PDF Download
3712285.3759887.pdf
29 January 2026
Total Citations: 2
Total Downloads: 2057

Published: 16 November 2025

Citation in BibTeX format

SC '25: The International Conference for High Performance Computing, Networking, Storage and Analysis
November 16 - 21, 2025
MO, St. Louis, USA

Conference Sponsors:
SIGHPC

RESEARCH-ARTICLE

STELLAR: Storage Tuning Engine Leveraging LLM Autonomous Reasoning for High Performance Parallel File Systems

CHRIS EGERSDOERFER, University of Delaware, Newark, DE, United States

PHILIP H CARNS, Argonne National Laboratory, Lemont, IL, United States

SHANE SNYDER, Argonne National Laboratory, Lemont, IL, United States

ROBERT BRIAN ROSS, Argonne National Laboratory, Lemont, IL, United States

DONG DAI, University of Delaware, Newark, DE, United States

Open Access Support provided by:

University of Delaware

Argonne National Laboratory

SC '25: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (November 2025)
https://doi.org/10.1145/3712285.3759887
ISBN: 9798400714665

STELLAR: Storage Tuning Engine Leveraging LLM Autonomous Reasoning for High Performance Parallel File Systems

Chris Egersdoerfer
University of Delaware
Newark, USA
cegersdo@udel.edu

Philip Carns
Argonne National Laboratory (ANL)
Lemont, USA
carns@mcs.anl.gov

Shane Snyder
Argonne National Laboratory (ANL)
Lemont, USA
ssnyder@mcs.anl.gov

Robert Ross
Argonne National Laboratory (ANL)
Lemont, USA
rross@mcs.anl.gov

Dong Dai
University of Delaware
Newark, USA
dai@udel.edu

Abstract

I/O performance is crucial to efficiency in data-intensive scientific computing; but tuning large-scale storage systems is complex, costly, and notoriously manpower-intensive, making it inaccessible for most domain scientists. To address this problem, we propose STELLAR, an autonomous tuner for high-performance parallel file systems. Our evaluations show that STELLAR almost always selects near-optimal configurations for the parallel file systems within the first *five attempts*, even for previously unseen applications. STELLAR's human-like efficiency is fundamentally different from existing autotuning methods, which often require hundreds of thousands of iterations to converge. STELLAR achieves this through autonomous end-to-end agentic tuning. Powered by large language models (LLMs), STELLAR is capable of (1) accurately extracting tunable parameters from software manuals, (2) analyzing I/O trace logs generated by applications, (3) selecting initial tuning strategies, (4) rerunning applications on real systems and collecting I/O performance feedback, (5) adjusting tuning strategies and repeating the tuning cycle, and (6) reflecting on and summarizing tuning experiences into reusable knowledge for future optimizations. STELLAR integrates retrieval-augmented generation (RAG), external tool execution, LLM-based reasoning, and a multiagent design to stabilize reasoning and combat hallucinations. We evaluate how each of these components impacts optimization outcomes, thus providing insight into the design of similar systems for other optimization problems. STELLAR's architecture and empirical validation open new avenues for tackling complex system optimization challenges, especially those characterized by vast search spaces and high exploration costs. Its highly efficient autonomous tuning will broaden access to I/O performance optimizations for domain scientists with minimal additional resource investment.

CCS Concepts

• **Computer systems organization → Parallel architectures**; • **Information systems → Storage architectures**; • **Computing methodologies → Artificial intelligence**.

SC '25, St Louis, MO, USA

ACM ISBN 979-8-4007-1466-5/25/11
https://doi.org/10.1145/3712285.3759887

Keywords

LLM Agent, Autonomous Tuning, Parallel File System

ACM Reference Format:
Chris Egersdoerfer, Philip Carns, Shane Snyder, Robert Ross, and Dong Dai. 2025. STELLAR: Storage Tuning Engine Leveraging LLM Autonomous Reasoning for High Performance Parallel File Systems. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '25), November 16–21, 2025, St Louis, MO, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3712285.3759887

1 Introduction

To deliver high performance, modern parallel file systems (PFSs) expose hundreds of configurable knobs to control I/O behaviors. For example, Lustre 2.12.5 has at least 159 tunable user parameters [53]. The newest Ceph Nautilus comes with 1,536 parameters, although some of them might not be tunable [40]. Determining the best I/O configurations to achieve optimized I/O performance for each individual application is a crucial task in high-performance computing (HPC) storage. Manual configuration is often impractical, however, because of the vast configuration space, the complexity of underlying hardware, and the diversity of scientific application workloads [13]. In practice, HPC system administrators will conduct benchmark runs during the installation phases and provide recommended configurations for the entire system [45]. This manpower-intensive process is costly and slow. More importantly, the resulting recommendations cannot capture the nuances of all workloads and will be suboptimal for some applications and I/O patterns.

Recently, automatic tuning methods that leverage heuristic rules [24, 35, 46], machine learning [4, 10, 40], and reinforcement learning [12, 34, 67] have shown great potential. For instance, ASCAR[35] proposed rule-based heuristics to respond to burst I/Os, SAPPHIRE [40] utilized Bayesian optimization to recommend the best configurations, and CAPES [34] leveraged reinforcement learning to tune parallel file system parameters. Although these methods are able to find configurations that outperform system defaults, they all incur significant tuning costs [10]. It has been repeatedly shown that the state-of-the-art autotuning frameworks require hundreds to thousands of iterations or training samples to explore the vast search space, which is formulated by the large number of tunable parameters and large number of choices for each parameter. Such high exploration costs make it impractical for existing autotuning frameworks to tune I/O performance for real-world applications in production environments.

In light of this situation, we draw inspiration from observing how human experts tune parallel file systems. When faced with a new application, HPC I/O experts typically begin by examining its I/O trace logs (e.g., Darshan logs) to understand the application's defining I/O patterns. Once the I/O patterns are understood, human experts rely on their knowledge to determine an initial tuning strategy. This knowledge was derived from the file system manuals, hardware specifications of the current cluster, and, most importantly the experts' prior experiences working with other applications on the same HPC platform. Using this knowledge, human experts can identify the most critical parameters to tune, the appropriate value ranges for each parameter, and the tuning direction. For example, for applications with large files shared across many processes, random I/O operations may benefit from larger stripe sizes and stripe counts [45].

After deciding on an initial optimized configuration, human experts typically run the application to validate their tuning strategy. In many cases, a full-scale run is conducted to accurately assess the application's performance in a production-like environment. Several outcomes are possible from this trial run. If performance improves in the expected direction, the experts may choose to stop tuning or test more aggressive configurations to seek further gains. If performance worsens, the experts will revisit and revise their previous decisions and try again. These failure cases are valuable learning opportunities, helping experts refine their understanding. Regardless of whether tuning is successful or not, human experts can summarize their experiences and distill them into "tuning knowledge" for that specific HPC platform. This accumulated knowledge becomes extremely valuable in reducing the number of tuning trials needed in the future.

While the approach that human experts take for tuning tasks differs greatly from existing autotuning methods, the outcomes are compelling. In practice, it is common for them to arrive at a *near-optimal configuration* within *a handful of attempts*. Such efficient tuning requires synthesis of expert knowledge of system parameters, application I/O workloads, hardware specifications, and empirical evaluations – a rare and manpower-intensive combination of skills. This is challenging for traditional autotuning frameworks to deliver but an ideal proving ground for agentic large language models (LLMs), which have the potential to autonomously employ diverse domain knowledge, conduct reasoning, and utilize external tools to address complex high-level tasks in a way that was not previously possible.

In this study we present STELLAR, a Storage Tuning Engine Leveraging LLM's Autonomous Reasoning, which reproduces exceptional tuning efficiency similar to that of human experts. STELLAR can be used by domain scientists to achieve *near-optimal* I/O performance for their applications within a *single-digit* number of attempts.

STELLAR consists of three main innovations to deliver this goal. First, STELLAR leverages retrieval-augmented generation (RAG) [32] to accurately integrate into the tuning process domain knowledge such as file system manuals and cluster hardware specifications. Second, STELLAR uses external tools to interact with the real world. Specifically, STELLAR uses an LLM to autonomously analyze I/O trace logs (e.g., Darshan logs [11]) based on its needs, conduct test runs, and gather real-world feedback by interfacing with the HPC cluster. Third, STELLAR organically integrates reasoning capabilities, in-context learning [57], and chain-of-thought prompting [64] of LLMs to generate configurations, reflect on past attempts, summarize their experiences, and distill valuable insights. As STELLAR is applied to more applications, it continuously accumulates new tuning *rule sets*, which can then be used to tune new applications with improved efficiency.

Our evaluations show that STELLAR significantly improves the I/O performance of various benchmarks and real applications, achieving up to 7.8x speedup compared with the default. The entire tuning typically finishes within five attempts, even without any accumulated tuning knowledge. The resulting I/O performance also is comparable to, or even surpasses, what human experts can achieve. More importantly, we demonstrate that the full version of STELLAR, augmented with tuning knowledge accumulated from prior tuning experiences on simple benchmarks, can consistently achieve near-optimal performance (compared with expert tuning) with less than five attempts when applied to new, previously unseen applications. These results highlight the promise of STELLAR, particularly its autonomous agentic LLM design, in tuning complex parallel file systems. The main contributions of this study are threefold:

- We propose the first LLM-based tuning engine for parallel file systems in HPC environments and show its effectiveness via extensive evaluations.
- We introduce a novel framework to combine general domain knowledge (from system manuals) with cluster-specific knowledge (from accumulated tuning rules) to efficiently tune I/O systems within a small number of iterations.
- We present an effective agentic LLM workflow that intelligently interacts with the system and distills tuning rules through iterative feedback.

The rest of the paper is organized as follows. In §2 we discuss relevant backgrounds of LLMs. §3 discusses the closely related work. In §4 we describe the architecture of STELLAR in detail. We present the extensive experimental results in §5. In §6 we present our conclusions and discuss future work.

2 Background

In this section, we provide the necessary background for understanding HPC storage tuning and its main challenges. We also briefly introduce LLMs, LLM agents, and the key challenges of using LLMs to address complex system optimization problems.

