Cloud Native DevOps: Migrating from Monoliths to Microservices

Release 0.0.2

Matthew Giassa

Aug 23, 2021

CONTENTS:

CHAPTER

ONE

INTRODUCTION

1.1 Overview

This work reflects nearly two decades of experience of mine in the fields of operations, software development, DevOps, and site reliability engineering. It is intended to function as a high-level overview of numerous technologies and design methodologies used in modern software engineering, and to also provide examples and exercises to help the reader better understand the underlying material.

Ultimately, I hope that this material, combined with the examples and code listings provided, will help the audience gain a better understanding of Cloud Native computing, with particular emphasis on how it applies to DevOps and day-to-day development tasks. Ideally, this should allow individuals to more easily apply these concepts to their own projects, further contributing to the body of knowledge (and size of the talent pool) in the field of cloud computing.

The work concludes with hands-on exercises and a project involving creating an on-premise cluster using low-cost computing equipment (i.e. Raspberry Pi hobby kits), with the intent of further helping the reader gain familiarity and confidence in provisioning their own clusters (either bare-metal setups like the Raspberry Pi based project, or with virtual machines or third-party hosted infrastructure).

1.2 Donations and Ethics Disclaimer

This work was carried out entirely by myself, in the form of independent learning and research carried out in personal time. At no time have I received any manner of compensation or donation (either in the form of currency or donated equipment, licenses, etc.).

Additionally, it is worth noting that while I volunteer regularly with the *CNCF*, this work (including the use of the term "Cloud Native" in the title) in no way whatsoever implies that this work is associated with (or officially endorsed by) the *CNCF*. That being said, members of the open source community have been courteous and encouraging in my writing of this book, and I have been permitted to post notifications about the material in said book in the `#mentoring` *CNCF* Slack channel on occasion, provided that the book remains freely accessible for use by the audience, and no deliberate vendoring or advertising is present.

Finally, the author acknowledges that specific implementations of various tools/technologies are mentioned throughout this book, including *FOSS* technologies associated with commercial entities (i.e. that provide tiered support options or non-*FOSS* plugins for their *FOSS*-based offerings). I have elected to use these tools in various examples due to their ease-of-use, and have made a deliberate effort to ensure that all examples use freely available tools, and opt for the use of *FOSS* technologies whenever possible and/or practical. Mention of a specific vendor is only done in very specific cases where it would help the reader learn more about the tool (i.e. learn who the maintainers are), are kept to a minimum, and in no way reflect an endorsement or advertisement by the author (myself).

1.3 Why PronK8S

The original version of this document focused mainly on the example project involving Raspberry Pi units, and was a reference to an animal known as a springbok, known for pronking/stotting to move rapidly. I decided that this could be a much larger and broader document than what I had originally planned.

Fig. 1.1: PronK8S: spring into action with K8S!

1.4 Dedication

To my wife, Lynn, and our wonderful sons. Thank you for putting up with the absurdly loud mechanical keyboards working at all hours of the day and resonating throughout the house.

1.5 License

CHAPTER

TWO

CONTINOUS INTEGRATION & CONTINUOUS DELIVERY

Warning: This section is not yet complete, but I'm aiming to complete all of them by end-of-year 2022. Check-in once in a while for updates.

2.1 Version Control and Branching Strategies

Previously the domain of developers. Now DevOps needs a serious voice in these decisions.

CHAPTER

THREE

CONFIGURATION MANAGEMENT & INFRASTRUCTURE AS CODE

Warning: This section is not yet complete, but I'm aiming to complete all of them by end-of-year 2022. Check-in once in a while for updates.

3.1 Declarative Versus Imperative Models

3.2 GitOps and Versioning Infrastructure Changes

3.3 Rolling Upgrades

3.4 Backup and Restore Operations

3.5 Panic Button

3.6 Tooling

3.6.1 Ansible

3.6.2 Terraform

CHAPTER

FOUR

DEVOPS BUILD PATTERNS

This chapter will cover what the author refers to as "DevOps build patterns". These are effectively common sets of scripts and methods for building software (i.e. artifacts, documentation, etc.), with particular emphasis on how repeatable, reliable, maintainable, and scalable (our four "build pattern viability metrics"), these approaches are. After going over numerous specific cases, we will conclude the chapter with examples for re-creating many of these use cases in a "real world" scenario.

4.1 Bare Metal

A "bare metal" build pattern is the most simple to initially configure: one provisions a physical device (i.e. *PC*, development board, Raspberry Pi hobby kit, etc.) with an operating system, and build tool chain (i.e. compilers, script interpreters, etc.) to produce build artifacts via some manner of build/compilation process (e.g. compiling `.c` files with `gcc`, generating plots with Python scripts, producing documents from `.rst` files with `sphinx`, etc.).

While initially provisioning such a setup is typically straightforward (e.g. "install Linux to a new laptop for an engineer/developer"), the end result leaves much to be desired in terms of our viability metrics. For example, this approach may be used to provision *PCs* or laptops for individual developers in one particular manner, and centralized build infrastructure (i.e. "build servers for production-worthy artifacts") are provisioned in another manner. This often gives rise to the age old statement "but it worked on my machine!". The software load-out on individual *PCs* may differ from that on build servers (e.g. due to *IT* security/company policies), or individual *PCs* may differ among teams (i.e. depending on what machines were purchased and when, what security updates and system packages have been installed and if they are kept current/up-to-date, etc.).

This approach is reasonable for new teams and projects, but due to the potential for variance in builds (i.e. developer *PCs* versus centralized build servers, or even variances from one developer *PC* to another), this approach can burn up a lot of engineering time and effort to maintain long-term. Additionally, without the use of configuration management, it becomes horrible to maintain and completely unscalable. As soon as the opportunity arises, seriously consider migrating to something more modern (such as the examples in the following sections).

Note: Build pattern viability metrics.

Repeatable: yes, but manual and repetetive.

Reliable: yes.

Maintainable: yes, but only via configuration managament or an inordinate amount of time and resources will have to be invested in maintaining such a deployment.

Scalable: no. Requires configuration management to be viable, and variance in hosts/machines leads to additional maintenance requirements.

4.2 Virtual Machines

Virtualization, in a simple sense, is an abstraction of real (i.e. typically physical) resources. For example, virtual machines are abstractions of "real" machines which can run an entire operating system plus system applications, with the system being under the impression it's running on real/physical hardware, when in reality, it is communicating with virtual hardware (that eventually communicates, through layers of abstraction/translation, to the actual underlying physical/real hardware).

This allows, for example, someone running Ubuntu Linux for a 64-bit Intel processor (i.e. a "real *PC*", running what is referred to as the "host *OS*"), to run a virtualized instance of the Windows XP operating system (i.e. a "virtual *PC*", running what is referred to as the "guest *OS*) as if it were just another application. Such setups are extremely useful, as it allows a host *OS* to run applications for a completely different *CPU*/architecture and/or *OS*, without requiring the actual hardware (which may no longer be available due to production ceasing) to run the guest *OS* and the applications it supports. This typically comes with a cost: virtualization is expensive in terms of *CPU* and memory overhead. Provided that the virtualization software itself is maintained, one could run an old legacy application for a long-dead architecture years after hardware is no longer available (though, ideally, one would not allow a critical business element to rely on end-of-life unsupported software long-term).

4.2.1 History and Hypervisors

While virtualization in the field of computer science has been around for a long time (e.g. the evolution of the IBM CP-40 into the CP-67 in the 1960s, allowing for multiple concurrent application execution [1]), we will focus primarily on a cursory analysis of more recent developments, particularly in the context of *VMs* and containers.

With this in mind, we introduce the concept of a hypervisor (also referred to as a virtual machine monitor, or *VMM*): specialized software used to virtualize (i.e. abstract) an *OS*. The primary responsibilities of the hypervisor are to provide (for the guest *OS*) abstractions of hardware (i.e. virtual hardware that eventually maps to real hardware), and to handle or "trap" system calls (*APIs* provided by an operating system for requesting specific, usually privileged, functionality from the kernel; [2]).

Diving further into hypervisors, there are two types of hypervisor (well, three, but two of relevance to this chapter): "type 1" and "type 2" hypervisors. First, with type 2 hypervisors, a simplified summary would be that type 2 hypervisors require various operations to be delegated or otherwise translated by the host *OS* on behalf of the guest *OS*. This results in a "true virtualization" of the guest *OS*, at the expense of increased overhead (and by extension, decreased performance by the guest *OS*). If we consider Fig. 4.1 [3], the guest *OS* will run (typically, not always the case) in ring 3, and operations such as system calls and hardware access are trapped by the host *OS* (whose kernel is the sole software entity with access to ring 0).

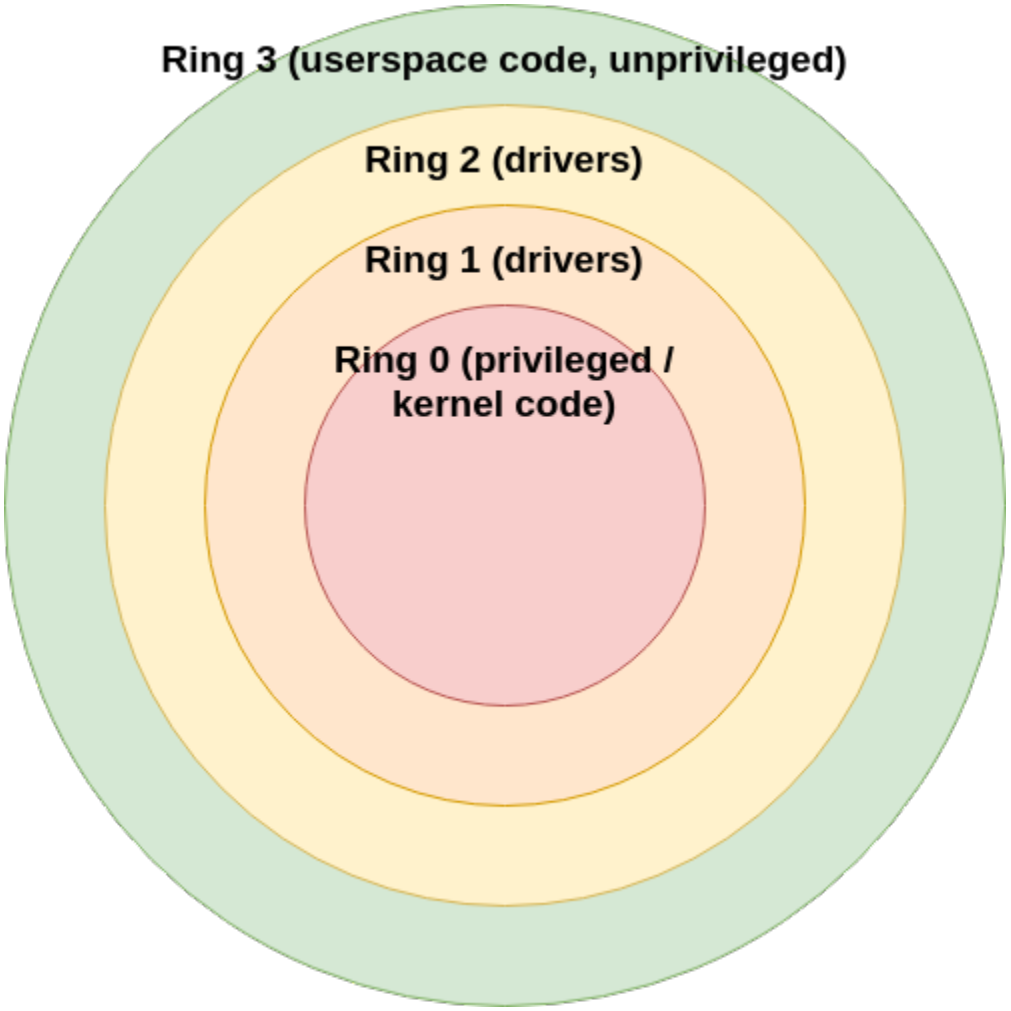

Fig. 4.1: Protection rings on Intel architecture.

In the case of a type 1 hypervisor, additional hardware support in the *CPU* (i.e. Intel VT-x "Vanderpool" or AMD V

"Pacifica", and their modern successors/counterparts) allows for the guest *OS* to have direct access to the underlying physical hardware, permitting for a drastic improvement in performance. If we consider Fig. 4.2, in the context of a type 1 hypervisor, the guest *OS*, still running in ring 3, is able to access the hardware in a much more direct manner through the hypervisor. It is worth noting that, in actuality, there are only really 4 rings (i.e. 0, 1, 2, and 3). The negative rings are really all processor features/extensions that are applied to or otherwise relate to ring 0. In any case, the key takeaway is that a type 1 hypervisor allows for improved performance through reduced overhead.

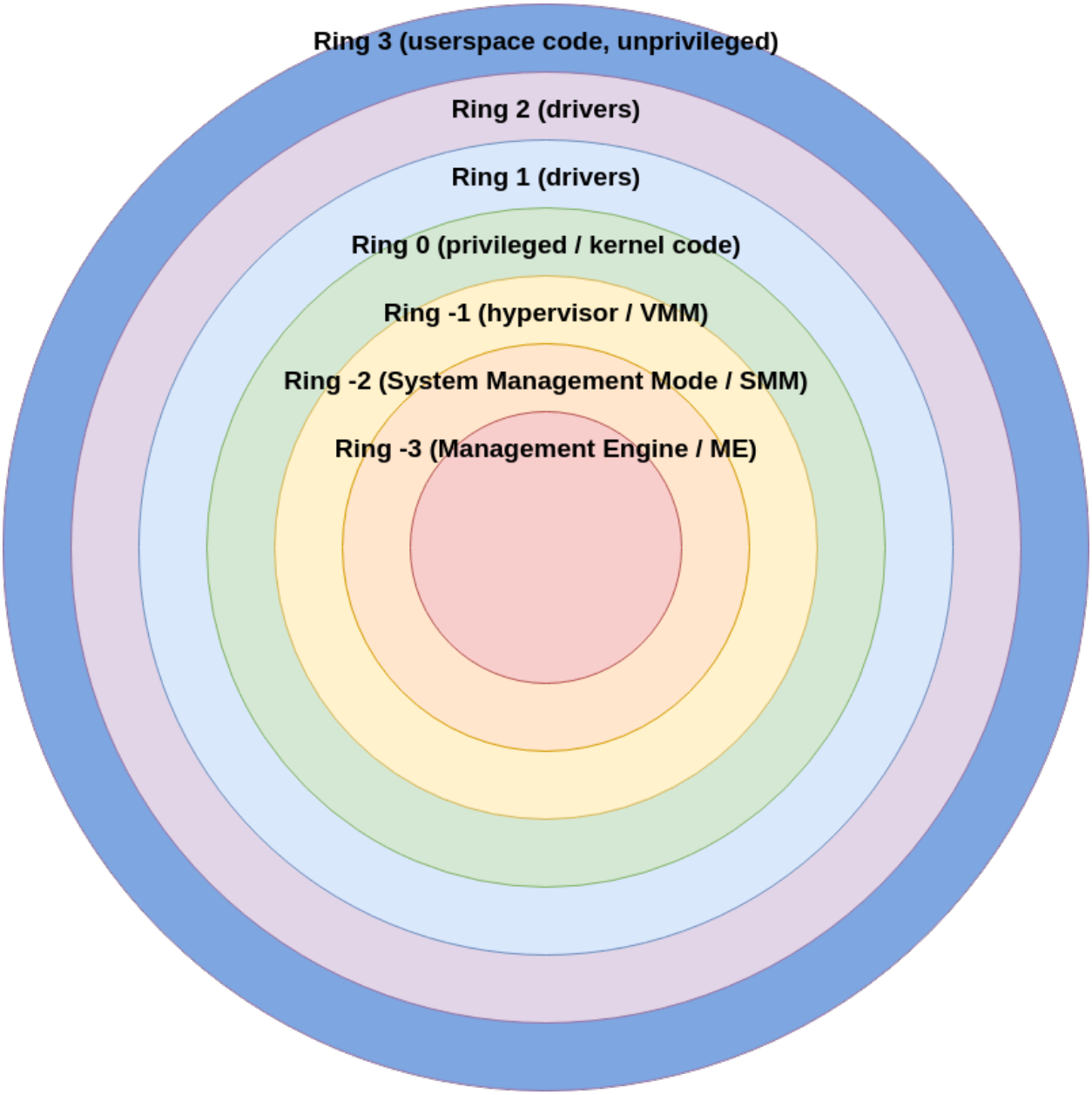

Fig. 4.2: Protection rings on Intel architecture (real and synthetic).

So, we've established that *VMs* allow for a convenient way to run software intended for combinations of *CPU* architectures and *OSs* in a guest *OS*, even if it differs wildly from the host *OS*. *VMs* are also portable as a side effect of this (i.e. pre-built *VM* setups can be easily copied between different physical host machines and re-used, provided the varying machines have the appropriate virtualization software present), allowing for varying degrees of scalability as well. This this being said, we will move on to the topic of containers.

Containers can be thought of as "lightweight virtual machines". Rather than employing the use of a hypervisor, containers are essentially means of running software on the host *OS* in private, isolated environments. A very primitive approach to containers, known as a "`chroot` jail", has been available for nearly 20 years now (at the time of the writing of this document). However, containers employ a greater degree of control and protection mechanisms, using three particularly useful Linux features:

- Namespaces (i.e. for isolation of file systems, hostnames, *IPC*, network resources, etc. [4]).
- Control groups, or "cgroups" (i.e. for logically grouping processes and applying monitor and limits on them, such as quotas on *CPU* and *RAM* usage, for example).
- Union mounts (i.e. a means of taking multiple folders and "stacking" them to create a virtual abstraction of the contents of all the folders in aggregate).

Through the use of these Linux-specific pieces of functionality, isolated execution environments, referred to as "containers", can allow for applications to run securely and independently from each other, relatively oblivious to the fact that they are executing within a container framework. The lack of a hypervisor and the associated virtualization mechanisms means that there is a significant improvement in performance over traditional virtualization solutions [5] [6] [7].

Note: These container technologies can be utilized on non-Linux operating systems such as Apple's OSX, or Microsoft Windows; but they are actually containers running within a hypervisor-based virtualization solution, so a massive amount of additional overhead is incurred on non-Linux systems. This has the unfortunate consequence of negating most of the benefits containers supply, namely improved performance and no need for a hypervisor.

There is one potential downside to this: the containers directly re-use the same kernel as the host operating system (i.e. Linux). If one wishes to use different kernel-specific features and drivers, for example, the host *OSs* kernel must support it, or it won't be available to applications/services running within the containers. It also implies that the software running in the containers must be compiled for the same *CPU* architecture and *OS* as the host *OS*. There is a loss of portability, but the trade-off is a significant boost in performance and an astounding increase in scalability (more on this when we discuss *K8S* and cluster orchestration in later sections).

With the topics of *VMs* and containers being briefly covered, let's move on to applications making use of the aforementioned technologies. Should the reader wish to go into this topic in greater detail, please refer to [1] and [8].

4.2.2 Single VM with Vagrant

Note: This section just scratches the surface on the topic of *VMs*. Tools such as Terraform [9], for example, can be used to provision entire cloud infrastructure deployments in environments such as Amazon *EC2*, Microsoft *Azure*, Google *GCP*, etc.; and can deploy virtul machines at scale efficiently and effectively. While the rest of this chapter focuses heavily on containers and *K8S*, don't disqualify the use of *VMs*, especially in off-site cloud environments like the aforementioned service providers.

At least topically, using a single *VM* as a builder appears to be very similar to bare metal builder:

- Both represent a complete appliance/host, with a dedicated operating system (albeit virtualized).
- Both require tooling such as configuration management to keep them up-to-date and secure.
- Both can be controlled in similar manners (i.e. via a graphical window manager or connecting via command line tools like `ssh`).

Where *VMs* really shine is in their provisioning and portability. Using tools like HashiCorp `vagrant` [10], for example, one may write scripts in a structured and standardized manner to produce a virtual machine on-demand. Rather than manually creating a virtual machine and provisioning it (i.e. configuring an `ISO` image containing the guest *OSs*

installation files as a virtual optical drive, "connecting" it to the *VM*, installing the guest *OS* and relevant applications, configuring said applications, etc.), one can download pre-created (and verified/trusted) images for common platforms such as various Linux distributions (i.e. Ubuntu, Arch, Fedora, etc.), and add customizations on afterwards (i.e. additional/custom packages, scripts, pre-compiled binaries, etc.). This removes one of the largest (and most tedious) steps involved in provisioning a bare metal builder, and the final artifact (i.e. the *VM* image itself) can be trivially copied from one physical host to another for duplication and re-use (with the appropriate re-provisioning steps in place, such as randomizing the *MAC* address of the network adapter and resetting credentials, etc.).

In addition to being able to rapidly provision *VMs* rapidly, they also lend themself to another especially helpful use case: ephemeral/throw-away *VMs*. With a bare metal builder, chances are the intent is to provision it, maintain it regularly, and eventually dispose of it when the need arises. For such a setup, it is not desirable to have to re-provision it more than necessary (i.e. if a persistent storage medium fails, for example). However, there are cases where someone may wish to have a "fresh" deployment every time a specific job is executed. For example, someone may have a project that creates the installer (i.e. akin to a `deb` package for an Ubuntu/Debian system, or a `setup.exe` for a Windows-based system), and has configured a build pipeline to automatically perform builds of this tool when one of its components change (i.e. in a `git` repository, due to a commit being pushed).

This is a sound strategy: automatically trigger unit and/or regression tests via the *CI/CD* infrastructure every time a change is introduced into the code base. If this automated testing is non-destructive (i.e. has minimal or no ability to adversely impact the host/machine used for testing), this is not a problem. However, if this testing is destructive (i.e. could corrupt the software loadout on a host/machine to the point of it outright requiring re-provisioning, including a re-installation of the *OS*), then it's going to incur significant overhead by technical staff who now have to periodically repair/re-image the build host/machine. If we were to use a *VM* for this task, we could dramatically cut down on the overhead involved: just pre-create a "golden" (i.e. known to be in a good, working, valid state) *VM*, and make a copy of it every time a build job needs to be triggered (and run said job inside the copy of the "golden" *VM* image).

