Reinforcement Learning for PanDA Job Scheduling

A data-driven PanDA simulation environment for safe policy training and evaluation

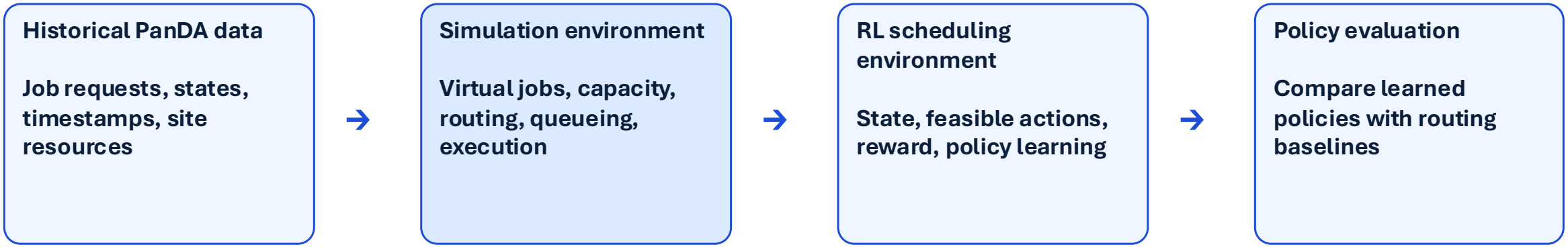
Tasnuva Chowdhury | October 12, 2026

Reinforcement Learning for PanDA Scheduling

Objective: Enable Realtime Optimization of PanDA Scheduling

Historical PanDA snapshot data supports a simulation environment where scheduling policies can train and be evaluated before testing with live data safely.

Each routing decision changes queueing and resource use across a distributed grid.



The simulator is the bridge between historical operations and safe RL experiments.

Historical PanDA Snapshot Data & Simulation Requirement

PanDA operates as a changing federated system.

- Sites differ in active workload, queue pressure, runtime profiles, and available capacity.
- The workload mix changes across processing types and global/local priority shares.
- State-aware scheduling can route work away from emerging congestion or constrained sites.

Research question

Can scheduling use this observed state to make better routing decisions for the next arriving job?

Complements CGSim

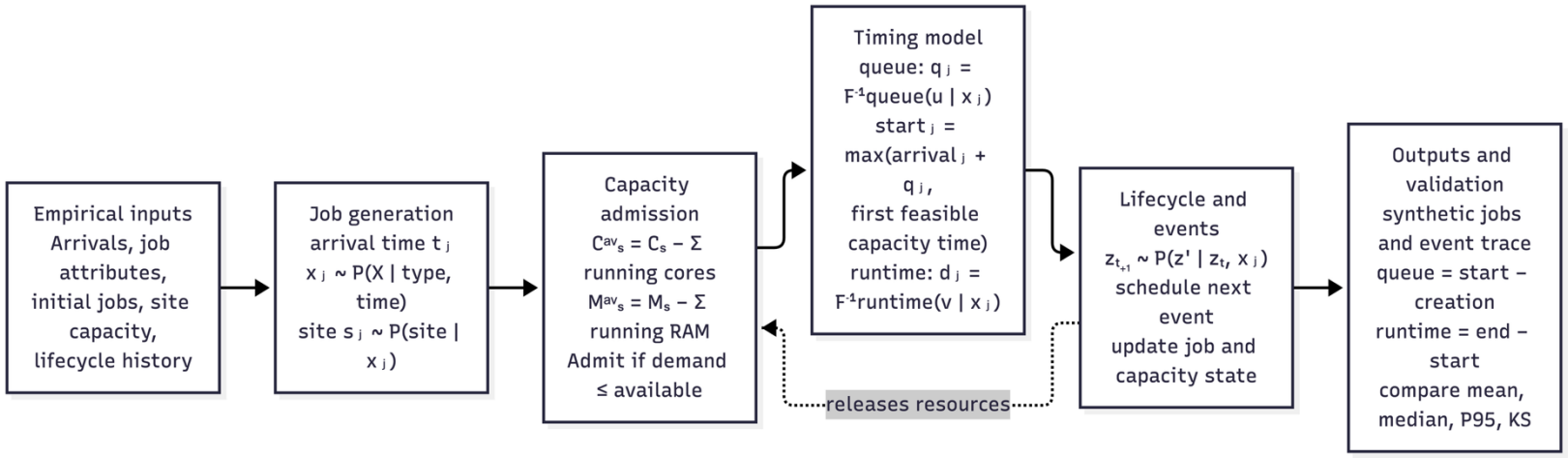
CGSim supports infrastructure-scale policy studies. This work adds a near-real-time operational view for short-horizon routing decisions from the observed PanDA state.

