# Running containers in production

Anurag Bhatia, Hurricane Electric

## Introduction

Working at Global backbone operator & datacenter provider - Hurricane Electric and based in Haryana, India. Spend lot of time in looking at BGP routing tables, traceroutes across the ocean, interesting patterns & tooling around those patterns.

Besides routing I got lot of interest in DNS, root DNS servers, network automation & virtualization.

And dad since Sept 2021! ;-)

## Quick housekeeping announcements...

1. Slides here is just reference material. There will be a bit of live screen demo.
2. Slides can be found on - https://bit.ly/3dbrORx & linked to APNIC 54 site (soon!)
3. Sample code / playbooks / Gitlab CI jobs - here
4. Sample code / playbooks for IX RS Automation via Git - here
5. Time for Q&A in the end

Slides

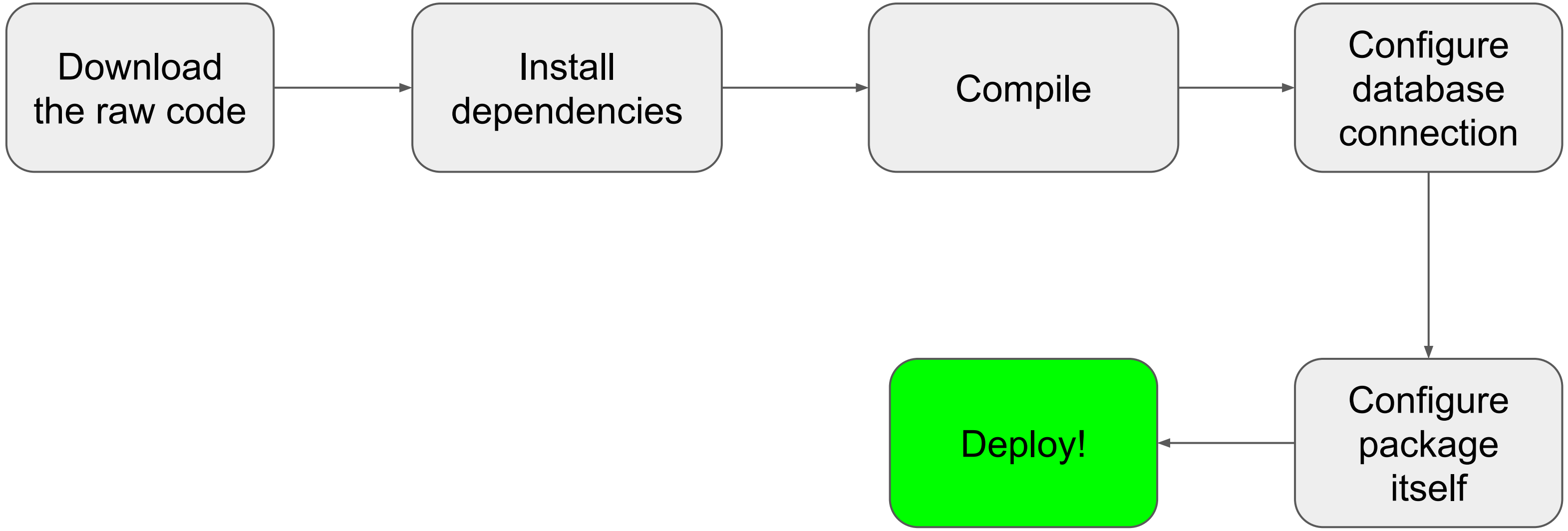
## Part 1 - Introduction to containers

Part 2 - Making use of CI/CD pipelines

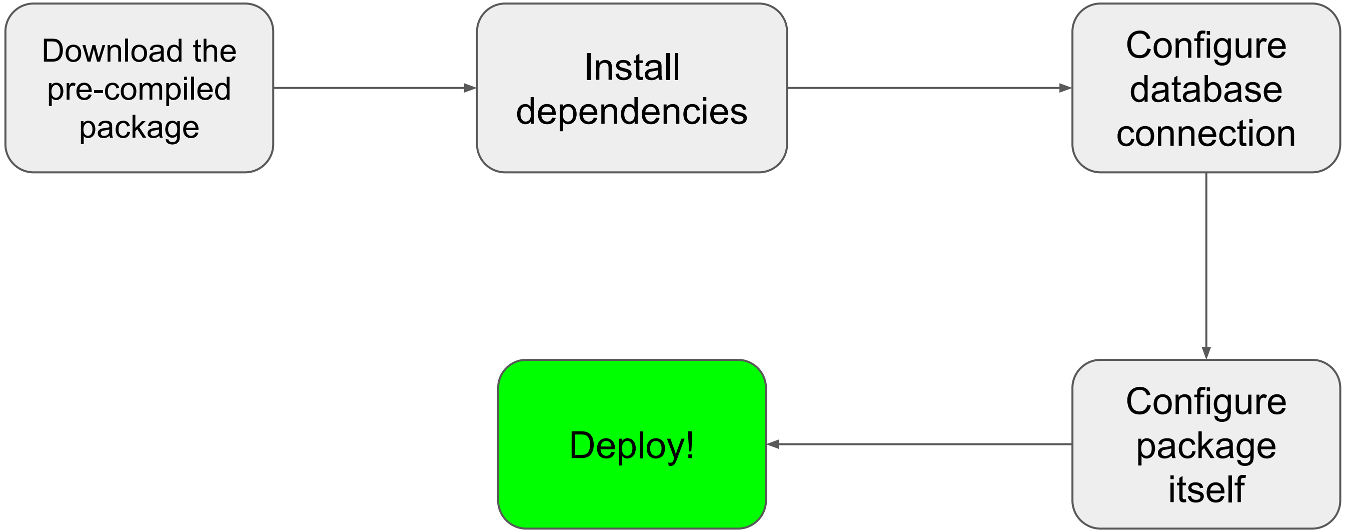
Part 3 - SSL + Backups + Monitoring

## ~~What is Docker?~~
## How software deployment traditionally works?

## Typical traditional software deployment (method 1)



## Typical traditional software deployment (method 2)


Download the pre-compiled package
Install dependencies
Configure database connection
Configure package itself
Deploy!

## Challenges with traditional software deployment

1. Takes time to go through documentation, install package and maintain it
2. Migration of software package to a different server requires almost as much effort as to setup fresh
3. Chance of old problem of “*it works on my system*” as claimed by developer but not on your system!
4. One application may dependent on one version of some binary, other application may want a different version. Thus dependency conflict can occur.

## Screenshare to walk through instructions for setup of Smokeping, Librenms, Zabbix etc

How to set up (in traditional way):

