Efficiently Executing High-throughput Lightweight LLM Inference Applications on Heterogeneous Opportunistic GPU Clusters with Pervasive Context Management

Thanh Son Phung, Douglas Thain
{tphung,dthain}@nd.edu
University of Notre Dame
Notre Dame, IN, USA

Abstract

The rise of Generative AI introduces a new class of HPC workloads that integrates lightweight LLMs with traditional high-throughput applications to accelerate scientific discovery. The current design of HPC clusters is inadequate to support this new class however, either incurring long wait times on static batch queues or repeatedly paying expensive LLM startup costs upon resource preemption. To circumvent both the long queues and high startup costs, we propose to "decouple" the LLM initialization context from the actual LLM inferences, and retain the context in GPUs until it is no longer needed, a technique we term "Pervasive Context Management". We transform a fact verification application to enable this technique, allowing it to reduce its execution time by 72.1% (from 3 hours to 48 minutes) using the same amount of GPUs, and scale opportunistically on 32.8% of all GPUs in the cluster and further reduce the execution time to 13 minutes.

ACM Reference Format:
Thanh Son Phung, Douglas Thain. 2025. Efficiently Executing High-throughput Lightweight LLM Inference Applications on Heterogeneous Opportunistic GPU Clusters with Pervasive Context Management. In . ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Background. Large Language Models (LLMs)[7, 10, 50] have emerged as a transformative technology, demonstrating a remarkable capacity for discerning intricate patterns within vast datasets, and promise to be exceptionally powerful tools across diverse scientific domains. Consequently, a **new class of HPC workloads** is on the rise that integrates lightweight LLM inferencing (typically with *billions* of parameters) into traditional high-throughput applications to accelerate the pace of scientific discovery. This trend, exemplified by LLM-backed advancements in protein folding[44, 48, 54, 57] and distributed AI-driven scientific computing frameworks[19, 21, 55], introduces a novel and challenging workload class to High-Performance Computing (HPC) clusters.

Conference'17, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

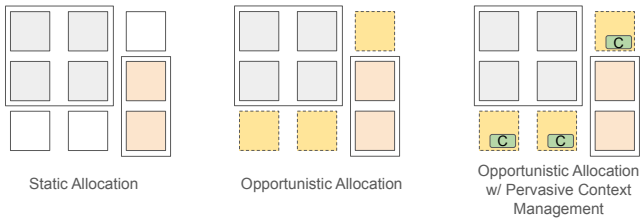


Figure 1: Different Resource Allocation Models
(Left) Static allocation model exclusively allocates a block of GPUs to an application, leading to long wait times on batch queues and cluster-wide resource under-utilization. (Middle) Opportunistic allocation model allows applications to run on transiently available GPUs, but incurs a high startup penalty from LLM context initialization. (Right) Our proposed model with Pervasive Context Management: LLM contexts - denoted as C - are decoupled from inferences and deliberately persisted on GPUs to alleviate the high startup penalty.

Currently, two primary allocation models exist for executing these LLM-integrated applications on GPU clusters. In the *static allocation model*, an application requests an exclusive and fixed-size batch of GPUs for a given period of time. This approach ensures predictable, maximal performance by eliminating resource contention issues during its execution. However, **its practicality is diminishing**: the combination of surging demand for GPU resources and high GPU market prices[16, 27, 40] has led to a huge number of GPU-accelerated applications oversubscribing a much smaller number of GPUs. This oversubscription results in heavily contended job queues, forcing applications to have significant wait times[17, 30, 32]. Furthermore, the rigidity of static allocations can lead to cluster-wide resource under-utilization due to resource fragmentation, wasting idle GPU cycles[25, 26, 58].

Instead of relying on fixed-size exclusive batches of GPUs, the *opportunistic allocation model* leverages dynamic resources in which GPUs can join and leave the application's resource pool at any given moment depending on the state of the cluster. The opportunistic resource pool increases its capacity as more fixed-size exclusive jobs finish their executions and release allocated GPUs, and decreases its capacity as incoming fixed-size jobs are allocated. To take advantage of this type of transient resource and circumvent the long queues associated with static allocations, high-throughput applications have traditionally been architected to operate on this elastic resource pool to maximize the throughput over a long period of time. Furthermore, this elasticity helps increase the cluster-wide resource utilization as previously idle GPU cycles are now filled

with high-throughput applications running opportunistically. Note that to accommodate this dynamicity, monolithic applications are typically decomposed into many smaller tasks which are independently executed on remote nodes or GPUs.

Motivation. The integration of lightweight LLMs, however, presents a fundamental challenge to this opportunistic paradigm. The volatile nature of the resource pool means that an executing task may be preempted at any time. For conventional HPC tasks, the cost of such a preemption is often manageable. For a lightweight LLM inference task, it is **catastrophic**. The startup procedure for a single task—loading a multi-billion parameter model from a distributed filesystem to local disk, then into host memory, and finally into GPU memory—is an I/O-bound process that can take minutes. Upon preemption, this entire costly procedure must be repeated from scratch on a new resource, and frequently dominates the task's useful computation time. While batching many inferences can amortize this cost, it simultaneously linearly increases the task's runtime, elevating the risk that the task will be preempted before completion and loses all its work. Thus, a user must be deliberate and careful in tuning the inference batch size per task to balance between the startup cost and the preemption rate.

This tuning process unfortunately adds another layer of complexity for non-technical users, and the optimal batch size, which depends on the startup cost and the preemption rate, is not straightforward to derive either, even for technical users. While the startup cost can be profiled on a local GPU, a typical GPU cluster has many models of GPUs, reflecting the cluster's evolution over time as older GPUs are gradually phased out and newer GPUs are incrementally added in. Table 1 shows the heterogeneity of our local HPC cluster with 8 major GPU models spanning 8 years and accounting for 75% of all GPUs in the cluster. This **heterogeneity** thus complicates the validity of an application's runtime profiling with a local GPU: opportunistic GPUs can come with *any* GPU model, and an optimal batch size for one GPU model doesn't necessarily translate to optimality on other GPUs. Furthermore, the rate of GPU preemption is completely *dynamic and unpredictable over time* as it depends on, among other factors, the current load on the local cluster, the arbitrary resource demands from other GPU jobs, and the specific allocation and scheduling policies of the cluster manager.

All issues we discussed above form the following central research question of this paper: **Is there a way that we can transform existing high-throughput lightweight LLM inference applications such that they can execute efficiently on heterogeneous opportunistic GPU resources without repeatedly paying the startup cost upon preemption and/or incurring a high toll on HPC users?**

Limitation of state-of-the-art approaches. Existing methodologies for managing elastic and fault-tolerant computations are not well-suited to resolve the core tension between the high startup cost of LLM inference and the volatile nature of opportunistic resources.

