Towards an HPC Service Oriented Hybrid Cloud Architecture Designed for Interactive Workflows

Samuel Kortas & Mohsin Ahmed Shaikh
KAUST Supercomputing Laboratory
King Abdullah University of Science and Technology (KAUST)
4700-Thuwal, Jeddah Zip Code 23955-6900
Saudi Arabia
Email: (samuel.kortas,mohsin.shaikh)@kaust.edu.sa

***Abstract*—We detail *Ludion*, a service-oriented hybrid architecture, well-adapted to launch, monitor and steer interactive services running either on on-premise HPC resource, on user laptop and workstation or in the cloud. Based on AWS serverless components at virtually no cost, *Ludion* requires no special privileges. It consists of a catalog of services, a dashboard hosted in the cloud and a set of commands to install on the target resources. In the lifetime of a job, a user can register and publish new service on the dashboard. From this unique location, it is possible to trigger basic commands that are forwarded to the corresponding job. In this article, we expose several typical use cases *Ludion* was designed for and detail its implementation as well as the services already in place relying on this architecture.**

1. Motivation

Whereas the vast majority of use of our HPC systems is composed of jobs launched via batch scripts, more and more users are requesting access to these resources in a more “interactive” way. It can be to analyze data recently produced and ’trapped’ in its file system, to build on the fly or select input data (as for designing/tuning meshes, or testing some sets instead of others), to steer forthcoming steps of a computation, or to run Jupyter Notebooks to test ideas, train external users and team members.

Portals like Open OnDemand [1] or JupyterHub [2] provide excellent solutions to allow users to submit batch jobs or interactive tasks. But as their installation requires root access privileges, it demands initial actions and monitoring from system administrators. In our case, the deployment of Open Demand on our supercomputer Shaheen [3], [4] and cluster *Ibex* [5] supposes a detailed scans of eventual security vulnerabilities and an in depth analysis of its architecture to obtain the installation green light. While waiting for it, we had the idea of another possible architecture, *Ludion*, only requiring user privileges and relying on Cloud serverless components to ease its installation.

After presenting underlying architecture and underlining its benefits (section 2), interesting use cases covered by *Ludion* are proposed (section 3). In the end, details of implementation are given as well as the current status of use cases already deployed on our site (sections 4 to 6).

2. *Ludion* functional architecture

Two of the most appealing features of Open Ondemand or JupyterHub are the authentication of users and the proxification of their resources (files, running jobs) served via a web portal. These two highly security sensitive services are also the ones requiring system administrative privileges to be installed.

As sketched in Figure 1, the functional architecture of *Ludion* relies on the three following principles:

- offload/delegate the user authentication to the cloud (using the service *Cognito user pools* in the case of AWS),
- maintain a catalog of resources declared and updated at the discretion of each user: this information is stored in the cloud as a NoSQL base (*DynamoDB* tables for AWS),
- have this catalog available on the Internet but protected by an authentication of the user via a web or API interface.

2.1. Why partly relying on Cloud services?

In this very first implementation of *Ludion*, we chose to rely partly on AWS services. Still, our architecture choices were always guided by the principle that what was implemented can be easily ported to other cloud providers (Microsoft Azure, Google Cloud...) or hosted on premise based on MongoDB and LDAP for example.

At this preliminary stage of our project, partly building on AWS presented the following benefits:

- a **certain ease of implementation**: relying on already existing and highly efficient components with minimal deployment overhead as we use serverless services like (*Cognito*, *DynamoDB*, and *Lambda* functions that don’t require the installation or management of any resources. Building on these services also remain highly

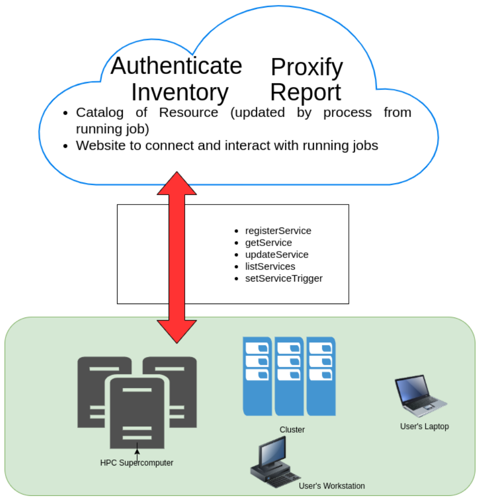


Figure 1. Ludion Functional Architecture

available and should scale easily when we switch to production mode.

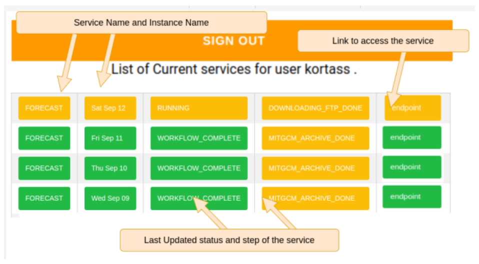
- a **minimal cost**: the use of these serverless components also signifies a expected minimal cost. As 25 GB of storage in *DynamoDB*, 1 Million of requests of *Lambda* functions, and several Millions of authentications via *Cognito* are included in the monthly free tier, we expect not be billed at a significant cost, and that even when the solution is deployed in production.
- a **straightforward management of user authentication** with *Cognito User pools*: this authentication solution is automatically set up once the web interface is deployed thanks to AWS *Amplify* and handles all the authentication issues at no cost either price wise or in term of management time.
- an **excellent interoperability** of the services: building a reactive and attractive website immediately relaying any update was easily implemented thanks to the supported triggering of a *Lambda* function as soon as any modification occurs some *DynamoDB* tables.
- most important, an **ideal connectivity** as AWS services appear to be reachable from every point of all our on-premise machines including from a compute node of our supercomputer Shaheen living in an internal network behind a firewall.

