openGRIS - Open Standard for Grid Computing

OSFF London June 2025

Ritesh Bansal

Enterprise Compute: A spectrum of use cases

Long running compute on thousands of cores (simulation, stress testing)

Case Study:

A market risk group at a large financial institution typically has grids that are used between the hours of 5P-11P and is mostly unused the rest of the day. These grids can cost more than $50MM a year and the low utilization also has negative environmental impact.

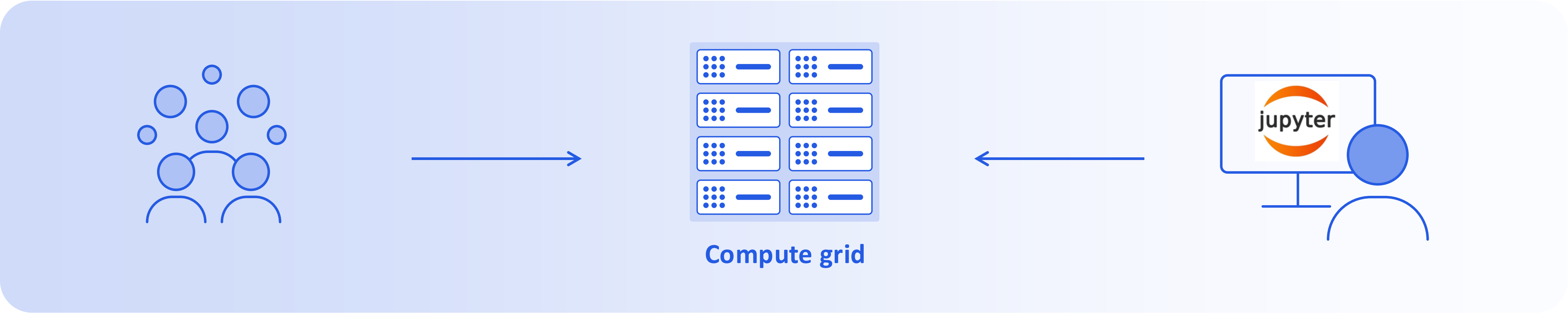
Someone sitting at desk want access on the fly (model development?)

Case Study:

A quant in a modeling team wishes to run an ad-hoc pricing or analytical run on a portfolio. The quant has to spend several hours setting up the environment on AWS and provisioning the cores.

Large compute

Small compute



Enterprises Face the Problem of Access to Scalable Compute

- What is Scalable compute?
  Easily "address" heterogenous compute pools with workloads.
  Compute pools have varying costs, time-based usage policies – peak, off peak, and compute features – GPU, AVX, etc.

- Write to one API and access different public cloud and grid providers

- On demand allocation (dynamic scaling) to avoid statically allocated grid which results in lower utilization

Rich Parallel Computing Ecosystem

Hardware, Public Cloud Providers

aws

Google Cloud

Microsoft Azure

IBM

OnPrem

Software Ecosystem

dask

RAY

APACHE Spark

Why does current ecosystem not work

Hardware and Cloud Service Providers

Cloud Service Providers

Write to the AWS, GCP, Azure APIs or run Dask, Ray, Spark on them to orchestrate the workloads. This is not an easy ask and companies like Coiled provide this as a service in the public realm.

Enterprise Grids

Write to IBM symphony or other Enterprise Elastic compute infra.

On-prem Hardware or VM

Parallelize using language primitives or use Dask, Ray, Spark on them.

Writing to each vendor API or public API prevents vendor lock-in and creates high adoption barriers.

Software Ecosystem

Dask

- Dask has been the mainstay of distributed programming in the Python ecosystem for both in-core and out-of-core computations
- Dask's increasing focus on out-of-core dataframes means its core implementation is increasingly complex
- Not ideal or efficient for simple task scheduling

Ray

- Complicated scheduler architecture—each machine has its own scheduler
- Poor graph scheduling performance
- Focus is distributed compute for machine learning applications

Spark

- Requires hosting a separate Spark cluster
- Large-scale code changes needed to conform to Spark APIs
- Focus is large out-of-core datasets and implicit data parallelism

What does the industry need?

TCP/IP solved a similar problem in the 1980s revolutionized networking and eventually dominated carrier networks for datea, voice and video.

Compute needs a light (thin) abstraction layer to enable routing (addressing) compute to various compute resources (EC2, Symphony, GCP, etc...)