First, conventional autoscaling frameworks[11, 13, 41, 61] are fundamentally mismatched with the opportunistic resource model. Autoscaling systems operate on a **proactive** principle: the application itself initiates scale-up or scale-down events based on its internal workload, such as a rising queue of user requests. In the opportunistic HPC setting, the application is purely **reactive**: it has no control over its allocated resources, and GPUs are allocated and preempted by the cluster manager based on external priorities. Therefore, an application cannot "request" more resources to meet demand, nor can it "release" them gracefully: it must simply adapt to the resources it is given, whenever they appear or disappear.

Device Name	Release Year	Count
NVIDIA Quadro RTX 6000	2018	106
NVIDIA A10	2021	78
NVIDIA TITAN X (Pascal)	2016	69
NVIDIA GeForce GTX 1080 Ti	2017	63
NVIDIA RTX 6000 Ada Generation	2022	36
NVIDIA GeForce GTX TITAN X	2015	34
NVIDIA A40	2020	26
NVIDIA H100 80GB HBM3	2023	15

Table 1: 8 Major GPU Models in the Local Cluster

Our local HPC cluster contains 567 GPUs in total, with 75% of them in one of the 8 major models in the table, showing the heterogeneity of the available GPUs and emphasizing the challenges of running scientific applications opportunistically on heterogeneous resources.

Second, traditional fault-tolerance techniques like progress checkpointing [22, 24, 45] offer only a partial and inadequate solution. While checkpointing the results of completed inferences allows a task to protect its progress, it does not address the primary problem: the prohibitive model loading cost. Upon preemption, a new task must still be instantiated on a different GPU, incurring the full startup penalty before it can resume work from the last checkpoint.

Key insights and contributions. Our approach is founded on a key insight: the primary performance bottleneck is not the computation but the tight coupling of inference execution with expensive context initialization. We propose to **decouple** these elements by elevating the LLM context (e.g., the model loaded in a GPU) to a first-class, persistent entity within the cluster, a technique we term Pervasive Context Management.

In this technique, contexts are deliberately held on remote nodes for reuse across multiple tasks rather than being torn down after each task's completion. When a task is preempted from one GPU, it is simply requeued and rapidly rescheduled to another opportunistic GPU that already holds the required context. Furthermore, when new GPUs join the resource pool, our system can bootstrap them by transferring an existing context template directly from another node, cutting down data transfer time and preventing a bottleneck at the shared filesystem. This strategy effectively makes the high cost of context initialization a one-time, amortizable expense. It also alleviates the complex problem of searching for an optimal batch size as the startup cost is now shared across many independent tasks. Figure 1 illustrates the conceptual differences between the static allocation model, the standard opportunistic model, and our proposed model that combines opportunistic resources with Pervasive Context Management.

Based on these ideas, this paper makes the following contributions:

(1) We implemented a high-throughput context-agnostic fact verification application backed by lightweight LLMs in the Parsl-TaskVine distributed data-intensive framework[12, 36, 46].

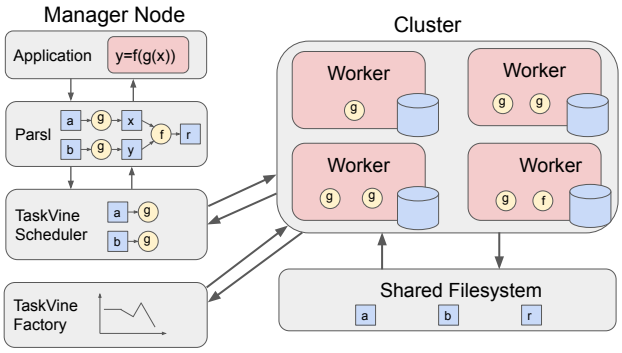


Figure 2: Overview of the Parsl-TaskVine Framework
The application defines LLM computations via Python functions and passes them to Parsl. Parsl manages dependencies between functions and sends ready ones to the TaskVine scheduler. The scheduler manages resources on workers, schedules functions to available ones, and controls their execution and I/O patterns. The TaskVine factory monitors the opportunistic resources and adjusts the quantity of workers accordingly.

(2) We detailed a quick application transformation that makes the application context-aware via the Pervasive Context Management technique.
(3) We conducted a comprehensive evaluation demonstrating that this transformation, enabled by Pervasive Context Management, significantly reduces the end-to-end execution time of the application by up to 72.1% (from 3 hours to 48 minutes), and allows the application to scale opportunistically on up to 32.8% of all GPUs in the cluster and further reduces the execution time to 13 minutes.

Limitation of the proposed approach. The primary constraint of the proposed approach is that our Pervasive Context Management system is designed for lightweight LLMs (up to billions of parameters depending on the GPU setup per node) whose context can fit within the resources of a single compute node. This is a direct consequence of the nature of opportunistic resources in HPC clusters, which are typically allocated and preempted on a per-node basis. Additionally, our system introduces its own management overhead for replicating and caching contexts, and its effectiveness is contingent on this overhead remaining substantially lower than the cost of repeated cold starts from a shared filesystem.

2 Implementation of a Context-Agnostic Fact Verification Application

Overview of the Parsl-TaskVine framework. The underlying framework that powers the fact verification application is the integrated software stack of two dynamic workflow systems - Parsl[12] and TaskVine[46]. Parsl is a Python-native parallel library that allows users to express their computational needs via generic Python functions and automatically scales the computation on thousands of compute nodes, mainly focusing on flexibility, portability, and ease of use. TaskVine is a low-level data-intensive workflow execution engine that allows users to express low-level details about

```python
from parsl import python_app
@python_app
def infer(model_path, claims):
    ...
    model = AutoModel.from_pretrained(model_path).to('gpu')
    verdicts = [model.generate(claim) for claim in claims]
    return verdicts
model_path = ...
claims = ...
verdicts = infer(model_path, claims).result()
```

Figure 3: Code Example of a Context-agnostic Fact Verification Application
An inference function is annotated with a Parsl-provided decorator, and remote execution is triggered by invoking the function as usual. Calling the ".result()" method blocks the execution until the result of the invocation is returned.

tasks and their inter-relationships. It then extracts values from the provided information to make intelligent scheduling and optimization decisions that accelerate large-scale data processing applications[37, 38].

