LibrumCI: Leveraging Container Cluster Management Framework Natives in continuous integration

Anthony Troy
School of Computing
Dublin City University
Dublin 9, Ireland
anthony.troy3@mail.dcu.ie

ABSTRACT

Continuous integration practices seek to reduce software deployment lead time through ensuring that team members are frequently integrating work that is verified by automated builds and tests in production-like environments. Effective continuous integration pipelines must be capable of automatically provisioning such environments for testing upon team members contributing any infrastructural or environmental changes. Similarly, robust continuous integration infrastructures must be capable of such automated provisioning to decrease build execution load. VM-based environments struggle to efficiently support such workflows due the challenges and limitations around the automated provisioning and configuration management of virtual machines.

We forward that leveraging container-based virtualisation can strongly reduce continuous these integration impedances through the lightweight and portable affordances of containers. We demonstrate that the alternative model of cluster management under containerisation offsets continuous integration bottlenecks by supporting thousands of concurrent build executions. Upon evaluating the application of fully-containerised CI workflows under existing CI systems we consider the successes and short comings of these approaches in the design of a purpose-built container-native CI system. Subsequently, we contribute LibrumCI, a first-effort continuous integration system designed to support expansive integration loads of container-based projects.

Keywords

continuous integration; Kubernetes; Jenkins; Docker; Cluster Management Frameworks; VCS; Agile

1. INTRODUCTION

With organisational structures diversifying to off-shore, insource and outsource development needs, software is increasingly being developed by fully- or partially-distributed teams. The contributions from team members must, at a minimum, be first validated under a test infrastructure that is automated and robust enough to allow for high levels of code churn. This development requirement is reflected in in the sustained and growing adoption of continuous integration (CI) practices in open-source and enterprise software projects (Duvall et al., 2007; Fitzgerald and Stol, 2014; Vasilescu et al., 2015). Traditionally most software projects have been characterised as having poor integration practices. More often than often not a siloed team will conduct all development activities, arguably over a lengthy period, and defer acceptance testing and code integration. Importantly, during this period one can assert that the software is in an inoperative state, as there has been no motivation to actually run the software and use it in a production-like environment.

Humble and Farley (2010) argue that many projects "schedule lengthy integration phases at the end of development to allow the development team time to get the branches merged and the application working so it can be acceptance-tested". More concerning is that when some projects arrive at this point "their software is not fit for purpose". These long-winded and challenging phases, commonly referred to as "integration hell", pose difficulties for the engineers working through the integration and too for the leads and managers estimating the project delivery. Applications of continuous integration practices have been found to effectively remove these phases at source (Vasilescu et al., 2015; Fitzgerald and Stol, 2014; Humble and Farley, 2010; Duvall et al., 2007). As a practice, CI seeks to reduce software deployment lead time by facilitating frequent code churn through verifying contributions with automated builds and tests on production-like environments. Humble and Farley (2010) advise that a team member integrate their work at least once daily, an effective undertaking of this process will mean that the software is always in a working state.

Effectively implementing a CI process first requires an extensive configuration management strategy and versioning approach wherein all sources and configurations are managed and tracked under version control. Thereafter, the processes of provisioning, environment configuration, building, testing and code integration can all be automated. Given that continuous integration environments involve teams sharing remote resources, workflows can easily be impeded as integration demand increases. Integrations can be similarly hampered as the complexity and configuration of the software changes. In practice many CI pipelines consists of two types execution environments: production-like (staging) environments for acceptance and system testing; and minimal environments for unit and functional testing.

To not hinder CI workflows staging environments need to be automatically reprovisioned and reconfigured to respect any environmental changes made by team members. Doing so using VM-based virtualisation is generally quite cumbersome and expensive (Gambi et al., 2015). These constraints also proceed effectively supporting elastic continuous integration environments. We argue that leveraging container-based virtualisation can strongly reduce continuous integration impedances around environment provisioning and configuration through the lightweight and portable affordances of containers. Additionally, we demonstrate that the alternative model of cluster management under containerisation offsets continuous integration bottlenecks by supporting up to thousands of concurrent build executions. Upon evaluating the application of fully-containerised CI workflows under existing CI systems we consider the successes and short comings of these approaches in the design of a purpose-built container-native CI system. Subsequently, we contribute LibrumCI, a first-effort continuous integration system designed to support expansive integration loads of container-based projects.

2. RELATED WORK

Early studies around container-based virtualisation techniques proceed with the proposal of resource containers (Banga et al., 1999) and security jails (Kamp and Watson, 2000). With Linux V-Server becoming the first notable system providing container virtualisation at operating system level, Soltesz et al. (2007) provides a comprehensive comparative study on the qualitative benefits of containers. Some years later came research involving Docker-based containers and their inherent capability of effectively reproducing research environments (Boettiger, 2015), which is particularly identical to software provisioning and configuring management. Later performance-specific comparative studies were also conducted using Docker (Agarwal et al., 2015).

Subsequent works around containerised clouds note the facets of clustering and managing distributed containers (Pahl and Lee, 2015; Verma et al., 2015), container cluster management frameworks are also highlighted. Expectedly, over the last two years many books targeted at practitioners have also been published, highlighting the nuances and benefits of containerised cluster management (Turnbull, 2015; Holla, 2015; Rensin, 2015; Brewer, 2015). No works yet exist around leveraging container-based virtualisation in continuous integration environments.

3. BACKGROUND

3.1 Continuous Integration

Practices resembling continuous integration could be said to stem as far back as the original lean manufacturing method. Nevertheless in software engineering the practice was codified by Beck (2000) as a core tenant of the Extreme Programming (XP) method. Following, CI was also included in the Agile Manifesto as an encouraged quality assurance practice. Amidst its wide adoption and popularity however, there is some variability in how teams are employing the practice, resulting is some debate as to exactly which integration activities one should follow (Ståhl and Bosch, 2014). Nevertheless, at its core, CI forwards several central principles and high-level activities.

