# Agentic Orchestration of HPC Applications

## A study using Google Gemini in Cloud

**November 19, 2025**

Vanessa Sochat and Daniel Milroy
Lawrence Livermore National Laboratory

## Abstract

Large Language Models (LLMs) are changing the career of software engineer from one of independent creator to collaborative assistant. As open source LLMs become easier to use and deploy, the impact on software engineering is yet unclear. The developments promise new degrees of autonomy, where categories of human work and decision making are replaced by autonomous, goal-oriented systems. This transition necessitates novel architectural paradigms that unify human and machine intelligence, and solid understanding of the strengths and limitations of LLMs. In this work, we design agents to intelligently deliver the entire lifecycle of a high performance computing (HPC) application experimental run in cloud – creation and build of a container image, deployment in Kubernetes, optimization with respect to a metric of interest, and orchestration of a scaling study. We pursue this task for 4 well-known HPC applications to build multi-platform images and optimize across a set of 21 instance types in Kubernetes. We demonstrate outcomes that are comparable to experts, designs that improve work time to completion, and review suggested best practices and strategies for agentic design and collaboration.

## 1 Introduction

The revolution of Large Language Models (LLM) for usage in artificial intelligence (AI) and machine learning (ML) workloads has taken the global research community by storm. The high performance computing (HPC) community is a representative subset of this user base that can benefit from using AI/ML models to advance science. The interfaces to interact with LLMs are typically services or agents. An agent is an application that can receive instructions and respond with context-aware, meaningful text in coordination with AI services [1]. As agents improve, they will become more tightly integrated into the life-cycle of research and scientific computing. Successful integration depends on understanding strengths and limitations and the strategy to holistically combine human goals with LLM productivity. A LLM is only as powerful as its ability to focus on a scoped task, and an agentic team of LLMs and humans will best achieve a desired outcome with proper protocol for guidance and validation. This collaboration can be domain-specific. An agentic team that is best suited for science likely needs more validation and checks than one intended for entertainment. As natural language becomes a part of a new type of software to create revolutionary new systems, the feedback loop between agents and humans must become tighter [2].

A variety of developer frameworks [3, 4] and hosted services are available for inference, including Google's Gemini, Anthropic's Claude, OpenAI's ChatGPT, and Microsoft Copilot, among others [5]. These agentic AI systems are being used in medicine, education, and science [6, 7, 8]. They are designed to derive objectives from prompts, and synthesize multiple data sources while using internal tools to plan, reason, and execute complex, multi-step tasks without human intervention. While the structure is not typically transparent to the user, we can speculate that these systems have checks and balances, can adapt to failure, and use a mixture of experts (MOE) [9] to target sub-tasks to models with tuned expertise. Although a response to the user may appear as a single, cohesive output, it likely results from synthesis of multiple sources of information, multiple agentic steps, each with support from external tools. These tools might include symbolic solvers for formal logic, code interpreters to produce accurate mathematical results, and search engines or databases for general knowledge retrieval. A planner that initially processes the prompt is likely responsible for orchestrating agents and tools into an execution pipeline. Industry vendors are ahead of the game to structure these interactions, developing the Model Context Protocol (MCP) at the end of 2024 to standardize AI system access to external data sources and tools along with client interactions [10], and application programming interface (API) definitions [11].

While corporations can afford and utilize these services, scientific groups are more cost constrained and can be limited by institutional regulations. While downloading models and using them on-premises might be allowed, the additional tools and agentic AI systems are not included. The inability to fully utilize hosted agentic AI systems hinders the ability of the scientific community to use them for scientific advances and learning. The requirement to deploy internal model servers on on-premises clusters is redundant. It adds management complexity along with capital and operational costs. However, the challenges to using agentic AI systems for computational science does not deem the task impossible. Scientists can still learn from hosted agentic systems to develop infrastructure and software approved for scientific institutions. A paradigm shift from laborious manual task execution to proactive, goal-oriented autonomy would be a considerable benefit to scientific discovery.

### 1.1 Agentic AI for Computational Science

Research specific to agents oriented to build, deploy, and optimize HPC applications is in its infancy. We might imagine a future when AI systems are used as first class citizens for scientific workflow development and execution. It is unclear how agentic steps could integrate with traditional HPC workload managers that rely on static, heuristic-based approaches or graph-based algorithms, often requiring substantial time to wait in a queue that would further exacerbate the ability of multiple agents to work together in real-time. However, in that many agentic AI systems decompose high-level tasks into directed graphs of compute, systems that support agentic orchestration [12] or graph-based schedulers such as the Flux Framework [13] could offer promise as frameworks to match resources to agent needs. Work to integrate AI into scheduling is already years old, however it is often focused on prediction of a specific attribute such as runtime remaining for the job [14]. An agentic framework to orchestrate a multi-step workflow that might require coordination with humans adds additional complexity to the task. Agents may not yet be able to act as an HPC scheduler, but they can optimize resource requests for a specific application.

Scientific workflows have traditionally used a combination of manually executed tasks to prepare for using workflow tools that have directed acylcic graph (DAG) architectures. For example, the task to build and deploy an HPC workload might start with a manual build of a Dockerfile by an application expert followed by orchestration of the resulting container via a scientific workflow tool [15] or abstraction in Kubernetes [16]. These DAG-oriented workflows are deterministic in that they do not need LLM routing [17] – an LLM agent-based approach promises to accelerate task execution in comparison with human operators. These details might include runtime parameters, resource requests, topology, and binding.

Agentic AI offers the opportunity to re-imagine this sequence of steps. While many scientific-based AI tools focus on hypothesis generation, data analysis, experiment design, and scientific writing [18, 19], a less studied and more challenging area of work is in the space of workload orchestration. A successful agentic framework for executing workload steps would require a collaboration between agentic step and human experts to get to a desired outcome. Akin to hosted services [20] and workflow tools, a top level manager would handle decision making, supplemented by human intervention to add further validation of experiment progress. In our work, we prototype and study such a tool – an agentic team of step-level experts orchestrated by a top level manager and supporting sub-agents to achieve a full build, deploy, and optimization of HPC applications. In this work we make the following contributions:

- Software prototype for agentic orchestration
- Execution and analysis of 5 HPC applications in Kubernetes
- Comparison of outcomes against human expertise
- Best practices for AI and human collaboration

We start with an introduction to agentic roles and definitions in Section 2.1, and describe our methods to orchestrate agentic execution and respond to failure. We describe experiments and move into results in Section 3. We finish with a discussion of the lessons learned, best practices for collaboration between agents and humans, and ideas for the future.

## 2 Methods

### 2.1 Overview

We aimed to orchestrate the entirety of a build, deployment, and optimization cycle for 5 HPC applications of interest in the leading cloud orchestration framework Kubernetes [21]. We chose Kubernetes because it provides declarative management and structured, programmatic interactions that can be easily used by respective agents. Our applications include a set that vary in difficult to build, including LAMMPS, AMG, Kripke, the OSU Benchmarks, and Laghos, all of which we have deployed and described in previous work [22]. For our setup, to give the optimization step a choice of resources, we aimed to use an autoscaling setup on Amazon Web Services (AWS) that includes 21 instance types (Table 1) that include each of arm64 and amd64 platforms. Instance sizes were chosen to be approximately $3.00 or less. Providing a dimension across platforms, micro-architectures, CPU, and memory would give the agents four dimensions to consider when selecting an instance type.

**Table 1:** Instance Types for Agentic Selection

| Instance | Processor | Cores/Freq. | Mem. | Cost/Hr |
|---|---|---|---|---|
| c6a.16xlarge | AMD EPYC 7R13 | 32/3.6GHz | 128GB | $2.448 |
| c6i.16xlarge | Intel Ice Lake | 32/3.5GHz | 128GB | $2.72 |
| c6id.12xlarge | Intel Ice Lake | 24/3.5GHz | 96GB | $2.4192 |
| c6in.12xlarge | Intel Ice Lake | 24/3.5GHz | 96MB | $2.7216 |
| c7a.12xlarge | AMD EPYC 9R14 | 24/3.7GHz | 96GB | $2.4634 |
| c7g.16xlarge | AWS Graviton3 | 64/2.5GHz | 128GB | $2.32 |
| d3.4xlarge | Intel Cascade Lake | 8/3.1GHz | 128GB | $1.998 |
| hpc6a.48xlarge | AMD EPYC 7R13 | 96/3.6GHz | 384GB | $2.88 |
| hpc7g.16xlarge | AWS Graviton3 | 64/2.6GHz | 128GB | $1.6832 |
| i4i.8xlarge | Intel Ice Lake | 16/3.5GHz | 256GB | $2.746 |
| m6a.12xlarge | AMD EPYC 7R13 | 24/3.6GHz | 192GB | $2.0736 |
| m6g.12xlarge | AWS Graviton2 | 48/2.5GHz | 192GB | $1.848 |
| m6i.12xlarge | Intel Ice Lake | 24/3.5 GHz | 192GB | $2.304 |
| m6id.12xlarge | Intel Ice Lake | 24/3.5GHz | 192GB | $2.8476 |
| m7g.16xlarge | AWS Graviton3 | 64/2.5GHz | 256GB | $2.6112 |
| r6a.12xlarge | AMD EPYC 7R13 | 24/3.6GHz | 384GB | $2.7216 |
| r6i.8xlarge | Intel Ice Lake | 16/3.5GHz | 256GB | $2.016 |
| r7iz.8xlarge | Intel Sapphire Rapids | 16/3.9GHz | 256GB | $2.976 |
| t3.2xlarge | Intel Skylake | 4/3.1GHz | 32GB | $0.3328 |
| t3a.2xlarge | AMD EPYC 7571 | 4/2.5GHz | 32GB | $0.3008 |
| t4g.2xlarge | AWS Graviton2 | 8/2.5GHz | 32GB | $0.2688 |

Cloud instance types available for selection.
Instances were selected to be under $3.00 to provide a 4 dimensional gradient.

## 2.2 Agents

We designed an agentic framework with a preference for simplicity. While several libraries exist to implement MCP with communication via http or stdin [10], for our own learning we chose to minimize external dependencies and design each agent with a common base class to query the Gemini API and perform a function of interest. While most MCP agent APIs are expecting request and responses exclusively in text, we wanted our design to be oriented around a directed graph, and based on execution of controlled commands that resulted in a clear result (e.g., a return code) that would not require returning to an LLM to complete an interaction. In this controlled environment, multiple agents that work together form an agentic system. Agentic systems contain different types of agents that operate at different path lengths from the root. Together, they form an agentic graph.

**What is a step** During the execution of a workflow, multiple tasks are connected by inputs and outputs and end in a final state. Each task accepts inputs, and executes a function on the inputs to transform them into outputs. Any tasks in a workflow path are dependent. Inputs, outputs, and processing together define a step. The inputs and outputs between two adjacent steps form a shared context. Multiple steps (nodes) that have dependency relationships (edges) and a shared context form a graph. In this graph, a step A is adjacent and directed from A to B if the output of step A serves as the inputs to step B. As an example, the resulting container unique resource identifier (URI) is an output of a build step, and could be used as input to a deployment step.

**Step Agent** A step agent is responsible for a specific task. A single step agent can act as an independent unit or entity, and can be represented as a node in a graph. Two step agents can be adjacent in the graph and joined by an edge if their inputs or outputs are compatible. For example, a build agent builds an image before it is provided to a deploy agent to test. The structure of any step agent is simple: a controlled set of tasks specific to the agent that are combined with structured prompts. The prompts are scoped to derive input (parameters, configuration files, or supporting regular expressions) to direct a task. Input comes by way of the context (Section 2.2), which includes pre-defined variables specific to the task at hand (e.g, a container URI for a build agent) and a general *details* section where a user can provide freeform text instruction to the agent. Each step agent is implemented with actions and validation checks for the task. For example, a build agent takes a response from the LLM and writes a Dockerfile to a temporary staging directory, and collects output and error to return as feedback to the LLM. A deploy agent is tasked to generate a YAML manifest for a particular container, and the LLM response is checked for that container before attempting a deploy. Step agents use a conversational client, so new attempts are made with memory (model context) of previous failures. Each agent instantiates its own LLM model, and this is done so that responsibility does not bleed between agents. For our agentic system, we define step agents for a Dockerfile *build*, a Kubernetes deployment for a *Job* and Flux Framework *MiniCluster*, an *optimization* task, and a *scaling* decision.

