Developing Edge Services for Federated Infrastructure Using MiniSLATE

Joe Breen
University of Utah
joe.breen@utah.edu

Lincoln Bryant
University of Chicago
lincolnb@uchicago.edu

Jiahui Chen
University of Utah
jiahui.chen@utah.edu

Emerson Ford
University of Utah
emerson.ford@utah.edu

Robert W. Gardner*
University of Chicago
rwg@uchicago.edu

Gage Glupker
University of Michigan
gglupker@umich.edu

Skyler Griffith
University of Utah
skyler.griffith@utah.edu

Ben Kulbertis†
University of Utah
ben.kulbertis@utah.edu

Shawn McKee
University of Michigan
smckee@umich.edu

Rose Pierce
University of Chicago
rosepierce@uchicago.edu

Benedikt Riedel
University of Chicago
briedel@uchicago.edu

Mitchell Steinman
University of Utah
mitchell.steinman@utah.edu

Jason Stidd
University of Utah
jason.stidd@utah.edu

Luan Truong
University of Utah
luan.truong@utah.edu

Jeremy Van
University of Chicago
jeremyvan@uchicago.edu

Ilija Vukotic
University of Chicago
ivukotic@uchicago.edu

Christopher Weaver
University of Chicago
cnweaver@uchicago.edu

ABSTRACT

Modern software development workflow patterns often involve the use of a developer's local machine as the first platform for testing code. SLATE mimics this paradigm with an implementation of a light-weight version, called MiniSLATE[?], that runs completely contained on the developer's local machine or scales to larger machines (laptop, virtual machine, or another physical server). MiniSLATE resolves many development environment issues by providing an isolated and local configuration for the developer. Application developers are able to download MiniSLATE which provides a fully orchestrated set of containers on top of a production SLATE platform, complete with central information service, API server, and a local Kubernetes cluster. This approach mitigates the overhead of a hypervisor but still provides the requisite isolated environment. They are able to create the environment, iterate, destroy it, and repeat at will. A local MiniSLATE environment also allows the developer to explore the packaging of the edge service within a constrained security context in order to validate its full functionality within limited permissions. As a result, developers are able to test the functionality of their application with the complete complement of SLATE components local to their development environment without the overhead of building a cluster or virtual machine, registering a cluster, interacting with the production SLATE platform, etc.

*Corresponding Author
†First Author

CCS CONCEPTS

• **Computer systems organization → Grid computing, Edge Computing**.

KEYWORDS

Distributed computing, Containerization, Edge computing

ACM Reference Format:
Joe Breen, Lincoln Bryant, Jiahui Chen, Emerson Ford, Robert W. Gardner, Gage Glupker, Skyler Griffith, Ben Kulbertis, Shawn McKee, Rose Pierce, Benedikt Riedel, Mitchell Steinman, Jason Stidd, Luan Truong, Jeremy Van, Ilija Vukotic, and Christopher Weaver. 2019. Developing Edge Services for Federated Infrastructure Using MiniSLATE. In *Practice and Experience in Advanced Research Computing (PEARC '19), July 28-August 1, 2019, Chicago, IL, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3332186.3332236

1 MOTIVATION

Multi-institutional research collaborations propel much of science today. These collaborations require platforms connecting experimental facilities, computational resources, and data distributed among laboratories, research computing centers, and in some cases commercial cloud providers. The scale of the data and complexity of the science drive this diversity. SLATE (Services Layer at the Edge) [? ?] provides a distributed multi-institutional research computing environment, utilizing an edge computing architecture [?], which can support projects with a rapid "DevOps" type of deployment model. These projects range from service deployments to application "batch" payloads. The development efforts attempt to follow a combination of science guidelines and "cloud native" best practices[? ?] so the results might easily migrate between institutional data centers to cloud services. In this paper we discuss our approach of developing containerized edge services across distributed research computing platforms. We also describe the MiniSLATE[?] platform we have developed in order to provide a seamless workflow from development to operations.