3.1.1 CI Activities

Before implementing CI a process for a given project, all software source code and assets must be captured under a version control system (e.g. Git, SVN or Mercurial). Such a system manages file changes using a controlled access repository. Importantly, this provides the mechanism for developers to persist and maintain a mainline branch of the working software (e.g. the head or trunk branch), this is where any desired contributions are to be frequently integrated/merged (Vasilescu et al., 2015). Version control systems also support reverting back to previous revisions of the software, which is commonly availed of in CI.

Central to CI is the mechanism in which integrations are built, tested and fed back to the team. More often than not an automated CI system will manage these tasks. Today existing systems are available as either hosted services or standalone setups (Gousios et al., 2015), and there are a multitude of such offerings in both the open-source and commercial spaces. The most notable of which are Jenkins, TravisCI, CruiseControl, CircleCI and Bamboo. As we will later explore, these systems offer varying approaches to how one configures and defines a build pipeline, nevertheless on an integration build machine or cluster the following workflow generally occurs:

- the CI server will poll, or listen to, contributions made against the version control repository.
- upon receiving a contribution, the CI server will retrieve the source files and subsequently run build and tests scripts.
- the following build results are most commonly made available through a web interface and possibly email.
- the built application package, such as a WAR or EXE file, is also made available (Ståhl and Bosch, 2014).

Continuously integrating reduces software project risk through the preventative deciton of defects by in turn reducing quality assumptions, and by eliminating manual repetitive integration processes "time, costs and effort" can be saved (Vasilescu et al., 2015). CI also affords the software being in a deployable state at any time (Humble and Farley, 2010). Indeed, as a process CI is quite arguably beneficial, nevertheless challenges do exist around implementing a robust CI system and encompassing process.

3.1.2 CI Challenges

Employing an effective configuration management strategy is perhaps one of the most challenging aspects of CI. It involves ensuring that every application-level configuration needed per build phase (e.g. build, deploy and test) is captured and managed under one's version control system. The same is required for the configurations around one's entire production infrastructure, from the patch-level of an operating system (OS) to the configuration files of databases and load-balancers. Furthermore, all of the dependencies between artefacts should also be expressed. Humble and Farley (2010) define this as "the process by which all artefacts relevant to your project, and the relationships between them, are stored, retrieved, uniquely identified, and modified". Codifying the entire state of an infrastructure in version control is certainly not trivial for most projects, particularly for non-greenfield projects.

Humble and Farley (2010) argue that a successful configuration strategy requires a "holistic approach to managing all infrastructure". This involves the "desired state", or current configuration, of one's infrastructure being tracked under version control. A major benefit of keeping absolutely everything in version control is that source changes made at any level will trigger a CI build, which is in turn integrated, tested and deployable. That includes DBAs updating SQL scripts and operations members changing firewall rules. However, supporting this requires implementing automated infrastructure provisioning. For instance, bumping the OS version which serves the application software must trigger a CI build, but first this must cause the OS to be re-provisioned on the CI server and the rest of the downward dependencies to be rebuilt (e.g. OS libraries, middleware, application build script). Tools like Chef, Puppet and Ansible are established automation tools in this space however, as we will later discuss, we argue that such heavyweight solutions are less well placed in modern CI practices.

True for both hosted services and standalone setups, powerful servers or build farms usually underly a CI system to ensure that build-times are as a quick as possible (Campos et al., 2014). As a consequence, these shared remote resources "become a bottleneck when the number of developers increases" as more concurrent builds can occur (Gambi et al., 2015). Several practices have been adopted to alleviate these limitations. One addresses this horizontally through the "componentisation of larger applications", wherein a codebase is divided into well-defined parts that are developed and integrated separately. Thus, resulting in separate CI builds for each component (Humble and Farley, 2010). Other approaches emphasise on test selection. For instance, those tests related to the changes made in a given contribution are ran first ("pre-submit"), and if successful the integration is made, then the remaining tests are executed ("post-submit") and if a failure occurs the integration is rolled back (Elbaum et al., 2014).

3.1.3 CI in Practice

Industry studies have found that 57% of companies are employing CI on a daily or weekly basis[1], however there is a clear disparity in how those practitioners are actually implementing the process. Ståhl and Bosch (2014) have revealed that there are several disagreements in the existing research around how CI is actually practiced in industry.

CI has also become a well adopted practice in open-source projects, particularly for those projects (75%) hosted on GitHub. Those surveyed integrators have been found to use "few quality evaluation tools other than continuous integration" in their integration workflows (Gousios et al., 2015). In the case of GitHub hosted projects, TravisCI is currently the most popular tool (Yu et al., 2015). Similar to many other modern hosted CI solutions, a repository is primarily configured with the system through a repository-level YAML file that declares the entire build workflow (Santos and Hindle, 2016). This file includes the infrastructural configuration (e.g. host OS type, host OS version, middleware dependancies), the application-level configuration, and finally the build and test scripts to be ran. The following `.travis-ci.yml` file is quite similar to that used on the Ruby on Rails project:

```
# use a Linux image with Ruby pre-installed
os: linux
language: ruby
sudo: false

cache:
  bundler: true
  directories:
    - # between builds cache installed 'bundler'
    - # packages (gems) and any other directories
before_install:
  - # before starting services and add-ons fetch
  - # and configure other system and/or
  - # middleware dependancies
# ensure services are running and available
# over the network
services:
  - memcached
  - redis
  - rabbitmq
addons: # use beta services
  postgresql: ``9.4"
before_script:
  - # before running tests run some commands
  - # like (re)configuring the application
script:
  - ci/run_tests.rb # test script to run
  - ci/run_inter.rb # integration script to run
# use each Ruby versions against the test script
rvm:
- 2.2.5
- ruby-head
# run the test script again under each specified
# environment variable
env:
  matrix:
    - `SOME_FLAG=0'
    - `SOME_FLAG=1'
notifications:
  - # send builds statuses to IRC
```

Upon receiving a contribution, such a service will provision and build a virtual environment based on the repository's CI configuration file, cache those whitelisted directories for subsequent builds, then and run any specified scripts or commands (which are generally for testing and integrating). Most services offer a web interface wherein build statuses and be observed, and integrations with other tools for notifying build statuses. As you would expect, a build failure occurs when a provisioning-, test- or integration-related command terminates unsuccessfully.

