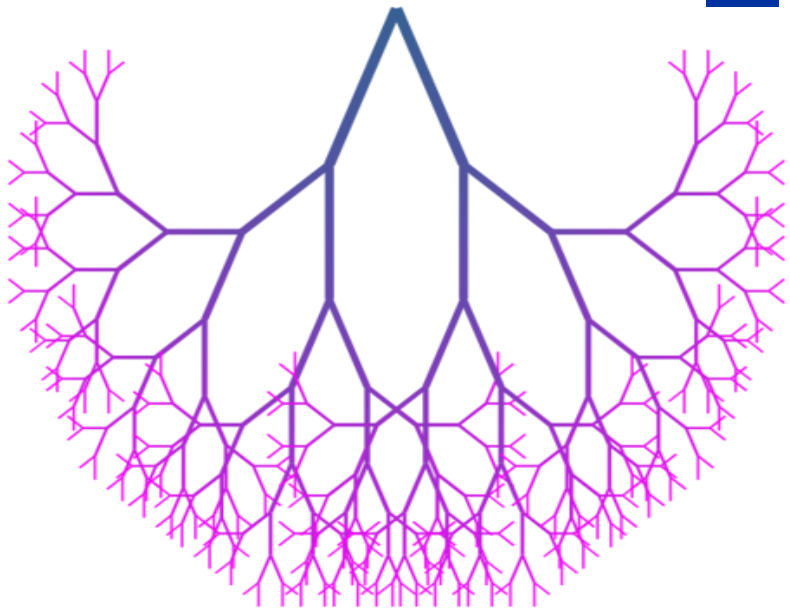


Flux, a next-generation resource management framework

April, 2026

Tom Scogland, Flux Lead

With work from the whole flux team

Prepared by LLNL under Contract DE-AC52-07NA27344.

What is Flux Framework About?

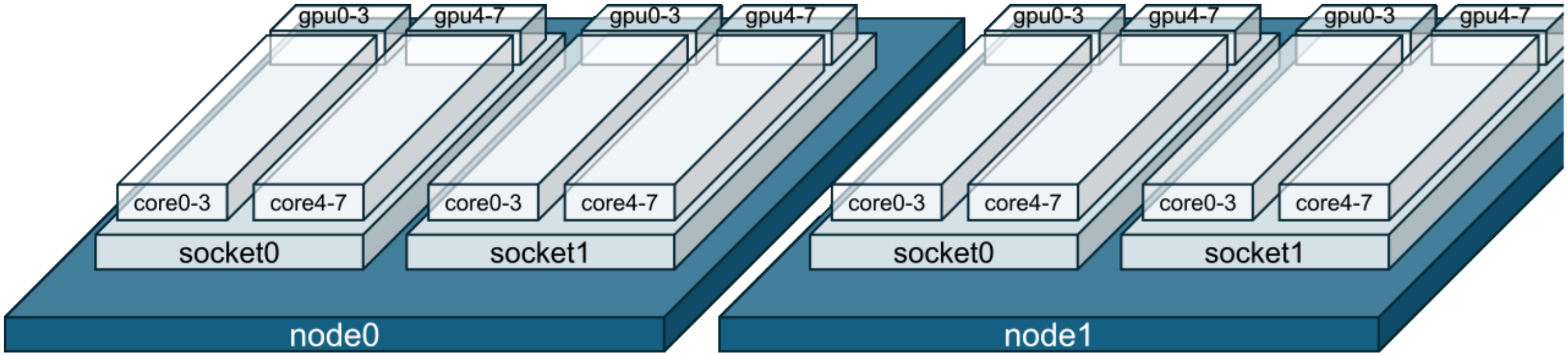
Flux Framework is the next-generation workload manager and resource orchestration suite designed for modern, complex, and heterogeneous high-performance computing (HPC) environments.

Backed by expertise

Flux has 25+ contributors, including principal engineers behind Slurm, and developers with expertise in container technologies, scheduling, workflows, and cloud computing.

What is Flux Framework About?

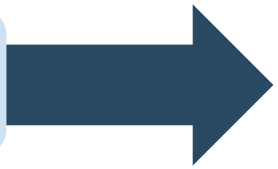
- Hierarchical scheduling breaks the traditional monolithic scheduler bottleneck by distributing scheduling decisions and managing resources at multiple levels.



What is Flux Framework About?

- Graph-based resource model to natively expresses and control complex hardware topologies and devices (like GPUs).

```
flux run -N 6  -n 216 ./app
```



```
version: 1
resources:
  - type: cluster
    count: 1
    with:
      - type: rack
        count: 2
        with:
          - type: slot
            label: myslot
            count: 3
            with:
              - type: node
                count: 1
                with:
                  - type: socket
                    count: 2
                    with:
                      - type: core
                        count: 18

# a comment
attributes:
  system:
    duration: 3600
tasks:
  - command: app
    slot: myslot
    count:
      per_slot: 1
```

How is Flux Framework Used?

Flux is built to solve the resource management challenges of the post-Exascale era, bridging the gap between traditional batch processing and modern, dynamic workflows.

- **Exascale & Top 500**: NNSA exascale system El Capitan and other Top 500 supercomputers.
- **Converged Computing (HPC + Kubernetes)**: fully functional batch schedulers inside Kubernetes pods, bypassing etcd bottlenecks (Flux Operator).
- **Complex, heterogeneous workflows**: oriented for workflows where simulation and AI/ML components must coexist and share resources efficiently.
- **Graph-based modeling**: handles complex hardware topologies, ensuring the right job gets the right hardware (e.g., GPUs, interconnects) for existing and future hardware.
- **Systems research and innovation**: prototype novel scheduling algorithms.

How did Flux come to be?

2012 — Project Established
Initial project funding proposal and staffing

2013 — Design
Preliminary design review

2014 — Instance Launch
Prototype Flux instance that could be launched by Slurm/mpiexec

2016 — Flux Core 0.1.0
First public release of Flux Core

2018 — Science
Flux used by Pilot2 cancer moonshot ensemble

2019 — Best Paper
Flux used by MuMMI, SC19 Best Paper Award winner

2020 — System Instances
Release of Flux Core with system instance support

2021 — MuMMI Workflow
Flux+MuMMI Integration used for RAS-RAF membrane dynamics ensemble

