Containerization/DMAC software

Ben Adams
RPS ASA

Why should I use containers?

- Greatly assists with the "works on my machine" problem, i.e. disparities between development and production
- Allows you to ship applications in a uniform fashion, hence the moniker of containers.
- Many applications already available through the Docker library, which obviates the need to do as much configuration.

When isn't Docker a good fit?

- You have very strict security policies or
- You are attempting to distribute an operating system with default set of applications. Use a virtual machine instead.
- Certain HPC workloads? Other projects underway such as Shifter which attempt to bridge some of these deficiencies.

"Is this like a virtual machine?"

- In short, not exactly.
- Containers usually abstract away applications and are still reliant on the underlying kernel of the OS*, whereas virtual machines have to take care of abstracting whole operating systems.

*if you're running on Windows/Mac running Docker containers for Linux, you're actually running a VM under the hood to provide the features of the Linux kernel.

Containers vs virtual machines

- Lightweight, depends on kernel
- Mainly used for single applications, although "lift and shift" of several applications is possible
- Faster to boot, doesn't need to allocate separate resources like RAM, HD space. Can still enforce resource limits if desired.
- Less separation mechanisms between container host and container

- More heavyweight, abstracts an OS on top of an OS
- Used to abstract away entire operating system environments
- Slower to boot, needs to allocate system resources like RAM, HD space in advance
- Separation, security between host and guest OSes

A word on security

- A full discussion on containerization security is beyond the scope of this tutorial. However, Docker has a number of security mechanisms offered. These include chroot, cgroups, PID isolation, network/firewall rules, SELinux (disabled by default on most), and user namespace remapping.
- In particular, members of docker group should be considered essentially tantamount to root access. As an example`docker exec -it -v / --rm busybox` will give a root shell with access to root dir.
- Avoid running containers as root – use USER Dockerfile instruction to switch users. If you really must use root, you should look into user namespace remapping via .dockerremap file to remap root user to another UID/GID. See https://docs.docker.com/engine/security/userns-remap/

Security, continued

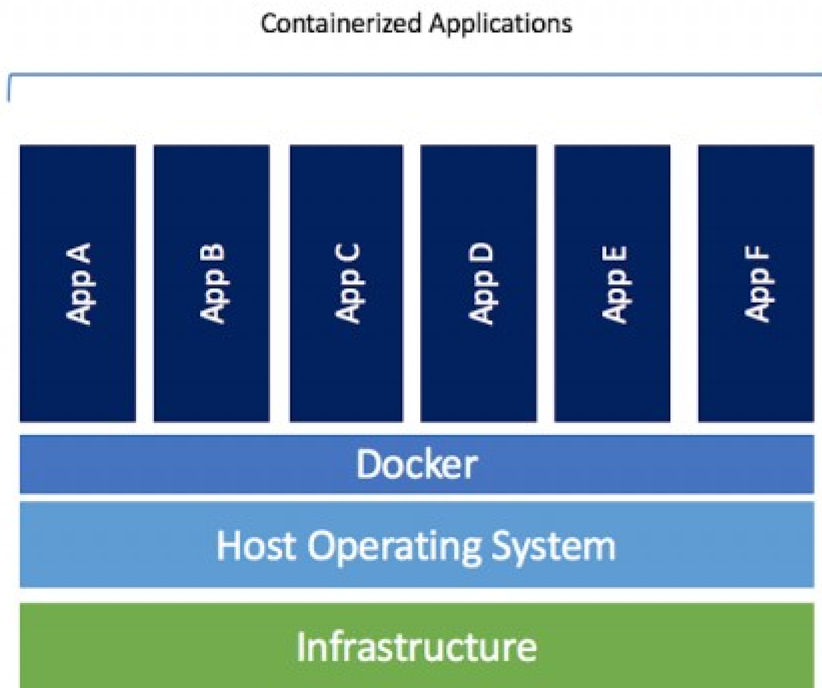
- Standard procedures still apply – check that dependencies are up to date, CVEs, etc.
- Check the build script contents and dependencies.
- Consider signing your own images and/or only using images signed by parties known to be trusted.
- You can also run Docker inside a VM so that in the event of a container escape, they only have access to the VM.