**Build Agent** The *build* agent is an expert at building Dockerfile scoped for an HPC application, and is allowed to design the Dockerfile to be given to a *docker build* or *docker buildx* command in a sandbox on the user system. The execution of the build is done by the agent's class, and provides the logic to check the return code, and either return to the manager to proceed to the next step, or in the case of a build error, send the output to a debug helper agent to diagnose the issue. The debug helper agent then provides scoped feedback to the builder agent to retry. Multiple build attempts form a cycle in the graph, and a successful build (return code of 0) determines breaking the cycle and moving on to the next step. Context variables for the build agent include a *container* URI, an *application* to build, an *environment* and *platforms* to build for, and whether to *push* or *load* the image into Docker. A push requires the build environment to have permissions for the task, and the build agent is only allowed retries up to a maximum number of attempts set for each step.

**Deploy Agent** The *deploy* agent is responsible for the single task of taking an instruction for an application of interest and a Kubernetes abstraction (e.g., Job or Flux Framework MiniCluster [16]) and successfully running the application in Kubernetes. The agent is instructed to deploy a container for a specific environment and application, and with additional context provided in the details section. The deploy agent class validates the manifest generated by attempting to load it into YAML, and then checking that the container URI matches what was requested. The class then writes the manifest to a temporary deploy directory, and applies it to the cluster with *kubectl*. The details in the manifest including the container, command, size, and resources, are decided by the LLM and can be influenced by details in the prompt. The application pods are then monitored, and the class looks for changes of state from Running to Completed or Failed. A maximum waiting time for a monitored Pod to complete can be set by the user, which allows for handling cases such as timeout due to error or insufficient resources. The class can handle edge cases such as timeout, out of memory errors (*OOMKIlled*), or (in the case of Flux) the request being unsatisfiable. The class handles even unexpected deletion (a pod disappearing that previously existed). In the cases of an error code, the error along with cluster diagnostics (Pod logs, and events) are sent first to a debug agent to determine and summarize the issue. This summary, or a description of what happened (e.g., "Your last attempt timed out") is sent back to the LLM with a request to retry. The deploy is attempted up to a maximum number of attempts or a successful case: the Pod (Job or MiniCluster) having a Succeeded status.

**Optimization Agent** The optimization agent can be viewed as a child of the deploy agent in that it is instantiated by the deploy agent, and run after a successful deploy to improve upon a metric of interest. The metric of interest must be defined in the context details – an *optimize* variable – that triggers its creation. The optimization agent works as follows. A completed result log and manifest are passed forward in the context from a deploy step. The prompt instructs the agent to optimize for the metric of interest, and return a JSON structure with variables to update the previous manifest. For the first execution, there is no means for the optimization agent to know what the metric of interest value is from the provided log, and so a child class (a helper agent instantiated on the class) is instructed to generate a regular expression to parse the metric of interest from the log. The result parsing helper agent receives a regular expression back from the LLM, and tests it against the log. Validation consists of ensuring there is a match, and then asking for human validation. The user deploying the pipeline is given the task to say it is correct (yes), incorrect (no), or to provide custom feedback (feedback). The result parser agent runs in cycles until a successful result – one that passes automated checks and is approved by the human assistant. Deriving the regular expression is a task that only needs to be done once for an experiment, as the result is cached and reused.

The optimization agent LLM can then be provided with a value for the current metric of interest, the previous log and manifest, and instruction from the user via prompt details. The LLM is instructed to return a formatted JSON object with fields that match the provided manifest. The JSON response is used to update the current manifest, and the update task is handled by another helper agent. This means that the optimization agent takes a successfully executing deploy step, and changes the manifest to improve the provided metric of interest. Each parsing of an attempt with the result parser saves the metric of interest, allowing for each retry prompt to the optimization agent to have a clear list of values that show the progression of work. In addition, on the first prompt, a *kubectl explain* is done for the resource of interest (e.g., Job or MiniCluster) to provide the exact fields and descriptions of the manifest that can be changed, and the Kubernetes Python SDK is used to get a summary of nodes (counts and resources) that are available for using. A *resources* field in the plan is also allowed for the user to provide explicit instructions about what resources to use. The *resources* field is especially useful for autoscaling clusters, where nodes available might not be present in the cluster. The optimization agent is required to return an updated manifest, a decision to *RETRY* or *STOP*, and a reason. The optimization agent is triggered to run when the user has defined optimization instructions in the plan for the step. The design of the optimization agent is clever in that to execute a test of a new configuration, it returns a call to the deploy agent, however, editing the context to indicate an optimization is in progress. An optimization in progress changes the execution flow to return to the optimization agent on a successful deploy. In the case of error, execution returns to the debug agent and then deploy to resolve it before optimization is continued. While the current implementation uses the optimization for Kubernetes, it is not tied to Kubernetes. The same optimization agent could be used by a bare-metal execution deployment agent and tasked equivalently.

Due to common interactions with Kubernetes to get logs or status, the two deployment agents share a common set of functions and underlying Python class. These controlled interactions are how we give the agents access to the execution environment. It is up to our software to carefully monitor each execution and determine when the state is erroneous. For example, deploying an abstraction to Kubernetes could fail immediately if the YAML file is invalid, or later if execution starts and fails. Collection and filter of error information from the right sources is a strategic task we must implement into the software to ensure that a failed attempt can be debugged properly. Direction of the next agent to make a request to with a specific update to the prompt is essential. For example, not capturing a timeout or out of memory issue and informing the agent can lead to updates to the manifest that are nonsensical like increasing the problem size. In the case of failure, a cleanup is typically required to prepare for the next attempt.

**Prompt Structure** Each agent typically has scoped prompts to perform and retry a task. While we initially designed prompts as generic string templates populated with user-provided variables, we quickly learned that added structure improved clarity and thus agentic ability to do a task. For each prompt, we define a clear persona, context, task, and set of instructions. The persona clarifies the agent's role. The task and context clearly define the goal for the interaction. The instructions are concise, single lines that describe what an agent *MUST* and *MUST NOT* do. Using capital letters to provide emphasis makes a difference. When we do not add this emphasis, we found the agents more often do not follow the instruction. We found that using regular expressions to filter text noise from prompts improves goal attainment. For example, when debugging output from the build agent, many thousands of lines are generated by apt-get during an Ubuntu build. These extra lines distract from identifying the core issue and can be filtered before sending to Gemini. This reduced our token count (Section 2.3) from the order of 300,000 tokens down to a few thousand, and reduced time to parse the response and cost.

**Helper Agents** A helper agent is an agent used by a step that is primed to perform a specific task, and typically one that does not require conversational memory. When agents are created, API clients are instantiated with the class that can either send one-off model messages, or a message to continue a conversation. We consider the conversation (a model with context) a form of superficial memory, as subsequent prompts are informed by previous ones. In the case of helper agents, we would not want any bias based on a previous prompt. Our two use cases for helper agents were debugging and synthesizing errors (i.e., identifying the issue and summarizing it succinctly), and parsing a result from a log. We call these helpers the *debugging* and *result* agents, respectively. In addition to summarizing and suggesting a fix for an error, the debugging agent can return execution flow to the manager if it determines that the step in question cannot resolve the issue. For example, a missing library in a container cannot be resolved by a deploy agent; the build agent must fix the underlying container build. The result agent is used by the optimization step agent to parse a metric of interest from a log. The result agent is prompted to generate a regular expression that is applied to the log. When a match is returned, it is interactively presented to the running user to validate correctness. A working regular expression only needs to be derived once for a run, as it can be validated and used for subsequent parses. This is an example of a human-validated step, and collaboration between the running human user and an agent. We found this kind of human input was needed to validate the output of the agentic task. The pattern of the agent performing a task and human validation is one that can be extended to other use cases.

**Manager** As we started to design single agents that were oriented to perform one task, it became clear that an agentic team would require a manager to orchestrate the sequence of steps. The manager would take as input a plan derived by a human defining a sequence of steps. The manager is then responsible for orchestration of step agents. Each step has a defined context. For example, the build agent minimally requires an application name and environment to tune the build for. The manager is the root of the graph, executing individual steps, and having flow return to it in the case of error or completion. Each step has a maximum number of attempts, or up to the point when a debugging agent directs flow back to the manager. The manager then saves an output file with metadata when the workflow completes. The manager combined with step agents and helper agents form an agentic team (Figure 1).

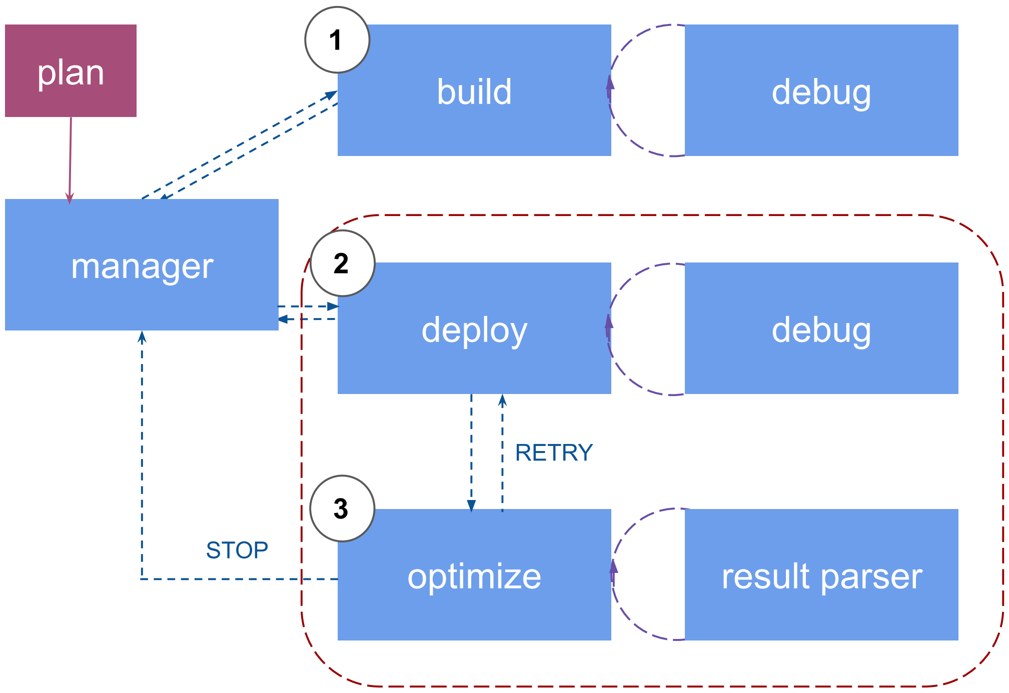

**Figure 1:** Agentic AI System. The manager receives a plan from the user, a YAML specification with a series of step agents and context for each. The manager orchestrates executing step agents defined in the plan, allowing each step to work independently up to a maximum number of attempts, and return to the manager if decided by a helper debug agent or on success. The optimization works with the deploy agent until a decision is made that the run is optimal.

**Context** A shared context object is responsible for sharing state between agentic steps, and provides an interface for setting and retrieving input variables. Each step requires a specific set of inputs to be provided in the context, and inputs can be populated in several ways. The developer user adds a function *_add_arguments* to the agent class that returns a subparser. Any argument defined there is added to the agent context, and can be customized via the command line, the plan YAML file, or by an agent setting it directly on the context. As an example, a build agent by default will generate and set a *container* URI on the context that matches the build. When the context is passed to the deploy agent, the input *container* is not only instructed for use, but validated by the agent class to appear in the output (a manifest to test). The validation ensures that cycles are not wasted testing a URI that does not exist. As another example, a user might create a plan for a previously built container, and provide the URI as the *container* field under the context. This design allows for each agent to define a different set of context variables as inputs, and flexibility for them to be set by a human, or another agent.

The manager loads the initial context from the user-defined plan, and each step is provided with the context for execution. Common variables between steps that aid with communication include a final *result* for the current step, and an *error_message* that provides an indicator if something went wrong that needs debugging. For example, when a build is erroneous, the full output is given to the debugging agent with instructions to process and summarize the issue. The more succinct description and suggested fix is then passed back to the build agent for another attempt. The build agent uses a conversational client so context about previous attempts is carried forward for subsequent attempts.