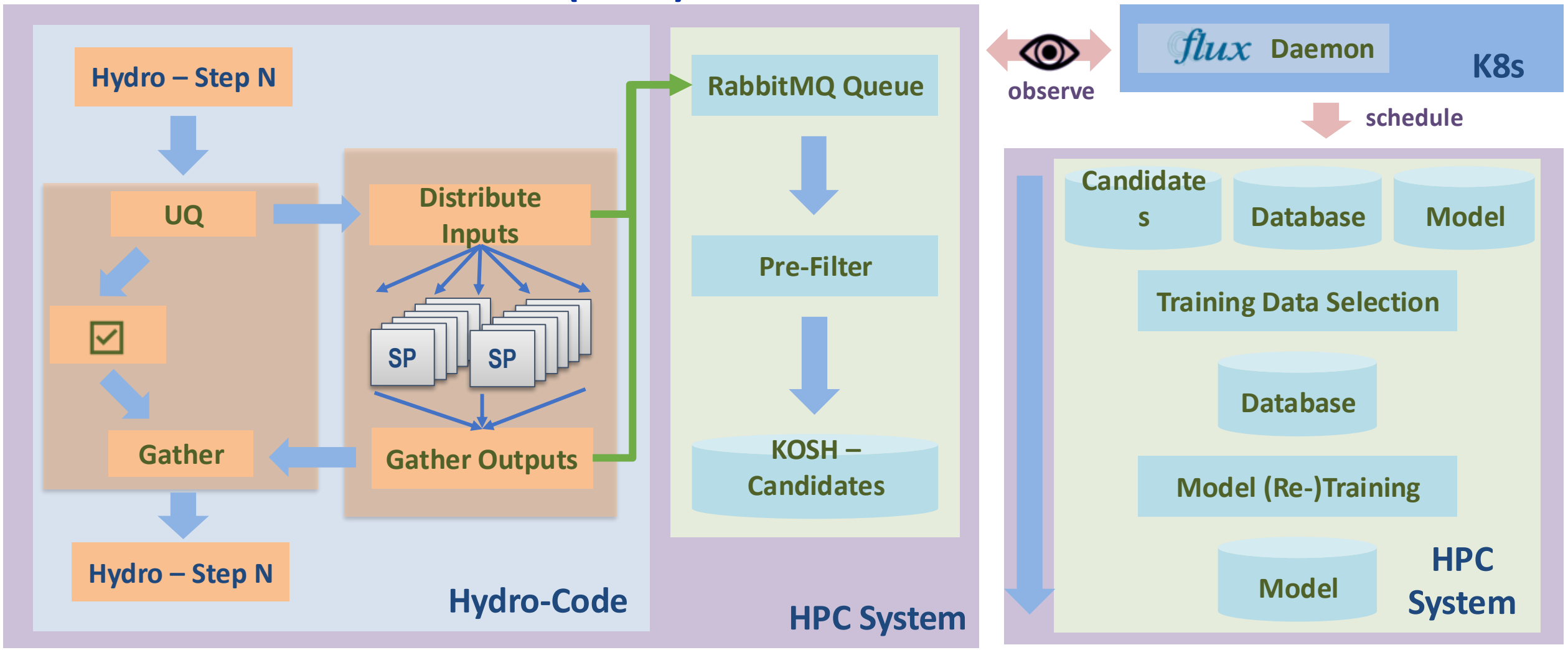
2024-2026 — El Capitan
Flux system instance deployed on El Capitan, #1 on Top500 (-2026)

...

Why build another one?

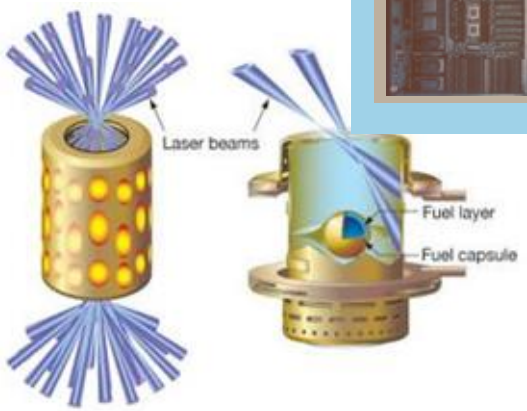
Or: Why not just use {slurm,PBS,LSF}?

Autonomous Multiscale (AMS) Workflow



https://github.com/LLNL/AMS

https://www.osti.gov/servlets/purl/1630805



flux

redis
Merlin
RabbitMQ

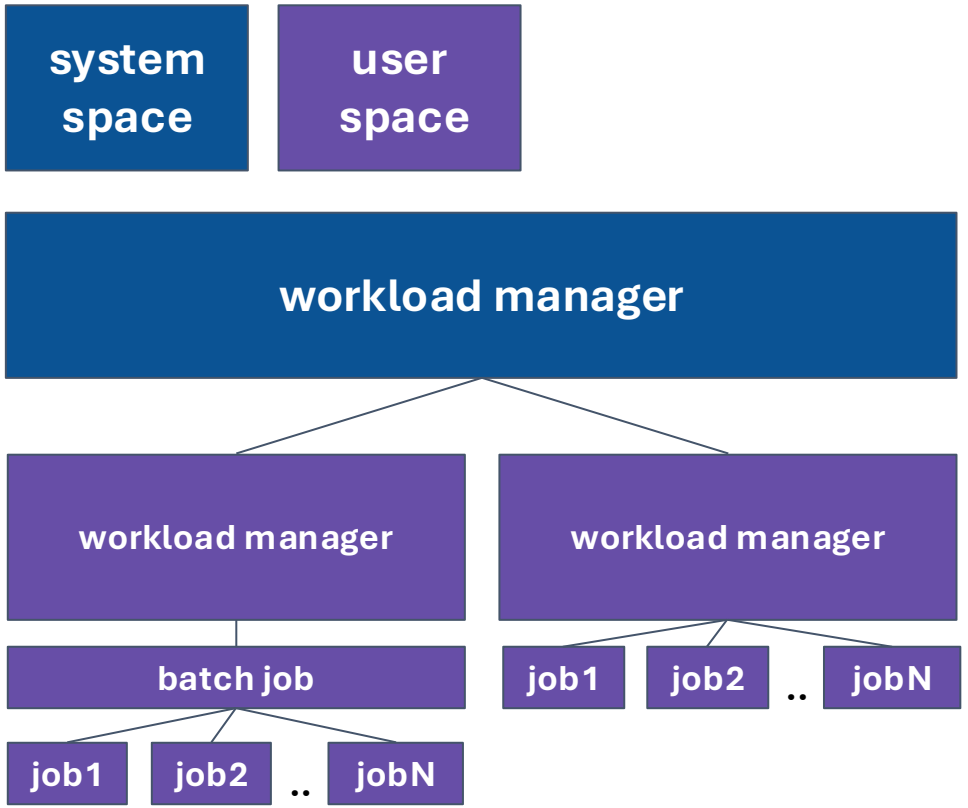
Combination of 1024- and 512-node jobs (blue subsquare is 32 nodes), 61,440 concurrent workers

1. Emerging workflows and systems need a flexible resource model

- Resources like network, power, storage, and **<insert future resource>** are not easily integrated into existing schedulers.
- A *flexible resource model* can express arbitrary resources, including those that don't exist yet.