Figure 2 shows how these two workflow systems work together in the big picture. On the manager node, a user expresses their application's computational needs (e.g., LLM inferences) via generic Python functions. Once the application is run and these functions are invoked, they are intercepted and passed to Parsl for inter-function dependency management and function-to-task translation. Parsl sends ready tasks to the TaskVine scheduler, where they are examined for common execution and I/O patterns and scheduled for execution on workers accordingly. The TaskVine scheduler manages resources in the system via TaskVine workers, where each worker is a small standalone pilot job that waits for instructions from the TaskVine scheduler and operates duly. Once tasks are completed, workers communicate the results back to the scheduler, which forwards them back to the application level. Note that the TaskVine scheduler does not delegate the local resource management to individual workers: each task comes with a specific amount of resource allocation, and each worker is directed by the TaskVine scheduler on how to utilize any local resource type (CPU, memory, SSD, GPU). The pool of resources is maintained by the TaskVine factory, a daemon-like process that monitors the current resource pool and adjusts it based on a given resource policy and the current load of the cluster.

Implementing the context-agnostic fact verification application. Given this software stack, it is then straightforward for a user to implement a high-throughput lightweight LLM inference application. A user first defines an arbitrary computation involving LLM inferences in a Python function. This function then flows through Parsl and the TaskVine scheduler to a TaskVine worker as a task to be executed. Each worker is allocated with a small number of GPUs such that a given task can run comfortably. Once the task completes, inference results are sent from the worker back to the application as described above. The scheduler has a queue of ready tasks, and its main job is to occupy connected workers with tasks at any given time. Therefore, the application will make progress as long as there are workers connected to the scheduler.

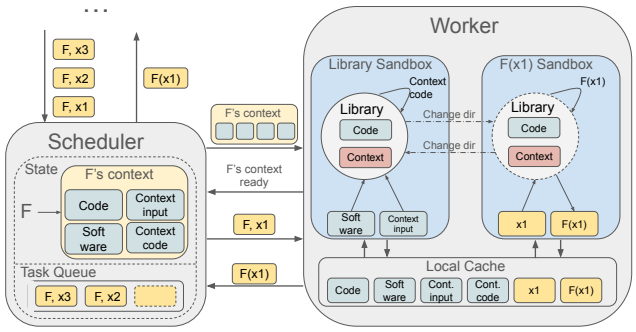


Figure 4: Overview of Pervasive Context Management
The TaskVine scheduler analyzes F for its context upon the first invocation request, and sends it to the worker. The library process in the worker registers F's code and creates and holds an instance of F's context. This context and registered code are then used to execute the request, and subsequent requests reuse this existing context to speed up their executions.

```python
from parsl import python_app
def load_model(model_path):
    ...
    model = AutoModel.from_pretrained(model_path).to('gpu')
    return {'model': model}
@python_app
def infer_model(claims, parsl_spec):
    from parsl import load_variable_from_serverless
    model = load_variable_from_serverless('model')
    verdicts = [model.generate(claim) for claim in claims]
    return verdicts
model_path = ...
claims = ...
parsl_spec = {'context': [load_model, [model_path], {}]}
verdicts = infer_model(claims, parsl_spec).result()
```

Figure 5: Code Example of a Context-aware LLM Inference Application
The previous inference function is now broken down into two new functions: "load_model" that creates the model state in the GPU, and "infer_model" that reuses the existing model state and runs inferences. "infer_model" specifies its model context as an argument, and the remote execution is triggered as usual.

Additionally, this software stack provides a seamless integration with opportunistic resources. The scheduler on the manager node directs all workers on what to do and thus keeps a globally consistent view of the application. This means that workers can leave and join the pool freely as tracked by the TaskVine scheduler and adjusted by the TaskVine factory, and any preempted task is detected, retrieved, and re-inserted into the queue of ready tasks by the scheduler.

Figure 3 shows a simplified implementation of an inference function of the fact verification application (we delay the full description of this application to Section 4.1). To minimize the toll on non-technical users when scaling up a local application, Parsl provides a decorator ("@python_app") that wraps around a typical inference function. Remote execution is triggered when the user invokes the inference function as usual, allowing Parsl to intercept the inference invocation and prepare it for remote execution. The inference is then given to TaskVine for scheduling and execution, and the result is transparently sent back to the application. Alternatively, a user can instruct the local execution to wait for the result by calling the ".result()" method before continuing its execution flow.

Notice that this context-agnostic implementation forces a function invocation to reload its context whenever it is preempted from an opportunistic resource as it couples the expensive context initialization with the actual inference execution into one executable unit (i.e., a task). The next section shows how we can decouple these two computational elements to allow the efficient reuse of a one-time startup cost per unit of opportunistic resources over multiple inference tasks.

3 Transforming the Application to Enable Pervasive Context Management

Overview of the Pervasive Context Management technique. Figure 4 gives a general example of how a function context is retained and reused in the Parsl-TaskVine stack. An application (not shown) starts up and invokes function F three times with arguments x1, x2, and x3, respectively. Parsl (not shown) sees that these three functions don't have any dependency between them and converts them into ready tasks to be sent to the TaskVine scheduler. The scheduler examines F and discovers its context recipe, including F's code, software dependencies, context code, and context inputs, (we discuss the context transformation later in this section) and sends this context recipe to be materialized and hosted on a given worker as part of F(x1) execution. The worker, upon receiving the context recipe, stores all of its components in a local cache and fork-execs a special process called "Library". The Library process is responsible for materializing and hosting F's context from its recipe and will cooperate with the worker to execute subsequent invocations of F. Upon the stage-in of F's context into its sandbox, the library registers F's code, executes the context code, stores the resulting context as an internal state in its process, and lets the worker know it's ready for invocations of F. The scheduler, upon receiving this ack from the worker, sends the first invocation request of (F, x1). The worker stores x1 in its cache, creates a sandbox for the invocation, and pings the library. The library then changes its working directory to F(x1)'s sandbox and executes the invocation directly in its address space, which already contains F's context, before returning to its sandbox. The result of the invocation is then returned to scheduler, which marks the completion of F(x1) and forwards the result to the application. Executions of (F, x2) and (F, x3) then reuse F's context via the library and follow the same path (F, x1) took.

Code transformation to enable Pervasive Context Management. It is then effortless to see how The Pervasive Context Management technique benefits the new class of high-throughput lightweight LLM inference applications on opportunistic resources. Since the startup cost of initializing an LLM is expensive, an application should define it as a context to the actual inference task. When

newly available opportunistic resources arrive to the resource pool, the manager sends the context template to be initialized once per node and retained for subsequent reuses. As multiple inference tasks arrive to the task queue, the manager sends them to nodes that already have the context initialized for immediate inference execution. When resources are preempted by the cluster, these tasks are seamlessly requeued by the manager for execution on other nodes that already host the needed context, eliminating the need to reinitialize the LLM from scratch per task.