2 APPROACH

Developing for a federated infrastructure such as SLATE requires a consistent pattern for the packaging and deployment of services. The idea of "services" can represent one or more software or data applications bundled to work together to deliver some deliverable based on specific input. The SLATE development model will allow a developer to create a deployment for a service in a local environment based on one or more applications and then bundle them together as a service to deliver to others. Once bundled, the SLATE development model allows the consistent packaging of the service so that the developer may deploy the service across geographically disparate sites with a few commands. SLATE supports both edge infrastructure services, such as caches and data transfer endpoints, and domain applications which perform particular scientific calculations. There can be important differences between these categories, but for the context of this paper they can generally be treated together and will be referred to generically as ‘applications’.

2.1 Example Scientific Application - XCache

XCache, based on the XRootD [?] transport protocol, is a distributed caching application used by many scientists to store localized or regional copies of large data sets. This application can reduce the latency of access, as well as reduce the bandwidth needs of the wide area network connections to the local environment. Reliable operation of the XCache application, without local systems administrator input, is very important for the successful utilization of network resources by the experiments at the CERN Large Hadron Collider, which must deliver data to hundreds of HPC centers. Resources needed by XCache depend on the scale of the computing resources served by the cache, which range from a single caching node with 10 Gbps network interface card and a few terabytes of disk, to a cluster of caching nodes and hundreds of terabytes of disk. The single-node use case is very simple: a single pod with the XRootD server. A cluster installation requires two applications per node (XRootD and CMSD), and a "master" application that unifies them. Figure 1 shows a production implementation which uses SLATE as the management layer in the edge network. A production deployment of the XCache application will have three more applications comprising the full application:

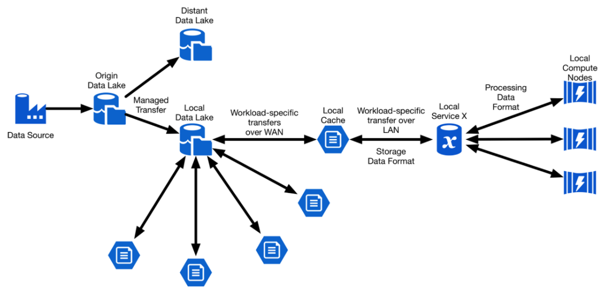


Figure 1: Example distributed caching and data delivery application using SLATE

A data lake can deliver experimental data to processing facilities using a network of caching servers and purpose-built delivery services that transform the data into the needed format.

- Application registration: A health-check probe
- Cache state reporting: Metadata plus cached blocks of data accessed
- Summary stream reporting: Reporting on operational parameters

2.2 Development Workflow

The review of the requirements of the XCache application and the review of other distributed scientific applications has allowed the SLATE project to abstract a typical development workflow into several major components. In order to ensure consistency across distributed environments, and, to provide security guarantees required by resource providers of the edge services, the SLATE model comprises the following major components for a typical Development workflow:

- Break scientific application into discrete processes with clearly defined dependencies.
- Create an appropriate set of Docker containers that compose the application processes and dependencies.
- Write appropriate configuration scripts for the set of containers.
- Write appropriate packaging scripts for the set of containers.
- Document ports, system level requirements and security requirements required by the scientific application.
- Submit packaged application to SLATE incubator catalog.
- Test deployment on SLATE platform.
- Validate functionality and security posture of scientific application.
- Request acceptance to production catalog.
- Deploy scientific application across acceptable sites.
- Deploy operational monitoring of scientific application.

This description of a typical workflow allows SLATE to focus and highlight areas where developers of science applications have struggled in getting their applications out to collaborators and other members of the community.

2.3 Containers

The SLATE project focuses on scientific applications which can deploy within containers. Containers are a lightweight and efficient construct which are able to present applications and processes in an isolated and consistent method. Containers allow the virtualization of only the operating system and key processes of a particular application. Due to their lightweight characteristics, containers are relatively straightforward to orchestrate with appropriate tools such as Kubernetes, Mesos, or Docker Compose. The orchestration of containers allows a scientific developer to break apart a application into discrete components and run these components in a modular and portable fashion. In turn, this modularity and portability allows a scientific developer to maintain and swap out components as the science workflow evolves over time. Modularity also has the additional advantages of isolating any conflicting libraries or versions of code. Use of containers lends itself to scientific reproducibility by allowing different versions of code to run, and also lends itself to enhanced security posture by providing minimal attack vectors on very small areas of isolated code. The portability aspects allow ease of support across different environments. With the help of the orchestration tools, the containers can also migrate easily across disparate sites.