REDWOOD impact

A state-aware policy can route around emerging congestion or constrained sites, improving the resilience of federated workflows in a heterogeneous computing environment.

Data Driven Simulation Environment

The environment captures key scheduling dynamics while keeping its operational boundaries explicit.



Represents

- **Job arrivals and request attributes from historical data**
- **Capacity-limited site admission using cores and RAM**
- **Site eligibility, routing, queueing, execution, and resource release**
- **Empirical timing and lifecycle behavior**

Model boundaries

- **Exact future job composition remains unknown**
- **Some production brokerage logic remains unobserved**
- **Future site conditions cannot be predicted exactly**
- **Results support policy comparison, not direct deployment decisions**

The simulation environment learns observed job and site behaviors from these historical records.
Purpose: controlled comparison of scheduling policies under realistic data-driven conditions.

Two Ways to Use the Simulation Environment

The same capacity-aware environment supports synthetic workload generation and real-request backtesting.

Generative workload mode

Creates future job arrivals from historical distributions

Simulates job population, arrivals, routing, queueing, and execution

Use: Offline RL training and stress tests

Request-driven mode

Uses real submitted jobs (with processing type mixing) with known request-time attributes

Simulates uncertain later behavior: routing, queueing, execution, and resource release

Use: historical back testing and operational decision support

Generative mode provides broad training scenarios; request-driven mode tests decisions against known historical requests.

Simulator Validation Approach

Validation tests internal correctness and similarity to historical PanDA behaviour.

Structural checks

Unique job identifiers
Valid statuses and state order
Valid lifecycle timestamps

→

Capacity checks

Core and RAM capacity respected

Waiting when capacity is unavailable

→

Distribution checks

Queue time and runtime comparison
Median, mean, 95th percentile
Distribution distance