When the job has concluded, just delete/discard the modified image that was just used (after extracting build artifacts, logs, etc.; from it), and we're done. This will ensure every build job will have the exact same initial conditions, cut down on the need for technical staff to re-provision physical hosts/machines, and due to the inherent portability of *VMs*, the "golden" image can be duplicated across a wide variety of machines (i.e. even with differing hardware): so long as they all support the same virtualization framework, they can all make use of the same "golden" image. With this said, let's move on to an example where we'll create an ephemeral/throw-away *VM* on-demand, use it to build a small C project, backup the build artifacts and logs, and then dispose of the *VM* image.

First, we'll just create a new `Vagrantfile` via the command `vagrant init`. Next, we'll customize the minimal/baseline `Vagrantfile` to be a bit more interesting (manually specify some options impacting performance, configure it to use Ubuntu 20.04 "focal" as the baseline *OS*, and configure it to use a provisioning script to install packages to the *VM* before we begin using it).

Note: In addition to installing Vagrant (i.e. `sudo apt install -y vagrant` on Ubuntu/Debian systems), the reader is also encouraged to install the `vbguest` plugin to avoid various errors that will require the use of researching via Google and StackOverflow to resolve (i.e. `vagrant plugin install vagrant-vbguest`).

Listing 4.1: Original Vagrantfile generated via `vagrant init` before we added our modifications to it.

```
# -*- mode: ruby -*-
# vi: set ft=ruby :

# All Vagrant configuration is done below. The "2" in Vagrant.configure
# configures the configuration version (we support older styles for
# backwards compatibility). Please don't change it unless you know what
# you're doing.
Vagrant.configure("2") do |config|
```

(continues on next page)

(continued from previous page)

```
  # The most common configuration options are documented and commented below.
  # For a complete reference, please see the online documentation at
  # https://docs.vagrantup.com.

  # Every Vagrant development environment requires a box. You can search for
  # boxes at https://vagrantcloud.com/search.
  config.vm.box = "base"

  # Disable automatic box update checking. If you disable this, then
  # boxes will only be checked for updates when the user runs
  # `vagrant box outdated`. This is not recommended.
  # config.vm.box_check_update = false

  # Create a forwarded port mapping which allows access to a specific port
  # within the machine from a port on the host machine. In the example below,
  # accessing "localhost:8080" will access port 80 on the guest machine.
  # NOTE: This will enable public access to the opened port
  # config.vm.network "forwarded_port", guest: 80, host: 8080

  # Create a forwarded port mapping which allows access to a specific port
  # within the machine from a port on the host machine and only allow access
  # via 127.0.0.1 to disable public access
  # config.vm.network "forwarded_port", guest: 80, host: 8080, host_ip: "127.0.0.1"

  # Create a private network, which allows host-only access to the machine
  # using a specific IP.
  # config.vm.network "private_network", ip: "192.168.33.10"

  # Create a public network, which generally matched to bridged network.
  # Bridged networks make the machine appear as another physical device on
  # your network.
  # config.vm.network "public_network"

  # Share an additional folder to the guest VM. The first argument is
  # the path on the host to the actual folder. The second argument is
  # the path on the guest to mount the folder. And the optional third
  # argument is a set of non-required options.
  # config.vm.synced_folder "../data", "/vagrant_data"

  # Provider-specific configuration so you can fine-tune various
  # backing providers for Vagrant. These expose provider-specific options.
  # Example for VirtualBox:
  #
  # config.vm.provider "virtualbox" do |vb|
  #   # Display the VirtualBox GUI when booting the machine
  #   vb.gui = true
  #
  #   # Customize the amount of memory on the VM:
  #   vb.memory = "1024"
  # end
  #
  # View the documentation for the provider you are using for more
```

(continues on next page)

(continued from previous page)

```
  # information on available options.

  # Enable provisioning with a shell script. Additional provisioners such as
  # Puppet, Chef, Ansible, Salt, and Docker are also available. Please see the
  # documentation for more information about their specific syntax and use.
  # config.vm.provision "shell", inline: <<-SHELL
  #   apt-get update
  #   apt-get install -y apache2
  # SHELL
end
```

Listing 4.2: Vagrantfile for generating our VM builder.

```
# -*- mode: ruby -*-
# vi: set ft=ruby :

# All Vagrant configuration is done below. The "2" in Vagrant.configure
# configures the configuration version (we support older styles for
# backwards compatibility). Please don't change it unless you know what
# you're doing.
Vagrant.configure("2") do |config|
  # The most common configuration options are documented and commented below.
  # For a complete reference, please see the online documentation at
  # https://docs.vagrantup.com.

  # Every Vagrant development environment requires a box. You can search for
  # boxes at https://vagrantcloud.com/search.
  config.vm.box = "ubuntu/focal64"

  # Disable automatic box update checking. If you disable this, then
  # boxes will only be checked for updates when the user runs
  # `vagrant box outdated`. This is not recommended.
  # config.vm.box_check_update = false

  # Create a forwarded port mapping which allows access to a specific port
  # within the machine from a port on the host machine. In the example below,
  # accessing "localhost:8080" will access port 80 on the guest machine.
  # NOTE: This will enable public access to the opened port
  # config.vm.network "forwarded_port", guest: 80, host: 8080

  # Create a forwarded port mapping which allows access to a specific port
  # within the machine from a port on the host machine and only allow access
  # via 127.0.0.1 to disable public access
  # config.vm.network "forwarded_port", guest: 80, host: 8080, host_ip: "127.0.0.1"

  # Create a private network, which allows host-only access to the machine
  # using a specific IP.
  # config.vm.network "private_network", ip: "192.168.33.10"

  # Create a public network, which generally matched to bridged network.
  # Bridged networks make the machine appear as another physical device on
  # your network.
```

(continues on next page)

(continued from previous page)

```
  # config.vm.network "public_network"

  # Share an additional folder to the guest VM. The first argument is
  # the path on the host to the actual folder. The second argument is
  # the path on the guest to mount the folder. And the optional third
  # argument is a set of non-required options.
  # config.vm.synced_folder "../data", "/vagrant_data"

  # Provider-specific configuration so you can fine-tune various
  # backing providers for Vagrant. These expose provider-specific options.
  # Example for VirtualBox:
  #
  config.vm.provider "virtualbox" do |vb|
    # Display the VirtualBox GUI when booting the machine
    # Nah, let's do everything via console/shell/command-line.
    vb.gui = false

    # Customize the amount of memory on the VM:
    vb.memory = "2048"

    # Add more cores.
    vb.cpus = 2
  end

  # Run our deployment script during `vagrant up --provision` (or first
  # `vagrant up`) operation.
  config.vm.provision "shell", path: "deploy.sh"

  # View the documentation for the provider you are using for more
  # information on available options.

  # Enable provisioning with a shell script. Additional provisioners such as
  # Puppet, Chef, Ansible, Salt, and Docker are also available. Please see the
  # documentation for more information about their specific syntax and use.
  # config.vm.provision "shell", inline: <<-SHELL
  #   apt-get update
  #   apt-get install -y apache2
  # SHELL
end
```

Listing 4.3: Difference between original and modified Vagrantfiles.

```
--- /work/examples/build_patterns/vm_example/Vagrantfile.original
+++ /work/examples/build_patterns/vm_example/Vagrantfile
@@ -12,7 +12,7 @@

   # Every Vagrant development environment requires a box. You can search for
   # boxes at https://vagrantcloud.com/search.
-  config.vm.box = "base"
+  config.vm.box = "ubuntu/focal64"

   # Disable automatic box update checking. If you disable this, then
```

(continues on next page)

(continued from previous page)

```
   # boxes will only be checked for updates when the user runs
@@ -49,14 +49,22 @@
   # backing providers for Vagrant. These expose provider-specific options.
   # Example for VirtualBox:
   #
-  # config.vm.provider "virtualbox" do |vb|
-  #   # Display the VirtualBox GUI when booting the machine
-  #   vb.gui = true
-  #
-  #   # Customize the amount of memory on the VM:
-  #   vb.memory = "1024"
-  # end
-  #
+  config.vm.provider "virtualbox" do |vb|
+    # Display the VirtualBox GUI when booting the machine
+    # Nah, let's do everything via console/shell/command-line.
+    vb.gui = false
+
+    # Customize the amount of memory on the VM:
+    vb.memory = "2048"
+
+    # Add more cores.
+    vb.cpus = 2
+  end
+
+  # Run our deployment script during `vagrant up --provision` (or first
+  # `vagrant up`) operation.
+  config.vm.provision "shell", path: "deploy.sh"
+
   # View the documentation for the provider you are using for more
   # information on available options.

```

Listing 4.4: Provisioning script to supplement Vagrantfile (Ansible playbooks are preferable in general).

```
#!/bin/bash

# Install some packages.
sudo apt update -y
sudo apt install -y \
    automake \
    binutils \
    cmake \
    coreutils \
    cowsay \
    gcc \
    iftop \
    iproute2 \
    iputils-ping \
    lolcat \
    make \
```

(continues on next page)

(continued from previous page)

```
    net-tools \
    nmap \
    python3 \
    python3-dev \
    python3-pip \
    toilet

# Some helpful python tools.
sudo pip3 install \
    flake8 \
    pylint

sudo apt clean -y
```

Now, we'll launch our *VM* via `vagrant up`, and see what happens (lots of console output is generated, so we'll have to trim it to keep just the relevant bits).

Listing 4.5: Launching a Vagrant VM (virtulbox provider/hypervisor).

```
# Start booting and configuring the VM.
owner@darkstar$> vagrant up
Bringing machine 'default' up with 'virtualbox' provider...
==> default: Importing base box 'ubuntu/focal64'...
==> default: Matching MAC address for NAT networking...
==> default: Checking if box 'ubuntu/focal64' version '20210803.0.0' is up to date...
==> default: Setting the name of the VM: vm_example_default_1628615617134_78200
==> default: Clearing any previously set network interfaces...
==> default: Preparing network interfaces based on configuration...
    default: Adapter 1: nat
==> default: Forwarding ports...
    default: 22 (guest) => 2222 (host) (adapter 1)
...
...
...

# Now it's successfully running our provisioning script.
The following additional packages will be installed:
  binutils binutils-common binutils-x86-64-linux-gnu build-essential cpp cpp-9
  dctrl-tools dpkg-dev fakeroot g++ g++-9 gcc gcc-9 gcc-9-base
  libalgorithm-diff-perl libalgorithm-diff-xs-perl libalgorithm-merge-perl
  libasan5 libatomic1 libbinutils libc-dev-bin libc6-dev libcc1-0 libcrypt-dev
  libctf-nobfd0 libctf0 libdpkg-perl libfakeroot libfile-fcntllock-perl
  libgcc-9-dev libgomp1 libisl22 libitm1 liblsan0 libmpc3 libquadmath0
  libasan5 libatomic1 libbinutils libc-dev-bin libc6-dev libcc1-0 libcrypt-dev
  libctf-nobfd0 libctf0 libdpkg-perl libfakeroot libfile-fcntllock-perl
  libgcc-9-dev libgomp1 libisl22 libitm1 liblsan0 libmpc3 libquadmath0
  libstdc++-9-dev libtsan0 libubsan1 linux-libc-dev make manpages-dev
0 upgraded, 43 newly installed, 0 to remove and 0 not upgraded.
Need to get 43.1 MB of archives.
After this operation, 189 MB of additional disk space will be used.
...
...
```

(continues on next page)

(continued from previous page)

```
...

# And we're eventually returned to our shell. Let's log in to the VM via SSH:
[10:14:57]: owner@darkstar$> vagrant ssh
Welcome to Ubuntu 20.04.2 LTS (GNU/Linux 5.4.0-80-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

  System information as of Tue Aug 10 17:22:57 UTC 2021

  System load:  0.0               Processes:               119
  Usage of /:   4.3% of 38.71GB   Users logged in:         0
  Memory usage: 12%               IPv4 address for enp0s3: 10.0.2.15
  Swap usage:   0%


1 update can be applied immediately.
To see these additional updates run: apt list --upgradable

# We're in: it works. Time to call it a day.
vagrant@ubuntu-focal:~$ logout
Connection to 127.0.0.1 closed.
```

Vagrant is a large topic that can encompass several books, so the reader is left to conduct their own research and learning/training exercises to become more versed in its use (if desired). The goal of this section has been accomplished: demonstrating how easy it is to rapidly prepare and deploy a *VM* via a tool like Vagrant (and using an *IAC* approach no less).

Note: Build pattern viability metrics.

Repeatable: very repeatable. Use of infrastructure-as-code techniques via Vagrantfiles allows us to achieve a high level of repeatability (similar to Docker containers; described later).

Reliable: generally reliable. Attempting to access specific hardware (e.g. *USB*-passthrough) can, in the author's experience, lead to stability problems (although such use cases are never typically encountered when using *VMs* as dedicated builders, so it's a moot point.

Maintainable: in the case of long-lived *VMs*, maintainable as long as configuration management is used (similar to bare metal hosts). In the case of ephemeral/throwaway *VMs*, very maintainable (similar to Docker containers, as we just invoke them on-demand, and discard them when no longer needed).

Scalable: generally scalable vertically (due to hardware acceleration being available for most hypervisors) and horizontally (i.e. via tools like Terraform). More resource overhead than containers, but still manageable.

4.3 Containers

One of the most (in recent memory) ubiquitous and useful patterns (in my own opinion) is container-based build patterns. As the following examples will show, they not only scale exceptionally well, but they are very easy to extend/manipulate to provide an assortment of features (without having to rely on *VM* vendor-specific functionality).

4.3.1 Single Container with Docker

Launching a single container interactively is very straightforward. We just need to make sure that Docker is installed [11]. In my own case, on an Ubuntu 20.04 installation, I simply needed to do the following:

- Execute `sudo apt install docker`.
- Add myself to the `docker group` so I could execute Docker commands without the use of the `sudo` command, i.e: `sudo usermod -a -G docker $(whoami)`, and then log out of all running sessions (or rebooting the machine might be easier).

In any case, please refer to the official Docker documentation [11] for guidance on installing Docker (on Linux; as noted earlier, this material focuses exclusively on Linux hosts). Now, let's take the latest Ubuntu 20.04 "Focal" distro for a spin, by instantiating an interactive instance of it (for more details on the command line arguments, please see [12]).

Listing 4.6: Example of launching a container.

```
# Launch an interactive instance, and auto-cleanup when done.
$> docker run -it --rm ubuntu:focal

Unable to find image 'ubuntu:focal' locally
focal: Pulling from library/ubuntu
16ec32c2132b: Already exists
Digest:
sha256:82becede498899ec668628e7cb0ad87b6e1c371cb8a1e597d83a47fac21d6af3
Status: Downloaded newer image for ubuntu:focal

# Now we're in our container, in it's own dedicated virtual/throw-away
# file system. Let's look around.
root@016a95a884a3:/# ls -a
.  ..  .dockerenv  bin  boot  dev  etc  home  lib  lib32  lib64  libx32
media  mnt  opt  proc  root  run  sbin  srv  sys  tmp  usr  var
```

Keep in mind, when we're using `docker run` (or `docker exec`) to instantiate and run a shell in the container, we're operating within a throwaway file system (i.e. when the container instance is terminated, that file system and the files present in it, even those we manually create, are gone). Let's start to fine tune our arguments to Docker to get more use out of it. Let's assume I'm currently logged in to my host *OS* as user `owner`, on a machine with host name `darkstar`, and my home directory is `/home/owner`. Let's create a "staging area" for our Docker-related experiments in `/home/owner/work` (the following examples will use the ~ literal and `$HOME` variable to avoid references to `owner` and/or `darkstar` being hard-coded into them).

Listing 4.7: Example of launching a container with a host volume mount.

```
# Create our staging path.
owner@darkstar$> mkdir -p ~/work

# Change directory. Manually make sure no files are present here via "ls", so
```

(continues on next page)

(continued from previous page)

```
# we're not destroying data in case, by coincidence, you already have a
# "~/work" directory on your host machine.
owner@darkstar$> cd ~/work

# Launch our Docker container, but mount the current directory so that it's
# accessible from within the container. We can also use the "pwd" command
# instead of "readlink -f", but I prefer the latter for cases where
# additional mounts are needed, so a single command is used throughout the
# (lengthy) set of command-line arguments.
owner@darkstar$> $> docker run -it --rm \
    --volume="$(readlink -f .):/work"
    --workdir="/work"
    ubuntu:focal

root@0c33ac445db4:/work# ls

root@0c33ac445db4:/work# touch foo

root@0c33ac445db4:/work# ls
foo

root@0c33ac445db4:/work# exit

owner@darkstar$> ls
foo
```

How exciting: the file we created via the `touch` command within our container survived the termination of the container, and is accessible to the current user session on the host *OS*. Let's take another step forward: let's actually build something within the container. We'll create the following two files within the current directory (i.e. *CWD* or *PWD*): `Makefile` and `helloworld.c` (the programming language doesn't really matter, and the example we're demonstrating is just a C-specific minimal "hello world" example, so there's no need to be versed in the C programming language to proceed).

Listing 4.8: Makefile for building a simple ANSI-C "hello world" example.

```
.DEFAULT_GOAL: all
.PHONY: all
all:
        gcc helloworld.c -o helloworld_app
```

Listing 4.9: C code for building a simple ANSI-C "hello world" example.

```
#include <stdio.h>
int main(void) {
    printf("Hello world!\n");
    return 0;
}
```

Now, let's attempt to manually compile our source file via GNU `make` within our container.

Listing 4.10: Example of launching a container with a host volume mount.

```
# Launch our Docker container, but mount the current directory so that it's
# accessible from within the container. We can also use the "pwd" command
# instead of "readlink -f", but I prefer the latter for cases where
# additional mounts are needed, so a single command is used throughout the
# (lengthy) set of command-line arguments.
owner@darkstar$> $> docker run -it --rm \
    --volume="$(readlink -f .):/work"
    --workdir="/work"
    ubuntu:focal

# Confirm our files are present (after creating them on the host OS in
# "~/work"): looks good.
root@f2fb4aeecfbc:/work# ls
Makefile  foo  helloworld.c

# Build our app.
root@f2fb4aeecfbc:/work# make
bash: make: command not found

# That's not good: let's try building directly via "gcc":
root@f2fb4aeecfbc:/work# gcc helloworld.c -o helloworld_app
bash: gcc: command not found

# Still no good. Well, let's try installing these apps.
root@f2fb4aeecfbc:/work# apt install make gcc
Reading package lists... Done
Building dependency tree
Reading state information... Done
E: Unable to locate package make
E: Unable to locate package gcc

# Oh yeah: need to update our apt cache, as it will be empty by default in a
# "fresh" container.

root@f2fb4aeecfbc:/work# apt update -q && apt install make gcc
Hit:1 http://security.ubuntu.com/ubuntu focal-security InRelease
Hit:2 http://archive.ubuntu.com/ubuntu focal InRelease
Hit:3 http://archive.ubuntu.com/ubuntu focal-updates InRelease
Hit:4 http://archive.ubuntu.com/ubuntu focal-backports InRelease
...
...
...
Need to get 33.3 MB of archives.
After this operation, 139 MB of additional disk space will be used.
Do you want to continue? [Y/n]

# Why not: let's install the packages (Y).

# Now, we should be able to build and run our app in this container.
```

(continues on next page)

(continued from previous page)

```
root@f2fb4aeecfbc:/work# make
gcc helloworld.c -o helloworld_app

root@f2fb4aeecfbc:/work# ls -la
total 36
drwxrwxr-x 2 1000 1000  4096 Aug  1 17:43 .
drwxr-xr-x 1 root root  4096 Aug  1 17:37 ..
-rw-rw-r-- 1 1000 1000    72 Aug  1 17:37 Makefile
-rw-r--r-- 1 root root     0 Aug  1 17:23 foo
-rw-rw-r-- 1 1000 1000    77 Aug  1 17:37 helloworld.c
-rwxr-xr-x 1 root root 16704 Aug  1 17:43 helloworld_app

root@f2fb4aeecfbc:/work# ./helloworld_app
Hello world!


root@f2fb4aeecfbc:/work# exit
```

Well, that was interesting: it turns out that these "baseline" Docker images for various Linux distributions (also referred to as "distros") are quite minimal in terms of packages present. We were able to manually install the needed packages however, and eventually build and run our example. Now, go ahead and re-run the example we just finished: notice anything odd/unexpected?

Note: Please go ahead and re-run the example. What is amiss?

As you've likely noticed, you need to re-install `make` and `gcc` again. While the files in `/work` within the container survive container termination (due to the volume mount we have in place), the rest of the container (including packages we've installed to places like `/usr` within the container) do not (this is by design, as containers are intended to generally be throwaway/ephemeral; anything intended for long-term storage needs to be backed up or otherwise exported via mounts or some other means of exporting the data from the running container). Well, this is going to consume a large amount of bandwidth and slow down our build process if we want to repeatedly re-build our example (and not keep the same container instance "live" indefinitely). Fortunately, we can easily extend our baseline Ubuntu container to have some modifications that will be helpful to us. Let's create a new file named `Dockerfile` in the *PWD*, and populate it like so:

Listing 4.11: Dockerfile that extends Ubuntu.

```
FROM ubuntu:focal as baseline

# System packages.
RUN apt update -y && \
    apt install -y \
        make \
        gcc \
    && \
    apt clean -y
```

The key things to keep in mind are that we're using "ubuntu:focal" as our baseline image ("baseline" is just an arbitrary name I chose, it's not a Dockerfile primitive/keyword), and we're using `apt` to install the extra packages we need. The remaining code (i.e. multi-line `apt` usage, `apt clean`, etc.) are just "common/best practises" to reduce the size of the overall image (i.e. optimizations), and are covered in a later section. It's also worth noting that I will typically refer to "Docker containers" as live/running instances of "Docker images", while "Docker images" are the static build artifacts

produced by `docker build` operations (i.e. "Docker containers are live instantiations of Docker images"), to avoid ambiguity.

