Creation of a Docker container

Dockerfile

```
# https://hub.docker.com/_/microsoft-dotnet
FROM mcr.microsoft.com/dotnet/sdk:5.0 AS build
WORKDIR /source

# copy csproj and restore as distinct layers
COPY *.sln .
COPY aspnetapp/*.csproj ./aspnetapp/
RUN dotnet restore

# copy everything else and build app
COPY aspnetapp/. ./aspnetapp/
WORKDIR /source/aspnetapp
RUN dotnet publish -c release -o /app --no-restore

# final stage/image
FROM mcr.microsoft.com/dotnet/aspnet:5.0
WORKDIR /app
COPY --from=build /app ./
ENTRYPOINT ["dotnet", "aspnetapp.dll"]
```

Start from an already existing container which has NET5 SDK.

Restore the nuGet packages.

Copy all the source codes and build them.

When the container is started, the dotnet command will run the app.

Running of a Docker container

- **Building the container**
  - docker image build –t myusername/app:latest .
- **Running the container**
  - docker container run –p 443:443 --name myAppName –d myusername/app:latest
- What can be done with a container?
  - Delete (docker container rm myAppName)
  - Start it in multiple instances (does it makes sense on 1 machine?)
  - Start it in multiple instances across multiple physical computers
    - For that: Docker Swarm / Kubernetes orchestrator
    - Scaling: if we monitor the response time of the container, CPU and RAM usage then the reaction can be to create a new container and redirect half of the requests to this new one

How does a good webapp looks like today?

- Layered correctly (SOLID principles, backend + frontend APIs)
- Developed using SCM like Git
- There are releases with versioning
- Merge to master/production has events for the build server
- The build server downloads, tests, builds and creates a new docker image out of that
- Moreover:
  - We have servers at some of the Cloud providers
  - The containers form a cluster
  - Disposes the old and runs the new ones
  - User will see new features immediately
  - The system can be scaled automatically even if 1M visitor appears out of nowhere
  - If there is any error no big problem → rollback using Git to a previous commit / version

Summary

- Creating such a complex full-stack application and it’s infrastructure is not an easy topic.
- It requires deep knowledge in multiple fields regarding the IT world.
- These fundamentals could not be learned within a few minutes from a YT video.
- This is one of the reasons why we learn at university, to learn how to think and plan such complex systems.

Thanks for your attention!

Sipos Miklós
sipos.miklos@nik.uni-obuda.hu