In theory, for most commercial software projects it is impossible to employ a mature CI process under such tools. The execution environments offered by such services are generally limiting. For instance, TravisCI supports only three virtual environments for build execution: Ubuntu 12.04, OS X Mavericks and Ubuntu 14.04. CI under these constraints may be adequate for software projects that only rely on unit- and functional- level testing. However, as we will later explore, more complex "n-tierd" applications require build pipelines that include acceptance testing on an environment akin to production.

Standalone setup CI systems offer greater flexibility around build execution environments. Duvall et al. (2007) forwards several tools such as CruiseControl, Apache Continuum, IBM's UrbanCode Deploy and Draco.NET as potential standalone solutions. In Jenkins, also a popular open-source contender in this space, one can define build pipelines comprised of one or more phases, and each of which can be executed on a different execution environment (Smart, 2011). For instance,

[1]http://info.thoughtworks.com/rs/thoughtworks2/images/Continuous%20Delivery%20$_$%20A%20 $Maturity$%20$Assessment$%20$ModelFINAL.pdf$

one might manually provision and configure a minimal compute environment for unit- and functional-level testing and a more production-like one for acceptance testing. Having installed and configured Jenkins on those machines, one could then configure the given repository and define the pipeline via the Jenkins UI. Tools such as Jenkins can also be extended to leverage cloud features. For instance, the Amazon EC2 Plugin, Azure Slave Plugin allow one to dynamically provision compute resources in a distributed master/slave Jenkins setup (Smart, 2011).

3.2 Containerisation

Selecting appropriate automation tools generally comes down to what is "best fit for your environment and development process". Duvall et al. (2007) argues that this includes the functionality, reliability, longevity and usability of the given tool. All too often these evaluations "often transcend the practical and escalate into what sounds like a religious debate". As we will subsequently explore, recently established standards and technologies in containerisation afford significant opportunities to CI.

3.2.1 Establishing the Container Standard

Containerisation is a recently resurged computing paradigm that is having a significant impact on how applications are being built, shipped and ran. Along with being less resource intensive and more portable, containers simplify dependency management, application versioning and service scaling, as opposed to deploying applications or application components directly onto a host operating system. Docker, albeit a relatively young project, has successfully established a container standard.

Containers have a long history in computing though much of their recent popularity surround the recent developments of both LXC and the Docker platform. The former can be described as a container execution environment, or more formally, a Linux user space interface to access new kernel capabilities of achieving process isolation through namespaces and cgroups (Pahl and Lee, 2015). The latter is an open-source suite of tools managed by Docker Inc. which extends upon container technology such as LXC, in turn allowing containers to behave like "full-blown hosts in their own right" whereby containers have "strong isolation, their own network and storage stacks, as well as resource management capabilities to allow friendly co-existence of multiple containers on a host" (Turnbull, 2015).

Uncertainties around Docker's maturity and production-readiness have been expressed (Kereki, 2015; Powers, 2015; Merkel, 2014), however over the last two years the states of both Docker and the containerisation ecosystem continue to rapidly progress. Last year Docker has seen an unprecedented increase in development, adoption and community uptake (Merkel, 2014). Most notably was the introduction of customisable container execution environments. This means as opposed to LXC one can "take advantage of the numerous isolation tools available" such as "OpenVZ, systemd-nspawn, libvirt-sandbox, qemu/kvm, BSD Jails and Solaris Zones". Also included in this 0.9 release was the new built-in container execution driver "libcontainer", which replaced LXC as the default driver. Going forward on all platforms Docker can now execute kernel features such as "namespaces, control groups, capabilities, apparmor profiles, network interfaces and firewalling rules" predictably "without depending on LXC" as an external dependency (Hykes, 2014).

Interestingly, libcontainer itself was the first project to provide a standard interface for making containers and managing their lifecycle. Subsequently the Docker CEO announced the coming together of industry leaders and others in partnership with the Linux Foundation to form a "minimalist, non-profit, openly governed project" named The Open Container Initiative (OCI), with the purpose of defining "common specifications around container format and runtime" (Golub, 2015). Thereafter Docker donated its base container format and runtime, libcontainer, to be maintained by the OCI. Docker Engine and Rocket (by CoreOS) are both container runtimes which adhere to these standards.

3.2.2 Applicability of Containers in CI

In terms of the configuration and provisioning of execution environments, containers offer a much a more lightweight and portable alternative to that of the traditional VM-based approach. Both approaches are "virtualisation techniques, but solve different problems". A container-based ecosystem lends itself better towards the "packaging, delivering and orchestrating" of execution environments, having a more PaaS focus. Whereas VMs emphasise more on "hardware allocation and management", having a more IaaS foucs (Pahl and Lee, 2015). Consequently, both approaches bear "significant qualitative differences". Most notably, the "start-up time of a full container is 6x lower than a VM, and memory footprint is 11x lower than a VM" (Agarwal et al., 2015).