2.2. A typical use Case

In this section, we detail a typical use case to explain how the different interactions between on-premise and on-cloud resources occur via *Ludion* and how a user interaction on the website is propagated to the concerned resource.

Let's take the example of a given service, a Jupyter Notebook for example, deployed on our internal resource: the *Ibex* cluster.

1) The deployment of this service remains at the initiative of the user himself who logs to *Ibex* and runs the following commands:
```
% module load ludion
% startService \
      --service JupyterNotebook \
      --instance example
```
2) By design, on all systems covered by *Ludion*, only one service identified uniquely as {*JupyterNotebook.example*} is allowed to run. In order to enforce it, in the background, *startService* first checks if a service under this instance name is currently active and registered in *Ludion*. If it is the case, it returns an error message to the user, asking him either to change its instance name or consider reconnecting to the already existing same service. Information about already registered service are obtained via the command:
```
% getService \
      --service JupyterNotebook \
      --instance example \
      --parameters \
       "endpoint,status,login,password,x1"
```
precising as arguments of *–parameters* the list of the parameters to be read. *–service* and *–instance* are mandatory to identify the service as unique. If a parameter does not exist, the return value is returned as *NOT_SET*. The parameters *status*, *user*, *service*, *instance*, *id*, *jobid*, *description*, *step*, and *endpoint* are always valued. If no parameter is given only the status of the service is returned. In our case, if a service {*JupyterNotebook.example*} was ran and registered before, the answer returned would be the following JSON structure:
```
{ service : "JupyterNotebook",
  instance: "example",
  endpoint: "Ibex_000123:2030",
  status  : "COMPLETE"
  login   : "jupyter",
  password: "ok",
  x1      : "NOT_SET"
}
```
The following command:
```
% getService \
      --service JupyterNotebook \
      --instance example \
      --all-parameters
```
will return the current value of all parameters related to the instance *example* of service *JupyterNotebook.*
3) If no other service with same identity is running, a job is spawned on *Ibex* in the name of the user thanks to the SLURM command *sbatch*. The job is queued as any other user job and remains in the *Pending* state till it's scheduled.
4) Once scheduled, the job starts *Running* on the compute node *Ibex_000xxx*. At this stage, it looks for the first port *Port_nnn* available and starts a Jupyter Notebook


Service Name and Instance Name
SIGN OUT
Link to access the service
Last Updated status and step of the service

List of Current services for user kortass .

FORECAST	Sat Sep 12	RUNNING	DOWNLOADING_FTP_DONE	endpoint
FORECAST	Fri Sep 11	WORKFLOW_COMPLETE	MITGCM_ARCHIVE_DONE	endpoint
FORECAST	Thu Sep 10	WORKFLOW_COMPLETE	MITGCM_ARCHIVE_DONE	endpoint
FORECAST	Wed Sep 09	WORKFLOW_COMPLETE	MITGCM_ARCHIVE_DONE	endpoint

Figure 2. Ludion website Dashboard View

server responding on this port. It also eventually sets up a random password *Password_yyy* to secure access to the Notebook.

5) At this point, *Ludion* allows the registration of this new service in the centralized catalog. In the job script, following the successful launch of the Jupyter Notebook, the following command is called:

```
% module load ludion
% registerService \
     --service JupyterNotebook \
     --instance example \
   --endpoint Ibex_000xxx:Port_nnn \
     --user $USER \
     --password Password\_yyy
```

6) In the cloud, this newly registered service adds a new line in the *DynamoDB* database, and triggers the sending of a message signaling to the user that a service JupyterNotebook is now ready for him to access at the address *Ibex_000xxx:Port_nnn* using the password *Password_yyy*. User's contact email was required when registering to *Ludion*.
7) As shown on figure 2, it also triggers the automatic update of website showing the services initiated by the logged-in user. More details about the website features are given in section 4.
8) If steering the current workflow is required at this point, linking a UI event to a given action executed in the job is also possible (see section 2.3 below).
9) At any time, in the job environment, any update to the service is propagated thanks to the command:

```
% updateService \
     --service JupyterNotebook \
     --instance example \
     --param1 value_1 \
     --param2 value_2
```

param1 and *param2* are examples of parameters considered as worth to be traced or published on the centralized website. They can be of any name and in any number. As data related to services are stored in a NoSQL database, they could be of any type, including complex ones as picture, array, structure, or even document or whole log file. In this first implementation, *Ludion* supports strings only, but other types should be supported in the next releases. Note that after each update, the website is immediately updated.

10) Once the job is done, it is automatically marked as DONE or COMPLETED in the website, but all its information previously set or updated remain available to the user via the centralized website or corresponding request API.

2.3. Dynamic Steering of running jobs

As the website is updated whenever a change occurs in a row, a running job can also create a task running in the background, systematically triggered when *DynamoDB* table gets updated.

Ludion uses this feature to implement a way of steering the current job as after events initiated from the centralized website. Among the possible steering cases, one could:

- cancel current job
- invoke any action defined by the user. Among them: checkpointing the current state of the application, tail certain files, update certain parameters periodically read by the job, saving database...
- monitor in real time the resource used (either on main on all nodes): periodically we can easily retrieve current CPU loads, and list of processes and publish it via a DynamoDB table (see use case *performance analysis* detailed in section 3.2.4)
- extend the current job by starting a new one and updating the endpoint on the portal (possible workaround to extend interactive session without a system administrator intervention),
- duplicate the current job (auto-scaling resource in the case of Dask for example)

Some of the use cases exposed below (see section 3) take full advantage of this feature. With these, we both aim to implement the *Ludion* interactions most commonly used in workflows (like canceling the current job, monitoring the resources or trigger a checkpoint) but to allow a job to add a user-defined interaction by calling a simple command like

```
% setServiceTrigger \
     --service JupyterNotebook \
     --instance example \
     --widget button \
     --label "Save Database" \
     --calls ./save_database.sh
```

As soon as this kind of command is called, a button named 'Save Database' appears on the web interface. When clicked, it triggers a call to the script *./save_database.sh* in the job currently running.

