

# Scaling Up Throughput-oriented LLM Inference Applications on Heterogeneous Opportunistic GPU Clusters with Pervasive Context Management

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

## Abstract

The widespread growth in LLM developments increasingly demands more computational power from clusters than what they can supply. Traditional LLM applications inherently require huge static resource allocations, which force users to either wait in a long job queue and accept progress delay, or buy expensive hardware to fulfill their needs and exacerbate the demand-supply problem. However, not all LLM applications are latency-sensitive and can instead be executed in a throughput-oriented way. This throughput orientation allows a dynamic allocation that opportunistically pools available resources over time, avoiding both the long queue and expensive GPU purchases. Effectively utilizing opportunistic resources brings numerous challenges nevertheless. Our solution, pervasive context management, exploits the common computational context in LLM applications and provides mechanisms and policies that allow seamless context reuse on opportunistic resources. Our evaluation shows an LLM application with pervasive context management on opportunistic resources reduces its execution time by 98.1%.

## 1 Introduction

AI is expected to become the next revolution in human productivity, attracting both significant interests and huge investments from governments, industries, and the academia around the world[6, 12, 43]. Most of these interests and investments concentrate on Generative AI (GenAI), a type of AI that creates new data based on patterns implicitly extracted from a huge amount of existing data. Large Language Models (LLMs) are the current powerhouse of GenAI, boasting astonishing generative capabilities in diverse subjects (e.g., language, mathematics, coding) and various formats (e.g., text, image, video), on par with or even surpassing the average human intelligence[1, 5, 57]. However, LLMs need a massive amount of computational power to tune its parameters (i.e., train) and create new data (i.e., infer), which can only be offered at the moment by 1) commercial general-purpose GPUs[3, 42], which are in high demand, or 2) custom ASIC or FPGA accelerators[17, 30], which are often proprietary and extremely complex to design, both of which are limited and expensive. Thus, while GenAI researchers and practitioners are optimistic about the infinite possibilities of LLM applications, system administrators are instead cautious in promising SLOs/SLAs due to the finite amount of computational resources, pressing the need for efficient resource management.

To realize a solution for this need, one needs to be conscious of different resource requirements that different modes of LLM developments require. LLM computations revolve around either training

or inference. LLM training typically requires a static allocation of resources in a cluster for a long period of time, which is usually satisfied by a dedicated cluster or a set of machines carved out of a shared pool of resources. This siloing approach ensures that the training process is uninterrupted and minimizes the turnaround time, but introduces an unavoidable waste from idle resources during the process[15, 20]. Inference serving is latency-sensitive as it directly interacts with end users, which also requires a static allocation of resources in an indefinite amount of time and suffers from expenses caused by the idle resource problem[29, 66]. Remedies[24, 25, 28, 65] have been proposed and implemented to alleviate this problem, but the root cause of transient idle resources can never be eliminated due to the nature of resource silos and the priority and urgency of LLM training and inference serving. Moreover, the high priority of LLM jobs can worsen the oversubscription of jobs in a cluster, creating a long queue time on the scale of hours, days, or even weeks. This inadvertently pushes users with tough deadlines away from sharing resources to buying their exclusive hardware, further exacerbating the market price of GPUs and the idle resource problem.

We observe that many (LLM) inference applications are not inherently latency-sensitive but can instead be more throughput-oriented (e.g., model research and validation, prompt engineering, analytics). A throughput-oriented model of execution allows a dynamic allocation of resources in which the current load of the cluster dictates the allocation or preemption of resources: resource allocation gradually climbs as more resources become available and gradually drops to make room for a more prioritized job; the application throughput should simply adjust proportionally. This resource dynamicity thus puts throughput-oriented inference applications out of the long static-allocation queue and allows the opportunistic harvesting of unallocated resources as a result of the cluster’s resource fragmentation. Therefore, we believe the key to scale up throughput-oriented inference applications is to run them in an opportunistic way.

Of course, opportunistic resource utilization comes with a unique set of problems. Such resources are unstable: resources can be evicted at any moment, threatening ongoing progress to be uselessly discarded, and resources can come at any moment, requiring an application to adjust quickly to the newly available resources. These resources are unpredictable: they can vary day-by-day, or even hour-by-hour, and only correlate with the current load of the cluster. Opportunistic resource acquisition ensures runtime execution as long as they are held, but that doesn’t guarantee throughput: each computational unit usually brings non-trivial initialization and cleanup overheads, and these overheads, if not carefully managed,

can substantially degrade the application’s performance and the overall cluster’s efficiency as computations may repeatedly run without any throughput.

Resource heterogeneity is often common in opportunistic resources: a typical cluster has a specific spectrum of slow and fast GPUs as it evolves over a long period of time, and while a user can ask for a static allocation of homogeneous GPUs, one cannot simply pick and choose opportunistically available GPUs as they may come and go in arbitrary orders and varieties. Data movement and I/O can also be spiky as computational units often start and stop arbitrarily in response to the instability of opportunistic resources. Finally, it is unclear how a user can choose a batch size for each computational unit: a batch size too large unlocks a higher inference throughput but risks a higher chance of eviction and thus no throughput, and a batch size too small safeguards the incremental smaller throughput but wastes resources from the accompanying initialization and cleanup overheads.

Our insight into unlocking the benefits of opportunistic resource utilization and negating its drawbacks is as follows: **individual computational units that batch multiple inferences often share the same computational context (e.g., software dependencies, model weights, prompt template), and managing the reuse of this context effectively is the key to productive opportunistic resource utilization.** Such context management must be pervasive: a computational context can reside in memory, local SSDs, or GPUs of any compute node in the allocated pool of opportunistic resources. Once this context is captured and registered to be managed by a given system, it is then straightforward to derive policies and mechanisms that adjust in real-time the distributions, retentions, and reuses of contexts to the organic arrival and departure of opportunistic resources. Our evaluation shows that, compared to the baseline of running a given inference application on a dedicated high-quality GPU, a user can reduce the execution time of that application by 98.1% (from 11.4 hours to 13.1 minutes) by carefully and effectively harvesting opportunistic resources in a GPU cluster. Our evaluation also shows that an inattentive solution trying to utilize opportunistic resources incorrectly leads to a terrible degradation of performance and increases the execution time of the inference application by 245.3% (from 11.4 hours to 1.6 days).

## 2 Background

### 2.1 Throughput-oriented Applications

Throughput-oriented applications have been one of the most well-studied subjects in the traditional HPC literature[40, 46, 54, 58, 62]. Almost all resource managers[21, 32, 59, 69] support this mode of execution via the concept of job priority and mechanisms like job eviction and pausing, and many runtime systems[7, 14, 53, 64] have been developed to help formulate and scale up applications in this paradigm. While their usage covers a wide range of fields, these applications have a limited set of common denominators as follows:

**Bulk data processing demand.** Throughput-oriented applications typically tackle data-intensive problems, both in size (on the order of TBs or PBs) and in number (e.g., many small data items).

**Divisible and parallelizable computation.** Such problems are addressed by dividing the whole computational need into many

smaller units, each of which can fit on a compute node, and parallelizing on multiple compute nodes in a given resource allocation.

**Per-task latency tolerance.** Users running throughput-oriented applications don’t concern with the latencies of individual computational units (i.e., tasks), but instead on the application (i.e., job) as a whole. Thus, minor deviations in tasks’ latencies are tolerated.

**Job delay tolerance.** Users generally don’t have a hard deadline for jobs, but instead worry about the throughput rate: as long as the rate is within an acceptable range, users are satisfied.

**Inter-task independence.** Each task processes its own chunk of data and has no synchronization with other tasks. Thus, tasks are independent from one another: a failing task doesn’t crash others.