2.1 HPC Parallel File System Tuning

Modern parallel file systems often provide a large number of configurable parameters (or "levers") that enable customizing behavior to meet the needs of different applications and HPC platforms. For instance, Lustre, one of the most widely used PFSs, exposes more than 150 tunable parameters [53], while Ceph, another popular PFS, includes thousands of parameters [40]. The sheer number of parameters, combined with their wide range of possible values, makes identifying the optimal configuration extremely challenging.

2.1.1 Tunable Parameter Importance. Not all parameters have the same impact on I/O performance. Some parameters are set before the parallel file system is mounted, such as *mount_point* and *mount_block_size*, and are therefore not considered tunable at runtime. Other parameters control specific functionalities, and their values should be determined based on user needs rather than solely for improving I/O performance. For example, in Lustre, the parameters *llite_checksums* and *osc_checksums* enable or disable checksum mechanisms at the *llite* and *osc* layers, respectively [7]. While both significantly affect I/O performance, they should not be tuned just for performance gains; instead, their configuration should be guided by data integrity requirements. Additionally, some parameters are I/O related but have minimal impact on performance. For instance, Lustre's *lru_size* parameter controls the number of client-side locks in the LRU cached locks queue. While this may optimize performance, it primarily affects memory usage rather than directly impacting I/O performance [18].

What truly matters for tuning are the parameters that are both tunable at runtime and have a significant impact on I/O performance. For example, Lustre uses *stripe_size* and *stripe_count* to define file layout, which in turn dictates how I/O accesses will be distributed across available resources [18]. Parameters such as *max_rpc_in_flight* and *max_pages_per_rpc* control the concurrency and size of data transfers, directly influencing both latency and bandwidth [54]. It is critical to identify this category of high-impact, tunable parameters and focus on them, rather than attempting to tune all parameters in a brute-force manner. STELLAR leverages a RAG mechanism to accurately extract tunable parameters. More details are discussed in §4.

2.1.2 I/O Patterns and Profiling. Tuning a parallel file system should be performed on a per-application basis, since different applications often exhibit distinct I/O patterns and thus respond differently to the same parameter values. Most parallel file systems support such tuning. For instance, Lustre's *stripe_size* and *stripe_count* can be set for individual files. Its *llite.**, *osc.**, *mdc.**, and many other client-side parameters can be configured differently across compute nodes, affecting individual applications.

Understanding the internal I/O behavior of applications is hence critical for tuning. In HPC environments, multiple I/O profiling tools have been developed, such as STAT [3], mpiP [62], IOPin [25], Recorder [63], and Darshan [11]. Each of these tools provides insights into various aspects of application I/O behavior. In this study, we leveraged Darshan logs for two primary reasons. First, Darshan is lightweight and requires no application modification, making it easy to apply to a broad range of applications [44]. For the same reason, it is widely installed across many HPC facilities [45]. Second, several recent studies using LLMs to analyze I/O behavior [5, 16] have shown that Darshan logs can be efficiently chunked and distilled into high-level, actionable summaries. Specifically, Darshan traces key statistical metrics for each file across different types of I/O interfaces, including POSIX I/O, MPI-IO, and Standard I/O. These metrics include the amount of read/write data, aggregate time for read/write/meta operations, the ID of the rank issuing I/O requests, and the variance of I/O size and time among different application ranks. More details about how we process Darshan logs will be discussed in later sections.

2.2 Large Language Models

The rapid development of LLMs has demonstrated their powerful capabilities across a wide range of tasks. The latest models, such as GPT-4.5 [51] and Claude 3.7 [2], further showcase their strengths in handling complex and in-depth tasks, such as programming or conducting scientific surveys [50].

Recently, LLMs have entered a new era with the emergence of LLM agents, software components powered by LLMs that are capable of perceiving environments, reasoning about goals, and executing actions [39]. Unlike traditional chatbots, LLM agents can interact with real systems through continuous exploration, reasoning, and adaptation, as demonstrated by systems such as DeepResearch [50], OpenAI Operator [52], and Manus [43]. Various tools are available for developing LLM agent systems, including AutoGen [41], LangChain [29], and Dify [30]. Numerous LLM agents have been developed across scientific domains, such as Sci-Agents [19], Curie [26], ChemCrow [8], and AgentHospital [33] among others [39].

2.2.1 Workflows and Agents. LLM-based agentic systems can be categorized into two types based on their level of autonomy: *Workflows* and *Agents* [1]. *Workflows* are systems where LLMs and tools are orchestrated through predefined code paths, whereas *Agents* are systems in which LLMs dynamically direct their own processes and tool usage, maintaining control over how they accomplish tasks. Generally, workflows prioritize reliability at the cost of flexibility. They are well-suited for well-defined tasks where steps can be predetermined and the total number of execution paths is limited. In contrast, agents may compromise some reliability but offer the flexibility to address problems that developers did not anticipate from the outset. They are best applied to problems where future execution steps depend heavily on the outcomes of previous ones and where the enumeration of possible execution paths is large. It is therefore critical to make informed design choices when developing an LLM-based agentic system.

2.2.2 Tool Usage. Tool use is another critical component of LLM-based agents, enabling them to interact with their environment, for example by performing calculations, accessing real-time information, and generating code. Tool usage involves two key steps: deciding when to use a tool and choosing the right tool. The tool-use decision refers to determining whether a tool is needed to solve a given problem. When the agent is generating content with low confidence or encountering problems that require specific tool functionalities, it should decide to invoke the appropriate tool. Tool selection, on the other hand, involves understanding the available tools and assessing the agent's current context to choose the most suitable one. A successful LLM-based agentic system must have a robust infrastructure to support both of these steps.

2.2.3 Self-Learning. Self-learning is key to continuously improving LLM agents' capabilities. The typical methods are *self-reflection* and *self-correction*, both of which enable LLMs to iteratively refine their outputs by identifying and addressing errors. These methods leverage the self-reflective capabilities of modern LLMs, which emerged during training as the models were encouraged to check and reflect on their answers. It has been shown that promoting self-reflection instructions can significantly enhance an LLM agent's

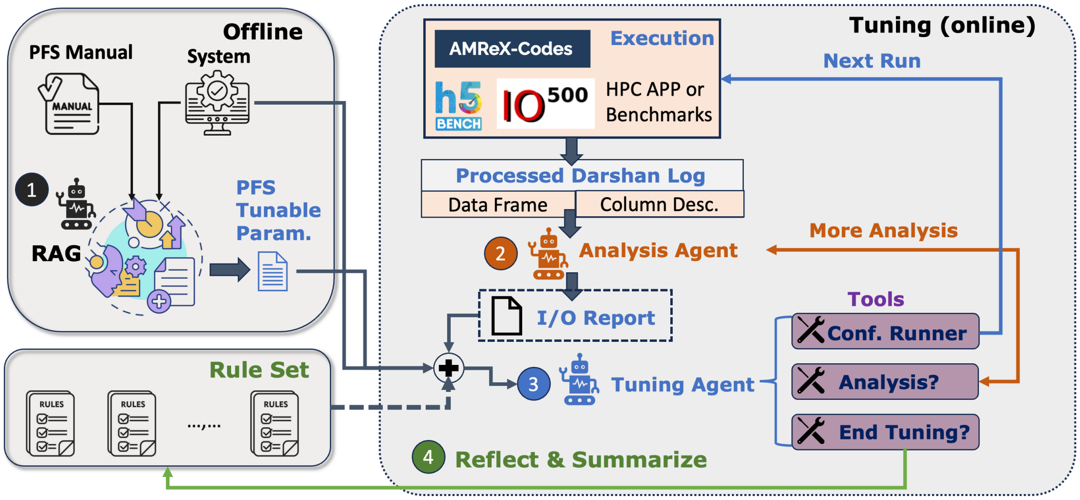


Figure 1: STELLAR design overview. The four numbered elements represent the four key modules in STELLAR.

performance [58]. When building a continuously learning LLM agent, it is therefore critical to effectively harness this property.

2.2.4 Hallucination Issues of LLM Agents. The knowledge of LLM agents remains limited in highly domain-specific areas, such as tuning the parameters of a particular parallel file system. Although their training data may include manuals for these systems and relevant discussions from online forums, the rarity of such data often means that LLMs are not sufficiently trained to capture accurate information. In such cases, when asked relevant questions, LLMs are prone to hallucinate knowledge. Worse yet, without a known ground truth, these hallucinations are difficult to detect, as LLMs often present them with authoritative language, potentially affecting downstream tasks in LLM workflows or agent systems. It is therefore critical to address the hallucination issue when implementing a practical tuner.

3 Related Work

We consider two threads of work that are most relevant with STELLAR: (1) the autotuning frameworks built for high-performance parallel file systems and (2) the autotuning frameworks that leveraged LLMs but are designed for other storage systems.

3.1 Autotuning for HPC Parallel File Systems

Autotuning HPC parallel file systems began from using heuristic-based approaches. For instance, ASCAR [35], TAPP-IO [46], DCA-IO [24], and IOPathTune [56] introduced rule-based strategies to tune parameters such as stripe size and stripe count to handle burst and imbalance I/Os. They share the same limitations in handling dynamically varying workloads, which motivate the machine learning-based autotuning that learns tuning strategies directly from the data samples. SAPPHIRE [40] utilizes Bayesian optimization to recommend the best configurations. Behzad et al. [4] implemented nonlinear regression models to model I/Os and employed genetic algorithms to explore the configuration space, which were similarly used by Rajesh et al. [55]. Cao et al. [10] further conducted a comprehensive comparative study across multiple autotuning strategies to assess their efficiency. CAPES [34], AIOC2 [12], and Magpie [67] leveraged deep reinforcement learning learning to adaptively learn tuning strategies. However, the high costs of iteratively sampling and testing workloads to explore the search space make these methods prohibitively expensive, limiting their usage at scale.

3.2 LLM-Based Database Tuning

Local storage systems, such as databases, face tuning complexities similar to those parallel file systems face. A variety of autotuning approaches, such as heuristic-based, machine learning-based, reinforcement learning-based, and even LLM-based, have been explored in the database context [14, 61, 65]. However, because of substantial differences between these two domains in terms of workloads (e.g., SQL queries vs. scientific applications), storage engines (e.g., SQL or key-value stores vs. parallel file systems), and hardware specifications (e.g., storage nodes vs. HPC clusters), it is not feasible to apply frameworks developed for one domain to the other without significant redesign. This distinction is also evident in the literature, where autotuning methods for SQL databases are typically not compared directly with those for parallel file systems. Following this established practice, we also do not quantitatively compare STELLAR with database-focused methods in our evaluations.