The need is for a **lambda** service that can span across compute provide and **route** compute to the right pool based on a set of criteria

Proposal: An open Standard for Grid Resource Scheduling with client and worker standards to enable clients to send compute to workers in a standardized, cost-effective and user-friendly way across compute pools.

openGRIS: Wishlist

Metric	Challenges	Wishlist
Utilization vs. Cost	Low Utilization for dedicated grids leads to high cost	**Interfaces to Existing Grids – AWS, Symphony, Physical Servers**
Static vs. Scaling	Sharing grids is hard to solve so static grids are the norm	**Multi-language bindings**
Emissions	Lower Utilization grids lead to increased emissions	**Cost Accounting**
Policies/Entitlements	Grid entitlements and time-based usage policies	**Time Scheduling – Grids can be Pre-empted for Production Jobs**
Data Sovereignty	Data location prem/off-prem	**Multi-Environment Support via Docker/Podman**

What is being contributed

openGRIS – Open standard for Grid Resource Scheduling

Scaler – Reference implementation

@pargraph and **@parfun** – Client libraries for implicit parallelization

System Components

Client:

- Submit compute tasks to the scheduler
- Retrieve back the task results

Scheduler: The role that

- Route tasks to workers
- Retrying tasks if the worker has failed
- Balance the task loads across workers based on different strategies

Worker:

- Announce what type of works it can process
- Actual execute tasks
- Control the per worker resource (not exceeding the limits specified)

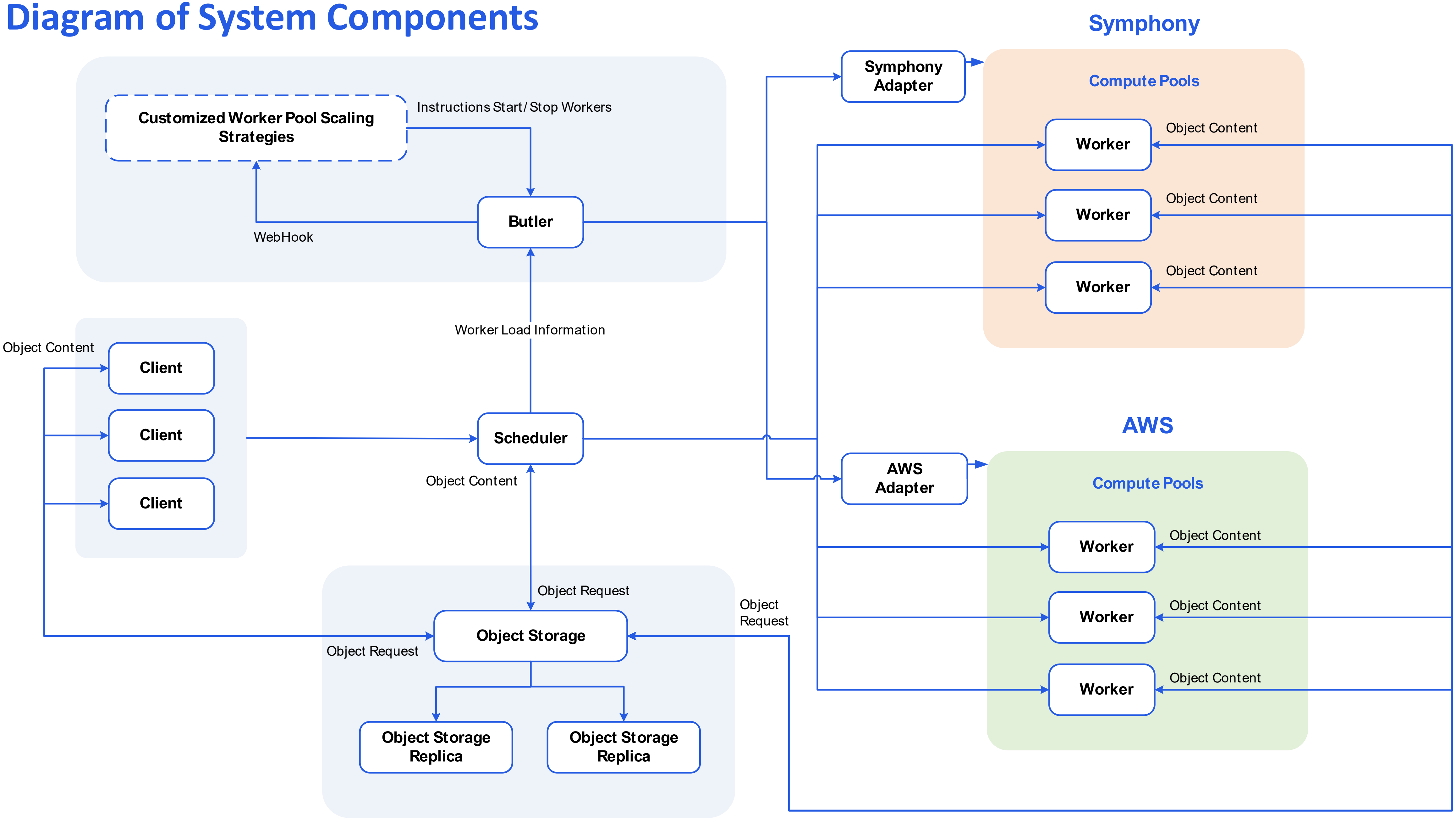
Object Storage:

