NotebookOS: A Replicated Notebook Platform for Interactive Training with On-Demand GPUs

Benjamin Carver
George Mason University
Fairfax, Virginia, USA
bcarver2@gmu.edu

Jingyuan Zhang
George Mason University
Fairfax, Virginia, USA
jzhang33@gmu.edu

Haoliang Wang
Adobe Research
San Jose, California, USA
hawang@adobe.com

Kanak Mahadik
Adobe Inc
San Jose, California, USA
mahadik@adobe.com

Yue Cheng*
University of Virginia
Charlottesville, Virginia, USA
mrz7dp@virginia.edu

Abstract

Interactive notebook programming is universal in modern ML and AI workflows, with interactive deep learning training (IDLT) emerging as a dominant use case. To ensure responsiveness, platforms like Jupyter and Colab reserve GPUs for long-running notebook sessions, despite their intermittent and sporadic GPU usage, leading to extremely low GPU utilization and prohibitively high costs. In this paper, we introduce NotebookOS, a GPU-efficient notebook platform tailored for the unique requirements of IDLT. NotebookOS employs replicated notebook kernels with Raft-synchronized replicas distributed across GPU servers. To optimize GPU utilization, NotebookOS oversubscribes server resources, leveraging high inter-arrival times in IDLT workloads, and allocates GPUs only during active cell execution. It also supports replica migration and automatic cluster scaling under high load. Altogether, this design enables interactive training with minimal delay. In evaluation on production workloads, NotebookOS saved over 1,187 GPU hours in 17.5 hours of real-world IDLT, while significantly improving interactivity.

CCS Concepts: • **Computer systems organization** → **Cloud computing**; • **Software and its engineering** → **Scheduling**; • **Computing methodologies** → **Artificial intelligence**.

Keywords: Jupyter Notebook; Interactive Deep Learning Training; GPU Scheduling; Systems for AI

ACM Reference Format:
Benjamin Carver, Jingyuan Zhang, Haoliang Wang, Kanak Mahadik, and Yue Cheng. 2026. NotebookOS: A Replicated Notebook Platform for Interactive Training with On-Demand GPUs. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '26), March 22–26, 2026, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 21 pages. https://doi.org/10.1145/3760250.3762230

*Corresponding author

1 Introduction

Interactive notebook programming is universal in modern ML (machine learning) and AI (artificial intelligence) workflows. Software such as Jupyter Notebook [58] and Google Colab [17] provides a user-friendly, interactive, web-based programming interface, and therefore, they have become the *de facto* interface for interactive data-driven programming, e.g., data analytics, data science, and AI/ML, spanning almost every science and engineering domain. Take Jupyter Notebook as an example. To date, Jupyter Notebook has been used by millions of users world wide [29] in education [51, 52], scientific research [20, 73], and collaborations [18, 59]. A series of industry initiatives [11, 75, 86], workshops [48, 49], and a vast array of resources, projects, and libraries [5] underscore the growing demand for using Notebooks in data-driven jobs and tasks. In addition, all major cloud providers, along with an increasing number of startups, now offer commercial Notebook services [2, 31, 47, 63, 67, 71, 91, 92].

Traditionally, GPU-based, batch deep learning training (BDLT) features batch-style, long-running workloads [35, 44, 66, 87, 104, 106] that require uninterrupted access to GPU resources over long periods of time, such as large-scale model training. These traditional systems focus on maximizing throughput and job-completion time (JCT), often prioritizing long-running tasks that can afford latency and resource contention. In contrast, notebook-oriented, *interactive deep learning training (IDLT)* workloads—which consist of tasks like model and program debugging, dynamic/iterative model adjustments, hyperparameter tuning—demand a different set of performance characteristics. This set includes low latency, responsiveness, high interactivity, and efficient management of many short-running tasks. The short-lived tasks that dominate IDLT workloads are executed within the context of long-lived user sessions, such as Jupyter Notebook sessions.

