Using the Autopilot pattern to deploy container resources at a WLCG Tier-2

Gareth Roy[1,*], *Emanuele* Simili[1,], *Samuel Cadellin* Skipsey[1,], *Gordon* Stewart[1,], and *David* Britton[1,]

[1]School of Physics and Astronomy, University of Glasgow, Kelvin Building, University Avenue, G12 8QQ, United Kingdom

Abstract. Containers are becoming ubiquitous within the WLCG, with CMS announcing a requirement for its sites to provide Singularity during 2018. The ubiquity of containers means it is now possible to reify the combination of an application and its configuration into a single easy-to-deploy unit, avoiding the need to make use of a myriad of configuration management tools such as Puppet, Ansible or Salt. This allows use to be made of industry-standard devops techniques within the operations domain, such as Continuous Integration (CI) and Continuous Deployment (CD), which can lead to faster upgrades and greater system security. One interesting technique is the Autopilot pattern, which provides mechanisms for application life-cycle management which are accessible from within the container itself. Using modern service discovery techniques, each container manages its own configuration, monitors its own health, and adapts to changing requirements through the use of event triggers. In this paper, we expand on previous work to create and deploy resources to a WLCG Tier-2 via containers, and investigate the viability of using the Autopilot pattern at a WLCG site to deploy and manage computational resources.

1 Introduction

Throughout industry, containers have rapidly become the method of choice to encapsulate complex software projects, offering the benefits of simplified deployment and repeatable application builds. Large LHC experiments such as CMS and ATLAS have also embraced containers for WLCG payloads; for example, CMS currently runs much of its production work in Singularity [18] containers, obtaining the images it requires via CVMFS [6]. Containers make it possible to reify configuration along with the applications being executed as a single easy-to-deploy unit, ensuring that all necessary dependencies are satisfied; however, as the number of containers which are deployed across a site increases, it becomes difficult to monitor, track and integrate these containers in an overarching system (a problem which is exacerbated if the containers are short-lived). Orchestration tools such as Kubernetes [14], Marathon [15] and Nomad [16] have been created to manage and track large-scale container deployments. Such tools offer benefits such as the automatic scaling and self-healing of containerised systems, but this comes at the cost of increased complexity for operation and maintenance.


[*]e-mail: gareth.roy@glasgow.ac.uk

A simpler alternative is to make each container responsible for the management of its own life-cycle, a technique which is sometimes referred to as the Autopilot pattern [17]. By adopting this approach, each container becomes responsible for configuring itself at start-up, and tearing down and tidying up on job completion; containers are also responsible for performing any necessary health checks, scaling their resource usage according to the active workload, and recovering from any failures which may occur. In this way, the containers are performing many of the tasks currently undertaken by the pilots found in experimental workload management frameworks, such as ATLAS's PandaPilot [10], LHCB's DIRAC [11] and CMS's CRAB3 [12], all of which use remote pilot jobs to interact with the workload management systems, monitor job payloads and report on errors.

The Autopilot pattern can then be coupled with modern service discovery techniques, which allow containers to be registered when they start, and permit remote monitoring and tracking of each container's state. This gives site administrators a dynamic picture of the overall state of the system, while providing them with the ability to investigate the behaviour of individual components in greater detail.

In this study, we apply the Autopilot pattern to containers running LHC experiment application payloads; these containers were developed in our previous work [1].

2 Container

2.1 Container Components
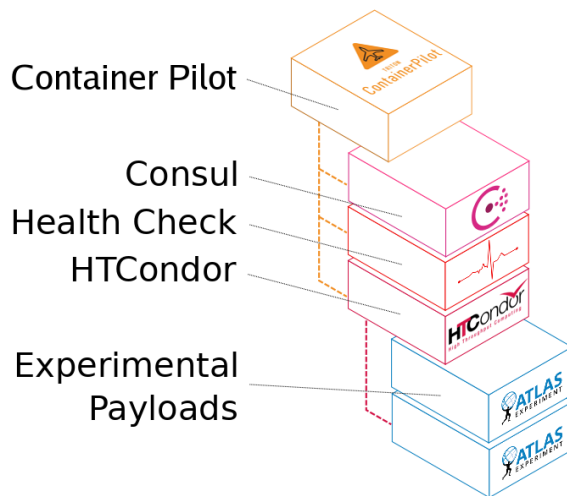
Figure 1. The components that make up our Containerised Worker Node.