**Per-task fault tolerance.** Common policies encoded in many resource managers allow them to preempt throughput-oriented jobs for a higher-priority ones. Therefore, tasks are designed to take this practice into account and tolerate external faults.

**High batch factor.** Tasks are regularly batched with many computations to amortize the non-trivial initialization and cleanup overheads and increase their overall throughputs.

### 2.2 Current LLM Training and Inference Serving Approaches

**2.2.1 LLM Training.** Currently, there are three main approaches to parallelism in LLM training: data parallelism, pipeline parallelism, and tensor parallelism. Note that these approaches commonly work in tandem with each other to parallelize the training process.

**Data Parallelism.** This approach[28] parallelizes the training process by distributing different chunks of input data to different GPUs for gradient computation. Specifically, each GPU in a resource allocation gets its own copy of a given model’s parameters and optimizer states. Each GPU is then given a different input batch, makes a 1-forward-1-backward pass on the model with the batch to compute a local gradient, exchanges its gradient with all other GPUs in a full-mesh manner, and updates its model’s parameters. Of course, a model might be too big for a GPU’s memory, necessitating latter approaches that slice the model parameters in different ways.

**Pipeline Parallelism.** This approach[26] "horizontally" slices a given LLM into multiple partitions that correspond to the layers in the LLM architecture, and hosts each partition on a GPU. A new input batch is processed sequentially by the partition order in both forward and backward passes, where the previous GPU sends relevant information to the next GPU in line. Since this sequencing is blocking - only 1 GPU is active at a time - an input batch is instead split into many smaller mini-batches, creating a pipeline between GPUs. Upon receiving gradients of all mini-batches, the first GPU in line broadcasts them to all other GPUs and updates its model.

**Tensor Parallelism.** This approach[51] "vertically" slices a given LLM’s layer into multiple partitions, each of which is hosted on a GPU. An input data is also sliced accordingly, where each portion of the input is run in parallel through the associated parameter partition on a GPU, and the results of a layer between GPUs are synchronized at the end of the layer.

**2.2.2 Inference Serving.** LLM inference serving resembles LLM training as it only makes a forward pass of the model to generate a prediction, and thus shares the pipeline and tensor parallelism approaches[35]. A distinct characteristic of LLM inference serving is its use of the KV cache to store representations of the previously

generated tokens and allow a linear growth of computation per output token. Without the KV cache, an LLM must repeatedly tokenize all input texts per output token which results in a quadratic growth of computation with the length of the output text.

The use of the KV cache of LLM inference serving draws two observations. First, the number of cached tokens (i.e., the token context window) can be too large to fit in a GPU. For example, two recently deployed LLM models, Gemini 2.5 [22] and Llama 4 Scout[38], have respective token context windows of 1 million and 10 million. Therefore, this requires a distributed approach to store the cache in multiple GPUs, which involves cross-GPU communication and synchronizations. Second, the LLM inferencing is commonly broken down into 2 different phases: a pre-fill phase and a decode phase. This separation comes from the fact that the cost of producing the first token is significantly higher than other output tokens as the whole prompt must be tokenized and cached in the process. Therefore, these two phases are usually executed on 2 different set of GPUs, where one set pre-fills the KV cache and then sends it to the other set of GPUs to continue token generations.

**2.2.3 Resource Requirements.** As described above, LLM training and inference serving exhibit a high degree of synchronization and cross-GPU communications. LLM serving has an even stronger latency requirement as it directly faces end-users, requiring 2 different sets of GPUs where one bootstraps the KV cache to be used by the other. Resource-wise, such tight coupling mode of computation mandates LLM training and inference serving to run on a large static allocation, and node failure is the exception rather than the norm. Techniques like checkpointing help preserve the work done but introduce high I/O and additional synchronization overheads and waste even more idle resources in the process.

However, the impacts made by LLM innovations are irrefutable as mentioned in Section 1. LLM training and inference serving jobs are hence placed in the highest priority in clusters, each of which now has a few but huge jobs occupying the majority of resources. Other users wishing to use multiple GPUs in their jobs either have to wait for days or weeks until these jobs finish and quickly fill in the gap, or buy more hardware to fulfill their needs. Both options are undesirable: the former is cheaper but blocks progress, and the latter allows progress but is much more expensive.

### 3 Throughput-oriented Inference Applications

The LLM computing landscape turns out to be quite extreme as users with lower-priority projects are stuck with two rather disheartening options. This section provides the rationale and three conditions for a third option that neither requires users to wait for their turn in a limited GPU cluster nor incentivizes users to purchase their own hardware. We first identify a class of LLM inference applications that has a less stringent resource footprint than traditional large-scale LLM training and inference serving, and provides a way to seamlessly pack these applications into an already busy GPU cluster.

### 3.1 Inferencing as Throughput-oriented Applications

**3.1.1 Advances in Smaller LLMs.** Growth in LLM developments is mostly driven by the promise of unlimited use cases and supported with evidence of incremental improvements on various AI benchmarks[11, 45, 68]. However, over the last few years, LLMs have reached a critical parameter mass of  $O(1 \text{ trillion})$  due to the computational bottleneck, and many research and initiatives have been undertaken to reduce the model size[18, 27, 70]. Besides financial benefits from the reduction, these works typically trade the generality of bigger LLMs for better performance and usability in specific use cases (e.g., domain specialization, model distillation, LLM interpretability). This push for model reduction also increases the accessibility to the LLM technology: instead of requiring access to tens of thousands of GPUs and a customized software stack backed by divisions of engineers, a standalone user with only a handful number of GPUs can build upon this technology and create new applications, which in turn increases the overall rate of innovation in the field. Thus, we arrive at the first condition:

**Condition #1:** An inference application uses a family of smaller LLMs as the backbone. Such LLMs have at most a dozen billion parameters that can fit nicely in a small number of GPUs.

**3.1.2 Throughput-oriented Computational Need.** Not all inference applications have to be interactive, especially those that use LLMs as an automation tool in a data processing pipeline. This relaxed latency requirement, combined with condition #1 above, allows many inference applications to neatly distribute their delay-tolerant work on individual compute nodes and thus exhibit seven throughput-oriented characteristics as described in Section 2. This class of inference applications can thus be executed on a dynamic allocation of resources that shrinks or expands according to the current load of a given cluster. Moreover, this resource dynamicity feature uniquely allows throughput-oriented inference applications to receive resources from not only opportunistic allocations that come without any promise, but also any existing associated static allocation. Thus, we arrive at the second condition:

**Condition #2:** An inference application can satisfy a given computational need in a throughput-oriented way.

### 3.2 Opportunistic Resource Utilization

Performance studies on clusters[9, 31, 48, 60] show that the average resource utilization is never close to 100%, leaving many resources stranded at any given time. There are two main sources of stranded resources: external fragmentation and internal fragmentation. External fragmentation occurs in a cluster when a set of statically allocated jobs does not perfectly fit the cluster’s resource capacity, resulting in unallocated resources. Internal fragmentation occurs when a job does not fully utilize all resources in its allocation, resulting in idle resources. Resource managers are fully aware of both types of resource inefficiency and commonly mitigate them via two solutions: backfilling and overcommitment. While backfilling alleviates the unallocated resources by scheduling small lower-priority jobs to fill the void, overcommitment allows a cluster to pack more jobs than it usually can, hoping that jobs don’t consume all of their allocations at all times. This common awareness is the third

condition: well-maintained clusters, with varying degrees, will always have backfilling and overcommitment enabled, thus opening the door of opportunistic resource utilization to any jobs that are designed to consume them. We arrive at the final condition:

**Condition #3:** Clusters rarely reach 100% resource utilization, thus providing jobs with opportunistic resources from unallocated or unused allocation.

Once these three conditions are upheld, a throughput-oriented inference application using a family of smaller LLMs can freely utilize available opportunistic resources without waiting for its turn in the static allocation queue or purchasing expensive hardware.