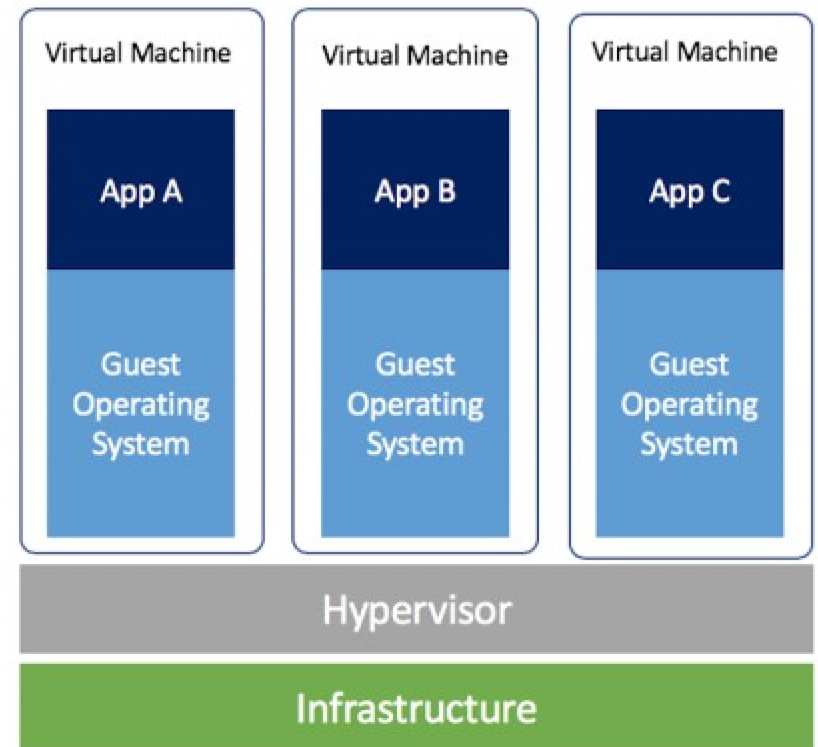
Upgrading Docker

- Always recommend using latest version of Docker from official Docker repos
- Can add `{ live-restore: true }` and `sudo systemctl daemon-reload` to prevent stop of containers upon docker daemon restart if container uptime is important. (Linux only)

Preliminaries

- Union filesystems/Copy on write (CoW)
- Image credit:





Volumes and data persisitence

- Remember when we discussed copy on write filesystems?
- By default, changes are applied to the containers' filesystem, unless a volume is specified.
- Continually writing to CoW layers is usually slower in terms of I/O performance
- Any aforementioned changes will be blown away if you destroy a container.
- Can sometimes check for container writes with `docker history`

Volumes and data persisitence

- To avoid these problems, Docker uses a persistence mechanism called volumes
- Essentially just writes to the file system and persists between writes.
- Three types of volumes – anonymous, host, and named.

Anonymous, named

- Anoynmous and named volumes both have Docker abstract away volume creation
- Can refer to named volumes more easily, also share with other containers via the reference.

Caveat emptor, bind mounts

- Bind mounts can be useful for dev, but be careful of permissions
- No guarantee on user creation having same UID/GID unless explicitly stated
- Can lack sufficient permission inside container or outside to read/write/execute files.

docker pull

- Pulls down an image from a remote location
- By default this will come from DockerHub unless you are logged into another image repository with `docker login`, such as Amazon ECR

docker build

- Builds an image from a Dockerfile
- Usually want `docker build -t <repo_name>/<image_name>:<version> <build_directory>`
- Version defaults to latest if not specified
- --cache-from , --no-cache to control build cache or ignore it entirely.

docker run

- Actually runs a container.
- Docker run [flags] <optional_cmd>
- **LOTS** of relevant flags. Can specify resource limits (mem, CPU, IO), volumes, networks, DNS, restart policy, whether to run as interactive, whether to remove container on closing.
- Try to summarize some of the more important ones

docker run (contd.)

- --rm : removes the container when exiting
- -it : **i**nteractive and allocate pseudo-**T**TY. Go-to for when you need to run things in the container
- -v [[HOST-DIR:]CONTAINER-DIR[:OPTIONS]]]: specify volumes
- --restart-policy: specify a restart policy upon container failure
- --name : assign a name to the container
- -m : set memory limits

docker exec

- Run a command in a container
- Useful invocation: `docker exec -it <container_name> <some_shell>` – create a shell in the container.
- -u/--user if you need to run something as another user (e.g. install something with the package manager)

Anatomy of a Dockerfile

- Virtually all contain FROM, RUN and CMD
- Each successfully run instruction creates a new layer in the container
- You can actually create a container from an underlying image step

The FROM Dockerfile instruction

- The FROM Dockerfile instruction indicates an image which we wish to use as a base.
- For example, we might use `FROM centos:8` to import CentOS 8 from the official Docker library files.
- You virtually always want to use FROM, unless you are building a base OS image.