With respect to VM-based environments, the container image standard established and implemented by Docker affords further qualitative advantages, particularly around the area of configuring execution environments. For a given software application or component, Docker allows one codify most of all potential environmental configuration into one manifest or declaration, a `Dockerfile`. Apart from the hardware resource requirements of the application (cpu, memory and disk), herein one can capture all system- and application- level configurations. Thus, from the perspective of a CI system, a `Dockerfile` can act as an all encompassing build manifest. Consider that, for unit and functional testing only, we require a build execution environment for a NodeJS application running on Ubuntu Xenial, where "`runTests.sh *`" executes our test suites. Such a `Dockerfile` would be as follows:

```
# base from a clean Ubuntu Xenial image
FROM ubuntu:16.04
# set static build configurations
ENV NPM_CONFIG_LOGLEVEL info
ENV NODE_ENV test
# fetch and install build/runtime dependancies
RUN apt-get update
RUN apt-get install -y apt-utils curl git
RUN curl -sL deb.nodesource.com/setup_6.x \
  | bash -
RUN apt-get install -y nodejs
RUN rm -rf /var/lib/apt/lists/* \
  && apt-get purge -y --auto-remove apt-utils curl
# specify forwarded container ports to the host
EXPOSE 8080
# create, and cd into, a HOME directory
WORKDIR /app
# cp local sources into the image
COPY ./myLocalApp/ /app/
```

```
# cp local app configuration into the image
COPY ./config-test.conf /app/
# install application dependancies
RUN npm install
# specify entrypoint for running the container
ENTRYPOINT ["/app/runTests.sh"]
# specify default arguments for the entrypoint
CMD ["*"]
```

In terms of CI, perhaps one of the greatest benefits that is afforded here is Docker's distributed and opportunistic caching mechanism for builds. Firstly, it is important to note that a built container manifest, an *image*, is just a stateless filesystem that acts as a parameter to a container runtime. Thus, a *container* is an running instance of an *image*. An image begins with a replica of a root filesystem (defined in a `Dockerfile` with the `FROM` instruction). Following, through leveraging "layered and versioning copy-on-write filesystems (AuFS or btrfs)", the termination of all subsequently defined instructions create additional image *layers*, and references to each are stored for caching alongside their respective filesystem (Arndt et al., 2015). Thus, for example, if a host machine is building our example image for the first time, and has not previously built `ubuntu:16.04`, then all instructions in the file will be terminated and cached. If we are rebuilding the image, and have only modified the `config-test.conf` source, the respective `COPY` layer for that source with all subsequent layers become invalidated and then rebuilt.

4. EVALUATION AND DESIGN

We evaluate in detail some applications of containerisation with existing CI systems, and subsequently forward the design and implementation of our first-effort fully-containerised continuous integration system. To fully avail of the advantages of containerisation we argue that the entirety of CI systems should be capable running under container-based virtualisation. Hence, the system and its runtime components being containerised, including the individual executions of its CI builds. Thus, the specific scope of our evaluation is to measure the effectiveness of existing systems running containerised builds while being ran themselves as containers. As previously highlighted, CI system offerings are either generally either hosted services or standalone setups. With the exception of TravisCI and some other less notable systems, most hosted services are proprietary. TravisCI is foremost a hosted service but also supports standalone deployments. GoCD[2] is a somewhat lesser known system, but perhaps only because it is the recent predecessor of CruiseControl, that too supports standalone deployments. Thus, we consider both TravisCI, Jenkins and GoCD for our evaluations.

As previously highlighted, scalable and robust CI systems are predicated by their ability to facilitate high-levels of code churn. That is, being capable of handling increasing numbers of contributions while sustaining build throughput. A container-native CI system is the first steps toward this path. By running and managing the system itself and all of its generated build executions through containers we can utilise less resources. Unfortunately, both TravisCI and GoCD do not fully support all of their components being ran as containers. For TravisCI[3] and similar for GoCD, one or more Docker worker hosts can run all "tests/jobs in isolated containers", however it does require its master and related services being hosted on a VM-based machine. Fortunately, Extensions exist for running Jenkins under a multi-container setup[4], i.e. one master Jenkins container and n amount slave containers.

4.1 Fully-Containerised Single-Host Jenkins

Given a single Docker host we evaluate a fully-containerised Jenkins distributed master/slave Jenkins setup. That is, on one container-based host machine we run a container master container and one or more slave containers, which themselves run the execution environments for builds as separate containers. We expect the aforementioned qualitative container benefits from running the involved static containers (master/slave) in this set up, however as we will demonstrate there are several approaches and considerations to be made when creating and managing dynamic and ephemeral containers.

We first consider the approach of running an internal Docker daemon within each slave container. Running the master and slave containers is straightforward. Jenkins slave agents can be registered with a master using either Java Web Start (JNLP) or SSH. Given our Unix-based environment, we favour SSH and generated key-pair. This can be achieved by extending the base Jenkins master image with a custom Dockerfile or by manually generating the keys through a command on the running container. We can then define a custom slave image which installs the special purpose (Docker-in-)Docker Engine as follows:

```
# base from latest Jenkins-slave image
FROM jenkinsci/ssh-slave
# fetch the Docker(-in-Docker) Engine
RUN wget "https://raw.githubusercontent.com\
  /docker/docker/\
  3b5fac462d21ca164b3778647420016315289034\
  /hack/dind" -O /usr/local/bin/docker \
  && chmod +x /usr/local/bin/docker
# expose ssh port for master
EXPOSE 22
# capture the ssh public key shared with master
ENV JENKINS_SLAVE_SSH_PUBKEY ssh-rsa AAAABccc123
```

A typical container is not allowed to access any host devices, however as our slave containers run their own Docker daemons they will need to be permitted access. Such is achieved by running the slaves with a "privileged" flag. Given we run n amount of slave instances with this flag, we can then begin configuring repositories and pipelines via the Jenkins master UI. Notwithstanding the system only having the compute resources of a single machine, we can see in figure 1 that there are some inefficiencies to this nested containerisation approach.