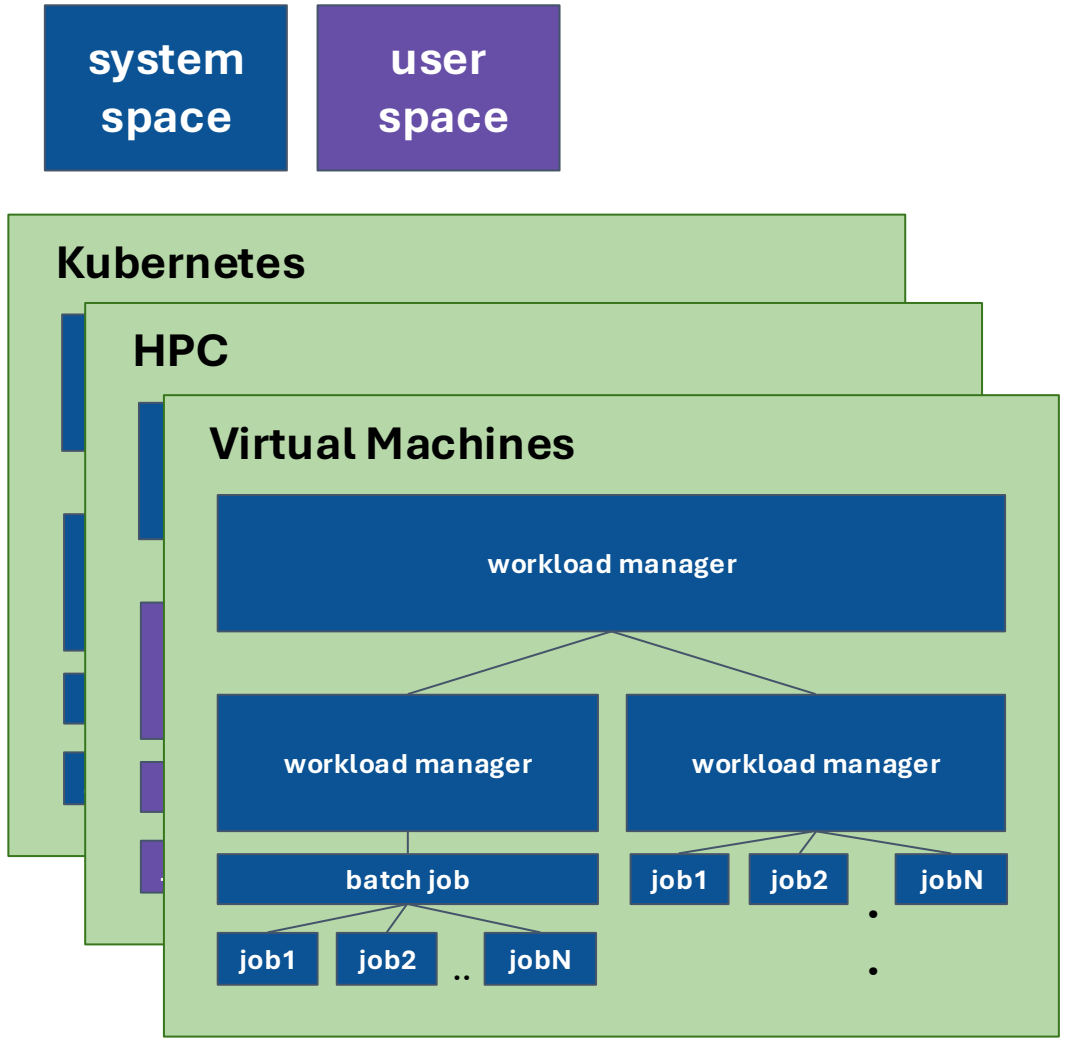
2. The hierarchical design facilitates portability

- A resource manager can be instantiated *as a job* under another resource manager
- The resource manager itself can be a portability layer.



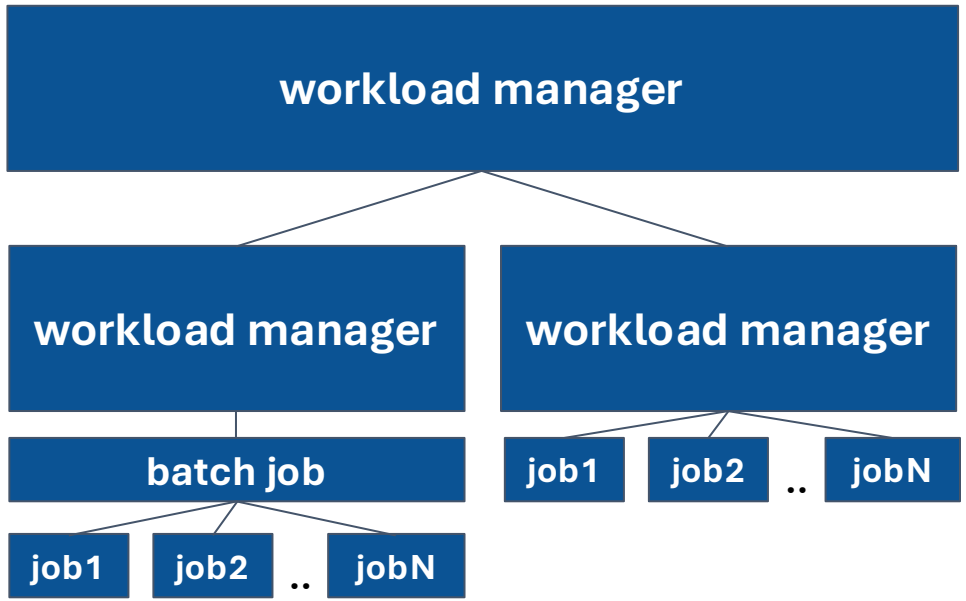
2. The hierarchical design facilitates portability

- A resource manager can be instantiated *as a job* under another resource manager
- The resource manager itself can be a portability layer.



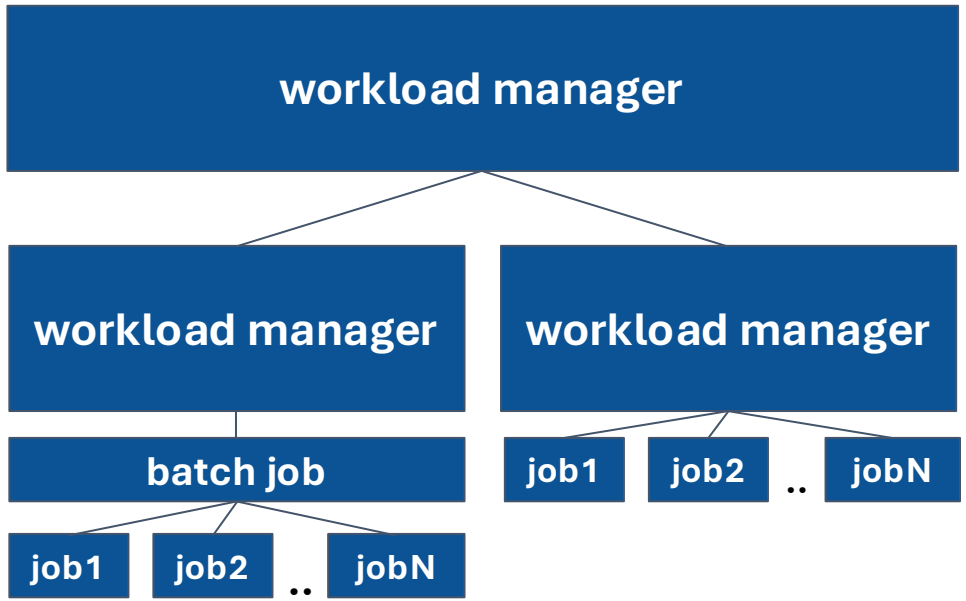
3. The hierarchical, modular design enables scheduler configurability

One scheduling algorithm to rule them all (what about queue)?



3. The hierarchical, modular design enables scheduler configurability

One scheduling algorithm to rule them all (what about queue)?

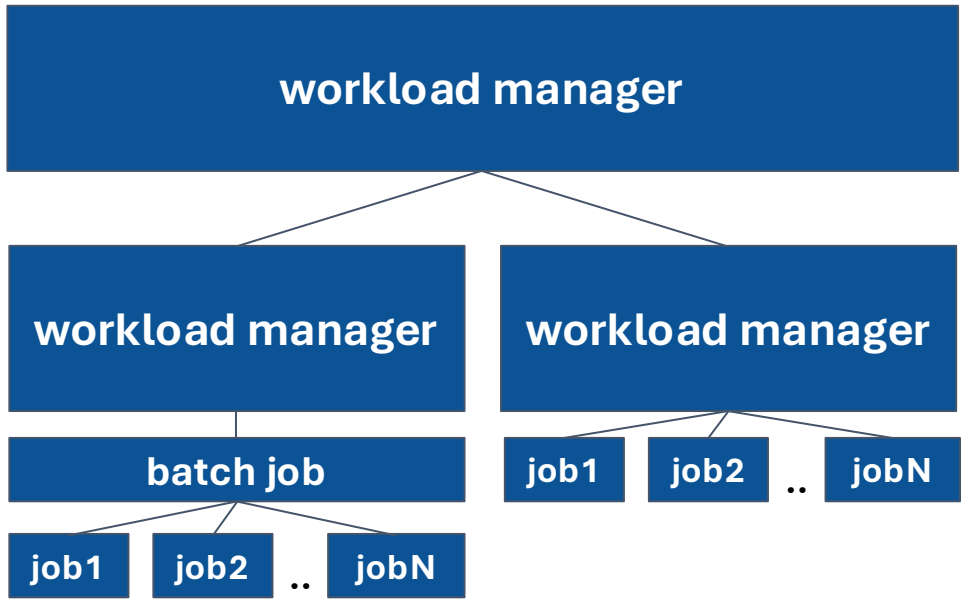


Did you know!

