POLARIS Operations Console: An Evidence-Aware Architecture, Prototype Evaluation, and Integration Roadmap

Aaron Singh

***Abstract*—This paper presents POLARIS Operations Console, 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*—POLARIS, AI infrastructure, reproducibility, prototype validation, systems evaluation, hardware–software co-design**

I. Introduction

POLARIS Operations Console (P20) addresses a bounded problem within the portfolio's track-05-hardware-software-codesign 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: `P20-polaris-operations-console.tex`.

II. Technical Context

Hardware–software co-design links analytical models, workload evidence, accelerator mappings, and operator tools. Modeled, simulated, and measured values must remain distinguishable so design tradeoffs are not overstated. 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:

$$V(F) = \{p \in P \mid F(p) = 1\}, \quad N_s = |\{p \in P : \text{status}(p) = s\}| \tag{1}$$

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/generate_dashboard_data.py`, `dashboard.js`, `index.html`, `styles.css`. Unsupported real-world conditions are surfaced as limitations rather than silently simulated.

IV. Detailed Script Operation and Rationale

The generator discovers all twenty PROJECT.yaml files, parses and validates the supported metadata subset, derives project identifiers and feature counts, and writes JSON consumed by the static dashboard. The browser code then renders portfolio statistics and track/status filters from current local metadata.

The execution path is:

1) Discover the twenty canonical PROJECT.yaml files.
2) Parse supported scalar and list fields and validate project identifiers.
3) Derive status, track, completed-feature, and planned-feature values.
4) Write project-data.json for the browser client.
5) Render and filter cards so portfolio changes follow metadata rebuilds.

V. Implemented Prototype

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

- Unique codename mapping for P01-P20
- Track filter for presentation views
- Live local project status generation from all PROJECT.yaml files
- Status filter and evidence-aware feature counts

`python3 -m unittest discover -s tests -p 'test_*.py'` completed successfully with 3 tests. The inspected test artifacts are `tests/unit/test_generate_dashboard_data.py`. This is evidence of local correctness for encoded cases, not production scale or hardware performance.

TABLE I
IMPLEMENTATION ARTIFACTS AND WHY THEY EXIST

Artifact	Observed responsibility	Engineering rationale
python/generate_dashboard_data.py	parse_scalar, parse_project_yaml, project_id, collect_projects, 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.

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 -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:

- No additional claim boundary is recorded.

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) Benchmark summary ingestion
2) README and Obsidian note links
3) Automated refresh during local development
4) Add provenance links, stale-data warnings, and role-based views.
5) Integrate P07 benchmark, P12 reliability, and P19 co-design summaries.

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

POLARIS Operations Console 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, "POLARIS Operations Console PROJECT.yaml," local portfolio repository.
[2] Aaron Singh, "POLARIS Operations Console: 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_generate_dashboard_data.py:test_collects_all_twenty_projects	Collects all twenty projects.	Pass (local)
tests/unit/test_generate_dashboard_data.py:test_active_projects_come_from_yaml	Active projects come from yaml.	Pass (local)
tests/unit/test_generate_dashboard_data.py:test_simple_yaml_lists_are_counted	Simple yaml lists are counted.	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	No additional mismatch recorded	PROJECT.yaml and ANALYSIS.md
Next proof required	Benchmark summary ingestion; README and Obsidian note links	Planned features

TABLE IV
PROJECT GOALS AND REQUIRED PROOF

ID	Goal	Completion evidence
G1	Benchmark summary ingestion	Passing tests and a reviewed source artifact
G2	README and Obsidian note links	Machine-readable result with reproduction metadata
G3	Automated refresh during local development	Reference-equivalence or domain-correctness report
G4	Ingest benchmark summaries	Named profiler, deployment, or integration artifact
G5	Link every displayed claim to its source artifact	Dashboard/report link preserving provenance and limitations