This user-defined interfacing will be enriched in the next release of *Ludion* supporting other User Interface widgets, like lists, multiple choice widgets, or web forms accepting predefined or free parameters to be filled at click time and forwarded to the ongoing job on distant resources.

2.4. Side benefits of *Ludion* approach

Initially designed as a temporary workaround, *Ludion* appears flexible enough to present the following benefits:

- It naturally integrates resources as diverse as on-premise HPC systems, user's own laptop or workstation, or cloud hosted resources in a unique dashboard
- it eases the deployment of complex workflows as they can be broken in set of well-defined services that are deployable independently and access each other using *Ludion* as a catalog gathering all the already existing service one can connect to.
- This oriented-service architecture, spanning on different machine and environment also gives the opportunity to port more easily complex workflows, deploying each component where it is supposed to run the most efficiently.
- It allows users to publish any information they think relevant in a configurable web-based dashboard providing a smart way to interact with the currently running job, to access even more details and to make decisions.

3. Use Cases to implement

3.1. Simple use Cases

This category of use cases is straightforward to implement and to publish within a *Ludion* environment. They just require to start the service and publish its port on *Ludion* using the command *registerService* similarly to what is described at section 2.2.

These services have in common to run on one single node only. Prior to their launching, the procedure is to scan the first free port of the machine hosting the service and eventually to forward it via a ssh if it runs on an internal node of a Resource not accessible from the common network (see section 5.3). In the following paragraphs, we detail three cases already implemented on our resources, stressing for each of them their specific features.

3.1.1. Use case *Standard Service*: a database service. This use case matches the case presented above in section 2.2 on all aspects. The only refinement consists in scanning the first port available on the machine hosting the service, before computing a password and starting the database service, as the current user, in a very traditional way. The machine, chosen port and password are then registered into *Ludion*'s catalog calling *registerService*. Database supported can be either MySQL, PostgreSQL, MongoDB, or any database that can be launched is user mode.

In the background is started a process checkpointing regularly this database. The root password, user, user password, name of the database, and location on the file system is also saved in the detailed section of the catalog in order to allow a future restart of this database if the service is stopped. This checkpointing can also be triggered on demand from the web interface via a simple button revealed by the command *setServiceTrigger*.

In order to monitor the service, via the command *updateService* called on a timely-basis, we choose to publish some scalar values representative of the good health and current load of the database: number of users connected, number of request per minutes, time and response time of the last request, date of the last checkpoint. One could also publish the tail of the database system log file and update them regularly. This information remains accessible via the *Ludion* API or its website.

In the current version, only the last value of these quantities is available. But we could imagine to keep an historic of those if requested by the user in a future version of *Ludion*.

3.1.2. Use case *Simple Web Interface*: a Jupyter environment. Consisting in a classical Jupyter Notebook [2] or a JupyterLab environment, this service has just to be installed as a simple python environment and published via *Ludion*.

In the back end, after having chosen an available port, and a password, the service is launched via a simple command line and starts a web server in the background. The address of this server of the form *http://current_machine:chosen_port* is declared to *Ludion*, as an endpoint to which the user can directly connect via his current browser.

At this point, Jupyter environment is started from the user home directory and only one python environment is proposed where all major modules and Jupyter kernels are already installed. In a next release of *Ludion*, we will provide the possibility to change this starting point on the fly via the *Ludion*'s dashboard and to select a specific python environment, including user-defined.

Note that a very similar implementation is to expect to publish other IDE as *Visual Studio*, or any application user interface wrapped as a web server.

3.1.3. Use case *Container*: RStudio via singularity. For RStudio, this case is very similar to the one previously exposed for Jupyter with the exception that RStudio is launched in the background via a Singularity container. The installation is directly inspired from the solution used in Open OnDemand and will be detailed in [6]. After a scan a the first post available, the singularity container can be parameterized at launch time to set the port and a password to allow the connection.

As RStudio launches an http server, the address of type *http://current_machine:chosen_port* is declared to *Ludion*, as an endpoint directly reachable in one click from the dashboard or via as a link in the message sent by email.

Triggers actions and monitored values can either be connected to main processes (as CPU load, or memory usage or container footprint) or taken from the container by issuing some probing commands regularly to access internal values in the containers.

Same strategy could be applied to any container that can be run in user mode.

3.2. Advanced use cases

3.2.1. Use case *multiple nodes*: Dask. Dask is a useful tool for creating task farms of computation that can run in parallel, while working with familiar numpy and pandas like

interfaces. dask_jobqueue uses scheduler to submit multiple jobs to achieve throughput. These jobs can be launched together but may not start at the same time depending on the resource availability. *Ludion* can be used in this case to add the list of jobs and monitor the completion of complete workflow submitted by dask_jobqueue.
In addition to displaying status of the jobs, link to Dask Dashboard can also be published in order for the user to monitor it on the fly, in a more traditional manner.

3.2.2. Use case *multiple services*: Pangu LIMS. This open source Laboratory Information Management System (LIMS) [7], [8] consist in the combination of two services previously detailed : a database service and a web interface (see section 3.1.2) connected together allowing a user to launch a series of computations via his/her browser and save their results into the database.

In this scenario, *Ludion* makes possible to launch this two services in a completely disjoint way. First the database service is initiated as described in section 3.1.1. Once it has successfully started, one can gather the database endpoint by calling *getService* and use it in the configuration files of the web interface which is launched after having selected a free port on the machine hosting it. In the end, the endpoint of this web site is published in *Ludion*'s catalog along with a reference to the database service it is using.

