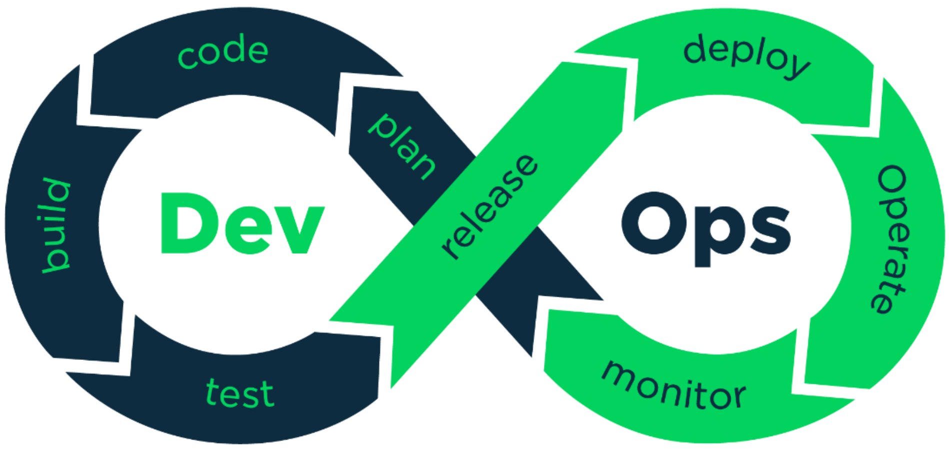
CONFIDENTIAL & RESTRICTED

DevOps for Noobs



A methodology that helps engineering teams build products by continuously getting user feedback.

DevOps Engineering is the practical use of DevOps within Software engineering teams. A DevOps Engineer should be able to build, test, release and monitor applications.

DevOps Engineering Pillars

1. **Pull Request Automation**
2. **Deployment Automation**
3. **Application Performance Management**

The goal of DevOps Engineering is to release high quality software quickly, and make sure it continues being delightful and bug-free for end users.

1. **Pull Request Automation**
   The business goal for a DevOps engineer working on pull request automation is to speed up the PR review process. How can it be achieved?
   - Automated test running with CI (Continuous Integration), which gives developers confidence that the change does not break existing functionality.
   - Per-change ephemeral environments, which helps interested parties actually interact with the proposed change to ensure it solves all required business goals.
   - Automated security scanning, which helps ensure that proposed changes do not introduce new vulnerabilities into the product.
   - Notifications to reviewers automatically, so that the correct reviewers can quickly request changes to a pull request.
2. **Deployment Automation**
   The efficiency of the build process isn't the only goal of Deployment automation, other goals include:
   - Deploying a new feature to a certain set of users as a final test before rolling it out more publicly (feature flagging and canary deployments)

- Starting new versions of services without causing downtime (blue/green deployments and rolling deployments)
- Rolling back to the prior version in case something does go wrong.

Broadly, success in deployment automation is finding the appropriate deployment tools to fulfill business goals and configuring them. In an ideal world, there should be little-to-no custom code for deploying.

3. **Application Performance Management**

   The core goal of this pillar is to ensure that your service continues to perform well in production. It has 4 core targets for automation:
   - Metrics: Numerical measurements of key numbers in production. Usually in the form of finite resources (disk space, memory usage) and times (average response time, job processing time, etc.)
   - Logging: Text descriptions of what is happening during processing. Logs often come with metadata about their source, time, and related metrics.
   - Monitoring: Take the metrics and logs and convert them into health metrics. Does the product feel slow, is it down, are all of the features working without errors?
   - Alerting: If monitoring detects a problem, the correct problem solver should be notifies automatically: in case of an outage, an on-call engineer should be notified. Performance issues and errors with features often automatically log tickets.

Test Driven Development

Test Driven Development (TDD) is a coding methodology where tests are written before the code is written. 3 steps

1. Choose something to work on
2. Write tests that *would* pass if the software worked
3. Keep building until all the tests pass

Continuous Integration (CI)

CI refers to developers continuously pushing small changes to a central Git repository numerous times per day. These changes are verified by automated software that runs comprehensive tests and ensures that no major issues are ever seen by customers.