To guarantee interactivity, today's notebook services typically provision and reserve GPU resources during the entire lifetime of a Notebook session. Although these sessions are long-running, cell executions are fragmented: IDLT workloads exhibit intermittent, sporadic, and often transient GPU usage. Users simply spend more time developing a notebook than they do executing the notebook's tasks. As a result, notebook sessions spend a majority of their time not using the reserved GPUs, leading to *extremely low GPU utilization and prohibitively high cost.*

We build NotebookOS, a *first-of-its-kind notebook platform* designed for the unique requirements of IDLT. NotebookOS is distinct in both purpose and design from traditional *GPU cluster computing* systems [35, 43, 66, 72, 88, 106, 107]. NotebookOS prioritizes interactivity, low response time, and efficient management of numerous short-lived GPU tasks. By addressing these unique needs of users actively working in notebooks, NotebookOS realizes a novel notebook platform that uses fine-grained, flexible GPU allocation to effectively meet the demands of short, dynamic GPU tasks.

At NotebookOS's core is a collection of novel techniques and design tenets that enable the efficient support of IDLT workloads. NotebookOS uses a pluggable notebook task scheduling and placement mechanism, whose default policy is designed to maximize interactivity. Specifically, NotebookOS uses a replicated kernel design, where each notebook kernel is replicated across multiple GPU servers. Any replica can execute CPU or GPU tasks. Small kernel state is synchronized across replicas using Raft [83], while large objects are asynchronously replicated via a distributed data store. This design decouples GPU allocation from notebook sessions: GPUs are dynamically allocated and assigned to one of the replicas only during code execution, maximizing the chances of immediate training upon code submission. To further improve GPU utilization, NotebookOS oversubscribes GPU resources on each server, leveraging the high task inter-arrival time (IAT) observed in IDLT workloads. This maximizes kernel availability and minimizes wait times during user code submission.

In summary, this paper makes the following contributions:

1. Characterization of an important and under-studied class of DL workloads called IDLT.
2. Design and implementation of NotebookOS tailored for the unique requirements of notebook IDLT.
3. Comprehensive evaluation of NotebookOS using prototype and simulation on production IDLT workloads.

NotebookOS aims to pave the way for more efficient AI compute via on-demand GPUs. NotebookOS is available at: https://github.com/ds2-lab/NotebookOS.

2 Background and Motivation

This section presents an overview of relevant background and a workload analysis that motivates the need for a new platform tailored for notebook IDLT workloads.

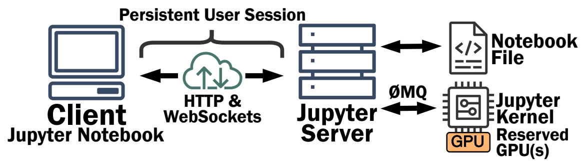


Figure 1. Client-server Jupyter Notebook architecture.

2.1 Jupyter Notebook

Jupyter Notebook [58] is an open-source, web-based interactive development environment (IDE) designed for interactive programming. Jupyter Notebook is most commonly used with Python and can be used with other languages [46], such as R [55] and Julia [54]. Jupyter Notebook has become the *de facto* development environment for data-driven programming: ML/AI, data science, and data analytics. This is due in large part to its versatility, ease-of-use, and interactivity.

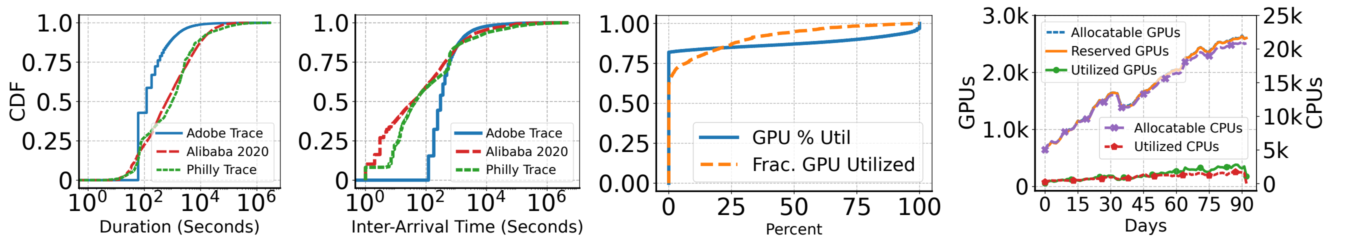
Terminology. *Notebooks* are interactive documents that combine code, text for explaining the code, and visualizations for displaying the code's inputs and outputs. These elements are contained within the *cells* of the notebook, which are the basic "units" of a notebook. When a user runs code in some cell(s), the code is sent to a separate process, called a *kernel*, which resides either on the user's local computer or on a remote server. The kernel executes the code and returns results back to the notebook document where they are displayed in the cell(s). We refer to this process as a *cell task execution.* A *notebook session* is a persistent, working instance of a Jupyter Notebook environment where the state of variables, imports, and other execution context are maintained and reused by the associated kernel. Finally, we define an *IDLT task* as a cell task execution of GPU operations—such as model training and inference—during an *interactive* workload involving debugging and testing, initial model design, and other similar activities. See Figure 1 for an illustration.