Now, let's build the Dockerfile to produce an image we can use (pay attention to the period `.` literal on the final line: it's not a typo; you can also replace it with `./` if desired for better readability):

Listing 4.12: Example build command to create a Docker image from a Dockerfile.

```
# Build our image.
DOCKER_BUILDKIT=1 docker build \
  -t "my_docker_builder:local" \
  --target baseline \
  -f Dockerfile \
  .

# Confirm it exists.
owner@darkstar$> docker image ls
REPOSITORY                      TAG       IMAGE ID       CREATED
SIZE
my_docker_builder               latest    5dd3b898dfc1   31 seconds ago
233MB
```

Note: The reader may notice the use of the `DOCKER_BUILDKIT` environment variable [13], along with manually specifying the path to the Dockerfile via the `-f` command-line argument. This is to allow for custom `Dockerfile` and `.dockerignore` [14], file names [15], greatly increasing build speeds (i.e. reducing build times/duration), providing increased security against accidentally bundling files unintentionally, [16], etc. The reader is encouraged to futher investigate these techniques if not already familiar with them, as a minor change to project structure can greatly improve the security and velocity of builds.

Now, let's repeat the earlier `make` example, but use our newly-minted container rather than the "vanilla" (i.e. unmodified) `ubuntu:focal` image.

Listing 4.13: Example of launching a container with a host volume mount via a custom Docker image.

```
# Launch our Docker container, but mount the current directory so that it's
# accessible from within the container. We can also use the "pwd" command
# instead of "readlink -f", but I prefer the latter for cases where
# additional mounts are needed, so a single command is used throughout the
# (lengthy) set of command-line arguments.
owner@darkstar$> $> docker run -it --rm \
    --volume="$(readlink -f .):/work"
    --workdir="/work"
    my_docker_builder:local

# Confirm our files are present (after creating them on the host OS in
# "~/work"): looks good. Let's remove the binary we compiled in our previous
# run to make sure we're really building it from source correctly.
root@6b60e799f6dc:/work# ls
Dockerfile  Makefile  foo  helloworld.c  helloworld_app

root@6b60e799f6dc:/work# rm helloworld_app
```

(continues on next page)

(continued from previous page)

```
root@6b60e799f6dc:/work# make
gcc helloworld.c -o helloworld_app

root@6b60e799f6dc:/work# ./helloworld_app
Hello world!

root@6b60e799f6dc:/work# exit
```

Hurrah! We now have a Docker image with the necessary tools present to build our application, without having to re-download them every time (saving bandwidth, decreasing the amount of time our build takes, and allowing this build process to work in environments without internet access, which would be necessary for downloading packages via `apt`). This Docker image can be used for local builds, as well as builds via *CI/CD* pipelines with tools like GitHub, Jenkins, GitLab, etc. At this point, the reader is strongly encouraged to review the use of `docker push` ([17], [18]) to learn how to back up your Docker images for long-term re-use (and for sharing them with colleagues and team members).

As a final example for this section, we leave the reader with a sample script they are encouraged to use with the source listings we've used, as a matter of convenience, e.g. `iax.sh` (the file name is arbitrary; please pick an alternative at your own discretion; also be sure to make it executable via `chmod +x iax.sh`).

Listing 4.14: Docker launcher script.

```
#!/bin/bash
################################################################################
# @brief:       Docker-bootstrap script for building projects via a Docker
#               image.
################################################################################

# Docker runtime image to use for building artifacts.
DOCKER_IMG="my_docker_builder:local"

# Launch docker container. Setup PWD on host to mount to "/work" in the
# guest/container.
docker run \
  --rm -it \
  --volume="$(readlink -f .):/work:rw" \
  --workdir="/work" \
  ${DOCKER_IMG} \
  ${@}
```

What use is this compared to the commands we've already been using so far? Well, there are four important consequences of using a launcher script like this:

- Without any arguments, it just launches an interactive instance of your container, like how we've been doing throughout this section (i.e. less typing).
- If arguments are passed to the container, it will run them as a non-interactive job, and terminate the container instance when done. For example, try executing something like `./iax.sh make`, and the script will launch the `make` command within the container, and then terminate the container, while leaving the build artifacts behind on your host *OS* (very handy if you want to script/batch builds and other operations in an automated, non-interactive manner using your builder containers).
- You can add a lot of other complex options (some of which will be covered in later sections) to get more functionality out of the script, without requiring users of the script (i.e. other team members) to have to memorize a

copious amount of Docker command-line arguments.

- The script can be modified to reflect the behavior of your *CI/CD* build system, to minimize differences between local/developer builds of a project, and builds launched on dedicated infrastructure as part of your *CI/CD* pipeline (i.e. no more cases of build-related bugs occurring and the response being "but it worked on my machine!"). Less headaches for developers thanks to pro-active, user-friendly infrastructure design by DevOps.

Note: Build pattern viability metrics.

Repeatable: very repeatable.

Reliable: very reliable, provided the host OS itself is stable.

Maintainable: yes, especially when most uses of containers are throwaway/ephemeral, as we just create new instances when needed, rather than maintaining old instances long-term.

Scalable: extremely scalable, especially when tools like K8S are added to the mix. More on such topics in later sections.

4.3.2 Multiple Containers with Docker

This use case is nearly identical to the single container user case (i.e. *Single Container with Docker*). *CI/CD* frameworks like GitHub, GitLab, etc.; support building applications in containers, and also support building different portions of a project/pipeline in different containers (e.g. consider a project with a C and a Go/Golang sub-project in it, and for the sake of convenience, the two sub-projects are built with different dedicated Docker images).

The one place where this can become a bit tedious is when handling the case of local developer builds. Using a launcher script like the `iax.sh` script is great if everything needs to be built under the same container (e.g. one can just execute `./iax.sh make` and have an entire project build end-to-end without any further user interaction). However, this pattern no longer works if different images need to be used throughout the build pipeline, as we'd need to execute different stages of the build with different launcher scripts (i.e. boilerplate copies of `iax.sh`: one per builder image needed).

One off-the-cuff solution to this would be to have a top-level shell script that launches (in the case of local/developer builds) various stages of the pipeline in different containers (example shown below).

Listing 4.15: Example of a multi-container build script.

```
# Launch our "C builder" to build a sub-directory in the PWD.
./iax_c.sh cd c_sub_project && make

# Do the same, for a Go sub-project with our "Go builder".
./iax_golang.sh cd go_sub_project && make
```

While this works, it has several issues with respect to maintainability (even if it seems fine according to our build pattern viability metrics). Namely:

- The top-level build script isn't launching within a container, so it immediately becomes less portable due to certain requirements (besides the existence of Docker) being present on the host *OS*. This may seem trivial, but can be very frustrating if suddenly the host *OS* requires a specific version of a specific tool like `cmake`, `scons`, etc.; present, and is further compounded if developers are using different distros/versions and these tools aren't available for the required combination of distro, tool version, etc. In short, we can avoid a lot of headache and frustration by ensuring the entirety of the build process is somehow "wrapped" via containers or some other virtualization technology.
- We need to have near-duplicate copies of the launcher script for each build image we support. They can easily drift (i.e. they become dissimilar from each other) over time, or as boilerplate copies are duplicated into other projects.

- The top-level build tool (generally, in the case of local/developer builds) has to be a specific tool (i.e. `bash` or some other interpreter). It prevents the (trivial) use of tools like `make` as the top-level build tool, which can prevent better (more optimal) means of building software.
- Executing incremental builds puts additional cognitive load on developers/engineers as they conduct their day-to-day tasks (i.e. "one more corner-case to remember"), as they have to possess a more intricate knowledge of the build chain for local/developer builds (i.e. "a shell script launches some more shell scripts that launch Makefiles in different containers for different sub-projects in the top-level project…"). Simplicity and ease-of-use are paramount for adoption.

With all this said, multiple container (i.e. serialized) builds are a perfectly viable pattern, but some serious thought needs to be put into how it will be presented to the consumer of these systems and scripts (i.e. development teams) so that it is helpful rather than a hindrance to day-to-day development tasks. Since the top-level build script is not containerized, it should ideally be something portable that works across multiple Linux distros and versions (e.g. `bash` or a specific version of `python3`, for example).

Note: Build pattern viability metrics.

Repeatable: very repeatable, but the difference between local/developer builds and CI/CD pipeline builds may result in additional work/load for individual contributors.

Reliable: very reliable, provided the host OS itself is stable.

Maintainable: yes, similar to single container use case.

Scalable: extremely scalable, especially when tools like K8S and docker-compose are added to the mix. More on these in later sections.

4.3.3 Nested Containers with Docker-in-Docker

While more complex in nature (not by too much, I promise), a nested Docker or "Docker-in-Docker" (*DIND*) provides the best of both the single container and multiple container uses cases for a build pattern, with a modest increase in complexity, **and**, very specific security requirements, as we'll be exposing the Docker socket (a *UDS* type of socket) to the top-level container. In such cases, the top-level container should be invoked from a trusted, verified, (and usually built in-house) Docker image.

Additionally, to clear up some terminology, there are at least two ways (at this time, excluding solutions that encapsulate containers in *VMs*) to run nested containers or "Docker-in-Docker":

- "Docker-in-Docker via `dind` [19]: a dedicated Docker image designed to allow for nested Docker deployments. Targeted primarily at use cases involving the development and testing of Docker itself.
- "Sharing the Docker socket" via mounting `/var/run/docker.sock` in "child" containers.

For the sake of our analysis of build patterns, we'll focus on the latter of the two approaches noted above. Also, to avoid ambiguity, I will use *DIND* to refer to the general approach of nested containers or "Docker-in-Docker", while the term (in monospaced font) of `dind` refers to the specific "Docker-in-Docker" implementation covered in [19] (which likely won't be mentioned much more, if at all, in the rest of this book). The use of this approach (i..e "sharing the Docker socket") is quite straightforward: we just launch a Docker container while mounting the Docker *UDS* (i.e. *API* entry point for communicating with Docker) in the top-level container we launch. This allows the top-level container to launch additional "child" containers (well, "sibling" containers actually; more on that later), which it could not accomplish without the use of this command-line argument. To use this feature, we just add the following to our `docker run` invocation, and we're all set: `--volume="/var/run/docker.sock:/var/run/docker.sock:rw"`

Warning: This option (i.e. mounting `/var/run/docker.sock` within a container) should **only** be used in specific circumstances where its use is needed and justified (i.e. build infrastructure that's only executing safe/trusted code and containers, local development use with safe/trusted environments, the DevOps engineers maintaining the system are aware that containers running with this capability effectively have root control over the host *OS* and all the security implications that go along with it, etc.).

This functionality should not be enabled in a non-hardened/isolated environment, as it is easily exploited to allow for priviledge esclatation and eventual compromise of the machine (and even encompassing infrastructure) [20] [21] [22] [23]. This being said, many common, modern *CI/CD* systems [24] [25]: either support or even advise (under specific circumstances) the use of such an approach. While acknowledging the usefulness of this approach and its security implications, I've elected to not qualify the approach overall, and will instead focus on how it is used as a build pattern. The reader is free to draw their own conclusion with respect to whether or not the utility provided by such a method warrants the extra security policies/implementations required to lock it down.

Also, the version of Docker installed within the container itself must be *API*-compatible with that running on the host *OS*, so there is at least that requirement on the host *OS* in terms of software loadout.

With all this being said, all of the examples in this chapter make use of "rootless Docker" at every step along the way, even when using "Docker-in-Docker", "Kubernetes-in-Docker", etc. This greatly mitigates the aforementioned concerns.

Now, for actually using this approach to execute a build operation.

Listing 4.16: Docker-in-Docker launcher: parent container.

```
# Docker runtime image to use for building artifacts.
DOCKER_IMG="ubuntu:focal"

# Launch docker container with DIND capabilities.
docker run \
    --rm -it \
    --volume="$(readlink -f .):/work:rw" \
    --workdir="/work" \
    --volume="/var/run/docker.sock:/var/run/docker.sock:rw" \
    ${DOCKER_IMG} \
    ${@}

# Confirm we can see some files from the host OS.
root@9ad1796a0ef5:/work# ls -la
total 24
drwxrwxr-x 2 1000 1000 4096 Aug  3 21:28 .
drwxr-xr-x 1 root root 4096 Aug  3 21:28 ..
-rw-rw-r-- 1 1000 1000  150 Aug  1 17:57 Dockerfile
-rw-rw-r-- 1 1000 1000   72 Aug  1 17:37 Makefile
-rw-rw-r-- 1 1000 1000   77 Aug  1 17:37 helloworld.c
-rwxrwxr-x 1 1000 1000  571 Aug  1 18:25 iax.sh

# Done.
root@9ad1796a0ef5:/work# exit
exit
```

Great: we've re-enacted our single container use case. Now, let's attempt to launch another container, the "child" container (depth is 1, as this is a single level of nesting), from within the "parent" container (depth is 0, as it was invoked directly from the host *OS* via `docker run`).

Listing 4.17: Docker-in-Docker launcher: parent container.

```
# Docker runtime image to use for building artifacts (parent image).
DOCKER_IMG="ubuntu:focal"

# Launch parent docker container with DIND capabilities.
docker run \
    --rm -it \
    --volume="$(readlink -f .):/work:rw" \
    --workdir="/work" \
    --volume="/var/run/docker.sock:/var/run/docker.sock:rw" \
    ${DOCKER_IMG} \
    ${@}

# Confirm we can see some files from the host OS.
root@b9b5d9166edf:/work# ls -la
total 24
drwxrwxr-x 2 1000 1000 4096 Aug  3 21:28 .
drwxr-xr-x 1 root root 4096 Aug  3 21:28 ..
-rw-rw-r-- 1 1000 1000  150 Aug  1 17:57 Dockerfile
-rw-rw-r-- 1 1000 1000   72 Aug  1 17:37 Makefile
-rw-rw-r-- 1 1000 1000   77 Aug  1 17:37 helloworld.c
-rwxrwxr-x 1 1000 1000  571 Aug  1 18:25 iax.sh

# Confirm this is Ubuntu "Focal" (i.e. 20.04 LTS).
root@b9b5d9166edf:/work# cat /etc/lsb-release
DISTRIB_ID=Ubuntu
DISTRIB_RELEASE=20.04
DISTRIB_CODENAME=focal
DISTRIB_DESCRIPTION="Ubuntu 20.04.2 LTS"

# Make sure our host OS files are visible: check.
root@b9b5d9166edf:/work# ls
Makefile  README.md  diagrams.drawio  doc  google2b0f328e0f93c24f.html
iax.sh

# Launch child container (no need to also give it DIND capabilities, in this
# case).
# Docker runtime image to use for building artifacts (child image).
DOCKER_IMG_CHILD="debian:buster"
docker run \
    --rm -it \
    --volume="$(readlink -f .):/work:rw" \
    --workdir="/work" \
    ${DOCKER_IMG_CHILD} \
    ${@}

bash: docker: command not found
# Oh? Seems the "docker" package isn't bundled into "baseline" container
# images by default (not typically needed, consumes extract space, etc.).

# Done, for now.
```

(continues on next page)

(continued from previous page)

```
root@b9b5d9166edf:/work# exit
exit
```

Well, that was a short exercise. Looks like we'll need to define our own custom "parent" or "top-level" Docker image rather than using a "vanilla" baseline image. While we're at it, let's define a custom "child" Docker image too, for the sake of being thorough. First, let's create the Dockerfiles for the parent and child image, along with the relevant `.dockerignore` files too.

Listing 4.18: Dockerfile.dind_example.parent that extends Ubuntu.

```
FROM ubuntu:focal as baseline

# System packages. Make sure "docker" is included.
RUN apt update -y && \
    apt install -y \
        docker.io \
        jq \
        lsb-release \
        make \
        gcc \
    && \
    apt clean -y
```

Listing 4.19: Dockerfile.dind_example.parent.dockerignore for Dockerfile.dind_example.parent

```
# Ignore everything by default. Have to manually add entries permitted
# for inclusion.
*
```

Listing 4.20: Dockerfile.dind_example.child that extends Debian.

```
FROM debian:buster as baseline

# System packages.
RUN apt update -y && \
    apt install -y \
        jq \
        lsb-release \
        make \
        gcc \
    && \
    apt clean -y
```

Listing 4.21: Dockerfile.dind_example.child.dockerignore for Dockerfile.dind_example.child

```
# Ignore everything by default. Have to manually add entries permitted
# for inclusion.
*
```

While we're at it, let's modify our `Makefile` to be able to build these images, so we're not constantly re-typing the `docker build` operations into the console manually. Let's include the `docker run` operations as goals we can invoke

as well.

Listing 4.22: Makefile for building our Docker images.

```
# Defaults and global settings.
.DEFAULT_GOAL: all
PARENT_TAG=dind_example_parent:local
CHILD_TAG=dind_example_child:local

# Default goal (no longer builds C app, but can via manually running
# "make app").
.PHONY: all
all: docker_parent docker_child
        @echo "Done build."

# Build parent Docker image.
.PHONY: docker_parent
docker_parent: Dockerfile.dind_example.parent Dockerfile.dind_example.parent.dockerignore
        DOCKER_BUILDKIT=1 docker build \
                        -t "$(PARENT_TAG)" \
                        --target baseline \
                        -f Dockerfile.dind_example.parent \
                        .

# Build child Docker image.
.PHONY: docker_child
docker_child: Dockerfile.dind_example.child Dockerfile.dind_example.child.dockerignore
        DOCKER_BUILDKIT=1 docker build \
                        -t "$(CHILD_TAG)" \
                        --target baseline \
                        -f Dockerfile.dind_example.child \
                        .

# Launch parent container.
.PHONY: run_parent
run_parent:
        docker run \
                --rm -it \
                --volume="$(shell readlink -f .):/work:rw" \
                --workdir="/work" \
                --volume="/var/run/docker.sock:/var/run/docker.sock:rw" \
                $(PARENT_TAG)

# Launch child container.
.PHONY: run_child
run_child:
        docker run \
                --rm -it \
                --volume="$(shell readlink -f .):/work:rw" \
                --workdir="/work" \
                --volume="/var/run/docker.sock:/var/run/docker.sock:rw" \
                $(CHILD_TAG)

```

(continues on next page)

(continued from previous page)

```
# Build our C app.
.PHONY: app
app: helloworld_app
helloworld_app:
        gcc helloworld.c -o $@
```

Alright then: let's try this again.

Listing 4.23: Docker-in-Docker ateempt: round 2.

```
# Launch the parent-level container.
owner@darkstar$> make run_parent
docker run \
      --rm -it \
      --volume="/home/owner/work:/work:rw" \
      --workdir="/work" \
      --volume="/var/run/docker.sock:/var/run/docker.sock:rw" \
      dind_example_parent:local

# Nice! Let's verify the OS used by the parent-level container. Should be
# Ubuntu 20.04.
root@d1d839fde6d4:/work# lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 20.04.2 LTS
Release:        20.04
Codename:       focal

# Confirmed! Let's make sure we can see the files we mounted from the host
# OS.
root@d1d839fde6d4:/work# ls -a
.  ..  Dockerfile  Dockerfile.dind_example.child  Dockerfile.dind_example.child.
↪dockerignore  Dockerfile.dind_example.parent  Dockerfile.dind_example.parent.
↪dockerignore  Makefile  helloworld.c  iax.sh

# So far, so good. Now let's try to launch the child container, since our
# parent-level container has Docker installed, and we're sharing the Docker
# socket with it.
root@d1d839fde6d4:/work# make run_child
docker run \
        --rm -it \
        --volume="/work:/work:rw" \
        --workdir="/work" \
        --volume="/var/run/docker.sock:/var/run/docker.sock:rw" \
        dind_example_child:local

# This looks promising. Let's verify the child container is a Debian 10
# "Buster" release.
root@7662f05f4b1b:/work# lsb_release -a
No LSB modules are available.
Distributor ID: Debian
Description:    Debian GNU/Linux 10 (buster)
```

(continues on next page)

(continued from previous page)

```
Release:        10
Codename:       buster

# Great! We have a parent-level container that can bootstrap our build
# process, and a child-level container that can run the actual build. Let's
# make sure our files are present.
root@7662f05f4b1b:/work# ls -a
.  ..

# Wait, what? None of our files are present. Time to debug. Exit child
# container.
root@7662f05f4b1b:/work# exit
exit

# Exit parent container.
root@d1d839fde6d4:/work# exit
exit
```

What happened here? We can wee from the console output of the `make run_child` command that we're mapping `/work` within the parent container to `/work` in the child container, and from the output of the `make run_parent` command that we're mapping the *PWD* in the host *OS* to `/work` in the parent container. By transitivity, we should expect that mounting *PWD* (host *OS*) to `/work` (parent container) to `/work` (child container) should expose the files on our host *OS* to the child container. This doesn't appear to be the case: we only get as far as making files on the host *OS* visible to the parent container. What happened?

Well, as it turns out, "Docker-in-Docker" or "nested containers", can be a bit of a misnomer. Even though we use terms like "parent container" and "child container", from an implementation perspective (i.e. "under the hood"), child containers might be better described as "sibling" containers. Consider Fig. 4.3 - this looks roughly what one may imagine represents the concept of "nested containers": the parent container runs directly from the host *OS*, and the child container runs within the parent container.

This assumption turns out to be incorrect, and in actuality, the "topology" of our containers is closer to that of Fig. 4.4.