Here *Ludion* framework eases the deployment of the LIMS infrastructure as it is split in two collaborative disjoint service (database and website) that can be even separately started on two different machines more adapted to their respective footprint or availability constraint. We could imagine that the database will be hosted by specific hosts with higher amount of memory or disk to handle the request more efficiently. The machine hosting the database should also authorize to run jobs for a long periods (several days or weeks or year) to guarantee the availability of the data. On the opposite, the website could be only spawned on demand, when a user needs to add some computation to an already existing database.

The same strategy could be applied to port any workflow: first split it in elementary services exhibiting well-defined API, spawn each of them on the best resource possible and register in *Ludion* catalog their endpoint, last and step by step connect them one to another thanks to the information stored in *Ludion*'s catalog.

3.2.3. Use case *monitoring and run on demand*: Igulf. This case covers a full-stack turn key robust solution that daily delivers a climate and ocean forecast running on our supercomputer *Shaheen* as a back end. We made it accessible via an intuitive web interface to launch on demand, additional pollutant spill simulation based on the latest daily forecast available.

In our current implementation, the daily forecasts are spawned thanks to a simple cron job running on *Shaheen* but the current status of the job, their traces or log, and the images and animation of the computation are published and immediately available for consultation to the expert user via a View dynamically built on the centralized *Ludion* web interface similar as the one shown on figure 3.

From an implementation point of view, it only required to add some calls to *updateService* at strategic points of the workflow and gave an excellent feeling of interactivity.

For the on demand computation, in order to submit a pollutant spill simulation, a specific web form has been designed to precise the parameter values to pass to te backend. A call to the *Ludion* API is made when pressing the form submit button which triggers the computation on a simple workstation that previously registered a Spilling service in *Ludion*'s catalog. At this point the workstation and *Shaheen* are sharing a common file system that does not require any transfer of file in order to run the spill simulation based on the most recent daily forecast available.

3.2.4. Use case *performance analysis*. This use case is completely orthogonal and can be instantiated along with any other case. The idea is to give access to a dynamic performance analysis of the resources currently used to the end user.

Its implementation is expected to be straightforward. The procedure is to start an additional monitoring process in the background of the desired job either from its very beginning or on demand via the API or web interface of *Ludion*. This added process periodically probes the resource in use (CPU, Memory, Swap space) on any node allocated by the job. These values are dumped into an additional *DynamoDB* table *Performances* and instant or historical values are made available to the end-user.

At will, one can adjust the collecting period time, and plot the time evolution of each value. Implementing alert with email notification in case of some particular threshold reached is easy to do as a part of a *Lambda* function triggered by any update of the *Performances* table. Monitoring a user-defined criteria should be also possible to support.

This performance monitoring feature has been expected by some of our advanced HPC users who are developing and tuning scientific applications. Its ease of use could be really appealing to them and authorize their own investigation where only system administrator's probing was the only solution available before on *Shaheen* for example.

With additional cost involved and specific permission in place, keeping the history of some job footprint may also be an option and the comparison of successive jobs run on same or different resources could be a valuable bonus. Granting access to this information to several users via *Ludion* web site with specific group permission could also provide a nice collaborative performance analysis tool.

3.2.5. Use case *Steering*: Orchestration of Checkpoint/restart of DL jobs. Most common deep learning frameworks such as PyTorch or TensorFlow offer a means of checkpointing and restart of process of training the experiment point of view this introduces resiliency in the training process so that a computationally expensive training job can be restart from where it left off/crashed due to some reason. From the point of view a multi-user environment with finite computational resources such as a shared HPC cluster, checkpointing allows breaking down the training process into a number of small jobs. This can help avoid congestion due to long running distributed training jobs which clogs the scheduler's queue for long duration. Resubmitting a job from where it left off may require some manual intervention before submission to validate the stop condition of the training process, e.g. checking if the training has reached maximum number of epochs or a prescribed quality metric, or even investigating if the loss is changing enough to allow further training or choosing an early stop. Given that an extra few lines of code is written to automate investigating the stop condition, one can automate the submission of subsequent jobs until the stop condition is satisfied. *Ludion*, in such case, is a good dashboard to monitor the state of this process which is broken into multiple jobs.

3.2.6. Use case *Steering*: Scaling out Hyperparameter Optimization. Hyperparameter optimization (HPO) to tune deep learning models may require to search through a parameter space of 10s to 100s of hyperparameters in order to get the best performance metrics. From the KSL users of HPC resources, these models usually end up contributing to research publication and the throughput of these standalone jobs submitted to the scheduler has both direct impact on the speed of science done and also to avoid congestion on a shared HPC cluster like *Ibex*.

Ludion's UI can be a good front end for describing an HPO experiment to and presenting the outcome of the complete HPO experiment, i.e the performance of each job as a grid to aide in the selection of best model parameters.

3.2.7. Use case *Launcher*: Service Launcher. Compared to a solution like Open Ondemand, a major missing feature of *Ludion* is the ability for the user to initiate a new service directly via the web interface. At this moment, security considerations prevent us to deploy this for any user. To do so, one would have to allow *Ludion* to connect to the account of any user and submit in his/her name a job or a process on the target resource. We are stopped here by the same constraint which stopped us to deploy Open Ondemand by lack of privilege.