How to integrate CI into my development process?

The following 4-step approach is most common:

1. Developers work on a feature branch and push at least a commit a day to that branch on a central git repository
2. After every commit to the feature branch, a CI server picks up the commits from the repo and runs tests, reporting any errors back to the developers of the code
3. Once the feature is complete, the developers open a "pull request" to have their changes be merged into the "master" branch. Another developer on the same team makes sure that the feature makes sense, and that there are no stylistic issues in a "code review."

CONFIDENTIAL & RESTRICTED

4. At some point, the "master" branch gets "prod pushed" (pushed to production) so that end users see the code. This might happen automatically in CD (continuous deployment) scenarios but is often triggered manually for smaller projects.

Code Coverage

A quantitative measure how comprehensive a code base's test are. Increasing code coverage often increases stability and reduces bugs.

Branch Coverage – Instead of measuring how many lines of code, it measures groups of lines. Instead of looking at individual lines of code, we could instead look at branches: body of loops, body of if statements, body of the main functions etc.

When to care about code coverage?

1. Your product has users, and those users might leave if they are affected by bugs
2. You are working with developers that aren't immediately trustworthy like contractors or interns
3. You are working on a very large code base with many individually testable components – here code coverage analysis can complement Test Driven Development as a project management tool

Also, do not force developers to write 2-5 unit tests for every new feature in order to maintain 100% code coverage, that would be time consuming, which is not the point of DevOps.

Linting

Linters look at a program's source code and find problems automatically. They are a common feature of pull request automation because they ensure that "obvious" bugs do not make it to production.

Follow this for good coding practices: https://google.github.io/styleguide/

The Nit Approach

Code reviewers leave little comments on the code called "nits" that the team can ignore until broader reviews. Nits are helpful as future references but prevent blocking important changes.

Ephemeral Environments

Temporary deployments that have self-contained versions of your application, generally every feature branch.

- Accelerates software development life cycle.
- Allows developers to share changes with designers, managers and other stakeholders.

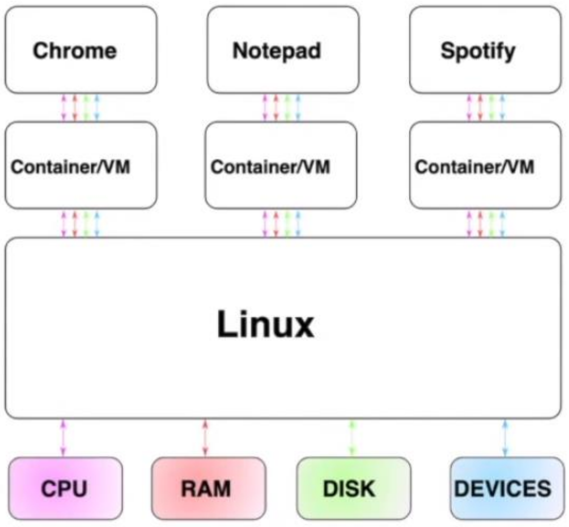
How do databases work in ephemeral environments?

1. Prepopulated data – it contains representative, anonymized data – to pass security audits, all Personally-Identifiable Information (PII) must be scrubbed from databases to be used
2. Undoable – if data is deleted or modified, it should be easy to reset the database to its original state.
3. Migrated – Database uses the schema currently used in production, and has proposed migrations run against it.

Continuous Staging

CI/CD is merged with ephemeral environments to form a unified CI/CD and review process for every commit.

