FLEX - Deployment

Simon Härer

Created: 2024-06-10 Mon 07:48

Deployment

After automating tests, the services are ready for deployment. Again, to get the most out of our independent setup, we want to fully automatize deployment.

Deployment Rate as System Quality

"We found that external approvals were negatively correlated with lead time, deployment frequency, and restore time, and had no correlation with change fail rate. In short, approval by an external body (such as a manager or CAB) simply doesn't work to increase the stability of production systems, measured by the time to restore service and change fail rate. However, it certainly slows things down. It is, in fact, worse than having no change approval process at all."

N. Forsgren, J. Humble, G. Kim: *Accelerate*, IT Revolutions, 2018

Automation as Key

Automation improves:

- reproducibility
- predictability
- quality of process results

Automated Deployment

Automation in deployment is possible due to a variety of tools and techniques that can be combined variously:

- Virtualization
- Scripts
- Tools
- Package Manager

Virtualization as Key

(Lightweight) virtualization makes the following possible:

- Reproducible deployment
- Fast immutable deployment
- Platform-agnostic containerization
- Orchestraction using containers

Automated Deployment with Scripts

- scripts are used to install service on a host
- scripts use system tools
- what happens when we call an installation script multiple times on the host? An update of a service is different from installation!

Deployment Artifacts

- executable file
- operating-system package
- Docker image
- virtual machine

Idempotent Automated Deployment

Idempotency in deployment means, that

- applying the same steps multiple times, results in the same systems state
- E.g. we say: "We need package XY" and not "Install package XY". Tools like Puppet check if the package is already available and does nothing if so.
- multiple nodes can be constantly kept on the same system state

Automated Deployment based on Immutable Servers

- every time the service gets released, the server is initiated from zero
- enables reproducibility in releases, it is *install only*
- data is persisted over multiple releases using virtual hard drive

Automated Deployment with Deployment Tools

Popular examples for deployment tools are Puppet, Chef, Ansible or Salt.

- special scripts describe what the system should look like after installation
- deployment tools takes care of what needs to be changed:
  - first run installs everything
  - second run with same parameters does nothing
  - another run with changed parameters adapts the system
- automate deployment of multiple hosts

Automated Deployment with Package Managers

Linux-based operating systems come with package manager that allows installation of pre-package applications. Configuration is a problem because

- it is often done at installation via scripts to generate configuration-files
- based on files that need to be shipped separately

A solution is shipping multiple preconfigured versions of a service

Risks of Deployment

Although microservice reduce the complexity in deployment, there are risks:

- data-schema changes, migrations in the services can cause inconsistencies
- introduction of bugs
- misconfiguration

Thus, there are different deployment strategies to handle the risks.

Deployment Strategies

- Rollback
- Roll Forward
- Continuous Deployment
- Blue/Green Deployment
- Canary Releasing
- Blind Microservices

Rollback

Service gets rolled back using an old version of the service

- restores the previous state
- difficult when data migrations have been applied

Roll Forward

Error is fixed fast and the next version of the service is deployed

- often only small changes in logic or configuration needed as a fix
- supported by fast testing & deployment in microservices
- in case of a bug it is hard to estimate how long it takes to get it solved

Deployment Pipelines

We can extend the testing pipeline of a deployment phase, after all test run through. Using that a service can be deployed in a pre-production system and integrated with other services, for click & other tests. We call it *continuous integration*.

Going one step further would be to directly deploy the system into production without any manual testing/action needed (*continuous delivery*).

Continuous Integration

- Components of the whole system are integrated routinely as part of the development process
- Close to production environment, *"pre-prod"*
- System can work as a whole, tests exceeding system boundaries are possible

Continous Delivery

Continuous Integration *and*:

- automatically produce a release that can be deployed whenever CI happens

Continuous Deployment

In *continuous delivery* every change that runs through the build and test pipelines is automatically deployed:

- requires continuous integration
- reduces the time for releases even more
- smaller changes between releases leads to faster bug finding and fixing
- forces teams to test code properly and write high quality code
- no manual tests are demanded, theoretically possible without a continuous integration (pre-prod) environment

Deployment & Development Environment

With virtualization and immutable servers it is easy to reproduce big parts of the deployment system in a local development setup. This has multiple benefits:

- Complex parts of the application can be evaluated and tested
- Debugging and replaying errors in production is possible
- Technological performance bottlenecks can be identified to some certain degree

Blue/Green Deployment

Creates a parallel environment where the new deployment lives beside the service in production. Once the team is sure everything is fine, this parallel environment becomes the production environment

- in case of an unforeseen error in production, it can easily be switched to the old environment
- hard to coordinate database changes in case of a migration between green and blue deployment

Canary Releasing

Few instances of a service in a cluster are updated to the new version. After checking the correct behavior the majority follows.