With the current architecture, some alternative solutions can be proposed though:

- **deploying on AWS**: In theory, a user could be allowed to deploy on a click new services on the cloud as long as he is granted with the correct permission. An acccount can be set up for this user inside our organization in AWS and be attached to the correct policy via AWS *IAM*. It allows him to use some serverless resources or spawn a job on a cluster started in AWS via *parallelCluster*.
  This strategy is suitable to initiate a Jupyter note book for a training for example. Of course, additional costs are implied.
- **Start one's own service of deployment**: The other way is for the user to take the initiative to start a service of Job spawner on each resource he wants to initiate a service. Even in the current version of *Ludion*, this can be achieved using standard commands: after registering his service thanks to *registerService*, the idea is to call *setServiceTrigger* as many time as one wants to trigger the spawn of different services on the current resource. For example:

```
% setServiceTrigger \
      --service ServiceLauncher \
      --instance example \
      --widget button \
      --label "Start JupyterLab" \
      --calls ./start_jupyterlab.sh
```

  makes available a button *B1* tagged "Start JupyterLab" in *Ludion*'s dashboard on the view specific to *ServiceLauncher*. When *B1* is pressed, it triggers an immediate call to the local script *./start_jupyterlab.sh* that executes every step detailed in the example use case exposed at section 2.2. It will start the desired JupyterLab environment, send a mail to the user when deployed and add the service in *Ludion*'s dashboard
- **Implementing a queue of tasks probed from a cron job**: The need for a process constantly running on the current resource would be the main criticism to address to the solution just exposed above. One other possibility would be to bind the click on *B1* to the creation of a new row in a table *commandToForward* hosted in *DynamoDB*.
  Thanks to a cron job initated by the user, this table would be regularly consulted from each resource and any raw mentionning the resource would be translated in a local command (submission of a job, spawning of a process, deletion of a job...) before being deleted.
  With this implementation, a minimal use of the resource is required but the launching of the service is not as immediate and only occurs when the cron job is activated.

Although it may appear like an unsatisfactory workaround, this way of initiating services presents the advantage to link all the permissions required to the only execution of the script *./start_jupyterlab.sh*. From a security point of view, it considerably narrows the perimeter of components to audit to qualify as safe the deployment of this solution on-premise. When the permissions are implemented in *Ludion*, a system administrator of a given HPC resource could initiate the service *ServiceLauncher*, and for certain users of *Ludion*, grant them the right to click on the button *B1* that will trigger a slightly modify *./start_jupyterlab.sh* script spawning the desired service *in the name* of the user. As *ServiceLauncher* is ran by a system administrator, it has the correct permission to take any identity. But this procedure of delegation can be deployed more securely as it only concerns a well-defined list of call to be done in the name of the end user. It therefore makes *Ludion* much easier to qualify in term of security.

4. *Ludion* Interfaces

4.1. Application Programming interface

Registering a new service and publishing some of its details are available from the user job itself and do not request any root privilege. These can be done either by running commands executed from the Unix shell or from a language thanks to *Ludion*'s API.

At this stage, *Ludion*'s API is under development. For now, its public API is only available as Unix commands and NodeJS, Python and Dart languages are expected to be supported in the near future. The required coding effort is expected to be light as we currently rely on AWS *AppSync* service that provides a GraphQL interface rich enough to interact with *Ludion*'s services.

As described in section 2.2, the following Unix commands are available:

- *registerService* to register a new service,
- *getService* to get details about a service already deployed,
- *updateService* to update values relative to a service expected to be published on *Ludion*'s dashboard,
- *listServices* to list all available services,
- *setServiceTrigger* to establish a graphical widget in the *Ludion*'s dashboard in the view related to the current service and triggers its activation to an action executed immediately in the corresponding job

.

The latest detail of *Ludion*'s API can be found on the *Ludion*'s official documentation pages [9].

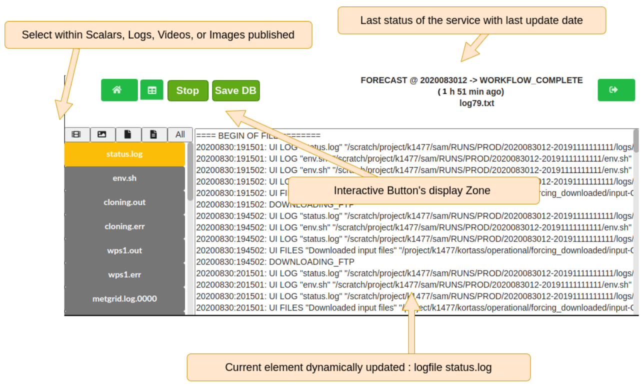
4.2. Web interface

Although *Ludion* web interface is still in development phase, here are its main features listed below.

4.2.1. User access and permission management. We designed the website to be freely accessible to any user of any of our HPC resources. They have to sign up by themselves and their access is granted after checking that their chosen login match their user name on the system. Via the *Cognito* service, they are free to change or reset their password and to modify their contact mail at any time without our intervention. *Ludion* uses this mail to send them any notification about the starting or change of their services. For security, a two-factor authentication can also be activated.

At this moment, a user can only interact with the services he initiated himself on the resources. In the future, a system of permissions will be implemented: access, monitoring and modification of services will be granted at spawn time by the user or later via the web interface. Among other applications, these permission management is relevant in the following scenarios:

- Database services are deployed on dedicated machine by database administrators whose responsibility is to take care of their security, availability, or planned backup and decides to grant their access to other services spawned by different users.
- One user could also be interested to publish some information produced by one of his services to some other users and allow them to interact with this service: browse data on demand, adjust a given strategy or trigger additional computation for example.



