Using Containers to Speed Up Development, to Run Integration Tests and to Teach About Distributed Systems

Marco Mambelli[1,*], *Bruno* Moreira Coimbra[1], *Namratha* Urs[1], and *Ilya* Baburashvili[1]

[1]Fermi National Accelerator Laboratory, PO Box 500, Batavia IL 60510-5011

Abstract. GlideinWMS is a workload manager provisioning resources for many experiments, including CMS and DUNE. The software is distributed both as native packages and specialized production containers. Following an approach used in other communities like web development, we built our workspaces, system-like containers to ease development and testing. Developers can change the source tree or check out a different branch and quickly reconfigure the services to see the effect of their changes. In this paper, we will talk about what differentiates workspaces from other containers. We will describe our base system, composed of three containers: a one-node cluster including a compute element and a batch system, a GlideinWMS Factory controlling pilot jobs, and a scheduler and Frontend to submit jobs and provision resources. Additional containers can be used for optional components. This system can easily run on a laptop, and we will share our evaluation of different container runtimes, with an eye for ease of use and performance. Finally, we will talk about our experience as developers and with students. The GlideinWMS workspaces are easily integrated with IDEs like VS Code, simplifying debugging and allowing development and testing of the system even when offline. They simplified the training and onboarding of new team members and summer interns. And they were useful in workshops where students could have first-hand experience with the mechanisms and components that, in production, run millions of jobs.

1 GlideinWMS and HEPCloud

GlideinWMS [1, 2] (GWMS) is a pilot and pressure-based Workload Management System (WMS) provisioning computing resources in a distributed environment. HEPCloud [3] is also a pilot-based WMS, but thanks to its Decision Engine [4], it can use more complex resource-provisioning strategies. Their users can request one or more customized elastic HTCondor [5] clusters, User Pools, in green in figure 1, where the users run their computations. GlideinWMS provisions the clusters by sending Glideins to a variety of computing resources, also called pilot jobs, to distinguish them from the scientific computations, the user jobs. GlideinWMS has been and is used at scale in production for more than 10 years by many collaborations, including the Compact Muon Solenoid (CMS) experiment, many Fermilab experiments, and the Open Science Grid (OSG). Most scientists do not use GlideinWMS directly or the clusters it provides, instead, they interact with the various tools or portals like CRAB, JobSub, or OSG-Connect, provided by the scientific collaborations.

*e-mail: marcom@fnal.gov

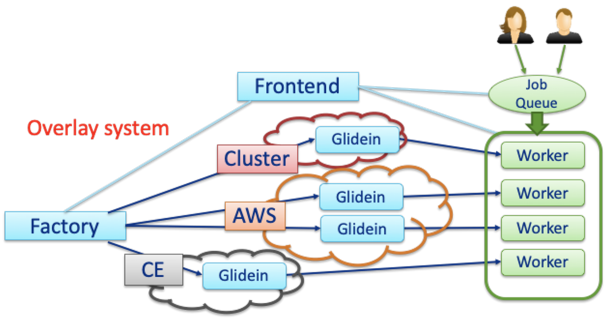

Figure 1. GlideinWMS system. GlideinWMS components are in blue, the User Pool is in Green, and the computing resources are in other colors.

The Glidein, the pilot job, is the key component in GlideinWMS. It is a program sent to many resources that match the preliminary requirements of the user jobs, to test and set up each computing resource to run the user jobs. It manages credentials, provides monitoring and audit information, can auto-detect and report node resources like CPU cores, memory, disk, and GPUs, and can install tools like a container runtime or a distributed file system. It finally joins the User Pool to run one or more user jobs, in parallel and in sequence, depending on the needs and availability.

The Factory and clients like the Frontend or HEPCloud's Decision Engine complete the GlideinWMS system. For this paper, we will consider a system with one Frontend, one Factory, and their Glideins. Actual deployments may include multiple clients, differing in how they calculate the requests for the Factory, and multiple Factories, providing a redundant distributed system.

The Factory is in charge of submitting Glideins to the different Compute Entrypoints (CEs). It knows how to reach each computing resource, which collaborations are supposedly supported, which protocols and authentication methods are supported, and if there are throttling requirements. It submits Glideins, maintaining the pressure on each CE, i.e. number of queued and running Glideins, requested by its clients. The Factory monitors the Glideins and hosts a secure mailbox to exchange requests and status messages with the clients.