Virtual Machines and Containers (Docker)



Nothing much has changed except for each program is now in a container/VM, so they are "farther"

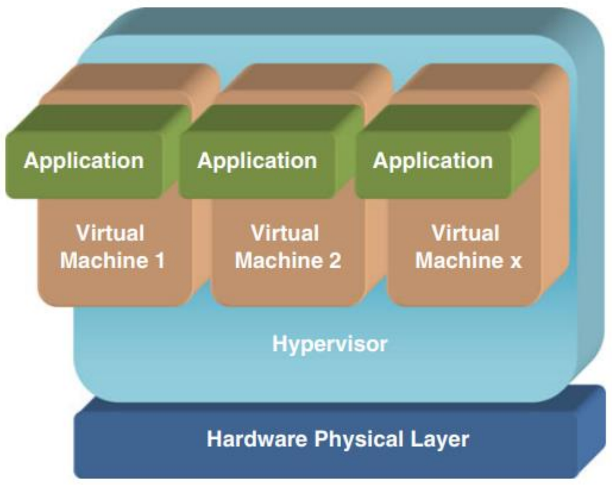
How Containers work?

Containers work by creating "namespaces", which are a Linux feature that group shared resources together. For example, if you had five processes running together within a Docker container, they'd still be running within Linux itself.

How VMs work?

If the idea of containers was to provide a "fake" version of Linux, the idea for VMs is to provide "fake" versions of the CPU, RAM, disk and devices. That is, VMs are "faking" one level deeper.

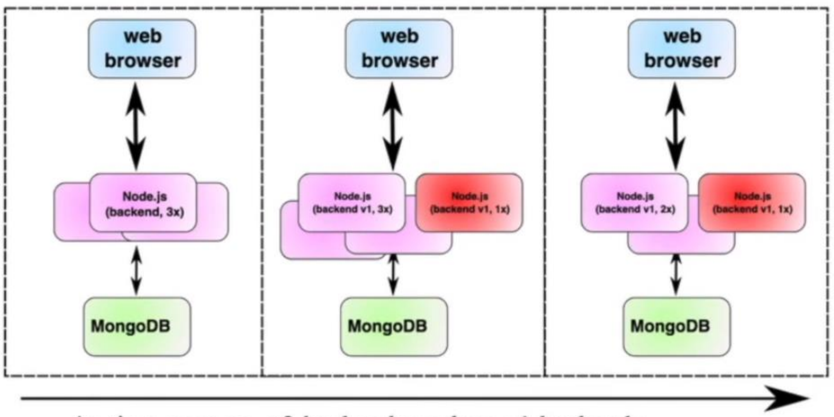
This is done by **Hypervisor**.



Rolling Deployments

Strategy to deploy a new version of an application without causing downtime. They work by creating a single instance of the new version of an application, then shutting off one instance of the old version until all instances have been upgraded.

Pros	Cons
• Well supported • No huge bursts • Easily reverted	• Speed • API Compatibility



As time goes on, v2 backends replace v1 backends.

Blue/Green Deployments

Strategy to deploy a new version of an application. They work by starting an entirely new instance of the application and then routing traffic over to it.

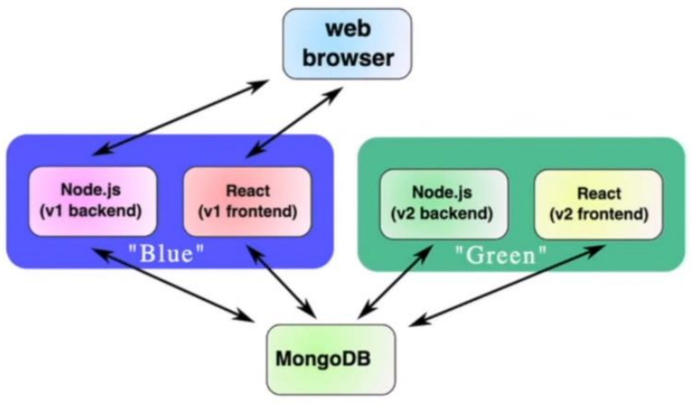


Diagram of a blue/green deployment - "blue" and "green" are versions of the application

Pros	Cons
• Easy to understand • Powerful • Extendable to workflows	• Difficult to make hotfixes • Resource allocation is not convenient • Clusters can affect each other

Rainbow Deployments

Have more than the two deployment clusters (blue/green). Instead, teams have many clusters ready (blue/green/yellow/red etc.)