Select within Scalars, Logs, Videos, or Images published
Last status of the service with last update date
Stop
Save DB
All
status.log
env.sh
cloning.out
cloning.err
wps1.out
wps1.err
metgrid.log.0000
Interactive Button's display Zone
Current element dynamically updated : logfile status.log

FORECAST @ 2020083012 -> WORKFLOW_COMPLETE
(1 h 51 min ago)
log79.txt

```
==== BEGIN OF FIL  =======
20200830:191501: UI LOG   tus.log" "/scratch/project/k1477/sam/RUNS/PROD/2020083012-20191111111111/logs/
20200830:191501: UI LOG "env.s   /scratch/project/k1477/sam/RUNS/PROD/2020083012-20191111111111/env.sh"
20200830:191502: UI LOG "env.sh" "/scr  h/project/k1477/sam/RUNS/PROD/2020083012-20191111111111/env.sh"
20200830:191502: UI LC                                                      12-20191111111111/logs/
20200830:191502: UI FI                                                      rcing_downloaded/input-C
20200830:191502: DOWNLOADING_FTP
20200830:194502: UI LOG "status.log" "/scratch/project/k1477/sam/RUNS/PROD/2020083012-20191111111111/logs/
20200830:194502: UI LOG "env.sh" "/scratch/project/k1477/sam/RUNS/PROD/2020083012-20191111111111/env.sh"
20200830:194502: UI LOG "status.log" "/scratch/project/k1477/sam/RUNS/PROD/2020083012-20191111111111/logs/
20200830:194502: UI FILES "Downloaded input files" "/project/k1477/kortass/operational/forcing_downloaded/input-C
20200830:194502: DOWNLOADING_FTP
20200830:201501: UI LOG "status.log" "/scratch/project/k1477/sam/RUNS/PROD/2020083012-20191111111111/logs/
20200830:201501: UI LOG "env.sh" "/scratch/project/k1477/sam/RUNS/PROD/2020083012-20191111111111/env.sh"
20200830:201501: UI LOG "status.log" "/scratch/project/  477/sam/RUNS/PROD/2020083012-20191111111111/logs/
20200830:201501: UI FILES "Downloaded input files" "/  ect/k1477/kortass/operational/forcing_downloaded/input-C
```

Figure 3. Ludion website perService View

4.2.2. Website views. The website currently offers three complementary views:

- a *Dashboard* View displays the current status of active services as shown on figure 2. Each instance running are showing in a table that displays "basic" information like the host machine, the service and instance name, the current status, the starting date of the service and last update time of its status. The endpoint of the instance is also given. In the case of a web interface a click on this endpoint directs to the http address of service currently running on a given resource. Before clicking, a reminder of login/password credential can be obtained. This table is updated dynamically in a matter of second after any modification has been signaled from the running service environment. Table can be sort with respect to any column and a click on a row opens a perService View described below.
  A toggle button allows to consult information related to services previously launched and not running anymore. This could be useful to restart a database or get the configuration of a given past instance for example.
- a *perService* View gives access to all the "additional" information that a service decides relevant to publish. Any additional piece of data is made available almost instantaneously. They can be scalar values, log files, images, or movies. Interactive actions are also posted here: they are composed of simple graphical widgets (buttons, TextField, checkbox, or slider). Their positioning and appearance is driven by commands issued from the service running in the current of a job for example. Their post on the website is quasi immediate providing a very good feeling of interactivity. Taken from the current implementation of the service *Igulf*

described at section 3.2.3, An example of this View is shown in Figure 3

- a *resource* View displays the resources used by current services. They can be very generic as job id, job starting and remaining time, file systems space, or specific to the service as permission or CPU performance.

An *Administrator* View allowing the website administrator to manage user access and a *Debug* View showing a given user view to the administrator for debugging purpose are also under development. If needed, a *File System* view might be implemented in the future

Most of them are already implemented from other project or mock up and we remain confident our forthcoming integration effort will lead to a major release of the website by December.

5. A few implementation details

Purpose of this section is not to present the implementation of *Ludion* in detail but to insist on architecture specific points showing how the interaction of diverse components makes this solution especially reactive and robust. *Ludion*'s architecture and complete installation procedure will also be fully available the official *Ludion* documentation [9].

5.1. Cloud services used

As mentioned in section 2, while most actual computation occur on on-premise resources, *Ludion*'s architecture also relies on serverless services hosted in AWS Cloud:

- The centralized website, written in ReactJS, is deployed on the Cloud using AWS **Amplify** that also installs a fully managed GraphQL interface via **Appsync**. Any call to this interface, either by **Lambda** functions, or by *registerService*, *updateService* or *setServiceTrigger* immediately shows on the browser page of people connected.
- **DynamoDB** tables:
  - *Services* and *Services_detail* store information related to services registered by the users in *Ludion*. *Services* gathers information shown on the *Dashboard* View of the website and is part of the resource deployed by AWS *Amplify*: they are the name of machine, name of the service and of the instance, current status of the job, its job id, the user that spawned it, the hosting resource, and the endpoint address where the service can be reached. *Services_detail* contains all the detailed information that the user finds relevant to make available on the website.
  - *Mails*, as a temporary buffer to host mails to be sent to the user,
  - *Logfiles* to store log files desired to be shared on the centralized website. For large log files will be stored in S3 and addressed via a reference in this table.
- the AWS **SES** service to send mail

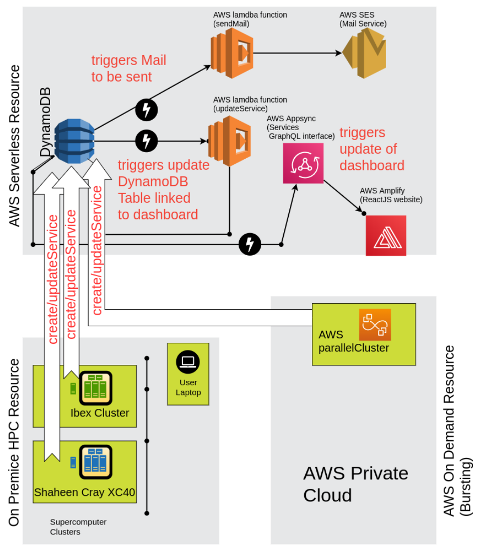


Figure 4. Ludion Architecture Detail

- 2 **Lambda** functions. *sendMail*, triggered by any update of *Mails*, forwards the mail to *SES*. *updateService*, triggered by any update of *Services_detail*, filters the updated information to the only ones published on the *Dashboard* View of the website and update the table *Services* via the GraphQL interface to trigger the update of the centralized website.
- **Cognito user pool** to handle authentication of users that wish to connect to the website. Users can register by themselves on the service, reinitialize their lost password or change their correspondence email. The only detail we have to check is that they choose the same login on **Cognito** as the one used on on-premise resource. In our specific case, users share the same login name on all resource (HPC supercomputer and cluster, laptop or workstation). The case of different login names for one same user depending on the resource will be handled in a future version of *Ludion*.