1. Smokeping
2. Librenms

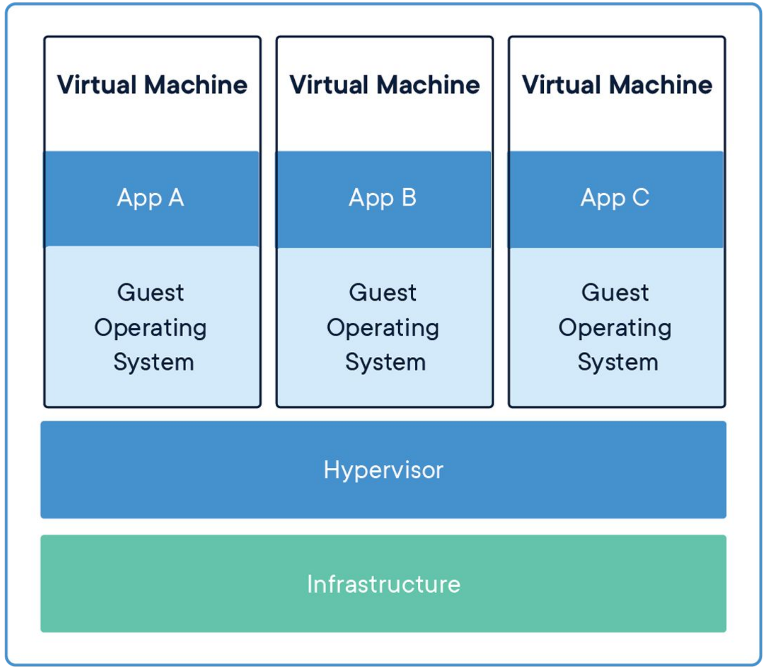
## Introduction to containers

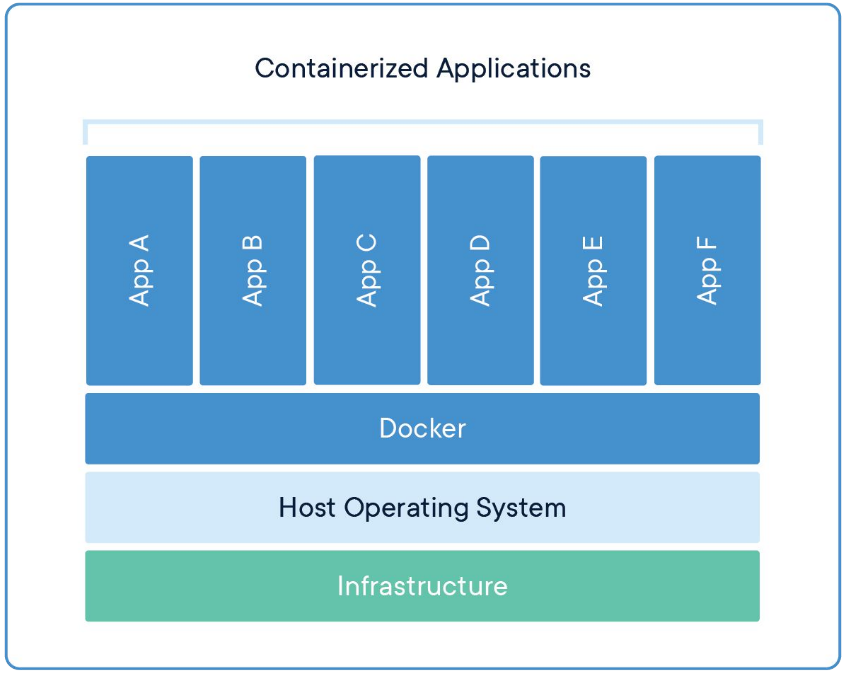
## What are Linux Containers?

A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another. It also typically has the configuration for dependencies to interact as needed for the working of software.

These are **not** virtual machine though at times might feel very similar.  They are extremely lightweight as compared to virtual machines, have low overhead and spin up within seconds instead of minutes. Containers share same Linux kernel & work on the principle of namespaces in Linux.

## VM Vs Container





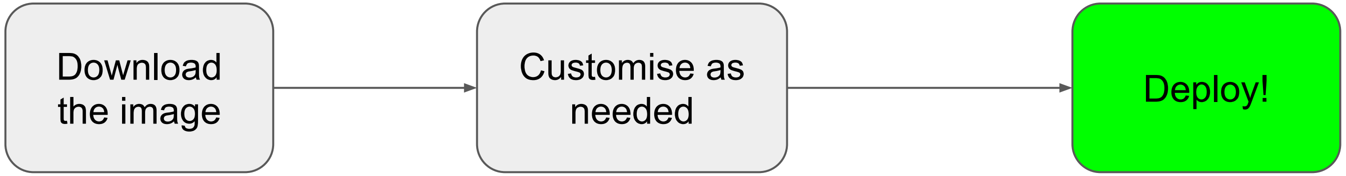
## What is docker?

One of few container technologies.

It’s one of most popular ones. One needs to have “Docker Engine” installed on the machine to run docker containers. It supports Linux, Windows, Mac etc.

