LYRA SGLang Continuous-Batching Lab: An Evidence-Aware Architecture, Prototype Evaluation, and Integration Roadmap

Aaron Singh

***Abstract*—This paper presents LYRA SGLang Continuous-Batching Lab, 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 3 project tests successfully. No accelerator, deployment, or performance conclusion is inferred unless a corresponding committed benchmark or profiler artifact exists.**

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

I. Introduction

LYRA SGLang Continuous-Batching Lab (P08) addresses a bounded problem within the portfolio's track-02-llm-serving 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: `P08-lyra-sglang-continuous-batching-lab.tex`.

II. Technical Context

Language-model serving couples execution with cache management, request scheduling, and latency objectives. A simulator can validate policy logic, but only a real model runtime and measured workload can support performance claims. 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:

$$L_i = C_i - A_i, \quad \text{throughput} = \frac{N}{\max_i C_i - \min_i A_i} \tag{1}$$

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/batching_simulator.py`. Unsupported real-world conditions are surfaced as limitations rather than silently simulated.

IV. Detailed Script Operation and Rationale

The batching simulator applies identical request arrivals to static and continuous policies, tracks completion times, and summarizes latency, makespan, and throughput. It explains scheduling effects while keeping SGLang and hardware performance outside the current claim boundary.

The execution path is:

1) Validate request and server-capacity inputs.
2) Run a static policy that waits for batch boundaries.
3) Run a continuous policy that admits work as capacity becomes available.
4) Record arrivals, completions, latency, makespan, and throughput.
5) Compare policies on the identical deterministic workload.

V. Implemented Prototype

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

- Deterministic static and continuous batching simulations
- Arrival, completion, latency, makespan, and throughput metrics
- Unit tests and JSONL result output

`python3 -m unittest discover -s tests -p 'test_*.py'` completed successfully with 3 tests. The inspected test artifacts are `tests/test_batching.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/batching_simulator.py	Request, validate, simulate_continuous, simulate_static, summarize, sample_requests, run, 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 ANALYSIS.md	Status, completed work, planned work, and claim boundaries	Separates declared intent from evidence-backed implementation.
streamlit_app.py	Local evidence and status visualization	Makes outputs inspectable without upgrading simulation into a hardware claim.

`-s tests -p 'test_*.py'`. All 3 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:

- Simulation improvement does not prove SGLang or XPU performance

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) Configurable workload file and policy sweep
2) SGLang request adapter
3) Intel XPU serving measurements
4) Measure fairness and tail latency across arrival distributions.
5) Connect policy experiments to CHRONOS and a live SGLang endpoint.

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

LYRA SGLang Continuous-Batching Lab 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, "LYRA SGLang Continuous-Batching Lab PROJECT.yaml," local portfolio repository.

[2] Aaron Singh, "LYRA SGLang Continuous-Batching Lab: 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/test_batching.py:test_all_requests_complete	All requests complete.	Pass (local)
tests/test_batching.py:test_continuous_improves_sample_latency	Continuous improves sample latency.	Pass (local)
tests/test_batching.py:test_empty_capacity_rejected	Empty capacity 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; 3 tests passed	Source plus local unittest run
Accepted measured results	None recorded in measured_results	PROJECT.yaml
Mismatch / claim boundary	Simulation improvement does not prove SGLang or XPU performance	PROJECT.yaml and ANALYSIS.md
Next proof required	Configurable workload file and policy sweep; SGLang request adapter	Planned features

TABLE IV
PROJECT GOALS AND REQUIRED PROOF

ID	Goal	Completion evidence
G1	Configurable workload file and policy sweep	Passing tests and a reviewed source artifact
G2	SGLang request adapter	Machine-readable result with reproduction metadata
G3	Intel XPU serving measurements	Reference-equivalence or domain-correctness report
G4	Connect SGLang requests	Named profiler, deployment, or integration artifact
G5	Evaluate policy fairness and tail latency	Dashboard/report link preserving provenance and limitations