Implementing Resource Rights at ALCF

A Practical Guide for the Argonne Leadership Computing Facility

Executive Summary

This document describes how the Resource Rights model can be implemented at ALCF to enable autonomous AI agents to operate on Aurora, Polaris, and future systems with bounded authority and full accountability.

The problem. Agents today either impersonate PIs (security risk, poor accountability) or require manually configured service accounts (doesn't scale). The scheduler has no concept of delegated authority.

The solution. Resource rights make authority explicit and delegable. A PI delegates a portion of their allocation to an agent via a signed token specifying node-hours, queues, and constraints. A submission proxy validates tokens and provisions execution contexts (currently pooled accounts). Agents access storage via Globus with delegated permissions. All actions are auditable back to the delegating PI.

Key components:

- *Agent identity*: Agents register as Globus applications with their own credentials
- *Resource rights as tokens*: Signed JWTs specifying project, limits, constraints, and validity
- *Submission proxy*: Validates tokens, provisions execution contexts, submits to PBS
- *Execution contexts*: Pooled accounts that isolate agent jobs without per-agent Unix accounts
- *Data access*: Globus-mediated storage with delegated permissions and transfer quotas

Deployment. Three phases: (1) pilot on Polaris with manual token issuance, (2) self-service portal and Aurora support, (3) native PBS integration and cross-facility federation.

Extensions. SSH certificates for interactive access, low-latency debug queues, build-node sandboxes, facility health APIs.

Contents

1 Introduction

This document describes how the Resource Rights model can be implemented at the Argonne Leadership Computing Facility (ALCF) to support autonomous scientific agents operating on Aurora, Polaris, and future systems. The goal is to enable AI agents to submit jobs, access data, and consume allocations on behalf of researchers—with bounded authority, full accountability, and minimal administrative overhead.

ALCF already has the building blocks: allocations, Globus for data transfer and authentication, PBS for job scheduling, and institutional identity. Resource rights provide the conceptual layer that connects these components into a coherent framework for delegated authority. This is not a proposal to replace existing infrastructure, but to extend it so that autonomous agents can participate in ALCF workflows with the same accountability expected of human users.

2 Current State and Gaps

ALCF allocations are already a form of quantitative resource right—authority to consume a bounded amount of compute. What's missing is the ability to delegate portions of that allocation to autonomous agents with explicit constraints.

Today, agents either run under a PI's credentials (impersonation) or require manually configured service accounts. The scheduler has no concept of "this job is submitted by Agent X on behalf of PI Y, drawing from a subset of Y's allocation that was explicitly delegated to X."

3 Architecture Overview

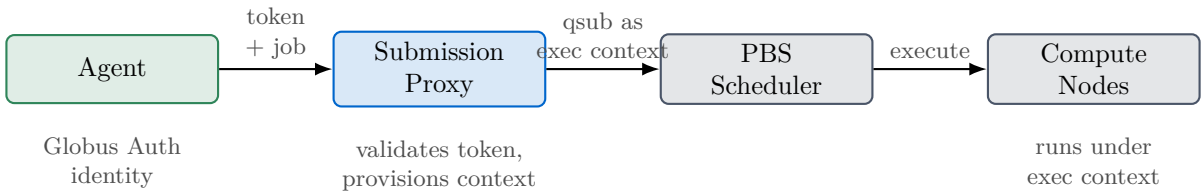

Figure 1: Submission flow. The agent authenticates with its own Globus identity and presents a resource right. The proxy validates the right, provisions an execution context, and submits to PBS. The job runs under that context, which is terminated and sanitized after completion.

The architecture has four components:

Agent identity. Agents register as Globus applications with their own client credentials, distinct from their supervising PI. This establishes the agent as a principal that can hold delegated rights.

Resource rights as tokens. A resource right is a signed JWT specifying the agent, project, delegating PI, quantitative limits (node-hours, max job size), constraints (allowed queues, approval thresholds), and validity period. Listing 1 shows an example.

Delegation service. A self-service portal where PIs create, monitor, and revoke delegations. The service issues signed tokens, tracks consumption, and pushes revocations to the submission proxy.

Submission proxy with execution context management. The proxy validates tokens and provisions execution contexts for jobs. At ALCF, contexts are currently implemented as pooled accounts, but the model abstracts over this implementation detail (see Section 4).