It is still insightful, however, to conceptually compare STELLAR with recent methods that also leverage LLMs for tuning database systems. Notable examples include DB-BERT [60], GPTuner [31], and E2ETune [23], which represent some of the most recent and representative efforts in this space. DB-BERT fine-tunes pretrained language models (i.e., BERT models) to translate natural language hints into configuration recommendations, followed by reinforcement learning to iteratively refine the initial selections. Similarly, GPTuner employs LLMs to extract tuning parameters and candidate values from manuals and online forums, then applies Bayesian optimization to converge on optimal configurations. E2ETune avoids the costly online iterative phase by shifting it to an offline data collection stage, where workloads are still repeatedly executed to gather samples using a Gaussian process. Notably, all these approaches successfully reduce the number of required iterations from thousands to fewer than 100. In the HPC context, however, the significantly higher cost of running parallel scientific applications makes even this reduced iteration count prohibitively expensive. In contrast, STELLAR adopts a fundamentally different approach by fully leveraging agentic LLMs to perform informed optimization, enabling it to converge within a single-digit number of attempts.

4 Design and Implementation

4.1 Overall Workflow

Figure 1 shows the overall design of STELLAR. The design consists of two main parts that together form the complete STELLAR engine. The *Offline* part (shown on the left) runs before any tuning occurs. In this part, STELLAR implements a RAG-based process to extract parameters and their definitions. The RAG system takes the parallel file system manual as input. It then outputs a filtered list of high-impact tunable parameters, along with accurate descriptions for each parameter. The *Tuning (Online)* part (shown on the right) conducts the actual iterative tuning process. STELLAR begins tuning after an initial run of the target application (e.g., E3SM, H5Bench). This initial run generates a Darshan log, which is further processed into a set of Pandas *DataFrames*, accompanied by a separate file describing the meaning of each column. The dataframes are sent to our first agent, the *Analysis Agent*, to extract application-specific runtime I/O behaviors. The *Analysis Agent* can autonomously write and execute analysis code to interpret processed Darshan logs based on specific requirements.

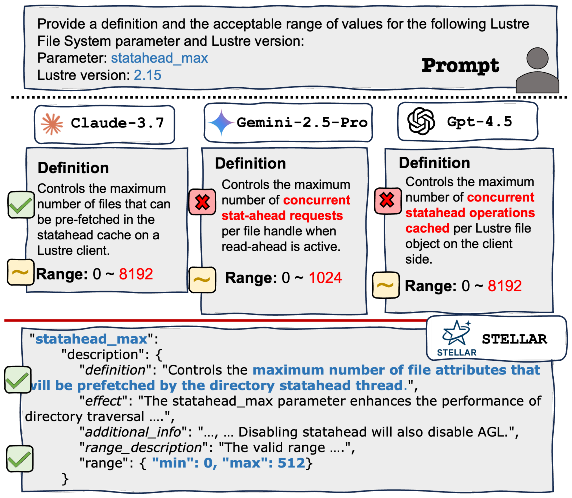
The I/O Report generated by the *Analysis Agent*, together with the PFS Tunable Parameters retrieved during the offline phase, is sent to the *Tuning Agent*, which drives the main trial-and-error loop. We have implemented three tools that the *Tuning Agent* calls to perform specific actions. The *Analysis Tool* is used to determine whether additional analysis is necessary. If so, it instructs the *Analysis Agent* to generate new code for the required analysis. The *Configuration Runner Tool* generates a new set of parameter values and drives the target application to run again with the updated parameter values. The *End Tuning Tool* determines whether the trial-and-error loop should stop, thereby concluding the tuning process.

Once tuning concludes, we use the *Reflect & Summarize* module to reflect on the entire trial-and-error loop and summarize rules derived from the experience. These new rules are merged with any existing rules to form a comprehensive global *Rule Set*. This global *Rule Set* will be utilized for tuning future applications as part of the input context provided to the *Tuning Agent*, as shown in Figure 1. A complete STELLAR run, leveraging a non-empty global *Rule Set*, ensures maximal tuning efficiency.

In the following subsections we introduce each individual component in greater detail.

4.2 RAG-Based Parameter Extraction

Successful and efficient parameter tuning for a parallel file system relies heavily on two key pieces of information: (1) a curated list of parameters significantly impacting I/O performance and (2) accurate descriptions of these tunable parameters, including how they affect I/O performance and their valid ranges. This detailed information helps the *Tuning Agent* generate correct parameter settings. In this subsection we discuss how STELLAR extracts these parameters.



Provide a definition and the acceptable range of values for the following Lustre File System parameter and Lustre version:
Parameter: statahead_max
Lustre version: 2.15

Definition
✓ Controls the maximum number of files that can be pre-fetched in the statahead cache on a Lustre client.
~ **Range:** 0 ~ 8192

Definition
✗ Controls the maximum number of **concurrent stat-ahead requests** per file handle when read-ahead is active.
~ **Range:** 0 ~ 1024

Definition
✗ Controls the maximum number of **concurrent statahead operations cached** per Lustre file object on the client side.
~ **Range:** 0 ~ 8192

```
"statahead_max":
        "description": {
            "definition": "Controls the maximum number of file attributes that
will be prefetched by the directory statahead thread.",
            "effect": "The statahead_max parameter enhances the performance of
directory traversal ....",
            "additional_info": "..., ... Disabling statahead will also disable AGL.",
            "range_description": "The valid range ....",
            "range": { "min": 0, "max": 512}
        }
```

Figure 2: Example of LLM hallucinations for storage system parameter details. We also show the RAG-based extraction result of STELLAR on the same parameter. Note that our RAG-based extraction leverages the older GPT-4o model.

4.2.1 *Limitations of LLMs in Extraction.*

Although LLMs have demonstrated remarkable abilities to answer various difficult questions, they still suffer from hallucinations — plausible but factually incorrect outputs. This issue is particularly problematic in our context because parallel file systems are domain-specific and lack extensive amounts of organized online documentation and highly popular discussion boards.

We illustrate such hallucinations in practice in Figure 2, where three state-of-the-art LLM models provide definitions and accepted ranges for the *statahead_max* parameter in Lustre version 2.15. Here, a check mark indicates a correct result, a cross an incorrect result, and a tilde an imprecise result. None of the three models provided entirely correct responses. All three were incorrect regarding the maximum accepted value for the parameter, and both OpenAI's GPT-4.5 and Google's Gemini-2.5-Pro provided flawed parameter definitions. We further show the output of STELLAR on the same parameter at the bottom of the figure. We can see that not only does it provide more detailed aspects about this parameter but it also generates correct information, which results from the RAG-based parameter extraction design.

Directly feeding entire manuals into language models expecting accurate parameter descriptions is also problematic due to the limitations of current LLMs. Manuals, such as the Lustre manual [53] with over 600 pages (>300k tokens), often exceed typical LLM context windows. Even models supporting extensive contexts encounter problems such as *lost-in-the-middle truncation*[37], *losing long-range dependencies*[6], and *context rot*[22], limiting their ability to extract and combine key pieces of relevant information scattered throughout large documents.

4.2.2 RAG-Based Extraction. STELLAR addresses the hallucination issue using a RAG-based workflow. Instead of directly feeding manuals into LLMs, we first build a *vector index* from the system manual, then implement a multistep filtering mechanism that leverages retrieved information to identify key parameters and generate accurate descriptions.

Generating the vector index. We create an embedding index by chunking and embedding the entire parallel file system manual using LlamaIndex [36], a popular open-source embedding and retrieval framework. We use the default chunk size of 1,024 tokens, 20-token overlap, and OpenAI's *text-embedding-3-large* model [49] to embed each chunk. The *vector index* stores the text chunks extracted from the parallel file system manual in a queryable database. Queries to the *vector index* then retrieve concise and highly relevant context for the LLMs to further analyze, thereby mitigating the risks associated with long contexts. Additionally, this index is easily updated when new manual versions become available.

RAG-based parameter definition. To facilitate runtime configuration, modern parallel file systems typically expose standard interfaces to access tunable parameters. These can serve as an initial source for parameter identification. For example, Lustre exposes parameters under */proc/fs/* and */sys/fs/*. Initially, a rough filter selects only writable parameters since these can be altered by STELLAR. We note that this step may not always be necessary because some storage systems directly expose tunable parameters via configuration files (e.g., DAOS [21]).

For each parameter, we query the *vector index* with the question *"How do I use the parameter [parameter name]?"*, retrieving the top K (e.g., 20) relevant chunks, leveraging the default LlamaIndex retriever. Using the retrieved chunks, an LLM (defaulting to OpenAI's GPT-4o [48]) is prompted to determine whether the documentation provides sufficient information to define the parameter's purpose and valid range. If the documentation is sufficient, the LLM is prompted to describe the parameter's purpose, its intended impact on I/O, and specify its valid range. An example output was shown in Figure 2. Parameters that are found to have insufficient documentation are filtered out based on the assumption that parameters that are not described in the documentation are likely to be of lesser importance than those that are.

Notably, in many cases a parameter may depend on other parameters, complicating this procedure. For instance, in Lustre the maximal value of *max_read_ahead_per_file_mb* is half of *max_read_ahead_mb*, whose maximal value is half of the system memory. For those parameters, deciding the range involves calculation and hardware specifics. In order to handle them, the LLM is instructed to use specific *dependent* and *expression* syntax rules that can be parsed and evaluated in the online tuning setting. These expressions will be calculated based on actual system values during tuning.

Selecting important parameters. After obtaining a filtered set of accurately described parameters with defined ranges, we select the most impactful ones since not all parameters equally influence I/O performance. To do so, we first exclude the binary parameters because their settings typically represent user trade-offs rather than tuning decisions. For example, the binary checksum parameters in Lustre significantly impact I/O performance [17, 59] but risk data integrity, a tradeoff best left to users.