Figure 5 shows a code example of how the fact verification application can be quickly transformed to benefit from this technique. We first decouple, or split, the previous inference function into two new functions: "load_model" and "infer_model". Lines 2-5 define the "load_model" function that creates an LLM context by loading its parameters from disk to GPU and returns this context via a dictionary to the Library process. This dictionary informs the Library of the relevant context to later be exposed to the actual invocation. Lines 7-12 define the actual computation via the "infer_model" function that loads the model directly from the context held by the library (instead of loading it from scratch), executes the inferences, and returns the results. Lines 14-17 connect the missing pieces of the example where the context computation is defined via the `parsl_spec` variable, and "infer_model" brings this context reference along with its inputs to the scheduler for execution.

4 Evaluation

This section begins with an in-depth description of the fact verification application along with the general experiment settings that apply to all evaluation efforts. Our evaluation then aims to answer the following research questions:

- **RQ1 - Application Performance on Static Resources**. How well does the application perform with and without the Pervasive Context Management technique on statically allocated resources?
- **RQ2 - Application Sensitivity to Varying Inference Batch Sizes**. How does enabling Pervasive Context Management help users pick the right inference batch size for the application?
- **RQ3 - Application Performance with Aggressive Resource Preemption**. How well does the context-aware application handle aggressive resource preemption from the cluster manager?
- **RQ4 - Application Performance with Opportunistic Resources**. How well does the context-aware application scale opportunistically when the capacity of transiently available resources in the cluster fluctuates?

4.1 Experiment Settings

Application. Fact verification is an active area of research given the lightning rise of online mis- and dis-information[23, 59]. Our application, Prompt for Fact (PfF), aims to find the *optimal* prompt for a given LLM where it is used as a fact verifier to check the correctness of an arbitrary claim. Specifically, we use the training data from FEVER[52] as our dataset containing 145,449 claims, each of which is labeled with either `SUPPORTED`, `REFUTED`, or `NOT ENOUGH INFO`. Each claim contains a statement about a given subject and a list of references to relevant Wikipedia pages. Per the LLM, we use the recently released SmolLM2 model with 1.7 billion parameters[8]. Our application takes the LLM and a prompt template, runs a full inference sweep across the dataset, and returns the fact verification accuracy. Note that the Pervasive Context Management technique is *not* limited to a specific model or application and is applicable to all high-throughput lightweight LLM inference applications subject to the limitation as described in Section 1.

Local cluster. Our local cluster runs two resource managers: Altair Grid Engine (AGE)[18] as the main static batch manager, and HTCondor[51] as a resource backfiller on AGE's unallocated machines. There are 567 GPUs in total in the cluster with 18 different models (see Table 1 for 8 major GPU models). We run all experiments through the HTCondor resource manager. Our cluster provides access to data via the Panasas ActiveStor 16[43, 56] shared filesystem with 77 nodes and supports up to 84 Gbs/s read bandwidth and 94k read IOPS.

Parsl-TaskVine framework. We configure parameters of our Parsl-TaskVine software stack as follows. We enable the peer transfer feature that allows workers to communicate and send arbitrary data between each other, which allows workers to send context templates in a peer-to-peer fashion and bootstrap newly arrived workers. Each task's resource allocation includes 2 cores, 10 GBs of memory, 20 GB of disk, and 1 GPU, providing a comfortable amount of resources for a smooth inference execution. Each TaskVine worker has 2 cores, 10 GBs of memory, 70 GBs of disk, and 1 GPU, thus providing the worker with just enough resources to run tasks in a 1-to-1 manner to preserve claimed opportunistic resources and plenty of disk space for local caching.

Finally, almost all experiments start with the same resource pool configuration consisting of 20 GPUs, where half are NVIDIA A10 and the other half are NVIDIA TITAN X (Pascal). This approach allows us to not only establish consistency and stability to our measurements and results but also mimic the heterogeneity of the actual GPU cluster. This constraint is removed at the end which allows the application to have access to up to 186 opportunistic GPUs. Storage-wise, the LLM takes up 3.7 GBs of disk and around 7.4 GBs of memory when fully loaded. The application's software dependencies are managed in a Conda[9] environment, containing 308 packages and totalling 10.5 GBs of disk.

4.2 RQ1 - Application Performance with Static Resources

To quantify the impact of the Pervasive Context Management technique on the Prompt for Fact application, we implement the *context-agnostic* version of the application and transform it into two other versions: *partial-context* and *full-context*. We detail the differences between these versions as below:

- **Context-agnostic**. In the *context-agnostic* version, a task makes all I/O calls for the model parameters and software dependencies to the shared filesystem and creates a fresh state of the LLM model from scratch, involving moving GBs of data from the shared filesystem to the local disk of the execution node, to the node's memory, and finally to the node's GPU.

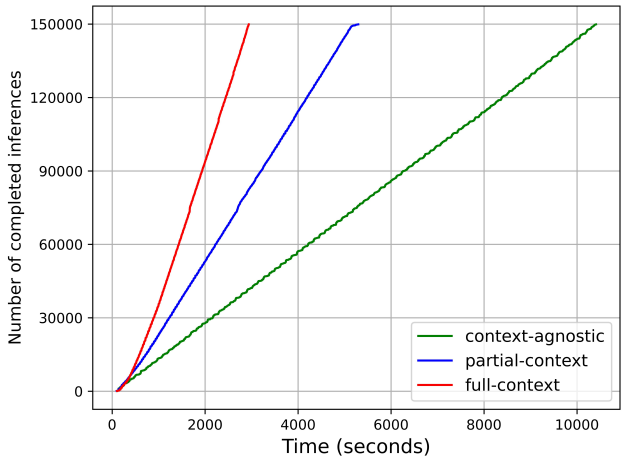


Figure 6: Execution Time of the Prompt for Fact Application with Increasing Level of Context Awareness on Static Resources

The application is run with 3 levels of context awareness: context-agnostic (no context is encoded in the application, forcing LLM state initialization per task), partial-context (LLM context is persisted only on local disk, which includes GBs of model parameters and software dependencies), and full-context (the context further includes the state of the LLM model loaded in the local GPU for quick reuse). The end-to-end execution time of the application reduces drastically as it is run with a higher level of context awareness.

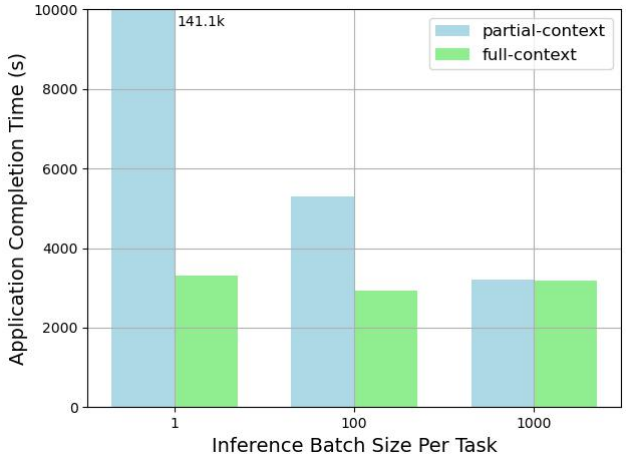


Figure 7: Effect of Inference Batch Size to the Application's Execution Time

Partial-context and full-context are run with 3 different batch sizes: 1, 100, and 1000. Partial-context introduces a large variance of execution time across batch sizes due to the expensive LLM model initialization cost per task, even with cached common input data. On the other hand, full-context stabilizes this range of execution time as each model is initialized once per GPU and reused across multiple tasks instead of once per task, alleviating the effect of a wrong choice of inference batch size.