Because docker is so popular, many app developers these days ship full fledged working containers with all needed binaries, connections, configurations to use them. That makes deployment extremely fast, easy to manage and update.

## Container based software deployment



# Docker terms to remember

1. Docker Engine - that’s the “engine” which runs on that host on which docker containers run

2. Docker Containers - Essentially the containers which run the applications

3. Images - These are pre-packages images and can be used to spin up a container

4. Container registry - A registry service hosts container images

5. Networks - By default Docker Engine create a bridged network docker0 and that is available for connecting any newly created containers via ad-hoc command. One can create more networks. One can bind ports of host to a port on container. E.g port 80 on host can be mapped to port 8080 of container (DNAT)

6. Volumes - One can create a docker volume and attach to a given container for persistent storage

7. Bind mounts - One can mount an existing location on host machine to a location on container

## Basic docker commands

1. docker container list - Lists all running containers
2. docker container list -a - Lists all containers
3. docker network list - Lists docker networks present on the host
4. docker image list - Lists local images present on the docker host
5. docker run --name=A image:latest - Creates container with with name “A” and image with the tag “latest”.
6. docker container stop <container-name> - Stop container
7. docker container start <container-name> - Start container
8. docker container rm <container-name>  - Remove container

## Ideas behind running docker containers

- One can get pre-packaged container images & simply spin them up to create containers.
- Containers **do not** store any unique processed data inside the container
- Typically if an app has dependencies like requirement of database engine etc then those extra services are configured in their own different containers
- Port binding is simply destination NAT
- Typically containers which are not needed to be exposed to the world are not exposed to the world for security reasons
- Typically one does not updates a container but instead just downloads new/fresh containers image & replace the old one.

## Environment variables

- Variables defining certain features as supported by the developer
- Can be provided in ad-hoc command or docker-compose or .env file
- Very often included by the developer on instruction page
- Enables features like providing username/password, time zone, etc without having to go inside the container

## Smokeping via ad-hoc command

```
docker run -d \
  --name=smokeping \
  -e PUID=1000 \
  -e PGID=1000 \
  -e TZ=Europe/London \
  -p 80:80 \
  -v </path/to/smokeping/config>:/config \
  -v </path/to/smokeping/data>:/data \
  --restart unless-stopped \
    ghcr.io/linuxserver/smokeping
```

Source: https://hub.docker.com/r/linuxserver/smokeping

## Challenge with docker ad-hoc commands

- Gets complicated when passing long arguments
- Becomes hard to reproduce a container as one has to remember/keep track of the command used
- If running a complex application needed 3 container then three ad-hoc commands have to be executed
- By default connects all containers to default docker bridge & that does not create network isolation.