In a previous study [1], a container was developed to execute WLCG application payloads. As deployed, the container relies on access to CVMFS [6], mounted from an external volume, to obtainall the necessary Grid middleware and WLCG experimental code. It includes the standard HEP_OSlibs packages to satisfy common software requirements shared by many HEP applications. Additionally, it contains all the configuration required by the local site (thereby reifying local configuration within the container), as well as the HTCondor

client packages which are required to connect to UKI-ScotGrid-Glasgow's [5] batch farm. The container is configured with standard pool accounts, and has HTCondor's CGROUPS [7] support disabled as all resource utilisation is restricted at the container level. HTCondor is also configured to use the Condor Connection Broker (CCB) to allow communication between the container and critical condor management processes, as the container's network access is otherwise restricted to the internal Docker network. The container itself is distributed to the hosts via a private Docker registry running on our headnode, with each node in our test pool spawning one container for every eight Hyper-Threaded CPU cores. The total size of this container is 851.9 MB, which is a similar order of magnitude to other LHC experiment containers, and it was based on CentOS 6 for backwards compatibility with existing workloads.

In the current implementation of our container worker node model HTCondor is treated as a long running service with each container connecting to the pool and running a number of experiment payloads (similiar to a standard worker node). It is hoped that in the future each container would be treated as ephemeral compute, and would have a lifespan only as long as a single payload. This would lead to a large number of relatively short-lived containers that need to be managed in a distributed fashion, and is why it is felt that the Autopilot pattern is an important system to consider.
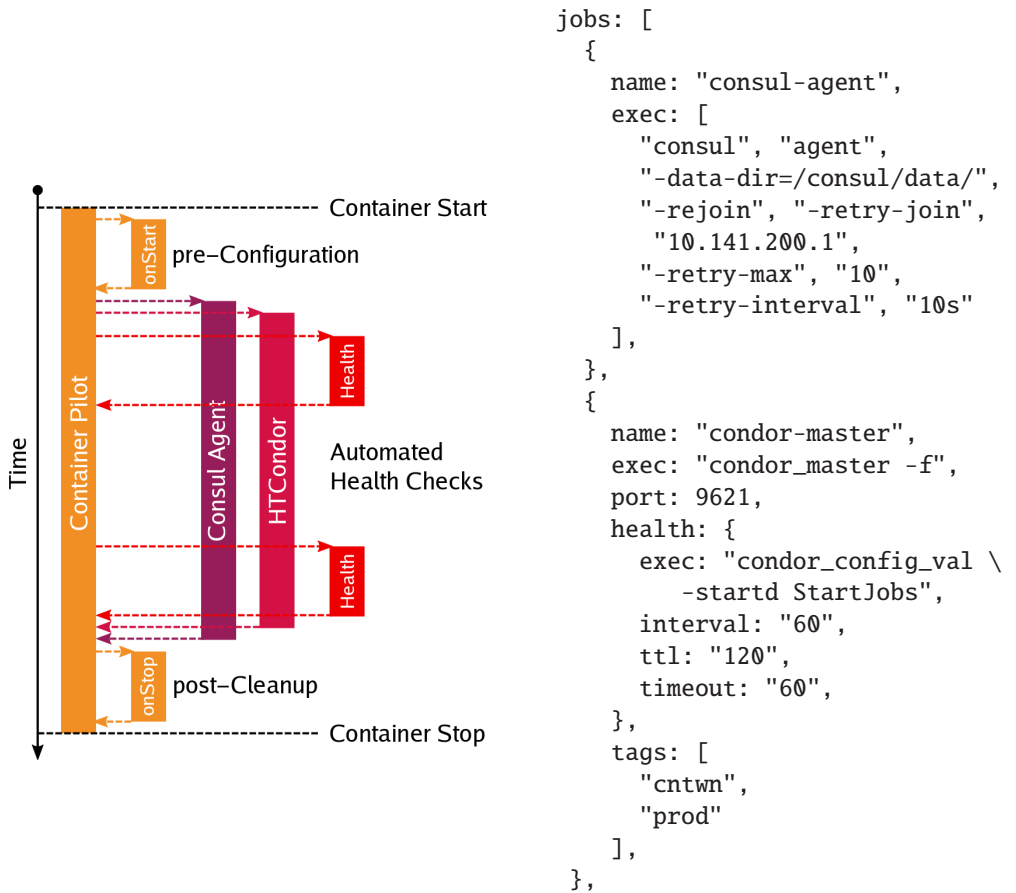
To enable the use of the Autopilot pattern in this work, the container described above has been modified to include two additional software components: ContainerPilot developed by Joyent Inc. [2], and Consul created by HashiCorp [3]. ContainerPilot is a software package that implements the Autopilot pattern; in effect, it runs as the `init` process of a container, and as such is able to manage the execution of all other processes within that container. It can monitor components by performing health checks and can respond to changing requirements by the use of event triggers at given points throughout its life-cycle. Consul is a distributed hierarchical key-value store, which has been developed to provide service discovery, and features built-in health checks to ensure that the software components for which it is responsible are available. Consul has been implemented using the Raft Consensus Algorithm [13], which provides a fault-tolerant mechanism for a group of servers to maintain consistency while sharing some agreed state.

