Multi-CLI Agent Orchestration for Scientific Code Migration at LCLS

Abstract

We propose a YAML DSL and orchestration semantics that will transform unreliable LLM invocations into reproducible pipelines through strict, constrained sequencing. Validating on C/CUDA-to-PyTorch migrations at LCLS, we wish to demonstrate that declarative dependency specification, combined with simple control flow constructs and inter-agent message-passing, are sufficient to change the human domain expert's role from time-consuming active supervision to upfront planning and post-hoc verification. This work directly supports HAI's **Augment Human Capabilities** theme by transforming how scientists interact with complex code migration tasks—shifting from manual implementation to strategic oversight—while maintaining scientific rigor.

Problem

The core challenge in LLM-assisted program transformation is not individual tool capability but reliable composition of unreliable components. Scientific code translation tasks are a good venue for pushing the capability of AI-based program translation / synthesis because tasks of realistic size are intractable to automate, but the *sub-parts* of such tasks are often solvable using frontier LLMs and state-of-the art agentic frameworks such as Claude Code and Codex CLI. Encouragingly, the program-correctness constraints – such as differentiability requirements – are domain-specific yet formalizable and easily described in natural language.

This creates a research opportunity: is there an analysis-and-synthesis formulation of natural language agent orchestration that yields deterministic, robust executions despite stochastic, faulty primitives? Unlike traditional fault tolerance, LLM orchestration must handle semantic failures where outputs appear syntactically correct but violate domain constraints. The intellectual challenge connects verification theory (ensuring correctness properties) with program synthesis (automated code generation) in a setting where individual components are fundamentally unreliable.

SLAC's data systems software environment provides a concrete testbed for such automation efforts. Scientific and engineering demands frequently call for cross-language or cross-framework porting efforts (e.g. C to PyTorch; Tensorflow to Pytorch; Matlab to Python) and yet the resources available for such unglamourous efforts are limited.

Team

Principal Investigator: Alex Aiken, Alcatel-Lucent Professor of Computer Science, Stanford University.

Co-Investigator: Oliver Hoidn, Postdoc, SLAC/LCLS Data Analytics Group. Experience in ML for X-ray science and large-scale scientific computing.

Graduate Research Assistant: To be supported in PI's lab.

Research Contributions

Building on Hoidn's direct recent experience in converting conventional physical simulation programs into modern differentiable simulators, we will combine domain expertise with best practices in natural language agent development to automate future instances of similar tasks.

The proposed work advances computational workflow orchestration through three contributions that address coordination of non-deterministic language model agents:

1. Formal orchestration model: The system implements a sequential state machine that treats LLMs as non-deterministic oracles with explicit capture points. The engine provides agent registration, shell access, control flow (i.e. conditional and unconditional branching), and a bridge between LLM input / output text and DSL data primitives (specifically strings and booleans).

2. Dependency resolution: A declarative YAML-based system that validates the existence of dependencies and automatically assembles prompts from file artifacts, removing the need for imperative-style code.

3. Empirical validation: We will conduct a systematic evaluation on a small but well-characterized set of translation problems related to crystallography and diffuse scattering simulation and refinement.

The DSL syntax is a simple subset of YAML; for example:

```
steps:
  - name: Plan
    provider: gemini-cli
    input_file: prompts/analyze_crystallography.md
    depends_on: { required: ["taskspec.md"], inject: true }
```

Technical Approach

Workflows are declared in a YAML DSL that defines steps, providers, per-step environment, and file dependencies for both CLI agent invocations and programmatics steps. The engine executes steps sequentially with LLM provider-specific wrappers. The orchestrator automatically injects declared dependencies into prompts. It preserves state integrity via atomic writes and workflow checksums, and records an environment snapshot for provenance. Multi-agent concurrency is realized using a file-based message-passing scheme. Dependencies are specified declaratively and enforced by the orchestrator.