**Explainability** As we develop our software, we will need to understand the weaknesses in both the design and instructions that are provided to different LLM agents. The extent to which we can track and explain the decisions of the LLMs is the degree of explainability [23]. We found it useful to display inputs, outputs, and prompts in a visually appealing way in the terminal, and always prompt the LLM agents to provide rationale for choices made. While this is not standardized collection of provenance [23] it allows us to track what is going on at a level appropriate for being able to improve upon it. Generally, when a developer observes an agentic team at work, they want to know what prompt or input data led to a decision, what information might be missing, and what factors led to surprising or improper behavior. In addition to terminal output, the manager saves all intermediate steps and output generated. Each step agent saves custom metadata, function timings, and relevant metrics of interest.

**Controlled Interfaces** The main tool that our agents use is an API client to query the Google Gemini API. We chose Google's Gemini because we had the opportunity to use credits, and Gemini is a foundation model that would be capable of agentic work. Each agent has defined environment interfaces that use the Python *subprocess* library to execute commands for building or deploying configuration files. We typically do not allow agents to write commands that are executed to the system, but instead to populate the content of configuration files used. We currently do not provision connectors to other external databases, search engines, or APIs to supplement the agent, as we assume Google's Gemini does this on Google's servers. The closest that the LLM will get to code execution is writing regular expressions for log parsing to be used by the result agent. We looked into a potential security risk with allowing execution of regular expressions, and saw the risk of a ReDOS (regular expression denial of service [24]). From a hosted service like Gemini provided by Google we saw this as unlikely, and if it happened, would be a bug in a product to report to Google. We recognize that allowing agents to write, for example, database commands would be much higher risk, however we did not need that functionality. As an extra layer of precaution, all experiments are run on cloud virtual machines. These controlled interfaces are essential for agents to interact with and adapt to the environment.

## 2.3 Experiments

**Single Node Experiments** We aim to build, deploy, and optimize 4 HPC applications in Kubernetes on a single node. We provide instructions in the plan to use the Flux Operator *MiniCluster* custom resource definition (CRD) [16] to deploy an HPC cluster to run each application. We give the optimization agent step 21 types of instances (Table 1) to select from, and request deployment on one node to maximize figure of merit (FOM). We start with just one node to assess performance of the agentic team when scale is not a factor. For each of LAMMPS, AMG, Laghos, and Kripke, we give the optimization agent instructions for how to make a decision to retry or stop, along with an instruction for an optimization strategy. For all applications we provide instructions to use a smaller problem size for testing, and increase to a larger size for the optimization. In all cases, the optimization agent step is required to return a decision value of *RETRY*, or *STOP* with a reason and resources if a retry is needed. We will test the following optimization strategies, each of which instructs the agent to maximize the FOM with additional instruction. Each of the following are implemented by way of distinct prompting.

- **LLM decision**: Instruct the agent to optimize the FOM and decide when to retry or stop
- **user function**: Require the agent to execute and follow an instruction exactly.
- **user guided function**: Provide metrics about scaling with decision and optimization hints.

For all optimization strategies, we provide in the context a description of resources available, including instance types (CPU and memory) in the autoscaling cluster. For the first optimization strategy, *llm decision*, we leave the entire decision up to the agentic model. This strategy can be considered free-form in that the prompt does not give the LLM an instruction or algorithm for making a decision. For the second strategy, *user function*, we provide a function in the prompt that calculates a configuration to use for the next step, and ask the agent to run it. This strategy is conceptually similar to using an MCP server in that it mimics the LLM executing a function and using the exact output. An example user-provided function signature is provided below:

```
def optimize_amg(
    problem_size: ProblemSize,
    topology: Topology,
    total_instance_cpu: int,
    total_instance_memory_gb: int,
    cores_per_node: int,
    threading_hint: int,
    current_fom: float,
    executable_command: str,
    threshold: float = 0.90,
    ideal_fom_per_core: float = 4.5e8,
    efficiency_sweet_spot: float = 1.0e6,
    retry_scale_factor: float = 1.5,
    dofs_per_gb: float = 1.2e7,
    memory_safety_factor: float = 0.95
) -> Dict[str, Any]:
    """
    Acts as an optimizer for a single-node run, suggesting new parameters
    and a full execution command. The only STOP condition is meeting the
    performance threshold or being unable to improve.
```

The function returns the expected "decision" as *RETRY* or *STOP*, along with key value pairs to update the execution. The output provided is given to the equivalent helper LLM step to update the current manifest with changes. Finally, the third strategy is an intermediate between these two extremes, taking in the current resources and returning a decision to *RETRY* or *STOP* with a strategy hint, and providing instruction to also return resources and instance type. An example is shown below:

```
def evaluate_amg_run(
    # Problem size
    problem_size: Any,
    total_ranks: int,
    nodes_used: int,
    # Hardware constraints
    cores_per_node: int,
    memory_gb_per_node: int,
    threading_hint: int,
    current_fom: float,
    threshold: float = 0.90,
    ideal_fom_per_core: float = 4.5e8,
    efficiency_sweet_spot: float = 1.0e6,
    dofs_per_gb: float = 1.2e7,
    memory_safety_factor: float = 0.95
) -> Dict[str, Any]:
    """
```

The main difference between a user-function and a user *guided* function is that the latter returns metrics that the LLM can use to make its own decision. Specifically, the agent runs a function to derive the performance ratio, scaling efficiency, memory utilization, and application-specific metrics to help guide the decision.

**Multi-Node Experiments** We perform a modified version of the single node experiments that instruct the agent to do the same build and deploy across multiple nodes. We decided to use the *m7g.16xlarge* due to reasonable cost and best availability across the top 3 contenders as determined by testing. Importantly, the selected instance needed to support the AWS Elastic Fabric Adapter (EFA) suggested for HPC workloads [25]. EFA allows network packets to bypass the operating system and go directly to the device for improved performance across nodes. The agents will be instructed to build OpenMPI with libfabric intended for the AWS EFA, and that they should maximize FOM and deploy each application on up to 4 nodes. We will use the best performing strategy identified in Section 2.3. Since we are running across nodes, we will add in the OSU Benchmarks in place of Laghos to test the ability of the LLM to figure out point to point and collective Message Passing Interface (MPI) calls.

**Scaling Study**

For a final experiment, we challenge the agents to complete the entirety of a scaling study. For this work, we test LAMMPS, Kripke, and AMG, and pin the container build to a known working variant from our previous experiments. A scaling agent is added to the agentic team, and is prompted to return a response that decides when to continue or stop scaling toward a user-defined goal (Figure 9). The optimization agent is provided a modified prompt with the context of the scaling study that instructs to change the problem size only at the first size, and then hold it constant (strong scaling).

An example optimize and scale plan is provided below for LAMMPS, demonstrating that both the scale and optimize agents run under the deploy (MiniCluster) agent. The execution of the deploy agent is allowed a maximum number of 10 attempts, and an execution timeout of 300 seconds. The container provided was previously built by a build agent, and provided to save time and not rebuild each time. The environment definition is minimal, as the details for the cluster size and resources are provided programmatically. To ensure that the execution loop has fewer edges, no returns are allowed to a human or to the manager. The scale agent is instructed to maximize FOM and pin a problem size from the smallest size, N=1, and to stop when the application stops strong scaling. The optimize instruction is much more specific, placing emphasis on points with *MUST*. Emphasized instructions typically result from an agent improperly performing on a testing attempt.

```yaml
name: Scaling Study for LAMMPS
description: Scale lammps up to 5 nodes.
plan:
- agent: minicluster
  context:
    environment: "AWS CPU instance in Kubernetes"
    container: ghcr.io/converged-computing/fractale-agent-experiments:lammps-reax
    max_attempts: 10
    max_runtime: 300
```

```
      allow_return_to_human: false
      allow_return_to_manager: false
      sizes: [1,2,3,4,5]
      scale: |
        Strong scale lammps to maxime the FOM and minimize running time at each size.
        You MUST choose a problem size at the smallest size (1) that you keep constant.
        Stop when you determine the application is no longer strong scaling.
      optimize: |
        You MUST maximize the LAMMPS FOM, *atom steps per second.
        When parsing the log you MUST consider it could be Matom or katom.
        You MUST NOT change parameters after the first scaling size (1).
        You MUST increase problem size at the first scaling size (1) until the job times out.
        You MUST increase the problem size ONLY at the first scaling size.
        At large sizes, you MUST adhere to the same resource limits, requests, and selectors.
        You MUST adhere to ONE node (64 tasks) for the initial optimization.
```

The study is a collaboration between the scaling, deploy, and optimization agents, where the optimization agent derives the configuration, the deploy agent is tasked with running work at each subsequent size, and the scaling agent response determines when to stop. A human is still asked to intervene to validate the parsing of a FOM from the log. We plan to instruct the agent to build and deploy a multi-node cluster starting at size 2 up to a maximize size of 32 nodes, and to choose the instance type for each that performed optimally for the experiments described in Section 2.3. We instruct the agents to stop when strong or weak scaling has ended, and instruct that each decision be explained with evidence from relevant literature.

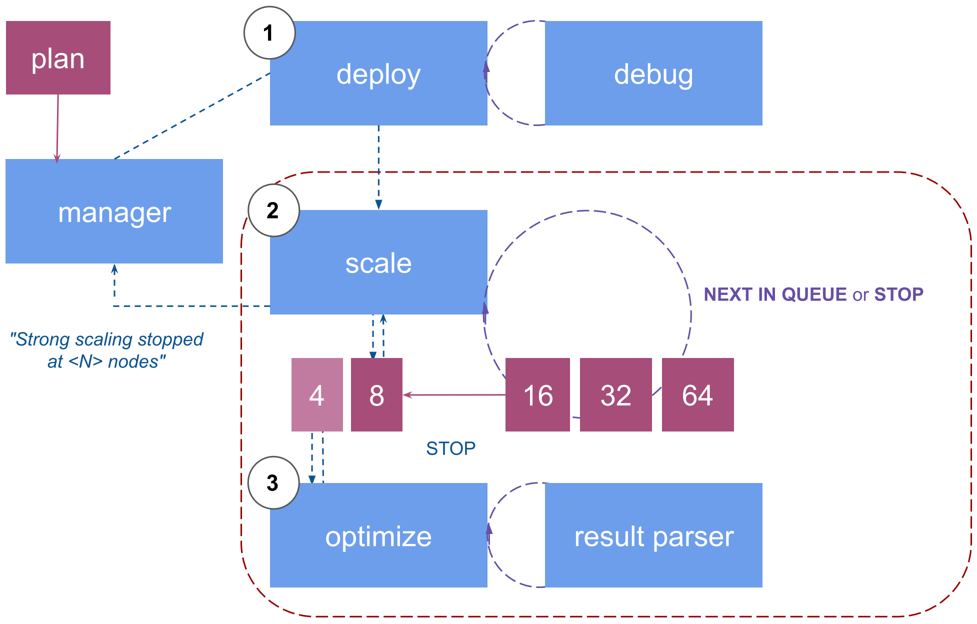

**Figure 2:** Scaling Study Agentic Team for a hypothetical scaling study for 4 to 64 nodes (maroon boxes). The deploy agent generates a manifest in a working state. The scaling agent receives instruction to start the study, and updates the prompt to deploy at each requested size. At the smallest size, the scale agent directs execution to the optimization agent, which decides on a configuration for subsequent sizes. At subsequent sizes, the scaling agent assesses the previous result, and decides to stop or proceed to the next size. Execution of subsequent sizes is done by the deploy agent. The scaling agent reports evidence and reasons for its decisions. Helper debug and result parsing agents assist primary step agents.