A schematic of the components in our complete worker node container is shown in Figure 1.

2.2 Container Life-cycle

The life-cycle of a container using ContainerPilot is illustrated in Figure 2. When the worker node container described above is instantiated, ContainerPilot acts as the container entry point and starts to carry out any defined pre-configuration tasks (such as ensuring that CVMFS is mounted, and that the required middleware is available within the container). ContainerPilot then starts both the HTCondor master daemon (in the foreground using the `-f` flag to ensure that it can terminate the process again later) and the Consul agent. During the execution of the job's payload, it carries out periodic health checks at a user-specified interval. Once the job has finished, ContainerPilot can execute any post-shutdown tasks, such as pushing logging data to a central repository for auditing purposes.

The JSON shown in Figure 2 is used to configure ContainerPilot, defining the jobs that it will spawn on our container worker node. It contains two job stanzas: the first of these starts the Consul agent, and the second starts the HTCondor master daemon. As Consul has its own built-in health checking mechanism, a health check is only specified for the HTCondor job (in the `health` stanza). In this case, the health check verifies that StartJobs is set with a 60-second interval and a time-to-live of 120 seconds, after which a test will be deemed to

```
jobs: [
  {
    name: "consul-agent",
    exec: [
      "consul", "agent",
      "-data-dir=/consul/data/",
      "-rejoin", "-retry-join",
       "10.141.200.1",
      "-retry-max", "10",
      "-retry-interval", "10s"
    ],
  },
  {
    name: "condor-master",
    exec: "condor_master -f",
    port: 9621,
    health: {
      exec: "condor_config_val \
        -startd StartJobs",
      interval: "60",
      ttl: "120",
      timeout: "60",
    },
    tags: [
      "cntwn",
      "prod"
    ],
  },
```

**Figure 2.** The life-cycle of a container worker node, along with ContainerPilot's JSON configuration.

have failed and the job will trigger the *unhealthy* event. This particular parameter is set by HTCondor's own health checking system, which verifies that CVMFS is functioning correctly and that the daemons are communicating with the requisite remote services.