The SLATE model relies on the ability of the developer to successfully break the application into discrete processes with clearly defined dependencies. The developers can then put these processes into separate containers which SLATE can orchestrate within a site or across federated sites.

2.4 SLATE Platform

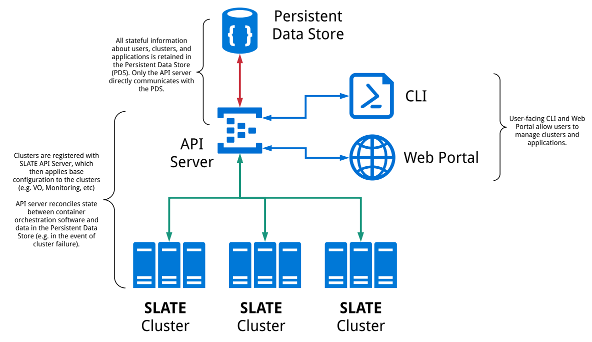
Once a developer has created the requisite containers, they can be deployed on the SLATE platform across geographically disparate sites. The SLATE team has created a platform architecture [?] with a centralized application deployment model in mind. Science groups interact with a centralized SLATE Application Programming Interface (API) to deploy their applications in a secure and consistent fashion across multiple federated SLATE clusters. This API has a RESTful design which both the SLATE command line tool and the web portal use.

The SLATE team has designed the platform to make deployment and operation as streamlined as possible. Figure 2 gives a schematic picture of the SLATE architecture. The SLATE Web Portal will provide views for science groups to observe the status of the applications they have deployed as well as for operators of the underlying edge clusters to view how science groups are utilizing the hardware.

3 MINISLATE: A PERSONALIZED DEVELOPMENT PLATFORM

3.1 Necessity of a Development Platform

Science applications are often made up of several discrete components. Each of these components have specific dependencies, configurations and characteristics. As the science evolves, the application must evolve with it. This fact implies that components and configurations with their requisite characteristics must occasionally change with minimal disruption to the end user. Science developers



Persistent Data Store
All stateful information about users, clusters, and applications is retained in the Persistent Data Store (PDS). Only the API server directly communicates with the PDS.
CLI
User-facing CLI and Web Portal allow users to manage clusters and applications.
API Server
Web Portal
Clusters are registered with SLATE API Server, which then applies base configuration to the clusters (e.g. VO, Monitoring, etc)
API server reconciles state between container orchestration software and data in the Persistent Data Store (e.g. in the event of cluster failure).
SLATE Cluster
SLATE Cluster
SLATE Cluster

Figure 2: SLATE architecture

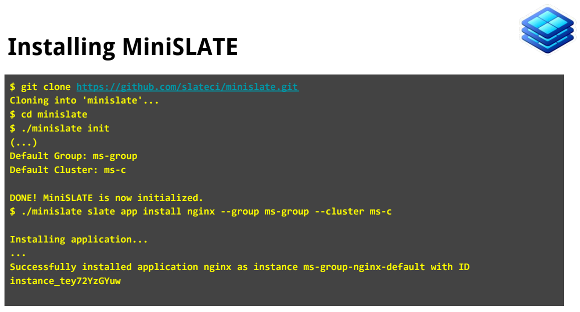
need a "sandbox environment" in which to regularly test new ideas, iterations of components, and changes to configuration files.

Given the number of steps involved, and potential for delay and disruption in the software development cycle, a local, standalone environment which mimics the essential features of the production SLATE platform is needed. MiniSLATE achieves this goal by providing a fully self-contained environment complete with the SLATE API server, SLATE command line and other tools. MiniSLATE allows the developer to test containers, configurations and application characteristics all within a determined state. The developer can rely on this state thereby reducing concern of wasted time and resources from potential unknown variables in their development environment.

The self-contained nature of the MiniSLATE development environment gives the developer confidence that the deployed application will transition from development to production in a seamless manner. Since the environment contains the same tooling and orchestration mechanisms as the production SLATE platform, the developer can test various iterations of the components of the application and have a fair confidence that the results will deploy consistently. The developer can also prepare an application simultaneously for different groups that require different characteristics and configurations because the MiniSLATE environment also inherently supports the same namespacing and other isolation techniques that the full SLATE platform supports.