So, what does this all mean? For a in-depth summary, the reader is encouraged to review [1], but for now, the key takeaway is this: if we want a mount point that is visible within the parent container to be visible in the child container, the volume mount options passed to the `docker run` invocation of the child container must match those that were passed to the parent container. For example, in our most recent attempt to use nested containers, notice the different in the volume mount commands:

Listing 4.24: Docker-in-Docker launchers: parent/child invocation comparison.

```
# Parent container.
docker run \
      ...
      ...
      --volume="/home/owner/work:/work:rw" \
      --workdir="/work" \
      ...
      ...

# Child container.
root@d1d839fde6d4:/work# make run_child
docker run \
```

(continues on next page)

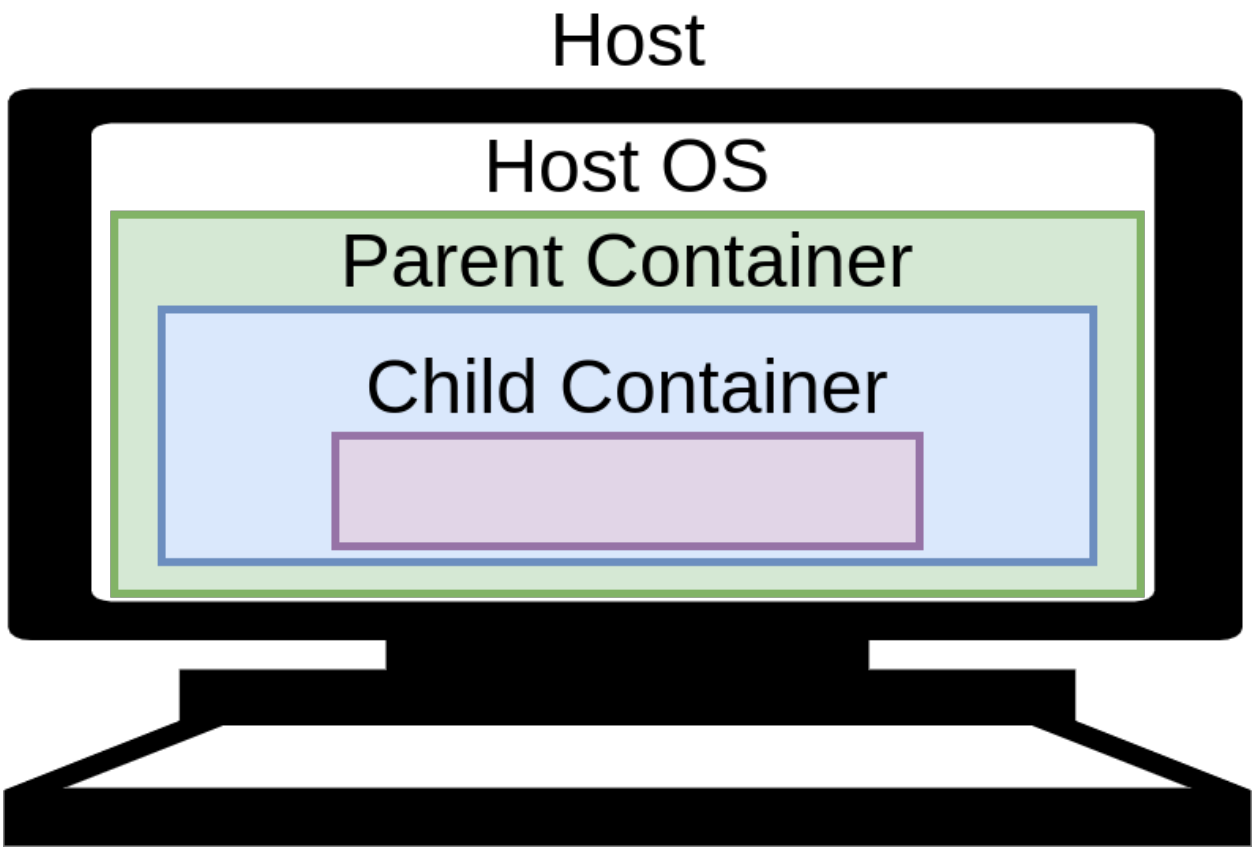

Fig. 4.3: Conceptual model (incorrect, as it turns out) for nested containers.

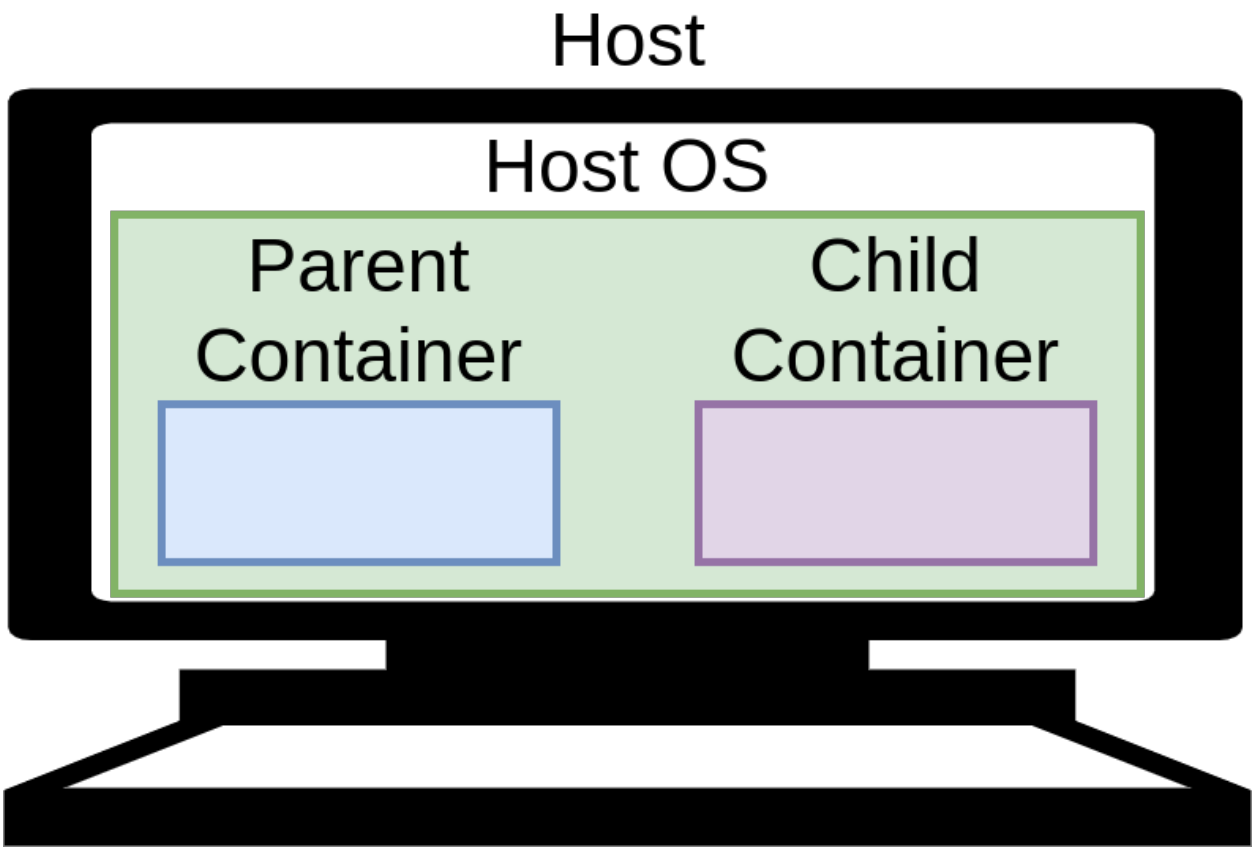

Fig. 4.4: Model (approximate) for sibling (i.e. “nested”) containers.

(continued from previous page)

```
        ...
        ...
        --volume="/work:/work:rw" \
        --workdir="/work" \
        ...
        ...
```

If we were to somehow pass the same `--volume` command used for the parent container, to the child container's invocation, we can make the files in `/home/owner/work` visible to **both** the parent container and the child container. Ignoring the trivial approach of simply hard-coding the paths (our project isn't very portable anymore if we do that, since every developer that checks out a copy of the project has to go modifying hard-coded paths, and hopefully not committing said changes back to the shared repository). Rather, let's just pass the values used by the parent container invocation to the child container's invocation as environment variables. That should do the trick. First, our modified `Makefile`:

Listing 4.25: Makefile.sibling_container_exports

```
# Defaults and global settings.
.DEFAULT_GOAL: all
PARENT_TAG=dind_example_parent:local
CHILD_TAG=dind_example_child:local

# For mounting paths in parent and child containers.
# Only set HOST_SRC_PATH if it's not already set. We expect the
# invocation/launch of the parent container to "see" this value as un-set,
# while the child container should already see it set via "docker run ... -e
# ...".
ifeq ($(HOST_PATH_SRC),)
HOST_PATH_SRC:=$(shell readlink -f .)
endif
HOST_PATH_DST:=/work

# Default goal (no longer builds C app, but can via manually running
# "make app").
.PHONY: all
all: docker_parent docker_child
        @echo "Done build."

# Build parent Docker image.
.PHONY: docker_parent
docker_parent: Dockerfile.dind_example.parent Dockerfile.dind_example.parent.dockerignore
        DOCKER_BUILDKIT=1 docker build \
                        -t "$(PARENT_TAG)" \
                        --target baseline \
                        -f Dockerfile.dind_example.parent \
                        .

# Build child Docker image.
.PHONY: docker_child
docker_child: Dockerfile.dind_example.child Dockerfile.dind_example.child.dockerignore
        DOCKER_BUILDKIT=1 docker build \
                        -t "$(CHILD_TAG)" \
```

(continues on next page)

(continued from previous page)

```
                        --target baseline \
                        -f Dockerfile.dind_example.child \
                        .

# Launch parent container.
.PHONY: run_parent_alt
run_parent_alt:
        docker run \
                --rm -it \
                --volume="$(HOST_PATH_SRC):$(HOST_PATH_DST):rw" \
                --workdir="$(HOST_PATH_DST)" \
                -e HOST_PATH_SRC="$(HOST_PATH_SRC)" \
                -e HOST_PATH_DST="$(HOST_PATH_DST)" \
                --volume="/var/run/docker.sock:/var/run/docker.sock:rw" \
                $(PARENT_TAG)

# Launch child container.
.PHONY: run_child_alt
run_child_alt:
        docker run \
                --rm -it \
                --volume="$(HOST_PATH_SRC):$(HOST_PATH_DST):rw" \
                --workdir="$(HOST_PATH_DST)" \
                -e HOST_PATH_SRC="$(HOST_PATH_SRC)" \
                -e HOST_PATH_DST="$(HOST_PATH_DST)" \
                --workdir="/work" \
                --volume="/var/run/docker.sock:/var/run/docker.sock:rw" \
                $(CHILD_TAG)

# Build our C app.
.PHONY: app
app: helloworld_app
helloworld_app:
        gcc helloworld.c -o $@
```

Now let's try this again:

Listing 4.26: Docker-in-Docker launcher: round 3.

```
# Launch the parent container with our new Makefile (need to specify Makefile
# name and rule, since we made a new Makefile to hold our changes).
make -f Makefile.sibling_container_exports run_parent_alt
docker run \
        --rm -it \
        --volume="/home/owner/work:/work:rw" \
        --workdir="/work" \
        -e HOST_PATH_SRC="/home/owner/work" \
        -e HOST_PATH_DST="/work" \
        --volume="/var/run/docker.sock:/var/run/docker.sock:rw" \
        dind_example_parent:local

# Files are visible from parent container.
```

(continues on next page)

(continued from previous page)

```
root@fc4380b8bc06:/work# ls
Dockerfile                      Dockerfile.dind_example.child.dockerignore  Dockerfile.
↪dind_example.parent.dockerignore  Makefile.sibling_container_exports  iax.sh
Dockerfile.dind_example.child   Dockerfile.dind_example.parent              Makefile
↪                                 helloworld.c

# Verify environment variables were passed to parent container (it needs to
# pass them along to the child container).
root@fc4380b8bc06:/work# export | grep HOST_PATH
declare -x HOST_PATH_DST="/work"
declare -x HOST_PATH_SRC="/home/owner/work"

# Looking good so far. Now let's launch the child container.
root@fc4380b8bc06:/work# make -f Makefile.sibling_container_exports run_child_alt
docker run \
        --rm -it \
        --volume="/home/owner/work:/work:rw" \
        --workdir="/work" \
        -e HOST_PATH_SRC="/home/owner/work" \
        -e HOST_PATH_DST="/work" \
        --workdir="/work" \
        --volume="/var/run/docker.sock:/var/run/docker.sock:rw" \
        dind_example_child:local

# Can we see our files in the child container?
root@a841398f3d24:/work# ls
Dockerfile                      Dockerfile.dind_example.child.dockerignore  Dockerfile.
↪dind_example.parent.dockerignore  Makefile.sibling_container_exports  iax.sh
Dockerfile.dind_example.child   Dockerfile.dind_example.parent              Makefile
↪                                 helloworld.c

# YES!!! Now, can our child container compile our application (the whole
# point of having the child container: a dedicated container to build a
# specific app via our top-level Makefile).
root@a841398f3d24:/work# make -f Makefile.sibling_container_exports app
gcc helloworld.c -o helloworld_app

# Beautiful! All done! Exit child container.
root@a841398f3d24:/work# exit
exit

# Exit parent container.
root@fc4380b8bc06:/work# exit
exit
```

Excellent! We are able to have a top-level container launch child containers on-demand to carry out various stages of the build. Furthermore, we could make additional modifications to a top-level launcher script (e.g. `iax.sh`) so that it can bootstrap the entire build from a parent-level container via something like `./iax.sh make run_parent`, which can then trigger the various intermediate build steps.

Before we conclude this section, there's one last edge-case I'd like to visit (which turns out to be rather common): what if we can't 100% control the parent-level container (i.e. have it export the `HOST_PATH_SRC` and `HOST_PATH_DST` environment variables to the child container)? This can happen in cases where using build systems as part of a *CI/CD*

pipeline that have the runners/builders (i.e. hosts, physical or virtual) execute build jobs from a container [24] [25]. If this is the case, we'll have problems with *CI/CD* builds, as we'll once again have the problem where the child container doesn't have the necessary information required for it to properly access the host *OS* file mount. What is one to do?

It turns out there is a means (albeit, somewhat "hacky'ish") that allows a container to query information on other running containers, if we allow our "parent" container to access `/var/lib/docker.sock` (hence the previous warnings that allowing containers access to this grant them elevated control of the host system). Before proceeding, let's establish some (loose) terminology.

- "Launcher container": the true "parent container". It's the top-level container that a *CI/CD* system uses to launch a build job.
- "Bootstrap container": our own "parent container" (i.e. Ubuntu 20.04) that we've been working with so far. Since it's not being used to the launch the build job, it's technically a "child container" (and the "child container" is actually a "grandchild" container). The "Launcher container" is responsible solely for invoking the "bootstrap container", which is then responsible for handling the rest of the build.
- "Builder container": our own "child container" (i.e. Debian 10 "Buster") that we've been using so far (is actually a "grandchild container" due to multiple levels of parent containers).

Now, let's assume that the launcher container invokes the bootstrap container via something like `docker run ... make run_child`, and that the Docker socket on any system is **always** `/var/run/docker.sock`, and that it is being mounted in the launcher container via `--volume=/var/run/docker.sock:/var/run/docker.sock` (i.e. while tools like `iax.sh` are useful for local builds, *CI/CD* systems will never use them, and prefer their own methods/scripts for bootstrapping builds).

With these assumptions in place, we should be able to, using some Docker commands (i.e. `docker inspect`) execute various queries within the context of the bootstrap container, and then pass them along to child (i.e. "builder") containers at run time. This should let us, within the bootstrap container, dynamically determine the equivalent of `HOST_PATH_SRC` and `HOST_PATH_DST`. First, let's review our modified `Makefile`, and our introspection helper script.

Listing 4.27: Makefile.introspection

```
# Defaults and global settings.
.DEFAULT_GOAL: all
PARENT_TAG=dind_example_parent:local
CHILD_TAG=dind_example_child:local

# For mounting paths in parent and child containers.
# Only set HOST_SRC_PATH if it's not already set. We expect the
# invocation/launch of the parent container to "see" this value as un-set,
# while the child container should already see it set via "docker run ... -e
# ...".
ifeq ($(HOST_PATH_SRC),)
HOST_PATH_SRC:=$(shell readlink -f .)
endif
HOST_PATH_DST:=/work

# Psuedo-random path to PWD. Build systems often use tools like `mktmp` to
# create throwaway intermediate build paths for jobs.
HOST_PATH_SRC_RANDOM:=$(HOST_PATH_SRC)_$(shell shuf -i 1000000-9999999 -n 1)

# Default goal (no longer builds C app, but can via manually running
# "make app").
.PHONY: all
all: docker_parent docker_child
```

(continues on next page)

(continued from previous page)

```
        @echo "Done build."

# Launcher image (our "parent" image from earlier examples). We'll also use it
# as our bootstrap image.
.PHONY: docker_parent
docker_parent: Dockerfile.dind_example.parent Dockerfile.dind_example.parent.dockerignore
        DOCKER_BUILDKIT=1 docker build \
                        -t "$(PARENT_TAG)" \
                        --target baseline \
                        -f Dockerfile.dind_example.parent \
                        .


# Builder image (our "child" image from earlier examples).
.PHONY: docker_child
docker_child: Dockerfile.dind_example.child Dockerfile.dind_example.child.dockerignore
        DOCKER_BUILDKIT=1 docker build \
                        -t "$(CHILD_TAG)" \
                        --target baseline \
                        -f Dockerfile.dind_example.child \
                        .

# Launch "launcher" container.
.PHONY: run_launcher
run_launcher:
        # Create pseudo-random path to emulate behavior of CI/CD systems.
        # For educational purposes only.
        # DON'T USE SUDO IN YOUR BUILD SCRIPTS!!!
        mkdir -p $(HOST_PATH_SRC_RANDOM)
        sudo mount --bind $(HOST_PATH_SRC) $(HOST_PATH_SRC_RANDOM)
        @echo "RANDOM PATH: $(HOST_PATH_SRC_RANDOM)"

        docker run \
                --rm -it \
                --volume="$(HOST_PATH_SRC_RANDOM):$(HOST_PATH_DST):rw" \
                --workdir="$(HOST_PATH_DST)" \
                -e HOST_PATH_SRC="$(HOST_PATH_SRC)" \
                -e HOST_PATH_DST="$(HOST_PATH_DST)" \
                --volume="/var/run/docker.sock:/var/run/docker.sock:rw" \
                $(PARENT_TAG) \
                docker run \
                        --rm -it \
                        --volume="$(HOST_PATH_SRC_RANDOM):$(HOST_PATH_DST):rw" \
                        --workdir="$(HOST_PATH_DST)" \
                        --volume="/var/run/docker.sock:/var/run/docker.sock:rw" \
                        $(PARENT_TAG)

        # Clean up.
        sudo umount $(HOST_PATH_SRC_RANDOM)

# Build our C app.
.PHONY: app
```

(continues on next page)

(continued from previous page)

```
app: helloworld_app
helloworld_app:
        gcc helloworld.c -o $@
```

Listing 4.28: docker_mount_resolver.sh

```
#!/bin/bash

# Needs an argument.
if [ -z "${1}" ]; then
    exit 1
fi

# Get the name of the image for the currently running container.
SELF_IMAGE_NAME=$(basename "$(head /proc/1/cgroup)")

# Might need to strip out some extra info depending how old your Docker
# installation is.
SELF_IMAGE_NAME=$(echo ${SELF_IMAGE_NAME} | sed "s/^docker-//g" | sed "s/\.scope$//g")

# Search mounts associated with currently running container. Return a match if
# found.
docker inspect --format '{{json .Mounts }}' "${SELF_IMAGE_NAME}" | jq -c '.[]' | while␣
↪read key ; do
    src=$(echo ${key} | jq -r .Source)
    dst=$(echo ${key} | jq -r .Destination)
    echo "SRC:${src} DST:${dst}" >&2
    if [[ "${1}" == "${dst}" ]]; then
        echo "${src}"
        exit 0
    fi
done
```

Now, let's launch our launcher container (e.g. represents something similar to a GitLab or Jenkins "runner" image), and then launch our bootstrap container (effectively the "parent" container we've been working with throughout this section), and see what useful information we can glean.

Listing 4.29: Docker-in-Docker launcher: using introspection.

```
owner@darkstar$> make -f Makefile.introspection run_launcher
# Create pseudo-random path to emulate behavior of CI/CD systems.
# For educational purposes only.
# DON'T USE SUDO IN YOUR BUILD SCRIPTS!!!
mkdir -p /home/owner/work_2338367
sudo mount --bind /home/owner/work /home/owner/work_2338367
RANDOM PATH: /home/owner/work_2338367
docker run \
        --rm -it \
        --volume="/home/owner/work_2338367:/work:rw" \
        --workdir="/work" \
        -e HOST_PATH_SRC="/home/owner/work" \
        -e HOST_PATH_DST="/work" \
```

(continues on next page)

(continued from previous page)

```
        --volume="/var/run/docker.sock:/var/run/docker.sock:rw" \
        dind_example_parent:local \
        docker run \
                --rm -it \
                --volume="/home/owner/work_2338367:/work:rw" \
                --workdir="/work" \
                --volume="/var/run/docker.sock:/var/run/docker.sock:rw" \
                dind_example_parent:local

# Launched 2 levels of Docker, now in the "bootstrap" container.
root@094da58d8a64:/work# ls
Dockerfile                    Dockerfile.dind_example.child.dockerignore  Dockerfile.
↪dind_example.parent.dockerignore  Makefile.introspection              docker_mount_
↪resolver.sh  helloworld_app
Dockerfile.dind_example.child  Dockerfile.dind_example.parent              Makefile
↪                              Makefile.sibling_container_exports  helloworld.c
↪      iax.sh

# Make sure no env vars to "cheat" with.
root@094da58d8a64:/work# export | grep HOST_PATH

# Try to find a Docker mount in the parent container matching "blah" (we know
# this will fail, just running the script to see what kind of results we're
# searching through).
root@094da58d8a64:/work# ./docker_mount_resolver.sh blah
SRC:/home/owner/work_2338367 DST:/work
SRC:/var/run/docker.sock DST:/var/run/docker.sock

# Done.
root@094da58d8a64:/work# exit
# Clean up.
sudo umount /home/owner/code/repos/iaxes/pronk8s-src/examples/build_patterns/dind_
↪example_2338367
```

So we can see that through the use of a tool like our `docker_mount_resolver.sh` script we can see the mounts used by the parent of our bootstrap container (i.e. the launcher container), and we could make an educated guess as to which of the results yields the pair of strings we'd pass to a child container if we wanted it to be able to access files from the host *OS* (hint: in the above example, it's `SRC:/home/owner/work_2338367 DST:/work`). However, the key point to keep in mind is that **it's still a guess**. Unless your *CI/CD* system exports some environment variables or provides some means of querying/exporting these values at run time, you'll have to use some sort of run-time introspection to fully leverage nested containers in your own *CI/CD* pipelines. This can be further compounded with minor discrepancies between runners (i.e. hosts that execute builds for the pipeline), such as a difference in path names or path patterns on in-house versus *AWS* instances of runners. Also, you'll need to get the necessary approvals from your *ITS* department due to the security implications of sharing the Docker socket, and so on. So, it's worth noting that as useful and flexible nested Docker/container builds are, they are not without their caveats, and you may end up finding it easier (in terms of not using introspection scripts, getting approval from *ITS* on how you'll implement your *CI/CD* pipelines, etc.) to use multi-container builds rather than nested container builds.