## 4 Scaling Challenges on Opportunistic Resources

Utilizing opportunistic resources brings a unique set of challenges however as they come without any guarantee. This section dissects the characteristics of opportunistic resources, challenges for applications to utilize them, and potential approaches to these challenges.

**Challenge #1: Instability.** Opportunistic resources are inherently unstable as they can join and leave the resource pool at any given moment. Resource managers commonly reclaim them by evicting a running job on a compute node, and tasks running on this node are killed without any chance of cleanup. Mitigations for this problem include designing fault-tolerance tasks such that a task crashing mid-execution does not affect the overall application and can later be rescheduled on other nodes for execution. On the other hand, an application should respond quickly to the availability of new resources as those resources are at best claimed by other jobs and at worst idle in the process. Thus, an application needs to run a daemon-like process that periodically monitors the availability of the cluster and reactively submits more small backfilling jobs.

**Challenge #2: Unpredictability.** Availability of opportunistic resources is generally unpredictable as it correlates with the current load of the cluster at a given time. The larger the cluster, the more unpredictable it becomes, so job planning for this resource is virtually unreliable. This can only be alleviated by observability tools that transparently inform users of the current rate of throughput and the overall progress of the application.

**Challenge #3: No throughput guarantee.** Acquiring an opportunistic resource allocation ensures that the application and its constituent tasks are running, but it does not guarantee any throughput. This is because tasks, especially LLM inferencing, have a non-trivial initialization overhead: an LLM’s parameters must be staged to a compute node’s SSDs and/or memory before being loaded into a GPU. This overhead, combined with individual tasks’ runtime, risks tasks’ preemption before they deliver any goodput. Mitigations include amortization of this cost by locally caching models on compute nodes and tuning the inference batch size based on a specific eviction risk model.

**Challenge #4: Resource heterogeneity.** Clusters are heterogeneous as they evolve over a long period of time, thus consist of a specific mixture of slow and fast GPUs. Applications cannot ask for an allocation of fast GPUs as opportunistic resources come and go with arbitrary orders and varieties, and users instead must be aware of this inherent heterogeneity when designing applications.

**Challenge #5: Spiky data movement and I/O.** It is common for applications to rely on the existence of a shared filesystem in a cluster to stage input data to and output data from compute nodes. Challenge #1 implies that this reliance will introduce a much more erratic data movement and I/O patterns as tasks start and stop based on the cluster’s availability. A possible scenario is when a large amount of resources suddenly become available, and tasks deployed on this opportunistic allocation all try to read billions of parameters out of the shared filesystem at once, which deteriorates the shared filesystem’s overall health and hurts co-located users. Resolving this problems requires an effective management of the distribution and caching of data on compute nodes.

**Challenge #6: Unclear optimal batch size.** Finally, the inherent uncertainty in opportunistic resource utilization

makes the problem of choosing an inference batch size to maximize throughput even more difficult. It’s also not clear if a user should use one batch size for all tasks when factored in resource heterogeneity and erratic data movement. A trial-and-error approach can be used to mitigate this problem and gradually narrow down the range of an optimal batch size, but Challenge #2 complicates this due to the frequent state change of opportunistic resources.

## 5 Pervasive Context Management

This section presents our solution, pervasive context management, to the problem of scaling up throughput-oriented LLM inference applications on heterogeneous opportunistic GPU clusters. Our solution comprises of three components: 1) a software stack of dynamic workflow systems that allows users to express their computational needs via intuitive abstractions and addresses Challenges #1 and #2, (2) the pervasive context management technique that allows efficient reuse of computational context between tasks and addresses Challenges #3, #5, and #6, and (3) supporting mechanisms and policies that provide an end-to-end performant execution of these applications and address Challenge #4.

### 5.1 Parsl-TaskVine Software Stack

The first component of our solution is the software stack of two dynamic workflow systems - Parsl[7] and TaskVine[53]. 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. It excels in flexibility, portability, and ease of use, and is the default runtime of the Globus Compute ecosystem[8]. TaskVine is a low-level data-intensive workflow execution engine. TaskVine’s main strengths are its rich APIs that allow users to express low-level details about tasks and their inter-relationships, intelligent scheduling and optimization algorithms that extract values from these details, and novel data-intensive capabilities that cater to large-scale data processing applications. These strengths therefore considerably accelerate such applications to complete their executions in a near real-time manner[47, 54]. The software stack thus reflects our work on combining the best of both systems without sacrificing ease of use or performance.

Figure 1 shows how these two workflow systems work together in the big picture. On the manager node, a user expresses their application’s computational needs via generic Python functions. Once the application is run and these functions are invoked, they

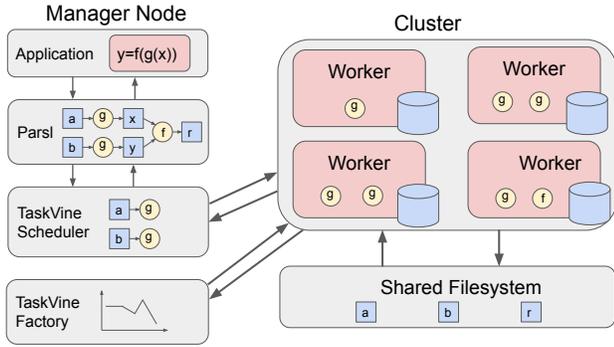


Figure 1: Software Stack of Dynamic Workflow Systems

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. 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.

Given this software stack, it is then straightforward for a user to scale up a throughput-oriented inference application. To satisfy Condition #1, 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. Condition #2 is inherently satisfied by the design of the software stack: 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. Condition #3 is reasonably assumed in the cluster as discussed.

Finally, 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 evicted task is detected, retrieved, and re-inserted into the queue of ready tasks by the scheduler. This software stack thus addresses Challenge #1 by design. Challenge #2 is satisfied by the maturity of Parsl and TaskVine as individual pieces of open-source software: each has its own suite of performance observability tools to help users visualize the throughput

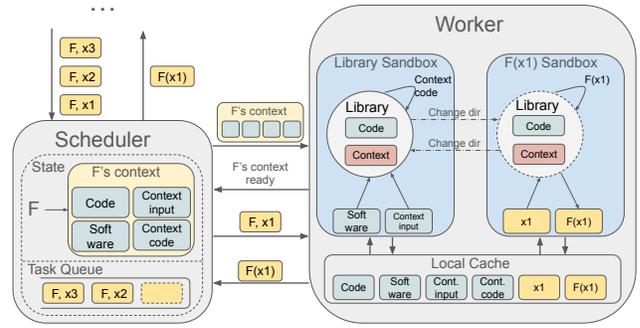


Figure 2: Context Reuse with Pervasive Context Management

rate and progress of an application. Note that Challenges #3-6 also need to be addressed to achieve a performant scaling of inference applications on opportunistic resources.

## 5.2 Pervasive Context Management

The second component of our solution is the pervasive context management technique that allows efficient reuse of a common computational context between tasks. We first point out the three following observations: (1) each task almost always needs to set up some computational context before any work is done, and thus carries some non-trivial initialization overheads (e.g., importing relevant libraries, constructing a local database, moving an LLM from SSDs or memory to GPU), (2) a large portion of this context is shareable and reusable between tasks as they usually follow the same code path and only diverge at the execution of individual inputs, and (3) since tasks are designed to be independent (see Section 2), each creates its own context for execution and tears the context down in the clean up phase, which forces each task to pay the same cost of context initialization and misses out on the opportunity of sharing and reusing such context. The core idea of pervasive context management is then to capitalize on this opportunity: a reusable context is extracted from each function, and subsequent invocations of that function can utilize this context to speed up its computations. Note that a context, in this sense, is an arbitrary computational state, which can be hosted on any worker in the pool of resources and can materialize in any format (disk, memory, GPU). This subsection focuses on the latter that concerns reusing an arbitrary context on a given worker, and delays the discussion on context distribution to Subsection 5.3.