**Result Assessment** Across experiments, we are interested in the agents' choices for the Docker build, the Kubernetes *MiniCluster* specification, and the decision sequence to retry or stop. Each application will be run for 10 iterations, with a total number of retries of 10 for each step agent, and 15 retries for the manager. Each application will be allowed a maximum running time of 5 minutes, and a timeout will cancel the run and alert the agent that the application run has timed out. Each result will be assessed by our team for quality, and agent logs will be inspected to learn how we can improve and tighten the orchestration. Our team has expertise in building and deploying HPC applications in cloud environments [22]. For each application, we will consider:

- Overall performance of final optimized variant
- Total time to complete each step
- Number of attempts to successfully complete a step
- Choices step agents take between and within steps
- Dockerfile build logic correctness and completeness
- MiniCluster design logic correctness and completeness
- Decision of when to *STOP* versus *RETRY*
- Reasons for failure

For reasons for failure, we will assess whether an additional prompt or requirement on our part could have better guided the agentic team to a successful outcome. We will run experiments in sessions, and make adjustments to prompts and software design to improve upon our initial attempts.

### 2.3.1 Token Counts and Completion Time

A token is a chunk of text that an LLM processes, which can be a single word, but can also be part of a word, punctuation, or a space. For Gemini, a token is about 4 characters [26]. We are interested in the relationship between token counts for requests and responses, and time (seconds) for completion. For example, it is not clear if providing more tokens in a request leads to longer processing time. It could be feasible to have a short response that requires more processing time by the LLM.

## 3 Results

We performed single node build, deploy, and optimization experiments for 4 proxy applications (AMG, LAMMPS, Kripke, Laghos) across 21 instance types on an autoscaling cluster (Section 3.1) followed by a multi-node (N=4) optimization study to add a component of networking (Section 3.3) benchmark (OSU). We finished with a scaling study (Section 3.4) that extended the experiments to deploy and optimize across sizes. Best FOM results are shown in Table 2 and each experiment group discussed below.

### 3.1 Tokens

We analyzed the relationship between token counts and seconds to complete the response (Figure 3). We did not observe any strong relationship between prompt token count and candidate token count, however a pattern reflecting the difficulty of building the application (Section 3.1) was seen in our plot. Applications that are more difficult to build, either for a human or the LLM, produced a higher token count for prompts and candidates. The higher token count is caused by the library framework sending more error output, and receiving more text content. As an example, Laghos (green in Figure 3 was hardest to build, and has the highest prompt and candidate token counts. We also see "stuck sequences" in the data, or a sequence of prompts that are similar in count that likely received similar responses from the debugging agent. This pattern – the inability of a request to return a working answer and trying something similar to return a similar result – might be predictive of an oncoming failure state. It could also be indicative of fine-tuning a response that is closer to a successful solution. We can hypothesize that the content of the prompts is similar due to what appears to be equivalent lengths. More work is needed to better identify, label, and investigate this hypothesis.

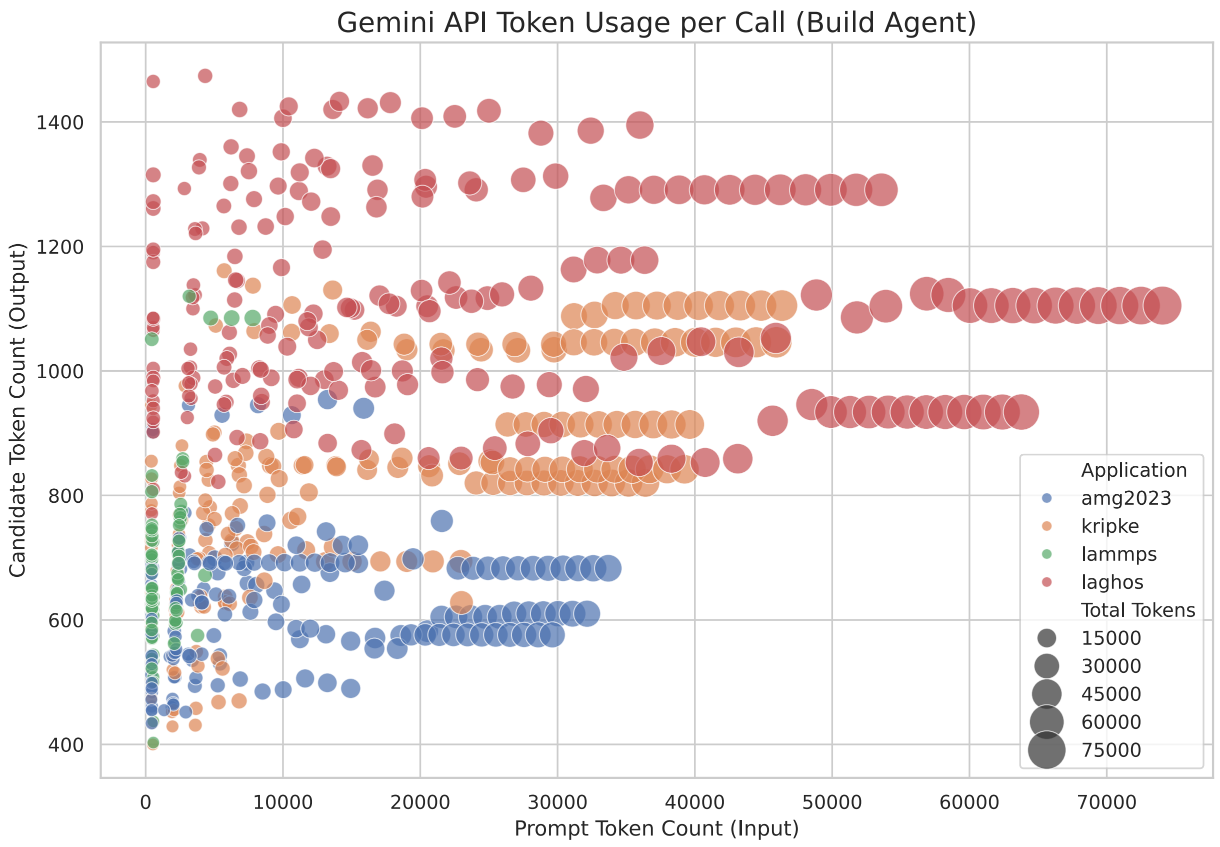

**Figure 3:** Gemini Token Counts for Build Agent. The linear pattern of receiving back a similar or equivalent output token count reflects a likely convergence on response, either closing in on a solution or getting stuck in a broken state. The ease of generating a working build for LAMMPS (green) is reflected in requiring fewer total tokens.

## 3.2 Single Node Experiments

We asked build, deploy, optimization, debugging, and parsing agents to work together with a manager to deploy AMG, LAMMPS, Kripke, and Laghos on one node. Overall success and failure rates are shown in Figure 4 and best performance metrics in Table 2. A failure indicates that the pipeline was not completed through optimization, either because build or deploy did not complete successfully. Akin to the patterns described in Figure 3, we found that the difficulty an agent had with a build or deploy step matched our perceived human difficulty, with LAMMPS being the easiest to build and execute, and AMG, Kripke, and Laghos more difficult. Subjectively, more difficult builds require more dependencies, and the dependencies need customization with respect to build flags or settings. LAMMPS is easy because it is built with a fairly straightforward *cmake* command, and an application like Laghos is much more challenging because it requires specific versions and flags for each of *hypre*, *mfem*, and *metis*. Results for different prompting strategies for one node, and for 4 nodes are described below.

**Table 2:** Best Application Figure of Merits for Experiment Types

| Application | Experiment | Best FOM | Instance Type |
|---|---|---|---|
| lammps | Single Node (llm) | $724.238$ katom steps/s | hpc7g.16xlarge |
| lammps | Multi Node | $2.50$ Matom steps/s | m7g.16xlarge |
| lammps | Scaling Study | $3.159$ Matom steps/s | hpc7g.16xlarge |
| amg2013 | Multi Node | $6.132476 \times 10^9$ nnz/s | m7g.16xlarge |
| amg2013 | Single Node (llm) | $1.602391 \times 10^9$ nnz/s | m7g.16xlarge |
| amg2013 | Scaling Study | $6.424653 \times 10^9$ nnz/s | hpc7g.16xlarge |
| kripke | Single Node (llm) | $7.338117 \times 10^{-10}$ (s/iter)/unknowns | c7g.16xlarge |
| kripke | Multi Node | $7.795293 \times 10^{-10}$ (s/iter)/unknowns | hpc7g.16xlarge |
| kripke | Scaling Study | $6.012693 \times 10^{-10}$ (s/iter)/unknowns | hpc7g.16xlarge |
| laghos | Single Node (llm) | $3.5295 \times 10^2$ megadofs/second | c7g.16xlarge |

Best FOM reported by experiment type. For single-node experiments, the best strategy is also reported. For the scaling study, the best FOM across sizes is reported. *llm*: llm decision, *user*: user guided. Laghos units are in megadofs x time steps / second

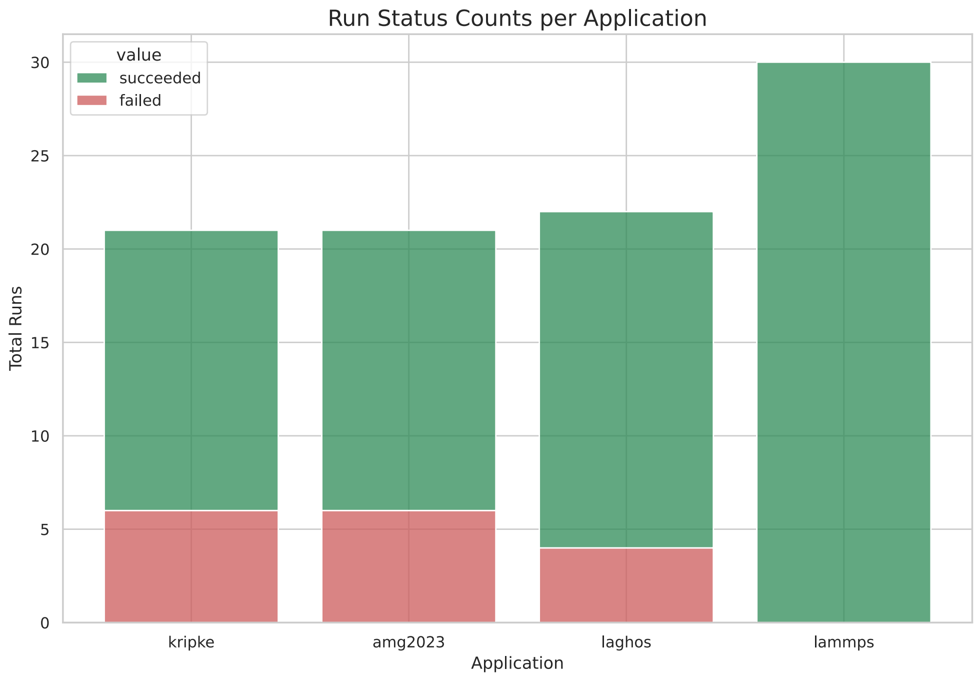

**Figure 4:** Agent Status. A failed run indicates that the application did not make it to a successful completion, meaning a build and deploy with an exit code of zero. A successful run (green) indicates that a FOM was generated. LAMMPS had more runs due to initial testing.

## 3.3 Instance Selection

Instance selection is the task for the LLM to choose a cloud instance type to assign a workload to, which can come from the running cluster resources or an autoscaling cluster hint in the prompt details. The deploy and optimize agents in our study could choose from a total of 21 instance types (Table 1). Instance selection is shown in Figure 5. For Laghos and Kripke, there was a preference for choosing the *hpc7g.16xlarge*, with the instance type selected more than 50% of the time. Looking at the agent reported reasons, we see a pattern of the first optimization attempt "selecting a 64-core, HPC-optimized instance." We hypothesize the choice is based on LLM training data that advertises the *hp7g* as the correct instance for HPC applications. When exploring multiple instance types, when the agent determines FOM has been optimized for one instance type it often decides to test a different instance type.