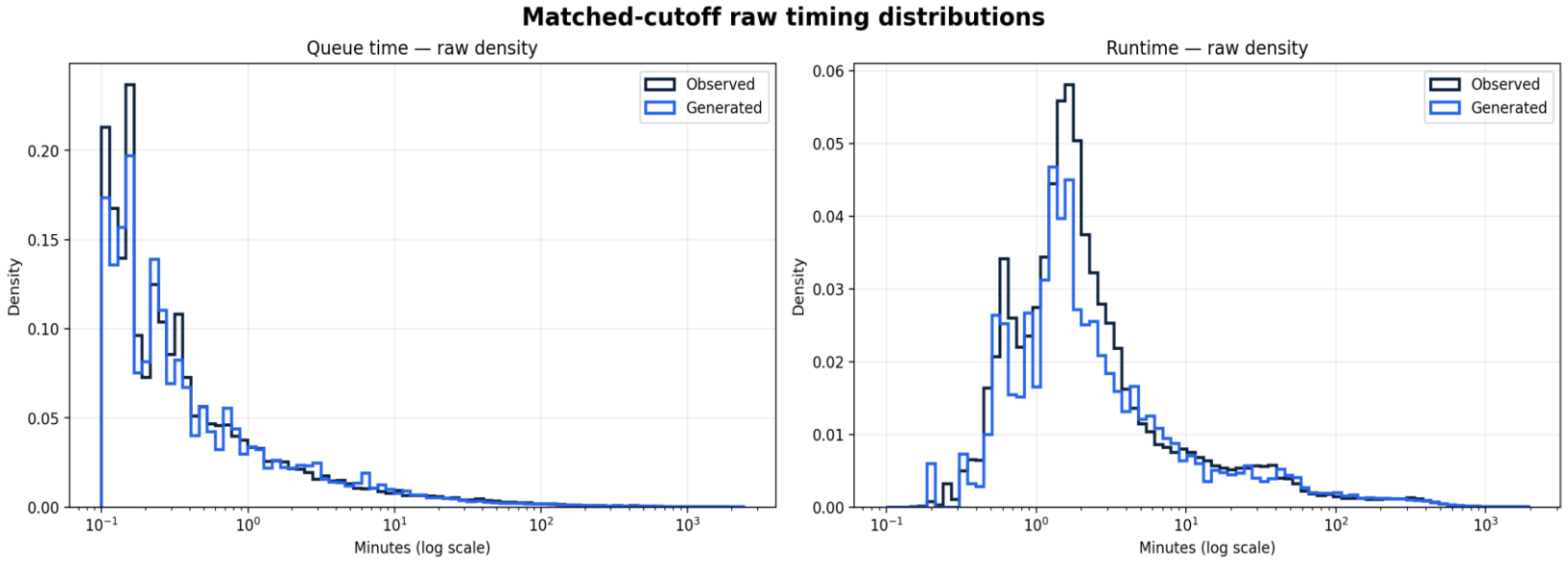
→

Population checks

Processing-type composition
Within-type timing comparison
Mixture-decomposition analysis

Population validation asks whether the generated workload resembles historical PanDA behaviour, not whether it repeats individual historical jobs.

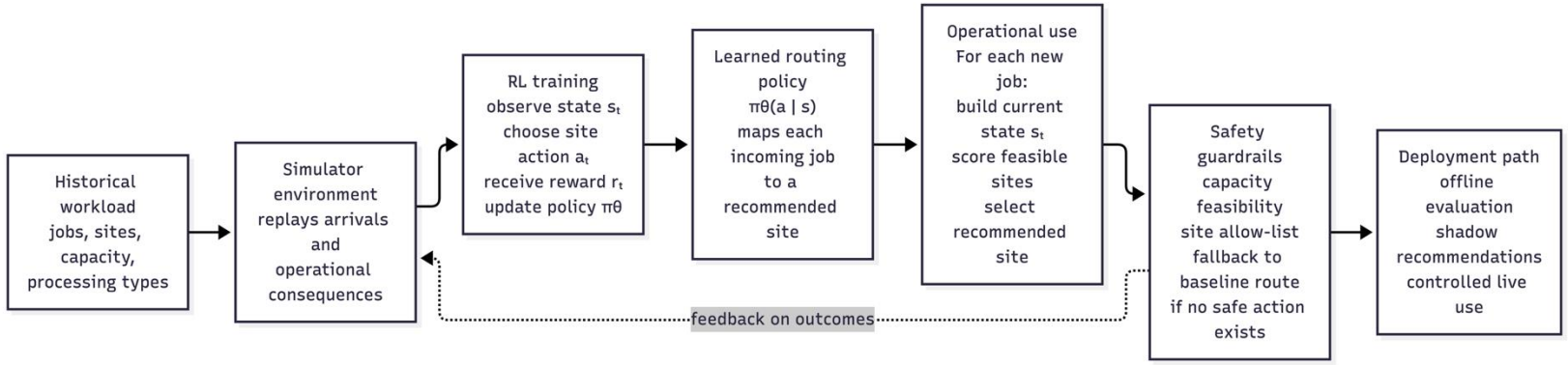
Generative Population Validation



Population-level validation: compares observed and synthetic timing distributions under the same observation cutoff. It does not attempt to reproduce each historical job one-to-one.
Result: the simulator reproduces the broad long-tailed shape of both queue-time and runtime distributions.
Remaining gap: the largest visible differences occur around the distribution peaks and lower-time ranges, particularly for runtime. Processing-type-level plots are needed to determine whether these differences come from within-type timing error or workload-mix shift.

RL Environment Design

The environment turns each scheduling event into a fixed-size observation and a valid-action choice



Observation

Ready-job window
Processing type, cores, RAM, age

Site state
Free cores, RAM, current load

Eligibility information

Action representation

A fixed ready-job window defines the candidate jobs.

The policy chooses a site for the selected candidate job.