The outer Docker daemon on the host machine will allow for layer-sharing between the master (jenkins-master) and the slave containers (jenkins-slave-docker-n). For instance, both images depend up open-JDK-7 so once the master image has fetched the layer it is cached for any subsequently built slaves. Importantly, the inner-daemon of slaves is not shared, and this is particularly inefficient if we wish to make

[2]http://martinfowler.com/articles/go-interview.html
[3]https://enterprise.travis-ci.com/docshost-machinesl

[4]https://www.docker.com/sites/default/files/UseCase/
RA$_CI\%20with\%20Docker_0 8.25.2015.pdf$

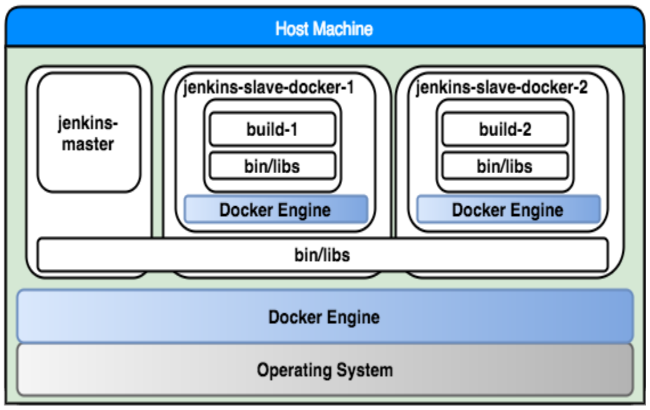

Figure 1: Single-host Distributed Jenkins with Nested Build Containers

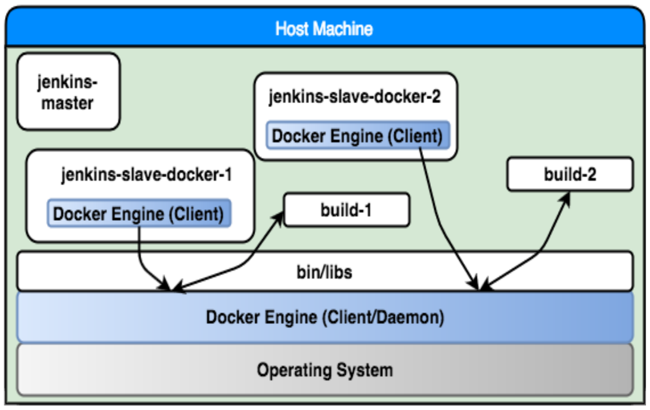

Figure 2: Single-host Distributed Jenkins with Sibling Build Containers

all slave containers ephemeral as their cache would be eliminated once removed (i.e. only provisioning slaves when required and removing them afterward).

Docker clients issuing *build* or *run* commands do so against the default daemon process which lives as a UNIX domain socket (*/var/run/docker.sock*). We can reduce these some of the shortcomings of our current approach by sharing this socket with the slave containers. The builds they create via the Docker client in turn become sibling containers that avail of the same cache. It is important to note that this is not a thorough solution, the daemon itself is a highly privileged process on the host machine. Sharing it with other processes has security implications. Notwithstanding this, in our slave image we instead install the standard Docker Engine to use the client API commands within CI builds.

```
                    ....
# fetch docker engine and don't run the daemon
RUN set -ex\
  && apt-get install apt-transport-https\
  ca-certificates\
  && apt-key adv --keyserver\
  hkp://p80.pool.sks-keyservers.net:80\
   --recv-keys\
  58118E89F3A912897C070ADBF76221572C52609D\
  && echo "deb https://apt.dockerproject.org/repo\
  debian-jessie main" > \
  /etc/apt/sources.list.d/docker.list \
  && apt-get update \
  && apt-get install docker-engine
                  ....
```

4.2 Fully-Containerised Multi-Host Jenkins

Compared to a VM-based approach, containers are inherently superior at scale. Containerisation is a lightweight virtualisation technique, best designed for the swift provisioning of execution units that are "short-lived and fragile". Thus, as opposed to being concerned with provisioning and configuration management, containers at scale instead require an emphasis on orchestration to "run efficiently and resiliently" (Rensin, 2015). Furthermore, containers running on multi-host environments have different scheduling concerns to that of VMs. Consider that multiple containers can run on one host, to best utilise resources new containers will need to be scheduled on the most appropriate host. We demonstrate this differing model of cluster management through evaluating fully-containerised multi-host Jenkins solutions.

Consider distributing our sibling-container Jenkins solution across multiple machines. In addition to the master node, we must provision n amount of slave-dedicated host machines. Jenkins supports executing only one concurrent job per slave agent (Smart, 2011). Thus, given that n is the number of slave container on a host, we afford for the system scheduling $n+1$ containers per host. Henceforth, dependant on compute capacity of each host we determine how many Jenkins slave containers we will run on each. Finally, we can configure the running slaves to discover the master and begin configuring repositories for CI via the UI.

Such a distributed solution affords build load to be spread reasonably effectively but utilises resources somewhat poorly while incurring notable bottlenecks. As a Docker daemon runs on a per-host basis, only slaves running on the same host will share the same build cache. Ideally any previously built layers should be cached for use across the entire cluster. Furthermore, the approach for orchestrating slave containers is manual and the mechanism for scheduling their respective CI build containers is static (as they will always be scheduled on the same host even if it is overcapacity).

Our sibling build containers already have their own orchestration instructions. Upon receiving a CI contribution a slave-agent will use its Docker client to *build*, *run* and then *remove* the respective test container. To utilise resources more effectively we would also like the slave-agent containers themselves to be orchestrated and ephemeral. A community plugin[5] exists for Jenkins which allows for slave containers to be orchestrated per CI contribution received. Upon the respective build job finishing it will then remove the that slave container. Notably, this approach lends itself somewhat better to the model of container cluster management, orchestrating containers over any ahead-of-time provisioning and configuration. Nevertheless, this plugin has

[5]https://wiki.jenkins-ci.org/display/JENKINS/Docker+Plugin

no scheduling capabilities. Such is unsurprising considering that viewing host resource consumption is a privileged action. Since the Jenkins system runs as containers it cannot look outward and introspect the host.

4.3 Cluster Management: Kubernetes

Container cluster management frameworks transparently govern the scheduling, orchestration and isolation of containers (Rensin, 2015). Such frameworks allow one "view a set of hosts as a unified programmable reliable cluster" (Goasguen, 2015), and generally all cluster nodes will use a shared Docker daemon. We continue our evaluation by considering how the integration of such tools might deliver a more optimal multi-host Jenkins CI system.