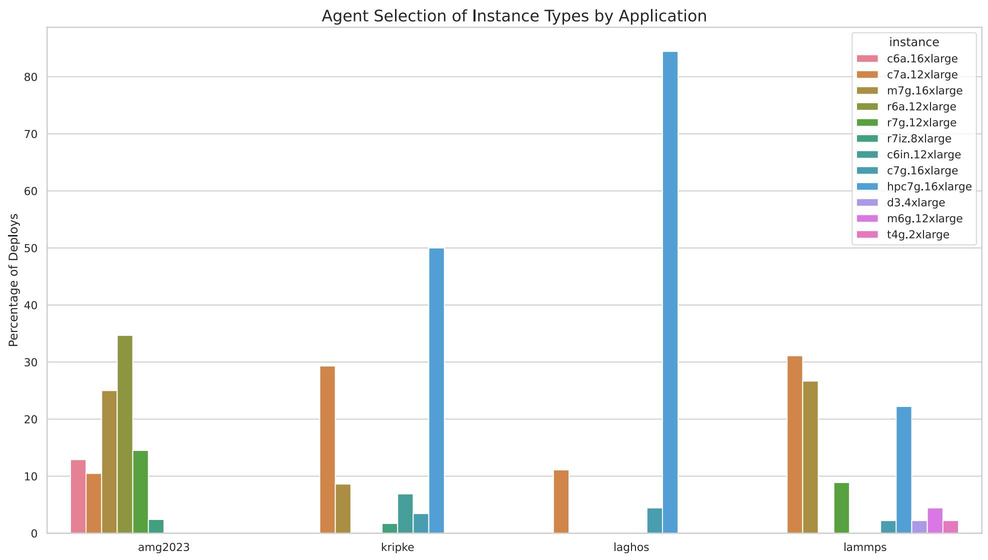

**Figure 5:** Agentic selection across experiment runs for all applications. The agents heavily selected the *hpc7g.12xlarge* for Laghos and Kripke, and did more sampling of the space for other applications.

**Common Errors** We found that the agent responses included common errors, and these consistent mistakes were good candidates to include in general prompt instructions for any application. The first issue was related to discovery of data and executables. If the build agent response did not place the final executable on the *PATH* or add a comment to the Dockerfile about data file names and locations, it was unlikely for the deploy agent to be successful. At best it would have to guess, and fail at the maximum number of attempts. The second error that was common and nonsensical was linking a binary to an install location from a build directory, but then cleaning up the entire build directory. Although this error would be fixed by the debugging agent, we suspect this resulted from different sources of information from the LLM populating the same Dockerfile. In *some* context, a symbolic link is a sane thing to do. In another context, removing the build directory is a good idea. It is not a good idea to combine the two. Finally, a consistent error across builds that was surprising was not installing certificates.

**AMG2013.** Results for AMG are included in Figure 6. Allowing the LLM to make a decision was the most successful strategy. The FOMs were $1.604893 \times 10^9$, $1.417084 \times 10^9$, and $8.048641 \times 10^7$ for the LLM decision, user guided, and user provided functions, respectively. Despite requesting *AMG2023* the agent always cloned and built the *LLNL/AMG* (AMG2013) repository. This preference could derive from the LLMs training set. The difference is significant because the problem the agent was instructed to run (2) cannot be compared between variants.

Observing the output from the agents, what became clear is that the instruction from the user func-

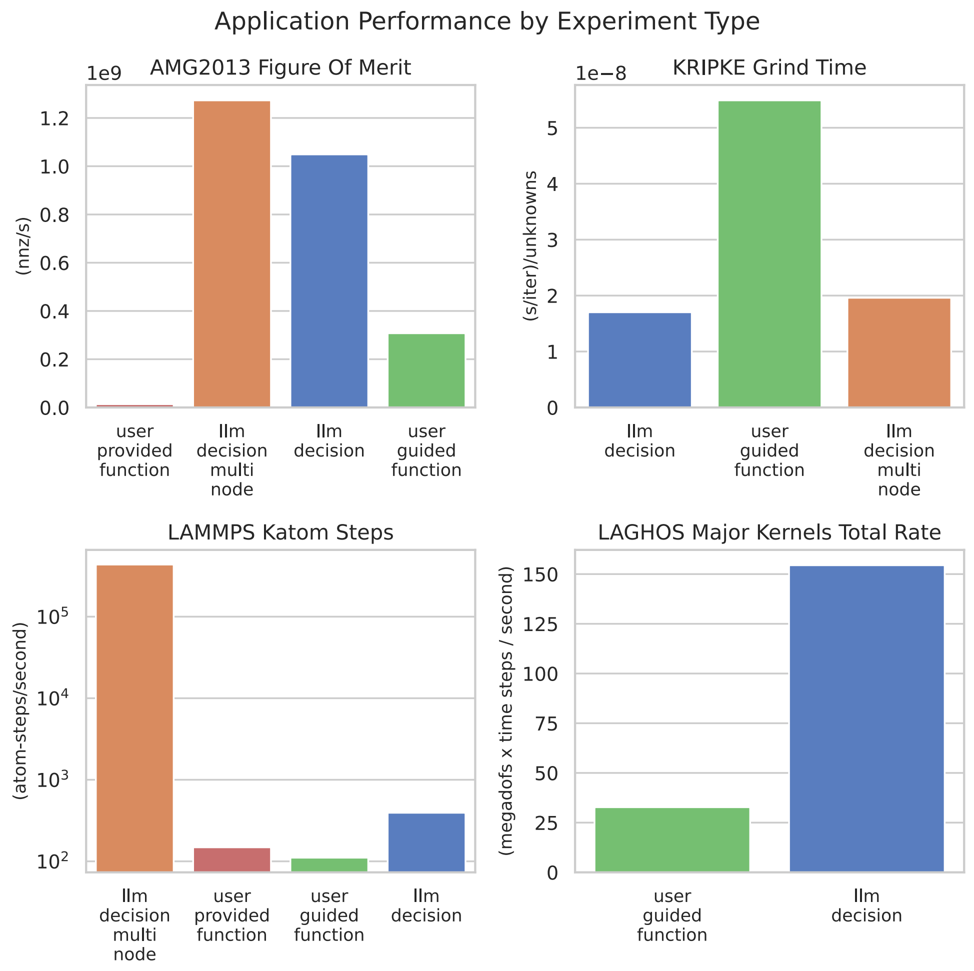

**Figure 6:** Application best FOM by Experiment Type. Allowing the LLM to decide how to optimize generally yielded the best results. For all FOMs, higher values are better, with the exception of Kripke (top right) where lower is better. The best strategy from single-node experiments was carried forward to multi-node experiments.

tion was in direct conflict with what the LLM determined was best. We think this conflict results from the lack of fidelity of the hard-coded functions in modeling an environment. The user provided function did not work beyond the testing case because the optimization function produced a resource estimate that was too large, leading to execution that would run out of memory or time out. We think that an expert on a specific instance type likely could write a well-performing function.

We learned that without providing guidance for the problem size for the first deploy (testing) step, the agent always chose a problem size that was too large and would lead to the pod being *OOMKilled*. Although the agent could adjust the problem size to be smaller each time, in practice it was not enough to get to a working result in under 10 attempts and the entire step would fail. We chose to advise the agent to choose a small problem size (no greater than 10) for testing of the initial deploy step. The subsequent information provided about how to optimize AMG for memory would then prevent the same error. If asking for a problem size that is too large is a common pattern, the agent might be given instructions for how to adjust problem sizes. Instead of a slow linear decrease, a binary search and then reaching a point when the FOM stops improving might be more efficient.

Through a collaborative process of watching experiments run, we noticed that the key to AMG running well was to force OpenMP and any math libraries to use only one thread per MPI rank. While this insight could have come from an expert user, it was the LLM that reminded us of threading and we could reliably give the advice to the agent to make it the application run performant.

**LAMMPS.** LAMMPS results are shown in the third panel of Figure 6. While the user provided function mean value outperformed that of the user guided function, the variance across runs (not shown) was twice as large, suggesting that the result is not consistent. The LLM agentic decision outperformed both, leading to a FOM that was over 2x improved. The ease of building and deploying LAMMPS was reflected in our results. Builds did not require many debugging prompts (green in Figure 3) and the number of attempts was consistently under 5. The only application to consistently reach a build ceiling of N=10 attempts (Figure 7) was Laghos, which needed an increase to N=20. The ease of building mirrors the human experience.

**Laghos.** Laghos was an exemplar of a build that required collaboration. In our testing runs, the agent was unable to build a container in the maximum attempts we allowed (N=10). The problem is that Laghos is hard to build, even for a person. There are several components (hypre, mfem, and metis) and each requires specific build flags and compile options, and there is the added complexity of specific versions working together. While the LLM might figure out a correct configuration eventually, the maximum attempts setting exists to disallow an unreasonably large number of builds. We fixed this by providing the agent with human expertise – a description of the exact versions to build. With this expertise, the agent was successful every time. However, the components still took many attempts to get working together, reflected in a large number of prompts and responses for the build (Figure 3). We did not attempt a user provided function for Laghos due to the length of the build and previously poor performance with AMG.

**Kripke.** The Kripke LLM decision outperformed the user guided function (Figure 6), as a smaller grind time indicates higher performance. The FOM, Grind time per units of work (where smaller values are better) was order $10^{-8}$.

## 3.4 Multi-Node Experiments

A multi-node experiment requires extending the base container to include an OpenMPI build with libfabric enabled for EFA. We chose OpenMPI since it comes packages with the EFA installer, and find that it works well across clouds and environments [22]. We learned that asking for this additional dependency, especially for a multi-platform build, increases the time and difficulty of the build. Strategies for caching and reuse of images become increasingly important, and we adopted our strategy to ask the build agent to first build a base image with OpenMPI and libfabric, and then for the application builds we instructed it to use the shared base image.

**AMG2013.** The optimized FOM for AMG on 4 nodes was $6.132476 \times 10^9$ nnz/s on the *m7g.16xlarge*. We observed that after the deploy agent deemed the application working in Kubernetes, the optimize agent increased the resources of the run to occupy the full node. While the agent was not instructed to, it chose to test two nodes before increasing to full size (N=4). We observed negligible improvement in performance, and attributed it to a small problem size. The next attempt scaled up to the full four nodes at maximum resources, and achieved a FOM of $6.13 \times 10^9$. From this point, it increased problem size until performance degradation, which it attributed to “exceeded the available memory or hit other scaling limits.”

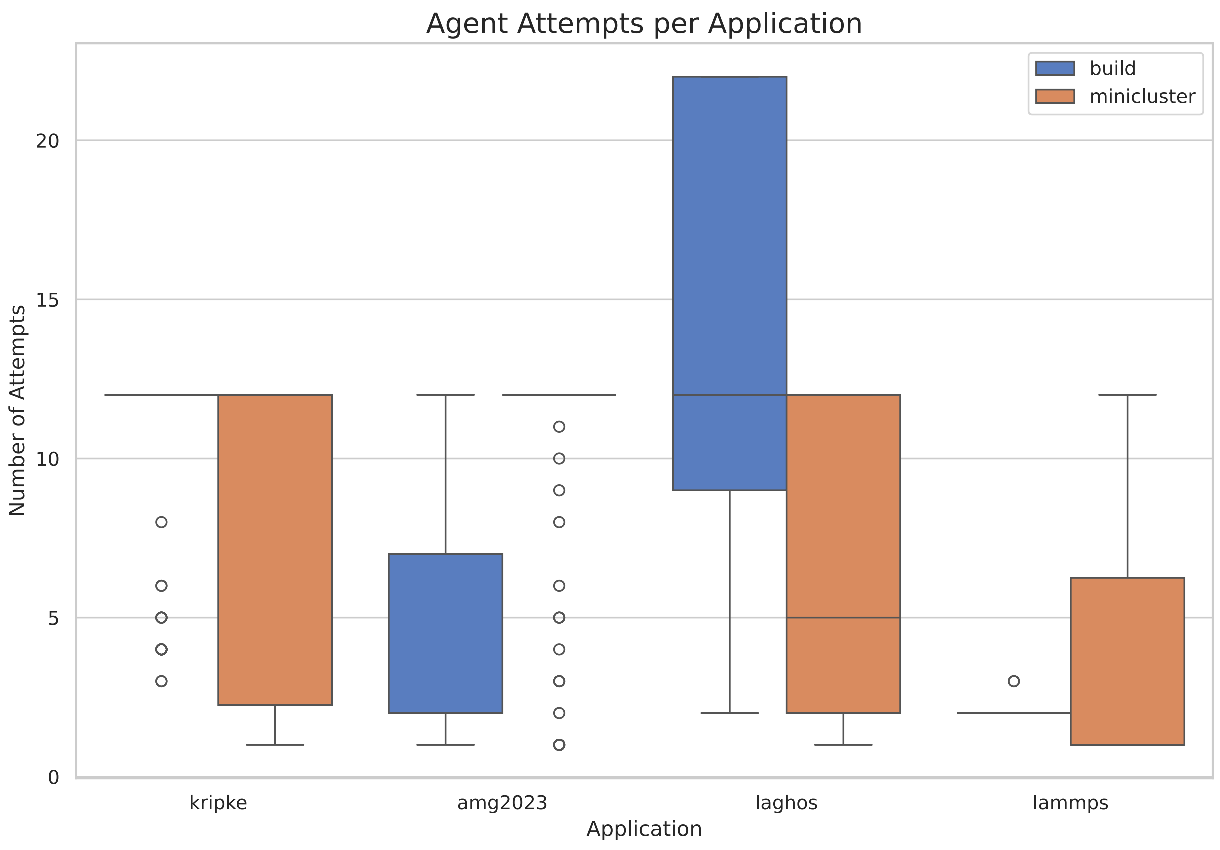

**Figure 7:** Agent Attempts. We allowed for a maximum of 10 attempts for all applications with the exception of Laghos. Laghos was an immensely challenging build that we allowed for a maximum attempts of 20. LAMMPS was an easy image to build.