The Frontend and other clients are aware of the users' requests and the running and queued Glideins that can be used for those requests, they receive resource status information from the Factories, and they use heuristics to update the requests to the Factories so that all the user jobs can run promptly, all limits and policies are respected, and no resources are wasted. The Frontend is generally operated by the scientific experiments or on their behalf and implements their policies for resource provisioning and job priorities.

A minimal deployment of GlideinWMS constitutes a Factory, a Frontend, a CE, and a virtual cluster using the Glideins as Execution Points. Figure 2 shows how this can be deployed on three hosts: a one-node cluster to run the Glideins, with an HTCondor-CE and HTCondor as batch system; a Frontend node also dubbing as Access Point, submit host, and Central Manager of the User Pool; and a Factory.

2 Using Containers

Containers, as a virtualization technology, have become ubiquitous in cloud native software development and deployment workflows over the past decade. Using virtualization at the operating system level, containers enable multiple isolated environments for applications within

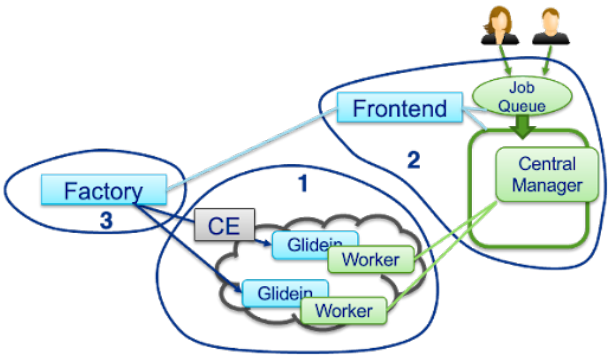

Figure 2. A minimal GlidinWMS deployment.

a single host. In contrast to virtual machines, containers are lightweight since they share the same Linux kernel and sometimes more, while virtual machines emulate the whole Operating System and sometimes even the hardware. However, thanks to the kernel *cgroups* and *namespaces*, containers can still manage dynamic allocations and provide isolation of processes, files, network, and users. Containers are very portable since everything needed to successfully run an application anywhere – the code and its dependencies, together with sections of the operating system – is packaged (aka containerized) as a single unit. This results in quick, efficient, and effortless deployment and management of applications without having to worry about dependencies and interactions with other installations.

Containerization has long existed before most modern-day solutions such as Docker [6], Podman [7], and Apptainer [8]. Docker has accelerated container adoption with its user-friendly platform. At Fermilab, we prefer Podman, a free open-source alternative, mainly for licensing reasons. Apptainer is the platform used to run containers in the Glideins because it is designed for distributed computing: it has compact single-file images, runs as a regular unprivileged application, and minimizes overhead, making nesting easier. All of these include containers' runtime support and tools to build them. Frequently, there exists an integrated environment, monitoring tools, and a graphical user interface (GUI); all of which make it easier to use containers.

Containers are instantiated starting from a blueprint, the (container) image, one or a set of self-contained, static files that encapsulate all the necessary components of the container, including code, libraries, and configurations. A running container, managed by its runtime support, also includes a context and the ability to interact with it. Images are normally defined using a "recipe" file, e.g., the `Dockerfile`, which includes all the instructions to build an image: where to start from, what to install, copy over, or configure. Additionally, containers have orchestration solutions to easily manage and coordinate services running in multiple containers on one or more host nodes. *Docker*, or *Podman*, *Compose* can create volumes and networks and start multiple services with a single command, as directed by its YAML configuration file, the `compose` file. Kubernetes and Red Hat OKD can manage and automatically scale multiple instances of many containers.

Traditionally, the GlideinWMS software has been distributed as a tarball or as a native RPM (RPM Package Manager) package, which is installed on production or development hosts. The development and testing of the GlideinWMS framework have been mainly carried out on virtual machines hosted in FermiCloud [9], an infrastructure-as-a-service (IaaS) private cloud deployed at Fermilab using OpenStack [10].

The normal GlideinWMS software development lifecycle (SDLC) includes active development and a couple of parallel releases, each supporting multiple OS versions, all at the same time. This requires many VMs, because a minimal system deployment includes three hosts: a Frontend, a Factory, and a Compute Entrypoint (CE), as described in section 1. The CEs can sometimes be shared by multiple deployments, but not the Factory and Frontend. FermiCloud always provided several VMs dedicated to GlideinWMS testing and development, some shared across the team and some private for individual developers. However, the number of VMs grew even more during the summer terms due to facilitating workshops and interns collaborating on the GlideinWMS project, which resulted in poor utilization of FermiCloud resources. Furthermore, there were operational overheads to manage the VMs: requesting and renewing host certificates, adapting Puppet configurations, configuring the firewall, extra training requirements, etc.