- **Partial-context**. Instead of transferring GBs of input data per task, *partial-context* caches these common input data on the local disk of the remote nodes so that subsequent tasks that run inferences on the same model can reuse the data available locally instead of making remote I/O calls to the shared filesystem. Each task still has to create a fresh state of the LLM model in the GPU however.
- **Full-context**. This version fully unlocks the benefits of the Pervasive Context Management technique and caches the LLM model state in the GPU for efficient context reuse and fast context transfer between successive task executions. In other words, the LLM model state is created and loaded to the GPU once over a worker's lifetime.

All applications are run on statically allocated resources to isolate the results from other noises (e.g., fluctuations in capacity of opportunistic resources). Each task carries 100 inferences (i.e., inference batch size of 100), resulting in 1,500 tasks per application execution.

Figure 6 shows the end-to-end execution time of 3 versions of the Prompt for Fact application, each totaling 150,000 inferences. The result aligns with our expectation as the version with more context awareness has a significantly lower execution time. Specifically, the *context-agnostic* version runs the longest, over 10.4k seconds, as each task has to repeatedly make remote I/O requests to load GBs common input data from the shared filesystem and construct a new GPU state of the LLM model upon initialization. *Partial-context* instead caches GBs of common input data on local disk of remote nodes and effectively converts most remote I/O requests into local ones. This thus helps bring the execution time down to 5.3k seconds, a significant reduction of the end-to-end execution time of 49.1%. *Full-context* cuts down the execution time even further to 2.9k seconds, a reduction in execution time of 72.1% and 45.3% compared to *context-agnostic* and *partial-context*, respectively. This is because it eliminates both a bulk of unnecessary remote I/O requests for common input data and the need for a task to reconstruct the LLM state in a GPU upon startup as a task can now reuse an available context already initialized in a worker.

We then conclude that transforming the Prompt for Fact application to enable the Pervasive Context Management technique allows a considerably more efficient execution with a huge reduction in the end-to-end execution time on static resources.

4.3 RQ2 - Application Sensitivity to Varying Inference Batch Sizes

Subsection 4.2 demonstrates how the Pervasive Context Management technique enables an efficient execution of the Prompt for Fact application and addresses the first part of the central research question as posed in Section 1. This subsection then addresses the second part of the question and shows how this technique makes it easy for users to pick an inference batch size without worrying about optimal and/or sub-optimal application execution.

Figure 7 shows the execution time of *partial-context* and *full-context* with 3 batch sizes: 1, 100, and 1000 (we skip *context-agnostic* as it is clearly suboptimal as demonstrated in Subsection 4.2.) With a wrong inference batch size of 1, *partial-context* takes a disastrous hit in its performance and needs 141.1k seconds to complete end-to-end.

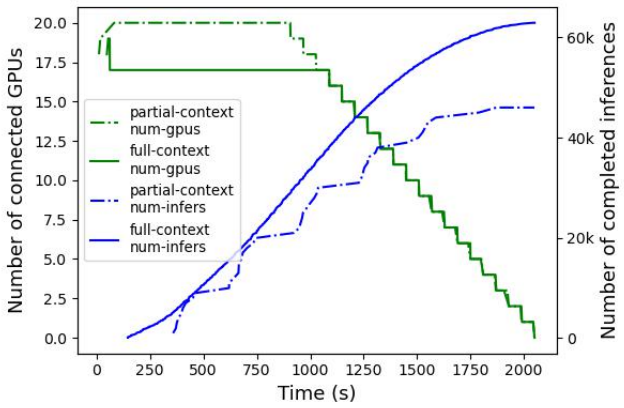


Figure 8: Number of Completed Inferences with Aggressive Resource Preemption

This figure shows the number of completed inferences over time between partial-context and full-context with aggressive resource preemption from the cluster (1 GPU preemption per minute). Despite an early drop of 3 GPUs, full-context still completes 16.9k inferences more than partial-context, and consistently has a higher inference completion rate at any given time.

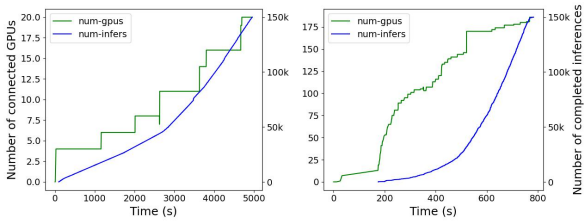
This is because each inference now needs to load the model from scratch, and the overhead of the model initialization dominates the total execution time. Even at the batch size of 100, *partial-context* still takes 5.3k seconds and is still far from the best execution time with a batch size of 1000 at 3.2k seconds.

On the other hand, *full-context* allows a much more stable range of execution time across all batch sizes. The application runs the "worst" with an inference batch size of 1 at 3.3k seconds, and the "best" with batch size of 100 at 2.9k seconds. The range of execution time is thus limited to approximately 400 seconds, or 13.6% of the best execution time, over the range of possible batch sizes of 1000 (from 1 to 1000). Such a stable range of execution time comes from the context cache and reuse between tasks, and the performance difference is only the cumulative overhead of the context transfer that happens once per task. Thus, enabling the Pervasive Context Management technique ensures that the application can never run disastrously with a wrong batch size and its execution is always optimal or near-optimal, removing the worry of batch size tuning from users.

4.4 RQ3 - Application Performance with Aggressive Resource Preemption

Opportunistic resources fluctuate frequently, increasing the capacity as more jobs exit and release their previously claimed resources, and decreasing the capacity as new jobs are allocated and scheduled by the cluster manager. This subsection focuses on the latter and quantifies how well *full-context* executes the application efficiently with aggressive resource preemption.