Practitioners and industry experts note that the cluster management tools supporting Docker vary greatly in terms of capability, architecture and target cluster proportion (Goasguen, 2015; Holla, 2015). Swarm is the native cluster management solution for Docker, and is comparatively lighter-weight than other tools in this space (Holla, 2015). For instance, by default Swarm is actually not distributed or highly available. Through configuring a supported discovery backend, such as Consul or ZooKeeper, Swarm nodes can then be added to the cluster and a fail-over master node can be then provisioned and configured.

Standalone Swarm may be a fitting clustering solution for some use-cases, however supporting cluster sizes from a hundred to thousands of nodes requires a more production-grade cluster platform such as Apache Mesos or Kubernetes. The former is foremost a "data center resource allocation system" with an interchangeable interface for accessing and scheduling cluster resources. The later is a higher-level platform specifically designed for managing containerised applications across multiple hosts including programmable mechanisms for container orchestration, scaling and monitoring (Goasguen, 2015). Thus, we discount Swarm and Mesos from our evaluation and consider distributed Jenkins CI in conjunction with Kubernetes.

Kubernetes (k8s) is a production-grade container management system that was originally built, and subsequently open-sourced, by Google. It builds upon a decade of lessons learned from the design and use of Borg, an internal tool that manages virtually all of Google's cluster workloads (Rensin, 2015). As illustrated in figure 3, we have implemented a distributed Jenkins CI system which supports full-containerised builds using our sibling build container approach. Before we evaluate the effectiveness of this CI system let us first detail the underlying Kubernetes cluster architecture and its primary components.

The k8s master node serves as the kernel of the cluster. Its runs administrative and regulatory services as containers such as ones for: cluster-user admin control (policy-controller); cluster network management (Flannel and kube-proxy); container scheduling (kube-scheduler); an API for cluster object manipulation (kube-apiserver); grouped container management (kube-controller-manager); and a cluster UI (Verma et al., 2015). Worker nodes must be provisioned with some agent-containers which communicate back to those service containers on the master node. The most notable of such required agents is the kubelet daemon, which is primarily responsible for responding "to commands from the master to create, destroy, and monitor the containers on that host".

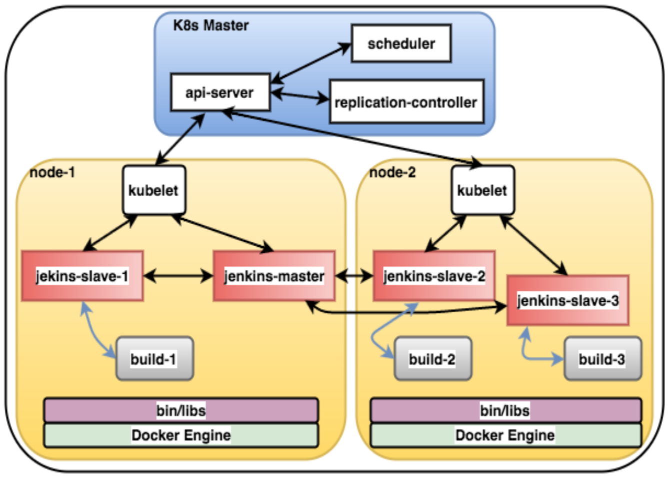

Figure 3: Distributed Jenkins under Kuberentes

Under a bare Docker setup, the expressions for inter-container relationships, container port-forwarding rules and container volume mounts are only definable as arguments as to the Docker *run* API. In k8s, Pods are the base mechanism for expressing these configurations and running one or containers. They act as a "resource envelope for one or more containers that are always scheduled onto the same machine and can share resources" (Verma et al., 2015). In terms of deploying long-running containers, pods may not be the appropriate k8s object of choice. Due possible changes in the overall cluster health "the master scheduler may choose to evict a pod from its host" and spin up a new instance on another node (Rensin, 2015). Higher-level k8s objects such as deployments and replication controllers should be for favoured when maintaining the state of long-running pods.

Revisiting our evaluation, we orchestrate and manage our *jenkins-master* pod via a deployment object. This allows us provide declarative updates to the running pod such as changing the version of Jenkins master image used or allocating more memory its container. Via the Jenkins UI we then configure the Jenkins k8s plugin[6] to use our custom slave image when dynamically creating *jenkis-slave* pods. This means that the respective build execution envoirnments for slave pods will be in the form of sibling-containers.

Upon receiving CI contributions Jenkins will request a *jenkis-slave* pod from the k8s API via HTTP. Thus, in conjunction kubelet with the k8s master will schedule that slave pod where most appropriate. With respect to our previously evaluated solutions this affords much greater resource utilisation as slave container orchestration and scheduling concerns are managed fully by k8s. Notwithstanding these gains this approach is inherently flawed. This is principally because k8s "schedules and orchestrates things at the pod level, not the container level" (Rensin, 2015). Our slave pods simply use the native Docker API to build and run their corresponding sibling-build on that host. Most importantly, this means that these containers are not actually scheduled or managed by k8s.

[6]https://wiki.jenkins-ci.org/display/JENKINS/Kubernetes+Plugin

4.4 LibrumCI

We forward the implementation of a first-effort CI system that effectively supports fully-containerised CI workflows through leveraging the native features of cluster management frameworks. Our evaluation of employing fully-containerised CI workflows with existing CI systems demonstrates the discordance in containerising applications originally architected for VM-based environments. Containerising software requires "thinking in terms of services instead of applications". Traditional CI systems such as Jenkins we not developed under these terms.

Consider the Jenkins master image, as a single UNIX process it is responsible for serving a front-end, user account management, system authentication, resource authorisation, build queuing, serving an API for resource manipulation and so on. Most importantly, it persists all of its state internally on disk (in */var/lib/jenkins* by default). As this process is stateful Jenkins master cannot be horizontally scaled. Furthermore, this monolithic design prohibits the scaling of individual segments of the application.

Additionally, our evaluations have revealed that effectively applying fully-containerised CI workflows in such tools through cluster management frameworks is challenging. Our approach of using Jenkins sibling-container-builds in k8s contravenes and circumvents the intended use of the framework in how one is to schedule containers and use node resources. To avail of the k8s orchestration and scheduling features all CI builds must be executed in pod form. One might note that k8s only supports building and running images that have been pushed to a registry. Thus, on a Jenkins slave we cannot dynamically create build pods based off the *Dockerfile* of a given CI contribution.