- Store serialized objects needed for task and task results in the task life cycle
- Can be replicated and close to client/scheduler/worker for faster access

Butler:

- Monitor worker pools workloads globally
- Send instructions to different pool to elastic manage compute resources

Diagram of System Components



System Component Interactions

	Client -> Scheduler	Scheduler -> Client	Scheduler -> Butler	Butler -> Scheduler	Scheduler -> Worker Pool	Butler -> Worker Pool	Worker Pool -> Butler
Transport Protocol	✓	✓	✓	✓	✓	✓	✓
Task Scheduling (within pool)							
Scaling (Elastic Computing)			✓		✓		
Compute Resource Management			✓		✓		
- Create/Stop Pods			✓		✓		
- Time Based		✓					
Pool Pod Policies							
- Cost Efficient	✓						
- Speed Efficient	✓						

What primitives will this standard support:

Tasks and Graphs

- Each task is unit that has necessary information to run on remote with inputs and results
- Graph is series of tasks that has dependencies

Objects Storage

- Object Storage managed serialized blobs associated with tasks, tasks itself holding references of serialized blobs

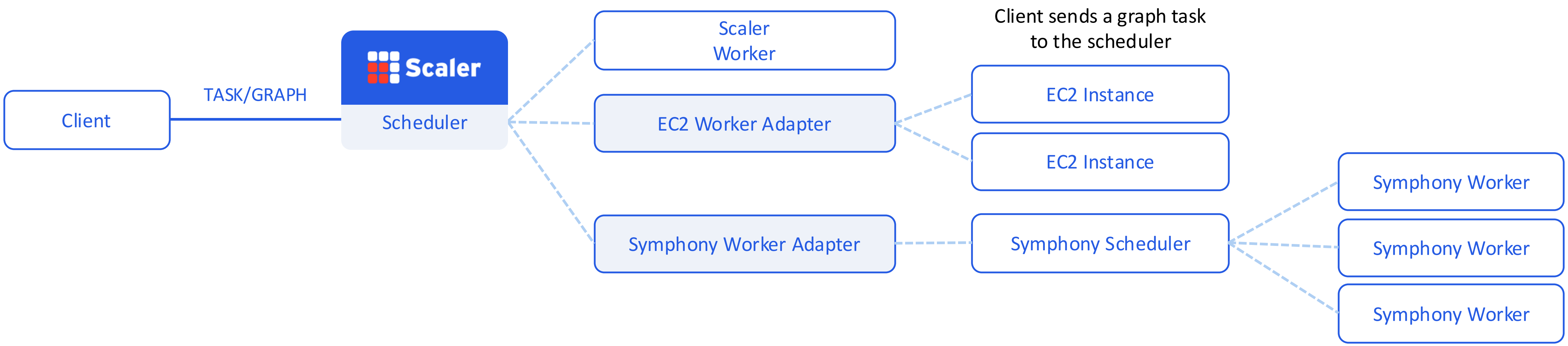
Tagging

- Tags are populated by worker/worker pools to indicate what supports they have
- Clients can specify tasks needs run on workers with special tags, scheduler will routing to the correct tags