Figure 8 shows the scenario where resources are preempted aggressively from the application by the cluster manager with the preemption rate of 1 GPU per minute from the 900-second mark



(a) Low opportunistic capacity **(b) High opportunistic capacity**

Figure 9: Application Scaling on Opportunistic Resources over Time

The Prompt for Fact application scales opportunistically on the local cluster over time at low opportunistic capacity (left) and high opportunistic capacity (right). The application scales linearly with the constantly changing amount of available resources.

until the opportunistic resource pool of the application is depleted (we preempt all NVIDIA A10s before NVIDIA Titan X Pascals). For *partial-context*, we can see the "rugged" rate of inference completion that gradually flats out from the opportunistic resource depletion and ends with 46k completed inferences. This is due to the expensive LLM model initialization cost per task that blocks goodput until the LLM model is fully loaded in a GPU, creating the effect of low goodput when the LLM models are being loaded on remote workers, and high goodput when inferences are executed.

Per *full-context*, despite having an early drop of 3 GPUs due to external preemption from the cluster manager, *full-context* still completes 16.9k inferences more than *partial-context* (ends with 62.9k completed inferences), and consistently has a higher inference rate than *partial-context*. Notice the smooth inference completion rate as it shows how tasks from preempted GPUs are seamlessly requeued and rerun with an already GPU-initialized LLM context, removing the rugged-like effect on the application's inference rate and allowing the slope to smoothly flat out at the end compared to that of *partial-context*. Thus, the Pervasive Context Management technique helps the application smoothly make more progress even when resources are preempted aggressively.

4.5 RQ4 - Application Performance with Opportunistic Resources

Finally, we focus on how well the context-aware application (i.e., *full-context*) scales opportunistically in the local cluster. Figure 9 shows the number of opportunistic GPUs used and the number of completed inferences over time for *full-context* when the cluster has low (Subfigure 9a) and high (Subfigure 9b) opportunistic capacity. In Subfigure 9a, the application only starts out with 4 GPUs and gradually goes to 20 opportunistic GPUs, finishing around 5000 seconds. Note that even with a limited amount of GPUs, the application still makes consistent progress as the completed inference rate scales linearly with the amount of opportunistically connected GPUs. On the other hand, when the cluster has many jobs exiting and releasing their claimed GPUs (Subfigure 9b), the application quickly grabs up to 186 transiently available GPUs (32.8% of all GPUs in the cluster) and finishes the execution in only 783 seconds.

This thus shows that the Pervasive Context Management technique allows the Prompt for Fact application to swiftly react and scale to the ever-changing amount of opportunistic GPUs in the cluster over time.

5 Related Works

Spot Instances. The use of underutilized compute capacity is a well-established practice in both commercial cloud computing and High-Performance Computing (HPC), though the implementation and guarantees vary. Cloud providers offer discounted Spot or Preemptible Instances[1, 3, 5] that can be reclaimed at any time, similar to how opportunistic resources are used in HPC. While several studies[33, 34] have explored running stateful applications like LLM inference on these cloud resources, they rely on a critical feature: a preemption warning[2, 4, 6]. This notification period, typically 30 to 120 seconds, allows applications to checkpoint state or transfer work before termination. On the contrary, opportunistic resources in many HPC environments offer no such warning, and preemption is instantaneous, rendering traditional state-saving mechanisms ineffective. Our work addresses this specific challenge by introducing the Pervasive Context Management technique designed to handle the abrupt and unpredictable nature of preemption in HPC clusters by retaining the common computational context between inferences in all connected GPUs.

LLM Inference Optimization. Many works optimize the inference process of huge LLMs by outputting several tokens in one forward pass based on the speculative decoding scheme[14, 29, 47, 49]. This scheme assumes that an LLM generating tokens sequentially takes too much time and resources, especially with easy-to-predict tokens. To speed this up, a smaller LLM is used to predict the next K tokens in advance, and the original LLM can make one forward pass that accepts tokens it agrees with and rejects others instead of making K forward passes. Other works focus on KV cache and memory management on both a single GPU and a pool of GPUs. Kwon et. al.[28] introduce a virtual paging mechanism that divides the dynamically-sized KV cache into blocks to remove GPU memory fragmentation, while Lin et. al. [31] distribute the KV cache and the attention computation to many GPUs. Cloud deployment of inference serving is also an active area of research. Fu et. al.[20] use local storage of individual instances to cache and distribute model checkpoints among each other. Our work extends the usage of local storage to memory and GPUs to hold and distribute the computational context.

Workflow Systems. Workflow systems evolve from the traditional resource managers and allow applications to express complex relationships between tasks via a directed acyclic graph (DAG) instead of a bag of tasks[15, 39, 53, 60]. These systems typically focus on applications' reliability, performance, and portability via novel architectural designs and runtime optimizations, but require users to describe the computational needs in detail via complicated and non-uniform abstractions. More modern workflow systems[12, 35, 42] tackle this usability problem by providing Pythonic abstractions that enable users to wrap their computational needs neatly into Python functions and translating these functions into tasks deployable to remote nodes. Our Parsl-TaskVine integration follows this movement and allows users to easily describe their computations in Python without losing performance, reliability, or portability. The Parsl-TaskVine stack extends this movement one step further with the support of computational context sharing between tasks on contrary to the traditional view of complete inter-task independence.

6 Conclusion

The LLM technology has been improving at an astonishing rate over the years and promises an unprecedented leap in human productivity. The current infrastructure however is ill-equipped to deal with the massive spike in interests and computational demands from LLM scientists and practitioners and requires new approaches to efficient resource management. This paper shows that, through the Pervasive Context Management technique, the new class of high-throughput lightweight LLM inference applications can be run efficiently on heterogeneous opportunistic GPU clusters without either sacrificing performance or imposing a heavy complexity toll on HPC users.

References

[1] Amazon Web Services (AWS) 2025. *Amazon EC2 Spot Instances.* Amazon Web Services (AWS). https://aws.amazon.com/ec2/spot/

[2] Amazon Web Services (AWS) 2025. *Spot Instance Interruption Notices.* Amazon Web Services (AWS). https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/spot-instance-termination-notices.html

[3] Microsoft Azure 2025. *Spot Virtual Machines.* Microsoft Azure. https://azure.microsoft.com/en-us/products/virtual-machines/spot

[4] Microsoft Azure 2025. *Spot Virtual Machines.* Microsoft Azure. https://learn.microsoft.com/en-us/azure/virtual-machines/spot-vms

[5] Google Cloud 2025. *Spot VMs.* Google Cloud. https://cloud.google.com/solutions/spot-vms

[6] Google Cloud 2025. *Spot VMs.* Google Cloud. https://cloud.google.com/compute/docs/instances/spot

[7] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).

[8] Loubna Ben Allal, Anton Lozhkov, Elie Bakouch, Gabriel Martín Blázquez, Guilherme Penedo, Lewis Tunstall, Andrés Marafioti, Hynek Kydlíček, Agustín Piqueres Lajarín, Vaibhav Srivastav, et al. 2025. SmolLM2: When Smol Goes Big–Data-Centric Training of a Small Language Model. *arXiv preprint arXiv:2502.02737* (2025).