A preliminary effort to containerize GlideinWMS resulted from the collaboration with the OSG Consortium, where the members were interested in containers to simplify their production operations of Frontends at first and then of the Factories. Taking inspiration from Web developers like the ones running the Alnoda Hub [11], we borrowed the name *workspaces* and designed containers that are similar to virtual machines, to facilitate quick and easy deployment of the GlideinWMS software, focused towards our development and testing goals. The workspaces differ from production containers, which are typically microservice-focused, each running a single application. Our workspaces are more like Linux hosts, running multiple services and including many tools, which is more conducive for development and testing workflows.

3 The GlideinWMS Workspaces

We started with four workspace images: one for each of the nodes in the minimal deployment illustrated in figure 2: the CE, Frontend, and Factory, and a fourth one, the GWMS workspace, which abstracts the common elements of the other three. Having a common base allowed us to localize the customizations for the different platforms in that image and parameterize the other three. We wanted to support multiple RHEL-based OSes, such as Alma Linux 9 with Python 3.9 and Scientific Linux 7 with Python 3.6, on both the AMD/Intel and ARM architectures.

The ***gwms-workspace*** image comes in three flavors: Alma Linux 9 and Alma Linux 8 with Python 3.9, and Scientific Linux 7 with Python 3.6, all supporting both *x86_64* and *aarch64* architectures. All images start from the official Alma Linux or Scientific Linux image and add some customizations, the OSG RPM repositories, HTCondor, common software packages, and development tools. The first difficulty was to support multiple services, as in the production hosts. Since containers are designed to run a single service, `systemd` [12] does not work in containers. To work around this, we installed `supervisord` and used an emulation of the `systemctl` command that allows us to use our documented commands to start and stop the GlideinWMS services.

The three additional development blueprints are parametric, i.e., they can extend any of the `gwms-workspace` images for the different platforms, and are each responsible for one component of the GlideinWMS framework:

1. ***ce-workspace*** – is a minimal yet fully functioning SciToken-authenticated computing resource. The template for this workspace installs and sets up the HTCondor batch system, HTCondor-CE, and a few utility scripts. Additionally, this can be configured to fake more computing cores than the available ones.

2. ***factory-workspace*** – is a GlideinWMS Factory, where the underlying template handles the installation and setup of the Factory, including the download of HTCondor tarballs for the Glideins and a working configuration for the minimal deployment. This blueprint also contains an optional setting to link the GlideinWMS installation to a Git repository, so developers can run using the source code instead of a release.

3. ***frontend-workspace*** – is a GlideinWMS Frontend and the HTCondor Access Point (AP) and Central Manager (CM) for the User Pool. The template installs and sets up both the Frontend and the User Pool, and downloads some utility scripts, including the one to link the Git repository, some to refresh credentials, and one to submit jobs to run a smoke test.

To have a fully working test environment, we need the containerized hosts to talk to each other. We opted to use *Docker*, or *Podman*, *Compose* [13] to orchestrate the containers, which is a simple and easy-to-deploy option, without the need for scaling and multi-host features offered by other solutions. A `compose.yml` file describes the configuration for the deployment of three hosts with the `ce-workspace`, `factory-workspace`, and `frontend-workspace` services (containers). The orchestrator provides a bridged network for the containers with name resolution for the fictitious *glideinwms.org* domain. It also mounts shared volumes on all hosts to share secrets or code. A startup script generates self-signed host certificates from the included trusted CA certificate.

This setup satisfies the basic needs for development, testing, and training. We added a few extensions to cover more use cases. A parameterized *Compose* configuration allows exposing containers' ports to interact with outside components, e.g., real computing resources not available as containers, like AWS or HPC centers, or GlideinWMS services deployed on hosts or VMs. To test new releases, we defined an Integration Test-Bed (ITB), the ***testbed-workspace***, using an automated deployment script and starting from a new bare-bone image with little more than a minimal OS installation, so that the installation procedure and RPM package dependencies could also be tested. Finally, we added ***build-workspace***, which provides a container to build and serve GlideinWMS releases. Using the GlideinWMS `ReleaseManager` tool (included in our software) and RPM tools, this workspace can build release packages starting from any Git reference, local or on GitHub. These can be served by the local YUM server or can be built and distributed by OSG, via its repositories and Koji server. The *build-workspace* can also be included in the ITB to package, install, and test new code automatically.