5.2. on-premise Resources

In the first release of *Ludion*, we aimed at potentially include all our internal HPC resources as well as any user workstation or laptop. Here are the machines currently covered:

- **Shaheen**, a Cray XC-40, composed of 4 front end nodes accessible from KAUST Network and 6,174 dual sockets compute nodes based on 16 core Intel Haswell processors running at 2.3GHz. Each node has 128GB of DDR4 memory running at 2300MHz. Overall the system has a total of 197,568 processor cores and 790TB of aggregate memory.
  It is worth noting that the compute nodes belong to a

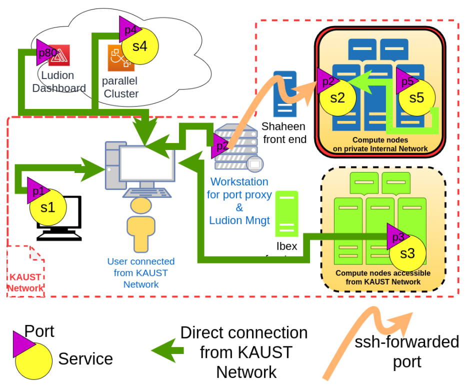

p80
Ludion Dashboard
p4
s4
parallel Cluster
p2
s2
p5
s5
Shaheen front end
Compute nodes on private Internal Network
p2
Workstation for port proxy & Ludion Mngt
p1
s1
User connected from KAUST Network
KAUST Network
Ibex
p3
s3
Compute nodes accessible from KAUST Network
Port
Service
Direct connection from KAUST Network
ssh-forwarded port

Figure 5. Ludion Network Connectivity

private network not directly reachable from the outside of *Shaheen*. Connecting a service running on *Shaheen*'s compute node therefore requires a forwarding of its port as described in section 5.3.

- *Ibex*, a heterogeneous cluster composed of nodes with various CPU architectures (e.g. Intel Skylake and Cascade Lake, AMD Rome) and an assortment of GPUs nodes with P100s, V100s, GTX1080Ti & RTX2080Ti, and with variable amounts of RAM ranging from 360GB to 700GB.
  Contrary to *Shaheen*'s architecture, *Ibex* compute are all directly reachable from KAUST Network easing the publication of *Ibex*'s Service to KAUST based users or other services
- **user own Resources**: *Ludion* allows users to register their own services running either on their workstation or laptop. They will appear on the same view than on-premise HPC services launched via a job, or on-cloud services. The only constraint is that the user login should be the same on all resources to appear under the same account.
- one additional **Workstation or VM** used as a proxy for services running on *Shaheen*'s compute nodes (see section 5.3 below), and to update periodically the status of jobs found completed or aborted either on *Shaheen* or *Ibex* by calling *updateService*. The only requirement for this workstation is to have a network interface on the KAUST network, and be reachable from a *Shaheen*'s compute node via ssh.

5.3. Network connectivity

In its current first implementation, *Ludion* architecture supposes that the end user connects to registered services only from the internal network of our university where all our on-premise HPC resources are located. For most of our users, this constraint appears reasonable as they are either on site or can join this network thanks to a VPN connection. As a consequence the endpoints advertised on *Ludion* website either correspond to the public address of a service hosted in the cloud, or to a private KAUST network IP address for a service running on an on-premise resource, a click on this latter type of link will only work from KAUST network that even if *Ludion* website is reachable from anywhere (see Figure 5).

For a service hosted on a compute node of *Shaheen* though, a direct connection is not possible as the compute nodes are part of an internal network not reachable from the outside. For these cases, the solution put in place is to build an ssh tunnel from the compute node to an intermediate workstation in the KAUST Network. This tunnel is built at the initiative of the service and the port on the compute node as well as the one mapped on the workstation are registered in *Ludion* tables *serviceDetail*. The endpoint published on the dashboard is the one on the workstation but everything is available for another service also running on a compute node of *Shaheen* to communicate on the internal port.

For *Ibex* or on other user workstation, the situation is straightforward as these resources are directly reachable from KAUST Network.

For a service hosted on the cloud, only services with public address are supported at this stage but one can easily build a VPN connection from KAUST network to an AWS Virtual Private Cloud and dynamically manage its security group

On any resource, we also need to scan for the first available port to start launching a new service. This port cannot be always the default one as some other process may run on the same host and already use this port. This is the case for example on *Ibex* which allows different jobs to share the same cluster node. One of the benefit of the dynamic catalog in *Ludion* is to allow services to connect to each other after their installation even if their port is dynamically chosen.

5.4. Security considerations

Security is a major concern when dealing with an architecture allowing a user to define and publish his/her own service. At this early stage, security measures are not fully implemented but should easy to configure relying on services provided in the cloud. For each connection we could for example set a token-based connection managed by AWS *Cognito identity Pools* service connected to KAUST onsite identity management system.

Widely adopted in an enterprise environment, we still have to think on how to make this feature at the user application level under a simple enough API. Without a doubt it will be a priority of the *Ludion* production release.