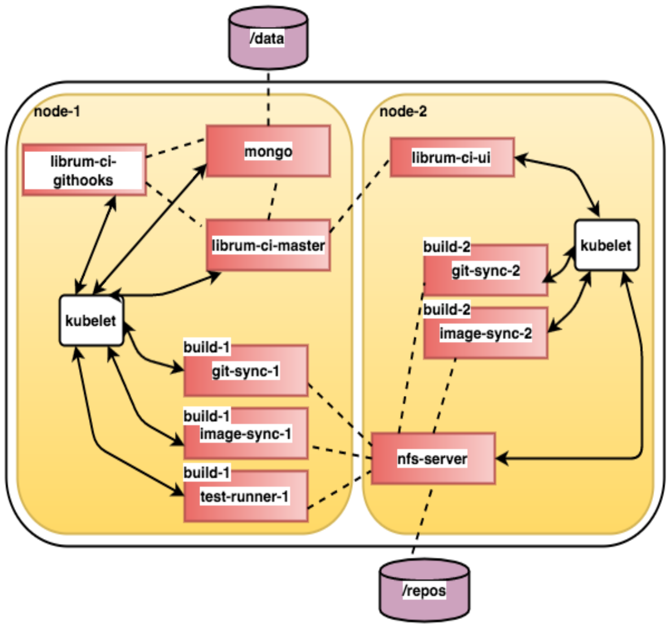

Figure 4: LibrumCI running concurrent CI builds

These observations have been applied to the design of LibrumCI. As illustrated in figure 4 we have adopted a micro-service architecture to allow for the scaling of individual components in the system. On a high-level the system is simply comprised three notable long-running services (Mongo, Githooks and Master) which coordinate together to dynamically schedule a build pipeline. A given pipeline can be comprised of up to three build-phases. Each of which occur sequentially in the form of an ephemeral, or "sidecar", pod. All of the system services are as follows:

- **Front-end web-service** is a single-page-application client of the LibrumCI Master API. It provides an interface for creating and configuring repositories, and for viewing repository build pipelines including their per build-phase log output.

- **Mongo database service** persists 'Repo', 'Branch' and 'Build' entity documents that are utilised by the Githooks and Master web-services.

- **Githooks web-service** is responsible for listening to Github push events (contributions). Upon receiving a contribution from a preconfigured repository, it will persist the contribution data (commit SHA, branch name, etc.), create a new build record and then request that the given build be scheduled via the LibrumCI Master API.

- **Master web-service** serves the RESTful API for resources and includes a schedule endpoint for build entities. This endpoint uses the k8s API to sequentially schedule the build phase sidecar pods. Using the k8s API each of these running pods will be streamed, where the phase status and logs are captured and persisted for the end user.

- **Network File System server** permits the same persistent volume (/repos/) being handed off between build phase pods, allowing for multiple simultaneous writers to that volume. On one external disk all configured repository files are name-spaced under this /repos/ volume. Thus, our NFS server allows build phases concurrently attach to the volume to synchronise and read repository files.

- **git-sync sidecars** synchronise the repository belonging to the incoming contribution by fetching it from the given commit SHA into persistent file storage (/repos/).

- **image-sync sidecars** synchronise the respective image of the incoming contribution by building the *Dockerfile* with Docker Engine, if this was successful the image (build artefact) is pushed to a registry. The registry address and password is preconfigured on the system-level.

- **test-runner sidecars** use the pushed image of the contribution to run the build tests. The command to run these tests is preconfigured on the repository-level.

5. CONCLUSIONS

Our evaluations of the applicability of containerisation with continuous integration systems reveals particular shortcomings in the supports of fully-containerised continuous integration. With the exception of commercial offerings, many CI tools do not yet meaningfully support containerised builds. This is particularly true when attempting to implement fully-containerised CI pipelines with standalone CI systems in distributed environment.

Due to the inherent differences between VM- and container-based virtualised, cluster management concerns shift from provisioning and configuration management to service orchestration and scheduling. This paradigm change becomes apparent when attempting to apply traditional clustering techniques to a fully-containerised CI system. In evaluating the integration of Jenkins with Kubernetes we have found challenges and limitations in running fully-containerised builds. To leverage container clustering frameworks such as Kubernetes we forwarded LibrumCI as part of this work. Unlike existing CI systems, which have been designed independently of containers and cluster frameworks, we explicitly architect LibrumCI to leverage the native capabilities of Kubernetes.

6. REFERENCES

P. M. Duvall, S. Matyas, and A. Glover, *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley Professional, 2007. [Online]. Available: http://www.amazon.com/Continuous-Integration-Improving-Software-Reducing/dp/0321336380%3FSubscriptionId%3D0JYN1NVW651KCA56C102%26tag%3Dtechkie-20%26linkCode%3Dxm2%26camp%3D2025%26creative%3D165953%26creativeASIN%3D0321336380

B. Fitzgerald and K.-J. Stol, "Continuous software engineering and beyond: Trends and challenges," in *Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering*, ser. RCoSE 2014. New York, NY, USA: ACM, 2014, pp. 1–9. [Online]. Available: http://doi.acm.org/10.1145/2593812.2593813

B. Vasilescu, Y. Yu, H. Wang, P. Devanbu, and V. Filkov, "Quality and productivity outcomes relating to continuous integration in github," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: ACM, 2015, pp. 805–816. [Online]. Available: http://doi.acm.org/10.1145/2786805.2786850

J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation (Addison-Wesley Signature Series (Fowler))*. Addison-Wesley Professional, 2010. [Online]. Available: http://www.amazon.com/Continuous-Delivery-Deployment-Automation-Addison-Wesley/dp/0321601912%3FSubscriptionId%3D0JYN1NVW651KCA56C102%26tag%3Dtechkie-20%26linkCode%3Dxm2%26camp%3D2025%26creative%3D165953%26creativeASIN%3D0321601912

A. Gambi, Z. Rostyslav, and S. Dustdar, "Improving cloud-based continuous integration environments," in *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ser. ICSE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 797–798. [Online]. Available: http://dl.acm.org.dcu.idm.oclc.org/citation.cfm?id=2819009.2819172

G. Banga, P. Druschel, and J. C. Mogul, "Resource containers: A new facility for resource management in server systems," in *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, ser. OSDI '99. Berkeley, CA, USA: USENIX Association, 1999, pp. 45–58. [Online]. Available: http://dl.acm.org/citation.cfm?id=296806.296810

P.-h. Kamp and R. N. M. Watson, "Jails: Confining the omnipotent root," in *In Proc. 2nd Intl. SANE Conference*, 2000.