A note about the decision to support multiple architectures: many developers on our team use Apple Silicon Macs, based on *aarch64* (ARM) processors. Thanks to VMs, it is possible to run *x86_64* containers on them, and we tested different options using QEMU or Apple's Rosetta 2, but all bring a considerable performance loss, of 20% or more. So we decided to add support for the *aarch64* architecture, also called *linux/arm64*. This required some modifications to the image description files and a more elaborate build process described in section 4.1, but the resulting use of images, thanks to container manifests, is completely transparent to users, who can simply pull or run the containers while the runtime support "magically" chooses the correct architecture.

4 Continuous Integration/Continuous Deployment

For a long time, GlideinWMS has used scripts and GitHub actions and workflows [14] to automate code testing and the release process. It was natural to design and develop similar workflows to build workspaces, thereby simplifying the execution of Continuous Integration/Continuous Deployment (CI/CD) pipelines in GlideinWMS.

4.1 Workspaces CI/CD using GitHub and Docker Hub

We added GitHub workflows to build workspace images for all the supported platforms and to push them to Docker Hub [15], an image registry. These images are pulled to run GlideinWMS workspace containers, and the push of changes to the containers in the Git repository triggers the build of new images. This creates the CI/CD loop shown in figure 3. The

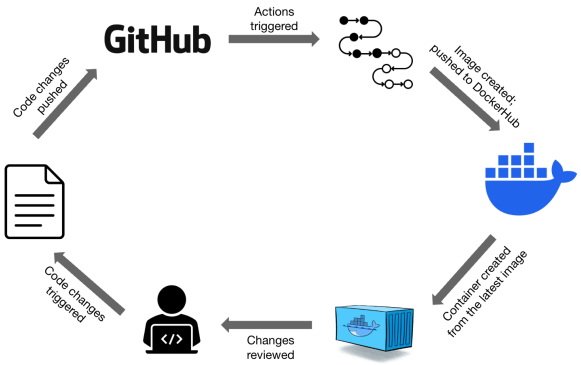

Figure 3. Steps of the CI/CD loop for GlideinWMS Workspaces.

workspaces' GitHub workflow is parametric and works for multiple OS platforms. It uses Docker Buildx [16] to provide manifests and multi-architecture images (*linux/amd64* and *linux/arm64*). Besides the automatic invocation mentioned above, it can also be triggered via *dispatch*, using GitHub's Web or CLI interfaces. The availability of updated multi-platform and multi-architecture images on a well-known public registry, such as Docker Hub, is very convenient to make the workspaces available to other groups or for training events.

4.2 GlideinWMS CI/CD using Testbed Workspaces

We are using the `testbed-workspace` to test new releases with the addition of scripts to simplify and automate most of the process. Future work will automate this test and add it to the GlideinWMS CI workflows on GitHub. This is not trivial because the test requires the tester to authenticate on a Web portal to obtain the SciToken required to submit Glideins to the CE [17]. Fermilab Managed Token service allows setting up a host that can request SciTokens on behalf of the user doing the initial setup. Using the tokens provided, we can fully automate the deployment and testing of new releases on that host. The next step will be to devise a similar solution for a dynamic setup like the one on GitHub.

5 Using GlideinWMS Workspaces for Development, Test, and Build

The introduction of GlideinWMS workspaces has simplified, sped up, and automated many workflows. Everyone can run the test system on their laptops and, after downloading the workspace images, they can be offline. Linux is the preferred container platform, but we have instructions to run on Windows using WSL2, and the ARM support allows running natively on M1 Macs, using CoLiMa or Podman. You can also run the workspaces from an Integrated Development Environment (IDE) such as Microsoft Visual Studio Code [18].

Developers can quickly spin up a GlideinWMS system, test their changes, switch to different versions, or reproduce an existing setup for troubleshooting. They don't need to set

up and maintain their build server on a Linux host. The testing of new releases on multiple platforms is almost fully automated.

The project onboarding is very efficient. Even inexperienced interns can run and tweak a complete working system in seconds, a process that used to take weeks. They can change parts of the system and inspect all its details. We also used the GlideinWMS workspaces for workshops and summer schools, such as the Computational HEP Traineeship Summer School 2024 [19].

