Moving from Composable to Programmable

Zhongyi Chen
Electronic Visualization Lab, Computer Science
Univ Illinois Chicago
Chicago, Illinois
zchen230@uic.edu

Luc Renambot
Electronic Visualization Lab, Computer Science
Univ Illinois Chicago
Chicago, Illinois
renambot@uic.edu

Lance Long
Electronic Visualization Lab, Computer Science
Univ Illinois Chicago
Chicago, Illinois
llong4@uic.edu

Maxine Brown
Electronic Visualization Lab, Computer Science
Univ Illinois Chicago
Chicago, Illinois
maxine@uic.edu

Andrew E. Johnson
Electronic Visualization Lab, Computer Science
Univ Illinois Chicago
Chicago, Illinois
ajohnson@uic.edu

Abstract—**In today's Big Data era, data scientists require modern workflows to quickly analyze large-scale datasets using complex codes to maintain the rate of scientific progress. These scientists often rely on available campus resources or off-the-shelf computational systems for their applications. Unified infrastructure or over-provisioned servers can quickly become bottlenecks for specific tasks, wasting time and resources. *Composable infrastructure* helps solve these problems by providing users with new ways to increase resource utilization. Composable infrastructure disaggregates a computer's components – CPU, GPU (accelerators), storage and networking – into fluid pools of resources, but typically relies upon infrastructure engineers to architect individual machines. Infrastructure is either managed with specialized command-line utilities, user interfaces or specification files. These management models are cumbersome and difficult to incorporate into data-science workflows. We developed a high-level software API, *Composastructure,* which, when integrated into modern workflows, can be used by infrastructure engineers as well as data scientists to reorganize composable resources on demand. Composastructure enables infrastructures to be programmable, secure, persistent and reproducible. Our API composes machines, frees resources, supports multi-rack operations, and includes a Python module for Jupyter Notebooks.**

Keywords—distributed systems, testbed implementation and deployment, composable infrastructure, deep learning, visualization, infrastructure as code

I. Introduction

We were first introduced to the company Liqid and the concept of 'composable infrastructure' at the IEEE/ACM Supercomputing (SC) 2017 conference. After several discussions, we purchased a small Liqid development system (devkit) in March 2018 as a proof of concept. Confident with this approach and the need to support the growing demand for GPUs from faculty in our College of Engineering, we received an NSF grant in November 2019 to purchase a large, two-rack system we named *COMPaaS DLV, COMposable Platform as a Service Instrument for Deep Learning & Visualization* [7, 8]. We began providing access to faculty in February 2020 and now, two years later, it is used by 40 research faculty and students in four College of Engineering departments and some external collaborators. The research applications are primarily GPU-centric for compute, with significant variability in storage and networking around the various applications' data requirements.

Composable infrastructure disaggregates a computer's components – CPU, GPU (accelerators), storage and networking – into fluid pools of resources, but typically relies upon infrastructure engineers to architect individual machines. Infrastructure is either managed with specialized command-line utilities, user interfaces or specification files.

Liqid provides an internal web-based interface and REST API to control and operate its hardware. Their interface enables the composition of bare-metal machines by managing pools of resources and monitoring the hardware. While this approach maintains system security and stability, it is cumbersome, time consuming and difficult to incorporate in workflows for data scientists and experimental computer science researchers who lack knowledge of the underlying hardware.

Also, each of COMPaaS's two racks/systems operates independently within its own fabric, making it cumbersome for infrastructure engineers to compose hardware based on users' requests. Early on, one of our major design goals was to support necessary functions across all composable fabrics with a single software environment.

Modern workflows require more diverse access models that are programmable. Today's major cyberinfrastructure (CI) projects are realizing the benefit and expressive power of *Infrastructure as Code,* where a user (an infrastructure engineer, CI researcher or data scientist) describes the required hardware and configuration, not through a portal and a series of panels (web-based gateways), but through code and APIs running inside a programming notebook (primarily in Python inside Jupyter).

Projects such as Chameleon [1, 2] and FABRIC [3] address similar functionalities at different levels. Chameleon provides bare-metal nodes that can be provisioned and configured through Python to build reproducible experiments. Similarly, FABRIC lets a user build virtual machines with specific requirements (in terms of SSD, NIC and GPU) that are passed from the host machine to a virtual machine.