We then prompt the LLM (e.g., GPT-4o) to decide, with documented reasoning, whether each remaining parameter is likely to have a significant impact on performance. This is feasible because the parallel file system manual typically describes how each parameter changes the I/O behavior from which potential to impact performance can be inferred. For example, Lustre's *max_rpcs_in_flight* parameter controls the maximum number of concurrent remote procedure calls (RPCs) between object storage clients (OSCs) and object storage targets (OSTs), clearly impacting I/O performance. In contrast, Lustre parameters such as *nrs_delay_min*, *nrs_delay_max*, and *nrs_delay_pct* simulate high server load scenarios, which is relevant but not directly connected to I/O performance. LLMs such as GPT-4o can make reliable selections by leveraging these detailed parameter descriptions.

The final output, *PFS Tunable Parameters*, is provided to the *Tuning Agent* as discussed next. The final set of parameters is likely to be much smaller than the complete set of parameters. For Lustre, STELLAR chooses a subset of 13 parameters to tune.

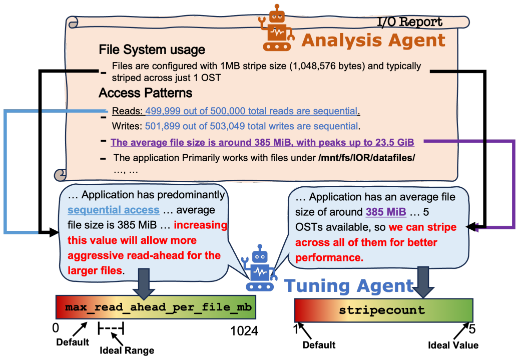
4.3 Agentic Online Tuning

The online tuning process in STELLAR is implemented as a fully autonomous agentic system involving the *Analysis Agent* and the *Tuning Agent*. There are essentially two loops during online tuning. The first is the main trial-and-error loop, which includes executing the target application, analyzing Darshan logs, generating a new set of configurations, and looping back to execute the application again with a different set of parameter values. The second is the minor loop, where the *Analysis Agent* generates an I/O Report, and the *Tuning Agent*, upon finding the report incomplete, requests additional analysis from the *Analysis Agent*.

4.3.1 Analysis Agent. The *Analysis Agent* serves two key purposes. First, it provides the context for the iterative tuning process by analyzing I/O behavior from Darshan logs collected at runtime and summarizing its findings. Its secondary role is conducting any additional specific analyses requested by the *Tuning Agent*. Both roles are enabled by designing the *Analysis Agent* as a code-executing LLM agent, leveraging the OpenInterpreter [47] framework for autonomous task completion. Given a task description, the agent plans, implements, and executes code until the task is considered complete by the agent.

The *Analysis Agent* operates on Pandas *DataFrames* along with string variables describing columns, rather than raw logs. Our preprocessing script extracts counters for each module (e.g., POSIX, MPI-IO) from Darshan and loads them into separate dataframes with corresponding counter descriptions. The log header is also loaded as a string variable. This process can be replicated for other tracing frameworks such as Recorder or less granular sources such as Elbencho [9], which reveal useful details about I/O behavior.

The *Analysis Agent* is tasked with providing a high-level summary of the application's I/O behavior by inspecting loaded variables (dataframes and descriptive strings), identifying files accessed by the application, and highlighting any information it deems useful for tuning the parameters. This high-level task description allows for dynamic analysis where the agent decides the most appropriate analysis based on application context. Although this approach may



File System usage

- Files are configured with 1MB stripe size (1,048,576 bytes) and typically striped across just 1 OST

Access Patterns

- Reads: 499,999 out of 500,000 total reads are sequential.
  Writes: 501,899 out of 503,049 total writes are sequential.
- The average file size is around 385 MiB, with peaks up to 23.5 GiB
- The application Primarily works with files under **/mnt/fs/IOR/datafiles/**
  …, …

… Application has predominantly sequential access … average file size is 385 MiB … **increasing this value will allow more aggressive read-ahead for the larger files.**

… Application has an average file size of around 385 MiB … 5 OSTs available, so **we can stripe across all of them for better performance.**

Figure 3: Example of decision-making via interactions between the *Analysis Agent* and *Tuning Agent*.

overlook certain details, the *Tuning Agent* can request further specific analysis during the iterative tuning process when necessary.

4.3.2 Tuning Agent. The *Tuning Agent* is the primary controller of the iterative tuning process. Its goal is to generate high-quality configurations, observe actual application performance, and reflect on the outcomes. At the beginning of the tuning process, the *Tuning Agent* receives the final filtered set of tunable parameters, details about the hardware and storage system setup, and the I/O Report generated by the *Analysis Agent*. The *Tuning Agent* then makes decisions using one of three potential environment interactions implemented as LLM tool calls.

If the *Tuning Agent* finds relevant information to be missing for the parameter it tries to tune, it selects the *Analysis?* tool and formulates a specific question (prompt) for the *Analysis Agent*.

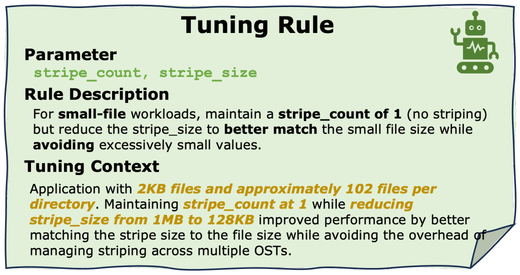
If confident, the *Tuning Agent* generates new configurations and executes the application to verify performance improvements. When generating new configurations, it explicitly documents the rationale behind each parameter value selection. This process encourages careful thought and facilitates validating LLM knowledge about parameter impacts by comparing stated reasoning against actual performance outcomes, which serves as the key to formulate *Tuning Rules*. To run the target application, STELLAR requires details on the initial execution process and interactions with the scheduling system. These should be provided by the domain scientists.

When the *Tuning Agent* selects the *End Tuning?* tool, it must provide reasoning for this decision. Specifically, in the system prompt we instruct the agent to finalize the process only when it believes that further tuning would not elicit further performance gains and to provide justifications. In practice, these guidelines cause the agent to explore more when significant performance improvement has not been found since it is less confident that improvement cannot be found elsewhere. The agent typically stops when diminishing returns are reached given the performance has noticeably improved beyond the default performance. The *End Tuning?* tool terminates the loop and initiates the *Reflect and Summarize* step. Here, the agent synthesizes rules learned during tuning, describing optimal parameter settings based on observed application I/O behavior. This step enables offline knowledge-base generation, as discussed further in the subsequent section.

4.3.3 Workflow vs. Agent? As previously discussed, the online tuning in STELLAR uses a system of two fully autonomous agents instead of more controllable *Workflows*, primarily because of the adaptive nature required at this stage. First, some applications with more complex behavior may be inherently less trivial to tune compared with others, requiring dynamic extension or shortening of the tuning process. Additionally, various applications have different I/O behavior that inherently requires a dynamic approach to summarizing their unique behaviors, one that is capable of dynamically adapting the focus of the summary to the most important aspects.

4.4 Rule Set Accumulation

Following the conclusion of every tuning procedure controlled by the *Tuning Agent*, the agent is asked to summarize, in the form of a rule set, what has been learned during the process. The first time STELLAR runs on a given system, the global *Rule Set* is empty. In subsequent runs, a new rule set will be added to the initial context and updated with new information learned from each run. Following this process, it continuously accumulates more rules as STELLAR is used to tune more applications.


Tuning Rule

Parameter

stripe_count, stripe_size

Rule Description

For **small-file** workloads, maintain a **stripe_count of 1** (no striping) but reduce the stripe_size to **better match** the small file size while **avoiding** excessively small values.

Tuning Context

Application with ***2KB files and approximately 102 files per directory***. Maintaining ***stripe_count at 1*** while ***reducing stripe_size from 1MB to 128KB*** improved performance by better matching the stripe size to the file size while avoiding the overhead of managing striping across multiple OSTs.

Figure 4: Example of generated tuning rule.

4.4.1 Rule Set Design. Each rule in the rule set generated by the agent refers to one or multiple parameters and contains both a definition for the rule itself and the I/O behavior context in which the rule applies. These are explicitly defined as the *parameter*, *rule description* and *tuning context* in the prompt which instructs the *Tuning Agent* to generate the rule set. Additionally, in the prompt we instruct the *Tuning Agent* to exclude the name of the application being tuned and to make general recommendations as opposed to specific ones. An example rule generated by the *Tuning Agent* is shown in Figure 4. As illustrated in the rule description, the recommendation for *stripe size* does not specify exact values to try, but instead offers guidance that the setting should be informed by the file size. The *tuning context* also clearly outlines the I/O characteristics of the workload where the rule was learned, making it easier to apply the rule to new workloads with similar characteristics. To avoid ambiguous rule sets potentially missing important information in each generated rule, we enforce a strict output structure when the LLM generates the rule set. Namely, the LLM must generate a JSON-structured rule set which is organized as a list of objects where each object contains a *Parameter*, *Rule Description*, and *Tuning Context* keys defined with their relevant values.

4.4.2 Rule Set Generation and Synthesis. After the rule set has been generated once, it can serve as a global *Rule Set* and be added to the initial prompt of any subsequent runs, and each subsequent run can serve to update the *Rule Set* to include any new information learned from that run. Concretely, runs following the initial one add the most recent global *Rule Set* to the initial prompt given to the *Tuning Agent*. Once the *Tuning Agent* ends the tuning process, it is asked to augment the existing set of rules rather than generate a completely new set. Since existing rules can conflict with new ones, the rule synthesis process can resolve these conflicts in two ways depending on the nature of the conflict. First, when a new rule directly contradicts an existing rule, the tuning agent is asked to remove it from the rule set. For example, if there exists a rule for the same parameter and equal tuning context as a new rule but the definitions suggest opposite guidance for how the parameter should be tuned, the *Tuning Agent* should remove both of these because it cannot be determined which is more correct. Alternatively, suppose the tuning context and intended parameter between two rules are equal and the rules offer only slightly different guidance. In that case, they should be noted as alternative approaches so that future tuning attempts could potentially try both. If both approaches are attempted in a future tuning run but only one of them produces a positive outcome, the tuning agent drops the negative approach when updating the rule set during this run, the same JSON structure as mentioned in the previous subsection is reused when merging rule sets to maintain a common format.