The dynamic management of the VPC security group is also an advantage as it is completely programmable via an API. We plan to update this security group for each registration of a new service and remove the access when the service has ended. This dynamic management of the firewall rules is not possible at this stage on our internal HPC resources but we will work on a solution with the system administrators of *Shaheen* and *Ibex*. Systematically go through the Workstation used as a proxy at least for the connection step to service could be an option but we have to explore further the possibilities available to us.

6. Current status and next steps

6.1. Services already deployed

At this early stage, the simple use cases listed in section 3.1 are deployed and fully supported: On demand, a user can launch a Database, start a Jupyter or JupyterLab notebook or RStudio session on any of our HPC resource and be notified by mail with all information required to connect. These services also appear in *Ludion*'s dashboard which can be remotely requested via API to connect service with one another.

As well, Users can spawn their own service using the Unix command line detailed in section 4.1 and following the example exposed in section 2.2. The triggering part needs to be stabilized and debugged more thoroughly before being deployed.

For the advanced use cases (see section 3.2), the three first ones (Dask, Igulf and LIMS) are deployed and used in a production environment whereas the profiling and two steering use cases are currently in the testing and development phase.

Every implementation details about use cases is fully available on the *Ludion* GitHub repository [6].

6.2. Next steps

Additionally to complete all other advanced used cases exposed in section 3.2, we also plan the following tasks:

- provide a user web interface to an AWS workflow very comparable to the one put in place for Dask (see section 3.2.1,
- finalize a cloud formation template to ease the deployment of *Ludion*architecture on AWS,
- Integrate *Ludion*'s dashboard in Open OnDemand,
- develop an on-premises only version of *Ludion* not using AWS services based on *MongoDB* and *Apollo*,
- provide an implementation of *Ludion* relying on Google Cloud or Microsoft Azure,
- develop a mobile application written with the *Flutter* toolkit once the *Dart* API is completed.

Ludion is also part of a wider project aiming at wrapping our on-premise HPC resources into a cloud friendly environment comparable to what user are used to interact with when using AWS, Azure or Google Cloud solutions. While *Shaheen* and *Ibex* are providing the computing resource and storage space to the user, we plan to add/develop the following missing components that would makes our HPC environment looks like a cloud to our users:

- An auto-scaling group component (see AWS scaling group introduction or Azure VM set),
- A file transfer service partially based on Globus,
- A global log service (equivalent to AWS *CloudWatch log insights*)
- An email and notification service,
- A message queue service to allow different process/job to easily communicate with other in a loosely coupled way (as AWS *SNS* service)
- the equivalent of spot instances accounted at low cost but that can be taken any time after a 2 minute notice,
- the support of *Lambda* functions is also studied.

7. Conclusion

In this article, we presented *Ludion*, a service-oriented hybrid architecture, currently under test and development at KAUST. *Ludion* is well-adapted to launch, monitor and steer interactive services running either on on-premise HPC resource, on user laptop and workstation or in the cloud. Requiring no special privileges to be installed and consuming serverless cloud resources at virtually no cost, it can offer an alternative to Resource portal like Open Ondemand.

Ten use cases have been detailed covering a wide range of actual requirements of our HPC users. Five of them are already developed and deployed in a quasi production environment. Their implementation in *Ludion* ease the access to our HPC Resources and provide a decent interactivity environment fully configurable by the user that was not possible so far. The details of *Ludion*'s API, as well as some specificity of its implementation provides a good idea of how adapted this solution could fit the needs of another resource center or company.

Released under Free BSD 2 license, *Ludion* is available as a GitHub repository and KAUST Supercomputing Laboratory is keen on collaborating on its development or providing help on its use, deployment or enhancement to potential educational or industrial partner.

Acknowledgments

For computer time, this research used the resources of the Supercomputing Laboratory at King Abdullah University of Science & Technology (KAUST) in Thuwal, Saudi Arabia.

References

[1] D. Hudak, D. Johnson, A. Chalker, J. Nicklas, E. Franz, T. Dockendorf, and B. McMichael, “Open ondemand: A web-based client portal for hpc centers,” *Journal of Open Source Software*, vol. 3, p. 622, 05 2018.

[2] T. Kluyver, B. Ragan-Kelley, F. Pérez, B. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. Hamrick, J. Grout, S. Corlay, P. Ivanov, D. Avila, S. Abdalla, and C. Willing, “Jupyter notebooks – a publishing format for reproducible computational workflows,” in *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, F. Loizides and B. Schmidt, Eds. IOS Press, 2016, pp. 87 – 90.

[3] Kaust Supercomputing Laboratory. Shaheen user web site. [Online]. Available: https://hpc.kaust.edu.sa

[4] B. Hadri, S. Kortas, S. Feki, R. Khurram, and G. Newby, “Overview of the KAUST’s Cray X40 system–Shaheen II,” *Proceedings of the 2015 Cray User Group*, 2015.

[5] Kaust Supercomputing Laboratory. Ibex user web site. [Online]. Available: https://hpc.kaust.edu.sa/ibex

[6] S. KORTAS. Ludion github repository. [Online]. Available: https://github.com/samkos/ludion

[7] J. Zhang, D. Kudrna, T. Mu, W. Li, D. Copetti, Y. Yu, J. L. Goicoechea, Y. Lei, and R. A. Wing, “Genome puzzle master (GPM): an integrated pipeline for building and editing pseudomolecules from fragmented sequences,” *Bioinformatics*, vol. 32, no. 20, pp. 3058–3064, 06 2016. [Online]. Available: https://doi.org/10.1093/bioinformatics/btw370

[8] J. Zhang. Laboratory information management system. [Online]. Available: https://jianwei-zhang.github.io/LIMS/

[9] S. KORTAS. Ludion documentation website. [Online]. Available: https://ludion.readthedocs.io/en/latest/