We developed a high-level software API that we named *Composastructure* that has the following capabilities:

- Programmable, secure, persistent and reproducible infrastructure for use by both infrastructure engineers and data scientists
- Simple API to compose machines and free resources, independent of various vendors' APIs
- High-level functionalities
- Support for multi-rack operations
- Python module for programmable notebooks that supports the above-mentioned capabilities

COMPaaS DLV is funded by NSF award #1828265 to the University of Illinois Chicago.

II. Composastructure

Composastructure is a REST API that is higher level and easier to use than Liqid's web-based API. Composastructure abstracts the low-level logic required to manipulate composable resources and provides programmability and reproducibility while maintaining security and persistence. We focused on abstracting Liqid APIs, but one could also abstract and integrate the APIs of other vendors (such as GigaIO, HPE, Fungible, etc.) in the same way, hiding their API specifics and complexities.

Composastructure provides data scientists and infrastructure engineers with the following capabilities:

- List of currently composed machines and their attached resources
- List of available resources in the general pool
- List of available PCI-express (PCIe) fabrics
- Ability to compose, modify, and delete machines
- Ability to monitor the state of the fabrics and save/restore an entire fabric at once (for use by infrastructure engineers)

To build the Composastructure API, we first analyzed how the Liqid API handled machines and resource management. When we started this project, their API and the endpoints that were required were not fully documented. The existing graphical user interface (GUI) leveraged many of the API features, so our required abstraction was possible. We had to reverse engineer the GUI to understand their protocol to compose a machine. There was documentation for many of the API endpoints, but they lacked instructions on how to use the endpoints together. Chrome developer tools enabled us to observe when and what API endpoints were called while performing tasks with the GUI. With trial and error, we prototyped the composition and decomposition (freeing resources) of machines.

Liqid's API did not provide functionality to monitor the state of the infrastructure (machine creation, deletion, device update, etc.). Using reverse engineering, we determined that the GUI listened for server-side events (Stomp.js). We integrated these events into our API and determined that it was a safe strategy that did not interfere with using the vendor's API and GUI.

We first prototyped an API for infrastructure engineers to automatically create/delete a machine on COMPaaS. Since development was done in Node.js, useful libraries, like Express.js, made it easy to design an API and web application. Over time, we developed additional functionality based on observed needs.

Given the variety and complexity of each of our use cases, restoring the state of COMPaaS's infrastructure after a system failure or upgrade became burdensome to infrastructure engineers using Liqid's point-and-click user interface. We added functionality to monitor, save and restore the state of the infrastructure, reducing restoration of a saved state from hours to minutes. Additional features included modifying the state of composed resources on machines and implementing a public, secure login portal, which enabled our infrastructure engineer to do remote administration of the composable equipment racks without using intermediary login portals for access. With these features developed, we built a custom user interface to test the endpoints of our API and to provide quicker access to common tasks (building a machine, adding a component to a machine, freeing resources, and switching between our three composable fabrics that consist of our two full racks and the small devkit system). With low-level tasks implemented and tested, we then focused on making these functions accessible to data scientists.

Today, data science is predominantly conducted using programmable notebooks (primarily Jupyter) [4] to combine code, documentation, and visualization in a single resource. Cyberinfrastructure (CI) is becoming programmable *(Software Defined Infrastructure),* enabling data scientists and infrastructure engineers to configure CI to be reproducible and shareable. Some of COMPaaS's early adopters already used Kubernetes to deploy and orchestrate their applications. One idea we had was to parse their configuration files for resource specifications and build appropriate machines matching their requirements. However, more of our faculty and students were familiar with Jupyter Notebooks, so we decided to expose our API as a Python module that could be loaded into a notebook. This gives data scientists better control over their resources and enables them to conduct their experiments in a controlled and reproducible fashion. Configurations, saved as code inside notebooks, can be shared with collaborators and students, and disseminated within their research communities.

Our progress enabled composable infrastructure to be introduced to College of Engineering faculty through programmable APIs and services, exposed in computational notebooks, for interactive, reproducible and monitored experiments. We recently purchased an additional composable fabric to provide a JupyterHub frontend to support both our infrastructure engineers and our data scientists (JupyterHub launches Jupyter instances for each user of the system). The new fabric is overprovisioned (empty slots) to support future users and growing requirements. We believe that it was important to combine science workflows, which researchers use, with emerging technology to ensure early adoption.

Composastructure was prototyped and tested on our three composable systems. It is mostly written in Typescript and is open source. We are making it available for research purposes in a GitHub repository but without guarantees or support [5].