5 Evaluation

5.1 Evaluation Setup

To demonstrate STELLAR's performance and validate the design decisions outlined earlier, we conducted comprehensive evaluations and present the results in this section. Through these evaluations, we aim to answer the following key questions:

- Within a limited number of attempts, can STELLAR successfully tune the parallel file system for improved I/O performance? If so, how does its performance compare with that of human experts?
- Can STELLAR effectively accumulate useful knowledge (i.e., a global *Rule Set*) and leverage it to continuously enhance its tuning efficiency?
- How do individual design choices, such as the RAG-based parameter extraction and the *Analysis Agent*, impact the performance?

We do not include comparison results with traditional machine learning-based autotuners because they all take too long (hundreds of iterations, taking hours or days) to converge to similar results to those STELLAR achieves within five attempts. Also, since they cannot tune 13 parameters at the same time, a side-by-side comparison would be unfair. Instead, We compare with human experts' suggestions when the final I/O performance matters.

In the following sections we refer to the entire tuning process of STELLAR as a *Tuning Run*, which starts from the application's initial execution to the end of the tuning. Between *Tuning Runs* we always perform the following steps to ensure the results are not contaminated: (1) delete all data files and directories, (2) clear all client-side caches, (3) remount the entire file system on all client nodes, and (4) wait until all queued Lustre sync changes are completed. To ensure that our results are not impacted by noise, we ran each cases eight times to get averages. We show the 90% confidence interval when needed.

5.1.1 Evaluation Platform. All evaluations were conducted on the CloudLab platform [15]. CloudLab is an open platform that allows others to easily replicate our experiments. Due to the nature of our work, we were unable to conduct evaluations on large-scale production HPC systems because many of the tunable parameters require root privileges to modify. Specifically, we allocated 10 CloudLab machines to build a cluster for the evaluations. Each machine is equipped with an Intel Xeon Silver 4114 processor with 10 physical CPU cores and approximately 196 GB of memory and is connected via a 10 Gbps network switch. We installed Lustre 2.15.5, configured with five object storage servers and a combined management server (MGS) and metadata server (MDS) [7]. The remaining five machines were used as client nodes to run benchmarks and real applications.

5.1.2 Benchmarks. We selected three classic HPC I/O benchmarks - IOR [38], MDWorkbench [28], and IO500 [27] - to systematically evaluate how STELLAR handles I/O-intensive, metadata-intensive, and mixed workloads. All benchmarks ran in parallel using 50 MPI processes across five client nodes.

We created two workloads using IOR. The first, labeled as *IOR_64K*, has each MPI process concurrently write/read a 128 MB block using 64 KB transfer size. The I/Os were conducted randomly to a shared file. The second, labeled as *IOR_16M*, has each MPI process write/read three 128 MB blocks using a large transfer size of 16 MB with a sequential access pattern to a shared file. These two workloads represent two representative I/O patterns: random small and sequential large writes.

We created another two workloads using MDWorkbench. The first, labeled *MDWorkbench_2K*, creates 10 directories per process and fills each directory with 400 files, each sized 2 KB. The second, labeled *MDWorkbench_8K*, also creates 10 directories per process and fills each with 400 files, but each file is 8 KB. Both MDWorkbench workloads ran for three rounds, where each round conducted `open`, `write`, `close`, `stat`, `open`, `read`, `close`, and `unlink` operations on each file.

The final workload is based on *IO500* [27], which combines IOR and MDTest workloads into one application running through multiple phases including sequential read/write with large access sizes (IOR-Easy), random read/write with small access sizes (IOR-Hard), and metadata-intensive workloads for empty (MDTest-Easy) and small files (MDTest-Hard). This workload challenges autotuners to find the best configurations for the combined workloads.

5.1.3 Real Applications. In addition to benchmarks, which were picked for better demonstration, we used a set of real applications to evaluate STELLAR. Specifically, to be representative, we selected one scientific application I/O kernel and one I/O proxy application. The scientific application I/O kernel uses the AMReX framework [66], which implements highly concurrent, block-structured adaptive mesh refinement. The I/O proxy application originates from MACSio [42], which is designed to model I/O workloads from multiphysics applications primarily, with highly variable data object distribution and composition. Since MACSio's object size and can be configured to take on various sizes, we evaluate one configuration using an object size of 512KB (labeled as *MACSio_512K*)

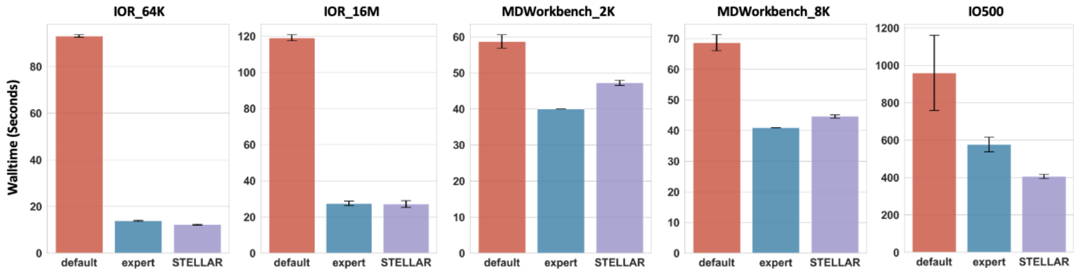


Figure 5: Comparison of STELLAR’s tuning performance with *default* and human *expert* baselines. Smaller values are better.

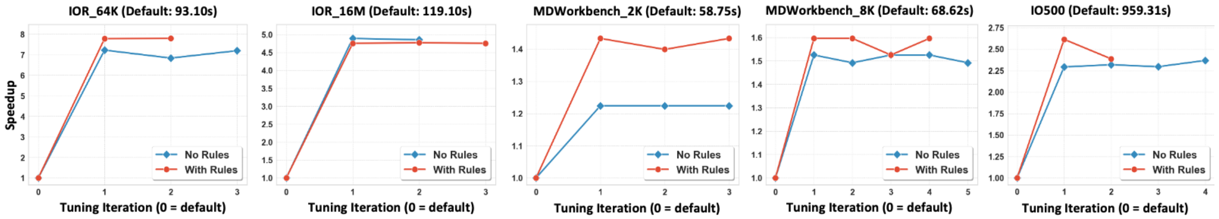


Figure 6: Speedup compared with default Lustre settings with and without the global *Rule Set*. Larger values are better.

and another configuration using an object size of 16MB (labeled as *MACSio_16MB*). Similar to the benchmark applications described previously, our evaluations of these applications were conducted using 50 MPI processes spread across 5 client machines.

5.2 STELLAR Tuning Performance

Our first evaluation answers whether STELLAR can successfully tune a PFS for improved I/O performance within a limited number of attempts.

In these evaluations, STELLAR starts from fresh, without any previous knowledge about these workloads nor any global *Rule Set*. The results are reported in Figure 5. Here, *default* refers to the configuration of default parameter settings in Lustre. The *expert* refers to the configuration provided by an I/O expert. To help the expert make better tuning decisions, in this case, we provided the full information about the benchmarks, including a full description of the benchmark settings and the full Darshan trace logs. The expert was also given practically unbounded time to generate the suggested set of parameters. The STELLAR results represent the best configuration generated from the tuning run. Note that we limited STELLAR to try at most five configurations and forced it to stop if not automatically stopped by that limit. Each bar represents the wall time (in seconds) of the workload; hence smaller values are better. We show the average and include a 90% confidence interval for each average to show the expected performance variance for each configuration. As indicated by these results, STELLAR was able to generate very high-performance parameter configurations that perform much better than the default and perform similarly to or even better than human-level performance. Notably, in the case of IO500, STELLAR was able to outperform the human expert baseline, which proves its ability to adapt to workloads that have multiple phases and variable I/O patterns.

5.3 STELLAR Tuning with Global Rule Set

In our second evaluation we answer two questions: (1) Can STELLAR successfully and consistently interpolate tuning rules as it tunes different applications? (2) Can the accumulated tuning rules be extrapolated to new and unseen applications?

5.3.1 Rule Set Interpolation. In the first scenario we used STELLAR to tune all the benchmark applications one by one without any rule set first. Since each tuning run will create some tuning rules, STELLAR needs to merge them into the global *Rule Set*. It is critical that such merging operations do not generate wrong or inconsistent rules. To verify that, we again tuned all the benchmark applications, but this time with the global *Rule Set* applied.

The results of this evaluation are presented in Figure 6, where each plot corresponds to a single benchmark application. Each plot’s x-axis indicates the number of tuning iterations that STELLAR conducted (i.e., number of configurations tried) until the *Tuning Agent* decided to end the process. Note that iteration 0 indicates the initial run without any tuning. The y-axis indicates the speedup compared with the average performance of eight runs using the default parameter settings. The results strongly support the fact that STELLAR successfully interpolates knowledge from the tuning rules. Four of the five cases show that leveraging the generated global *Rule Set* resulted in a significantly improved first guess at the optimal parameter settings. In fact, in the case of *MDWorkbench_2K*, where our previous results (i.e., Figure 5) showed the human expert-generated configuration to be slightly better, adding the rule set helped STELLAR fill this gap and generate configurations that perform equally well as the expert-generated configuration. Further, because of the high performance of the initial guess, STELLAR required less additional exploration of the parameter settings, leading to faster conclusion of the tuning process in 3 out of 5 cases and equal amounts in 1 out of 5. Again, these results further confirmed that STELLAR tunes the PFS within five attempts.

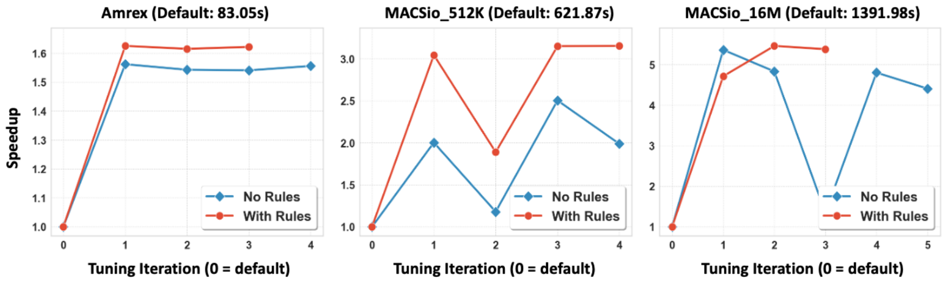


Figure 7: Speedup compared with default Lustre settings with and without a global *Rule Set* for real applications. For all the results, larger values are better.

