

OLYMPUS Rack AI Cluster: An Evidence-Aware Architecture, Prototype Evaluation, and Integration Roadmap

Aaron Singh

Abstract—This paper presents OLYMPUS Rack AI Cluster, a project in a twenty-project AI infrastructure portfolio. The current repository is an active prototype with implemented behavior, automated correctness tests, and explicit boundaries around unverified hardware or production claims. We describe the system model, current implementation, goals, and evaluation method, then propose five integrations that advance the project toward reproducible system-level validation. A local audit executed 4 project tests successfully. No accelerator, deployment, or performance conclusion is inferred unless a corresponding committed benchmark or profiler artifact exists.

Index Terms—OLYMPUS, AI infrastructure, reproducibility, prototype validation, systems evaluation, hardware–software co-design

I. INTRODUCTION

OLYMPUS Rack AI Cluster (P11) addresses a bounded problem within the portfolio’s track-03-distributed-infrastructure track. Its declared status is *Active Prototype* [1]. The project validates its central mechanism before adding device-specific acceleration, production orchestration, or real-world hardware. This ordering matters because optimization without a trusted reference can make incorrect behavior appear successful.

The project goals are to establish deterministic behavior, encode correctness as tests, create machine-readable evidence, identify limiting resources through measurement, and integrate results with adjacent portfolio systems without losing provenance. The repository contains one paper named exactly after its singular folder: `P11-olympus-rack-ai-cluster.tex`.

II. TECHNICAL CONTEXT

Distributed inference depends on scheduling, capacity, communication, reliability, and observability. Later multi-node experiments must record topology, synchronization costs, retries, and tail latency. The portfolio benchmark standard requires environment, workload, method, metric, artifact, reproduction, and limitation fields; unknown values remain explicitly unmeasured [3].

III. SYSTEM MODEL AND ARCHITECTURE

The prototype is organized around the following domain model:

$$R_{max} = \min_j \left\lfloor \frac{C_j}{D_j} \right\rfloor \quad (1)$$

The equation is a design and test abstraction rather than a claimed empirical law. It supports invariants and expected-value checks while later implementations replace synthetic inputs with representative workloads or devices.

The software architecture contains an input/configuration layer, a deterministic core, validation and evidence output, and a local visualization. The principal inspected source artifacts are `python/rack_planner.py`. Unsupported real-world conditions are surfaced as limitations rather than silently simulated.

IV. DETAILED SCRIPT OPERATION AND RATIONALE

The rack planner validates a JSON inventory, totals rack units, power, memory, and accelerators, then estimates pooled workload replica capacity from the limiting resource. It supports transparent capacity planning without presenting the result as a deployed rack.

The execution path is:

- 1) Load and structurally validate rack inventory JSON.
- 2) Check per-item rack-unit and resource requirements.
- 3) Aggregate rack units, power, memory, and accelerator totals.
- 4) Divide each pooled capacity by per-replica demand.
- 5) Report the minimum ratio as capacity and name the limiting resource.

V. IMPLEMENTED PROTOTYPE

The metadata and source audit found these completed features [2]:

- Validated JSON rack inventory
- Python rack-unit, power, memory, and accelerator totals
- Pooled workload replica capacity calculation
- Unit tests and machine-readable analysis output

`python3 -m unittest discover -s tests -p 'test_*.py'` completed successfully with 4 tests. The inspected test artifacts are `tests/unit/test_rack_planner.py`. This is evidence of local correctness for encoded cases, not production scale or hardware performance.

VI. TESTING METHODOLOGY AND OBSERVED RESULTS

Testing uses Python’s standard `unittest` discovery and exercises the public behavior of the reference implementation. The audit reran the suite from the project folder with `python3 -m unittest discover`

TABLE I
IMPLEMENTATION ARTIFACTS AND WHY THEY EXIST

Artifact	Observed responsibility	Engineering rationale
python/rack_planner.py	ServerGroup, RackLimits, load_inventory, summarize_rack, estimate_workload_capacity, git_value, main	Implements the inspectable, unit-tested project core.
scripts/reproduce.sh	Fixed test and demonstration entry point	Gives another developer one command for local reproduction.
PROJECT.yaml	and	Separates declared intent from evidence-backed implementation.
ANALYSIS.md		Makes outputs inspectable without upgrading simulation into a hardware claim.
streamlit_app.py		

`-s tests -p 'test_*.py'`. All 4 discovered tests passed. The result establishes correctness only for the encoded local cases; it does not establish accelerator correctness, real-device behavior, production reliability, or benchmark completion.