Interlude: Choosing a base image

- If there's an image in the official Docker libraries that have what you need, just use it, e.g. Python, WordPress
- If it's non-official, check the build scripts to make sure they're sane – I've seen some crazy stuff
- Minimal images are lightweight, but can take a long time to build numerical libraries due to different toolchains (show up in lack of Python wheels)

LABEL

- Adds key value pair information to the image which can later be inspected, e.g. author, organization.
- LABEL <name>="<value>"

EXPOSE

- The expose instruction allows a TCP port to be exposed outside the container. This may be the outside world, or just the internal network shared by containers.
- Can either use `docker run -p <local_port>:<exposed_port> or `docker run -P`

RUN instructructions

Runs a given command under the current SHELL (default is bash under Linux) and creates a new layer under the Dockerfile

VOLUME

- Declares that a given location within the container will use a volume. By default this will use an anonymous volume if the volume is not declared with `-v` or similar mechanisms.

ADD/COPY

- Move a file from build context into
- Usually want COPY
- ADD can also extract compressed files and fetch remote files. Usually still want `RUN … curl/wget remote_file_loc && some_other_operations` for remote files
- COPY and ADD can also set file access permissions via --chmod flag (Linux containers only)
- COPY has a --from instruction which can copy files from another image. This is known as a multi-stage build. It can be useful for dependencies which require compilation of many libraries, but for which you don't wish to include the source dependencies.

ENV

- Set default environment variables
- Can be overridden
- Overriding variables declared with ENV won't break build cache on subsequent image builds.

WORKDIR

- Changes the current working directory to the directory supplied for any subsequent Dockerfile instructions
- NB: instructions like `RUN cd <some_dir>` won't work like you might expect; will change the directory for that invocation **only** and then return to the previous dir when the next instruction is run.

ARG

- Adds build time args
- These will invalidate the build cache at the instruction they are used.
- Can be useful for specifying certain build time args, i.e. compile-time flags, version of an application to build.
- Don't use for storing secrets – can be viewed with `docker history` and similar.

ENTRYPOINT/CMD

- Need at least one to run – if only one specified, use this as the command to run when starting the container and not overridden.
- If both CMD and ENTRYPOINT are specified, ENTRYPOINT
- Use ENTRYPOINT command form, generally speaking ["some_command", "arg1", "arg2"]. CMD will get passed to this command.
- ENTRYPOINT often does some runtime setup, and then replaces the process specified in CMD using **exec**

Multi stage builds

- Allow you to use COPY from another image's layers
- Useful for keeping image size down
- Example use case: compile source with numerical optimizing container, COPY from original build and only actually push binaries/libs, not source and compilers