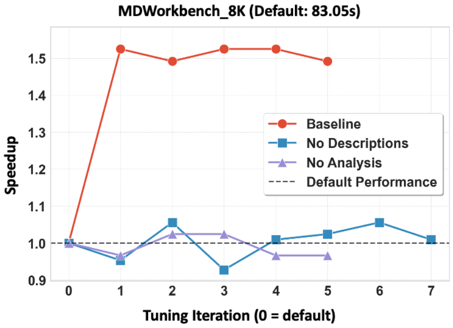


Figure 8: Tuning performance on MDWorkbench with removed STELLAR components.

5.3.2 *Rule Set Extrapolation.* In the second scenario we first used STELLAR to tune each of the real application workloads without a rule set and then tune these applications again using the rule set generated accumulated from only tuning the benchmark workloads.

In contrast to the previous scenario, this scenario evaluates STELLAR's ability to extrapolate knowledge from the learned rule set to previously unseen workloads. The results are shown in Figure 7, where each plot corresponds to a single real application. We can easily observe, even in such a more difficult scenario, that the positive impacts of the global *Rule Set* hold. This is evidenced by the fact that in all cases, the rule set enabled more stable convergence and higher performance of each generated configuration on average. Notably, for the *MACSio_16M* application, the rule set helped avoid exploration of configurations that had similar performance to the default settings, which were explored without the rule set. Similarly, for *MACSio_512K*, the addition of the rule set helped avoid the worst settings that were explored without the rule set.

5.4 STELLAR Ablation Tests

This evaluation answers the third question: how each component of STELLAR contributes. We conducted two ablation tests to validate STELLAR's primary components. Because of space limitations, we show only the results conducted on the *MDWorkbench_8K* benchmark workload since previous results showed that STELLAR required more iterations for this workload, indicating it is more difficult to tune.

In our first ablation test we evaluated the impact of the RAG-based Parameter Extractor. Specifically, we removed the parameter descriptions generated via the RAG process and observed the tune process of STELLAR. The results are shown in Figure 8, labeled as *No Descriptions*. Note that for this test we maintained the valid value ranges for each parameter, which were also generated via our RAG process. This was required because missing these value ranges causes the tuning to fail in most cases when the *Tuning Agent* would often attempt to set invalid values. From the results, we can observe a significant drop in performance when parameter descriptions are missing. Upon further analysis of the tuning agent's decisions during this run, we concluded that the primary reason for this drop is the *Tuning Agent*'s lack of accurate understanding of the parameters. For example, when generating *stripe settings*, it understands that a stripe count of 1 is more efficient for small files but then states that changing the parent directory's stripe count to -1 could "distribute the files more evenly across all OSTs." This is a flawed interpretation of how stripe count affects the files in a directory. This misunderstanding can be avoided by our RAG-based extractor, which defines the *stripe count* parameter clearly as "the number of Object Storage Targets (OSTs) across which *a file* will be striped". Avoiding such misinterpretations is critical to avoiding misguided tuning decisions and failing to converge to better parameter settings.

Our second ablation test evaluates the impact of removing the *Analysis Agent* from the tuning process. As described in preceding sections, the *Analysis Agent* is a key component in the tuning process as it is tasked with conducting accurate analysis of applications' I/O behavior, which helps guide the *Tuning Agent* toward reasonable initial predictions. Removing the *Analysis Agent* removes both the initial I/O report and the ability to answer additional clarifications from the *Tuning Agent*. The result of removing these features is also catastrophic, as presented in Figure 8, labeled *No Analysis*. The performance is similar to the previous as the *Tuning Agent* fails to generate configurations that significantly outperform the default settings. Additionally, the reason for this level of degraded performance is clearly indicated by the *Tuning Agent*'s decisions, since in this case the agent attempts to increase readahead and RPC size-related parameters. Detailed information on the application's I/O behavior, especially its predominant access to many very small files, would have ruled out such misguided parameter settings.

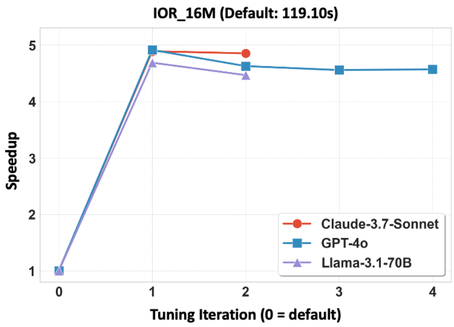


Figure 9: Tuning performance on *IOR_16M* with different LLMs as the *Tuning Agent*.

5.5 Model Comparison

While each of the preceding evaluations was conducted with Claude-3.7-Sonnet acting as the *Tuning Agent*, STELLAR's performance is not determined by this choice. In fact, any tool-calling LLM should be able to act as STELLAR's tuning agent. To showcase this, we conducted tuning runs with up to five iterations each for the *IOR_large* benchmark application with two different LLMs acting as the *Tuning Agent*: OpenAI's GPT-4o and Meta's significantly smaller open source Llama-3.1-70B-Instruct model [20]. The results are reported in Figure 9 and clearly highlight the fact that all three models are able to generate configurations that perform similarly and achieve significant speedups (up to x4.91) compared with the default configuration.

5.6 Scalability Limitation and Future Directions

As discussed, all our experiments were conducted on CloudLab due to root-access requirements for system-level parameter tuning. While this constraint limited our evaluation scale, we discuss here the implications for production HPC systems and our path toward broader deployment in the future.

We accept that large-scale systems expand the configuration space and may introduce scale-dependent phenomena. For instance, systems with thousands of nodes enable broader parallelism configurations (e.g., Lustre stripe count and stripe size settings can span wider ranges), while heterogeneous hardware may shift optimal parameter values (e.g., the presence of burst buffers, NVMe-based storage tiers, or varying network topologies can influence the effectiveness of specific parameter combinations). However, we argue that STELLAR's fundamental approach remains scale-invariant: it automates the same iterative process experts use, regardless of system size, by analyzing the execution, adjusting parameters, and refining based on observations. In fact, larger systems may even facilitate automated tuning by exhibiting more pronounced performance responses to parameter changes, helping STELLAR identify causal relationships more clearly.

Recognizing root-access constraints in production environments, our future work will target user-accessible tuning opportunities: (1) application-layer parameters including HDF5 settings and MPI-IO hints, (2) user-space storage systems like DAOS that provide extensive tunability without privileges, and (3) hybrid approaches where STELLAR recommends both user-controllable and system-level parameters.

5.7 Cost and Latency Analysis

Latency overhead. LLM inference introduces only a few seconds of latency per tuning decision across all evaluated models (GPT-4o via OpenAI API, Claude-3-Sonnet via Anthropic API, and Llama-3.1-70B via TogetherAI API). This overhead is negligible compared to application runtime, which typically ranges from minutes to hours for HPC workloads. Thus, the end-to-end tuning time is dominated by application execution rather than LLM processing.

Token usage and costs. Each of STELLAR's tuning runs depends on the LLM models behind both the *Tuning Agent* and *Analysis Agent*. For reference, when using Claude-3.7-Sonnet as the *Tuning Agent*, a single complete tuning run processes ~100k input tokens and generates ~13k output tokens on average. Using GPT-4o as the *Analysis Agent*, an equivalent run processes ~400k input tokens and generates ~8k output tokens on average. The total cost of these token generations is highly dynamic and continues to decrease as more performant models become available at lower inference costs. Notably, the nature of the *Tuning Agent's* iterative tuning process, as well as the *Analysis Agent's* analysis process, allows for the majority of key-value matrices calculated for input tokens in each inference request to be cached and reused. In fact, when prompt caching is enabled, between 85 and 90 percent of the total input tokens are resolved via cache over the course of a tuning run. This significantly reduces API costs when using proprietary inference services and reduces computation costs when using local inference platforms.

6 Conclusion and Future Plan

In this study, we proposed STELLAR, an autonomous tuner for high-performance parallel file systems. By leveraging agentic LLMs, STELLAR consistently selects near-optimal configurations within the first five attempts, demonstrating a level of efficiency that closely mirrors human expertise. This human-like capability sets STELLAR apart from traditional autotuning methods, which often require hundreds or even thousands of iterations to converge. We believe STELLAR opens a promising new direction for autonomously optimizing complex HPC infrastructure. Beyond its research value, its practical implementation can democratize I/O performance tuning for domain scientists, ultimately accelerating scientific discovery. In the future, we plan to extend STELLAR's evaluation to larger-scale systems, potentially in production systems, with the focus on user-accessible tuning parameters.

Acknowledgments

We sincerely thank the anonymous reviewers for their valuable feedback. This work was supported in part by the National Science Foundation (NSF) under grants CNS-2008265 and CCF-2412345. This effort was also supported in part by the U.S. Department of Energy (DOE), Office of Science, Office of Advanced Scientific Computing Research (ASCR) under contract number AC02-06CH11357 with Argonne National Laboratory.

References

[1] Anthropic. 2024. Building effective agents. https://www.anthropic.com/engineering/building-effective-agents Accessed: 2025-03-20.

[2] Anthropic. 2025. Claude 3.7 Sonnet and Claude Code. https://www.anthropic.com/news/claude-3-7-sonnet Accessed: 2025-03-20.

[3] Dorian C. Arnold, Dong H. Ahn, Bronis R. de Supinski, Gregory L. Lee, Barton P. Miller, and Martin Schulz. 2007. Stack Trace Analysis for Large Scale Debugging. In *2007 IEEE International Parallel and Distributed Processing Symposium*. 1–10. doi:10.1109/IPDPS.2007.370254

[4] Babak Behzad, Surendra Byna, and Marc Snir. 2019. Optimizing I/O performance of HPC applications with autotuning. *ACM Transactions on Parallel Computing (TOPC)* 5, 4 (2019), 1–27.