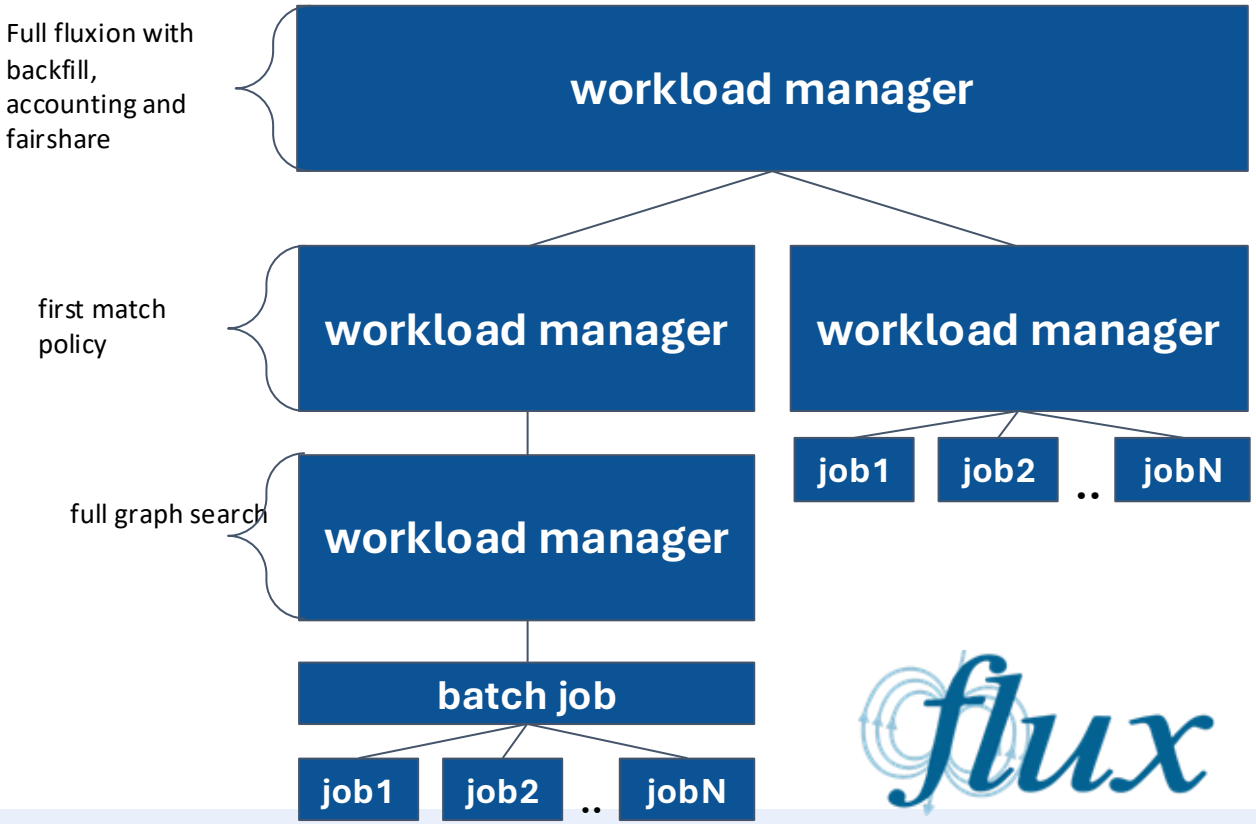
A **policy** can dictate how to schedule. Should we stop at the first match? Continue until we find all matches? Use a different algorithm?

3. The hierarchical, modular design enables scheduler configurability

One scheduling algorithm to rule them all (what about queue)?

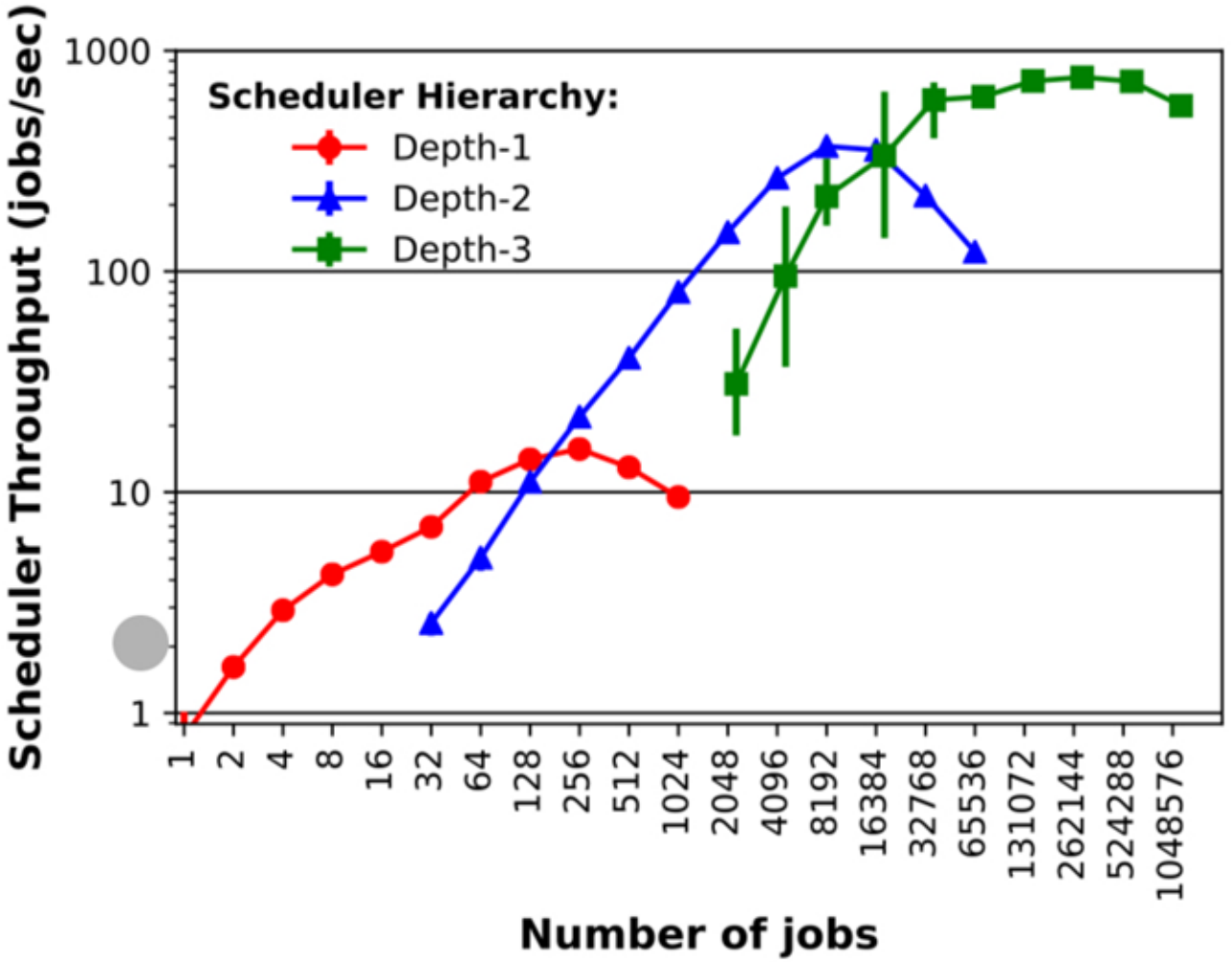


Choose an algorithm for each level (instance, or sub-graph) in the hierarchy.