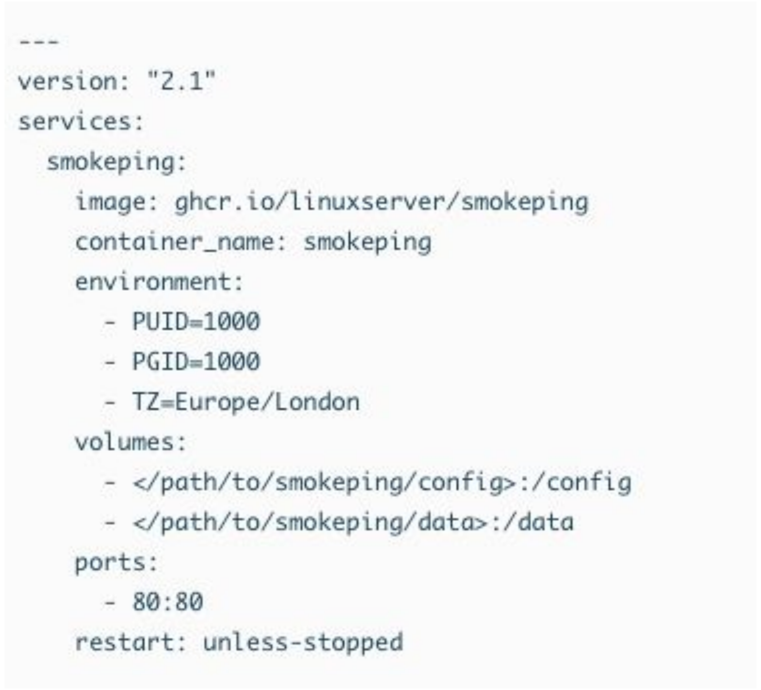
## Introducing docker-compose

## docker-compose.yml

- Single text file in yaml format. Easy to read and include all the instructions needed to setup container(s)
- Includes link to container images
- Includes details about network to attach to. If none-specified, new network is automatically created
- Includes details of volumes to create/attach to or bind mounts to bind to
- Makes it very easy to upgrade containers to their latest images

## Live demo of smokeping via docker-compose

## Smokeping via docker-compose.yml



```
---
version: "2.1"
services:
  smokeping:
    image: ghcr.io/linuxserver/smokeping
    container_name: smokeping
    environment:
      - PUID=1000
      - PGID=1000
      - TZ=Europe/London
    volumes:
      - </path/to/smokeping/config>:/config
      - </path/to/smokeping/data>:/data
    ports:
      - 80:80
    restart: unless-stopped
```

Source: https://hub.docker.com/r/linuxserver/smokeping

## Live demo for setup of Uptime Kuma

(Instructions here)

## Live demo for setup of LibreNMS

(Instructions here)

## How these containers are created?

- Containers are created using some sort of base image OS like Ubuntu, CentOS, Alpine etc. Alpine is smallest image ~ 4MB and has almost nothing installed by default making it very secure.
- Developer creates “Dockerfile” with reference to base image & all the commands which are needed to create that software environment like install x,y,z packages, copy script1, create user etc.
- More on this later today!

## From where should I get container images?

- If you have heard of the developer, know them well then try to get their official container images for security and trust reasons

- If official image is not available, review the Dockerfile of the image published by non-official source

- If it’s highly secured environment, build own images based on reviewed dockerfile (though it’s OK to use official images as long as one trusts the developer)

- Docker hub for searching for projects: hub.docker.com

## Note on IPv6

- Docker does supports IPv6 however some manual config is needed
- When giving publicly routed IPv6 to each container, one should ensure that IPv6 firewall is configured on expected lines.
- For now widely adopted practice is to dual stack frontend proxy container and let it “speak” to backend containers over IPv4. Though this is expected to change over time where containers are IPv6 only.

## Part 2 - Making use of CI/CD pipelines

~~Part 1 - Introduction to containers~~

Part 3 - SSL + Backups + Monitoring

CI/CD

## What is Git & CI/CD pipeline?

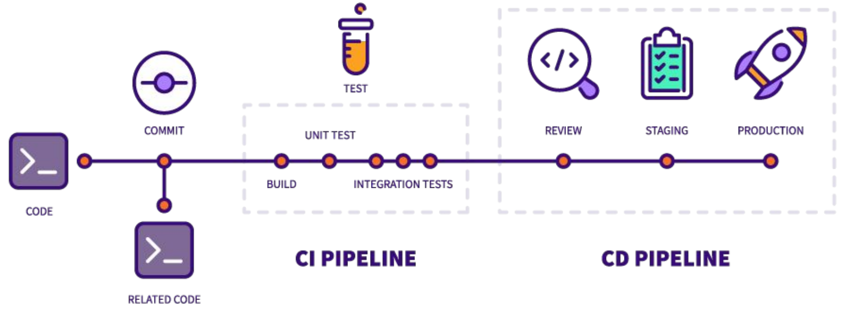
Understanding concept from the programming world...