[5] Jean Luca Bez, Hammad Ather, and Suren Byna. 2022. Drishti: Guiding End-Users in the I/O Optimization Journey. In *2022 IEEE/ACM International Parallel Data Systems Workshop (PDSW)*. 1–6. doi:10.1109/PDSW56643.2022.00006

[6] Sebastian Borgeaud, Arthur Mensch, Jordan Hoffmann, Trevor Cai, Eliza Rutherford, Katie Millican, George van den Driessche, Jean-Baptiste Lespiau, Bogdan Damoc, Aidan Clark, Diego de Las Casas, Aurelia Guy, Jacob Menick, Roman Ring, Tom Hennigan, Saffron Huang, Loren Maggiore, Chris Jones, Albin Cassirer, Andy Brock, Michela Paganini, Geoffrey Irving, Oriol Vinyals, Simon Osindero, Karen Simonyan, Jack W. Rae, Erich Elsen, and Laurent Sifre. 2022. Improving language models by retrieving from trillions of tokens. arXiv:2112.04426 [cs.CL] https://arxiv.org/abs/2112.04426

[7] Peter Braam. 2019. The Lustre storage architecture. *arXiv preprint arXiv:1903.01955* (2019).

[8] Andres M Bran, Sam Cox, Oliver Schilter, Carlo Baldassari, Andrew D White, and Philippe Schwaller. 2023. Chemcrow: Augmenting large-language models with chemistry tools. *arXiv preprint arXiv:2304.05376* (2023).

[9] Sven Breuner. 2025. elbencho: A Distributed Storage Benchmark for Files, Objects & Blocks with Support for GPUs. https://github.com/breuner/elbencho Accessed: 2025-04-03.

[10] Zhen Cao, Vasily Tarasov, Sachin Tiwari, and Erez Zadok. 2018. Towards Better Understanding of Black-box {Auto-Tuning}: A Comparative Analysis for Storage Systems. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 893–907.

[11] Philip Carns, Kevin Harms, William Allcock, Charles Bacon, Samuel Lang, Robert Latham, and Robert Ross. 2011. Understanding and Improving Computational Science Storage Access through Continuous Characterization. *ACM Trans. Storage* 7, 3, Article 8 (oct 2011), 26 pages. doi:10.1145/2027066.2027068

[12] Wen Cheng, Shijun Deng, Lingfang Zeng, Yang Wang, and André Brinkmann. 2021. AIOC2: A deep Q-learning approach to autonomic I/O congestion control in Lustre. *Parallel Comput.* 108 (2021), 102855.

[13] Matthieu Dorier, Romain Egele, Prasanna Balaprakash, Jaehoon Koo, Sandeep Madireddy, Srinivasan Ramesh, Allen D Malony, and Rob Ross. 2022. Hpc storage service autotuning using variational-autoencoder-guided asynchronous bayesian optimization. In *2022 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 381–393.

[14] Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. 2009. Tuning database configuration parameters with ituned. *Proceedings of the VLDB Endowment* 2, 1 (2009), 1246–1257.

[15] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, et al. 2019. The Design and Operation of {CloudLab}. In *2019 USENIX annual technical conference (USENIX ATC 19)*. 1–14.

[16] Chris Egersdoerfer, Arnav Sareen, Jean Luca Bez, Suren Byna, and Dong Dai. 2024. ION: Navigating the HPC I/O Optimization Journey using Large Language Models. In *Proceedings of the 16th ACM Workshop on Hot Topics in Storage and File Systems*. 86–92.

[17] John Fragalla. 2014. Configure, Tune, and Benchmark a Lustre Filesystem. Presentation at Rice University. https://bpb-us-e1.wpmucdn.com/blogs.rice.edu/dist/0/2327/files/2014/03/Fragalla.pdf Accessed: 2025-04-02.

[18] Anjus George, Rick Mohr, James Simmons, and Sarp Oral. 2021. *Understanding Lustre Internals*. Technical Report. Oak Ridge National Laboratory (ORNL), Oak Ridge, TN (United States).

[19] Alireza Ghafarollahi and Markus J Buehler. 2024. SciAgents: Automating Scientific Discovery Through Bioinspired Multi-Agent Intelligent Graph Reasoning. *Advanced Materials* (2024), 2413523.

[20] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, and Aiesha Letman et. al. 2024. The Llama 3 Herd of Models. arXiv:2407.21783 [cs.AI] https://arxiv.org/abs/2407.21783

[21] Michael Hennecke. 2020. Daos: A scale-out high performance storage stack for storage class memory. *Supercomputing frontiers* 40 (2020).

[22] Kelly Hong, Anton Troynikov, and Jeff Huber. 2025. *Context Rot: How Increasing Input Tokens Impacts LLM Performance*. Technical Report. Chroma. https://research.trychroma.com/context-rot

[23] Xinmei Huang, Haoyang Li, Jing Zhang, Xinxin Zhao, Zhiming Yao, Yiyan Li, Tieying Zhang, Jianjun Chen, Hong Chen, and Cuiping Li. 2025. E2ETune: End-to-End Knob Tuning via Fine-tuned Generative Language Model. arXiv:2404.11581 [cs.AI] https://arxiv.org/abs/2404.11581

[24] Sunggon Kim, Alex Sim, Kesheng Wu, Suren Byna, Teng Wang, Yongseok Son, and Hyeonsang Eom. 2019. DCA-IO: A dynamic I/O control scheme for parallel and distributed file systems. In *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE, 351–360.

[25] Seong Jo Kim, Seung Woo Son, Wei-keng Liao, Mahmut Kandemir, Rajeev Thakur, and Alok Choudhary. 2012. IOPin: Runtime Profiling of Parallel I/O in HPC Systems. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*. 18–23. doi:10.1109/SC.Companion.2012.14

[26] Patrick Tser Jern Kon, Jiachen Liu, Qiuyi Ding, Yiming Qiu, Zhenning Yang, Yibo Huang, Jayanth Srinivasa, Myungjin Lee, Mosharaf Chowdhury, and Ang Chen. 2025. Curie: Toward rigorous and automated scientific experimentation with ai agents. *arXiv preprint arXiv:2502.16069* (2025).

[27] Julian M. Kunkel, John Bent, Jay Lofstead, and George S. Markomanolis. 2016. Establishing the IO-500 Benchmark. White Paper, the IO500 Foundation, Tech. [Online]. Available: https://www.vi4io.org/_media/io500/about/io500-establishing.pdf.

[28] Julian Martin Kunkel and George S Markomanolis. 2018. Understanding metadata latency with MDWorkbench. In *High Performance Computing: ISC High Performance 2018 International Workshops, Frankfurt/Main, Germany, June 28, 2018, Revised Selected Papers 33*. Springer, 75–88.

[29] LangChain. 2025. LangChain: Framework for Developing LLM-Powered Applications. https://www.langchain.com/ Accessed: 2025-03-20.

[30] LangGenius. 2025. Dify: Open-Source Platform for LLM Application Development. https://dify.ai/ Accessed: 2025-03-20.

[31] Jiale Lao, Yibo Wang, Yufei Li, Jianping Wang, Yunjia Zhang, Zhiyuan Cheng, Wanghu Chen, Mingjie Tang, and Jianguo Wang. 2023. Gptuner: A manual-reading database tuning system via gpt-guided bayesian optimization. *arXiv preprint arXiv:2311.03157* (2023).

[32] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems* 33 (2020), 9459–9474.

[33] Junkai Li, Yunghwei Lai, Weitao Li, Jingyi Ren, Meng Zhang, Xinhui Kang, Siyu Wang, Peng Li, Ya-Qin Zhang, Weizhi Ma, et al. 2024. Agent hospital: A simulacrum of hospital with evolvable medical agents. *arXiv preprint arXiv:2405.02957* (2024).

[34] Yan Li, Kenneth Chang, Oceane Bel, Ethan L Miller, and Darrell DE Long. 2017. CAPES: Unsupervised storage performance tuning using neural network-based deep reinforcement learning. In *Proceedings of the international conference for high performance computing, networking, storage and analysis*. 1–14.

[35] Yan Li, Xiaoyuan Lu, Ethan L Miller, and Darrell DE Long. 2015. ASCAR: Automating contention management for high-performance storage systems. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 1–16.

[36] Jerry Liu. 2022. *LlamaIndex*. doi:10.5281/zenodo.1234

[37] Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2023. Lost in the Middle: How Language Models Use Long Contexts. arXiv:2307.03172.

[38] LNLL. 2012. ior. https://github.com/hpc/ior/tree/d68c9755bbadb48ba9cc0da7b495b3f089569102

[39] Junyu Luo, Weizhi Zhang, Ye Yuan, Yusheng Zhao, Junwei Yang, Yiyang Gu, Bohan Wu, Binqi Chen, Ziyue Qiao, Qingqing Long, et al. 2025. Large Language Model Agent: A Survey on Methodology, Applications and Challenges. *arXiv preprint arXiv:2503.21460* (2025).

[40] Wenhao Lyu, Youyou Lu, Jiwu Shu, and Wei Zhao. 2020. Sapphire: Automatic configuration recommendation for distributed storage systems. *arXiv preprint arXiv:2007.03220* (2020).

[41] Microsoft. 2025. AutoGen: Multi-Agent Framework for LLM Applications. https://github.com/microsoft/autogen Accessed: 2025-03-20.

[42] Mark C Miller. 2015. *Design & implementation of macsio*. Technical Report. Lawrence Livermore National Laboratory (LLNL), Livermore, CA (United States).

[43] Monica. 2025. Manus: Autonomous AI Agent for Complex Tasks. https://manus.im/ Accessed: 2025-03-20.

[44] Nafiseh Moti, André Brinkmann, Marc-André Vef, Philippe Deniel, Jesus Carretero, Philip Carns, Jean-Thomas Acquaviva, and Reza Salkhordeh. 2023. The I/O Trace Initiative: Building a Collaborative I/O Archive to Advance HPC. In *Proceedings of the SC'23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*. 1216–1222.

[45] NERSC. 2025. NERSC Documentation: I/O tuning. https://docs.nersc.gov/performance/io/lustre/ Accessed: 2025-03-20.

[46] Sarah Neuwirth, Feiyi Wang, Sarp Oral, and Ulrich Bruening. 2017. Automatic and transparent resource contention mitigation for improving large-scale parallel file system performance. In *2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 604–613.

[47] Open Interpreter Contributors. 2025. Open Interpreter: A Natural Language Interface for Computers. https://github.com/OpenInterpreter/open-interpreter Accessed: 2025-04-03.

[48] OpenAI. 2024. GPT-4o. https://openai.com/index/hello-gpt-4o/ Accessed: 2025-04-02.

[49] OpenAI. 2024. text-embedding-3-large. https://openai.com/index/new-embedding-models-and-api-updates/ Accessed: 2025-04-02.