[9] Anaconda. 2025. Anaconda Documentation. https://www.anaconda.com/docs/main

[10] Anthropic. 2024. The Claude 3 Model Family: Opus, Sonnet, Haiku. https://www-cdn.anthropic.com/de8ba9b01c9ab7cbabf5c33b80b7bbc618857627/Model_Card_Claude_3.pdf.

[11] Dariusz R Augustyn, Łukasz Wyciślik, and Mateusz Sojka. 2024. Tuning a kubernetes horizontal pod autoscaler for meeting performance and load demands in cloud deployments. *Applied Sciences* 14, 2 (2024), 646.

[12] Yadu Babuji, Anna Woodard, Zhuozhao Li, Daniel S Katz, Ben Clifford, Rohan Kumar, Lukasz Lacinski, Ryan Chard, Justin M Wozniak, Ian Foster, et al. 2019. Parsl: Pervasive parallel programming in python. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*. 25–36.

[13] Marta Catillo, Umberto Villano, and Massimiliano Rak. 2023. A survey on autoscaling: how to exploit cloud elasticity. *International Journal of Grid and Utility Computing* 14, 1 (2023), 37–50.

[14] Ziyi Chen, Xiaocong Yang, Jiacheng Lin, Chenkai Sun, Kevin Chang, and Jie Huang. 2024. Cascade speculative drafting for even faster llm inference. *Advances in Neural Information Processing Systems* 37 (2024), 86226–86242.

[15] Ewa Deelman, Karan Vahi, Gideon Juve, Mats Rynge, Scott Callaghan, Philip J Maechling, Rajiv Mayani, Weiwei Chen, Rafael Ferreira Da Silva, Miron Livny, et al. 2015. Pegasus, a workflow management system for science automation. *Future Generation Computer Systems* 46 (2015), 17–35.

[16] Deloitte Insights. 2025. 2025 Global semiconductor industry outlook. https://www.deloitte.com/us/en/insights/industry/technology/technology-media-telecom-outlooks/semiconductor-industry-outlook.html

[17] Qiyang Ding, Pengfei Zheng, Shreyas Kudari, Shivaram Venkataraman, and Zhao Zhang. 2023. Mirage: Towards Low-interruption Services on Batch GPU Clusters

with Reinforcement Learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–13.

[18] Altair Engineering. 2025. Grid Engine Users's Guide. https://2021.help.altair.com/2021.1/AltairGridEngine/8.7.0/UsersGuideGE.pdf

[19] Shengda Fan, Xin Cong, Yuepeng Fu, Zhong Zhang, Shuyan Zhang, Yuanwei Liu, Yesai Wu, Yankai Lin, Zhiyuan Liu, and Maosong Sun. 2024. Workflowllm: Enhancing workflow orchestration capability of large language models. *arXiv preprint arXiv:2411.05451* (2024).

[20] Yao Fu, Leyang Xue, Yeqi Huang, Andrei-Octavian Brabete, Dmitrii Ustiugov, Yuvraj Patel, and Luo Mai. 2024. {ServerlessLLM}:{Low-Latency} serverless inference for large language models. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 135–153.

[21] Xin Gao, Qizhi Pei, Zinan Tang, Yu Li, Honglin Lin, Jiang Wu, Lijun Wu, and Conghui He. 2025. A Strategic Coordination Framework of Small LLMs Matches Large LLMs in Data Synthesis. *arXiv preprint arXiv:2504.12322* (2025).

[22] Henrique Goulart, Alvaro Franco, and Odorico Mendizabal. 2023. Checkpointing techniques in distributed systems: A synopsis of diverse strategies over the last decades. In *Workshop de Testes e Tolerância a Falhas (WTF)*. SBC, 15–28.

[23] Anisha Gunjal and Greg Durrett. 2024. Molecular facts: Desiderata for decontextualization in llm fact verification. *arXiv preprint arXiv:2406.20079* (2024).

[24] Tanzima Zerin Islam, Kathryn Mohror, Saurabh Bagchi, Adam Moody, Bronis R De Supinski, and Rudolf Eigenmann. 2012. MCREngine: A scalable checkpointing system using data-aware aggregation and compression. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–11.

[25] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. 2019. Analysis of {Large-Scale}{Multi-Tenant}{GPU} clusters for {DNN} training workloads. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 947–960.

[26] Myeongjae Jeon, Shivaram Venkataraman, Junjie Qian, Amar Phanishayee, Wencong Xiao, and Fan Yang. 2018. Multi-tenant GPU clusters for deep learning workloads: Analysis and implications. *Technical report, Microsoft Research* (2018).

[27] Christoforos Kachris. 2025. A survey on hardware accelerators for large language models. *Applied Sciences* 15, 2 (2025), 586.

[28] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 611–626.

[29] Yaniv Leviathan, Matan Kalman, and Yossi Matias. 2023. Fast inference from transformers via speculative decoding. In *International Conference on Machine Learning*. PMLR, 19274–19286.

[30] Feng Liang, Zhen Zhang, Haifeng Lu, Chengming Li, Victor Leung, Yanyi Guo, and Xiping Hu. 2024. Resource allocation and workload scheduling for large-scale distributed deep learning: A survey. *arXiv preprint arXiv:2406.08115* (2024).

[31] Bin Lin, Chen Zhang, Tao Peng, Hanyu Zhao, Wencong Xiao, Minmin Sun, Anmin Liu, Zhipeng Zhang, Lanbo Li, Xiafei Qiu, et al. 2024. Infinite-llm: Efficient llm service for long context with distattention and distributed kvcache. *arXiv preprint arXiv:2401.02669* (2024).

[32] Yizhou Luo, Qiang Wang, Shaohuai Shi, Jiaxin Lai, Shuhan Qi, Jiajia Zhang, and Xuan Wang. 2024. Scheduling Deep Learning Jobs in Multi-Tenant GPU Clusters via Wise Resource Sharing. In *2024 IEEE/ACM 32nd International Symposium on Quality of Service (IWQoS)*. IEEE, 1–10.

[33] Ziming Mao, Tian Xia, Zhanghao Wu, Wei-Lin Chiang, Tyler Griggs, Romil Bhardwaj, Zongheng Yang, Scott Shenker, and Ion Stoica. 2025. Skyserve: Serving ai models across regions and clouds with spot instances. In *Proceedings of the Twentieth European Conference on Computer Systems*. 159–175.

[34] Xupeng Miao, Chunan Shi, Jiangfei Duan, Xiaoli Xi, Dahua Lin, Bin Cui, and Zhihao Jia. 2024. Spotserve: Serving generative large language models on preemptible instances. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 1112–1127.