III. Composastructure API

Composastructure is comprised of a set of web routes (functions) and a server that hosts and runs those functions. We describe both aspects in the sections below.

A. API Description

This section shows elements of the Composastructure API that a programmer can use to operate the low-level fabric and compose machines out of available components (GPUs, SSDs, NICs and compute nodes). The code contains groups of web HTTP requests (i.e., routes), grouped into 'collections,' 'lookup,' 'details' and 'control' routes. Parameters to those routes are shown between '{}' (such as *id* or *fabric_id*). The requests are processed on a web server and return specific data types specified after the route's name. All data-type definitions are available in the public source code repository. The server code is described in the next section.

- *Collections:* Return the state of a particular group of elements in the system, such as groups, machines, available devices and fabrics (PCIe network). Groups are collections of machines defined by infrastructure engineers for administrative purposes.

```
GET /api/groups: Group
GET /api/machines: Machine
GET /api/devices: Device
GET /api/fabrics: Overview
```

- *Lookup:* Given an element ID (within a specific fabric), it returns the elements within the requested components, such as a group, a machine, or a device.

```
GET /api/group/{fabr_id}/{id}: Group
GET /api/machine/{fabr_id}/{id}: Machine
GET /api/device/{fabr_id}/{id}: Device
```

- *Details:* Given an element ID, it returns detailed information about that element; e.g., CPU or RAM information for a machine, core count and speed for a GPU, or capacity for a SSD.

```
GET /api/details/group/{fabr_id}/{id}:
 GroupDetails

GET /api/details/machine/{fabr_id}/{id}:
 MachineDetails

GET /api/details/device/{fabr_id}/{id}:
 DeviceDetails
```

- *Control:* Control-group routes include the operation that a user can perform on a composable system, mainly composing a machine out of available components and deleting such a machine.

```
POST /api/group GroupCreateOptions: GroupInfo

POST /api/machine ComposeOptions: MachineInfo

DELETE /api/group/{fabr_id}/{id}: GroupInfo

DELETE /api/machine/{fabr_id}/{id}: MachineInfo
```

B. *Composastructure Server*

We provide a web server that hosts and execute the routes, translating the requests into low-level commands to the appropriate fabric. You can host this server, which composes resources on multiple composable fabrics, on any machine with network access to those fabrics. The server handles the HTTP requests on the routes defined above and returns results and error information when needed.

The server is designed around two abstractions: an 'Observer' class that monitors the state of fabric and gets notifications of all updates and changes happening internally, and a 'Controller' class that provides commands to operate on the infrastructure, such as composing a machine or releasing devices. We also added high-level functionalities, such as clearing a whole rack at once, saving a rack configuration into a JSON file, and restoring a rack from any previously saved state (useful after outages or system upgrades). All events and activities happening inside the system can also be recorded over time for analysis and auditing (all compose and decompose events, components used, etc.).

```
var config: RestServerConfig = {
    systems: [{
        ip: '10.0.100.125',
        name: 'DevKit'
    }],
    hostPort: 3000,
    enableGUI: true,
    sslCert: {
        privateKey: "XXX…",
        certificate: "YYY…",
        ca: "ZZZ…"
    },
    adminLogin: {
        username: 'admin',
        password: 'compose'
    }
}

var Server = new RestServer(config);

Server.start().then(() => {
    console.log('Server started');
});
```

You can set up an administrator account, secure the communication with SSL certificates (using HTTPS protocol), and run the service on a specific port. We also include an experimental web user interface built on top of our API that is easier to use and customizable for infrastructure engineers. Above is the server configuration and execution code.

C. *System Design*

The architecture diagram (Figure 1) shows the overall design of the system, where applications use our API to talk to a single server in order to discover available components over multiple PCIe fabrics. The applications can then request composing machines out of these available components. Our API handles the requests and returns results (information, success or failure) to the applications. As composable hardware becomes more commonly deployed from multiple vendors, Composastructure could be expanded to support multiple low-level vendor-specific APIs.