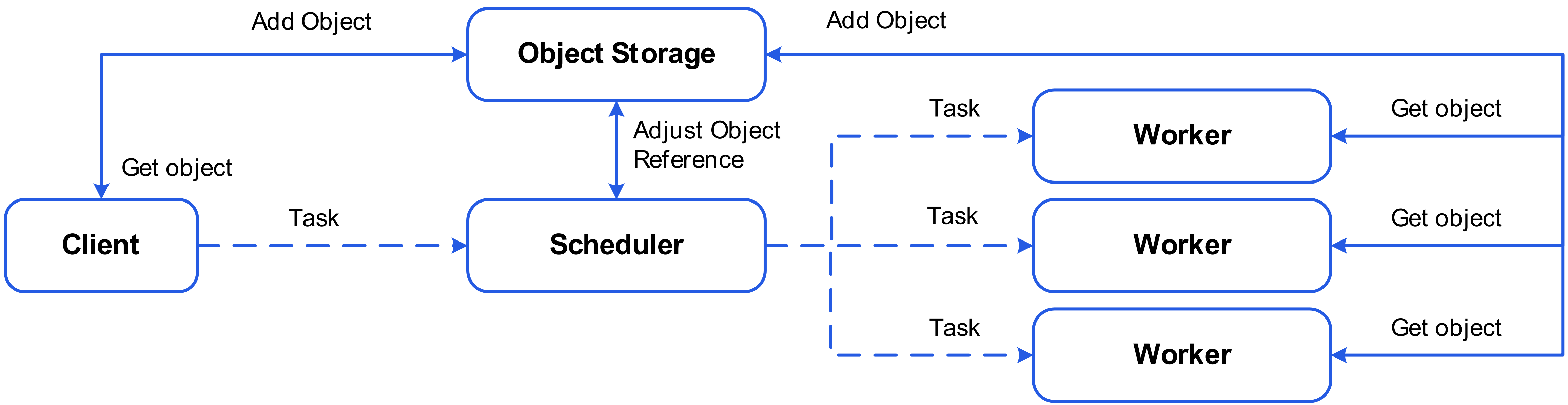
Compute resource control primitives

- This is to unify and simplify the interface that start or stop the workers in worker pool, the key primitives to elastic allocate compute resource

openGRIS on Heterogenous Compute Backends



Object Storage



Revisit Case Study: Enterprise Compute

Long running compute on thousands of cores (simulation, stress testing)

Case Study:

A market risk group at a large financial institution typically has grids that are used between the hours of 5P-11P and is mostly unused the rest of the day. These grids can cost more than $50MM a year and the low utilization also has negative environmental impact.

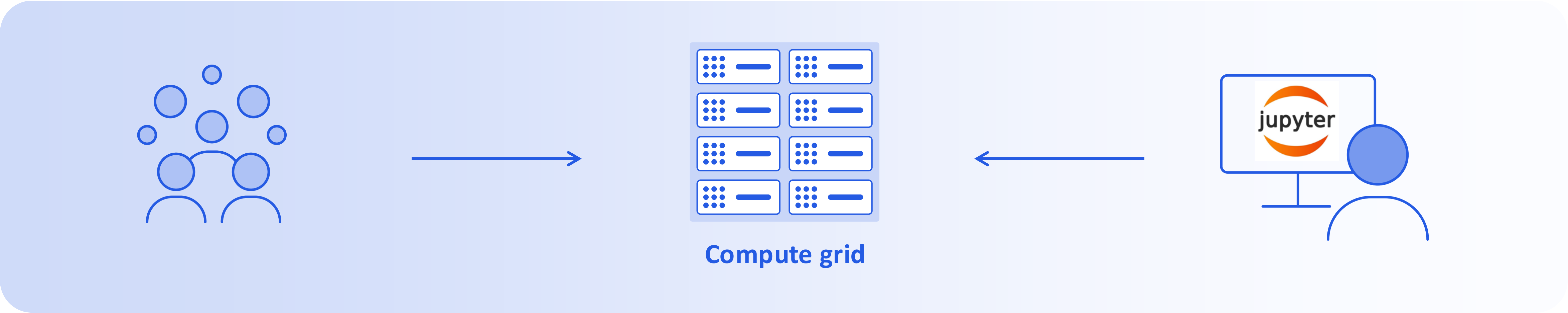
Someone sitting at desk want access on the fly (model development?)

Case Study:

A quant in a modeling team wishes to run an ad-hoc pricing or analytical run on a portfolio. The quant has to spend several hours setting up the environment on AWS and provisioning the cores.