S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson, "Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 3, pp. 275–287, Mar. 2007. [Online]. Available: http://doi.acm.org/10.1145/1272998.1273025

C. Boettiger, "An introduction to docker for reproducible research," *SIGOPS Oper. Syst. Rev.*, vol. 49, no. 1, pp. 71–79, Jan. 2015.

K. Agarwal, B. Jain, and D. E. Porter, "Containing the hype," in *Proceedings of the 6th Asia-Pacific Workshop on Systems*, ser. APSys '15. New York, NY, USA: ACM, 2015, pp. 8:1–8:9.

C. Pahl and B. Lee, "Containers and clusters for edge cloud architectures - a technology review," in *3rd International Conference on Future Internet of Things and Cloud)*, ser. FiCloud-2015. IEEE, 2015.

A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at google with borg," in *Proceedings of the Tenth European Conference on Computer Systems*, ser. EuroSys '15. ACM, 2015, pp. 18:1–18:17.

D. K. Rensin, *Kubernetes - Scheduling the Future at Cloud Scale*, 1005 Gravenstein Highway North Sebastopol, CA 95472, 2015. [Online]. Available: http://www.oreilly.com/webops-perf/free/kubernetes.csp

E. A. Brewer, "Kubernetes and the path to cloud native," in *Proceedings of the Sixth ACM Symposium on Cloud Computing*, ser. SoCC '15. ACM, 2015, pp. 167–167.

K. Beck, *Extreme Programming Explained: Embrace Change*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2000.

D. Ståhl and J. Bosch, "Modeling continuous integration practice differences in industry software development," *J. Syst. Softw.*, vol. 87, pp. 48–59, Jan. 2014. [Online]. Available: http://dx.doi.org.dcu.idm.oclc.org/10.1016/j.jss.2013.08.032

G. Gousios, A. Zaidman, M.-A. Storey, and A. van Deursen, "Work practices and challenges in pull-based development: The integrator's perspective," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 358–368. [Online]. Available: http://dl.acm.org.dcu.idm.oclc.org/citation.cfm?id=2818754.2818800

J. Campos, A. Arcuri, G. Fraser, and R. Abreu, "Continuous test generation: Enhancing continuous integration with automated test generation," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '14. New York, NY, USA: ACM, 2014, pp. 55–66. [Online]. Available: http://doi.acm.org/10.1145/2642937.2643002

S. Elbaum, G. Rothermel, and J. Penix, "Techniques for improving regression testing in continuous integration development environments," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: ACM, 2014, pp. 235–245. [Online]. Available: http://doi.acm.org/10.1145/2635868.2635910

Y. Yu, H. Wang, V. Filkov, P. Devanbu, and B. Vasilescu, "Wait for it: Determinants of pull request evaluation latency on github," in *Proceedings of the 12th Working Conference on Mining Software Repositories*, ser. MSR '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 367–371. [Online]. Available: http://dl.acm.org.dcu.idm.oclc.org/citation.cfm?id=2820518.2820564

E. A. Santos and A. Hindle, "Judging a commit by its cover: Correlating commit message entropy with build status on travis-ci," in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR '16. New York, NY, USA: ACM, 2016, pp. 504–507. [Online]. Available: http://doi.acm.org.dcu.idm.oclc.org/10.1145/2901739.2903493

J. Smart, *Jenkins: The Definitive Guide.* O'Reilly Media, 2011. [Online]. Available: http://www.amazon.com/Jenkins-Definitive-Guide-Smart/dp/1449305350%3FSubscriptionId%3D0JYN1NVW651KCA56C102%26tag%3Dtechkie-20%26linkCode%3Dxm2%26camp%3D2025%26creative%3D165953%26creativeASIN%3D1449305350

J. Turnbull, *The Docker Book: Containerization is the new virtualization*, 1st ed. Amazon Digital Services, Inc., 2015.

F. Kereki, "Concerning containers' connections: On docker networking," *Linux J.*, vol. 2015, no. 254, Jun. 2015.

S. Powers, "The open-source classroom: Doing stuff with docker," *Linux J.*, vol. 2015, no. 254, Jun. 2015.

D. Merkel, "Docker: Lightweight linux containers for consistent development and deployment," *Linux J.*, vol. 2014, no. 239, Mar. 2014.

S. Hykes, "Docker 0.9: Introducing execution drivers and libcontainer," http://blog.docker.com/2014/03/docker-0-9-introducing-execution-drivers-and-libcontainer/, 2014.

B. Golub, "Docker and broad industry coalition unite to create open container project," http://blog.docker.com/2015/06/open-container-project-foundation/, 2015.

N. Arndt, M. Ackermann, M. Brümmer, and T. Riechert, "Knowledge base shipping to the linked open data cloud," in *Proceedings of the 11th International Conference on Semantic Systems*, ser. SEMANTICS '15. New York, NY, USA: ACM, 2015, pp. 73–80.

S. Goasguen, *Docker Cookbook.* O' Reilly & Associates Inc, 2015.

S. Holla, *Orchestrating Docker : manage and deploy Docker services to containerize applications efficiently.* Birmingham: Packt Publishing, 2015.