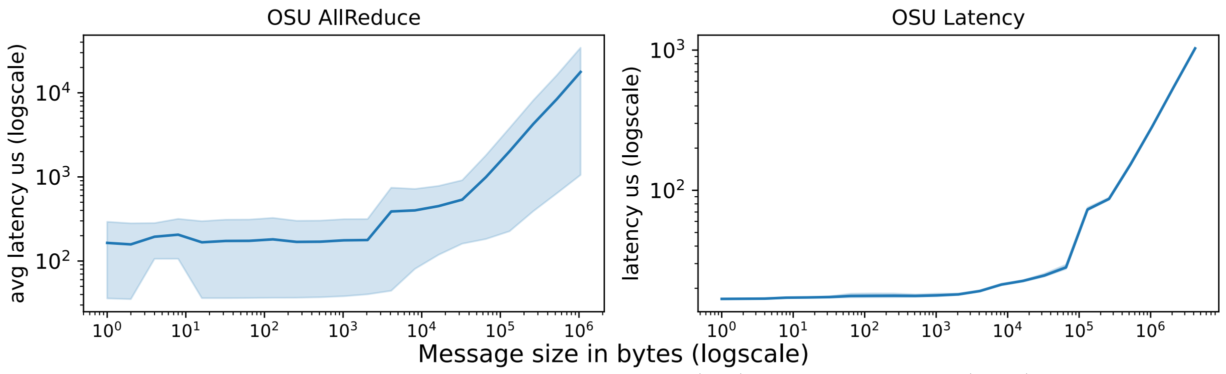

**Figure 8:** OSU Benchmarks OSU AllReduce (left) and OSU Latency (right).

**LAMMPS.** LAMMPS built, executed, and optimized on 4 nodes resulted in a maximum FOM of 2.50 Matom steps/s. During this optimization, the agent jumped immediately from 1 to the maximum of 4 nodes and 256 tasks, and increased problem size to *-v x 32 -v y 32 -v z 16*. This initial attempt was too optimistic and timed out, leading the agent to adjust to a smaller problem size (*-v x 24 -v y 12 -v z 12*) to establish a baseline (2.267 Matom/s). The final attempt increased the problem size to the deemed optimized FOM (*-v x 30 -v y 15 -v z 15 -in ./in.reaxff.hns -nocite*).

**OSU Benchmarks.** The OSU Benchmarks executed successfully without much prompt tuning. OSU Latency coincided with performance we have seen in previous performance studies using EFA [22] and OSU AllReduce showed high variability across message sizes (Figure 8). The agents successfully enabled collective and point to point communication using EFA.

**Kripke.** The Kripke Grind time optimized to run across 4 nodes was comparable to 1 node, with a slightly higher time to perform a unit of work ($8 \times 10^{-10}$ and $7 \times 10^{-10}$ for each of 4 and 1 nodes, respectively). The small decrease in performance likely results from the added need for network communication. Since it could be challenging in our experience [22] to get Kripke parameters correct, we requested the LLM to test on one node simply running *kripke* without parameters.

## 3.5 Scaling Study

We performed an agentic scaling study for LAMMPS, Kripke, and AMG. We had wanted to use a large cluster with 32 nodes, but were only able to provision 5 *hpc7g.16xlarge* nodes over the course of a week. The FOMs are shown in Figure 9.

We closely monitored each study for correctness. We learned that the agents need specific instructions about rules for each step. For example, the optimization agent must be given instructions to keep trying until a condition like a timeout, and whether to minimize or maximize FOM. The specific size and history of FOM progression at each size is helpful. In early testing, we observed that when we got application-specific prompts correct, the study ran smoothly, meaning observing subsequent attempts that were carefully changing the application in a way we thought reasonable without making what we deemed to be mistakes (e.g., stopping too early, changing a problem size incorrectly, or similar). Optimization agents were instructed to optimize parameters for size 1 runs, and to simply execute for larger sizes.

We carefully monitored each decision and step in the scaling experiments, and learned how to handle unexpected edge cases and how to improve usability. For example, we had a case where the job controller did not recreate a Pod that did not bootstrap correctly with Flux. The ideal outcome

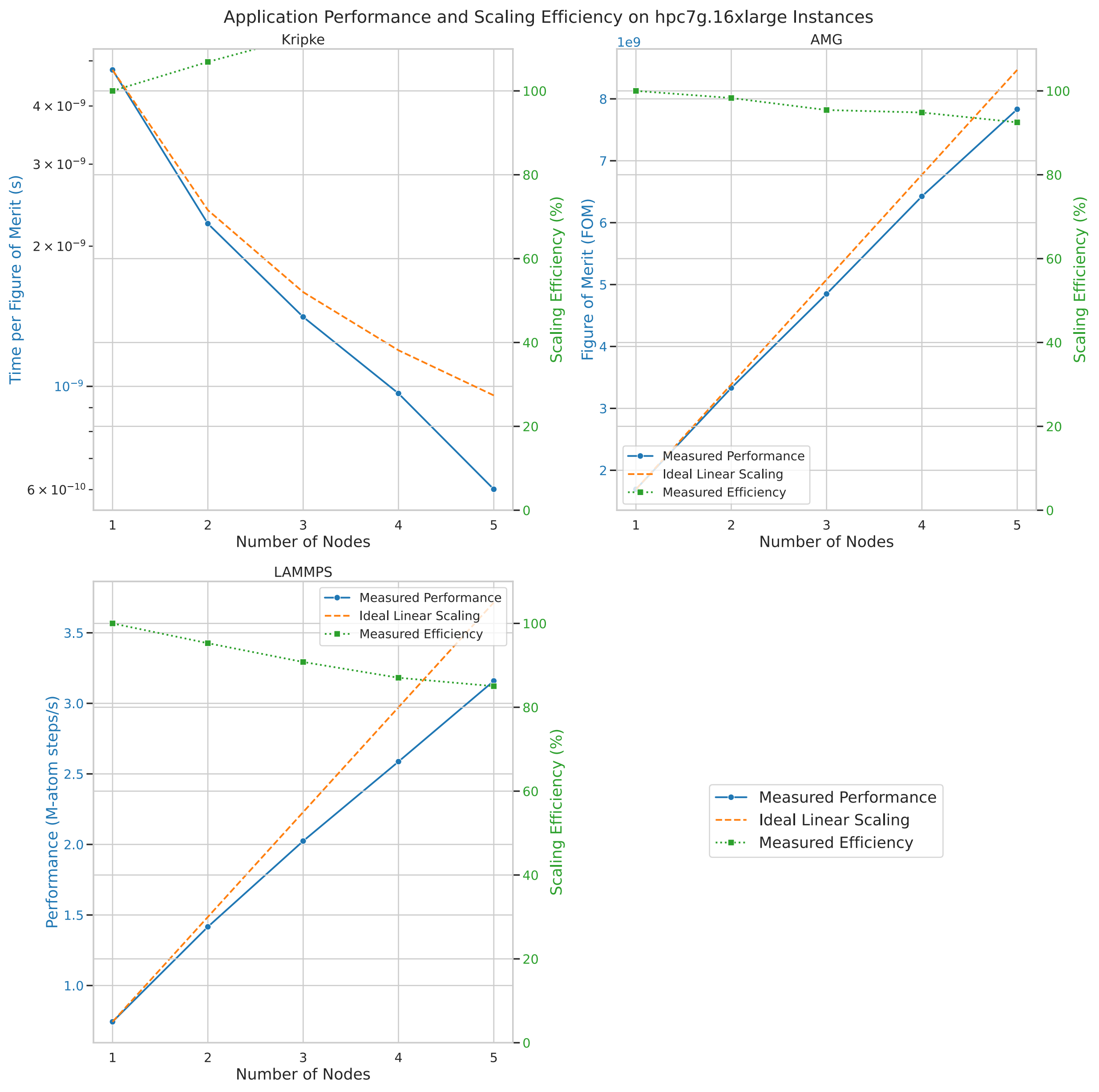

**Figure 9:** Application Performance and Scaling Efficiency on *hpc7g.16xlarge* Instances for each of Kripke, AMG, and LAMMPS. An initial successful deployment using a small problem size is done first, then an optimization on one node, and pinning of the decided problem size up to the maximum size. For all applications, higher is better, with the exception of Kripke, where lower is better. The decision to stop or proceed is done by a scaling agent.

would be to delete the Job, disregard the run, and try again. We added support for this case, a "Lost" status that can be triggered by unexpected deletion of the MiniCluster. We also learned that it was strategic to prompt the deploy agent to give unique names to Job with an incrementing number to avoid naming conflicts and to provide a live record via completed Pods for a human participant to inspect. Finally, we identified a tradeoff between choosing fewer edges in the graph and adding edges for error debugging. During the single-node optimization step, it would require fewer edges to give the optimization agent information about a failed run to debug. However, this practice violated our principle to have each agent perform one function, and in practice we found better error debugging by exiting the scaling and optimization loops and sending the diagnostic logs to the debugging agent to synthesize the problem and redeploy back into a working state.

**LAMMPS.** The LAMMPS optimization agent running on five nodes reached a maximum FOM of 3.159 Matom steps/s. We observed the scaling agent carefully increasing the problem size until the FOM stopped improving, even reporting on the percentage improvement between attempts at times. We observed that agent responses included correctly calculated percent changes in FOM output, suggesting that Gemini uses supporting tools (e.g., MCP) on its backend. The agent step stopped and returned the result when the execution timed out. Attempts for 2,3,4, and 5 nodes followed instructions to use the fixed problem size *18 x 18 x 18*. Comparing the single node optimization to the single node study, the FOM was slightly improved, hitting 743.061 katom-step/s, an improvement from 724.238 katom/steps using a problem size of *-v x 16 -v y 16 -v z 16*.

**AMG2013.** The scaling study produced a best FOM of $6.424653 \times 10^9$ nnz/s for AMG on 5 nodes. For optimization on one node, the agent immediately increased the problem size to *-n 270 270 270*, which resulted in the pods being *OOMKilled*. Subsequent attempts slowly decreased the size *(-n 270, 215, 150, 100)* until a successful run was found *(-n 80 80 80)*. The agent then reversed the pattern, increasing the problem size slowly until the "optimal" *(-n 90 90 90)*, where the agent stopped when the size *95* ran out of memory. While other applications typically had one run for sizes 2-5, the agent acted differently here, trying several variants of the processor topology for several sizes.