In a rainbow deployment, old clusters would only be shut off after all of their long-running jobs were done processing.

Acceptance Tests

New version of the app could be tested against the production database in the very environment which will soon become production.

QA and product stakeholders can choose to “accept” the developer team’s latest release.

Canary Deployments

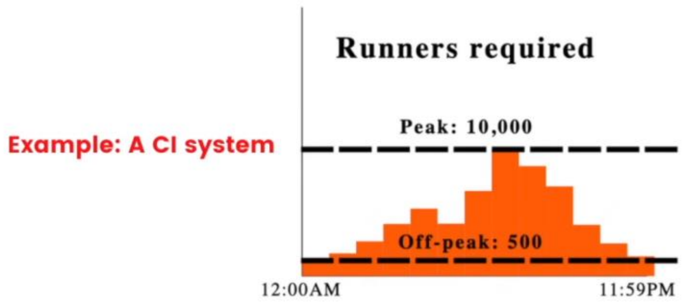
In the context of blue/green deployments, a “canary deployment” would be to route around 5% of your users at random to the new cluster, and check that those users do not have negative feedback.

Blue/Green deployments are a powerful and extensible deployment strategy that works well with teams that are deploying a few times per day. The strategy only starts being problematic in continuous deployment scenarios where there are many services being deployed many times per day.

What is Autoscaling?

Autoscaling automates horizontal scaling to ensure that the number of workers is proportional to the load on the system.

CONFIDENTIAL & RESTRICTED



Most popular Autoscaling services:

- Aws EC2 Spot Instances
- Kubernetes horizontal pod autoscaling

Serverless vs Autoscaling

Autoscaling is usually discussed on the timeline of ~1-hour chunks of work. If you took the concept of autoscaling and took it to its limit, you'd get serverless, which is to define resources that are quickly started and use them on the timeline of ~100ms.

Use Case for Serverless

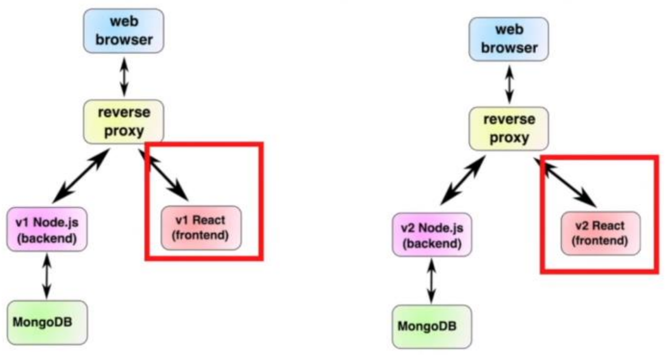
Serverless is primarily used for services that are somewhat fast to start, and stateless. You wouldn't run something like a CI run within a serverless framework, but you might run something like a webserver or notification service.

Service Discovery

It is the process of automatically detecting devices and services on a network. It can locate a network automatically making it so that there is no need for a long configuration set up process. Ex. Kubernetes Service Discovery, AWS Service Discovery etc.

There are 2 types of Service Discovery:

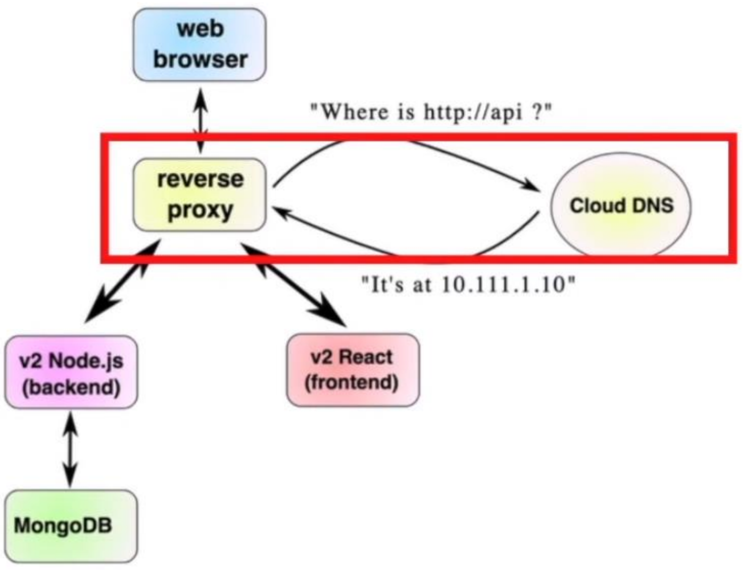
1. Server-side service discovery – It allows clients applications to find services through a reverse proxy.
2. Client-side service discovery – It allows clients applications to find services by looking through or querying a service registry, in which service instances and endpoints are all within the service registry.