3.2 MiniSLATE Architecture

MiniSLATE is a complete implementation of the production SLATE platform packaged in Docker Containers and orchestrated within the Docker Compose [?] framework. This configuration allows a minimum set of requirements for the endpoint (python, Docker, Docker Compose), and also allows the installation and orchestration of the entire development environment to be light-weight, consisting of only a small number of scripts and configuration files. Figure 3 shows an example of the simplicity of installing MiniSLATE. With the addition of a few performance tweaks for personal



Installing MiniSLATE

```
$ git clone https://github.com/slateci/minislate.git
Cloning into 'minislate'...
$ cd minislate
$ ./minislate init
(...)
Default Group: ms-group
Default Cluster: ms-c

DONE! MiniSLATE is now initialized.
$ ./minislate slate app install nginx --group ms-group --cluster ms-c

Installing application...
...
Successfully installed application nginx as instance ms-group-nginx-default with ID
instance_tey72YzGYuw
```

Figure 3: Example MiniSLATE installation

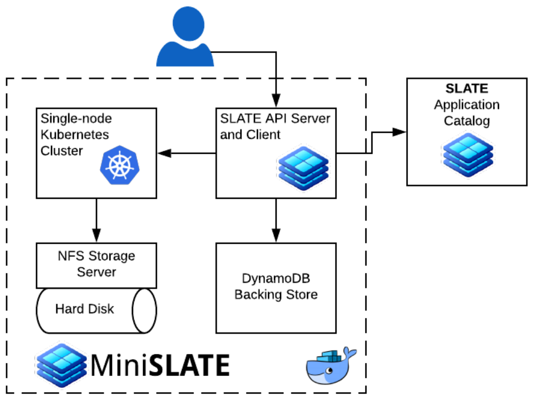

Single-node Kubernetes Cluster
SLATE API Server and Client
SLATE Application Catalog
NFS Storage Server
Hard Disk
DynamoDB Backing Store
MiniSLATE

Figure 4: MiniSLATE architecture

machines, MiniSLATE provides the science developer with the isolated environment necessary to create and iterate an application. Figure 4 shows the MiniSLATE architecture.

MiniSLATE utilizes four Docker containers to provide the actual development environment . These containers are the Kubernetes Node container, the SLATE Management container, the DynamoDB container and the NFS storage container. MiniSLATE pulls these docker containers from the respective sources during its build process. Docker Compose places each of these containers in the same local Docker network to allow direct communication between them. The orchestration provided by Docker Compose allows MiniSLATE to bring up the full development environment consisting of the four containers in a consistent and isolated manner.

3.2.1 Docker-in-Docker Kubernetes Node. The production SLATE platform uses Kubernetes for its underlying orchestration tooling. The MiniSLATE development platform offers this same tooling for its development environment. Kubernetes has various system requirements that one may not wish to satisfy on their development workstation. Therefore, MiniSLATE utilizes an architecture whereby it sets up a Docker-in-Docker[**?**] environment, and places Kubernetes inside the first Docker layer. MiniSLATE utilizes the second Docker layer for the container runtime interface. MiniSLATE provides some configuration changes to kubeadm on initialization to ensure compatibility with this environment. This architecture is similar to the kubernetes-sigs/kubeadm-dind-cluster project[**?**].

3.2.2 SLATE Management Container. All management controls for SLATE, including the SLATE client and API server with all dependencies exist within the SLATE Management container. This container contains other useful tools such as vim [**?**], kubectl [**?**], and helm [**?**]. A shared Docker volume exposes the "kubeconfig" file from the Kubernetes container into this container, directing the clients to connect to the Kubernetes API over the Docker network.