Figure 2 gives a general example of how context reuse works in the Parsl-TaskVine stack. An application (not shown) starts up and invokes function  $F$  three times with arguments  $x_1$ ,  $x_2$ , and  $x_3$ , 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 discoverability mechanism later in Subsection 5.3) and sends this context recipe to be hosted on a given worker as part of  $F(x_1)$  execution. The worker, upon receiving the context recipe, stores all of its components in a local cache and fork-execs a special

```

1 from parl import python_app
2 def load_model(model_path):
3     ...
4     model = AutoModel.from_pretrained(model_path).to('gpu')
5     return {'model': model}
6
7 @python_app
8 def infer_model(inputs, parl_spec):
9     from parl import load_variable_from_serverless
10    model = load_variable_from_serverless('model')
11    outputs = [model.generate(input) for input in inputs]
12    return outputs
13
14 model_path = ...
15 parl_spec = {'context': [load_model, [model_path], {}]}
16 inputs = ...
17 results = infer_model(inputs, parl_spec).result()

```

**Figure 3: Code Example of an LLM Inference Application**

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.

The pervasive context management technique addresses Challenges #3, 5, and 6 as follows. For Challenge #3, the hosting of F’s context acts as a cache of the initialization overhead: instead of unnecessarily repeating the same initialization computation, the cost of putting up a common context is paid once as part of the first function invocation and amortized by subsequent invocations of the same function. Challenge #5 is addressed by the localization of individual tasks’ I/O: TaskVine stages in all inputs a function needs and stages out all outputs a function produces via workers’ local caches, thus removing the reliance on a shared filesystem. Finally, Challenge #6 is addressed by the fact that a cost of creating a common context is paid only once, no matter the batch size. Specifically, the overhead is constant per worker with pervasive context management, and a given application’s runtime now depends only on the net throughput each worker can produce.

Figure 3 shows a code example of how an inference application can utilize this technique in the Parsl-TaskVine stack. Lines 2-5 define a `load_model` function that creates an LLM context by loading its parameters from disk to GPU and returns this context via a dictionary. 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 from the context held by the library, executes the inferences, and returns the results. Lines 14-17 connect the missing

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**

pieces of the example where the context computation is defined via the `parl_spec` variable, and `infer_model` brings this context reference along with its inputs to the scheduler for execution.

## 5.3 Supporting Mechanisms and Policies

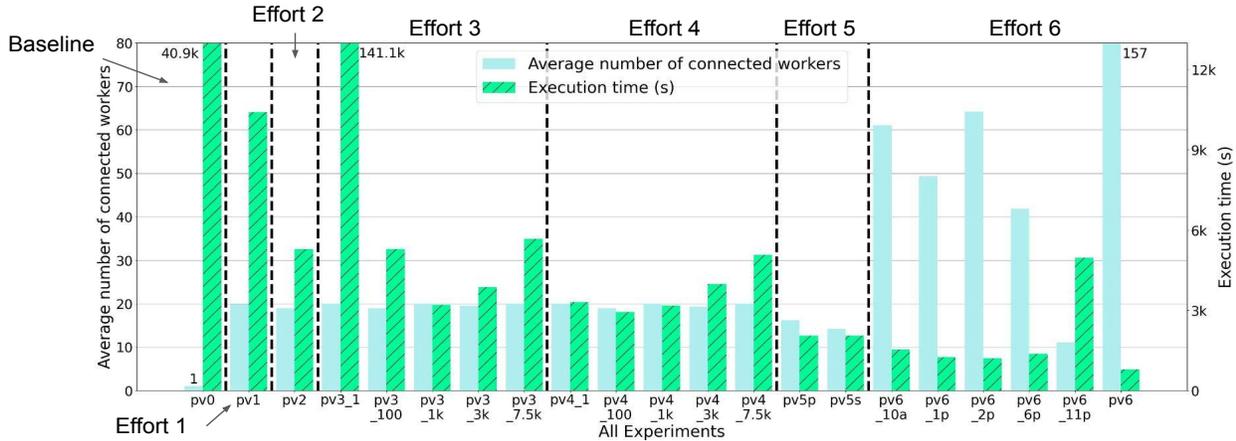
**5.3.1 Mechanisms.** We now describe the supporting mechanisms that enables pervasive context management, focusing on the discoverability and distribution of a function’s context between connected workers. Note that the context retention mechanism is described in detail in Subsection 5.2.

A computational context consists of 4 elements: the function’s code, its software dependencies, the context code, and the context inputs. To discover a function’s context, we first use the Poncho toolkit[52] to pack the function’s software dependencies into a portable format such that the scheduler can send this package to any worker. The function’s code is serialized via the `cloudpickle`[13] module and sent to the worker, which is then deserialized by the library back into a Python code object. Finally, we rely on the user to manually create the context code and its inputs and bind it to the function via Parsl primitives, and these elements also undergo the same serialization and deserialization process.

To distribute the context quickly between workers, we use the built-in peer transfer feature from TaskVine. This feature allows workers to communicate and send arbitrary data to each other under the direction of the scheduler, and each worker is capped at N transfers at a given moment. The context distribution then takes the shape of a spanning tree: the scheduler first sends the context to an arbitrary worker, and this worker sends the context to N other workers, and so on until the context is fully distributed and hosted across all workers in the system.

**5.3.2 Policies.** In the Parsl-TaskVine software stack, a TaskVine worker is the base unit of resource acquisition: each worker holds a certain amount of resources, and resources join and leave the application’s pool via the instantiation and eviction of these workers. The rate at which resources join and leave thus depends on how resource requirements of each worker are designed. On one hand, a user wanting a higher rate of resource acquisition would submit a smaller number of larger workers as each worker instantiation instantly grabs a large chunk of available resources. On the other hand, a user who wishes to alleviate the effect of worker eviction would prefer submitting many smaller workers as resources lost due to eviction are more fine-grained as opposed to losing a large chunk of resources per worker.

When inference applications are run on opportunistic resources, eviction becomes more common and thus is a larger concern. Our policy consequently is the latter approach where we design the



**Figure 4: Average Number of Connected Workers and Execution Time of All Experiments**

All 21 experiments are grouped in 6 incremental efforts from left to right. Later efforts generate better results, subject to the number of workers.

resource requirements of each worker to be as small as possible to better conserve the existing resource allocation. Thus, each worker is submitted independently in a different batch job with a minimal amount of resource requirements, and once instantiated, runs at most 1 task at any given time. This 1-to-1 ratio between tasks and workers also helps addressing Challenge #4 as workers that land on compute nodes with better hardware (e.g., GPU) run tasks faster and thus run more tasks compared to workers with slower GPUs. The former approach, while having a better rate of resource acquisition, risks binding too many tasks to the slower workers and thus exhibits the undesirable straggling effect.

## 6 Evaluation

This section demonstrates the effectiveness of pervasive context management on executing a given throughput-oriented LLM inference application on opportunistic resources. We curate this section as a series of incremental scaling efforts and interweave in-depth discussions about our reasoning, results, explanations, and potential pitfalls. We first give a thorough description of the inference application we scaled along with the general settings that apply to all experiments, and finally present our scaling efforts.

### 6.1 Optimal Prompt Search in Fact Verification

Fact verification is an active area of research given the lightning rise of online mis- and dis-information[23, 71]. Our application, Prompt for Fact (Pff), uses a given LLM as a fact verifier to check the correctness of an arbitrary claim. Since there are many LLMs to try as a fact verifier and even more abundant prompting strategies, Pff seeks to find an optimal pair of (LLM, prompt template) that yields the highest accuracy in a particular fact verification dataset. We implemented an MVP of this application that takes an LLM that satisfies Condition #1 and a prompt template and returns the aggregated accuracy. Extending this MVP to many LLMs and prompt templates is straightforward as each combination is independent from one another and thus fully parallelizable.