Note: Build pattern viability metrics.

Repeatable: moderately repeatable. DevOps team members will likely have to debug numerous corner-cases that arise due to differences in CI/CD builds versus local/developer builds.

Reliable: very reliable, provided the host OS itself is stable.

Maintainable: yes, but additional testing/debugging by developers and DevOps to "get it right initially" (i.e. working out quirks in build infrastructure for CI/CD builds versus local/developer builds). More complex to maintain as developers responsible for maintaining the build scripts will need to be sufficiently versed in Linux and Docker.

Scalable: extremely scalable.

4.3.4 Advanced Example: Single Container with Docker

This section focuses exclusively on an advanced version of the `iax.sh` launcher script covered in *Single Container with Docker*, with emphasis on day-to-day "real world" scenarios where one would use a Docker container for building projects from source locally (and has no relevance to *CI/CD* build infrastructure). Therefore, the following section is only recommended for those that will be building (or supporting builds) of source trees on local development environments (e.g. workstations/laptops for individual contributors).

Now, with that said, let's just demonstrate our "improved" launcher script, and then we'll go through the various options, line-by-line, and explain their significance (with "real-world" examples) as we go along.

Listing 4.30: iax.sh - advanced Docker launcher script.

```
#!/bin/bash
################################################################################
# @brief:       Docker-bootstrap script for building projects via a Docker
#               image.
# @author:      Matthew Giassa.
# @e-mail:      matthew@giassa.net
# @copyright:   Matthew Giassa (IAXES) 2017.
################################################################################

# Docker runtime image to use for building artifacts.
DOCKER_IMG="iaxes/iaxes-docker-builder-docs"

# Need to dynamically create a set of "--group-add" statements while iteration
# across all groups for which we are a member, otherwise a simple
# "--user=$(id -u):$(id -g)" argument to "docker run" will only capture our
# primary group ID, and overlook our secondary group IDs (which means we won't
# be a member of the "docker" group when executing nested containers as a
# rootless user, causing headaches).
MY_GROUPS=("$(groups)")
MY_GROUPS_STR=""
for grp in ${MY_GROUPS[@]}; do
    if [[ "${grp}" == "$(whoami)" ]]; then
        continue
    else
        # Need to use GID, not group name.
        gid="$(echo $(getent group ${grp}) | awk -F ':' '{print $3}')"
        MY_GROUPS_STR+="--group-add ${gid} "
    fi
done

# Debug logging.
# echo "My groups: ${MY_GROUPS[@]}"     >&2
```

(continues on next page)

(continued from previous page)

```
# echo "Group string: ${MY_GROUPS_STR}" >&2

# Launch docker container.
# * Setup PWD on host to mount to "/work" in the guest/container.
# * Forward access to SSH agent and host credentials (so container uses same
#   user/group permissions as current user that launches the script).
# * Make DIND possible (i.e. expose host Docker Unix domain socket to
#   guest/container).
# * Make the home directory be "/work" so that it's always writeable (allows us
#   to better handle rootless DIND and KIND).
# * Use host networking (non-ideal, but necessary for top-level containers to
#   access certain services like KIND-exposed kubeadm ports redirecting to port
#   6443 inside a pod without requiring a bridge being setup in advance).
docker run \
    --rm -it \
    --user="$(id -u):$(id -g)" \
    ${MY_GROUPS_STR} \
    --volume="/etc/passwd:/etc/passwd:ro" \
    --volume="/etc/group:/etc/group:ro" \
    --volume="$(readlink -f .):/work:rw" \
    --workdir="/work" \
    --volume="$(readlink -f ~/.ssh):$(readlink -f ~/.ssh):ro" \
    --volume="$(dirname ${SSH_AUTH_SOCK}):$(dirname ${SSH_AUTH_SOCK})" \
    -e SSH_AUTH_SOCK=${SSH_AUTH_SOCK} \
    --volume="/var/run/docker.sock:/var/run/docker.sock:rw" \
    -e HOME="/work" \
    --network=host \
    ${DOCKER_IMG} \
    ${@}
```

Before proceeding, it is worth noting that, for certain types of work, I will use a command-line *SSH* agent to cache credentials. This is so that I don't find myself repeatedly having to type in passphrases every single time I execute a `git` or `ssh` command. A common way to do this, assuming one has a `~/.ssh/id_rsa` private key and `~/.ssh/id_rsa.pub` public key, for example, is like so:

Listing 4.31: Example of launching an SSH agent instance to cache credentials.

```
eval `ssh-agent`
ssh-add ~/.ssh/id_rsa
```

After running this, subsequent commands will inherit the relevant environment variables the above-noted commands generate (i.e. `SSH_AGENT_PID`, `SSH_AUTH_SOCK`), so you could launch an instance of `tmux` [26] for example, and all panes/windows/etc. created within it would have access to this credential cache (helpful for having multiple "tabs" open at once and not having to manually enter credentials into all of them manually). For example, in Fig. 4.5, I have various editors, tools, etc.; running in the various *tmux* panes, and I've only had to enter my credentials once rather than 6+ times. When I eventually terminate the `tmux` instance itself at the end of my work day, the credential cache is gone, and we can just repeat the whole process again another day.

Why is this relevant, you might ask? Well, while the bulk of coding and editing (in my case) is done on the host *OS*, the majority of the build operations take place in containers. These containers, by default, do not have access to my *SSH* keys, credentials, etc.; so they cannot interact with various pieces of build infrastructure to carry out common tasks, such as checking out code from a `git` repository, downloading dependencies/intermediate-build-artifacts from secured repositories, etc. While in theory, one could duplicate their credentials into a Docker container (i.e. by modifying the

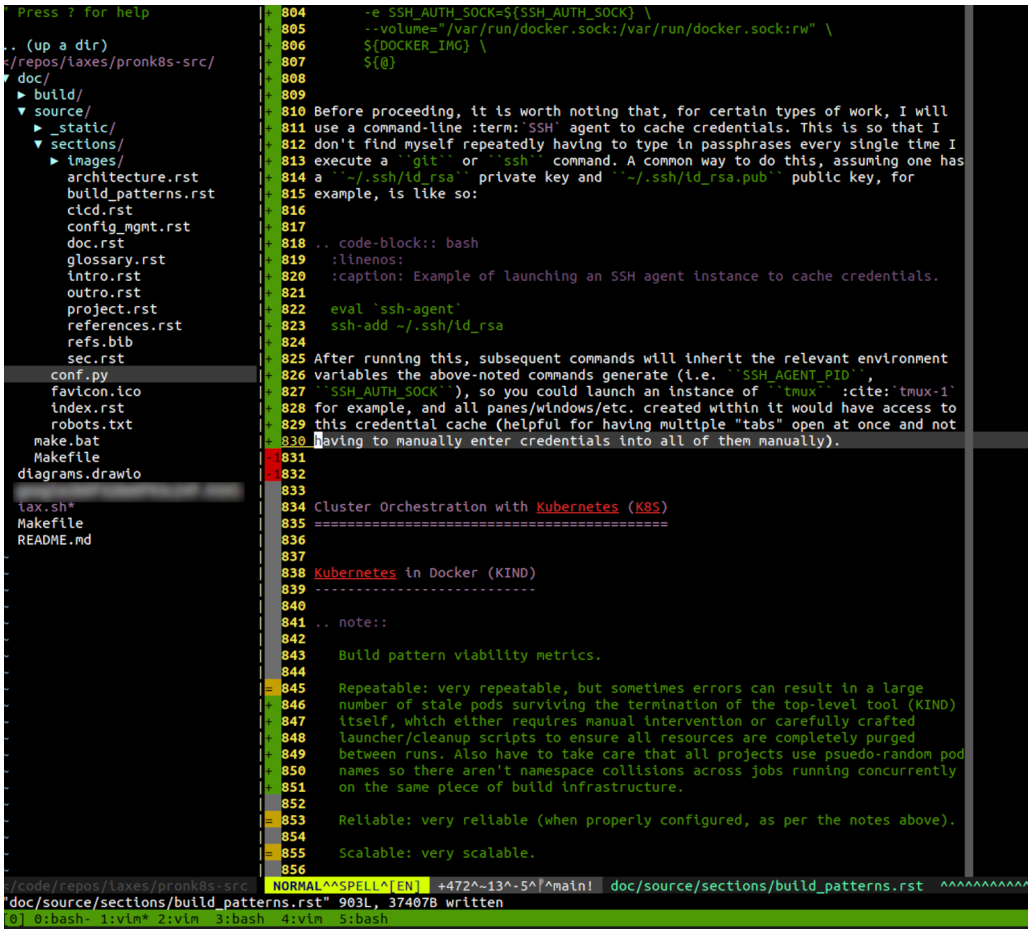

Fig. 4.5: Example of my (console-centric) development environment.

`Dockerfile` to bundle credentials and keys into the image, **which is a horrible idea**, as discussed in further detail in *Best Practises & Security Concerns*), there's a better, and much more temporary way to accomplish this (without having build artifacts containing credentials floating around on your workstation, or even worse, shared build infrastructure in case you accidentally published the image).

Now, on to the specific options we use when launching our builder image.

Listing 4.32: Description of advanced Docker launcher command.

```
docker run \
    --rm -it \
    --volume="$(readlink -f .):/work:rw" \
    --workdir="/work" \
    ...
    ...
    ...
    ${DOCKER_IMG} \
    ${@}
```

Nothing new here: this is how we've been launching builder containers so far (i.e. attach `STDIN`, allocate a pseudo-TTY, clean-up when done, mount the *PWD* on the host *OS* to `/work` inside the running container, pass along command-line arguments, etc.).

Listing 4.33: Description of advanced Docker launcher command.

```
docker run \
    ...
    ...
    ...
    --user="$(id -u):$(id -g)" \
    --volume="/etc/passwd:/etc/passwd:ro" \
    --volume="/etc/group:/etc/group:ro" \
    ...
    ...
    ...
    ${DOCKER_IMG} \
    ${@}
```

These commands instruct Docker to run under a specific user and group name (i.e. that of the current user session running on the host *OS*). The volume mounts (set to read-only on purpose so we don't give the container the opportunity to accidentally or deliberately corrupt them) supplement the `--user` command so we can map user and group names to their appropriate numerical *IDs* within the container. Why is this helpful? Well, if you just `docker run` a container with a volume mount in place (like in the various examples found in *Single Container with Docker*), and create a file (e.g. `touch /work/myfile.txt`), you'll notice the file is owned by the `root` user (example below).

Warning: The use of `--user="$(id -u):$(id -g)"` works in general, but it only associates the user's primary group, rather than the union of the primary group and secondary groups (necessary if we want to do certain types of complex operations like "rootless docker-in-docker"). Please see the *K8S* build pattern section later in this chapter for more details.

Listing 4.34: Container launched with root credentials.

```
# Launch our Docker container.
```

(continues on next page)

(continued from previous page)

```
owner@darkstar$> $> docker run -it --rm \
    --volume="$(readlink -f .):/work"
    --workdir="/work"
    ubuntu:focal

# Create a file while in the container.
root@8111f7b149c7:/work# touch myfile.txt

# Exit container, and verify file ownership.
root@8111f7b149c7:/work# exit
owner@darkstar$> ls -la myfile.txt
-rw-r--r-- 1 root 0 Aug  1 12:39 myfile.txt
```

This is no good! We now have a file owned by `root` in our *PWD* that we can't delete. If we want to purge our build stage or just change branches in `git`, we're out of luck. We now either require our user account to have `sudo` privileges (which is not always possible), or we need an administrator to purge the file for us (which isn't very helpful for a workflow we're aiming to automate significantly). If we use the `--user` and volume mounts in the previous example, all new files created in our volume mount are owned by the user that launched `iax.sh` in the first place (allowing for files to be cleaned up, deleted, etc.) at the user's discretion. On to the next set of options.

Listing 4.35: Example of launching a container with a host volume mount.

```
docker run \
    ...
    ...
    ...
    --volume="/var/run/docker.sock:/var/run/docker.sock:rw" \
    ...
    ...
    ...
    ${DOCKER_IMG} \
    ${@}
```

This is just for allowing functionality like Docker-in-Docker to work. On to the final set of "customizations". The security implications of this method are discussed earlier in this chapter, and the reader is encouraged to review it in case they've skipped said section.

Listing 4.36: Example of launching a container with a host volume mount.

```
docker run \
    ...
    ...
    ...
    --volume="$(readlink -f ~/.ssh):$(readlink -f ~/.ssh):ro" \
    --volume="$(dirname ${SSH_AUTH_SOCK}):$(dirname ${SSH_AUTH_SOCK})" \
    -e SSH_AUTH_SOCK=${SSH_AUTH_SOCK} \
    ...
    ...
    ...
    ${DOCKER_IMG} \
    ${@}
```

This last set of options should only be used in cases where you implicitly trust the contents of the builder container (i.e. provided to you by a trusted party and you've verified the contents via checksums or digital signatures). The first volume mount grants read-only access to your *SSH* key files (public **and** private files, and whatever other keys you have present in `~/.ssh`; I hope you're using passphrases to protect these files). The second volume mount provides access to the currently running `ssh-agent` instance that is caching our credentials in the background (while the subsequent environment variable export allows the container to know where to find the socket/path associated with the agent).

When we combine these various settings, we can take an ordering builder container (provided we trust it and can verify its contents), and give it various capabilities (i.e. make use of our currently running session to authenticate things like `git` and `ssh` commands, launch nested containers via Docker-in-Docker, make sure build artifacts have the same ownership as the user that invoked the container, etc.). Most importantly, all these settings are ephemeral, are provided entirely by additional command-line arguments supplied to `docker run`, and at no time have we taken credential files or sensitive information and "baked it in" to a concrete Docker image.

4.4 Cluster Orchestration with Kubernetes (K8S)

This section extends the topics in the preceding sections by throwing in a useful technology to the mix: *K8S*. This is accomplished by using container-centric build patterns, and introducing tools to create on-demand, throwaway clusters, and deploying applications into them on-the-fly. This is particularly useful for emulating complex build (i.e. *CI/CD*) systems that operate within a cluster, and we don't want to have to provision a dedicated centralized cluster for developers.

In fact, if individual developers have sufficient computing resources at their disposal (i.e. powerful-enough laptops/workstations/etc.), this is quite convenient, as individuals may test their code changes against a private, throwaway cluster, rather than having to share a common cluster that others may be using at the same time (requiring isolation between different deployments, possibly reserving the cluster outright if potentially-destructive testing needs to be done, etc.).

These topics will be revisited in future sections when we address topics involving automated testing (*Testing Strategies*). Finally, it is worth noting that in this chapter so far, various scripts have accumulated in `/home/owner/work` on the host *OS* (i.e. Dockerfiles, Makefiles, various build scripts) that were used in examples so far. From this point forward, we're going to purge the contents of this path and start with a fresh slate (i.e. an empty/clean `/home/owner/work` path).

4.4.1 Kubernetes in Docker (KIND)

Perhaps the most extreme example of a cloud native build pattern, the use of "Kubernetes in Docker" (*KIND*) makes extensive use of containers, as well as container orchestration in the form of *K8S* itself. In short, it literally invokes a Docker container, and configures a *K8S* cluster within it, which is amusing due to the fact that the cluster itself, which handles "cluster orchestration", which is effectively a grouping of various types of containers (some to manage or support normal cluster operations, and others which are just applications that we design ourselves). The net result is a container, running a cluster, than runs more containers ("containers in a cluster in a container", Fig. 4.6).

With respect to Fig. 4.6, we're hiding some implementation details, namely the critical components for a *K8S* installation (i.e. `etcd`, `kube-apiserver`, `kube-scheduler`, and `controller-manager` for the control plane; and `kubelet`, `kube-proxy`, and a container runtime such as `docker` for the nodes/workers [27]). A detailed example of these components can be seen in Fig. 4.7. For the sake of simplicity, the small purple boxes in Fig. 4.6 more-or-less equate to the architecture shown in Fig. 4.7, which the small red boxes represent "business applications" (i.e. programs that we write ourselves, encapsulate in Docker containers, and deploy to a *K8S* cluster).

Before proceeding, we're going to make this example just a bit more involved: we're going to throw more, believe it or not, Docker at the problem. For starters, I'd like this example to be as portable as possible, and in order to proceed, we'll need some additional tools (namely `kubectl`, for administering a *K8S* cluster), so I'd like to encapsulate the

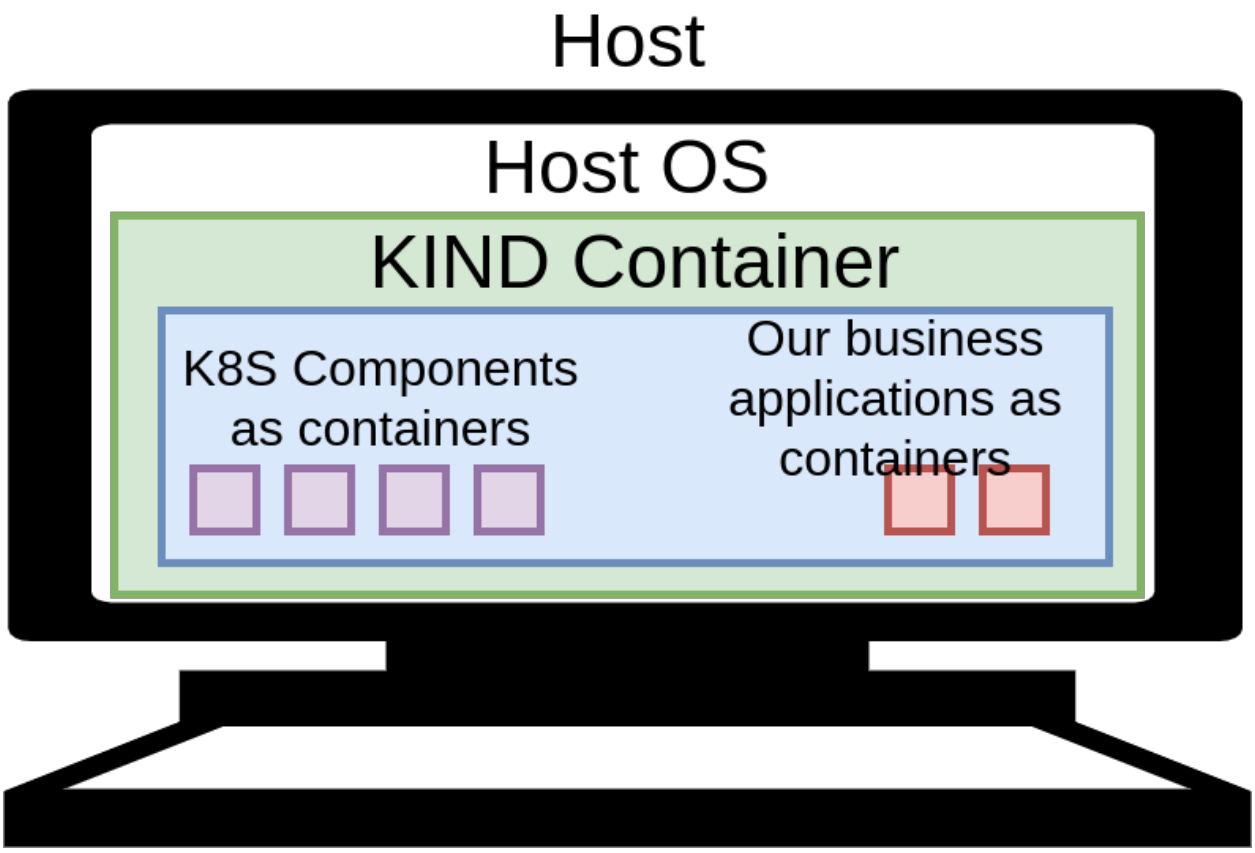

Fig. 4.6: Conceptual model of Kubernetes-in-Docker. Recall from earlier that nested containers, including use cases like *KIND*, are really "sibling containers".