**Kripke.** Kripke's best FOM for the scaling study at 5 nodes was a Grind time of $6.012693 \times 10^{-10}$ (s/iter)/unknowns. The optimization agent started at a small number of cubic zones (with edge length 32, 48, 64, 80, 96, 112, 120, and 128), and methodically increased until the execution ran out of memory. The size 128 attempt failed, and a size of 120 determined to be the maximum for a single node. At sizes 2, 3, 4, and 5 nodes, the same problem size was returned in the response to run, and responses tested three different configurations of processor topology to maximize the FOM. We observed a super linear speedup that would be possible if the problem size per core became small enough to fit into the CPU's fast cache (96 MB L3 cache), reducing time waiting for data from the slower main memory. As an example, our problem running on 5 nodes of the *hpc7g.16xlarge* had a fixed total problem size of 103,555 MB. As we scaled up to 5 nodes, the data footprint per core decreased from 1,618 MB to 324 MB. While the full domain per core still exceeds the cache capacity, the Kripke wavefront of actively computed data is much smaller. By scaling to five nodes, this active wavefront becomes small enough to be cache-resident. We think that the transition to using the cache shifted the performance bottleneck from slow main memory to the much faster L3 cache, and that this shift resulting in an efficiency greater than 100%. We view this optimization and scaling effort as highly successful, as tuning happened not just for the problem size, but consistently for processor topology.

## 3.6 Result Assessment

For our result assessment, we went through each study reported in Table 2 and qualified the good and bad design choices. We were interested in Dockerfile and MiniCluster configuration logic correctness and design, and parameter or other environment choices. We include a summary of common good and bad choices for applications in Table 3.

**Table 3:** Evaluation of Practices in Experiments

| Choice | LAMMPS | Kripke | Laghos | AMG2013 |
|---|---|---|---|---|
| Non-interactive setup | 2 | 2 | 1 | 2 |
| `apt clean` usage | 2 | 1 | 1 | 0 |
| Allow root / pass envars | 1 | 1 | 1 | 1 |
| Shallow Git clone | 2 | 1 | 0 | 0 |
| Monolithic / multiple layers | 2 | 1 | 1 | 2 |
| Assumes specific network | 1 | 0 | 1 | 1 |

* Summary of common agent choices in experiments. Good practices (green) reflect practices that would be made by a human expert, as determined by our team of evaluators. Bad practices (red) indicate a less than ideal design.

**Application-specific Choices.**

**LAMMPS** Despite not being advised about networking, the LAMMPS build agent set several environment variables to assume ethernet as the network, which we considered a non-ideal choice in the case that the user deployed a different device. The multi-node build had similar patterns of build logic, and we generally had no negative criticism for the MiniCluster configuration files.

**OSU Benchmarks.** The OSU Benchmarks require careful choices with respect to number of tasks (e.g., point-to-point tests must be run on 2 nodes with exactly two processes) and the agents consistently produced the correct MiniCluster. Akin to LAMMPS, the Dockerfile had layers that included too many logical units, but best practices to run in non-interactive mode were used. The agents consistently chose Ubuntu 22.04 (Jammy Jellyfish). We think the choices of container bases (operating system versions) and good build practices reflect the training set of code the LLM was trained on. These practices are common if not a standard across open source communities.

**Kripke.** Unlike OSU and LAMMPS, the build of Kripke was done as an isolated logical unit to clone, compile, and cleanup, which we consider best practice. Environment variables to allow OpenMPI to run as root were included, which we think a sound step given that most containers provision root as the default user. The Kripke build resulted in behavior where no network parameters were hard coded in the environment, but rather were included as advice to the user as comments.

**Laghos.** The Laghos build was the most challenging of the set. The agent ultimately required exact versions of dependency components to use in the instructions, which we were surprised by as they are available in the main repository documentation. The Laghos build is the one build in our study that would not have been possible without very strict prompt input.

**AMG2013.** AMG2013 builds consistently built the *LLNL/AMG* (AMG2013) repository, which was surprising due to being older, and the license less permissive. The agent responses included good choices to add environment variables for running as root.

## 4 Discussion

In this work, we demonstrate the ability of an agentic team to semi-autonomously execute the entire workflow to build, deploy, and optimize HPC applications. There are several topics for discussion.

### 4.1 LLM Collaboration

An insight from this work is the importance of collaboration between LLM agents and humans for a balanced execution. The execution of each application is a multi-step process where an LLM is allowed to make baseline attempts, and the human assistant observes common patterns of error and behavior that can lead to erroneous attempts. In our experiments, we quickly learned that the LLM agents require not only succinct and clear prompting, but also detail about strategy to approach a problem. An instruction to optimize can be improved upon with added detail about a testing size, starting point for optimization, method to change parameters, and how to decide when to stop. The agents would not have reached successful outcomes without our insights, and often our insights were not possible without observing the LLM behavior.

We discovered there are many cases when the LLM makes an error that could easily be avoided with expert guidance, and the collaboration extends beyond prompting to interacting with the LLM during execution. While the LLM is capable of acting autonomously, a human can validate a pipeline along the way. In the case of generating a method to derive the result that can be cached or saved for subsequent operations (e.g., a regular expression) the need for human support is less frequent. For each decision, we ask the LLM to provide a reason for its choice to allow for an expert to see flaws or missing information in the logic of the response. This transparency guides prompt adjustment.

When we observed the LLM traversing into an impossible-to-fix state, for example a missing library discovered during deployment that required a different build, we could provide two strategies to the LLM to resolve it. A return to the manager would allow the manager agent to decide which agent to re-run, and adding a line to the prompt with instruction for custom setup commands would allow the agent to fix the error independently.

**Tight Feedback Loops** In the case of optimization, it could be the case that the agent response chose a problem size slightly too large that would lead to a timeout that was deemed a failure state. Allowing the agent response to request a return to the manager in this case added an additional sequence of prompts to debug, an extra deployment to re-assert the MiniCluster CRD still functioned, and then a retry. Instead, we opted to stay in the optimization loop, but inform the agent that the last attempt timed out due to likely a problem size too large or resources too small. We also implemented a context variable to prevent either return to the manager or the human. These settings would be appropriate for deployment cases when we observed a high rate of success, and that returns to the manager would likely be erroneous.

**Uncertainty** Unexpected behavior can be dealt with well by LLMs. An error that a user has never seen before can often be corrected by an agent. However, uncertainty coupled with expectations of resource availability can lead to erroneous behavior. For example, in testing the *hpc7g.16xlarge*, we encountered cases where the autoscaler could not deliver a node in 10 minutes of waiting time. This exceeded the agent's waiting time. The agents did not know a timeout could be do to insufficient resource capacity, and often did not respond correctly, changing problem sizes instead of instance type.

**MiniCluster Burstable QoS** We noticed that the MiniCluster and Job agent (not used for experiments but available) often chose to add both resource limits and requests to the pod, which would put it into Guaranteed mode. This is heavily documented, but not a common practice because it sets a hard limit using cgroups that makes the pod more likely to get *OOMKilled* and fail. Assuming that the LLM learns from documentation, we think there can be cases when configuration is so well documented that the LLM learns it and uses it as a best practice despite this not being the case.

**Resource Translation (vCPU)** Instance types vary based on the number of cores, and within types, often 2 vCPU correspond to 1 logical core. The exception is for the Graviton (ARM) instance type, for which the mapping is 1:1. However, even when we explicitly told the LLM about the definition of vCPU, it would often ask for double the resources, leading to a request that was unsatisfiable to the scheduler that saw logical cores. Although the next attempt would typically fix the issue, it was common enough that we decided to tell the LLM the exact number of logical cores and not vCPU for each instance type.

Practically speaking, this is avoiding an additional processing step that would require the knowledge to convert a vCPU value to CPU by dividing by 2. The lesson we learned is that the human should provide an exact description of all possible resources, and not rely on the LLM to guess or figure it out through trial and error. While the LLM can often arrive at the correct values, it adds additional attempts and cost.

**Hidden Dependencies** The build agent typically did not install certificates into the container, leading to an error about an insecure endpoint that would trigger another attempt that would fix the issue. This particular issue was so common that we added an instruction to the LLM to install the software, and the problem disappeared. While it is not possible to hard code every possible issue into an instruction, we think it to be a good practice to include common issues that show up frequently between applications such as this one. Retrieval-Augmented Generation (RAG) with more build context and errors specific to application types could be warranted here [27].

**Guardrails for Agents** Guardrails are constraints set on the problem size, parameter search space, or configuration options that set reasonable boundaries for the agentic search space. They are important so that the agents do not go too far in a direction that might ultimately lead to a failure. We noticed in early testing that agents would often go into guessing and failure loops when initial attempts to resolve a bug did not work. For example, in early testing we did not tell the agents about data or executable paths, and observed that they would retry ad infinitum using different paths and naming conventions. To resolve this issue, we instructed the build agent that it *MUST* add the application executable to the *PATH* and place a specific dataset in the present working directory of the container. We then directed the deploy agent to the exact data filename in the container present working directory. We think this is reasonable for a scientific user to do, as there is usually a specific task desired for completion. This paradigm also fits well in existing workflow tools that are designed to carefully track inputs and outputs. We generally found that commenting was a strong means of communication between agents. With shared context between agents, the text tokens provide the meat of a prompt, and thus comments from previous agents can be passed easily in this way.

**Application Requirements** When the deploy agent is first instructed to run the application, without any information about how the application needs or uses resources we noticed a tendency to choose a small number of resources that would require a long execution time. Only in cases when the running time exceeded our maximum time of 5 minutes would the feedback propagate and the agent decide to increase resources. We also learned in designing applications that our own user expertise is important. If we do not know how to properly build or run an application, we cannot instruct the LLM to do it either.

## 4.2 Insights

**Self-healing Systems** The choice of number of retries to allow an agent, and the strictness of guardrails provided must strike a balance that allows the agent to explore, but up to reasonable constraints. In any case, the entire system must be able to self-heal, or come back from a deeply erroneous state. While there are many strategies for self-healing systems [28], we chose a simple strategy to ensure several means to restart: a debugging agent can decide at any time to return to the manager, or the same outcome can result if the number of maximum attempts is exceeded. We also give the agent the option to return to a human, where a summary is presented to the human experimenter, and they can respond to the LLM with textual advice.

Another feature of a self-healing system is an ability to respond to agentic context drift [29] into more erroneous state. We noticed, for example, that the result parser might produce a regular expression that was close to correct, but did not match the log. Receiving feedback about the incorrect generation would then push the regular expression in an entirely different (and often wrong) direction. We decided to attenuate the behavior by allowing a human user to intervene with an instruction at the 5$^{th}$ attempt, and resetting history at the the 10$^{th}$ attempt. In the case of the example, the user could provide details that the first regular expression tried was the correct item to parse and very close, and to focus on it.

**Temporal Pinning and Dependency Versions** Care must be taken when advising the agents to use specific versions of software or dependencies. We discovered that asking for a specific version of a library tended to pull versions of other software and base images toward specific points in time. Specifically, asking for OpenMPI 4.1.2, the version that we used in our usability study, produced Docker builds with Ubuntu 20.04 and libfabric 1.13. While some cases of this behavior could be acceptable, in this case, the old libfabric did not function with the newer hardware on the instance. When not prompted to use a specific version, we tended to see more up to date versions (Ubuntu 22.04 and OpenMPI 5.x) that likely reflected a more recent bulk of textual data from training.

**Decision to STOP** Asking the LLM the algorithm it was using to determine when the application was optimized gave us insight into observed behavior. The LLM often reported a greedy hill-climbing approach, where it would evaluate the last choice, determine if the change improved or was detrimental to performance, and then would either step back to a previous configuration or continue in the same direction. The LLM seems to move toward a local optimum and stop when there is not obvious improvement and other resource tweaks to try. While the LLM would often choose a different instance type to start and change once or twice, it was not common for the LLM to want to try multiple instance types. We observed that the LLM agent always attempted to produce reasonable logic (comments in the configuration files) for making a choice, even if the logic was wrong. Many of the stopping conditions we saw as premature, and we think more human guidance about an approach would be needed. For example, we might instruct the agent to make a particular number of configuration changes before deciding to stop, or reaching a performance plateau.

**Optimization of Build Steps** Best practices for individual steps can often make the step more challenging for the agent. For example, while it is the case that a multi-stage build is a best practice for Docker images to reduce image size, in practice it made it more challenging for the build agent, because library dependencies or required data were less likely to be carried forward between stages. Allowing for the build step to build on two platforms, on the other hand, increased build time without adding additional error. We chose to build on two platforms to double the number of possible instances the deploy step could choose from.