DNS Based Service Discovery

The idea of DNS is just to map hostnames to IPs. This is the industry standard. There is a global DNS and an internal DNS.

It would be ideal if we could connect to http://frontend from our reverse proxy and have DNS respond with the IP for the correct version(s) of the frontend.



If you configure service discovery in an appropriate manner for your deployment (e.g., dns-based for a Kubernetes cluster), it makes it significantly easier for developers to have microservices talk to each other.

Instead of a developer having to write

Connect to MongoDB at 'mongodb://'+process.env("MONGODB_IP"]+':27017/myapp'

They can simply say

Connect to MongoDB at 'mongodb://mongo:27017/myapp'

Decoupling the application logic from the deployment logic. = BIG BRAIN

Application Performance Management

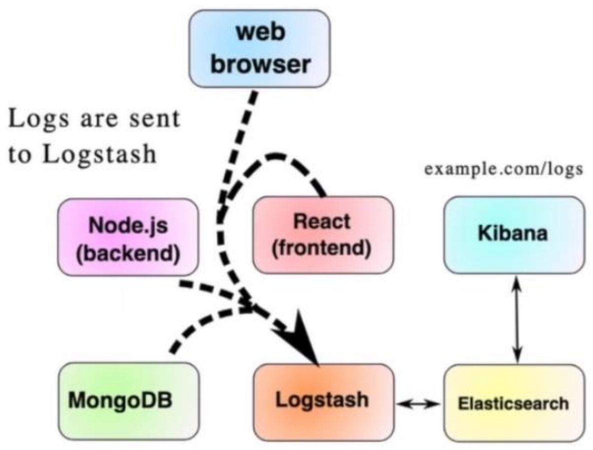
Log Aggregation

It is a way of collecting and tagging application logs from many different services into a single dashboard that can easily be searched.

Essence of a good log aggregation platform: Efficiently collect logs from everywhere that emits them and make them easily searchable in case of a fault.

Example, ELK (Elasticsearch, Logstash and Kibana)

"ELK" is the acronym for three open-source projects: Elasticsearch, Logstash, and Kibana. Elasticsearch is a search and analytics engine. Logstash is a server-side data processing pipeline that ingests data from multiple sources simultaneously, transforms it, and then sends it to a "stash" like Elasticsearch. Kibana lets users visualize data with charts and graphs in Elasticsearch.



3 components of a Log Aggregation Platform

1. The Log processor (Logstash)
   The log processor takes log lines and parses data out of them. For example, a log line like,
   1997/01/01 18:03UTC HTTP/1.1 GET/
   Might be processed into the object
   {"date": "01-01-1997 18:03:00", "service": "backend", "msg": "HTTP/1.1 GET/" }
2. The Data store (Elasticsearch)
3. The log frontend (Kibana)

Vital Production Metrics / Metric Aggregation

The are data points that tell how healthy a production system is.

If log aggregation is the first tool to set up for production monitoring, metrics monitoring should be the second.

Metric aggregation deals with numbers.

Examples, **Prometheus** is an open-source solution for metric aggregation.

What metrics to collect? Some ideas are

- Request fulfillment times
- Request counts
- Service resources

Server resource metric examples

- Database size and maximum database size
- Web server memory
- Network throughput and capacity
- TLS certificate expiry time and life left