No numerical performance result is promoted by this test run. Where scripts emit JSON or JSONL, those outputs remain raw or simulation-specific until a reviewed summary includes hardware, software, workload, warm-up, repetition, correctness threshold, Git revision, and limitations.

VII. CLAIM BOUNDARIES AND RISKS

The project records these unverified or excluded claims:

- Capacity result is a pooled planning simulation, not a deployed rack

The main risk is confusing synthetic or modeled behavior with deployed-system behavior. Other risks include incomplete workloads, platform-dependent timing, missing failure injection, and interfaces not yet exercised across device boundaries. Performance claims require a reviewed record meeting the portfolio standard.

VIII. EVALUATION PLAN

Evaluation proceeds through correctness tests, deterministic reproduction with Git and environment metadata, repeated benchmarks reporting latency/throughput/memory/error metrics, and a named profiler capture tied to exact hardware and source revision. Success requires reference equivalence within a documented tolerance, preservation of safety and resource invariants, clear failures, and evidence reproducible from a clean environment.

IX. GOALS, MILESTONES, AND SUCCESS CRITERIA

The project goals are staged so that correctness precedes performance and integration. A goal is complete only when its proof artifact is committed or otherwise reviewable; prose or a simulated number alone is insufficient.

X. FUTURE WORK AND INTEGRATIONS

The five project-specific next steps are:

- 1) Per-node placement constraints
- 2) Cooling, redundancy, and network capacity model

- 3) Measured workload throughput inputs
- 4) Generate deployment manifests and validate power, cooling, and network constraints.
- 5) Supply capacity to P10 and operational state to P20.

The early items complete declared evidence; the later items connect downstream portfolio consumers. Each integration should add tests and a reviewable artifact such as JSONL evidence, a report, profiler capture, deployment manifest, trace, or labeled data set.

XI. CONCLUSION

OLYMPUS Rack AI Cluster is an evidence-aware active prototype: its implemented behavior and tests are real, while unbuilt hardware, deployment, and performance goals remain labeled. Completing the five integrations in dependency order will advance it from a learning artifact toward a credible portfolio component.

REFERENCES

- [1] Aaron Singh, "OLYMPUS Rack AI Cluster PROJECT.yaml," local portfolio repository.
- [2] Aaron Singh, "OLYMPUS Rack AI Cluster: README, ANALYSIS, source, and test artifacts," local portfolio repository.
- [3] Aaron Singh, "AI Infrastructure Portfolio Benchmark Standard," local portfolio repository.

TABLE II
AUDITED TEST MATRIX

Test artifact and case	Behavior being checked	Observed result
tests/unit/test Rack planner.py:test Rack totals	Rack totals.	Pass (local)
tests/unit/test Rack planner.py:test Capacity identifies limiting resource	Capacity identifies limiting resource.	Pass (local)
tests/unit/test Rack planner.py:test Over limit inventory has zero capacity	Over limit inventory has zero capacity.	Pass (local)
tests/unit/test Rack planner.py:test Bad requirements are rejected	Bad requirements are rejected.	Pass (local)

TABLE III
DECLARED STATUS VERSUS AUDITED EVIDENCE

Audit field	Finding	Evidence source
Declared status	Active Prototype	PROJECT.yaml
Evidence-backed status	Active local prototype; 4 tests passed	Source plus local unittest run
Accepted measured results	None recorded in measured_results	PROJECT.yaml
Mismatch / claim boundary	Capacity result is a pooled planning simulation, not a deployed rack	PROJECT.yaml and ANALYSIS.md
Next proof required	Per-node placement constraints; Cooling, redundancy, and network capacity model	Planned features

TABLE IV
PROJECT GOALS AND REQUIRED PROOF

ID	Goal	Completion evidence
G1	Per-node placement constraints	Passing tests and a reviewed source artifact
G2	Cooling, redundancy, and network capacity model	Machine-readable result with reproduction metadata
G3	Measured workload throughput inputs	Reference-equivalence or domain-correctness report
G4	Add cooling, redundancy, and network constraints	Named profiler, deployment, or integration artifact
G5	Drive placement from measured workload demand	Dashboard/report link preserving provenance and limitations