Action mask

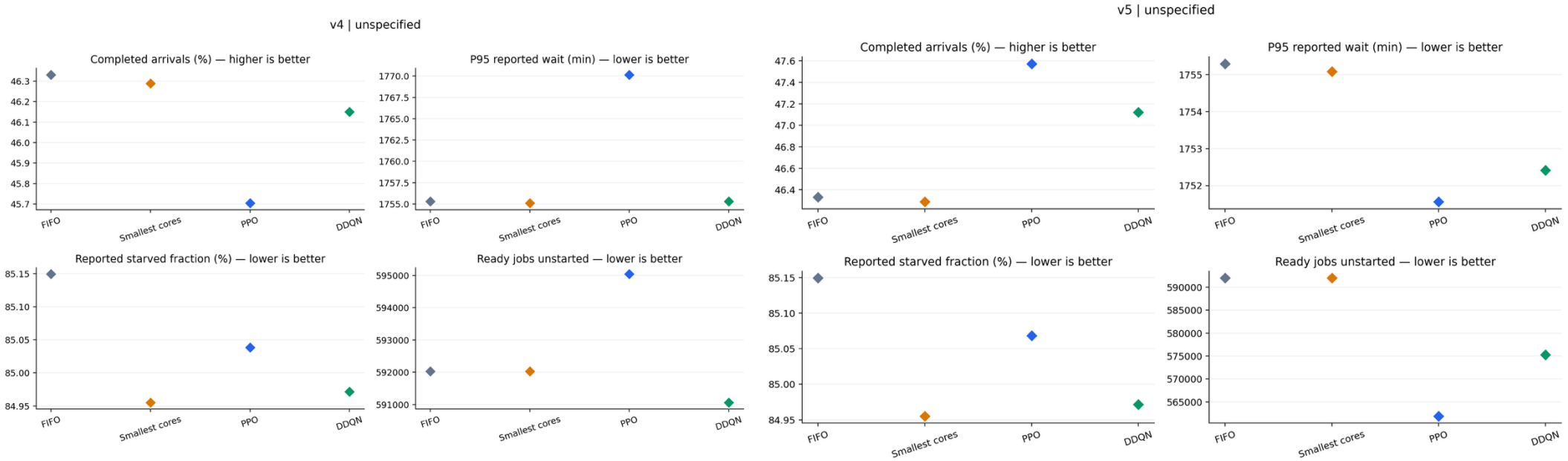
Blocks invalid choices before the policy samples an action.

Examples: ineligible site, insufficient capacity, empty candidate slot.

Action masking keeps the RL policy within operational constraints while still allowing it to learn which feasible assignment works best.

RL Training and Evaluation

Maskable PPO is tested against transparent routing baselines under the same simulated conditions



In the full-workload pilot, PPO with reward v5 completed more arriving jobs and left fewer ready jobs unstarted than FIFO. P95 waiting time improved slightly, but the reported starvation metric did not lead the baselines. These are preliminary results requiring repeated-seed evaluation

To Do

Train through **complete episodes** with checkpoints | Repeat across training and evaluation seeds | More policy

Summary

- Data-driven PanDA simulator from historical snapshots.
- Modelled job lifecycles, routing, queueing, and core/RAM capacity.

Simulator needs **refinement** in timing calibration and routing realism.

- Tested FIFO, smallest-cores, PPO, and DDQN policies.
- Reward v5 showed promising PPO gains in the pilot.

Next: improve simulator fidelity, extend training, and validate across seeds and unseen workloads.

Timeline: ACAT 2027 (abstract submission by end of October)

Backup

Attempted policies	Current evidence	Future direction
FIFO — arrival-order baseline	Tiny test completed for all four policies	Add periodic checkpoints and recovery
Smallest-cores — resource-size baseline	Zero invalid actions reported	Train through complete episodes and record simulated-time coverage
Maskable PPO — learned policy	Identical completions and zero measured waiting in this test	Compare minimal and richer observations at matched training budgets
DDQN — value-based learned policy	No performance separation demonstrated in the tiny test	Repeat across training and evaluation seeds; report variability

Reward Function

Reward balances completed work against normalized queue and runtime penalties

R = start reward − α × normalized queue penalty − β × normalized runtime penalty − γ × remaining-wait penalty

Start reward

Positive reward when the policy starts a feasible job.

Encourages useful completed work rather than idle decisions.

Normalized timing

Queue and runtime compare each job with historical timing for its processing type.

Long job types are not penalized simply for being long.