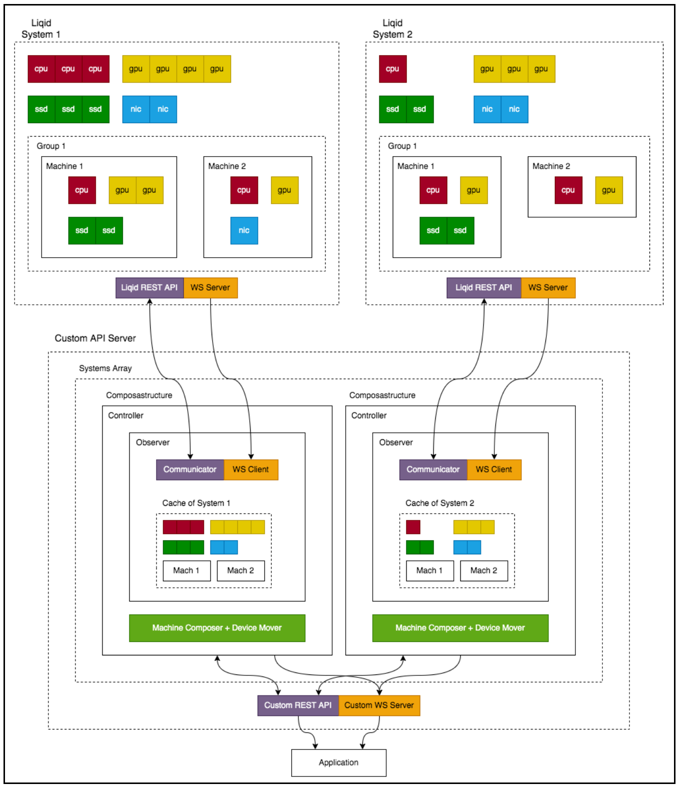


Fig. 1. Composastructure System Architecture

IV. Discussion

We used many common and tested packages to build our API and server, including Express.js [10] for the web framework, Passport.js [11] for authentication, and Socket.IO [9] for websocket communications. Passport.js enables authentication with many login strategies (Google, OpenId, CiLogon and many more). All services use secure communications with SSL certificates for data encryption. Socket.IO enables real-time bi-directional secure communication from the server to the clients listening for updates.

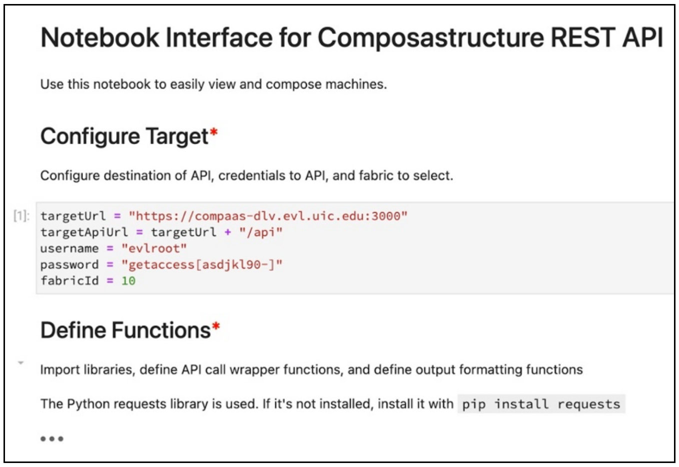
After prototyping the Composastructure API in late 2020, our infrastructure engineers have been successfully using it and we are now considering exposing it to a subset of our data-science users so they, too, can experience the benefits of composable infrastructure. It can be used to build an infrastructure predictably and reliably for many scenarios, from experimental testbeds to production systems. Commonly used packages and frameworks (Kubernetes, Containers, Jupyter, Python, REST APIs) can then be used to deploy the appropriate workload, from bare-metal experiments to container deployments.

For infrastructure engineers, our work exposes the minimal overhead of composability, offering a higher utilization rate of the hardware without overprovisioning bare metal, and lowering the management load using a Software Defined Infrastructure. It enables engineers to rapidly build simple configurations (i.e., it removes repetitive tasks) and complex configurations precisely matching application and workflow requirements (i.e., complex configurations based on knowledge and experience using the infrastructure). This knowledge can be captured either as code in notebooks or as templates for later use. Higher-level APIs will be exposed to facilitate common tasks, such as decomposing a whole rack, restoring a rack to a known configuration (previously saved), composing many similar machines, etc. One can envision templates for Data Transfer Nodes (DTNs, high-speed disk-to-disk dataset transfers), large AI/ML model training, or high-throughput inference, to cite a few examples.

To expose the potential of composable infrastructure to the widest audience (not just infrastructure engineers and advanced programmers), we developed a Python module for inclusion of composability in programmable notebooks. This enables experiments and system configurations to be done within a persistent, shareable and reproducible format [4].