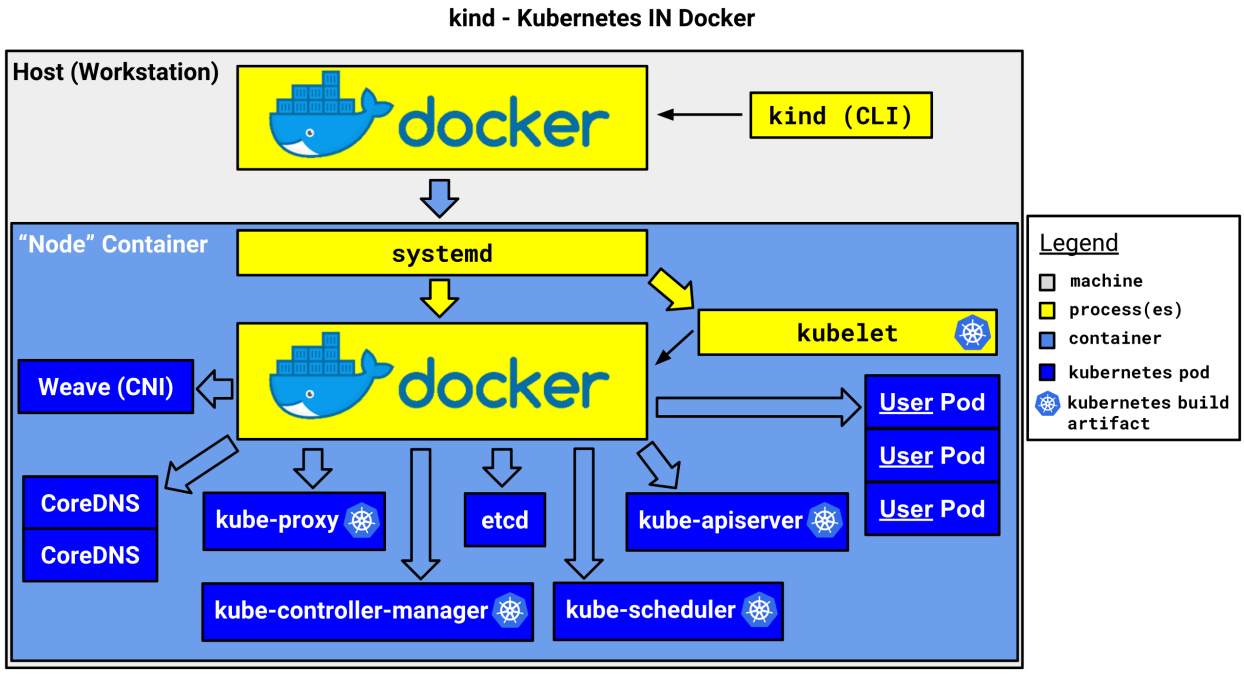

Fig. 4.7: Overview of core *K8S* components in KIND. (C) 2021 the KIND Authors. Used under the fair dealings clause under section 29 of the Copyright Act of Canada, circa 2012; for the purposes of non-profit scientific research and the promotion of scientific and educational advancement.

KIND container in a top-level "builder container", which will "house" the *KIND* Docker container and its own child processes ("containers in a cluster in a container in yet another container").

While this might appear to border on the absurd, it's a helpful way to make the overall project easier to share with colleagues, and allows it to be a more viable build pattern (i.e. allowing local/developer build environments and *CI/CD* pipelines to differ as little as possible). The "builder container" will also house some additional child containers for validating our application (i.e. test suites), and the "KIND container" for actually running the final result in a cluster. Our overall model will look something like Fig. 4.8. In the future (i.e. *Testing Strategies*) we can extend this even further to include additional testing containers as children of the "builder container" to do automated unit testing, among other tasks.

> **Warning:** At this time, the "app testing container" will be created, but not used. This will be revisited in *Testing Strategies*. For now, any (basic) testing we do will be processes that reside in the "builder container" communicating with the "business app" running within the KIND container.

Now, let's get started. First, we need to create the image for the builder container, and the various containers it encapsulates. To maximize the portability of this overall process, we're going to "bootstrap a builder image". This is effectively building a Docker image (that includes, at a minimum, a copy of Docker itself along with some build tools) which we build from either a bare metal or *VM*-based host *OS*. When the image is built, it is now capable of building new releases of itself from the same build scripts we originally used on the original host that built it in the first place. Why go to such hassle?

Well, we can create a self-sustaining build environment that's completely containerized (after the initial bootstrapping phase), and re-use this builder image to produce the other various containers in our final product (i.e. it can build the "app tester container", etc.) in addition to new versions/releases of the builder image itself. In the end, **all** of our infrastructure, including the build-time and runtime images, are completely encapsulated in containerization technology,

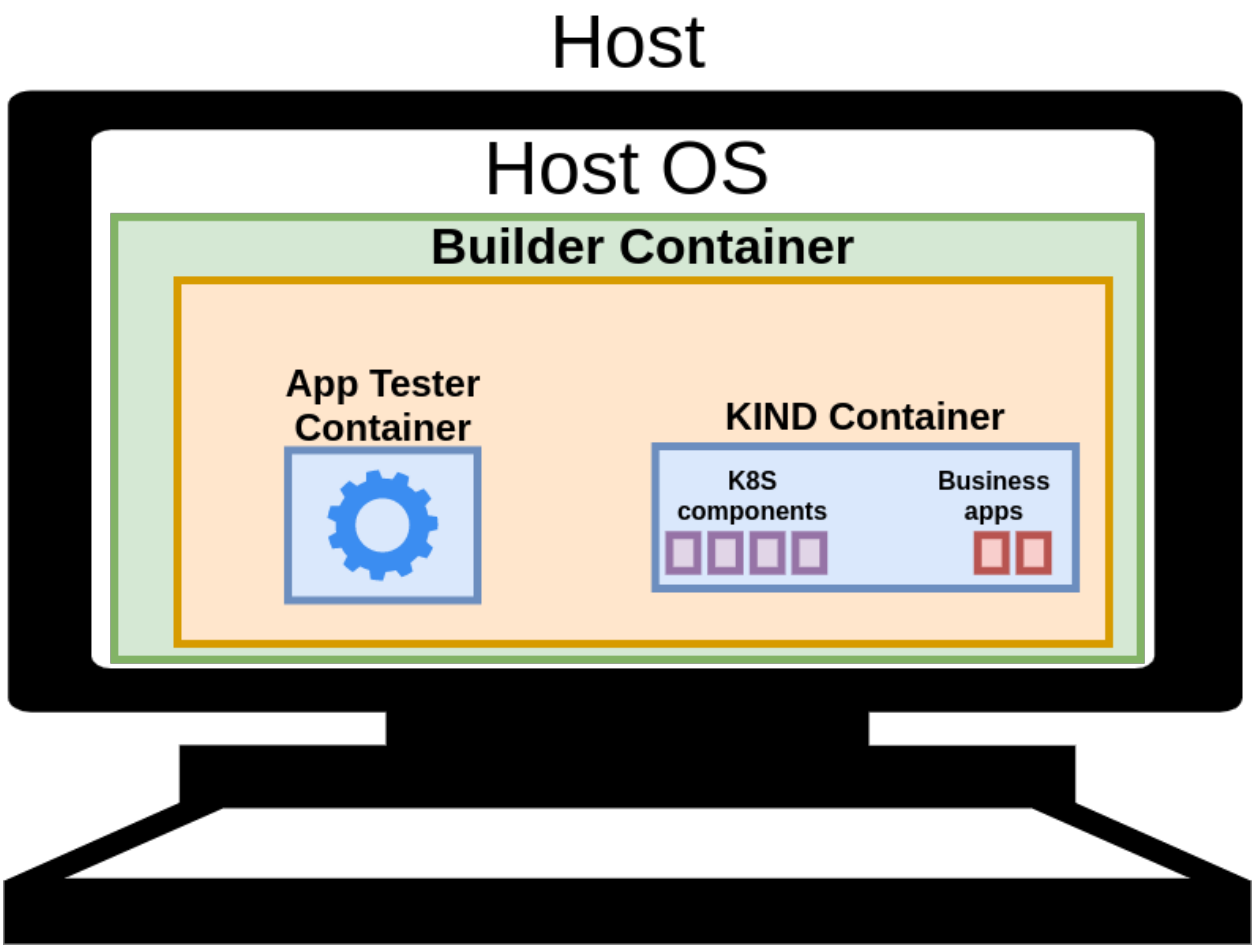

Fig. 4.8: Conceptual model of a Docker-in-Docker configuration that encapsulates a build system, and KIND-based system for temporary/ephemeral cluster generation.

increasing overall scalability and re-usability significantly. Now, let's get the scripts in order for the initial "bare metal build" of our "pre-bootstrap builder image".

Listing 4.37: /code/Dockerfile.bootstrap_builder.pre

```
FROM ubuntu:focal as baseline

# System packages.
RUN apt update -y && \
    apt install -y \
        binutils \
        build-essential \
        coreutils \
        curl \
        docker.io \
        gcc \
        git \
        iproute2 \
        iputils-ping \
        jq \
        lsb-release \
        make \
        net-tools\
        python3 \
        python3-dev \
        python3-pip \
        uuid-runtime \
        vim \
        wget \
    && \
    apt clean -y
```

Listing 4.38: /code/Dockerfile.bootstrap_builder.pre.dockerignore

```
# Ignore everything by default. We can manually add individual files as we
# need them to this permit-list.
*
```

Listing 4.39: /code/Makefile.bootstrap_builder.pre

```
# Defaults and global settings.
.DEFAULT_GOAL: all
BUILDER_TAG=docker_builder:0.0.0

# Default goal.
.PHONY: all
all: docker_builder
        @echo "Done build."

# Build Docker image.
.PHONY: docker_builder
docker_builder: Dockerfile.bootstrap_builder.pre Dockerfile.bootstrap_builder.pre.
↪dockerignore
```

(continues on next page)

(continued from previous page)

```
	DOCKER_BUILDKIT=1 docker build \
				-t "$(BUILDER_TAG)" \
				--target baseline \
				-f Dockerfile.bootstrap_builder.pre \
				.

# Launch Docker container.
.PHONY: run_builder
run_builder:
	docker run \
		--rm -it \
		--volume="$(shell readlink -f .):/work:rw" \
		--workdir="/work" \
		--volume="/var/run/docker.sock:/var/run/docker.sock:rw" \
		$(BUILDER_TAG)
```

Now let's trigger the build of our "pre-bootstrap builder image".

Listing 4.40: Creating the pre-bootstrap version of our baseline builder image.

```
# Build the image.
owner@darkstar$> make -f Makefile.bootstrap_builder.pre
DOCKER_BUILDKIT=1 docker build \
			-t "docker_builder:0.0.0" \
			--target baseline \
			-f Dockerfile.bootstrap_builder.pre \
			.
[+] Building 40.6s (6/6) FINISHED
 => => naming to docker.io/library/docker_builder:0.0.0                                    ␣
↪                                                                                          ␣
↪                        0.0s
Done build.

# Let's confirm it's properly tagged.
owner@darkstar$> docker image ls | grep docker_builder.*'0.0.0'
docker_builder                   0.0.0     1c407d6c0b93   14 minutes ago 860MB

# Great.
```

Now we have a `0.0.0` release of our builder image. Let's duplicate the various build scripts (so we can compare the before-and-after), along with the top-level `iax.sh` we use to bootstrap the whole process. Note how the version of the builder image in `iax.sh` always trails/follows the version in in Makefile.builder (i.e. `0.0.0` in `iax.sh` versus `0.0.1` in `Makefile.builder`; we're using an old version of the builder image to create a newer one, and we'll need to periodically increment **both** of these values).

Listing 4.41: /code/Dockerfile.builder

```
FROM ubuntu:focal as baseline

# System packages.
RUN apt update -y && \
    apt install -y \
```

(continues on next page)

(continued from previous page)

```
        binutils \
        build-essential \
        coreutils \
        curl \
        docker.io \
        gcc \
        git \
        iproute2 \
        iputils-ping \
        jq \
        lsb-release \
        make \
        net-tools\
        python3 \
        python3-dev \
        python3-pip \
        uuid-runtime \
        vim \
        wget \
    && \
    apt clean -y

# kubectl - for administering K8S clusters.
ADD kubectl /usr/local/bin/

# helm - for K8S package management.
ADD helm /usr/local/bin/

# kind - for spinning-up throwaway/test K8S clusters on-demand.
ADD kind /usr/local/bin/
```

Listing 4.42: /code/Dockerfile.builder.dockerignore

```
# Ignore everything by default. We can manually add individual files as we
# need them to this permit-list.
*

# Build artifacts we want in the final image.
!kubectl
!helm
!kind
```

Listing 4.43: /code/Makefile.builder

```
# Default shell (need bash instead of sh for some inline scripts).
SHELL := /bin/bash

# Defaults and global settings.
.DEFAULT_GOAL: all
BUILDER_TAG=docker_builder:0.0.1

# Default goal.
```

(continues on next page)

(continued from previous page)

```
.PHONY: all
all: docker_builder
        @echo "Done build."

# Build Docker image.
.PHONY: docker_builder
docker_builder: Dockerfile.builder Dockerfile.builder.dockerignore kubectl kind helm
        DOCKER_BUILDKIT=1 docker build \
                        -t "$(BUILDER_TAG)" \
                        --target baseline \
                        -f Dockerfile.builder \
                        .

# Download and verify kubectl binary.
kubectl:
        # Purge old copy if it exists from previous (failed) run.
        rm -f ./kubectl

        # Pull file.
        curl -o $(@) -L https://dl.k8s.io/release/v1.21.0/bin/linux/amd64/kubectl

        # Validate checksum. Terminate build if it fails.
        EXPECTED_CHECKSUM=
↪"9f74f2fa7ee32ad07e1721172599224847031 0ca1988214518806b39b1dad9f0"; \
        CALCULATED_CHECKSUM=$$(sha256sum $(@) | cut -d ' ' -f1); \
        echo "Sums: $${EXPECTED_CHECKSUM} --> $${CALCULATED_CHECKSUM}"; \
        if [[ $${EXPECTED_CHECKSUM} == $${CALCULATED_CHECKSUM} ]]; then  \
          echo "Checksum matches."; \
          true; \
        else \
          echo "Checksum failure."; \
          false; \
        fi ;
        chmod +x $(@)

# Download and verify kind binary.
kind:
        # Purge old copy if it exists from previous (failed) run.
        rm -f ./kind

        # Pull file.
        curl -o $(@) -L https://github.com/kubernetes-sigs/kind/releases/download/v0.11.
↪1/kind-linux-amd64

        # Validate checksum. Terminate build if it fails.
        EXPECTED_CHECKSUM=
↪"949f81b3c30ca03a3d4effdecda04f100fa3edc07a28b19400f72ede7c5f0491"; \
        CALCULATED_CHECKSUM=$$(sha256sum $(@) | cut -d ' ' -f1); \
        echo "Sums: $${EXPECTED_CHECKSUM} --> $${CALCULATED_CHECKSUM}"; \
        if [[ $${EXPECTED_CHECKSUM} == $${CALCULATED_CHECKSUM} ]]; then  \
          echo "Checksum matches."; \
          true; \
```

(continues on next page)

(continued from previous page)

```
        else \
          echo "Checksum failure."; \
          false; \
        fi ;
        chmod +x $(@)

# Download and verify helm binary.
helm:
        # Purge old copy if it exists from previous (failed) run.
        rm -f ./helm
        rm -f ./helm.tar.gz
        rm -rf "./linux-amd64"

        # Pull file.
        curl -o helm.tar.gz -L https://get.helm.sh/helm-v3.6.3-linux-amd64.tar.gz > helm.
↪tar.gz

        # Validate checksum. Terminate build if it fails.
        EXPECTED_CHECKSUM=
↪"07c100849925623dc1913209cd1a30f0a9b80a5b4d6ff2153c609d11b043e262"; \
        CALCULATED_CHECKSUM=$$(sha256sum helm.tar.gz  | cut -d ' ' -f1); \
        echo "Sums: $${EXPECTED_CHECKSUM} --> $${CALCULATED_CHECKSUM}"; \
        if [[ $${EXPECTED_CHECKSUM} == $${CALCULATED_CHECKSUM} ]]; then  \
          echo "Checksum matches."; \
          true; \
        else \
          echo "Checksum failure."; \
          false; \
        fi ;

        # If the job hasn't failed at this point, checksum is good. Extract the
        # binary and delete the original archive.
        tar -zxvf helm.tar.gz
        mv ./linux-amd64/helm .
        chmod +x $(@)
        rm -f ./helm.tar.gz
        rm -rf "./linux-amd64"

# Launch Docker container.
.PHONY: run_builder
run_builder:
        docker run \
                --rm -it \
                --volume="$(shell readlink -f .):/work:rw" \
                --workdir="/work" \
                --volume="/var/run/docker.sock:/var/run/docker.sock:rw" \
                $(BUILDER_TAG)
```

Listing 4.44: /code/iax.sh

```
#!/bin/bash
################################################################################
```

(continues on next page)

(continued from previous page)

```
# @brief:       Docker-bootstrap script for building projects via a Docker
#               image.
# @author:      Matthew Giassa.
# @e-mail:      matthew@giassa.net
# @copyright:   Matthew Giassa (IAXES) 2017.
################################################################################

# Docker runtime image to use for building artifacts.
DOCKER_IMG="docker_builder:0.0.0"

# Need to dynamically create a set of "--group-add" statements while iteration
# across all groups for which we are a member, otherwise a simple
# "--user=$(id -u):$(id -g)" argument to "docker run" will only capture our
# primary group ID, and overlook our secondary group IDs (which means we won't
# be a member of the "docker" group when executing nested containers as a
# rootless user, causing headaches).
MY_GROUPS=("$(groups)")
MY_GROUPS_STR=""
for grp in ${MY_GROUPS[@]}; do
    if [[ "${grp}" == "$(whoami)" ]]; then
        continue
    else
        # Need to use GID, not group name.
        gid="$(echo $(getent group ${grp}) | awk -F ':' '{print $3}')"
        MY_GROUPS_STR+="--group-add ${gid} "
    fi
done

# Debug logging.
# echo "My groups: ${MY_GROUPS[@]}"     >&2
# echo "Group string: ${MY_GROUPS_STR}" >&2

# Launch docker container.
# * Setup PWD on host to mount to "/work" in the guest/container.
# * Forward access to SSH agent and host credentials (so container uses same
#   user/group permissions as current user that launches the script).
# * Make DIND possible (i.e. expose host Docker Unix domain socket to
#   guest/container).
# * Make the home directory be "/work" so that it's always writeable (allows us
#   to better handle rootless DIND and KIND).
# * Use host networking (non-ideal, but necessary for top-level containers to
#   access certain services like KIND-exposed kubeadm ports redirecting to port
#   6443 inside a pod without requiring a bridge being setup in advance).
docker run \
    --rm -it \
    --user="$(id -u):$(id -g)" \
    ${MY_GROUPS_STR} \
    --volume="/etc/passwd:/etc/passwd:ro" \
    --volume="/etc/group:/etc/group:ro" \
    --volume="$(readlink -f .):/work:rw" \
    --workdir="/work" \
    --volume="$(readlink -f ~/.ssh):$(readlink -f ~/.ssh):ro" \
```

(continues on next page)

(continued from previous page)

```
    --volume="$(dirname ${SSH_AUTH_SOCK}):$(dirname ${SSH_AUTH_SOCK})" \
    -e SSH_AUTH_SOCK=${SSH_AUTH_SOCK} \
    --volume="/var/run/docker.sock:/var/run/docker.sock:rw" \
    -e HOME="/work" \
    --network=host \
    ${DOCKER_IMG} \
    ${@}
```

Listing 4.45: Building our "post-bootstrap" builder image, with an older version of itself.

```
# Start the build.
owner@darkstar$> ./iax.sh make -f Makefile.builder
DOCKER_BUILDKIT=1 docker build \
                -t "docker_builder:0.0.1" \
                --target baseline \
                -f Dockerfile.builder \
                .
[+] Building 0.0s (6/6) FINISHED
...
...
...
Done build.

# Let's verify that it built.
owner@darkstar$> docker image ls | grep -i docker_builder
docker_builder                    0.0.0     8dc0244d19fa   About an hour ago   862MB
docker_builder                    0.0.1     1c407d6c0b93   2 hours ago         860MB

# Excellent: we still have our original "0.0.0" image we built via "bare
# metal building", and the new "0.0.1" image. From this point forward, we can
# produce new releases by modifying our "Dockerfile.builder" as needed, and
# then just incrementing the versions in "iax.sh" and "Makefile.builder".
```

Bravo! We have just bootstrapped a builder image that can create new releases/versions of itself going forward. If anyone else wants to contribute to this builder image, the only requirement is that they have Docker on their local/developer machine, rather than requiring extra tools on their machine (in the case of the initial "bare metal build" steps). Now, to build the actual "business app" we'll deploy into our *KIND* cluster). First, the `Dockerfile`, `dockerignore` file, and `Makefile`, along with a slightly modified version of `iax.sh` that uses the latest builder image release, `0.0.1` (normally I'd just bump the tag version in `iax.sh`, but I'm deliberately maintaining different copies of the files so they can be compared manually, and provided verbatim at a later time as examples for the reader).