6 Conclusion

GlideinWMS is a distributed system that can be emulated using at least 3 nodes: a CE and Cluster, a Frontend and Virtual Cluster, and a Factory. Workspaces are multi-process containers used to run each of the nodes, and container composition allows the use of a single command to make a dedicated network and run all the containers in the network. We experimented with complex deployments, including multiple clients, adding a node to build and serve new releases, and connecting with external elements. Multi-platform container images are distributed via Docker Hub, making the process seamless even on different architectures. GitHub workflows are used for CI/CD to automate the testing and building of the images that are available on Docker Hub. The workspaces introduced in this paper have been actively used for development, testing, and training by the GlideinWMS team, other groups within Fermilab, and students, thereby demonstrating their inherent nature of being easily adoptable by others. Our prior experience in deploying GlideinWMS on virtual machines was extremely valuable in understanding how to effectively design these workspaces and containerize them for rapid deployments, while improving usability.

7 Acknowledgments

The authors' work was performed using the resources of the Fermi National Accelerator Laboratory (Fermilab), a U.S. Department of Energy, Office of Science, HEP User Facility. Fermilab is managed by Fermi Forward Discovery Group, LLC, acting under Contract No. 89243024CSC000002.

References

[1] I. Sfiligoi, glideinwms—a generic pilot-based workload management system, Journal of Physics: Conference Series **119**, 062044 (2008). https://dx.doi.org/10.1088/1742-6596/119/6/062044

[2] I. Sfiligoi, M. Mambelli, P. Mhashilkar, D. Box, M. Mascheroni, K. Larson, B. Holzman, J. Weigand, A. Tiradani, H.W. Kim et al., glideinwms/glideinwms: Glideinwms 3.10.5 (2023), https://doi.org/10.5281/zenodo.8383959

[3] Mhashilkar, Parag, Altunay, Mine, Berman, Eileen, Dagenhart, David, Fuess, Stuart, Holzman, Burt, Kowalkowski, James, Litvintsev, Dmitry, Lu, Qiming, Moibenko, Alexander et al., Hepcloud, an elastic hybrid hep facility using an intelligent decision support system, EPJ Web Conf. **214**, 03060 (2019). https://doi.org/10.1051/epjconf/201921403060

[4] P. Riehecky, K. Knoepfel, D. Litvintsev, P. Mhashilkar, M. Mambelli, V.D. Benedetto, P. Gartung, S. Bhat, L. Goodenough, B. Coimbra et al., HEPCloud/decisionengine: HEPCloud decisionengine 2.0.2 (2022), https://doi.org/10.5281/zenodo.7108649

[5] T. Tannenbaum, D. Wright, K. Miller, M. Livny, in *Beowulf Cluster Computing with Linux*, edited by T. Sterling (MIT Press, 2001)
[6] S. Ratliff, Docker: Accelerated container application development (2025), last Accessed: February 28, 2025, https://www.docker.com/
[7] Podman, last Accessed: February 28, 2025, https://podman.io/
[8] Apptainer - portable, reproducible containers, last Accessed: February 28, 2025, https://apptainer.org/
[9] S. Timm, K. Chadwick, G. Garzoglio, S. Noh, Grids, virtualization, and clouds at fermilab, Journal of Physics: Conference Series **513**, 032037 (2014). https://doi.org/10.1088/1742-6596/513/3/032037
[10] Open source cloud computing infrastructure – openstack, last Accessed: February 28, 2025, https://www.openstack.org/
[11] Alnoda workspaces, last Accessed: February 28, 2025, https://alnoda.org/
[12] System and service manager, last Accessed: February 28, 2025, https://systemd.io/
[13] Docker compose | docker docs (2024), last Accessed: February 28, 2025, https://docs.docker.com/compose/
[14] N. Friedman, Github actions now supports ci/cd, free for public repositories (2021), last Accessed: February 28, 2025, https://github.blog/news-insights/product-news/github-actions-now-supports-ci-cd/
[15] Docker hub container image library | app containerization, last Accessed: February 28, 2025, https://hub.docker.com/
[16] docker/buildx: Docker cli plugin for extended build capabilities with buildkit, last Accessed: February 28, 2025, https://github.com/docker/buildx
[17] Mambelli, Marco, Coimbra, Bruno, Box, Dennis, Transitioning glideinwms, a multi domain distributed workload manager, from gsi proxies to tokens and other granular credentials, EPJ Web of Conf. **295**, 04051 (2024). https://doi.org/10.1051/epjconf/202429504051
[18] S. Swaminathan, M. Mambelli, Windows setup · glideinwms/glideinwms wiki (2024), last Accessed: February 28, 2025, https://github.com/glideinWMS/glideinwms/wiki/Windows-Setup
[19] M. Mambelli, Virtualization: Containers, Computational HEP Traineeship Summer School 2024 (2024), last Accessed: February 28, 2025, https://indico.cern.ch/event/1405035/