Below are a few screenshots of such work, with our Composastructure API inside a Jupyter Notebook, showing how to initialize the API, to list available resources, to build a machine, and to add a device to a machine.

Figure 2 shows the configuration of a client to communicate with the server (URL, name and password). The Python code is then loaded into a notebook (the only software dependency is the commonly used '*requests'* Python module to perform REST calls).



Notebook Interface for Composastructure REST API

Use this notebook to easily view and compose machines.

Configure Target*

Configure destination of API, credentials to API, and fabric to select.

[1]:
```
targetUrl = "https://compaas-dlv.evl.uic.edu:3000"
targetApiUrl = targetUrl + "/api"
username = "evlroot"
password = "getaccess[asdjkl90-]"
fabricId = 10
```

Define Functions*

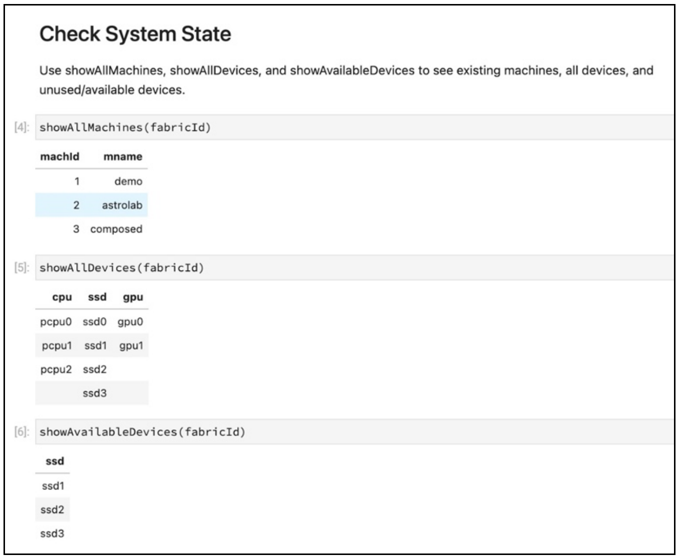
Import libraries, define API call wrapper functions, and define output formatting functions

The Python requests library is used. If it's not installed, install it with `pip install requests`

•••

Fig. 2. Initialization

Figure 3 shows how to request information about the state of the system for a given fabric (*fabricId* parameter). Here, three machines have been configured. We can request a list of all the devices in the system and see which ones are currently available. All data queries are presented in an easy-to-read format inside the notebook.



Check System State

Use showAllMachines, showAllDevices, and showAvailableDevices to see existing machines, all devices, and unused/available devices.

```
[4]: showAllMachines(fabricId)
```

machId	mname
1	demo
2	astrolab
3	composed

```
[5]: showAllDevices(fabricId)
```

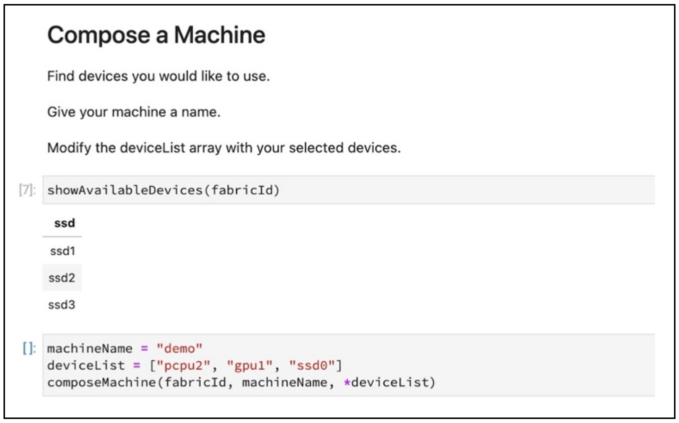
cpu	ssd	gpu
pcpu0	ssd0	gpu0
pcpu1	ssd1	gpu1
pcpu2	ssd2	
	ssd3	

```
[6]: showAvailableDevices(fabricId)
```

ssd
ssd1
ssd2
ssd3

Fig. 3. System State

Figure 4 shows how easy it is to create a machine, given a list of available components. Here, compute node 'pcpu2' is combined with 'gpu1' and 'ssd0' to form a machine called 'demo'. The API returns 'success' if components are currently available and if the compose operation performs successfully.



Compose a Machine

Find devices you would like to use.

Give your machine a name.

Modify the deviceList array with your selected devices.