Listing 4.46: /code/Dockerfile.app

```
FROM ubuntu:focal as baseline

# System packages.
RUN apt update -y && \
    apt install -y \
        python3 \
        python3-pip \
    && \
```

(continues on next page)

(continued from previous page)

```
    apt clean -y

# Main app.
ADD entrypoint.sh /

ENTRYPOINT ["/entrypoint.sh"]
```

Listing 4.47: /code/Dockerfile.app.dockerignore

```
# Ignore everything by default. We can manually add individual files as we
# need them to this permit-list.
*

!entrypoint.sh
```

Listing 4.48: /code/Makefile.app

```
#-------------------------------------------------------------------------------#
# Defaults and global settings.
.DEFAULT_GOAL: all
APP_TAG = docker_app:0.0.0

# Version of KIND container to use.
KIND_IMAGE = kindest/node:v1.21.
↪1@sha256:69860bda5563ac81e3c0057d654b5253219618a22ec3a346306239bba8cfa1a6

# Path to KIND config file.
KIND_CONFIG = config.yaml

# How long we'll wait for the cluster to be ready before aborting (seconds).
KIND_TIMEOUT = 60s

# Name of our cluster. Make this pseudo-random, or it's possible to (if we run
# multiple instances of this script concurrently) have two clusters with the
# same name, resulting in name collisions, both clusters breaking, and a lot of
# manual cleanup (and possibly restarting the Docker daemon).
# MUST match regex: `^[a-z0-9.-]+$`
CLUSTER_NAME := docker-app-$(shell uuidgen -r)
#-------------------------------------------------------------------------------#

#-------------------------------------------------------------------------------#
# Goals/recipes.
#-------------------------------------------------------------------------------#
# Default goal.
.PHONY: all
all: test_app_in_cluster
        @echo "Done build."

# Build Docker image.
.PHONY: docker_app
docker_app: docker-app.tar
docker-app.tar: Dockerfile.app Dockerfile.app.dockerignore
```

(continues on next page)

(continued from previous page)

```
        # Build.
        DOCKER_BUILDKIT=1 docker build \
                        -t "$(APP_TAG)" \
                        --target baseline \
                        -f Dockerfile.app\
                        .
        # Save to tarball.
        docker save "$(APP_TAG)" > $(@)

# Spin-up a KIND cluster and test our "business app" in it.
.PHONY: test_app_in_cluster
test_app_in_cluster: docker_app
        @echo "Creating test cluster: $(CLUSTER_NAME)"
        @echo "Using Kindest image: $(KIND_IMAGE)"

        # Purge stale kube config.
        rm -rf "/work/.kube"

        # Setup kubectl.
        export KUBE_EDITOR="vim"
        export KUBECONFIG="/work/.kube/config"

        # Pull kindest image (kind binary will already pull it, but may as well
        # be deliberate, in case we want to swap this out down the road with our
        # own private/cached/hardened copy).
        docker pull "$(KIND_IMAGE)"

        # Spin-up the cluster (rootless;
        # see https://kind.sigs.k8s.io/docs/user/rootless/).
        #export DOCKER_HOST=/var/run/docker.sock
        kind create cluster \
                --name "$(CLUSTER_NAME)" \
                --image="$(KIND_IMAGE)" \
                --config="$(KIND_CONFIG)" \
                --wait="$(KIND_TIMEOUT)"

        # Command to manually purge all clusters. Maybe we shoudl run this
        # inside a shell script so we can use an exit TRAP to guarantee we
        # clean-up when done (rather than bail out of the build script early on
        # the first failure and leave the system in an unknown state).
        # (IFS=$'\n'; for i in $(kind get clusters); do echo $i; kind delete clusters $i;␣
↪done)

        # Show clusters.
        kind get clusters

        # Get cluster info.
        kubectl cluster-info --context kind-$(CLUSTER_NAME)

        # Install an ingress controller (so we can curl/test our app).
        # TODO: cache a copy of the chart tarball and the container so we can
        # make this run offline and avoid extra bandwith consumption every time
```

(continues on next page)

(continued from previous page)

```
        # we would run this.
        kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/
↪master/deploy/static/provider/kind/deploy.yaml
        # Temporary hack: delete validation webhook until issue is resolved.
        # https://github.com/kubernetes/ingress-nginx/issues/5401
        kubectl delete -A ValidatingWebhookConfiguration ingress-nginx-admission

        # Load Docker images for business app(s) into cluster.
        kind load \
                image-archive docker-app.tar \
                --name "$(CLUSTER_NAME)"

        # Install Helm chart.
        helm install docker-app charts/docker-app

        # Wait for app(s) to be ready.
        # Arbitrary sleep (1 min), just so our app chart finishes deploying.
        sleep 60

        # Execute some basic tests via curl, kubectl, etc.
        kubectl get pods -A

        # Confirm we can "reach" our pod via the cluster's IP address, then the
        # ingress controller, then the service object, then the pod object, and
        # finally the application running in our container in our pod (phew,
        # lengthy).
        # 80 is the port our ingress controller is listening on, "/" maps to our
        # docker-app's ingress rule (should make it something more specific like
        # "/docker-app" in production), and we need to manually specify the host
        # name in the HTTP header (ingress-nginx doesn't allow wildcarding
        # hostnames at this timel please see
        # https://github.com/kubernetes/kubernetes/issues/41881).
        # Lastly "chart-example.local" is the hostname for the ingress rule
        # which we define in our chart's "values.yaml" file.
        curl --header "Host: chart-example.local" 127.0.0.1:80

        # Done: clean up.
        kind delete cluster \
                --name "$(CLUSTER_NAME)"

# Launch Docker container.
.PHONY: run_app
run_app:
        docker run \
                --rm -it \
                $(APP_TAG)
```

Listing 4.49: /code/iax_0.0.1.sh

```
#!/bin/bash
################################################################################
# @brief:       Docker-bootstrap script for building projects via a Docker
```

(continues on next page)

(continued from previous page)

```
#               image.
# @author:      Matthew Giassa.
# @e-mail:      matthew@giassa.net
# @copyright:   Matthew Giassa (IAXES) 2017.
################################################################################

# Docker runtime image to use for building artifacts.
DOCKER_IMG="docker_builder:0.0.1"

# Need to dynamically create a set of "--group-add" statements while iteration
# across all groups for which we are a member, otherwise a simple
# "--user=$(id -u):$(id -g)" argument to "docker run" will only capture our
# primary group ID, and overlook our secondary group IDs (which means we won't
# be a member of the "docker" group when executing nested containers as a
# rootless user, causing headaches).
MY_GROUPS=("$(groups)")
MY_GROUPS_STR=""
for grp in ${MY_GROUPS[@]}; do
    if [[ "${grp}" == "$(whoami)" ]]; then
        continue
    else
        # Need to use GID, not group name.
        gid="$(echo $(getent group ${grp}) | awk -F ':' '{print $3}')"
        MY_GROUPS_STR+="--group-add ${gid} "
    fi
done

# Debug logging.
# echo "My groups: ${MY_GROUPS[@]}"     >&2
# echo "Group string: ${MY_GROUPS_STR}" >&2

# Launch docker container.
# * Setup PWD on host to mount to "/work" in the guest/container.
# * Forward access to SSH agent and host credentials (so container uses same
#   user/group permissions as current user that launches the script).
# * Make DIND possible (i.e. expose host Docker Unix domain socket to
#   guest/container).
# * Make the home directory be "/work" so that it's always writeable (allows us
#   to better handle rootless DIND and KIND).
# * Use host networking (non-ideal, but necessary for top-level containers to
#   access certain services like KIND-exposed kubeadm ports redirecting to port
#   6443 inside a pod without requiring a bridge being setup in advance).
docker run \
    --rm -it \
    --user="$(id -u):$(id -g)" \
    ${MY_GROUPS_STR} \
    --volume="/etc/passwd:/etc/passwd:ro" \
    --volume="/etc/group:/etc/group:ro" \
    --volume="$(readlink -f .):/work:rw" \
    --workdir="/work" \
    --volume="$(readlink -f ~/.ssh):$(readlink -f ~/.ssh):ro" \
    --volume="$(dirname ${SSH_AUTH_SOCK}):$(dirname ${SSH_AUTH_SOCK})" \
```

(continues on next page)

(continued from previous page)

```
    -e SSH_AUTH_SOCK=${SSH_AUTH_SOCK} \
    --volume="/var/run/docker.sock:/var/run/docker.sock:rw" \
    -e HOME="/work" \
    --network=host \
    ${DOCKER_IMG} \
    ${@}
```

Listing 4.50: Difference between original and modified iax.sh launcher scripts.

```
--- /work/examples/build_patterns/kind/iax.sh
+++ /work/examples/build_patterns/kind/iax_0.0.1.sh
@@ -8,7 +8,7 @@
 ################################################################################

 # Docker runtime image to use for building artifacts.
-DOCKER_IMG="docker_builder:0.0.0"
+DOCKER_IMG="docker_builder:0.0.1"

 # Need to dynamically create a set of "--group-add" statements while iteration
 # across all groups for which we are a member, otherwise a simple
```

We'll also include the various additional configuration files and pieces of Helm chart used to allow us to install our "business app" container to a cluster (I just created it via `helm create docker-app` inside the builder container invoked by `iax_0.0.1.sh`). Rather than include all the (many) files verbatim, we'll settle with knowing that the skeleton chart was created with `helm v3.6.3`, and we'll include the (slightly) modified `values.yaml` file here for reference/completeness.

Note: Helm is a large topic all on its own, so tutorials regarding its use will not be covered in this book. Readers interested in learning more about this topic (highly recommended, by the way), are encouraged to learn more about it [28]. For now, just think of Helm charts as a means of packaging *K8S* applications for later deployment/installation to a cluster, similar to using `rpm` files on Fedora and RedHat based distros, `deb` files on Ubuntu and Debian distros, `dmg` files on Mac *OS*, or `setup.exe` binaries on Windows-based systems. In general, I tend to just create a skeleton chart via `helm create some-chart-name` and continue working from there. For more details on the `config.yaml` file, please see [29] [30].

Listing 4.51: /code/config.yaml, with ingress controller enabled.

```
# Minimal config file/YAML required by KIND.
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4

# So we can get a working ingress controller in KIND.
nodes:
- role: control-plane
  kubeadmConfigPatches:
  - |
    kind: InitConfiguration
    nodeRegistration:
      kubeletExtraArgs:
        node-labels: "ingress-ready=true"
```

(continues on next page)

(continued from previous page)

```
  # Changing the host ports may break "helm install" operations unless we make
  # additional changes to our overall configuration of KIND.
  extraPortMappings:
  - containerPort: 80
    hostPort: 80
    protocol: TCP
  - containerPort: 443
    hostPort: 443
    protocol: TCP
```

Listing 4.52: /code/charts/docker-app/values.yaml

```
# Default values for docker-app.
# This is a YAML-formatted file.
# Declare variables to be passed into your templates.

replicaCount: 1

image:
  repository: docker_app
  pullPolicy: IfNotPresent
  # Overrides the image tag whose default is the chart appVersion.
  tag: "0.0.0"

imagePullSecrets: []
nameOverride: ""
fullnameOverride: ""

serviceAccount:
  # Specifies whether a service account should be created
  create: true
  # Annotations to add to the service account
  annotations: {}
  # The name of the service account to use.
  # If not set and create is true, a name is generated using the fullname template
  name: ""

podAnnotations: {}

podSecurityContext: {}
  # fsGroup: 2000

securityContext: {}
  # capabilities:
  #   drop:
  #   - ALL
  # readOnlyRootFilesystem: true
  # runAsNonRoot: true
  # runAsUser: 1000

service:
  type: ClusterIP
```

(continues on next page)

(continued from previous page)

```
  port: 80

ingress:
  enabled: true
  className: ""
  annotations:
    kubernetes.io/ingress.class: nginx
    kubernetes.io/tls-acme: "true"
  hosts:
    - host: chart-example.local
      paths:
        - path: /
          pathType: ImplementationSpecific
  tls: []
  #  - secretName: chart-example-tls
  #    hosts:
  #      - chart-example.local

resources: {}
  # We usually recommend not to specify default resources and to leave this as a␣
↪conscious
  # choice for the user. This also increases chances charts run on environments with␣
↪little
  # resources, such as Minikube. If you do want to specify resources, uncomment the␣
↪following
  # lines, adjust them as necessary, and remove the curly braces after 'resources:'.
  # limits:
  #   cpu: 100m
  #   memory: 128Mi
  # requests:
  #   cpu: 100m
  #   memory: 128Mi

autoscaling:
  enabled: false
  minReplicas: 1
  maxReplicas: 100
  targetCPUUtilizationPercentage: 80
  # targetMemoryUtilizationPercentage: 80

nodeSelector: {}

tolerations: []

affinity: {}
```

For reference, the directory listing should look like the following:

Listing 4.53: Directory listing for example KIND project. Could use some cleaning up down the road, but for now, we leave all the files in one place for easily re-creating the examples in this section.

```
kind/
  charts/docker-app/
    charts/
    templates/
      tests/
        test-connection.yaml
      _helpers.tpl
      deployment.yaml
      hpa.yaml
      ingress.yaml
      NOTES.txt
      service.yaml
      serviceaccount.yaml
    .helmignore
    Chart.yaml
    values.yaml
  config.yaml
  Dockerfile.app
  Dockerfile.app.dockerignore
  Dockerfile.bootstrap_builder.pre
  Dockerfile.bootstrap_builder.pre.dockerignore
  Dockerfile.builder
  Dockerfile.builder.dockerignore
  entrypoint.sh
  iax.sh
  iax_0.0.1.sh
  Makefile.app
  Makefile.bootstrap_builder.pre
  Makefile.builder
```

Now, to use our latest-and-greatest builder image to build our "app", create a Docker image that encapsulates our app, generate a helm chart tarball that makes our containerized app "*K8S* ready", and finally deploy it to *KIND* cluster to make sure it installs and runs as expected in a cluster. Lots of verbose logging information is about to fly by.

Listing 4.54: Launching our KIND builder + tester.

```
owner@darkstar:~$ ./iax_0.0.1.sh make -f Makefile.app
...
...
...
... curl --header "Host: chart-example.local" 127.0.0.1:80/
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/html4/strict.dtd
↪">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<title>Directory listing for /</title>
</head>
<body>
```

(continues on next page)

(continued from previous page)

```
<h1>Directory listing for /</h1>
<hr>
<ul>
<li><a href="bin/">bin@</a></li>
<li><a href="boot/">boot/</a></li>
<li><a href="dev/">dev/</a></li>
<li><a href="entrypoint.sh">entrypoint.sh</a></li>
<li><a href="etc/">etc/</a></li>
<li><a href="home/">home/</a></li>
<li><a href="lib/">lib@</a></li>
<li><a href="lib32/">lib32@</a></li>
<li><a href="lib64/">lib64@</a></li>
<li><a href="libx32/">libx32@</a></li>
<li><a href="media/">media/</a></li>
<li><a href="mnt/">mnt/</a></li>
<li><a href="opt/">opt/</a></li>
<li><a href="proc/">proc/</a></li>
<li><a href="root/">root/</a></li>
<li><a href="run/">run/</a></li>
<li><a href="sbin/">sbin@</a></li>
<li><a href="srv/">srv/</a></li>
<li><a href="sys/">sys/</a></li>
<li><a href="tmp/">tmp/</a></li>
<li><a href="usr/">usr/</a></li>
<li><a href="var/">var/</a></li>
</ul>
<hr>
</body>
</html>
```

Success: we can bootstrap our own Docker builder image, use it to create containerized applications, wrap them in a Docker container, deploy them to a *K8S* (throwaway/ephemeral) cluster, test them, all with a single invocation of `./iax_0.0.1.sh make -f Makefile.app`. More importantly, the entire process, from building to running to deploying and testing, is completely containerized, so we can easily reproduce it in a *CI/CD* pipeline: fantastic!

Note: Build pattern viability metrics.

Repeatable: very repeatable, but sometimes errors can result in a large number of stale pods surviving the termination of the top-level tool (KIND) itself, which either requires manual intervention or carefully crafted launcher/cleanup scripts to ensure all resources are completely purged between runs. Also have to take care that all projects use psuedo-random pod names so there aren't namespace collisions across jobs running concurrently on the same piece of build infrastructure. In short, needs to be designed with multi-tenant capabilities in mind when deploying as part of shared infrastructure such as a *CI/CD* system.

Reliable: very reliable (when properly configured/designed, as per the notes above). May take a few rounds of integration attempts to "get it right".

Scalable: very scalable.

4.5 Best Practises & Security Concerns

We will conclude this chapter with a somewhat lengthy, but by no means exhaustive, collection of "best practises" one is advised to employ in the development of cloud native build patterns (as well as building applications with Docker in general). I have made every effort to keep this collection limited to "generally accepted as true" recommendations, and left out topics that may be construed as opinionated in nature.

It is worth mentioning that there are almost always exceptions to rules, best-practises, etc.; (earlier notes in the chapter mention the trade-offs of security versus functionality in using "Docker-in-Docker" via sharing the Docker socket). I would encourage readers to treat the following best-practises I describe as "very strong suggestions" as opposed to technical dogma. That being said, the authors cited below may have differing opinions on these topics, and I make no claim that my opinions necessarily reflect or otherwise align with theirs.

With this being said, collections of material (similar to what is presented in this section) may also be found in [31] [20] [21] [22] [23], to name a few examples.

4.5.1 Privileged Containers & Rootless Containers

An often conflated topic is the concept of a root-enabled Docker container versus a privileged Docker container. Let's explore this in greater detail.

A root-enabled Docker container is simply a container that is invoked with the *UID* (and associated permissions/access/etc.) of the root user (i.e. *UID* is zero). For example, if I were to launch a container via `sudo`, we'll see that the container is running as the root user.

Listing 4.55: Launching Docker as root.

```
owner@darkstar$> sudo docker run --rm -it ubuntu:focal
root@059d5d126fac:/# whoami
root

root@059d5d126fac:/# id -u
0
```

OK. Well, what if I launch a container without the use of the `sudo` command? After all, I added my own account to the `docker` group earlier in this chapter (i.e. via `sudo usermod -a -G docker $(whoami)`). Let's try this again.

Listing 4.56: Launching Docker as a non-root user that's a member of the docker group.

```
owner@darkstar$> docker run --rm -it ubuntu:focal
root@d17ad08b754e:/# whoami
root

root@d17ad08b754e:/# id -u
0
```

Still root. Even though our own user account is a non-root account, Docker (and the processes/containers it launches) are launched via the root user account (adding our account to the `docker` group merely grants us access to use Docker: it doesn't affect runtime permissions/access; [32]). To actually launch the container as a non-root user (less important on local development environments if the container comes from a trusted source, but very important if the container is being used in a production environment, especially if it's part of a user-facing or public deployment), one may either make use of the `--user` argument to `docker run` [33], or by use of the `USER` keyword in the Dockerfile used to build the Docker image (that is later used to invoke a container).

Now, the other topic of interest: privileged containers. Privileged containers are simply containers that have been invoked via `docker run` (or a similar *API* call) with the `--privileged` flag set. This flag grants the container access to all devices on the host *OS*, and makes it trivial to break out of the container (even more trivial than running a container as root). It should only be used for very specific cases where the container needs access to specific file systems or hardware, and all users of the system acknowledge in advance that the container is not at all secured, having full read/write access to the host system.

Now, with that out of the way: don't use either of these methods (root-enabled containers or privileged container) unless there is a **very** good reason for doing so, and all stakeholders are well aware of the potential consequences of using containers with these features enabled.

4.5.2 Using dockerignore for Security and Image Size Reduction

Earlier in this chapter, the reader may recall mention of the use of the `DOCKER_BUILDKIT` environment variable when executing `docker build` operations, along with the recommendation of the use of `.dockerignore` files (*Single Container with Docker*). It's an important-enough topic to merit a second mention, as it can help reduce the size of the final Docker image built, as well as preventing cases where privileged information (i.e. *SSH* keys, signing certificates, user credentials, etc.) could accidentally be bundled into the image as well (a terrible practise in general, and catastrophic if the image is published and made accessible to a broad audience; even more so if publicly released). Rather than reproduce a lengthy tutorial on the topic here, the reader is encouraged to review [13] [14] [15] [16].

4.5.3 Reducing Image Size and Number of Layers with apt

The reader may have noticed a common pattern in Dockerfiles throughout this chapter (example below).

Listing 4.57: Dockerfile that extends Ubuntu.

```
FROM ubuntu:focal as baseline

# System packages.
RUN apt update -y && \
    apt install -y \
        make \
        gcc \
    && \
    apt clean -y
```

Why cram so many commands (separated with the `&&` functionality our shell provides) into a single Dockerfile `RUN` statement? Why not do something more "clean" like so:

Listing 4.58: Dockerfile that extends Ubuntu.

```
FROM ubuntu:focal as baseline

# System packages.
RUN apt update -y
RUN apt install -y make
RUN apt install -y gcc
RUN apt clean -y
```

Well, there are a few reasons for this. For starters, each `RUN` statement effectively results in an additional layer being created in our overall Docker image [34]: something we generally try to minimize wherever possible. Additionally, Dockerfiles will often have (especially in the case of "builder" images used for compiling/building projects, like we're

covering in this chapter) several dozen packages (possibly hundreds, depending on how many additional dependencies `apt` pulls in). Therefore, the general practise is to have all the `apt`-supplied packages stuffed into a single layer, rather than attempting to have more granular control over it. Also, if we're installing numerous packages that have common/shared dependencies, a granular approach would be inconsistent at best when trying to force specific dependencies into a specific layer. So, at a minimum, our "optimal" Dockerfile should at least look a bit like so:

Listing 4.59: Dockerfile that extends Ubuntu.

```
FROM ubuntu:focal as baseline

# System packages.
RUN apt update -y
RUN apt install -y make \
                   gcc
RUN apt clean -y
```

We could always put `make` and `gcc` on the same line, but, at least in my own opinion, it's a lot more readable this way (especially when dozens of packages are listed). So, what about the `apt update` and `apt clean` steps? Why also cram those in to the same `RUN` statement as the `apt install` step? As mentioned earlier, every `RUN` statement will result in a new layer. Furthermore, if we add a file to our image in one stage of our Dockerfile, and remove the same file in a later stage, **the file is still in our overall Docker image!** This means that the image size is still adversely impacted by this file, but the file isn't (easily) accessibly. This also has security implications that are covered in the subsequent sub-section.

In the case of installing packages via `apt install`, by condensing these various commands into a single `RUN` statement, we're able to effectively get a list of the latest packages available, install them, and clean up the `apt` cache (and copy of the package listing) in a single step, avoiding a plethora of intermediate (and completely unneeded) files from being bundled into our overall image. Hence, we arrive at our original example again:

Listing 4.60: Dockerfile that extends Ubuntu.

```
FROM ubuntu:focal as baseline

# System packages.
RUN apt update -y && \
    apt install -y \
        make \
        gcc \
    && \
    apt clean -y
```

Current, official versions of `apt` automatically execute the equivalent of `apt clean` (i.e. cache isn't preserved after installation completes), so the `apt clean` step is less relevant on something current like Ubuntu 20.04 "Focal". However, for completeness, we'll present a couple of example Dockerfiles and demonstrate that there is indeed a difference in overall image size and the number of layers due to whether-or-not condensed `RUN` statements are used.

Listing 4.61: Dockerfile example: no "squashing".

```
FROM ubuntu:trusty as baseline
# System packages.
RUN apt-get update -y
RUN apt-get install -y make
RUN apt-get install -y gcc
RUN apt-get clean -y
```

Listing 4.62: Sample run: no-squashing Dockerfile example.

```
owner@darkstar$> docker build . -t my_docker_builder:local
Sending build context to Docker daemon  35.33kB
Step 1/5 : FROM ubuntu:trusty as baseline
 ---> 13b66b487594
Step 2/5 : RUN apt-get update -y
 ---> Using cache
 ---> e4173983516e
Step 3/5 : RUN apt-get install -y make
 ---> Using cache
 ---> 1ec3953ddcc5
Step 4/5 : RUN apt-get install -y gcc
 ---> Using cache
 ---> dc9f68dcdfd4
Step 5/5 : RUN apt-get clean -y
 ---> Using cache
 ---> 9fc945253a98
Successfully built 9fc945253a98
Successfully tagged my_docker_builder:local

owner@darkstar$> docker image ls | grep my_docker_builder.*local
my_docker_builder                 local     9fc945253a98   7 minutes ago    285MB

owner@darkstar$> echo "Number of layers: $(( $(docker history --no-trunc my_docker_
↪builder:local | wc -l) - 1 ))"
Number of layers: 9
```

Now let's repeat this experiment with "squashing" (i.e. condensing of commands passed to a `RUN` statement).