Listing 1: Example resource right token

```
{
  "iss": "alcf.anl.gov",
  "sub": "agent-uuid-12345",
  "exp": 1738368000,
  "project": "ProjectX_INCITE2026",
  "delegator": "pi-username",
  "resources": {
    "system": "aurora",
    "node_hours": 10000,
    "max_nodes_per_job": 512,
    "queues": ["prod", "debug"]
  },
  "constraints": {
    "require_approval_above": 2048
  }
}
```

4 Execution Context Model

Agents must not require permanent local Unix accounts. The resource rights model separates *agent identity* (who is making the request) from *execution context* (the environment in which jobs run). This separation enables the resource provider to manage execution contexts independently of the agent population.

4.1 Abstract Interface

From the agent's perspective, the execution context lifecycle has four operations:

- **Provision** — agent requests an execution context for a job or session

- **Interact** — agent submits jobs, accesses files, or connects via SSH under that context
- **Extend** — agent requests continued use (for session-based lifetimes)
- **Terminate** — context is released and sanitized

SSH access is important for interactive work: building software, debugging jobs, inspecting state. The resource provider can issue short-lived SSH credentials (e.g., certificates) scoped to the provisioned execution context, enabling MFA-free interactive access without permanent accounts.

The resource provider implements these operations according to local constraints and capabilities. The agent need not know whether the underlying mechanism is a pooled account, a dynamically created account, a container, or something else entirely.

4.2 An Implementation Approach: Account Pool

If the creation of an account is expensive, then execution contexts can be implemented via a pre-provisioned pool of Unix accounts. This choice is driven by current infrastructure costs: dynamic account creation involves LDAP propagation latency across compute nodes, home directory provisioning, and cleanup complexity. A pool amortizes these costs.

How it works:

1. Agent calls **Provision** (implicitly, by submitting a job via the proxy)
2. Proxy allocates an available account from the pool (e.g., `agent-pool-037`)
3. Agent **Interacts**: job runs under that account's UID
4. On job completion, proxy calls **Terminate**: account is sanitized and returned to the pool

For session-based work (multiple related jobs), the agent can **Extend** to retain the same execution context across jobs, avoiding repeated provisioning overhead. Session lifetime might be tied to the resource right's validity period or to explicit agent requests.

Pool sizing. A pool of 500 accounts can support thousands of agents over time, since pool size tracks peak concurrent jobs, not total agent population. If the pool is exhausted, new submissions queue until accounts free up—providing natural backpressure.

Reset procedure. The **Terminate** operation sanitizes the account: home directory cleared, credentials revoked, cron entries removed, lingering processes terminated, temp files deleted. The reset must be thorough, fast, and verified before the account returns to the pool.

File access. Jobs access project data through group permissions. All pool accounts belong to a shared group that projects can grant access to specific directories.

4.3 Alternative Implementations

The abstract interface accommodates other implementations as infrastructure evolves:

- **Dynamic account creation** — if LDAP propagation becomes fast enough, accounts could be created on demand and destroyed after use
- **Container-based contexts** — jobs run in containers with ephemeral UIDs; the container image defines the execution environment
- **VM-based isolation** — for stronger isolation requirements, each execution context could be a lightweight VM

The choice is a local optimization invisible to agents. ALCF's current pool implementation can evolve without changing the agent-facing interface or the resource rights model.

5 Data Access via Globus

Agents access Eagle and Grand storage through Globus, not local accounts:

- PIs share specific paths with agent identities via Globus
- Agents use the Globus Transfer API with their own OAuth tokens
- Resource rights can include transfer quotas (e.g., "5 TB to this collection")

Globus is useful here because it supports access to storage based on OAuth tokens rather than local accounts.

For compute jobs needing filesystem access during execution, the pool account's group membership provides read/write access to project directories.

6 End-to-End Example

Dr. Jones has an INCITE allocation spanning Polaris and Aurora. She deploys "MatSynth-Agent" to explore battery electrolyte candidates.

Setup:

1. Dr. Jones registers MatSynth-Agent as a Globus application
2. Via the delegation portal, she issues a resource right: 50K node-hours on Polaris, 200K on Aurora, 5 TB transfer quota on Eagle, 60-day validity

Execution:

1. Agent pulls 500 GB of molecular structures from ORNL to Eagle via Globus Transfer

2. Agent submits 2,000 DFT screening jobs to Polaris via the proxy; each provisions an execution context
3. Agent transfers results within Eagle, then submits ML training jobs to Aurora
4. Agent iterates: analyze results, submit follow-up DFT jobs, repeat

The agent never had a permanent Unix account on any system. Identity flowed through Globus Auth; execution contexts were provisioned on demand and terminated after use; data access was mediated by Globus.