4. The design enables increased scalability and throughput

Fractal scheduling enables high throughput



- Flux addresses demand for high throughput
- A hierarchy of Flux instances managing jobs increases throughput.
- We don't face the bottleneck of one, central scheduler

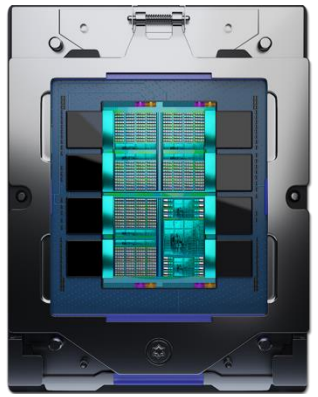
Ok, but how can we extend it?

Flux's interfaces are designed to be extensible

- Most of flux itself, and all of the framework projects, are extensions!
- From the bottom up:
  - Broker modules: Things like the fluxion scheduler and accounting module
  - Jobtap plugins: Direct feedback and control *in the job management loop*
  - Validator and Frobnicator plugins: Check or modify incoming jobs
  - Extending the Flux command line
  - The usual (prolog and epilog scripts)
  - User APIs

So, how do I customize it?

- ”I need to support a specific type of hardware”
- “I need to track a specific type of user’s job”
- “I want to add shortcuts for commonly-used submission combinations”



AMD MI300A can be split into multiple CPU/GPU configurations

```
(s=1153,d=0)  tuolumne2152 ~ $ flux alloc -N 4 --setattr=gpumode=TPX -o mpibind=on --env=MPIBIN
D_TOPOFILE=/tmp/hwloc.xml script.sh
```

“This is too much typing! Help!”

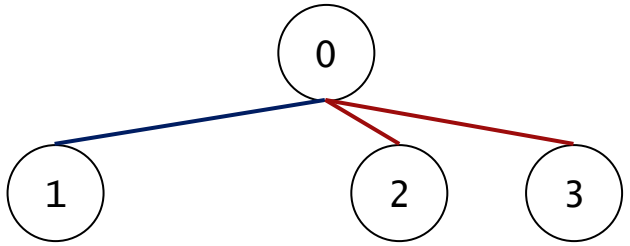
All of these customization interfaces can (with varying difficulty) be written in Python.

Why would I want a broker module?

- Modules are the core of flux, and can do nearly anything
- Things built as modules:
  - Both flux schedulers, sched-simple and fluxion
  - flux-accounting
  - The central key-value-store
  - The job-manager
- They are dynamic libraries or python files with full access to flux's messaging system

Modules are driven by messages (and callbacks)

job ingest module on login node



User on a login node sends a message

- *"Hey! I want to run a job!"*

Module on node picks up message, routes it appropriately

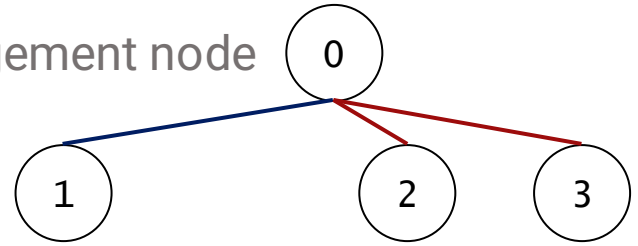
- *"This user wants to run a job! The job-manager should know!"*

The job-manager fulfills the appropriate callback for the specific message

- *"This user from node 1 wants to run a job on node 3, but I'm node 0. Phone my friends!"*

Modules are driven by messages (and callbacks)

job manager module on management node



User on a login node sends a message

- *"Hey! I want to run a job!"*

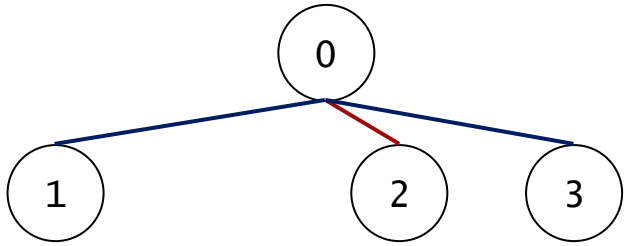
Module on node picks up message, routes it appropriately

- *"This user wants to run a job! The job-manager should know!"*

The job-manager fulfills the appropriate callback for the specific message

- *"This user from node 1 wants to run a job on node 3, but I'm node 0. Phone my friends!"*

Modules are driven by messages (and callbacks)



dispatches the work to compute nodes

User on a login node sends a message

- *"Hey! I want to run a job!"*

Module on node picks up message, routes it appropriately

- *"This user wants to run a job! The job-manager should know!"*

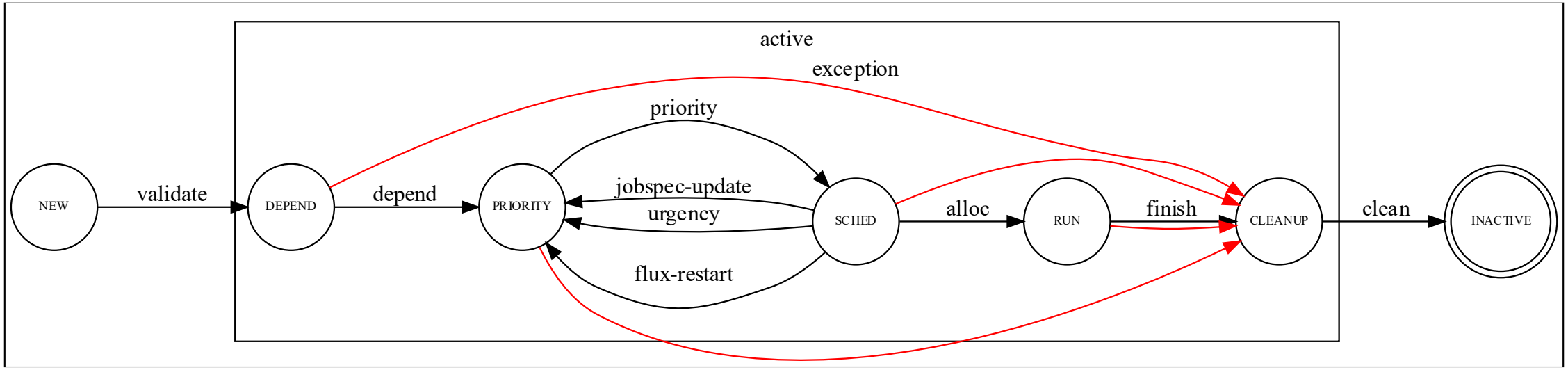
The job-manager fulfills the appropriate callback for the specific message

- *"This user from node 1 wants to run a job on node 3, but I'm node 0. Phone my friends!"*

Why would I want a jobtap plugin?