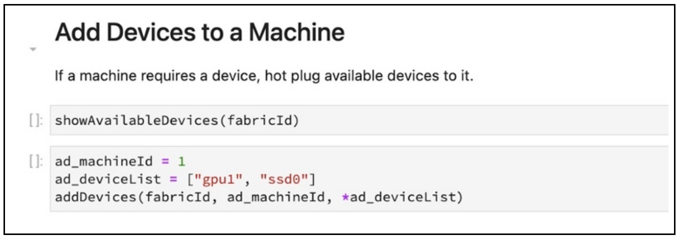
```
[7]: showAvailableDevices(fabricId)
```

ssd
ssd1
ssd2
ssd3

```
[ ]: machineName = "demo"
     deviceList = ["pcpu2", "gpu1", "ssd0"]
     composeMachine(fabricId, machineName, *deviceList)
```

Fig. 4. Composing a Machine

Figure 5 shows how to dynamically add a GPU and an SSD to an existing machine. More operations are available, such as releasing a component from a machine, decomposing a machine completely and information queries, described in our API description (Section III.A API Description).



Add Devices to a Machine

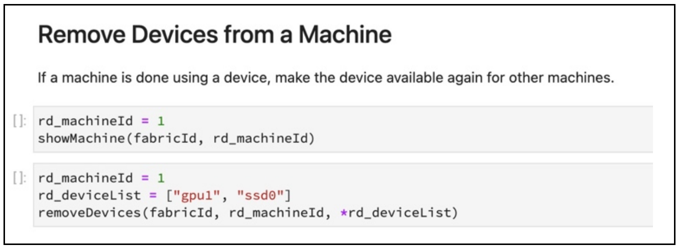
If a machine requires a device, hot plug available devices to it.

```
[ ]: showAvailableDevices(fabricId)
```

```
[ ]: ad_machineId = 1
     ad_deviceList = ["gpu1", "ssd0"]
     addDevices(fabricId, ad_machineId, *ad_deviceList)
```

Fig. 5. Adding a Device to an Existing Machine

Figure 6 shows how to delete components from an existing machine.



Remove Devices from a Machine

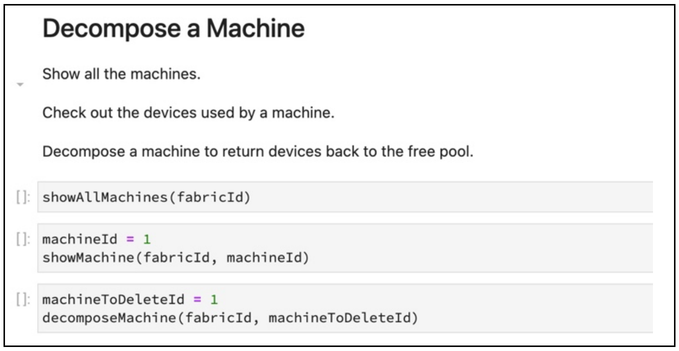
If a machine is done using a device, make the device available again for other machines.

```
[ ]: rd_machineId = 1
     showMachine(fabricId, rd_machineId)
```

```
[ ]: rd_machineId = 1
     rd_deviceList = ["gpu1", "ssd0"]
     removeDevices(fabricId, rd_machineId, *rd_deviceList)
```

Fig. 6. Removing a Device from an Existing Machine

Finally, Figure 7 shows how to decompose a machine and return the hardware to the available pool of components.



Decompose a Machine

Show all the machines.

Check out the devices used by a machine.

Decompose a machine to return devices back to the free pool.

```
[ ]: showAllMachines(fabricId)
```

```
[ ]: machineId = 1
     showMachine(fabricId, machineId)
```

```
[ ]: machineToDeleteId = 1
     decomposeMachine(fabricId, machineToDeleteId)
```

Fig. 7. Decompose a Machine

V. Conclusions

We presented our work to Liqid's product management team and our lessons learned helped influence their future implementations. We discussed documentation improvements, changelog implementations, and Software Development Kit (SDK) designs based on our experiences. These changes and additions will make it easier for their customers to develop software for composable infrastructure as it becomes more widely deployed.

Ben Bolles, Executive Director, Product Management, Liqid, said, "The University of Illinois Chicago team has been a tremendous development partner for Liqid through our joint collaboration on management and API integration projects, including integrating their management tools and Kubernetes with the Liqid composability and management APIs. … The team has helped influence and shape the Liqid software roadmap based on their early adoption of CDI (composable disaggregated infrastructure) for the benefit of other university researchers worldwide." Our work and experiences are shared in a white paper and press releases on the company website [12].