- easy testability and feedback, even including feedback from real users
- features must be able to run in parallel
- coordination of database migration is hard

Blind Microservices

Microservice are deployed parallel in production and all requests are handled by the old and new system. Responses of the new version are muted until the correct behavior is verified.

- requires extensive logging and monitoring
- features must be able to run in parallel
- coordination of database migration is hard

Controlling Deployment

Sometimes services need to be restarted manually or in case of an emergency, configuration must be adapted quickly. This can be achieved by various mechanisms:

- restart the virtual machine a service is running on
- the service is based on OS-supported service mechanisms, which can be used to control it
- the service offers an interface to control it and its configuration

Infrastructure

Having one physical server for each running service is not only impractical, but also expensive and inflexible when it comes to scaling the system. Virtualization is one solution to solve this shortcomings. Multiple service can run on one physical service and still live in their own environment.

Another possibility is to not own physical hardware, but deploy the services in the cloud. There are various providers.

Software as a Service (SaaS)

- Offers a specific software service
- No administrative overhead
- Low initial investment risk
- Dependency to service provider
- Examples are: DynamoDB, ...

Wikipedia, Platform as a Service

Infrastructure as a Service (IaaS)

- Infrastructure for running and provisioning application
- Provides infrastructure through an abstraction hiding low-level details
- Often provides cloud orchestration
- Mostly on-demand pricing model
- Examples are Amazon EC2, Azure IaaS, Google Cloud, ...

Platform as a Service (PaaS)

- Provision of runtime or development platform
- Configuration using API, no direct access to OS
- Builds and deploys automatically when there is new code
- Scales automatically
- Provides IaaS
- Restricts technology choice

PaaS Examples

- Heroku
- Google App Engine
- Azure Cloud Services
- Amazon Lambda
- Amazon EC2

Docker

- semi-virtualization, uses kernel of host
- thus, Linux-based containers only
- also available on macOS, Windows, where it runs containers in a Linux VM
- communication between containers using Docker port exports and a virtual network
- Docker registry allows it to store and download premade images

Docker in Production

We can benefit from using Docker in production in different setups:

- multiple Docker containers on one server
- Docker executed directly in a cluster (scheduler needed, e.g. Apache Mesos)
- Kubernetes distributes Pods in a cluster. A pod contains Docker containers. Kubernetes is a container orchestration tool that offers distribution, deployment and easy scaling for services based on Docker.

Podman

- POD manager
- alternative to Docker
- daemonless

Technology has a large impact on deployment and, thus, must be considered carefully.

Infrastructure as Code

As we learned in the previous testing chapter, we have greate benefits from describing infrastructure as code:

- repeatability
- automation
- is maintained as normal code (revisioned, reviewed, ...)

Deployment Complexity as System Quality

Infrastructure related code is seen as normal code. This implies, that the complexity of this code (not necessarily the complexity of the infrastructure) becomes a quality of our system. If the code used for infrastructure gets too complex it gets hard to maintain and prone to errors. This influences the project's time to market and robustness.

Infrastructure as Code: docker-Example

Configuration utilizing environment variables

```
statements:
  environment:
    - NODE_NAME=statements
    - ACCOUNTS_HOST=accounts@accounts
    - TRANSFERS_HOST=transfers@transfers
    - LOG_LEVEL=info
  hostname: statements
  container_name: statements
  command: ["deploy-with-elk"]
  build: statements/
  depends_on:
    - transfers

    networks:
      - net1
```

DevOps

With teams with end-to-end responsibility and infrastructure as code, the role DevOps arises:

- A developer that is able to do operations
- DevOps describes the fusion between development and operations–pure operations doesn't exist anymore
- In SCS everyone should be a DevOps, since everyone on the team should be able to maintain every part of the application

DevOps Role and Organisation

- With the DevOps role, there is no need for a horizontal infrastracture team
- Every team manages its own infrastructure means that resources and budgets for ops are team specific
- Scaling infrastructure can be done on every teams behalf

Dependencies

- Deployment repository is always centralized
- Deployment repository mirrors application structure, thus team structure
- Which team is responsible for deployment?

Tools

- GitLab CI
- GitHub CI
- Jenkins
- Chef, Puppet, Ansible
- Nix
- Kubernetes
- ...

Practice : *Demonstration of deployment with docker compose*

We show how docker compose can be used to orchestrate a bunch of services and start a local setup for development or deployment. This includes:

- Banking services
- Nginx for frontend integration
- OpenSearch, Logstash and OpenSearch Dashboard for Logging

Exercise: Deployment Strategy

Sketch an alternativ deployment strategy for Erlbank with technological alternatives for:

- reproducible OS environment
- deployment artifact format
- service configuration

Weigh these approaches against the current choices.

OS configuration

- ZooKeeper: more heavyweight, more flexible
- shared filesystem: easier to use, but harder to setup
- scp: less coupling, but fragile