Listing 4.63: Dockerfile example: with "squashing".

```
FROM ubuntu:trusty as baseline
RUN apt update -y && \
    apt install -y \
        make \
        gcc \
    && \
    apt clean -y
```

Listing 4.64: Sample run: squashing Dockerfile example.

```
owner@darkstar$> docker build . -t my_docker_builder:local
Sending build context to Docker daemon  35.33kB
Step 1/2 : FROM ubuntu:trusty as baseline
---> 13b66b487594
Step 2/2 : RUN apt-get update -y &&     apt-get install -y         make
gcc     &&     apt-get clean -y
---> Using cache
---> 655083adb0ef
Successfully built 655083adb0ef
Successfully tagged my_docker_builder:local

```

(continues on next page)

(continued from previous page)

```
owner@darkstar$> docker image ls | grep my_docker_builder.*local
my_docker_builder                 local     655083adb0ef   9 minutes ago 282MB

PWD: ~/code/repos/iaxes/pronk8s-src/examples/build_patterns/dind_example
[09:44:37]: owner@darkstar$> echo "Number of layers: $(( $(docker history --no-trunc my_
↪docker_builder:local | wc -l) - 1 ))"
Number of layers: 6
```

So, we're able to reduce the image size and number of layers, and this will also play an important part in the following sub-section on credential leaks.

4.5.4 Credential Leaks - Don't Embed Credentials

Follow hot on the heels of the previous sub-section, we cover another reason to make use of `dockerignore` files and be cautious about what files get pulled in during `docker build` operations: credential leaks. This happens when privileged information (i.e. signing certificates, *SSH* keys, usernames/passwords, *API* keys, etc.; are accidentally made public or available to unauthorized parties). Sometimes this can be due to a plain text file containing credentials being created during automated *CI/CD* builds. Masked environment variables are preferable, as covered in *Continous Integration & Continuous Delivery*, as it avoids having to rely on things like `dockerignore` files being the last (or only) line of defense against bundling credentials into images, or having to zero-fill persistent storage devices out of concern mission-critical credentials are scattered, unencrypted on disk drives throughout your build infrastructure.

In any case, we'll cover a simple example where credentials are deliberately "baked in" to a Docker image, with the intent (by the hapless user) of only having them available "temporarily", due to them being needed for some intermediate build operation to run successfully.

Listing 4.65: Dockerfile for "silly secrets" example.

```
FROM ubuntu:focal as baseline

# Add our super secret file. Some intermediate part of the build needs to know
# this information to complete.
ADD super_secret.txt /

# System packages.
RUN apt update -y && \
    apt install -y \
        cowsay \
    && \
    apt clean -y

# Remove our secret so we're all safe and no secrets get leaked. Right?
RUN rm /super_secret.txt
```

Listing 4.66: Mock credentials for "silly secrets" example.

```
One, if by land, and two, if by sea.
```

Now, let's build this image, and see how "secure" it really is.

Listing 4.67: Silly secrets example: recovering "deleted" files from a Docker image.

```
# Build our image.
owner@darkstar$> docker build . -t silly_secrets:local
Sending build context to Docker daemon  3.072kB
Step 1/4 : FROM ubuntu:focal as baseline
 ---> 1318b700e415
Step 2/4 : ADD super_secret.txt /
 ---> 0df172df41b1
Step 3/4 : RUN apt update -y &&     apt install -y         cowsay     &&     apt clean -y
 ---> Running in 4b8d868e2866
...
...
Step 4/4 : RUN rm /super_secret.txt
 ---> Running in 9814d6c45b09
Removing intermediate container 9814d6c45b09
 ---> 092e3a86d509
Successfully built 092e3a86d509
Successfully tagged silly_secrets:local

# Image built. Let's run it and see if our secret file is visible.
owner@darkstar$> docker run --rm -it silly_secrets:local ls -a /
.   .dockerenv  boot  etc   lib    lib64   media  opt   root  sbin  sys  usr
..  bin         dev   home  lib32  libx32  mnt    proc  run   srv   tmp  var

# Hmm, looks like our file isn't in the final image. Let's dive deeper and
# inspect the individual layers.
owner@darkstar$> docker history --no-trunc silly_secrets:local
IMAGE                                                                 CREATED
      CREATED BY                                                                      SIZE      COMMENT
sha256:092e3a86d50915fb48b4278716574b46cc2514bae8eaa48d6a2051cf59fd1a9b   About a minute
ago   /bin/sh -c rm /super_secret.txt                                                  0B
sha256:db6b071129a483ca39a6b8e367ea7a2af63ff057acfcdc5a70e5bad8c00be768   About a minute
ago   /bin/sh -c apt update -y &&     apt install -y         cowsay     &&     apt
clean -y               75.6MB
sha256:0df172df41b168cd1ec523591c0b2e603210ced2f85a427247b093e87c377be8   About a minute
ago   /bin/sh -c #(nop) ADD
file:8401e4e245be4f0e09057e36a6ef99968758e9152fb5acbcb66fcadf5f2cc224 in /    38B
sha256:1318b700e415001198d1bf66d260b07f67ca8a552b61b0da02b3832c778f221b   12 days ago
      /bin/sh -c #(nop)  CMD ["bash"]                                                 0B
<missing>                                                             12 days ago
      /bin/sh -c #(nop) ADD
file:524e8d93ad65f08a0cb0d144268350186e36f508006b05b8faf2e1289499b59f in /    72.8MB

# Well: that looks interesting: some stage of the build is deleting a
# "secret" file? Let's try running the intermediate container (i.e. just
# before the deletion step).
owner@darkstar$> docker run --rm -it
db6b071129a483ca39a6b8e367ea7a2af63ff057acfcdc5a70e5bad8c00be768 ls -a /
```

(continues on next page)

(continued from previous page)

```
.           bin   etc   lib32   media  proc  sbin              sys  var
..          boot  home  lib64   mnt    root  srv               tmp
.dockerenv  dev   lib   libx32  opt    run   super_secret.txt  usr

# This bodes poorly for the developer.
owner@darkstar$> docker run --rm -it␣
↪db6b071129a483ca39a6b8e367ea7a2af63ff057acfcdc5a70e5bad8c00be768 cat /super_secret.txt
One, if by land, and two, if by sea.

# Oh dear: deleting the already-included file is just shy of useless in terms
# of security.
```

In summary, avoid having credential files anywhere near your build jobs, employ the use of masked credentials or more sophisticated techniques when supplying credentials/tokens/keys to automated build jobs, and always keep in mind that the concept of "deleting files from containers" is a misnomer.

4.5.5 Verify Downloaded Packages via Checksums

I've encountered numerous online examples (among the few times I won't provide a citation, out of courtesy primarily) where packages (some secure, legitimate, and safe to use; some not quite so much) are provided online in the form of a tarball or pre-compiled binary, and the author encourages the end-user to simply install it via something like so:

Listing 4.68: Example of how **not** to safely install software.

```
wget http://this***-site-is-safe-i-promise.xyz/totally_not_a_virus.tar.gz
tar -zxf totally_not_a_virus.tar.gz
make
sudo make install
```

Yeah: don't do this. Even if we trust the author, there's always the chance an attacker has potentially compromised some piece of the chain of communication between where that file is hosted, and us (man-in-the-middle attack, hosting provider has been compromised, supply chain attack results in malware being embedded in the tarball, etc.). If we're going to pull a file off the internet an embed it into our Docker image as part of an automated build, we should at least do a minimal security assessment of the payload (i.e. tarball) via checksum analysis.

Listing 4.69: Download Go binaries for x64 - the safe way.

```
# URL of the file we want to download.
PAYLOAD_URL="https://golang.org/dl/go1.16.7.linux-amd64.tar.gz"

# sha256sum (don't use older checksums like Md5) of the package, as provided
# by the maintainer. We accessed these value via HTTPS with no security
# warnings, we're reasonably certain the maintainer is reliable and has a
# secure infrastructure, etc.
PAYLOAD_SUM="7fe7a73f55ba3e2285da36f8b085e5c0159e9564ef5f63ee0ed6b818ade8ef04"

# Pull the payload.
wget "${PAYLOAD_URL}"

# Validate the checksums (of type SHA256).
# There are better ways to templatize, prettify, automated, etc.; this step.
# This is just to provide a minimal, readable example.
```

(continues on next page)

(continued from previous page)

```
DOWNLOADED_SUM="$(sha256sum go1.16.7.linux-amd64.tar.gz | cut -d ' ' -f1)"

# Compare, and proceed if the checksum we calculated matches what we
# expected.
echo "Expected checksum: ${PAYLOAD_SUM}"
echo "Received checksum: ${DOWNLOADED_SUM}"
if [[ "${PAYLOAD_SUM}" == "${DOWNLOADED_SUM}" ]]; then
    echo "Checksums match. Proceed with build job."
else
    echo "Checksum failure: aborting build job."
    exit 1
fi
```

This is an important concept to keep in mind: even if we completely trust the maintainer and provider of an external package, there are numerous venues for an attacker to take advantage of that trust and put a "mine in the pipeline" if care is not taken for all dependencies pulled into your images (e.g. supply-chain attacks in this particular case).

One final note: **don't run these checks in a `Dockerfile`, run them in a pre-build script or a `Makefile` first**. I've seen a number of examples online where such operations are executed in a `Dockerfile`, and it's just increasing the overall size of the image (and number of layers) drastically for no reason at all. Prepare all your artifacts in advance, add them to a permit-list in the form of a `.dockerignore` file, and **then** run your `docker build` operation against your `Dockerfile`. The performance metric gains alone will be their own reward.

4.5.6 Additional Reading

I would encourage the reader to start with [35] and [36], and from there, broaden one's research independently. The materials referenced in the aforementioned citations also contain a plethora of relevant security knowledge, and also merit the attention of the reader. From that point, I leave it to the reader to begin their own foray into security research on the topics of Docker, containers, and cloud native development.

Additionally, I encourage readers to visit the Reproducible Builds community [37], as there is considerable overlap (in my own opinion) between the material presented in this chapter, and the methods and goals presented by the aforementioned.

Lastly, I encourage those engaged in researching and learning about cloud native security to consider reaching out to the members of the *CNCF* Security Technical Advisory Group, "*TAG* Security" [38] via their Slack channel. (disclaimer: I volunteer with this group). I have found it to be a courteous and knowledgable community, and when initially joining the team's Slack channel years back, I was able to get help from community members in "knowing where to look" for answers to security-related technical questions.

CHAPTER

FIVE

ARCHITECTURE DESIGN PATTERNS

Warning: This section is not yet complete, but I'm aiming to complete all of them by end-of-year 2022. Check-in once in a while for updates.

5.1 General Structure

5.2 IPC

5.3 Storage

5.4 Logging

5.5 Sidecars

5.6 Debugging, Core Dumps, etc.

CHAPTER

SIX

TESTING STRATEGIES

Warning: This section is not yet complete, but I'm aiming to complete all of them by end-of-year 2022. Check-in once in a while for updates.

6.1 Pre-amble: Containers & Clusters

6.1.1 Unit Tests

6.1.2 API Tests

6.1.2.1 Internal APIs and Unit Testing

6.1.2.2 Public-Facing API Testing

6.1.3 Feature Tests

6.1.4 Regression Tests

6.1.5 Fuzz Tests

6.1.6 Negative Testing

6.1.7 Mocks and Simulated Tests

6.1.8 Chaos Testing: Break Everything to Improve your App

6.1.9 Testing in Production

6.1.9.1 The Historical Joke

6.1.9.2 The Reality of CICD and Cloud Environments

6.2 Regression Tests & Statistics

6.2.1 Quick Tests & Gate Keeping Code Reviews

6.2.2 Hypothetical Builds for Downstream Projects

6.2.3 Soak Tests: Finding Obscure Bugs

6.3 End-to-End Tests

6.4 Testing in Production: a Misnomer?

6.4.1 The Necessity of GitOps

6.4.2 Rolling Upgrades & Recovery

CHAPTER

SEVEN

DOCUMENTATION

Warning: This section is not yet complete, but I'm aiming to complete all of them by end-of-year 2022. Check-in once in a while for updates.

7.1 Overview: Keeping Documentation Current

7.2 Markup-based Approaches

7.2.1 Read the Docs: rST + Sphinx

7.2.2 Markdown

7.3 Diagrams

7.3.1 PlantUML/PUML

7.3.2 Mermaid/MMD

7.4 Graphical and WYSIWYG Editors

7.4.1 Bridging the Gap: GUIs for Markup Languages

7.4.2 Google docs

CHAPTER

EIGHT

SECURITY

Warning: This section is not yet complete, but I'm aiming to complete all of them by end-of-year 2022. Check-in once in a while for updates.

8.1 Container Scanning

8.2 Dependencies and Supply Chain Attacks

8.2.1 Poetry

8.2.2 Snyk

8.3 Common Vulnerabilities and Exposures (CVEs)

8.3.1 Reporting

8.3.2 Documenting

8.3.3 Automatic Testing

CHAPTER

NINE

PRACTICAL PROJECT: RASPBERRY PI DEVOPS CLUSTER

Warning: This section is not yet complete, but I'm aiming to complete all of them by end-of-year 2022. Check-in once in a while for updates.

9.1 Build Pattern Implementations: Language-specific Boilerplate Examples

This section draws on the material covered throughout the book to provide the reader with ready-to-use boilerplate examples (in at least a few programming languages) of projects that use the concepts discussed so far. Ideally, it provides a skeleton project that handles a lot of the basics (i.e. containerizing an application, providing a Helm chart so that it can be trivially deployed and scaled within a cluster, unit testing at the code and cluster levels of operation, etc.). This allows one who is not yet fully versed in various cloud native technologies to rapidly get a "hello world" example program running in a throwaway cluster, and be able to "learn by doing" rather than having to learn and mentally context-load numerous technologies first.

9.1.1 Python 3.x

9.1.2 Go/Golang

CHAPTER

TEN

CONCLUSION

CHAPTER

ELEVEN

REFERENCES

CHAPTER

TWELVE

GLOSSARY

Warning: This section is not yet complete, but I'm aiming to complete all of them by end-of-year 2022. Check-in once in a while for updates.

API Application Programming Interface.

AWS Amazon Web Services.

Azure Microsoft Azure (Cloud Computing Services).

CD Continuous Delivery.

CI Continuous Integration.

CI/CD Continuous Integration - Continuous Delivery.

CNCF Cloud Native Computing Foundation.

CPU Central Processing Unit.

CWD Current Working Directory.

DIND Docker-in-Docker.

EC2 Amazon Elastic Compute Cloud.

FOSS Free Open Source Software.

GCP Google Cloud Platform.

IAC Infrastructure as Code.

ID Identity.

IPC Inter-Process Communication.

IT Information Technology.

ITS Information Technology Services.

K8S Kubernetes.

KIND Kubernetes-in-Docker.

MAC Media Access Control.

OS Operating System.

PC Personal Computer.

PWD Present Working Directory.

QA Quality Assurance.

RAM Random Access Memory.

SSH Secure Shell.

TAG Technical Advisory Group.

TOC Technical Oversight Committee.

UDS Unix Domain Socket.

UID User Identity.

USB Universal Serial Bus.

VM Virtual Machine.

VMM Virtual Machine Monitor.

BIBLIOGRAPHY

[1] Shashank Mohan Jain. *Linux Containers and Virtualization*. Apress, India, 2020.

[2] Andrew Tanenbaum, Herbert Bos. *Modern Operating Systems, Fourth Edition*. Pearson, India, 2014.

[3] Intel Corporation. Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3a-part-1-manual.pdf, accessed 2021-07-24.

[4] Michael Kerrisk. namespaces(7) — Linux manual page. https://man7.org/linux/man-pages/man7/namespaces.7.html, accessed 2021-07-24.

[5] Zhang et al. A Comparative Study of Containers and Virtual Machines in Big Data Environment. https://arxiv.org/pdf/1807.01842.pdf, accessed 2021-07-24.

[6] Sumit Maheshwari, Saurabh Deochake, Ridip De, Anish Grover. Comparative Study of Virtual Machines and Containers for DevOps Developers. https://arxiv.org/pdf/1808.08192.pdf, accessed 2021-07-24.

[7] Roger Rogers and Susan Proietti Conti. Virtual Machines versus containers. https://www.ibm.com/downloads/cas/POANK8YE, accessed 2021-07-24.

[8] Tavis Ormandy and Julien Tinne, Google Inc. Virtualisation security and the Intel privilege model. https://www.cr0.org/paper/jt-to-virtualisation_security.pdf, accessed 2021-07-24.

[9] HashiCorp. Write, Plan, Apply: Terraform - by HashiCorp). https://www.terraform.io/, accessed 2021-08-07.

[10] Roger Rogers and Susan Proietti Conti. Vagrant by HashiCorp. https://www.vagrantup.com/, accessed 2021-07-24.

[11] Docker Inc. Install Docker Engine. https://docs.docker.com/engine/install/, accessed 2021-08-01.

[12] Docker Inc. Use the Docker command line. https://docs.docker.com/engine/reference/commandline/cli/, accessed 2021-08-01.

[13] Docker Inc. Build images with BuildKit. https://docs.docker.com/develop/develop-images/build_enhancements/, accessed 2021-08-03.

[14] Docker Inc. Dockerfile Reference. https://docs.docker.com/engine/reference/builder/, accessed 2021-08-03.

[15] Tristan Zajonc, The Docker Maintainers. Add support for specifying .dockerignore file with -i/–ignore #12886. https://github.com/moby/moby/issues/12886#issuecomment-480575928, accessed 2021-08-03.

[16] Alexei Ledenev. Do not ignore .dockerignore (it's expensive and potentially dangerous). https://codefresh.io/docker-tutorial/not-ignore-dockerignore-2/, accessed 2021-08-03.

[17] Docker Inc. docker push. https://docs.docker.com/engine/reference/commandline/push/, accessed 2021-08-01.

[18] Docker Inc. Repositories. https://docs.docker.com/docker-hub/repos/, accessed 2021-08-01.

[19] Docker Inc. Docker in Docker! https://hub.docker.com/_/docker, accessed 2021-08-03.

[20] Matthew Close. Tutorial- Understanding the Security Risks of Running Docker Containers: Don't Lose Your Sock(et)s. https://www.ctl.io/developers/blog/post/tutorial-understanding-the-security-risks-of-running-docker-containers, accessed 2021-08-02.

[21] Peter Benjamin. Docker Security Best-Practises. https://dev.to/pbnj/docker-security-best-practices-45ih, accessed 2021-08-03.

[22] Srdjan Grubor. Top 3 Things to Avoid When Using Containers. https://www.conjur.org/blog/top-3-things-to-avoid-when-using-containers/, accessed 2021-08-03.

[23] OWASP Cheat Sheet Series Team. Docker Security Cheat Sheet. https://cheatsheetseries.owasp.org/cheatsheets/Docker_Security_Cheat_Sheet.html, accessed 2021-08-03.

[24] GitLab Inc. Run GitLab Runner in a container. https://docs.gitlab.com/runner/install/docker.html, accessed 2021-08-03.

[25] The Jenkins Project. Docker plugin for Jenkins. https://plugins.jenkins.io/docker-plugin/, accessed 2021-08-03.

[26] The tmux Maintainers. Welcome to tmux! https://github.com/tmux/tmux/wiki, accessed 2021-08-01.

[27] The Kubernetes Authors. Kubernetes Components. https://kubernetes.io/docs/concepts/overview/components/, accessed 2021-08-07.

[28] The Helm Authors. Helm - The Package Manager for Kubernetes. https://helm.sh/, accessed 2021-08-08.

[29] The Kubernetes Authors. kind - Quickstart Guide. https://kind.sigs.k8s.io/docs/user/quick-start/, accessed 2021-08-08.

[30] Kevin Sookocheff. Local Kubernetes Development with kind. https://sookocheff.com/post/kubernetes/local-kubernetes-development-with-kind/, accessed 2021-08-08.

[31] Docker Inc. Best Practises for Writing Dockerfiles. https://docs.docker.com/develop/develop-images/dockerfile_best-practices/, accessed 2021-08-08.

[32] Docker Inc. Post-installation steps for Linux. https://docs.docker.com/engine/install/linux-postinstall/, accessed 2021-08-03.

[33] Docker Inc. Run the Docker daemon as a non-root user (Rootless mode). https://docs.docker.com/engine/security/rootless/, accessed 2021-08-03.

[34] Docker Inc. About storage drivers. https://docs.docker.com/storage/storagedriver/, accessed 2021-08-08.

[35] Sven Vetsch et al., OWASP Foundation. OWASP Container Security Verification Standard. https://owasp.org/www-project-container-security-verification-standard/migrated_content, accessed 2021-08-08.

[36] Dirk Wetter et al., OWASP Foundation. OWASP Docker Top 10. https://owasp.org/www-project-docker-top-10/, accessed 2021-08-08.

[37] The Reproducible Builds Team. Reproducible Builds. https://reproducible-builds.org/, accessed 2021-08-23.

[38] Cloud Native Computing Foundation. Cloud Native Security). https://github.com/cncf/tag-security, accessed 2021-08-03.

[39] Matthew Giassa. Cloud Native DevOps: Migrating from Monoliths to Microservices. https://github.com/IAXES/pronk8s, accessed 2021-07-05.

[40] Michael Kerrisk. cgroups(7) — Linux manual page. https://man7.org/linux/man-pages/man7/cgroups.7.html, accessed 2021-07-24.

[41] Valeri Aurora. A brief history of union mounts. https://lwn.net/Articles/396020/, accessed 2021-07-24.

[42] Docker Inc. Dockerfile reference - USER. https://docs.docker.com/engine/reference/builder/#user, accessed 2021-08-03.

INDEX