Another example of integration of composable infrastructure into existing workflows is Liqid's recent Slurm [6] integration. It better manages deployments of bare metal and containerized workloads and services for the composable infrastructure community. Slurm provides cluster management and job scheduling. This integration enables applications to request resources using Slurm, which are then automatically provisioned from a composable pool of available hardware.

Liqid is also working towards future integrations with Kubernetes.

We recently deployed an early prototype of GPUoE (GPU over Ethernet). Using a GPU expansion chassis connected to compute nodes over Ethernet, we were able to add remote GPUs into COMPaaS. Additional API endpoints were developed by the vendor to manage this type of remote GPU. We quickly extended our own work to support these functions. Hardware is now as malleable as software and, going forward, transitioning hardware resources between applications will become as commonplace as moving containers between hardware. This symbiosis of models creates an agile foundation for data scientists to continue to fully utilize resources and continue scientific progress.

Composastructure API can be used by modern application workflows to compose machines, servers, or make hardware changes based on an application's request. Concurrent work by industry to support Slurm and soon Kubernetes demonstrates a general trend in this direction. The popularity of Jupyter Notebooks and machine learning workload variability requires these types of responsive resources and infrastructures. Our API reduces the complexity of these emerging infrastructure designs and helps to lower the cost of management, while contributing techniques and best practices to the community.

ACKNOWLEDGMENT

This publication is based on work supported by NSF award #1828265. We also wish to acknowledge Liqid, Inc., who has been a significant industry collaborator, providing invaluable technical support. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of our funding agency or collaborator.

REFERENCES

[1] K. Keahey, J. Anderson, Z. Zhen, P. Riteau, P. Ruth, D. Stanzione, M. Cevik, J. Colleran, H.S. Gunawi, C. Hammock, J. Mambretti, A. Barnes, F. Halbach, A. Rocha, J. Stubbs, "Lessons Learned from the Chameleon Testbed," Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20), USENIX Association, July 2020

[2] Chameleon Infrastructure: https://www.thechibox.com/

[3] I. Baldin et al., "FABRIC: A National-Scale Programmable Experimental Network Infrastructure," IEEE Internet Computing, Vol. 23, No. 6, pp. 38-47, 1 Nov.-Dec. 2019, https://doi.org/10.1109/MIC.2019.2958545.

[4] Jeffrey M. Perkel, "Why Jupyter is data scientists' computational notebook of choice," Nature, 563, 145-146, 2018, https://doi.org/10.1038/d41586-018-07196-1

[5] Composastructure GitHub source code repository: https://github.com/Zhongy1/Composastructure

[6] Andy B. Yoo, Morris A. Jette, Mark Grondona, "SLURM: Simple Linux Utility for Resource Management," In: Feitelson, D., Rudolph, L., Schwiegelshohn, U. (eds) Job Scheduling Strategies for Parallel Processing. JSSPP 2003. Lecture Notes in Computer Science, Vol 2862, Springer, Berlin, Heidelberg, https://doi.org/10.1007/10968987_3

[7] M. Brown, L. Renambot, L. Long, T. Bargo, A. Johnson, "COMPaaS DLV: Composable Infrastructure for Deep Learning in an Academic Research Environment," MERIT (Midscale Education and Research Infrastructure and Tools) Community Event Workshop, 27th IEEE International Conference on Network Protocols (ICNP 2019), Chicago, Illinois, USA, October 7, 2019, http://doi.org/10.1109/ICNP.2019.8888070

[8] Lance Long, Tim Bargo, Luc Renambot, Maxine Brown, Andrew Johnson, "Composable Infrastructures for an Academic Research Environment: Lessons Learned," First Workshop on Composable Systems (COMPSYS '22), co-located with the 36th IEEE International Parallel and Distributed Processing Symposium, Lyon, France, June 3, 2022, accepted

[9] Socket.IO, Bidirectional and low-latency communication for every platform, https://socket.io

[10] Express, Fast, unopinionated, minimalist web framework for Node.js, https://expressjs.com

[11] Passport, Simple, unobtrusive authentication for Node.js, https://www.passportjs.org

[12] Liqid newsroom and blog, https://www.liqid.com/blog/case-study-uic-deploys-composable-infrastructure-for-uneven-applications-in-scientific-research