We provide a detailed specification of the system here: `https://github.com/hoidn/agent-orchestration`

Validation

Evaluation focuses on numerical accuracy with relative error $< 10^{-6}$ on target simulations and refinement tasks.

Ethics and Society Review Statement

This project develops tooling to assist code translation in a scientific setting. The primary concerns are accuracy, oversight, data handling, intellectual property, and safe operation. We address them as follows.

1. Accuracy and scientific validity

- Risk: translated code may silently diverge from reference implementations and lead to incorrect analyses or wasted beamtime.
- Mitigation: every workflow includes verification gates before any result is used. Gates include differential tests against the reference (CPU/CUDA) implementation, property-based tests for invariants, numeric tolerances tied to the task (e.g., relative error $< 10^{-6}$ for specified kernels), and determinism checks. Any deviation triggers a fail-closed outcome and requires revision.

2. Human oversight and responsibility

- Risk: over-reliance on automation or ambiguous ownership of results.
- Mitigation: a named domain expert must review diffs, test results, and documentation and must approve a final gate in the workflow before code is used in analysis or experiments. We will document roles and sign-off points in the repository and in reports.

3. Data handling and privacy

- Risk: sending proprietary code or experimental details to third-party APIs.
- Mitigation: the default workflow restricts inputs to approved repositories; no personal data are involved. Use of external LLM APIs requires prior approval from code owners; when available, provider settings to opt out of training on inputs are enabled. Sensitive content is redacted or excluded. Local execution or offline runs are used where required by policy.

4. Intellectual property and licensing

- Risk: license incompatibilities or unclear provenance of generated code.
- Mitigation: we scan inputs and outputs with a license checker, record file-level provenance in the workflow state, and retain original headers. We will only translate code that we are authorized to process and will release new artifacts under a permissive license agreed with the PI and code owners.

5. Safe execution and security

- Risk: generated code could perform unsafe operations when run.
- Mitigation: generated code is executed in a sandbox during testing with restricted filesystem and network access. We run static checks (linters, import allow-lists) and require explicit review for any operation that touches external systems.

6. Resource use and environmental impact

- Risk: unnecessary compute or API usage.
- Mitigation: we track token/compute use, set budget caps, cache intermediate results, and prefer low-cost tests (CPU, small cases) early in pipelines.

7. Access considerations

- Observation: some labs may lack budget for commercial APIs.
- Step: we will document workflows that avoid paid APIs when feasible and provide clear instructions to run verification and benchmarks with local resources.

8. Scope limits

- This project does not process personal or clinical data, does not control hardware or experiments, and is not intended for safety-critical use. Results are research artifacts and require human review before operational use.

All code, specifications, tests, and decision logs will be released publicly to support inspection and reuse. We will comply with Stanford policies and complete the ESR process, including documenting risks, mitigations, and any residual concerns.

Timeline

Phase 1 (Months 1–3): Core engine, interagent protocols, and agent library
Phase 2 (Months 4–8): Develop workflows, establish manual and semi-automated baselines
Phase 4 (Months 9–12): Evaluation, documentation, deployment guide, public release

Budget

$75K total distributed as follows:

- Stanford student support (PI's lab): $20,000 (27%)
- SLAC research personnel: $40,000 (53%)
- LLM API costs (OpenAI, Anthropic, Google): $8,000 (11%)
- Travel and contingency: $7,000 (9%)

Note: 8% infrastructure charge included in total. GPU compute provided via SLAC facility allocation.

Related Work

Prior LLM-based program generation approaches build on different abstractions. Most directly relevant are agent frameworks such as AutoGPT and LangChain. These operate at the LLM provider API level and do not encapsulate SOTA agentic frameworks. Genuinely agentic tools such as Amp and Claude Code, while powerful, lac the programmatic orchestration primitives that we wish to develop here.

Deliverables

- YAML DSL specification and Python orchestration engine implementation
- Translation workflows for 3+ LCLS crystallography/diffuse scattering simulation and refinement pipelines, including test harnesses
- Deployment guide for SLAC infrastructure
- Public repository with documentation