Notebook-as-a-Service In recent years, Jupyter Notebook-as-a-Service (NaaS) platforms have emerged, providing fully-managed, configurable, and scalable notebook environments. NaaS enables users to focus on coding, analysis, and research by abstracting infrastructure complexities such as server management and resource scaling. Popular Jupyter-based NaaS platforms include Google Colab [17], Amazon SageMaker Studio [8], and CoCalc [16].

2.2 Interactive Deep Learning Training

IDLT is a popular type of applications performed using Jupyter Notebook. Examples of IDLT include interactive model debugging, exploratory data analysis, hyperparameter tuning, and fast iteration of ideas, among others. Notebook-based IDLT is typically deployed and executed on large-scale GPU clusters [8, 9, 17, 71] and demands high interactivity/low response time for optimal performance.

IDLT workloads exhibit intermittent and highly variable GPU usage patterns. In order to guarantee high interactivity, modern NaaS platforms typically provision and reserve



(a) Task duration CDF. (b) Inter-arrival time CDF. (c) GPU util. CDF (AdobeTrace). (d) GPU & CPU usage (AdobeTrace).

Figure 2. Workload characteristics of three representative GPU cluster traces.

GPU resources for the entire lifetime of the notebook session (Figure 1). As a result, notebook sessions tend to spend a majority of their lifetime *not* using their reserved GPU resources: while users are debugging their code or performing other similar tasks, the GPUs are idle, and thus resources are wasted. This pattern ultimately results in extremely low GPU utilization and prohibitively high monetary costs.

2.3 Real-World IDLT Workload Analysis

Real-world IDLT workloads exhibit distinct behaviors compared to traditional long-running training jobs. In order to better understand these workloads and their requirements, we present a workload analysis of the production notebook IDLT traces collected from an Adobe internal research cluster [1]. Adobe research cluster hosts containerized notebook sessions within AWS EC2 GPU virtual machine (VM) instances. Adobe research cluster also manages the GPU resource allocations for the notebook sessions. To deploy a notebook container, users specify how many GPUs should be allocated. Once launched, GPUs are bound to the notebook container, and the user can interact with the back-end notebook kernel (i.e., an IPython process) via a web-based Jupyter Notebook interface.

The IDLT traces (AdobeTrace) include 545,467 individual training events. We focus on a representative subset spanning June 1–August 31. The sample granularity of this trace is 15 seconds.

For context, we compare AdobeTrace with two other popular, publicly available GPU cluster traces. First, we use the Alibaba GPU Cluster'20 trace [104] (AlibabaTrace), which captures both training and inference jobs running state-of-the-art ML algorithms from July to August 2020. Data were collected from a large production cluster with 6,500 GPUs across 1,800 servers. We also include the Philly GPU trace [44] (PhillyTrace), a 6.6GB representative subset of first-party deep learning training (DLT) workloads from Microsoft's internal Philly clusters. PhillyTrace consists of 117,325 jobs collected from a cluster of 2,490 GPUs distributed across 552 servers between August 7 – December 22, 2017.

We chose these two traces for comparison because they represent typical DLT workloads, which are characterized by long-running training tasks that are scheduled by a batch scheduler and span several hours. As we show in our analysis, the workload captured by AdobeTrace differs considerably from that of PhillyTrace and AlibabaTrace.

2.3.1 Training Task Duration.

Figure 2(a) displays a cumulative distribution function (CDF) of the training task durations, in seconds, of DLT tasks submitted by users to the Adobe research cluster. The data used to plot the CDF were taken from the June, July, and August 2021 trace dataset. 50% of user-submitted tasks are 2 minutes or less, while 75% are 5 minutes or less. 90% of training tasks are 17 minutes or less, 95% are 36 minutes or less, and the duration of the 99^{th} percentile training task is 182 minutes (i.e., $3.0\bar{3}$ hours). Based on this, it is clear that an overwhelming majority of training tasks are short-lived.