- Jobtap plugins run *inside* the job manager, and can change its behavior
- Things you can build with this
  - A plugin to dispatch work to be serviced by another instance of flux
  - A new type of dependency between jobs
  - Anything that needs to make decisions on job-state changes that block the job manager waiting on their decision
- Jobtap plugins are libraries with a C ABI, should be quick to respond, but are *very powerful*

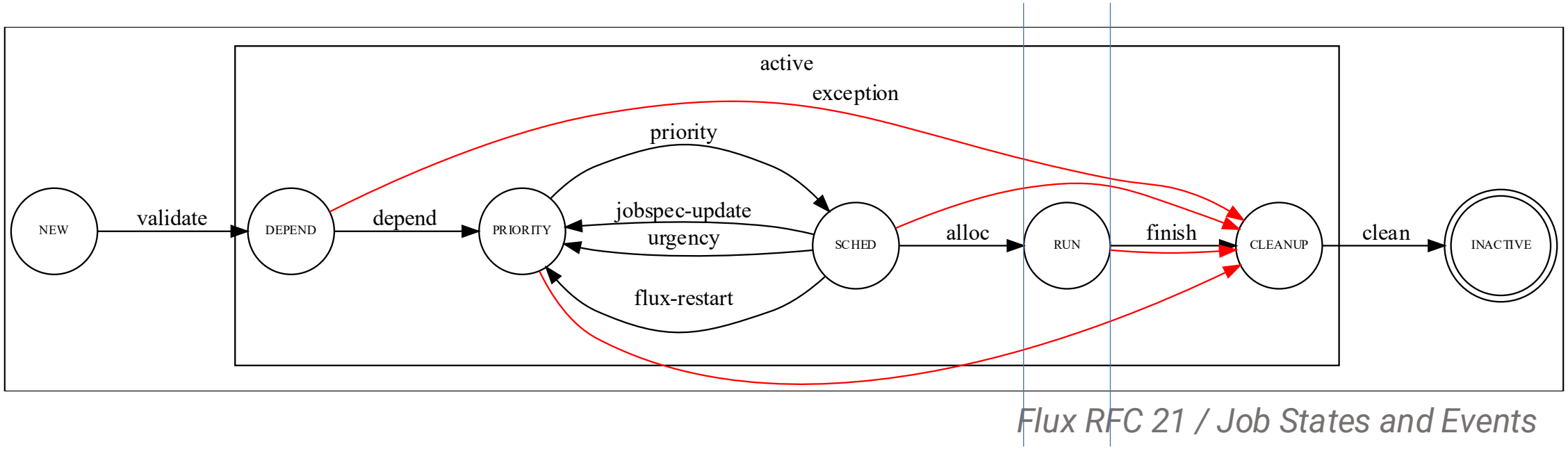
Jobtap Plugins: “tapping in” to job state



Flux RFC 21 / Job States and Events

- One module, the job manager, is involved in each *state transition* in a Flux job.
- The job manager allows callbacks to be invoked on each transition.

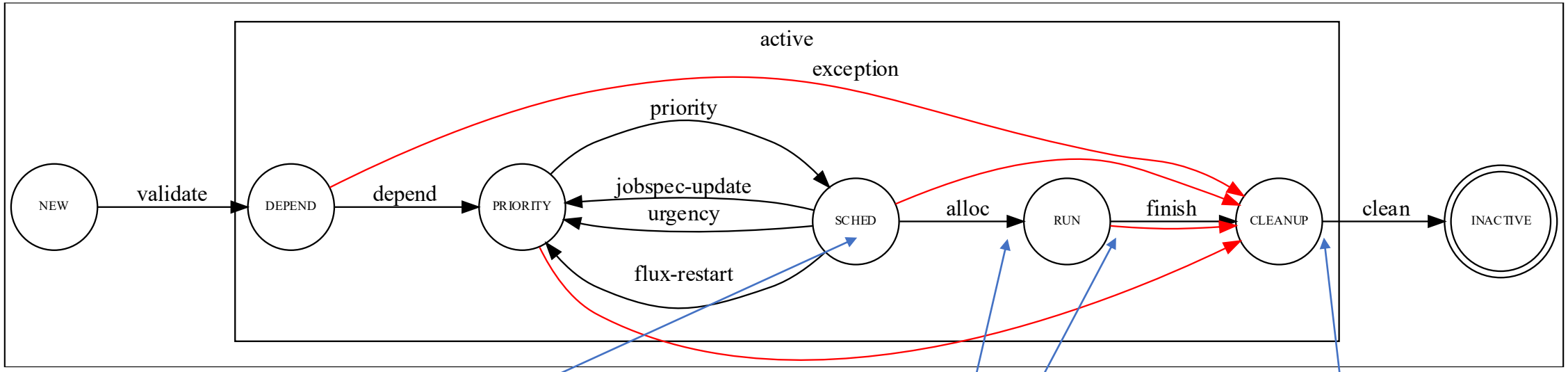
`sudo` actions happen before/after the RUN state



Flux RFC 21 / Job States and Events

- In addition to callbacks, before and after a job RUNs, an elevated privilege “prolog” and “epilog” runs on each node in the allocation for system services

Example use for plugins: Cray VNI support



Flux RFC 21 / Job States and Events

VNI ”tag” issued (round robin 64k integer) through a “jobtap plugin” (event callback)

VNI ”tag” released back into Flux-managed pool of tags

CXI service started via prolog / shut down via epilog

Why would I want a validator or frobnicator plugin? (and what on earth is a *frobnicator?*)

- Validator plugins… *validate*
  - Use them to take in jobspec (the complete description of a job request) and accept or reject it
  - Good for site-specific rules with complex logic
  - Example: Reject jobs *below* a certain size
- Frobnicator plugins take jobspec and *modify it* before it's signed
  - Good for adding requirements or modifying limits on all jobs submitted
  - Example: Add **requires: !hosts:donotuse[1:5]** to all incoming jobs

Prolog, epilog and housekeeping

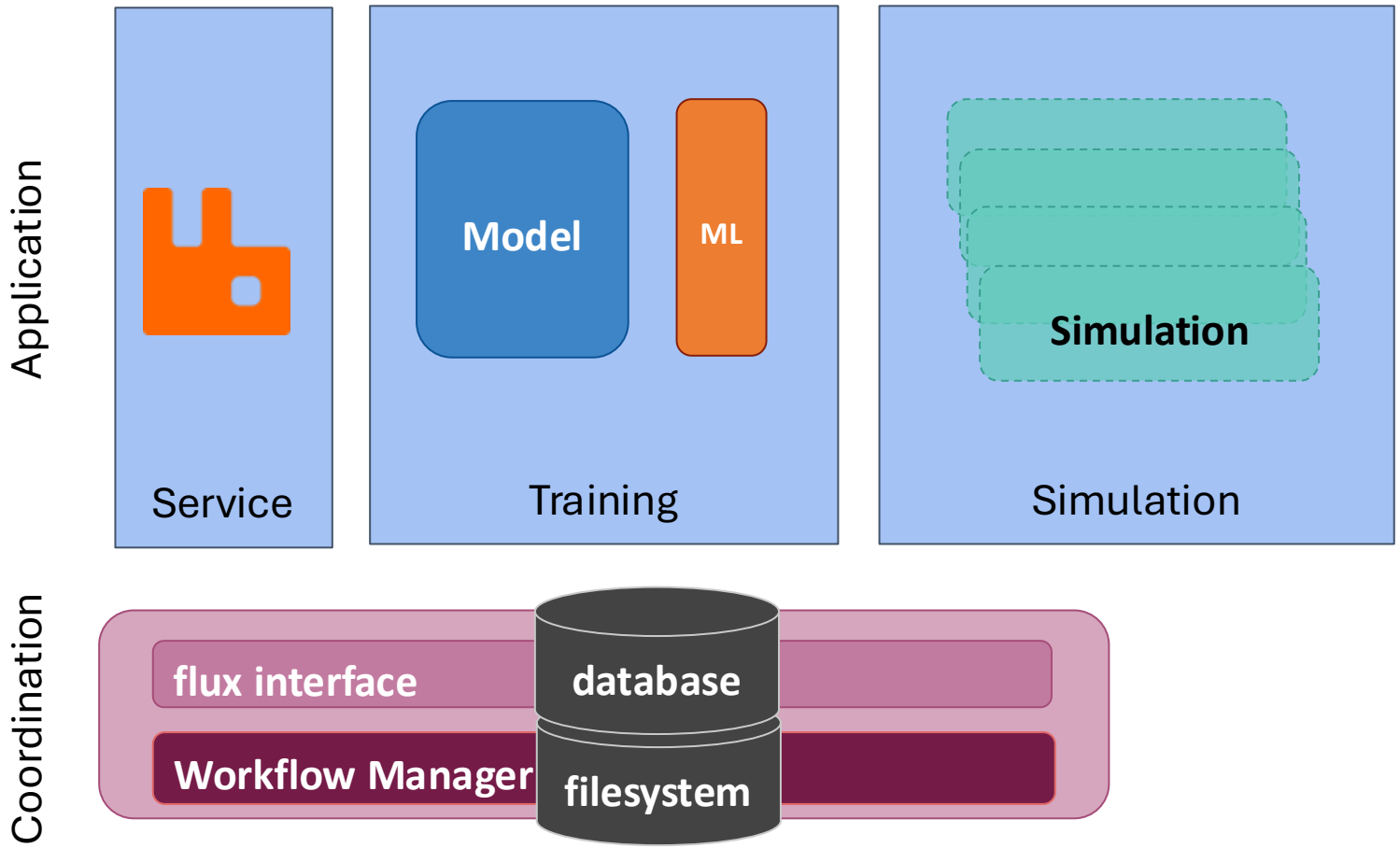
- In a system instance
  - Business as usual, runs as a privileged user:
    - Prolog: Before the job
    - Epilog: After the job *before the finish event*
    - Housekeeping: After the finish event, before releasing to the scheduler
- In a user instance
  - All the same still exist, but run as the user owning the instance
  - Sub-instances do not run the prologs/epilogs of the system instance by default