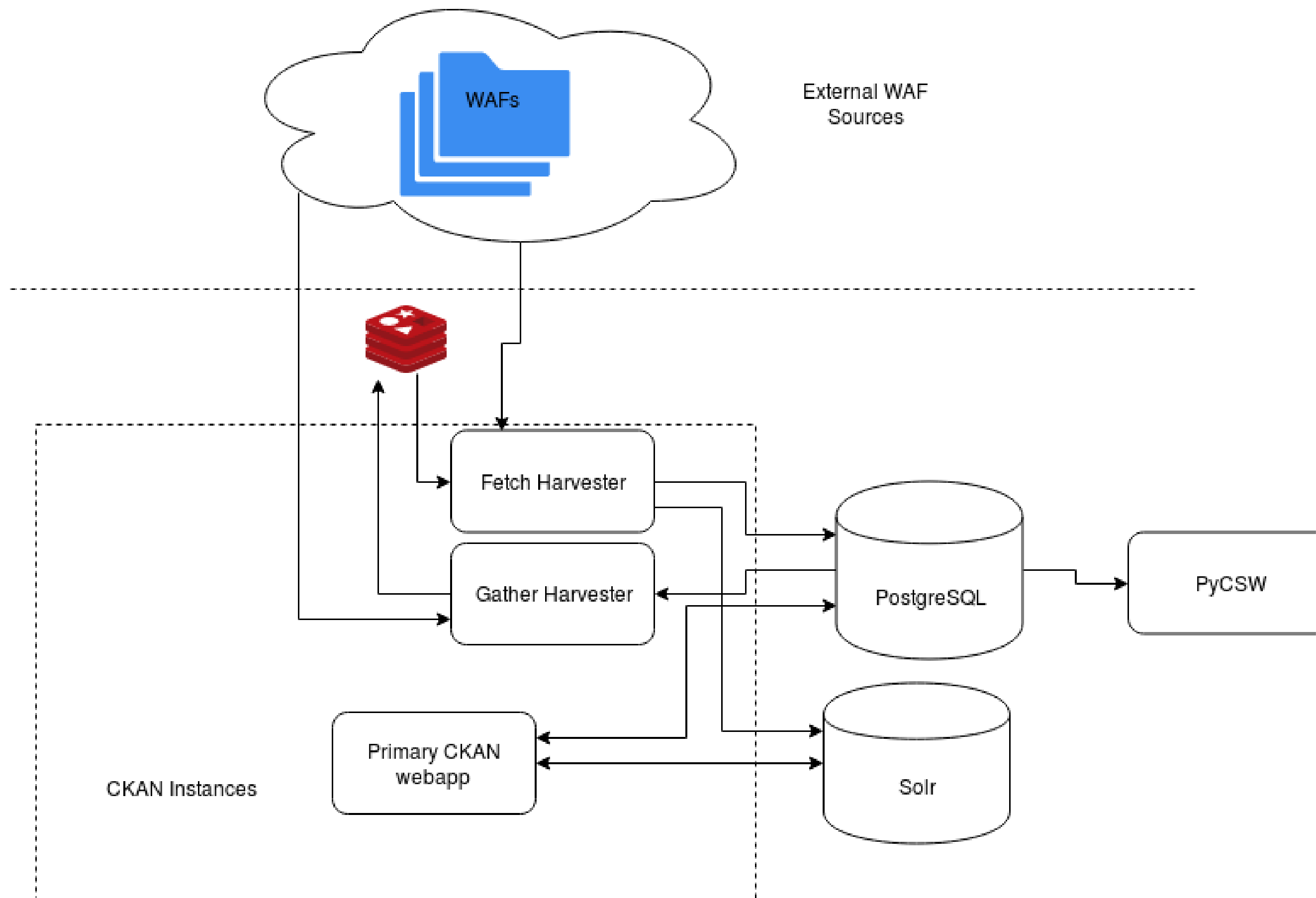
Monitoring containers in production

- Containers produce detailed stats – can see with `docker stats`.
- Can monitor metrics with InfluxDB/Telegraf, or Prometheus
- Visualize stats
- Roll out Telegraf on container hosts using tools such as Ansible
- Use Prometheus use using K8S?

IOOS Catalog

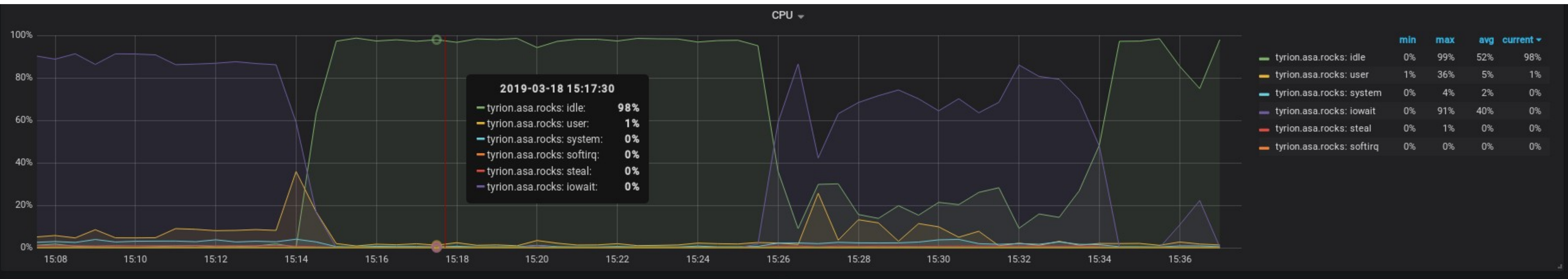
- First used in circa 2016, migrating from service monitor
- Originally used phusion-baseimage for PID 1 issue, kind of bloated
- Moved everything over to CKAN 2.8

IOOS Catalog architecture



Actual use case

- CKAN was using a very large amount of system I/O on IOOS Catalog Host.
- Wasn't immediately clear what was slowing down.



Fixing problems

- Enable restart policies (on-failure, always, unless-stopped)
- If you have an issue with a file, look at the traceback
- You can run `docker cp` even against a stopped or exited containers to apply hotfix then restart.
- `docker commit` can be useful to create hotfixes.
- `docker update` useful for changing settings on the fly (mem limits, reset policy, etc)
- Can also run container from previous build step in Dockerfile

Stability and running with multiple services

- Standalone applications can be useful, but many also need to communicate with other services, such as databases.
- Use docker-compose for simple use cases, consider moving to K8S if more advanced uses are needed
- Combine HEALTHCHECK with entrypoint scripts which spinlock/wait for required resources (e.g. wait-for-it.sh, pg_isready, etc.)

Introducing docker-compose

Docker-compose is an application which reads YAML files which specify container configuration.

- The majority of the arguments are analogous to their docker run counterparts.
- Example IOOS Catalog

Questions?