The AdobeTrace workload consists of a significantly high fraction of *short* training tasks compared to both PhillyTrace and AlibabaTrace. Specifically, the 50^{th} percentile of task durations is 120 seconds (2 minutes), 621 seconds (10.35 minutes), and 957 seconds (15.95 minutes) for AdobeTrace, PhillyTrace, and AlibabaTrace, respectively. This observation highlights the difference between traditional BDLT workloads (i.e., PhillyTrace and AlibabaTrace) and notebook IDLT workloads (i.e., AdobeTrace). Note that a meaningful comparison of task durations less than 15 seconds is impossible due to the 15-second granularity of AdobeTrace.

Observation 1: *IDLT workloads contain a large percentage of very short tasks—with 75% of tasks completing in 5 minutes or less—significantly shorter than traditional BDLT workloads.*

2.3.2 Training Task Inter-Arrival Time (IAT).

Figure 2(b) shows a CDF plot of the IATs of training tasks from the Adobe research cluster trace. For all three traces, the IATs were measured within each user session *independently*, rather than the cluster-wide IATs of tasks submitted by any active user session, to ensure a fair comparison. From this graph, we can see that the 50^{th} percentile of task IATs is 300 seconds (5 minutes), 44 seconds, and 38 seconds for AdobeTrace, PhillyTrace, and AlibabaTrace, respectively.

The observed AdobeTrace IATs align with the nature of IDLT workloads, where users *intermittently* make small, debugging-like changes to their models or Python code before submitting a *short* cell task to test the changes. In particular, AdobeTrace exhibits significantly longer IATs for more than 50% of tasks compared to PhillyTrace and AlibabaTrace. This is likely due to AdobeTrace users engaging in more extensive interactive cycles, iteratively testing and refining their code before each submission, rather than executing fully tested, production-ready DLT tasks scheduled by a batch GPU scheduler.

Observation 2: *IDLT tasks are submitted less frequently, with 75% having an IAT of at most 480 seconds (8 minutes), as users do not submit concurrent tasks and often make iterative modifications and tests after a task completes.*

2.3.3 Cluster GPU Utilization.

Figure 2(c) plots two different but related data series whose units are a percentage. The first series, shown as the solid blue line, is a CDF of cluster GPU utilization, measured every 15 seconds across all reserved GPUs throughout the trace's duration. The second, shown as a dashed orange line, is a CDF of the percentage of each session's lifetime during which allocated GPUs were actively utilized: a 50% value means GPUs were actively utilized for half of the time that their sessions were active.

One key observation from Figure 2(c) is that the reserved GPU resources were idle over 81% of the time. Moreover, nearly 70% of GPUs were *completely idle* throughout the entire lifetime of the notebook session to which they were assigned. (The lifetime of the notebook session begins when the associated container is provisioned and ends when that container is terminated.) Between 74% and 75% of user sessions actively used their allocated GPUs at most 5% of the time, and 90% of sessions only used their allocated GPU resources at most 31.13% of the time.

Figure 2(d) plots the number of GPUs (and CPUs) that are actively utilized compared to the number of reserved GPUs (CPUs). There is a significant gap between the number of utilized GPUs (CPUs) and the number of reserved GPUs (CPUs). By the conclusion of the 3-month period, only about 15% of all reserved GPUs are actively utilized.

These results highlight the urgent demand for a fundamentally new approach to notebook IDLT resource management to address the high degree of wasted resources.

Observation 3: *Notebook users often underutilize their allocated GPU resources during IDLT: reserved GPU resources were idle over 81% of the time.*

2.4 Insight and Challenges

Key Insight. IDLT workloads fundamentally differ from traditional BDLT workloads. This difference arises in task duration, submission frequency (i.e., IATs), and the need for user feedback or user input. During conventional BDLT workloads, ML models are trained over long periods of time—many hours or even days–with minimal intervention, except in case of failures [102, 104]. In contrast, a key insight is that IDLT workloads are characterized by intermittent and low GPU usage patterns. This is because, a user's Jupyter Notebook session has GPUs bound to it from the beginning, even though the user may spend a lot of time doing work (coding, debugging, etc) that does not require a GPU and only occasionally executing training tasks that require a GPU.

Challenges. This workload analysis highlights key challenges that must be addressed to design an efficient system for supporting notebook-based IDLT workloads.