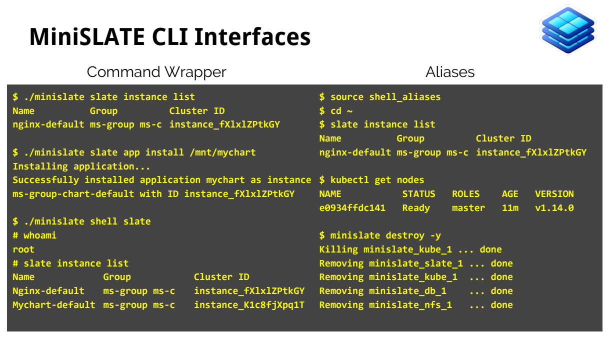
3.2.3 DynamoDB Container. The SLATE API server utilizes DynamoDB as a database for storing persistent information. The container image, dwmkerr/dynamodb on Docker Hub[**?**], provides a simple and easy to use deployment of DynamoDB that the SLATE API server utilizes while running in the SLATE Management Container.

3.2.4 NFS Storage Container. Production SLATE deployments currently utilize NFS as a persistent storage solution that is compatible with Kubernetes. In order to maintain parity with production deployments and to avoid the need to change code when moving from testing to production, MiniSLATE utilize a NFS storage container as well. The image, itsthenetwork/nfs-server-alpine on Docker Hub[**?**], provides a quick deployment of NFS4. An NFS client provisioner[**?**] is also installed by default on the Kubernetes node that can be utilized immediately by developers for their packages. Developers can also utilize other remote storage systems from MiniSLATE to handle existing data of any size by providing a storage class configured for the desired storage system.

3.3 Alternatives to MiniSLATE

Prior to developing MiniSLATE, the SLATE team evaluated current available solutions which might fit the needs of development. Primarily, MiniKube[**?**] was used, a popular local development solution for Kubernetes. MiniKube includes a Virtual Machine that runs Kubernetes and allows users to connect from their host. We found this solution to be workable, but with several drawbacks. Primarily, MiniKube did not allow for much customization within Kubernetes. Storage also had no support for dynamic provisioning. MiniSLATE provides a more robust solution, mimicking the storage solution used in the production SLATE platform. In addition, clients for Kubernetes, Helm, or SLATE had to be placed and configured on the host as well as any requirements to run MiniKube, such as a compatible hypervisor. MiniSLATE only requires Python and Docker/Docker Compose. All other dependencies are housed within the deployment and can be removed as easily as they are put in place. MiniSLATE also has no hypervisor requirement, helping save resources.

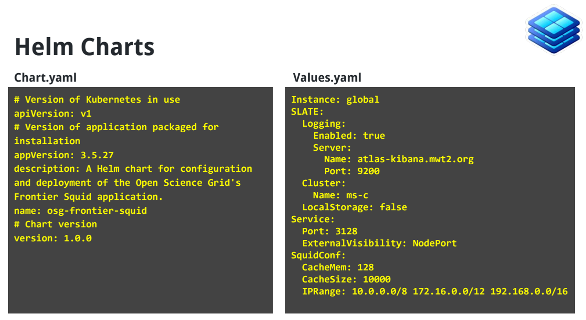
The SLATE project team also investigated a dedicated centralized SLATE development model which would require minimal effort on behalf of developers. The team set up this model, and like MiniKube, it provided a usable but imperfect solution. Issues such as configuration consistency, maintenance, refreshes, and most of all volatility resulted in the solution being difficult to keep in working order.



```
$ ./minislate slate instance list
Name          Group         Cluster ID
nginx-default ms-group ms-c instance_fXlxlZPtkGY

$ ./minislate slate app install /mnt/mychart
Installing application...
Successfully installed application mychart as instance
ms-group-chart-default with ID instance_fXlxlZPtkGY

$ ./minislate shell slate
# whoami
root
# slate instance list
Name            Group           Cluster ID
Nginx-default   ms-group ms-c   instance_fXlxlZPtkGY
Mychart-default ms-group ms-c   instance_K1c8fjXpq1T
```

```
$ source shell_aliases
$ cd ~
$ slate instance list
Name          Group         Cluster ID
nginx-default ms-group ms-c instance_fXlxlZPtkGY

$ kubectl get nodes
NAME           STATUS   ROLES    AGE   VERSION
e0934ffdc141   Ready    master   11m   v1.14.0

$ minislate destroy -y
Killing minislate_kube_1 ... done
Removing minislate_slate_1 ... done
Removing minislate_kube_1  ... done
Removing minislate_db_1    ... done
Removing minislate_nfs_1   ... done
```