Specifically, we use the training data from FEVER[61] 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 SmoLM2 model with 1.7 billion parameters[2]. Our MVP takes the LLM and a prompt template, runs a full inference sweep across the dataset, and returns the fact verification accuracy. Note that the MVP also satisfies Condition #2 as we only care about the aggregated accuracy over the whole dataset.

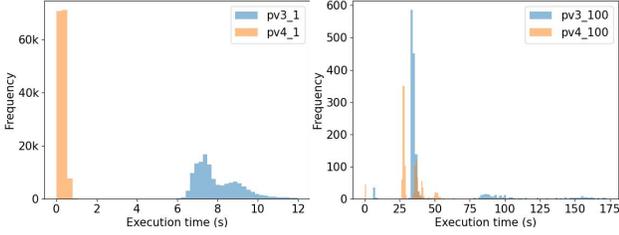
Additionally, to reflect the progression of our understanding of pervasive context management, we differentiate two types of context: partial context containing only software dependencies and model parameters, and pervasive context that includes software dependencies, inference code, context code, and context inputs.

### 6.2 General Experiment Settings

Our local cluster runs two resource managers: it uses the Altair Grid Engine (AGE)[16] as the main static batch manager, and runs HTCondor[59] as a resource backfiller on AGE’s unallocated machines. There are 567 GPUs in total in the cluster with 18 different models. Table 1 shows 8 major GPU models that account for 75% of all GPUs in the cluster. Each model is annotated with its release year and the number of GPUs in that model, showing the evolution of our local GPU cluster over time and the fulfillment of Condition #3. We run all experiments through the HTCondor resource manager. Our cluster provides access to data via the Panasas ActiveStor 16[50, 67] shared filesystem with 77 nodes and supports up to 84 Gbs/s read bandwidth and 94k read IOPS.

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. 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.

Almost all experiments start with the same resource pool configuration consisting of 20 GPUs, where half are NVIDIA A10 and



**Figure 5: Effect of Pervasive Context on Task Exec. Time**  
 Pervasive context ( $pv4_{[1, 100]}$ ) results in faster and more predictable execution times of tasks, compared to partial context ( $pv3_{[1, 100]}$ ).

Exp. ID	Mean	Std. Dev.	Min	Max
pv3_1	15.10	27.26	5.55	390.03
<b>pv4_1</b>	<b>0.32</b>	<b>0.13</b>	<b>0.0008</b>	<b>15.25</b>
pv3_100	46.78	32.88	5.93	195.89
<b>pv4_100</b>	<b>31.91</b>	<b>9.3</b>	<b>0.0008</b>	<b>79.05</b>

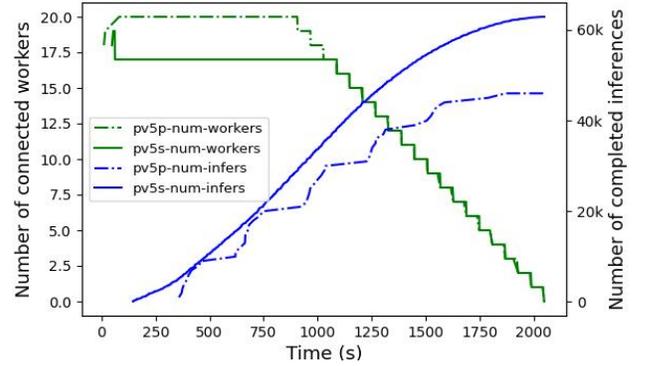
**Table 2: Statistics of Tasks’ Execution Time in 4 Experiments**  
 Pervasive Context (in bold) greatly reduces statistics of tasks’ execution time in experiments with smaller batch sizes.

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. An experiment starts when 95% of all GPUs join the pool. These constraints are removed at the end which allow the application to have access to up to 186 opportunistic GPUs.

Finally, we present the parameters of our application. Since the original FEVER dataset contains references of relevant Wikipedia pages in each claim, a data joining is required to resolve these references to the exact source text. We preprocessed this joining and store it in a local database, and all experiments pull fully resolved claims from this database to speed up the process. We introduce a small number of empty claims as the control group and bring the total number of inferences to 150k. Each inference task carries a batch size of 100 claims (thus 100 inferences) by default. This batch size is arbitrarily chosen and we provide our analysis on optimal batch sizing later. 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[4] environment, containing 308 packages and totalling 10.5 GBs of disk. The Poncho package of this environment brings the disk size down to 3.7 GBs. Note that these settings apply to all experiments unless explicitly noted otherwise.

### 6.3 Scaling Efforts

Figure 4 shows the average number of connected workers and the execution time of all experiments. Each experiment has a specific ID, and our incremental efforts are shown from left to right. We report the average number of connected workers as worker evictions and joining are common in HTCondor and happen in almost all experiments (both involuntarily and intentionally). A closer look into worker evictions and joining is provided in the later efforts. Note that from this point, we implicitly refer to Figure 4 whenever we discuss any set of experiments.



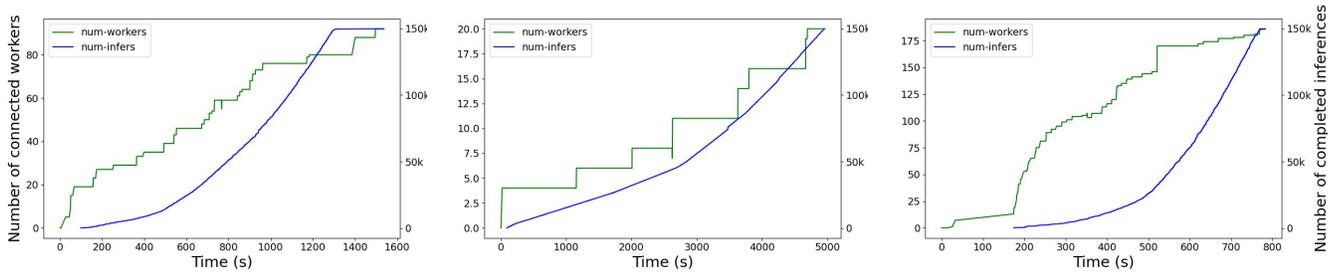
**Figure 6: Effect of Pervasive Context on Throughput**  
 Use of pervasive context ( $pv5s$ ) results in 36.7% more work done than partial context ( $pv5p$ ) in a busy cluster with diminishing availability.

**Baseline: 1 GPU [pv0].** The pv0 experiment sweeps of the fact verification dataset on 1 worker with a dedicated NVIDIA A10 GPU as our baseline, which takes 40.9k seconds from start to finish.

**Effort 1: Naive Scaling [pv1].** The pv1 experiment shows the results when we scale the application to 20 GPUs using a naive implementation of the Parsl-TaskVine stack. In this implementation, we minimize the amount of code changes compared to the baseline, and only divide the dataset into 1,500 tasks, each with 100 inferences, and run these tasks in parallel on 20 workers. This effort thus produces a disappointing speedup of 3.9 (from 40.9k to 10.4k seconds) on 20 GPUs due to several reasons. Since all inference tasks run independently and register no reusable context, the peer transfer feature is effectively disabled as there’s no registered data to be transferred between workers. All software dependencies are pulled from the shared filesystem so the I/O performance depends on its current load. Additionally, each task downloads its own copy of the model from the Internet to its local execution sandbox instead of reusing a local cache of model parameters as sandboxes are created upon execution and destroyed upon cleanup.