User APIs

- Not strictly “extensions”
- Users can interface with flux internal interfaces through:
  - Python
  - Lua
  - C
- Asynchronously submit jobs without needing **fork**
- Get callbacks on job state transitions
- Main use: Stop writing schedulers for workflow managers!

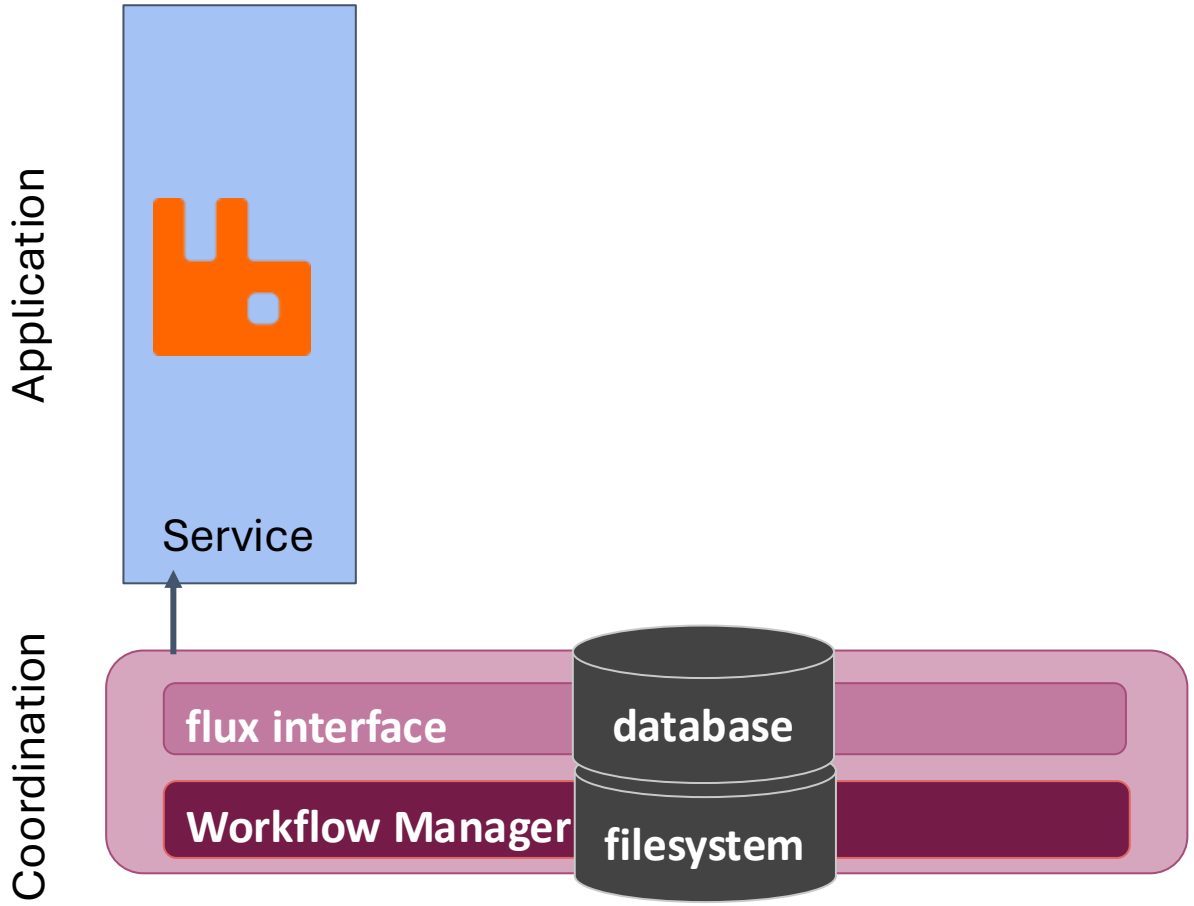
Flux APIs integrate with workflow managers

How might a workflow manager run this?



Flux APIs integrate with workflow managers

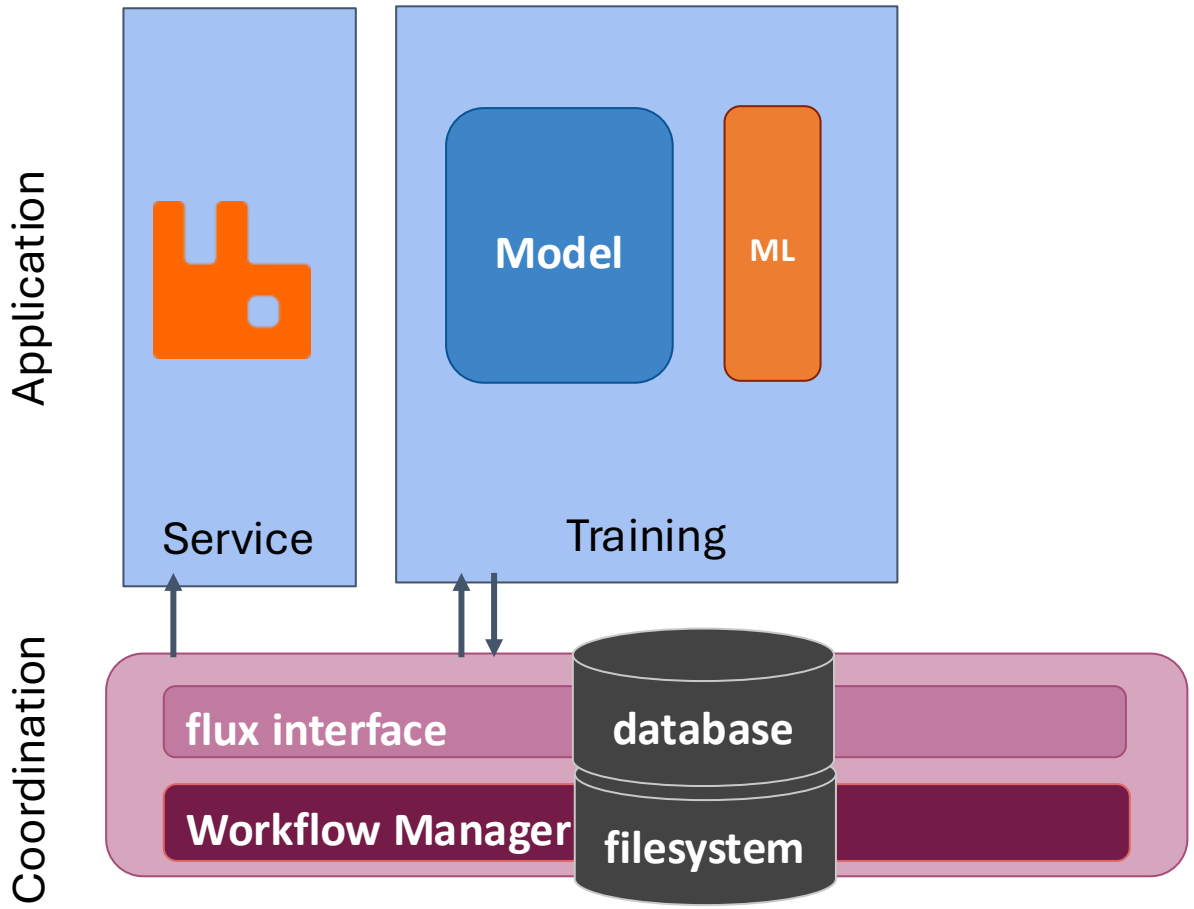
First submit a job for the service (not blocking)