Large compute

Small compute



Revisit Case Study: Enterprise Compute

Long running compute on thousands of cores (simulation, stress testing)

Case Study:
A market risk group at a large financial institution typically has grids that are used between the hours of 5P-11P and is mostly unused the rest of the day. These grids can cost more than $50MM a year and the low utilization also has negative environmental impact.

Large compute

With **traditional** solution, IT pre-allocates the grid for peak usage, which contributes to significant usage inefficiencies, and they must write code to AWS/Azure/etc, leading to vendor lock-in and static grid

With the **openGRIS** standard, users can access the different grids to allocate time, reducing overhead and improving efficiencies, and users can leverage multiple providers.

Revisit Case Study: Enterprise Compute

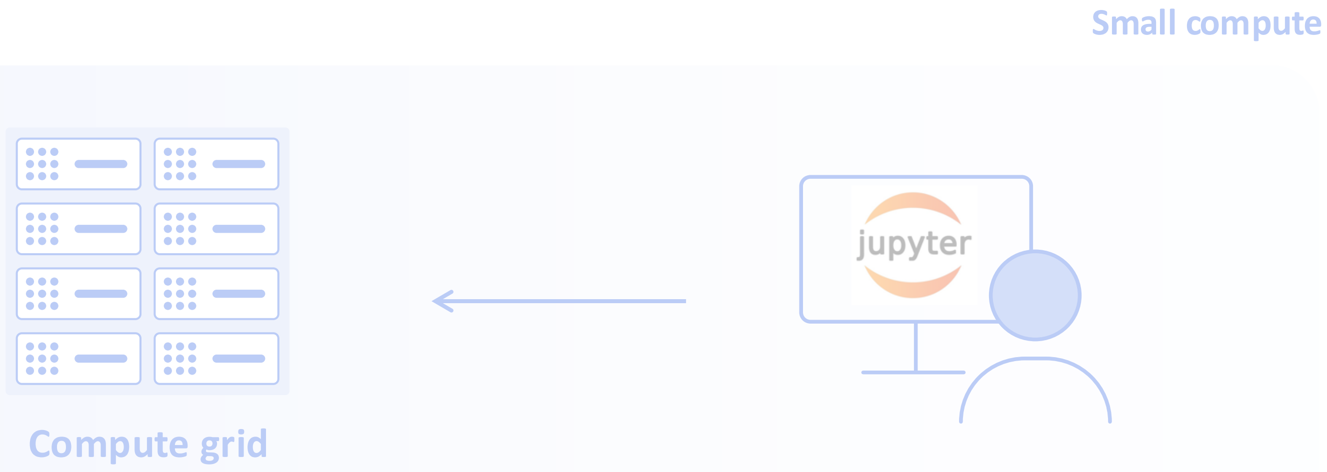
With **traditional** solution, most quants cannot access large grids or AWS because overhead for deployment is large for small quant teams. Simulation might take hundred cores for 5 min, but because the overhead accessing is very expensive, quant teams will not be able to use it.

With **openGRIS** standards, a quant can access the grid or AWS at any time with minimal lines of code. The openGRIS project has demonstrated a working prototype. By adding 3 lines of code, a quant team can enable grid or AWS access from Jupyter notebook, highlighting engineering efficiency for quant teams

Someone sitting at desk want access on the fly (model development?)

Case Study:

A quant in a modeling team wishes to run an ad-hoc pricing or analytical run on a portfolio. The quant has to spend several hours setting up the environment on AWS and provisioning the cores.



Roadmap

Category	Milestone	Target Date	Status
Proposal	Standard 1.0	6/30/24	Done
Implementation	Reference Implementation	7/30/24	Done
Implementation	Open Source Scaler	9/30/24	Done
Proposal	1.0 Discussion with Nvidia, AWS, Intel, IBM	9/30/24	Done
Implementation	Interface to IBM Symphony	1/30/25	Done
Proposal	Formalize Standard 2.0	2/15/25	Done
Implementation	EC2 Static Compute Adapter	7/30/25	Pending
Implementation	Multi-language bindings: C++	8/30/25	Pending
Implementation	Time based Policies for Scheduling	9/30/25	Pending
Implementation	Dynamic Scaling for Symphony and EC2	10/30/25	Pending
Implementation	Multi-Environment support via Docker/Podman	11/30/25	Pending
Implementation	AWS HTC Grid Adapter	TBD	Pending