3 Service Discovery

3.1 Consul

The Consul agent has been included to allow each container to register itself with the overarching Consul cluster. A complete Consul cluster consists of a number of clients and servers. Within a cluster, the clients (in our case, these take the form of agents which are run within each container) are responsible for checking the service nodes and communicating with the Consul servers. A set of Consul agents are nominated as the servers; this set usually comprises an odd number of agents greater than three, in order to form a voting quorum. The servers are responsible for retaining the configuration information in a key-value store, and for ensuring consistency among themselves. As noted previously, consistency is maintained between servers by use of the Raft Consensus Algorithm, with leadership elections being carried out if a quorum member is lost.

Consul has been designed to provide service discovery both within and across geographically-distributed data centres, and so is a good choice for use within the varied environments provided by WLCG sites. It has been demonstrated to support sites at scales significantly larger than a typical WLCG Tier-2, ensuring that it will not become a bottleneck even in larger deployments.
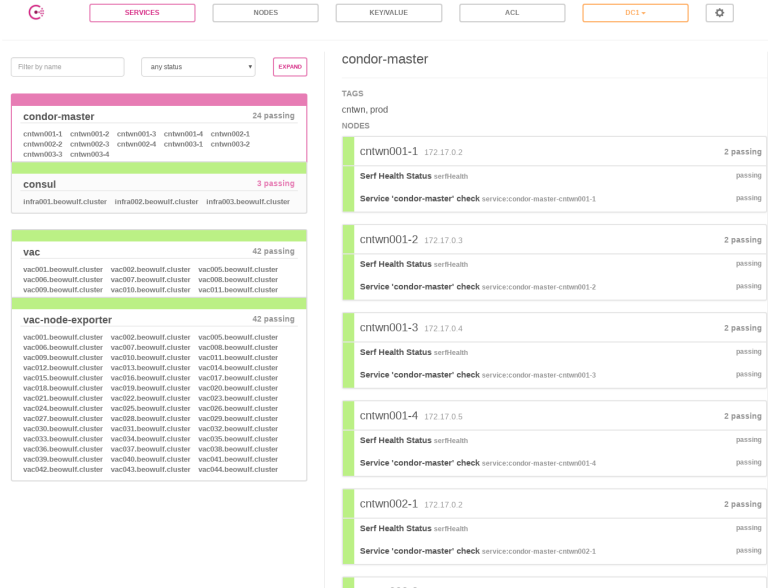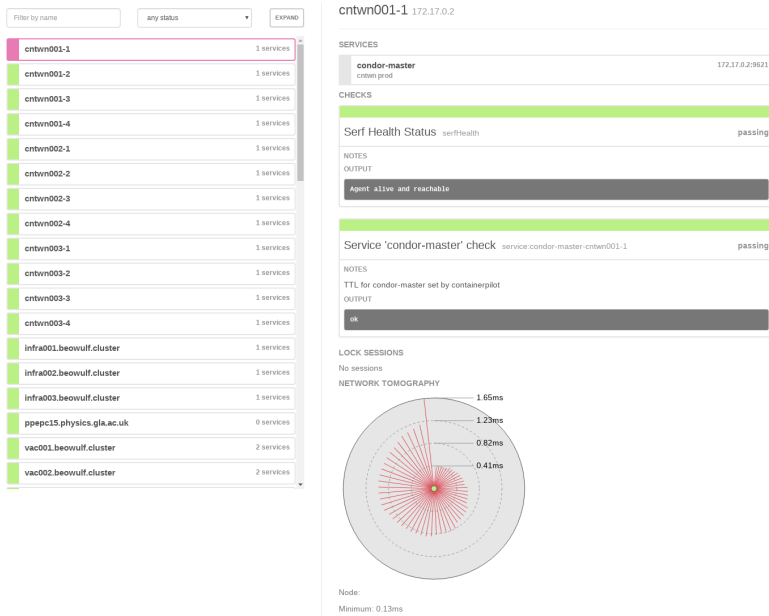

Figure 3. Service view of the Consul Web interface highlighting the registered condor-master services.