flux submit sh service.sh
flux run -N4 -g1 python train.py
flux submit –wait -N4 python simulation.py

3. Flux APIs integrate with workflow managers

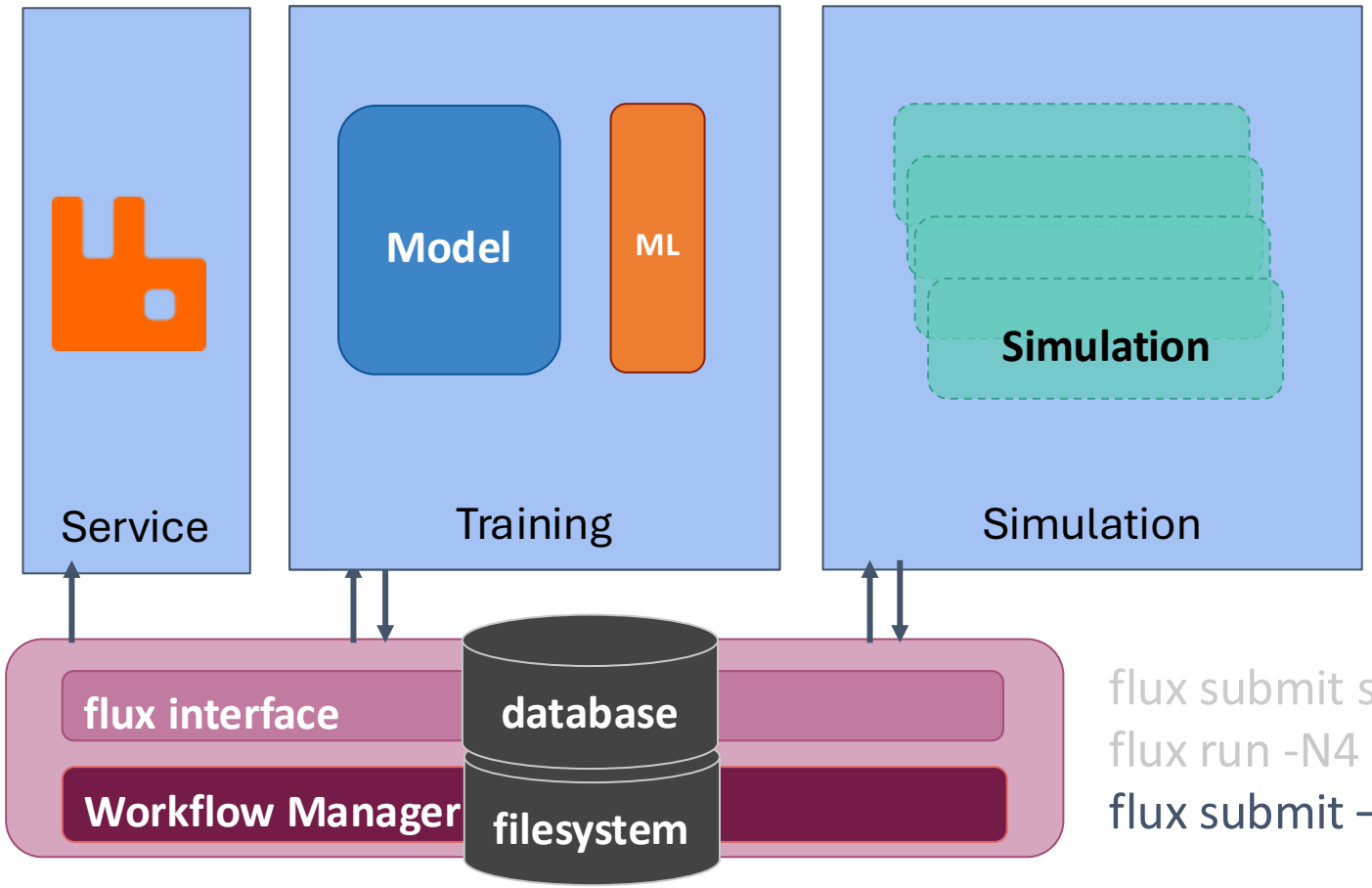
Next run training - we need to train *before* we run simulations, so block



flux submit sh service.sh
flux run -N4 -g1 python train.py
flux submit –wait -N4 python simulation.py

3. Flux APIs integrate with workflow managers

Finally, finish with simulations. Add "wait" so we wait until they finish!



flux submit sh service.sh
flux run -N4 -g1 python train.py
flux submit –wait -N4 python simulation.py

Which workflow and orchestration tools support Flux?

Streamflow

What have we been working on?

- **Support for grow/shrink of jobs (flux-sched):** Allows the underlying resource pool and active jobs to dynamically adjust their resource allocations mid-run.
- **Custom Resource Module Policies (flux-sched):** Provides flexibility for users or administrators to define their own resource management logic.
- **More Options for Generating Taskmaps:** Allows for fine-tuned control over process placement and task distribution on heterogeneous hardware.
- **Resource/Job Events Journal Consumer Interfaces (Python API):** programmatic access via Python to stream and consume real-time job and resource events. Includes utility for users to post events.
- **Interfaces for Watching and Querying Jobs and Queues:** Standardized methods for tracking job status and queue state, and queue metadata (e.g., runtime statistics).
- **Command flux update to change job duration after submit:** Allows users to modify job wall-time limits post-submission.
- **Optimizations:** ongoing to maintain performance on exascale-class machines.
- **Relaxation of scheduling constraints from data dependencies:** exposes further opportunity to optimize scheduling with data dependencies across/within jobs while allowing to leverage local storage

What's next?

El Capitan has helped us prov that a system-level Flux at the ~13,000 node scale is both possible and stable, but we're always working to improve things:

- **Rolling software upgrades:** Current upgrades to the Flux system instance require downtime and a full restart of the instance, losing running jobs.
- **Restart with running jobs:** restarting any Flux instance currently kills running jobs.
- **Reservation support:** support for deferring a job/reservation start time into the future, which is currently supported with admin intervention, scripting and queues.
- **Package availability:** Source RPMs are published with each tagged release, although `dnf install flux-core` is not currently available. Collaboration is underway with Fedora/Red Hat for inclusion in their Extensions Repository.

Thanks to everyone on the Flux team over the years!