Remaining-wait penalty

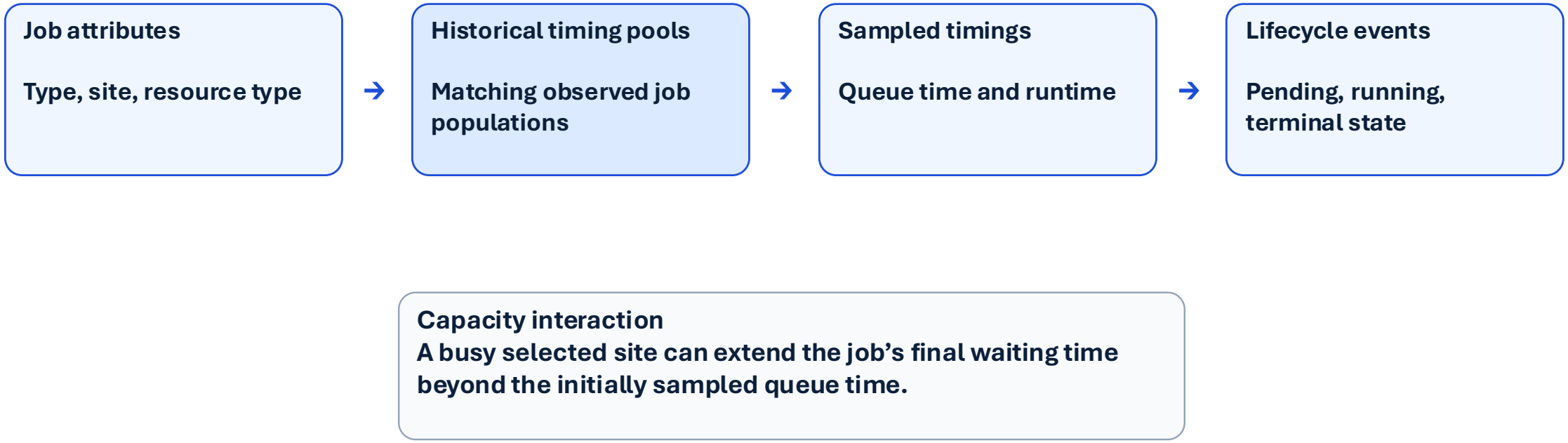
Jobs still waiting at the episode horizon receive a penalty based on accumulated wait.

Prevents the policy from ignoring difficult jobs.

α, β, and γ are tunable weights. They express how strongly the policy values throughput, timing, and unfinished waiting work.

Empirical Timing and Lifecycle Behaviour

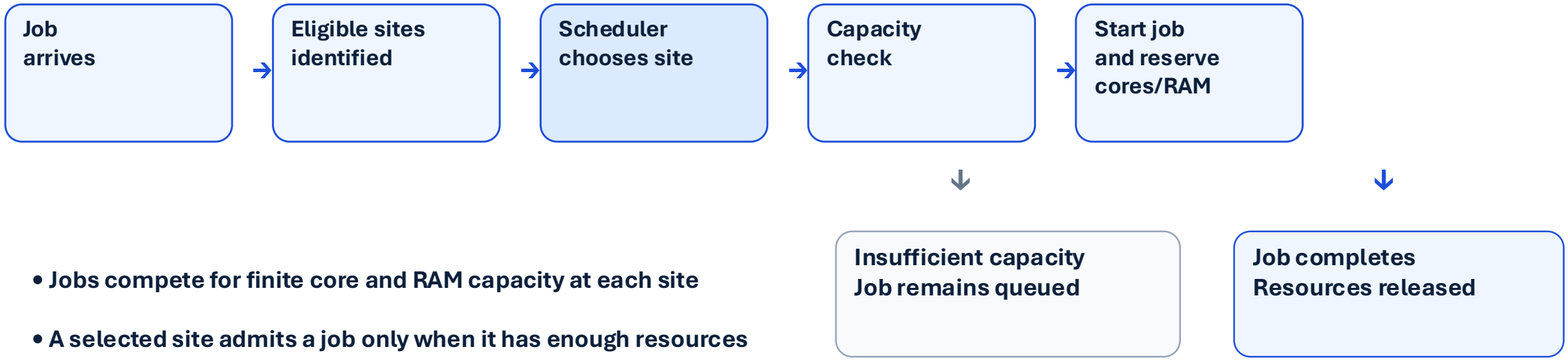
The simulator samples observed timing and lifecycle patterns instead of assigning fixed durations.