Figure 5: Example of MiniSLATE Command Line Interface



```
# Version of Kubernetes in use
apiVersion: v1
# Version of application packaged for
installation
appVersion: 3.5.27
description: A Helm chart for configuration
and deployment of the Open Science Grid's
Frontier Squid application.
name: osg-frontier-squid
# Chart version
version: 1.0.0
```

```
Instance: global
SLATE:
  Logging:
    Enabled: true
    Server:
      Name: atlas-kibana.mwt2.org
      Port: 9200
  Cluster:
    Name: ms-c
  LocalStorage: false
Service:
  Port: 3128
  ExternalVisibility: NodePort
SquidConf:
  CacheMem: 128
  CacheSize: 10000
  IPRange: 10.0.0.0/8 172.16.0.0/12 192.168.0.0/16
```

Figure 6: Example of Helm chart and Values file

These attempts provided insights that the development model should support systematic code rule-sets and be completely destructible, so it could be reset to a known state whenever desired. As a result, the SLATE team emphasized these ideas as priorities in the development of MiniSLATE.

4 DEVELOPING ON MINISLATE

Once a developer has installed MiniSLATE, the full SLATE platform toolset is available. This includes the Kubernetes toolset for orchestrating the application component containers, Helm for packaging the component containers into a service to deploy, and the SLATE toolset, including SLATE client and API server, for deployment testing. Figure 5 shows an example of the MiniSLATE CLI.

4.1 Science Application Packaging

In the Kubernetes ecosystem, Helm [?] has become the leading de-facto standard for packaging and installing applications. This approach is similar to packaging applications in the Linux world, i.e. the RedHat Package Manager (RPM) for RedHat [?] and the Advanced Packaging Tool (APT) for Debian based Linux [?]. Helm provides templated YAML[?], called "charts", for packaging. Kubernetes uses this packaging to deploy applications in a consistent fashion with sets of initial configurations and values provided by Helm. This feature allows a developer to deploy a complete application made up of one or more applications on clusters that match the required specifications and permissions. This feature is a primary reason the SLATE project has adopted the Helm approach for providing packaging templates for applications. Figure 6 shows examples of a Helm chart and the Values file used for providing initial settings.

4.2 Science Application Deployment

Modern science workflows no longer depend on one or two applications independently installed, they often depend on a whole set of applications deployed together in a cohesive manner often at multiple locations. The use of the Helm toolset with the SLATE toolset allows for this type of deployment. The SLATE toolset has the concept of a "catalog" which lists packaged science applications from which users can pick to deploy. The SLATE toolset allows a user or developer to pick a science application and deploy it across one or many federated sites, provided the proper permissions exist.

The MiniSLATE development platform allows science developers to locally package one or more applications in a fairly straightforward manner. MiniSLATE utilizes Docker volumes for accessing developer code that is stored on the host. Developers mount their code inside a container directory and install their Helm charts from this directory as opposed to the production SLATE application catalog. This container directory is a feature available for development purposes only and is not available for production SLATE deployments. A developer can make iterative changes to their code and reset the environment as needed. When a developer has a relatively stable chart that they wish to make available in the SLATE catalog they can make a pull request to the catalog repository, placing their code in an "incubator" catalog. Charts will undergo a curation process which checks for compliance with the SLATE platform functionality requirements, minimal level of documentation, and validation that the application configuration will deploy in a reasonable fashion. When the curation process is complete and the science developer is ready, the developer can make a pull request to be in the SLATE "stable" catalog for other science users and developers to access at will and deploy at remote federate sites, provided proper permissions and requirements exist.

5 SUMMARY & OUTLOOK

MiniSLATE provides developers with a full suite of tools to develop and validate the functionality of their application in a consistent and easy to use environment. Developers use MiniSLATE for the development of packaged services to deploy onto the production SLATE platform. Developers are able to create, destroy, and recreate their environment as many times as needed to facilitate useful iterative debugging. At the same time, MiniSLATE provides a small footprint and isolates the development from the host machine. MiniSLATE will continue to evolve and adapt as further needs arise at https://github.com/slateci/minislate and http://slateci.io/.

ACKNOWLEDGEMENTS

This work is supported by the National Science Foundation Office of Advanced Cyberinfrastructure (OAC), grant number 1724821.