**Effort 2: Software and LLM as Partial Context [pv2].** Experiment pv2 shows the next effort that registers software dependencies and model parameters as partial context to the inference tasks, bringing the speedup to the factor of 7.7 (from 40.9k to 5.3k seconds). Note that this effort uses an inference batch size of 100, and our next effort searches for an optimal batch size.

**Effort 3: Partial Context Management with Batch Size Tuning [pv3\*].** Experiments  $pv3_{[1, 100, 1k, 3k, 7.5k]}$  represent our search for an optimal batch size with partial context management, where each experiment’s batch size is the suffix of its ID. Figure 4 shows a parabolic pattern of execution times as both ends show much higher values than  $pv3_{1k}$ . This pattern is due to two competing factors: resource heterogeneity and initialization overheads. On one hand, a larger batch size better amortizes the overhead of creating a model state and moving it to a GPU. Over-batching inferences in a task however risks running many inferences on slower GPUs due to resource heterogeneity, which increases the execution time of the application with  $pv3_{7.5k}$  being the most extreme case



**Figure 7: Application Resilience Against Dynamic Opportunistic Resources for pv6\_10a (left), pv6\_11p (middle), and pv6 (right)**  
Note that plots share the right y axis but have their own scales of the x axis and the left y axis. Application’s inference progress seamlessly adapts to the availability of opportunistic resources (represented via connected workers) in all cases.

of this effect. Specifically, since we have 150k inferences and 20 GPUs, a batch size of 7.5k divides all inferences equally into 20 batches where each batch is run in a GPU on a worker. pv3\_7.5k’s execution time thus equals to that of the slowest GPU, no matter how fast the other 19 GPUs are. pv3\_3k only alleviates this effect with a smaller batch size which results in a slightly better execution time. On the other hand, a smaller batch size helps spread more inferences to faster GPUs but risks paying a costly penalty of many more initialization overheads. pv3\_1, the other end of the spectrum demonstrating this detrimental effect, runs only 1 inference per model creation and loading which substantially increases the execution time to 141.1k seconds. With partial context management, pv3\_1k empirically becomes the best run with a batch size of 1,000 that is both small enough to distribute more computations to fast GPUs and large enough to amortize the overhead cost. We also observe two competing factors, resource heterogeneity and initialization overheads, as important considerations for next efforts.

**Effort 4: Pervasive Context Management with Batch Size Tuning [pv4\*].** Since resource heterogeneity is an intrinsic feature of opportunistic resources, effort 4 instead focuses on eliminating the initialization overheads with pervasive context management. Each worker now only pays a one-time cost of model creation and loading that allows overhead amortization regardless of batch sizes. Experiments pv4\_[1, 100, 1k, 3k, 7.5k] in Figure 4 show the execution times of the application with pervasive context management on respective batch sizes. While the execution times of pv4\_[1k, 3k, 7.5k] don’t considerably change as their batch sizes shield them from the overheads’ penalty (pv4\_7.5k is slightly better than pv3\_7.5k as the latter has 1 worker eviction), the performance of pv4\_[1, 100] are much better than their pv3 counterparts by 97.8% and 44.5%, respectively, and drives down the fastest execution time to 2.9k seconds with a speedup factor of 13.9. We point out two observations: 1) the optimal batch size now shifts from 1k to 100, showing that an even finer-grained batch size can benefit from pervasive context management, and 2) the performance cost of choosing the wrong batch size decreases significantly as any batch size in the range of 1 to 1,000 now results in an execution runtime increase of at most 12.3% instead of 4306%. Note that the ideal speedup factor of 20 is impossible as the pool of resources is heterogeneous. Moreover, pervasive context management enables much lower and more stable execution times of individual tasks in runs with smaller batch sizes. Figure 5 shows the histograms

of execution time of two pairs of runs, pv[3, 4]\_1 (left) and pv[3, 4]\_100 (right). On the left histogram, two runs both have 150k tasks, each with 1 inference, but pv3\_1 employs partial context management while pv4\_1 follows pervasive context management. This difference results in the stark contrast of tasks’ execution time, as most tasks in pv4\_1 reuse existing contexts on workers more effectively and cluster around the (0, 1) second range, while tasks in pv3\_1 must pay the cost of model loading and mostly spread from 6 to 12 seconds. A similar pattern shows in the right histogram with the pv[3, 4]\_100 pair as pv3\_100 has a much wider spread of execution time compared to pv4\_100. Note that we trim some values in histograms for better visualization. Table 2 instead shows the full statistics of tasks’ execution time of 4 runs, demonstrating the superiority of pervasive context management as it has lower mean, standard deviation, min, and max execution times in both pairs.

**Effort 5: Pervasive Context Management In a Busy Cluster [pv5\*].** We describe the last set of experiment on the 20-GPU setup before running the application without any restriction on the local cluster. This last set contains pv5p and pv5s which follow the partial and pervasive context management approaches and have the empirically optimal batch sizes of 1k and 100 respectively, and simulates an eviction scenario where a GPU cluster suddenly becomes busy and reclaims opportunistic resources gradually. Specifically, both experiments run without any artificial interruption for first 15 minutes, and then we drain all GPUs from the resource pool at the rate of 1 GPU/minute, prioritizing all NVIDIA A10s before NVIDIA Titan X Pascals. Figure 6 shows the number of connected workers and the amount of completed inferences over time under the two context management policies. Even though pv5s involuntarily loses 2 workers from the beginning, its throughput rate is still consistently higher than that of pv5p at any given time. We thus see another benefit of pervasive context management besides the constant cost of overhead per worker: while both experiments eventually have 20 workers and thus 20 tasks evicted, the amount of inferences evicted in pv5s is only  $20 * 100 = 2k$  while that of pv5p is  $20 * 1000 = 20k$ . This results in a striking difference of 16.9k completed inferences at the end of the experiments, and thus demonstrates the effectiveness of the pervasive context management technique.

**Effort 6: Unrestricted Scaling [pv6\*].** Experiments pv6\_[10a, 1p, 2p, 6p, 11p] represent the application running with pervasive context management and a batch size of 100 in a given day on the

local cluster without any restriction, with each experiment’s suffix denoting the time of day the application starts (from 10am to 11pm). We can see that the amount of claimed opportunistic resources on average fluctuates by a large margin in the day, going as high as 64 to as low as 11 GPUs, with later experiments showing fewer opportunistic GPUs as users tend to run more jobs overnight. Nevertheless, the application’s execution time adapts proportionally to the instability and unpredictability of the opportunistic GPUs, with pv6\_2p’s execution time going as low as 1,211 seconds. We also show that pv6, an experiment with the same settings but run on a different day when the cluster is less busy, is able to claim an average of 157 GPUs during its shortest execution time of 783 seconds. Figure 7 shows more details of 3 experiments, pv6\_10a, pv6\_11p, and pv6, plotting the number of connected workers and completed inferences over time. We can observe the different rates of resource acquisition during the application’s execution, showing the uncertainty of the opportunistic resources. However, the throughput rate of the application doesn’t lag too far behind the amount of acquired resources and adapts seamlessly to the availability of opportunistic resources in all cases. This final effort thus demonstrates the success of pervasive context management in scaling up a throughput-oriented inference application on a heterogeneous opportunistic GPU cluster.

## 7 Related Works

**LLM Inference Optimization.** Many works optimize the inference process by outputting several tokens in one forward pass based on the speculative decoding scheme[10, 34, 55, 56]. 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.[33] introduce a virtual paging mechanism that divides the dynamically-sized KV cache into blocks to remove GPU memory fragmentation, while Lin et. al. [36] 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.[19] 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. Mao et. al. [37] mix on-demand (equivalent to static resources in clusters) with spot instances (generally equivalent to opportunistic resources) and over-provision spot instances to serve latency-sensitive LLM inferences. Our work runs LLM inferences only on opportunistic resources instead. Miao et. al.[39] also exclusively serve LLM inferences on spot instances, but rely on a grace period from 30 seconds to 2 minutes to send the state of an ongoing request to other instances, while opportunistic resources in our work evict workers immediately upon reclamation.