[35] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. 2018. Ray: A distributed framework for emerging {AI} applications. In *13th USENIX symposium on operating systems design and implementation (OSDI 18)*. 561–577.

[36] Thanh Son Phung, Ben Clifford, Kyle Chard, and Douglas Thain. 2023. Maximizing data utility for hpc python workflow execution. In *Proceedings of the SC'23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis*. 637–640.

[37] Thanh Son Phung and Douglas Thain. 2024. Adaptive Task-Oriented Resource Allocation for Large Dynamic Workflows on Opportunistic Resources. In *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 300–311.

[38] Thanh Son Phung, Colin Thomas, Logan Ward, Kyle Chard, and Douglas Thain. 2024. Accelerating Function-Centric Applications by Discovering, Distributing, and Retaining Reusable Context in Workflow Systems. In *Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing*. 122–134.

[39] Thanh Son Phung, Logan Ward, Kyle Chard, and Douglas Thain. 2021. Not all tasks are created equal: Adaptive resource allocation for heterogeneous tasks in dynamic workflows. In *2021 IEEE Workshop on Workflows in Support of Large-Scale Science (WORKS)*. IEEE, 17–24.

[40] Konstantin F Pilz, James Sanders, Robi Rahman, and Lennart Heim. 2025. Trends in AI supercomputers. *arXiv preprint arXiv:2504.16026* (2025).

[41] Haoran Qiu, Weichao Mao, Chen Wang, Hubertus Franke, Alaa Youssef, Zbigniew T Kalbarczyk, Tamer Başar, and Ravishankar K Iyer. 2023. {AWARE}: Automate workload autoscaling with reinforcement learning in production cloud systems. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. 387–402.

[42] Matthew Rocklin et al. 2015. Dask: Parallel computation with blocked algorithms and task scheduling.. In *SciPy*. 126–132.

[43] Tim Shaffer and Douglas Thain. 2017. Taming metadata storms in parallel filesystems with metaFS. In *Proceedings of the 2nd Joint International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems*. 25–30.

[44] Aayush Shah and Shankar Jayaratnam. 2024. Energy Efficient Protein Language Models: Leveraging Small Language Models with LoRA for Controllable Protein Generation. *arXiv preprint arXiv:2411.05966* (2024).

[45] George Siachamis, Kyriakos Psarakis, Marios Fragkoulis, Arie Van Deursen, Paris Carbone, and Asterios Katsifodimos. 2024. Checkmate: Evaluating checkpointing protocols for streaming dataflows. In *2024 IEEE 40th international conference on data engineering (ICDE)*. IEEE, 4030–4043.

[46] Barry Sly-Delgado, Thanh Son Phung, Colin Thomas, David Simonetti, Andrew Hennessee, Ben Tovar, and Douglas Thain. 2023. Taskvine: Managing in-cluster storage for high-throughput data intensive workflows. In *Proceedings of the SC'23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis*. 1978–1988.

[47] Benjamin Spector and Chris Re. 2023. Accelerating llm inference with staged speculative decoding. *arXiv preprint arXiv:2308.04623* (2023).

[48] Ning Sun, Xingyi Cheng, Shentong Mo, Chiming Liu, Hui Li, Eric Xing, and Le Song. [n. d.]. Liteformer: Lightweight Evoformer for Protein Structure Prediction. ([n. d.]).

[49] Ruslan Svirschevski, Avner May, Zhuoming Chen, Beidi Chen, Zhihao Jia, and Max Ryabinin. 2024. Specexec: Massively parallel speculative decoding for interactive llm inference on consumer devices. *Advances in Neural Information Processing Systems* 37 (2024), 16342–16368.

[50] Gemini Team, Petko Georgiev, Ving Ian Lei, Ryan Burnell, Libin Bai, Anmol Gulati, Garrett Tanzer, Damien Vincent, Zhufeng Pan, Shibo Wang, et al. 2024. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. *arXiv preprint arXiv:2403.05530* (2024).

[51] Douglas Thain, Todd Tannenbaum, and Miron Livny. 2005. Distributed computing in practice: the Condor experience. *Concurrency and computation: practice and experience* 17, 2-4 (2005), 323–356.

[52] James Thorne, Andreas Vlachos, Christos Christodoulopoulos, and Arpit Mittal. 2018. FEVER: a Large-scale Dataset for Fact Extraction and VERification. In *NAACL-HLT*.

[53] Matteo Turilli, Vivek Balasubramanian, Andre Merzky, Ioannis Paraskevakos, and Shantenu Jha. 2019. Middleware building blocks for workflow systems. *Computing in Science & Engineering* 21, 4 (2019), 62–75.

[54] Luiz C Vieira, Morgan L Handojo, and Claus O Wilke. 2024. Scaling Down for Efficiency: Medium-Sized Transformer Models for Protein Sequence Transfer Learning. *bioRxiv* (2024), 2024–11.

[55] Logan Ward, Ganesh Sivaraman, J Gregory Pauloski, Yadu Babuji, Ryan Chard, Naveen Dandu, Paul C Redfern, Rajeev S Assary, Kyle Chard, Larry A Curtiss, et al. 2021. Colmena: Scalable machine-learning-based steering of ensemble simulations for high performance computing. In *2021 IEEE/ACM Workshop on Machine Learning in High Performance Computing Environments (MLHPC)*. IEEE, 9–20.

[56] Brent Welch, Marc Unangst, Zainul Abbasi, Garth A Gibson, Brian Mueller, Jason Small, Jim Zelenka, and Bin Zhou. 2008. Scalable Performance of the Panasas Parallel File System.. In *FAST*, Vol. 8. 1–17.

[57] Jeremy Wohlwend, Mateo Reveiz, Matt McPartlon, Axel Feldmann, Wengong Jin, and Regina Barzilay. [n. d.]. MiniFold: Simple, Fast, and Accurate Protein Structure Prediction. *Transactions on Machine Learning Research* ([n. d.]).

[58] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. 2020. {AntMan}: Dynamic scaling on {GPU} clusters for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 533–548.

[59] Xuan Zhang and Wei Gao. 2023. Towards llm-based fact verification on news claims with a hierarchical step-by-step prompting method. *arXiv preprint arXiv:2310.00305* (2023).

[60] Chao Zheng, Ben Tovar, and Douglas Thain. 2017. Deploying high throughput scientific workflows on container schedulers with makeflow and mesos. In *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE, 130–139.

[61] Ding Zou, Wei Lu, Zhibo Zhu, Xingyu Lu, Jun Zhou, Xiaojin Wang, Kangyu Liu, Kefan Wang, Renen Sun, and Haiqing Wang. 2024. OptScaler: A Collaborative Framework for Robust Autoscaling in the Cloud. *Proceedings of the VLDB Endowment* 17, 12 (2024), 4090–4103.