When each container worker node is instantiated, ContainerPilot starts a Consul client which registers both itself and the HTCondor master daemon as services in the Consul cluster. Figure 3 shows the report available through the Consul Web interface for the `condor-master` service. All the information presented in Figure 3 is also available through a RESTful API, which allows automated tools to query the Consul service and obtain a list of services with a particular description. Additionally, user-defined tags can be assigned to each service which can be used to further filter the service list (note the `tags` stanza in Figure 2). This approach allows each container to register itself dynamically, and offers the possibility for easy integration with other tooling and monitoring solutions (such as Prometheus [19]) which can be used to automatically build a picture of the current state of all registered container resources. At present service discovery is used to enumerate each container that is currently running and to obtain the status of healthchecks as described in the next section. In the future, when each container runs only a single application payload as described in the previous section, it is planned to obtain all service information from Consul so that when a container is instantiated it can dynamically configure itself.

3.2 Health and Network Latency

As well as simple service registration, Consul also monitors each of its clients to ensure that they are healthy and contactable. Figure 4 shows the report available through the Consul

Figure 4. Consul's view of a single worker node container, which shows the state of the health checks, and also a network tomography view.

Web interface for a single container worker node. The report lists each service provided by the container (in this case, the only service is the `condor-master` daemon, but in the future other critical services could be included), alongside the state of each health check. In this example, there is one health check for the Consul agent itself, along with the results of the test specified by ContainerPilot, which is detailed in the JSON job description shown in Figure 2.

Figure 4 also shows the network latency to all clients and servers in the Consul cluster, which is measured and illustrated in the network tomography graphic at the bottom. Consul was designed for use in large, distributed installations spanning multiple data centres, and so provides detail like this as a matter of course: this information is very useful when attempting to identify nodes with networking issues, or when attempting to map out potential problems within the cluster environment.

4 Autopilot System

4.1 System Overview

The worker node containers described in Section 2 have been deployed at UKI-ScotGrid-Glasgow to run ATLAS multi-core production work as part of the system shown in Figure 5. Containers were run on three physical hosts, with each container being allocated eight Hyper-Threaded CPU cores and 16 GB of RAM. Each container connected to the production HTCondor pool located at UKI-ScotGrid-Glasgow; the only difference between the container worker nodes and our regular HTCondor worker nodes was the fact that, on the former, HTCondor's CGROUPS support was disabled in favour of resource restrictions placed on the
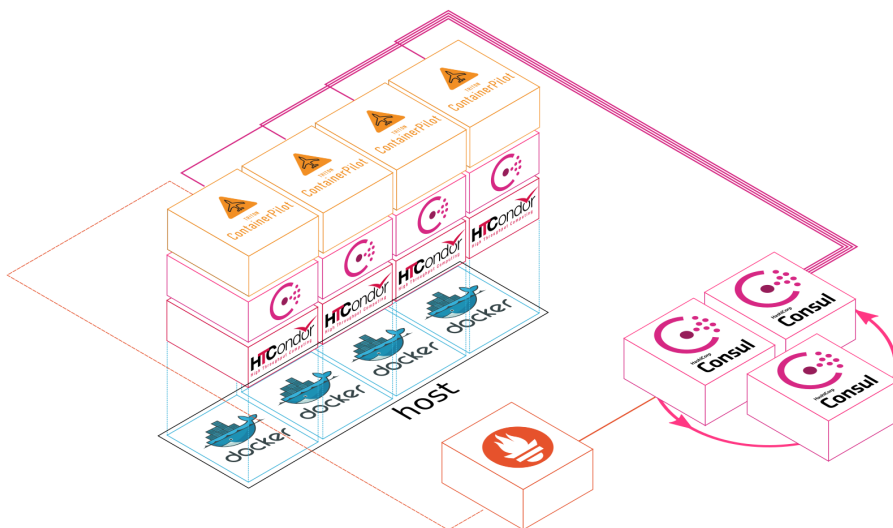
Figure 5. The components that make up our container worker node.