Timing and state changes retain historical variability rather than using one average job duration.

Event-Based Capacity Simulation

Routing decisions interact with finite site resources, so queueing emerges from the simulated events.

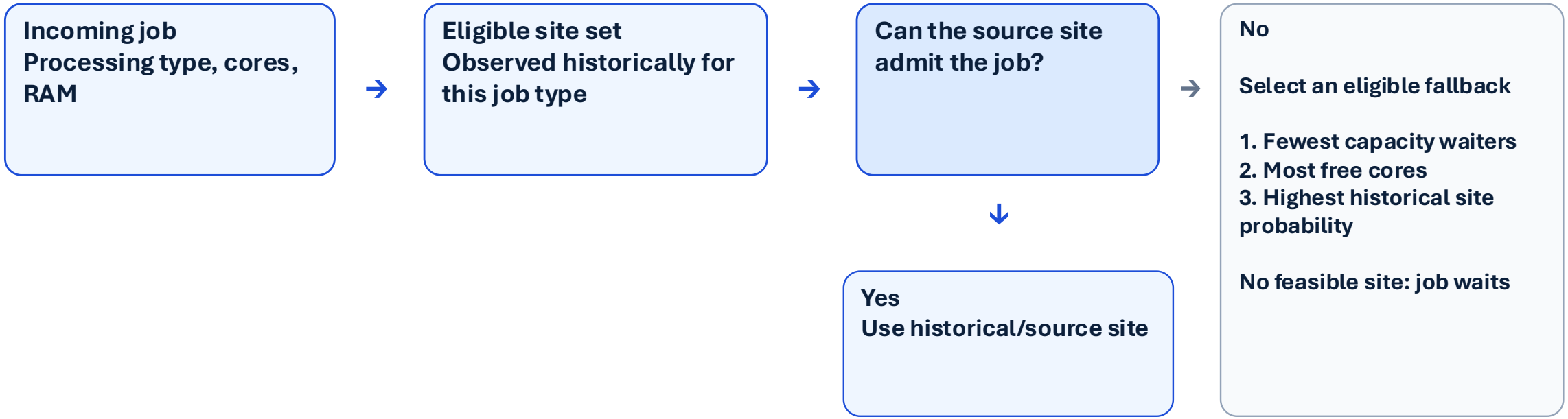


- **Jobs compete for finite core and RAM capacity at each site**
- **A selected site admits a job only when it has enough resources**
- **Completion releases resources for later jobs**

Released capacity allows queued jobs to be reconsidered.

Site Eligibility and Simulator Routing

The simulator restricts routing to historically observed compatible sites and preserves source-site preference.



The simulator reroutes only when the source site cannot currently admit the job.

One simulated job	Model / historical source	What the simulator does
Arrival time	Simulator rule	Currently places new arrivals uniformly across the simulation horizon. This is not yet a learned time-of-day arrival model.
Processing type	Arrival-composition model	Samples the overall mix, such as the proportion of deriv, simul, jedi-run, etc.
Job attributes	Attribute pool	Samples a historical job with the selected processing type: cores, RAM, resource type, priority-related fields, and site-related attributes.
Initial state for already-active jobs	Initial-population pool	Samples observed jobs and their status, job age, and status age from historical snapshots.
Candidate / source site	Historical site-eligibility model	Uses historically observed processing-type-to-site associations.
Whether the job can start	Capacity-admission model	Checks simulated free cores and RAM at the selected site.
Base queue delay	Queue-time empirical pool	Samples a historical queue-time value for that processing type, or type-site-resource group when conditional timing is enabled.
Extra queue delay	Event-based capacity model	Adds further waiting if the site lacks free cores or RAM when the job becomes ready.
Runtime	Runtime empirical pool	Samples a historical runtime value for that processing type, optionally with site/resource conditioning and calibration.
Next lifecycle status	Transition-probability model	Samples observed transitions, such as running → finished or running → failed, conditional primarily on processing type and current status.
Time spent in intermediate status	Dwell-time pool	Samples observed time spent in transitions such as merging before the next state.
Final status and released capacity	Lifecycle + event engine	Applies terminal transition, records the result, and returns cores/RAM to the site.