Exception handling:

- *Quota exhausted:* Proxy rejects submission; Dr. Jones issues more delegation via portal
- *Unauthorized queue:* Proxy rejects with "constraint violation"
- *Revocation:* Delegation service pushes to proxies; next submission fails immediately
- *Security incident:* Audit trail shows full delegation chain; affected execution contexts can be quarantined; no permanent user accounts compromised

7 Deployment Plan

Phase 1 (3 months): Pilot on Polaris with manual token issuance.

- Deploy submission proxy with token validation and account pool
- Manual token issuance via support ticket
- Pilot with 2–3 projects; validate accounting and audit trail

Phase 2 (6 months): Self-service portal and Aurora.

- Web portal for creating/revoking delegations
- Extend to Aurora; add storage rights via Globus
- Consumption alerts and reporting

Phase 3 (9 months): Native integration and federation.

- PBS hooks for direct token validation (proxy becomes optional)
- Advanced constraints (time-of-day, reservations)
- Cross-facility token recognition with OLCF/NERSC

Each phase is independently valuable; later phases add convenience without requiring earlier completion.

8 Extensions

The core model—resource rights, execution contexts, Globus-mediated data access—addresses the most common agent workflows. Several extensions would further reduce friction for advanced use cases.

SSH certificates. The execution context interface includes interactive access, but the current implementation requires agents to use the submission proxy for all interactions. For build, debug, and inspection tasks, direct SSH access is essential. The delegation service could act as an SSH certificate authority: the agent presents its resource right, receives a short-lived certificate scoped to the provisioned execution context, and connects to login nodes without MFA. Certificates expire with the execution context, and all sessions are logged against the delegation chain.

Low-latency debug tier. Iteration speed on Aurora and Polaris is often limited by queue wait times, even for trivial smoke tests. A token-metered *debug QoS right* could grant access to a small standing reservation with strict limits (e.g., 5 launches/hour, $\leq$5 minute walltime). Alternatively, trivial tests ("does it import on one tile?") could route to an always-on micro-test endpoint entirely outside the production queues. Either approach encodes fast-iteration capacity as a delegable resource.

Build-node sandbox. Aurora UANs restrict outbound network access, causing `pip install` to hang silently. A *build-node right* could authorize access to a network-connected build environment—a small node or container with PyPI/conda access—where agents install dependencies and write results to a shared-stack path. The build artifact then transfers to compute nodes via Globus, sidestepping network restrictions entirely.

Facility health API. Login nodes go down for maintenance; hardcoding a specific node is a latent failure. The submission proxy could expose a simple health endpoint (`GET /status/aurora/login-nodes`) returning current reachability. For interactive SSH access, the certificate issuer could return a certificate valid across all live frontends, with the agent (or proxy) selecting a reachable one. This eliminates "which login node is up?" as a human-intervention event.

9 Security

- Tokens signed with HSM-backed keys; 30–90 day validity
- Immediate revocation via delegation service
- Every job records delegation chain in metadata
- Anomaly detection on consumption patterns

10 Conclusion

ALCF already manages implicit resource rights through allocations. This proposal makes them explicit, delegable, and machine-consumable. The execution context model enables agents to run jobs without permanent local accounts—the resource provider provisions contexts on demand and terminates them after use, with the implementation (currently pooled accounts) hidden behind an abstract interface.

The implementation is incremental: Phase 1 requires only a submission proxy and manual token issuance. By the time large-scale autonomous science campaigns arrive, ALCF will have the authorization infrastructure to support them.

For the underlying conceptual model, see: "Resource Rights: Authority as a Managed Resource for Autonomous Systems." For the delegation service API and implementation details, see: "Delegation Service Design: A Resource Rights Management Service."

A Why Pooled Accounts for ALCF Today

The abstract execution context interface (provision/interact/extend/terminate) can be implemented in multiple ways. ALCF uses pooled accounts because alternatives have significant drawbacks in the current infrastructure:

Per-agent accounts don't scale: LDAP propagation across thousands of compute nodes adds latency, and decommissioned agents leave orphaned state requiring cleanup.

PI or project mapping provides no isolation between concurrent agent jobs and confuses accountability.

Containers alone add complexity (MPI, GPUs, high-speed interconnects) and still require an underlying identity for filesystem access.