containers themselves (as discussed in Section 2). Each container registered itself with a Consul cluster comprising three Consul servers for resiliency, and this was also used to register other software components within the UKI-ScotGrid-Glasgow production site. Each host was monitored via a Prometheus endpoint, which scraped metrics from both the node and the container; at present, this information is statically assigned, but it is hoped that this will be dynamically provided by Consul in the future.

Using this configuration, we verified that the containers correctly connected to the Consul cluster when joining the HTCondor pool, and that ATLAS jobs were obtained and executed successfully. We also verified that the health of each container worker node was correctly reported, and conducted tests to ensure that problems injected into the system were identified satisfactorily and reported by both ContainerPilot and Consul. These activities confirmed that the Autopilot pattern provides a resilient and robust mechanism for running a large-scale container infrastructure, without the requirements of complicated orchestration systems such as Kubernetes. However, the Autopilot pattern itself can be used in conjunction with Kubernetes or other similar orchestration tools, should it be desirable to make use of their other capabilities in the future.

5 Conclusions

In this paper, we have investigated the use of the Autopilot pattern when deploying resources at a WLCG Tier-2 site. Using the ContainerPilot and Consul software packages, we have extended a container worker node to include automated system and health checks, and have also shown the potential for the inclusion of additional capabilities which might be useful in the future. We have explored the use of modern service discovery techniques to dynamically track container resources and to monitor automatic health checks, and have shown how this can be used to simplify the operation of a large, WLCG Tier-2 cluster.

References

References

[1] *G Roy et al.*, J. Phys.: Conf. Ser. **1085** 032026 (2018)

[2] *ContainerPilot* [online] Available https://www.joyent.com/containerpilot [accessed 25 October 2018]

[3] *Consul* [online] Available https://www.consul.io [accessed 25 October 2018]

[4] *Docker.io* [online] Available https://www.docker.io [accessed 14 May 2015]

[5] *ScotGrid* [online] Available at http://www.scotgrid.ac.uk/ [accessed 14 May 2015]

[6] *J Blomer et al.*, J. Phys.: Conf. Ser. **331** 042003 (2011)

[7] *Menage, Paul B.*, Proceedings of the Linux Symposium, **2** 45-57 (2007)

[8] *Blomer, Jakob, et al.*, J. Phys.: Conf. Ser. **513** 032009 (2014)

[9] *Merino, G.*, "Transition to a new CPU benchmarking unit for the WLCG." HEPIX Benchmarking WK (2009)

[10] *Nilsson, P et al.*, J. Phys.: Conf. Ser. **331** 062040 (2011)

[11] *Stagni, F et al.*, J. Phys.: Conf. Ser. **396** 2104 (2012)

[12] *Mascheroni, M et al.*, J. Phys.: Conf. Ser. **664** 062038 (2015)

[13] *Ongaro, Diego and Ousterhout, John*, Proc. USENIX Conf. USENIX Annu. Tech. Conf. 305-320 (2014)

[14] *Kubernetes* [online] Available https://kubernetes.io/ [accessed 25 October 2018]

[15] *Marathon* [online] Available https://mesosphere.github.io/marathon/ [accessed 25 October 2018]

[16] *Nomad* [online] Available https://www.nomadproject.io/ [accessed 25 October 2018]

[17] *Autopilot Pattern* [online] Available http://autopilotpattern.io/ [accessed 25 October 2018]

[18] *Singularity* [online] Available https://www.sylabs.io/ [accessed 25 October 2018]

[19] *Prometheus* [online] Available https://prometheus.io/ [accessed 25 October 2018]