**User-provided Functions** Our experiments demonstrated that the LLM agents outperformed our functions, likely due to not having rigidity and our lack of expertise of applications running on specific instance types. We observed that while the user provided function did not yield results for AMG, it outperformed the user guided function for LAMMPS. We think that a scoped, user provided function could be comparable to an LLM agent decision if it is written by an expert for the environment. This is work we anticipate doing.

**Emphasis in Prompts** We learned early on that without emphases added to our prompts, (e.g., *You MUST*) the LLM would often not follow the instruction. Adding capital letters made a large difference when we absolutely did not want a behavior. We also observed that when the LLM made a decision that went against one of our requests, it added code commentary that would justify the choice, even if not completely logical to us.

**Experiential Learning** The most dangerous failure cases were those that the agent could not learn from. For example, setting a small problem size with few too tasks would lead to timeout, and the agent would not get any feedback beyond the timeout condition. It would need to infer which of its choices led to the slow application execution, and try something differently. Only by trial and error and observation that threading made a huge difference in avoiding timeouts and increasing problem size that we learned to explicitly provide this advice to the LLM. Another fruitful strategy was the option to return to the manager. In this case, the manager was required to summarize the issue to return to an agent, and we found consistent issues in the summary messages that we were able to integrate into prompts. After this change, there were no subsequent returns to the manager due to the success of the deployment.

**Optimization Goals** Our most successful experiments were specific about optimization. Our first attempts combined the potential instance types provisioned by the *MiniCluster* with the optimization task message. Since the optimization task message is also provided to the result parser to extract a metric of interest, this mixing muddled the input prompt, and a small percentage of the time the parser would return more than the directed regular expression. We learned that we needed to separate the specific optimization task from the resources, and selectively provide them to agents.

Within the optimization task, it is important to define a specific goal. Minimizing walltime without specifying the problem size is insufficient because the LLM can simply run a small problem more quickly. Not providing a small test case can allow the LLM trying to run problems that are too large too quickly. Not providing a strategy or function for incrementing resources can lead to jumps that are too large or too small for the number of attempts allowed. A good strategy was to prompt the agents to focus on optimizing one variable, such as the FOM, and we chose the desired problem size.

**Learning from Agents** We often saw parameter or configuration choices that we were not familiar with. For example, in testing the MiniCluster agent set a CPU resource request with *7900m* for CPU. The *m* unit was not recognized by our team, however it is a unit called a millicore [30]. The LLM taught us a new concept. We also observe that agentic choices mirror common patterns of our development. For example, without specifically requesting a CPU setup, the build agent will generate a GPU-enabled Dockerfile – a much more difficult build. We think this reflects our current culture that is heavily invested in GPU workloads. Finally, what is appealing about an agentic approach is the ability to request build, deploy, optimization, and scaling for applications the user is not familiar with.

## 4.3 Future Work

**Function Execution** We observe that LLM agents are very good at function execution, whether that happens internally or via a regular expression provided to us. We would like to extend our result agent to also accept Python functions as results that can be visually validated and tested. A design of this type will work very well in an MCP setup. For example, an agent that requests a function and has test cases ready to validate it would be useful. A strategy to derive a regular expression or function to be validated and used for subsequent processing by the same helper agent is a lucrative strategy to provide both consistency of processing and cost savings in not needing to query to Gemini API. Doing validation in a way that does not require calling an API endpoint that incurs cost is an ideal strategy. It is a very compelling idea to be able to write software on the fly, only when you need it. We briefly tested using this strategy for our optimization agent, namely asking it to derive an optimization function, but found that the function would only return a single string to *STOP* or *RETRY*. We think this is an issue with our prompt, and that the LLM was confounding the need to write a function with the overall request to return a decision. The design of asking for a function will need to be better developed by our team.

**Scheduling and Asynchronous Execution** Future work should further consider agentic work in the context of scheduling [31]. For example, if agents form a graph, each step would need to be scheduled to ensure that completing steps have resources available when they are needed. If a linear execution pipeline is transformed into more of a workflow with components that can run in parallel, then agents can work asynchronously.

**Model Context Protocol** Having a more solid understanding of agentic systems and collaborating with agents, we plan to update our agentic software to use MCP, and instead of execution via subprocess commands, to design each agent task as a proper model context server. The validation implemented in our classes, and explicit direction of application flow is both a strength and a weakness. It is a strength in that we have implemented tighter control on what to do in different cases. It is a weakness in that it requires more hard-coded logic, and is less robust to cases that we have not seen. All decisions about workflow execution that we have currently instantiated as part of an agent class can be provided as functions to an LLM, and giving the LLM the decision to choose. As an example, we might implement a validation endpoint for a manifest, and give the LLM a choice about whether to validate or not. However, if a step is required, it should not be subject to choice in the first place. While there might be a tendency to give the LLM ability to make all decisions, it is not necessarily the case that allowing free reign to run all possible options will lead to the best outcome. Future work is needed to explore constraints and adjusting constraints based on the context. However, the benefits of using a MCP are many, adding standardization, authentication and authorization, and extended ability to provide tasks as services. We will want to maintain the abstraction of a graph of agents, and plug the standard into this design.

**Scaling and Optimization Logic** Our approach assumes a particular method of conducting a scaling study where application parameters are chosen based on optimization at the smallest size. We recognize that this choice of method does not reflect how every researcher conducts a scaling study, and would like to test other patterns. For example, an alternative to this approach is to optimize at the largest size, and then perform the entire study from smallest to largest.

### 4.4 Limitations

We recognize that other LLM API endpoints could have been used given equivalent opportunity. We recognize that more research is needed to better understand when and how a human should intervene.

## 5 Conclusion

Agentic frameworks are the future for not just the HPC community, but the entire software ecosystem. Autonomous, converged HPC infrastructure that can collaborate with humans to orchestrate the entire lifecycle of scientific workloads is becoming feasible. From autonomous task execution to failure recovery and AI-supported scheduling techniques, these powerful transitions will enable scientists and engineers to focus on asking more interesting questions and higher-level problem-solving. We look forward to a new era of faster, more accurate, and more accessible computing for the next generation of HPC.

### Acknowledgements

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 LLNL-TR-2013651-DRAFT.

## References

[1] DAIR.AI, "LLM agents." `https://www.promptingguide.ai/research/llm-agents`. Accessed: 2025-9-1.

[2] Y. Combinator, "Andrej karpathy: Software is changing (again)," June 2025.

[3] Google, "Agent development kit." `https://google.github.io/adk-docs`. Accessed: 2025-9-1.

[4] Microsoft, "Azure AI foundry." `https://azure.microsoft.com/en-us/products/ai-foundry`. Accessed: 2025-9-1.

[5] A. Analysis, "Comparison between total model parameters and parameters active during inference by model."

[6] Q. Wei, Z. Yao, Y. Cui, B. Wei, Z. Jin, and X. Xu, "Evaluation of chatgpt-generated medical responses: a systematic review and meta-analysis," *Journal of biomedical informatics*, vol. 151, p. 104620, 2024.

[7] M. Laun and F. Wolff, "Chatbots in education: hype or help? a meta-analysis," *Learning and Individual Differences*, vol. 119, p. 102646, 2025.

[8] B. Jr and B. Samuel, "Integrating artificial intelligence into science gateways," Tech. Rep. INL/CON-23-72098-Rev000, Idaho National Laboratory (INL), Idaho Falls, ID (United States), July 2023.

[9] W. Gan, Z. Ning, Z. Qi, and P. S. Yu, "Mixture of experts (MoE): A big data perspective," *arXiv [cs.LG]*, Jan. 2025.

[10] Anthropic, "Introducing the model context protocol." `https://www.anthropic.com/news/model-context-protocol`. Accessed: 2025-8-31.

[11] "API reference - OpenAI API." `https://platform.openai.com/docs/api-reference/introduction`. Accessed: 2025-8-31.

[12] Z. Asgar, "Efficient and scalable agentic AI with heterogeneous systems." `https://arxiv.org/html/2507.19635v1`. Accessed: 2025-8-31.

[13] D. H. Ahn, N. Bass, A. Chu, J. Garlick, M. Grondona, S. Herbein, H. I. Ingólfsson, J. Koning, T. Patki, T. R. Scogland, B. Springmeyer, and M. Taufer, "Flux: Overcoming scheduling challenges for exascale workflows," *Future Generation Computer Systems*, vol. 110, pp. 202–213, 2020.

[14] Q. Wang, H. Zhang, C. Qu, Y. Shen, X. Liu, and J. Li, "RLSchert: An HPC job scheduler using deep reinforcement learning and remaining time prediction," *Appl. Sci. (Basel)*, vol. 11, p. 9448, Oct. 2021.

[15] F. Mölder, K. P. Jablonski, B. Letcher, M. B. Hall, C. H. Tomkins-Tinch, V. Sochat, J. Forster, S. Lee, S. O. Twardziok, A. Kanitz, A. Wilm, M. Holtgrewe, S. Rahmann, S. Nahnsen, and J. Köster, "Sustainable data analysis with snakemake," *F1000Res.*, vol. 10, p. 33, Jan. 2021.

[16] V. Sochat, A. Culquicondor, A. Ojea, and D. Milroy, "The flux operator," *F1000Res.*, vol. 13, p. 203, Mar. 2024.

[17] P. Pranoy, Raghav Magazine, D. Chaitanya, T. Sho, and S. Vishal, "Adaptive LLM routing under budget constraints," *arXiv [cs.LG]*, Aug. 2025.

[18] M. Gridach, J. Nanavati, K. Z. E. Abidine, L. Mendes, and C. Mack, "Agentic ai for scientific discovery: A survey of progress, challenges, and future directions," *arXiv preprint arXiv:2503.08979*, 2025.

[19] A. Datar, "Transforming R&D with agentic AI: Introducing microsoft discovery." `https://azure.microsoft.com/en-us/blog/transforming-rd-with-agentic-ai-introducing-microsoft-discovery/`, May 2025. Accessed: 2025-8-31.

[20] D. Salomone and A. Prabhat, "Build KYC agentic workflows with google's ADK." `https://cloud.google.com/blog/products/ai-machine-learning/build-kyc-agentic-workflows-with-googles-adk`, June 2025. Accessed: 2025-8-31.

[21] Honeypot, "Kubernetes: The documentary [PART 1]," Jan. 2022.

[22] V. Sochat, D. Milroy, A. Sarkar, A. Marathe, and T. Patki, "Usability evaluation of cloud for hpc applications," in *Proceedings of the SC '25 Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC Workshops '25, (New York, NY, USA), p. 135–150, Association for Computing Machinery, 2025.

[23] R. e. a. Souza, "PROV-AGENT: Unified provenance for tracking AI agent interactions in agentic workflows." `https://arxiv.org/html/2508.02866v2`. Accessed: 2025-9-1.

[24] A. Weidman, "Regular expression denial of service - ReDoS." `https://owasp.org/www-community/attacks/Regular_expression_Denial_of_Service_-_ReDoS`. Accessed: 2025-9-1.

[25] Amazon Web Services, "The elastic fabric adapter (EFA)," 2022. Accessed: 2024-12-12.

[26] G. Inc., "Understand and count tokens." `https://ai.google.dev/gemini-api/docs/tokens`, 2025. Accessed: 2025-11-7.

[27] F. Cuconasu, G. Trappolini, F. Siciliano, S. Filice, C. Campagnano, Y. Maarek, N. Tonellotto, and F. Silvestri, “The power of noise: Redefining retrieval for rag systems,” in *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pp. 719–729, 2024.

[28] P. Nutalapati, “Self-healing cloud systems: Designing resilient and autonomous cloud services,” *International Journal of Science and Research*, vol. 11, no. 8, pp. 1173–1187, 2022.

[29] P. Martin, “Context drift: Why AI loses coherence over time and how to fix it.” `https://www.linkedin.com/pulse/context-drift-why-ai-loses-coherence-over-time-how-fix-martin--zlfuf/`. Accessed: 2025-9-10.

[30] K. Community, “Metric-server cpu and memory units - general discussions.” `https://discuss.kubernetes.io/t/metric-server-cpu-and-memory-units/7497`, Aug. 2019. Accessed: 2025-9-1.

[31] A. Doosthosseini, J. Decker, H. Nolte, and J. M. Kunkel, “Chat AI: A seamless slurm-native solution for HPC-based services,” *arXiv [cs.DC]*, June 2024.