## Basic idea of Git

- Allows versioning of code, making it possible for teams to work together
- Allows creating “branches” from code to try/test different things & merge branches
- Allows working locally and pushes changes to centrally hosted Git repo to be shared
- Makes complicated software development easier. Earlier tools with similar features Git, CVS, SVN etc

## What is CI/CD

- Continuous Integration (CI)
- Continuous Delivery (CD)
- Continuous Deployment (CD)



Term comes from Gitlab ecosystem but concept is also used across various hosted Git options (e.g Github actions in Github)

Image source: Gitlab

## Continuous Integration (CI), Continuous Delivery (CD) & Continuous Deployment (CD)

- Actions happen when code is pushed to Git
- Can run tests & allow commit only if certain tests are passed
- Helps in adding checks & balance in code development
- All of it can be automated or manual or a mix of it depending on the need

## But where does the code run?

## Where does code runs?

- Code typically runs inside a docker container as a job
- One can use available popular containers like alpine, ubuntu, centos, or application specific containers
- One can also build own container using base image of any of the available containers if installing multiple packages

## And where does Job runs?

## Ofcourse in Netherlands....! :-)

## And where does a job runs?

- Typically in a runner - can be shared runner offered by popular hosted Git providers like Gitlab, Github etc and also dedicated runners which you can host on your machine (desktop/server)

- Runner can be a program installed & running on machine or simply a docker image with special permissions

- One can have multiple runners configured in a project & use them as needed across various tasks. E.g task 1 on runner on server1, task 2 on runner on server2 etc

## Utilising docker for running Gitlab jobs

- For running multiple jobs efficiently, it’s more efficient to create container with all needed packages.
- Containers can be stored within Gitlab’s infrastructure (Container registry)
- Containers stay cached at the runners & hence after 1st run, it’s usually fast
- For executing commands on routers, one can have alpine container with Ansible installed & configured
- Jobs are independent in nature but if needed data can be stored & passed along using Gitlab artifacts

## How to trigger a job?

- Automatically upon commit, change in a specific file etc
- Automatically at a given point in time as per schedule
- Manually from Web UI
- Automatically or manually via REST API call to Gitlab

## Some demo cases of CI/CD pipelines...

- Deploy & manage DNS servers
- Add/remove user accounts from routers, servers, switches etc
- Update software packages, containers etc
- Manage IXP route server out of Gitlab

Anything else....which can be fully or semi-automated

## Live demo for Gitlab CI/CD Pipelines

## Part 3 - SSL + Backups + Monitoring

~~Part 1 - Introduction to containers~~
~~Part 2 - Making use of CI/CD pipelines~~

## Concept of reverse proxy

1. Sits between outside world and your containers
2. Can map all applications running on any of the ports: 8081, 8092, 9000 etc to a DNS hostname on port 80 and 443
3. Makes it easy to deploy SSL certificates
4. Makes it easy to dual stack public facing services and thus apps become available on IPv6 as well

## Notes on backup

1. Docker containers are reproducible. No need to backup containers unless you have created custom containers.

2. The actual user data is stored using volumes or bind-mounts. In both cases ensure those locations are backed up.

3. One can use tools like Restic, duplicati *(comes as a docker container as well)* and mount required locations for it to backup.

4. Data storage is extremely cheap these days ~ $0.005/GB/month. Thus for low to mid sized storage, one can explore that.

5. If self-hosting backup servers, one can make use of Minio to expose internal storage with S3 endpoint over HTTPs

6. Always encrypt data before storing in the cloud.
   ***Cloud is just someone else's computer!***

## Live demo of restic...

## Some cool containers to play with...

- RIPE Atlas - https://hub.docker.com/r/jamesits/ripe-atlas
- HTML 5 speedtest - https://hub.docker.com/r/adolfintel/speedtest
- iperf3 - https://hub.docker.com/r/networkstatic/iperf3
- Nextcloud - https://hub.docker.com/_/nextcloud
- Docker-speedtest-grafana - https://github.com/frdmn/docker-speedtest-grafana
- Frigate - https://github.com/blakeblackshear/frigate
- Linux-server.io - Many great images actively maintained by the open source community

## Happy automating!

## Questions?

anurag@he.net

Web: he.net