**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[14, 63, 72]. 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[7, 41, 49] 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. Ray[41] is a Python workflow system that also relaxes the independence abstraction and offers a similar sharing abstraction via Ray Actors. Specifically, Ray Actors allow users to define an object with associated methods such that users can deploy these objects on compute nodes and remotely invoke their methods in a similar fashion to Java RMI[44]. Thus, users of the Parsl-TaskVine stack defining a computational context can equivalently do so by putting the context code in the object’s class constructor, which both holds the context on compute nodes as long as the object is alive and provides tasks access to this context via the object’s Python reference. The major difference between the two approaches lies in their abstraction as Parsl-TaskVine supports the task abstraction while Ray supports the object abstraction. In terms of scheduling, Parsl-TaskVine users only have to submit tasks and the system automatically maps tasks to available contexts in the cluster. On the other hand, Ray users must explicitly choose an object for every remote task execution. This also has an implication in reliability. When a worker holding a context is evicted, Parsl-TaskVine seamlessly requeues the task and moves it to another node and shields users from this failure, while Ray users must manually deal with an actor eviction before sending tasks for remote execution.

## 8 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 analyzes the resource requirements of traditional LLM training and inference serving applications, points out a class of LLM applications that relaxes the latency and synchronization requirements, and introduces the pervasive context management technique that allows this class of inference applications to seamlessly and performantly scale on heterogeneous opportunistic GPU clusters. Specifically, it points out three conditions that enable a throughput-oriented execution of inference applications and six challenges in effective utilization of opportunistic resources, followed by in-depth descriptions of the Parsl-TaskVine stack and pervasive context management as the solution, and a thorough demonstration of our scaling efforts, resulting in a 98.1% reduction in execution time of a given LLM inference application.

## References

- [1] 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).
- [2] Loubna Ben Allal, Anton Lozhkov, Elie Bakouch, Gabriel Martín Blázquez, Guilherme Penedo, Lewis Tunstall, Andrés Marafioti, Hynek Kydlíček, Agustín Piqueres Lajarin, Vaibhav Srivastav, et al. 2025. SmoLLM2: When Smol Goes Big—Data-Centric Training of a Small Language Model. *arXiv preprint arXiv:2502.02773* (2025).
- [3] AMD. [n. d.]. AMD CDNA 3 Architecture. <https://www.amd.com/content/dam/amd/en/documents/instinct-tech-docs/white-papers/amd-cdna-3-white-paper.pdf>
- [4] Anaconda. 2025. Anaconda Documentation. <https://www.anaconda.com/docs/main>
- [5] Anthropic. [n. d.]. The Claude 3 Model Family: Opus, Sonnet, Haiku. [https://www-cdn.anthropic.com/de8ba9b01c9ab7cbabf5c33b80b7bbc618857627/Model\\_Card\\_Claude\\_3.pdf](https://www-cdn.anthropic.com/de8ba9b01c9ab7cbabf5c33b80b7bbc618857627/Model_Card_Claude_3.pdf)
- [6] Computing Research Association. [n. d.]. 2023 Taulbee Survey All Degree Levels Exhibit Record Number of Graduates and Strong Enrollment. <https://cra.org/wp-content/uploads/2024/05/2023-CRA-Taulbee-Survey-Report.pdf>
- [7] 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.
- [8] André Bauer, Haochen Pan, Ryan Chard, Yadu Babuji, Josh Bryan, Devesh Tiwari, Ian Foster, and Kyle Chard. 2024. The globus compute dataset: An open function-as-a-service dataset from the edge to the cloud. *Future Generation Computer Systems* 153 (2024), 558–574.
- [9] Robert Birke, Lydia Y. Chen, and Evgenia Smirni. 2012. Data Centers in the Cloud: A Large Scale Performance Study. In *2012 IEEE Fifth International Conference on Cloud Computing*, 336–343. doi:10.1109/CLOUD.2012.87
- [10] 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.
- [11] Wei-Lin Chiang, Lianmin Zheng, Ying Sheng, Anastasios Nikolas Angelopoulos, Tianle Li, Dacheng Li, Banghua Zhu, Hao Zhang, Michael Jordan, Joseph E Gonzalez, et al. 2024. Chatbot arena: An open platform for evaluating llms by human preference. In *Forty-first International Conference on Machine Learning*.
- [12] CNBC. [n. d.]. Tech megacaps plan to spend more than \$300 billion in 2025 as AI race intensifies. <https://www.cnbc.com/2025/02/08/tech-megacaps-to-spend-more-than-300-billion-in-2025-to-win-in-ai.html>
- [13] Cloudpickle contributors. 2023. Cloudpickle. <https://github.com/cloudpipe/cloudpickle>
- [14] 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.
- [15] Paul Delestrac, Debjyoti Battacharjee, Simee Yang, Diksha Moolchandani, Francky Catthoor, Lionel Torres, and David Novo. 2024. Multi-level Analysis of GPU Utilization in ML Training Workloads. In *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1–6.
- [16] Altair Engineering. 2025. Grid Engine Users’s Guide. <https://2021.help.altair.com/2021.1/AltairGridEngine/8.7.0/UsersGuideGE.pdf>
- [17] Amin Firoozshahian, Joel Coburn, Roman Levenstein, Rakesh Nattoji, Ashwin Kamath, Olivia Wu, Gurdeepak Grewal, Harish Aepala, Bhasker Jakka, Bob Dreyer, et al. 2023. Mtia: First generation silicon targeting meta’s recommendation systems. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, 1–13.
- [18] Yao Fu, Hao Peng, Litu Ou, Ashish Sabharwal, and Tushar Khot. 2023. Specializing smaller language models towards multi-step reasoning. In *International Conference on Machine Learning*. PMLR, 10421–10430.
- [19] 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.
- [20] Yanjie Gao, Yichen He, Xinze Li, Bo Zhao, Haoxiang Lin, Yoyo Liang, Jing Zhong, Hongyu Zhang, Jingzhou Wang, Yonghua Zeng, et al. 2024. An empirical study on low gpu utilization of deep learning jobs. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 1–13.
- [21] Wolfgang Gentzsch. 2001. Sun grid engine: Towards creating a compute power grid. In *Proceedings First IEEE/ACM International Symposium on Cluster Computing and the Grid*. IEEE, 35–36.
- [22] Google. [n. d.]. Gemini 2.5: Our most intelligent AI model. <https://blog.google/technology/google-deepmind/gemini-model-thinking-updates-march-2025/>
- [23] Anisha Gunjal and Greg Durrett. 2024. Molecular facts: Desiderata for decontextualization in llm fact verification. *arXiv preprint arXiv:2406.20079* (2024).
- [24] Sekeba Hayashi, Ruguhi Fujimoto, and Giritaha Okamoto. 2024. Enhancing compute-optimal inference for problem-solving with optimized large language model. *Authorea Preprints* (2024).
- [25] Soka Hisaharo, Yuki Nishimura, and Aoi Takahashi. 2024. Optimizing llm inference clusters for enhanced performance and energy efficiency. *Authorea Preprints* (2024).
- [26] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, and zhaifeng Chen. 2019. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. In *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d’Alché-Buc, E. Fox, and R. Garnett (Eds.), Vol. 32. Curran Associates, Inc. [https://proceedings.neurips.cc/paper\\_files/paper/2019/file/093f65e080a295f8076b1c5722a46aa2-Paper.pdf](https://proceedings.neurips.cc/paper_files/paper/2019/file/093f65e080a295f8076b1c5722a46aa2-Paper.pdf)
- [27] Mojan Javaheripi, Sébastien Bubeck, Marah Abdin, Jyoti Aneja, Sébastien Bubeck, Caio César Teodoro Mendes, Weizhu Chen, Allie Del Giorno, Ronen Eldan, Sivakanth Gopi, et al. 2023. Phi-2: The surprising power of small language models. *Microsoft Research Blog* 1, 3 (2023), 3.
- [28] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, et al. 2024. {MegaScale}: Scaling large language model training to more than 10,000 {GPUs}. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, 745–760.
- [29] Yunho Jin, Chun-Feng Wu, David Brooks, and Gu-Yeon Wei. 2023. S3: Increasing GPU Utilization during Generative Inference for Higher Throughput. *Advances in Neural Information Processing Systems* 36 (2023), 18015–18027.
- [30] Norman P Jouppe, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. 2017. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*, 1–12.
- [31] Karthik Kambatla, Vamsee Yarlagadda, Íñigo Goiri, and Ananth Grama. 2018. UBIS: Utilization-Aware Cluster Scheduling. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 358–367. doi:10.1109/IPDPS.2018.00045
- [32] Kubernetes. [n. d.]. Kubernetes Documentation. <https://kubernetes.io/docs/home/>
- [33] 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.
- [34] Yaniv Leviathan, Matan Kalman, and Yossi Matias. 2023. Fast inference from transformers via speculative decoding. In *International Conference on Machine Learning*. PMLR, 19274–19286.
- [35] Baolin Li, Yankai Jiang, Vijay Gadepally, and Devesh Tiwari. 2024. Llm inference serving: Survey of recent advances and opportunities. *arXiv preprint arXiv:2407.12391* (2024).
- [36] 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).
- [37] 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.
- [38] Meta. [n. d.]. Llama 4: Leading Multimodal Intelligence. <https://ai.meta.com/blog/llama-4-multimodal-intelligence/>
- [39] Xupeng Miao, Chunan Shi, Jiangfei Duan, Xiaoli Xi, Dahua Lin, Bin Cui, and Zhihao Jia. 2024. Spotserve: Serving generative large language models on pre-emptible instances. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 1112–1127.
- [40] Ruben S Montero, Rafael Moreno-Vozmediano, and Ignacio M Llorente. 2011. An elasticity model for high throughput computing clusters. *J. Parallel and Distrib. Comput.* 71, 6 (2011), 750–757.
- [41] 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.
- [42] NVIDIA. [n. d.]. NVIDIA A100 Tensor Core GPU Architecture. <https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf>
- [43] U.S. Department of Energy. [n. d.]. Frontiers in Artificial Intelligence for Science, Security and Technology (FASST). <https://www.energy.gov/fasst>
- [44] Oracle. 2025. The Java Remote Method Invocation API (Java RMI)n. <https://docs.oracle.com/javase/8/docs/technotes/guides/rmi/index.html>