The pool amortizes provisioning costs while providing real per-job isolation. As infrastructure evolves—faster directory services, better container integration—the implementation can change without affecting the agent-facing interface.

B Storage Resource Rights via Globus

Section 5 describes data access via Globus at a high level: PIs share paths with agent identities, and agents use the Transfer API with their own OAuth tokens. This appendix summarizes how Globus sharing becomes a proper resource right. For complete details—including the full API specification, background jobs, and deployment options—see the companion document *Delegation Service Design*.

B.1 What Globus Provides

Globus supports granting a specific client identity (an agent's registered application) read and/or write access to a path on a mapped collection (Eagle, Grand, or future `/flare` exposure). This is the *enforcement mechanism*—Globus checks identity and permits or denies the transfer.

B.2 What Makes It a Resource Right

A Globus share alone is an access control entry. A resource right adds:

1. **Provenance.** Every grant traces back through a chain of delegations to a recognized authority (the PI's allocation).
2. **Cascading revocation.** Revoking a delegation invalidates all rights derived from it.

B.3 Data Model

The delegation service uses a unified schema for both storage and compute rights. For storage, the key fields are:

Field	Description
`delegation_id`	UUID, primary key
`parent_id`	UUID, nullable (null = root delegation from PI)
`delegator`	Globus identity of principal creating this delegation
`grantee`	Globus identity receiving the right
`resource_type`	`"storage"` for storage rights
`resource_id`	Globus collection UUID (e.g., Eagle)
`scope`	JSON: `{path, operations}`
`quota`	JSON: `{bytes: N}` or null
`consumed`	JSON: `{bytes: N}`
`suspended`	Boolean, default false
`enforcement_ref`	Globus ACL rule ID
`expires_at`	Timestamp, nullable
`revoked`	Boolean, default false

See *Delegation Service Design* for the complete schema, including compute rights and consumption tracking tables.

B.4 Operations

Issue (PI → Agent). A PI uses the delegation portal to grant an agent access to a path:

1. Portal verifies PI has authority over that path (via project membership or existing delegation)
2. Portal calls Globus to create an ACL rule granting the agent's identity the specified operations
3. Portal records the delegation with `parent_id = null` (or pointing to the PI's root authority)

Delegate (Agent → Sub-agent). An agent holding a storage right can delegate a subset to another agent:

1. Delegation service verifies the delegating agent holds a valid (non-revoked, non-expired) right covering the requested path and operations
2. Service calls Globus to create an ACL rule for the sub-agent
3. Service records the delegation with `parent_id` pointing to the agent's delegation

The delegated path must be equal to or a subdirectory of the parent's path; operations must be a subset.

Revoke. When a delegation is revoked:

1. Mark the delegation as `revoked = true`
2. Call Globus to delete the ACL rule
3. Recursively revoke all child delegations

B.5 Expiration and Lifetimes

Delegation constraint. A delegated right cannot outlive its source:

$$\texttt{child.expires\_at} \leq \texttt{parent.expires\_at}$$

Validity. A delegation is valid if and only if: it is not revoked, it is not expired, and its parent (if any) is valid.

Enforcement. Two mechanisms provide defense in depth:

1. *Globus-native expiration.* When creating an ACL rule, set its expiration to `expires_at`. Globus refuses transfers after this time automatically.
2. *Periodic cleanup job.* A background job revokes expired delegations, cascading to children and removing ACLs.

B.6 Transfer Quotas

Globus has no native transfer quota. The delegation service combines soft quotas (alerts) with automatic write suspension.

Delegation constraint (conservation). When delegating, child quotas cannot exceed available capacity:

$$\sum \texttt{child.quota.bytes} \leq \texttt{parent.quota.bytes} - \texttt{parent.consumed.bytes}$$

Consumption tracking. The delegation service subscribes to Globus transfer completion events and updates `consumed.bytes` accordingly.

Write suspension. When consumption reaches quota:

1. Update the Globus ACL to read-only
2. Set `suspended = true`
3. The agent can still read; writes are blocked

Restoration. When the PI increases the quota, restore write permissions and clear `suspended`.

This approach provides meaningful enforcement without the complexity of pre-transfer size estimation. See *Delegation Service Design* for implementation details.

B.7 Integration with Compute Rights

Storage and compute rights share the same delegation service and tree structure. An agent can hold both types simultaneously, and the submission proxy can query the delegation service to determine what resources an agent may access. The delegation service is the single source of truth for all resource rights.