[50] OpenAI. 2025. Introducing Deep Research. https://openai.com/index/introducing-deep-research/ Accessed: 2025-03-20.

[51] OpenAI. 2025. Introducing GPT-4.5. https://openai.com/index/introducing-gpt-4-5/ Accessed: 2025-03-20.

[52] OpenAI. 2025. Introducing Operator: A Browser-Based AI Agent. https://openai.com/index/operator Accessed: 2025-03-20.

[53] Oracle Corporation and Intel Corporation. 2017. *Lustre* Software Release 2.x Operations Manual.* https://doc.lustre.org/lustre_manual.pdf Accessed: 2025-04-02.

[54] Yingjin Qian, Xi Li, Shuichi Ihara, Lingfang Zeng, Jürgen Kaiser, Tim Süß, and André Brinkmann. 2017. A configurable rule based classful token bucket filter network request scheduler for the lustre file system. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12.

[55] Neeraj Rajesh, Keith Bateman, Jean Luca Bez, Suren Byna, Anthony Kougkas, and Xian-He Sun. 2024. TunIO: An AI-powered Framework for Optimizing HPC I/O. In *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 494–505. doi:10.1109/IPDPS57955.2024.00050

[56] Md Hasanur Rashid, Youbiao He, Forrest Sheng Bao, and Dong Dai. 2023. IOPath-Tune: Adaptive Online Parameter Tuning for Parallel File System I/O Path. *arXiv preprint arXiv:2301.06622* (2023).

[57] Leonard Salewski, Stephan Alaniz, Isabel Rio-Torto, Eric Schulz, and Zeynep Akata. 2023. In-Context Impersonation Reveals Large Language Models' Strengths and Biases. arXiv:2305.14930 [cs.AI]

[58] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems* 36 (2023), 8634–8652.

[59] Eddy Taillefer, Koji Tanaka, and Shuichi Ihara. 2019. Performance Evaluation of Lustre on All-Flash Storage System at OIST. Presentation at the Lustre User Group (LUG) Conference. https://wiki.lustre.org/images/a/a2/LUG2019-Performance_Lustre_All_Flash_OIST-Tanaka.pdf Accessed: 2025-04-02.

[60] Immanuel Trummer. 2022. DB-BERT: a Database Tuning Tool that" Reads the Manual". In *Proceedings of the 2022 international conference on management of data*. 190–203.

[61] Dana Van Aken, Andrew Pavlo, Geoffrey J Gordon, and Bohan Zhang. 2017. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM international conference on management of data*. 1009–1024.

[62] Jeffrey S. Vetter and Michael O. McCracken. 2001. Statistical scalability analysis of communication operations in distributed applications. In *Proceedings of the Eighth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming* (Snowbird, Utah, USA) *(PPoPP '01)*. Association for Computing Machinery, New York, NY, USA, 123–132. doi:10.1145/379539.379590

[63] Chen Wang, Jinghan Sun, Marc Snir, Kathryn Mohror, and Elsa Gonsiorowski. 2020. Recorder 2.0: Efficient Parallel I/O Tracing and Analysis. In *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 1–8. doi:10.1109/IPDPSW50202.2020.00176

[64] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2023. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. arXiv:2201.11903 [cs.CL] https://arxiv.org/abs/2201.11903

[65] Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jiashu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, et al. 2019. An end-to-end automatic cloud database tuning system using deep reinforcement learning. In *Proceedings of the 2019 international conference on management of data*. 415–432.

[66] Weiqun Zhang, Andrew Myers, Kevin Gott, Ann Almgren, and John Bell. 2021. AMReX: Block-structured adaptive mesh refinement for multiphysics applications. *The International Journal of High Performance Computing Applications* 35, 6 (2021), 508–526.

[67] Houkun Zhu, Dominik Scheinert, Lauritz Thamsen, Kordian Gontarska, and Odej Kao. 2022. Magpie: Automatically tuning static parameters for distributed file systems using deep reinforcement learning. In *2022 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 150–159.

Appendix: Artifact Description

A Overview of Contributions and Artifacts

A.1 Paper's Main Contributions

C$_1$ An autonomous framework that leverages agentic large language models (LLMs) to tune parallel file systems (PFSs) parameters to maximize application performance, performing similarly to or better than human experts within a small number of attempts.

C$_2$ The framework implements an LLM-based agent to autonomously analyze applications' runtime I/O behavior, generate tuning predictions, and decide when to end the tuning process.

C$_3$ The framework leverages a rule set accumulated from previous tuning experiences to improve tuning performance for workloads used to generate the rule set, as well as new, unseen workloads.

A.2 Computational Artifact

A$_1$ https://zenodo.org/records/15880567

Artifact ID	Contributions Supported	Related Paper Elements
A_1	C_1	Figure 5
	C_2	Figures 6 and 7
	C_3	Figures 6 and 7

B Artifact Identification

B.1 Computational Artifact A_1

Relation To Contributions

The artifact encapsulates the computational components necessary for implementing our proposed framework on an HPC system. This includes the implementations of all key contributions, as well as the processes used to execute each component, as outlined in the contributions.

Expected Results

(1) To support contribution C_1, experiments will compare the performance of file system parameter configurations selected by the proposed agent framework with both the default settings of the system and those created by an independent system expert across five benchmark workloads. This will show the framework's ability to perform similarly to or better than human expert baselines. The framework should perform significantly better than the default on all benchmark workloads and achieve a similar level of performance to the expert baseline for all benchmark workloads.

(2) To support contribution C_2, the framework will be applied to 5 benchmark workloads and 3 real-application workloads to generate tuning traces highlighting the performance of each tuning attempt made by the agent framework during the tuning process. These will be generated without the addition of any prior tuning knowledge of the applications, serving as the baseline cases (No Rules) in Figures 6 and 7. The results will demonstrate the agent's ability to quickly find high-performance parameter configurations and decide to terminate its tuning process after a reasonable exploration of the parameter space in a small number of iterations.

(3) For contribution C_3, the framework will tune the same 8 workloads while leveraging a combined set of tuning rules accumulated only from the tuning experiences of the 5 benchmark workloads. The results of this will complete Figures 6 and 7, as they represent the alternative case (With Rules). The results will also show the potential benefits of applying the combined rule set, including improved tuning efficiency and avoidance of low-performance parameter configurations.

Expected Reproduction Time (in Minutes)

To set up the artifact for evaluation, a public CloudLab profile and software configuration scripts are included. Instantiating the profile and executing the required setup scripts is estimated to take 2 hours (120 minutes). Once the artifact is set up, the included execution scripts will run the experiments, collect the results, and generate the plots associated with each contribution. This process in its entirety is estimated to take 20 hours (1,200 minutes). Since the execution scripts will automatically create the necessary plots, the analysis should take only a few minutes. In total, the process should take approximately 1,320 minutes.

Also note that since the reproduction of the results relies on the use of two proprietary LLM models, there is a monetary cost component associated with reproduction. The total monetary cost of reproducing the results is estimated to be less than $20 USD in total.

Artifact Setup (incl. Inputs)

The artifact setup procedure is simple, as a large portion of the environment is pre-configured in the CloudLab profile provided, and most extraneous configuration is automated by shell scripts. Specifically, once the CloudLab profile is installed on an allocated group of resources, only a small number of environment variables need to be defined, and the setup scripts need to be launched.

Hardware. We evaluated our artifact on 11 machines on the Cloudlab platform, all of which were in the c220g5 category. All of the machines in this category share the same hardware capabilities. Specifically, each machine features an Intel Xeon Silver 4114 processor with 10 physical CPU cores, approximately 196 GB of memory, and is connected via a 10 Gbps network switch. While the framework is not dependent on this hardware, the provided setup and execution scripts have only been verified to work on CloudLab resources from the Wisconsin cluster of groups c220g1, c220g2, and c220g5.

Software. Reproduction of the listed contributions using the provided artifact is only reliant upon the CloudLab profile and the code contained within the artifact itself. During the setup procedure, the cluster will be configured with Lustre version 2.15.5, the Darshan I/O profiling library, and several Python packages installed from an included 'requirements.txt' file. All additional dependencies, including the application source codes used for evaluation, are pre-installed on the CloudLab profile. The CloudLab profile can be accessed using the following link: https://www.cloudlab.us/p/DIRR/STELLAR_Profile

Datasets / Inputs. All datasets used during evaluation are contained within or generated by the artifact itself. There are minimal additional inputs in the form of environment variables required to set up and run the artifact. Specifically, since the evaluations to be reproduced by the artifact, presented in Figures 5, 6, and 7, use Anthropic's Claude-3.7-Sonnet model as the Tuning Agent and OpenAI's GPT-4.1 model as the Analysis Agent, both an OpenAI and Anthropic API key (OPENAI_API_KEY and ANTHROPIC_API_KEY, respectively) must be set in the environment of the client node from which the evaluation scripts are launched. Beyond this, there are no additional inputs or modifications required unless the cluster layout used during reproduction of the results is different from the expected layout (5 clients and 6 servers).

Installation and Deployment. The artifact only needs to be installed and deployed to a single client node as it includes automated functionality to manage the necessary actions across all client nodes without the need for additional manual installation on any other nodes.

Artifact Execution

The artifact's workflow consists of four tasks: T_1, T_2, T_3 and T_4. The first task, T_1, is to set up the CloudLab cluster using the CloudLab profile linked previously. The second task, T_2, is to download the artifact to one of the nodes in the CloudLab cluster and configure software dependencies by running the setup scripts found in the setup_scripts directory of the artifact contents as well as adding the necessary API keys to the environment (Anthropic and OpenAI). The third task, T_3, is to run the combined execution script found in the run_scripts directory of the artifact. This script will automatically run the necessary experiments to support all of the described contributions and format the final results as plots. The fourth task, T_4, involves analyzing the plots generated by task T_3 to confirm the successful replication of the reported results.

Artifact Analysis (incl. Outputs)

The output produced by running the outlined tasks is a set of plots with results similar to those reported. The plots are organized in subdirectories Eval_1, Eval_2, and Eval_3, which correspond to Figures 5, 6, and 7, respectively. For evaluations in which we directly compare application runtime (Eval_1), we report the actual runtime of the application. Alternatively, for those reporting speedup (Eval_2 and Eval_3), the speedup is calculated relative to the average runtime of the application using the default file system parameter configuration.