- [45] Long Phan, Alice Gatti, Ziwen Han, Nathaniel Li, Josephina Hu, Hugh Zhang, Chen Bo Calvin Zhang, Mohamed Shaaban, John Ling, Sean Shi, et al. 2025. Humanity's Last Exam. *arXiv preprint arXiv:2501.14249* (2025).
- [46] 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.
- [47] 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.
- [48] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. 2012. Heterogeneity and dynamics of clouds at scale: Google trace analysis. In *Proceedings of the Third ACM Symposium on Cloud Computing (San Jose, California) (SoCC '12)*. Association for Computing Machinery, New York, NY, USA, Article 7, 13 pages. doi:10.1145/2391229.2391236
- [49] Matthew Rocklin et al. 2015. Dask: Parallel computation with blocked algorithms and task scheduling. In *SciPy*. 126–132.
- [50] 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.
- [51] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053* (2019).
- [52] Barry Sly-Delgado, Nick Locascio, David Simonetti, Brett Wiseman, Ben Tovar, and Douglas Thain. 2022. Poncho: Dynamic package synthesis for distributed and serverless python applications. In *Proceedings of the 2nd Workshop on High Performance Serverless Computing*. 8–14.
- [53] 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.
- [54] Barry Sly-Delgado, Ben Tovar, Jin Zhou, and Douglas Thain. 2024. Reshaping High Energy Physics Applications for Near-Interactive Execution Using TaskVine. In *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–13.
- [55] Benjamin Spector and Chris Re. 2023. Accelerating llm inference with staged speculative decoding. *arXiv preprint arXiv:2308.04623* (2023).
- [56] Ruslan Svirshchewski, 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.
- [57] Gemini Team, Rohan Anil, Sebastian Borgeaud, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, Katie Millican, et al. 2023. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805* (2023).
- [58] George Teodoro, Tony Pan, Tahsin M Kurc, Jun Kong, Lee AD Cooper, Norbert Podhorszki, Scott Klasky, and Joel H Saltz. 2013. High-throughput analysis of large microscopy image datasets on CPU-GPU cluster platforms. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. IEEE, 103–114.
- [59] 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.
- [60] Kundjanasith Thongle, Kohei Ichikawa, Keichi Takahashi, Hajimu Iida, and Chawanat Nakasan. 2019. Improving Resource Utilization in Data Centers using an LSTM-based Prediction Model. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*. 1–8. doi:10.1109/CLUSTER.2019.8891022
- [61] James Thorne, Andreas Vlachos, Christos Christodoulopoulos, and Arpit Mittal. 2018. FEVER: a Large-scale Dataset for Fact Extraction and VERification. In *NAACL-HLT*.
- [62] Benjamin Tovar, Rafael Ferreira da Silva, Gideon Juve, Ewa Deelman, William Allcock, Douglas Thain, and Miron Livny. 2017. A job sizing strategy for high-throughput scientific workflows. *IEEE Transactions on Parallel and Distributed Systems* 29, 2 (2017), 240–253.
- [63] Matteo Turilli, Vivek Balasubramanian, Andre Merzky, Ioannis Paraskevavos, and Shantenu Jha. 2019. Middleware building blocks for workflow systems. *Computing in Science & Engineering* 21, 4 (2019), 62–75.
- [64] Matteo Turilli, Andre Merzky, Vivek Balasubramanian, and Shantenu Jha. 2018. Building blocks for workflow system middleware. In *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE, 348–349.
- [65] Yiding Wang, Kai Chen, Haisheng Tan, and Kun Guo. 2023. Tabi: An efficient multi-level inference system for large language models. In *Proceedings of the Eighteenth European Conference on Computer Systems*. 233–248.
- [66] Yingcan Wei, Matthias Langer, Fan Yu, Minseok Lee, Jie Liu, Ji Shi, and Zehuan Wang. 2022. A gpu-specialized inference parameter server for large-scale deep recommendation models. In *Proceedings of the 16th ACM Conference on Recommender Systems*. 408–419.
- [67] 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.
- [68] Colin White, Samuel Dooley, Manley Roberts, Arka Pal, Ben Feuer, Siddhartha Jain, Ravid Shwartz-Ziv, Neel Jain, Khalid Saifullah, Siddhartha Naidu, et al. 2024. Livebench: A challenging, contamination-free llm benchmark. *arXiv preprint arXiv:2406.19314* (2024).
- [69] Andy B Yoo, Morris A Jette, and Mark Grondona. 2003. Slurm: Simple linux utility for resource management. In *Workshop on job scheduling strategies for parallel processing*. Springer, 44–60.
- [70] Peiyuan Zhang, Guangtao Zeng, Tianduo Wang, and Wei Lu. 2024. Tinyllama: An open-source small language model. *arXiv preprint arXiv:2401.02385* (2024).
- [71] 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).
- [72] Chao Zheng, Ben Tovar, and Douglas Thain. 2017. Deploying high throughput scientific workflows with container schedulers with makeflow and mesos. In *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE, 130–139.