• **C1: Resource Utilization.** Modern notebook platforms typically reserve GPUs, delegating their management to users [17, 53]. Users must manually request GPUs, CPUs, and host memory, as well as start and shut down their notebook sessions. To avoid the hassle of termination, many leave notebooks idling until the provider reclaims resources [50]. This behavior leads to extremely low GPU utilization, as discussed in §2.3.

• **C2: Interactivity.** Jupyter Notebook offers a user-friendly, interactive interface for development, execution, and debugging within a web browser. Some NaaS providers, like Azure Machine Learning [9], link the front-end notebook interface to a backend batch scheduler. However, this setup introduces high (container startup and batch queueing) delays, degrading interactivity and user experience [74, 85].

• **C3: Resource Elasticity.** Low resource utilization stems from static GPU over-provisioning and a lack of elastic resource management in NaaS platforms. Adobe research cluster, operating roughly 12k V100 GPUs across 3k p3.16xlarge EC2 instances, incurs roughly $18.3 million in monthly costs even with long-term reservation discounts. This high TCO (total cost of ownership), combined with the extremely low resource utilization, highlights the urgent need for a GPU-efficient notebook platform that can flexibly adapt based on real-time workload requirements.

3 NotebookOS Design

Workload Requirements. We present NotebookOS, a GPU-efficient notebook platform to address the challenges outlined in §2. Traditional GPU cluster schedulers are designed with long-running GPU-intensive tasks in mind. They often prioritize job completion time (JCT) [35, 87, 106], throughput [72], and fairness [66, 72] over responsiveness and interactivity in that they allocate GPUs in a way that balances the competing demands of long-running jobs. Therefore, they are poorly-suited for notebook IDLT. Whereas slight delays may have been acceptable before, even short delays can severely impact the user experience when performing highly interactive tasks. *In NotebookOS, the primary goal is to provide users with fast GPU access for interactive sessions. This demands flexible and low-latency resource allocation.*

Another requirement is *user sessions*, which traditional GPU schedulers overlook. They treat each task as an independent request competing for resources. This is a poor match for notebook IDLT where users typically work within a long-running, *stateful* notebook session, incrementally modifying the Python code, and intermittently submitting GPU tasks. NotebookOS addresses this gap by introducing mechanisms to efficiently schedule short GPU tasks within an ongoing session, enabling smarter resource allocation based on real-time user needs rather than isolated batch jobs.

Design Choices. NotebookOS introduces several novel techniques and design decisions that harmonize between long-running, stateful notebook sessions and elastic GPU allocations for short-lived notebook tasks, including:

• **Distributed Notebook Kernels:** NotebookOS adopts a novel distributed notebook design, where each logical Jupyter kernel is mapped to a NotebookOS *distributed* kernel, each of which consists of *N* kernel replicas scheduled across NotebookOS's GPU server cluster. This distributed kernel is designed to oversubscribe the under-utilized GPU resources. Its benefit is twofold. (1) Replicated kernels increase the likelihood that at least one replica will have immediate access to GPU resources upon code submission (**C2**). (2) Significantly improved resource utilization via GPU oversubscription (**C1**).

• **State Machine Replication:** NotebookOS's kernel replicas use the Raft consensus protocol [83] for leader elections and State Machine Replication (SMR). SMR enables efficient, transparent CPU state synchronization between the kernel replicas that occurs *off the critical path*. This design is motivated by the observation that interactive Python applications frequently use global variables to store intermediate state. These variables can be synchronized using SMR to ensure state changes are seen by all kernel replicas (**C2**).

• **Dynamic GPU Binding:** NotebookOS does not exclusively commit GPU resources to notebook kernel replica containers long-term. Instead, GPUs are exclusively allocated to kernel replica containers only while user-submitted cell execution tasks are actively-running. Once a task completes, the GPUs are released, enabling them to be allocated to another co-located kernel replica. This approach allows NotebookOS to adapt to fluctuating resource demands, allocating GPUs based on a session's *current* needs rather than its peak requirements. Users performing smaller-scale, less demanding tasks can request fewer GPUs, improving fine-grained resource efficiency (**C1 and C3**).

• **Transparent GPU State Checkpointing:** Instead of performing SMR, NotebookOS checkpoints large GPU state objects (model parameters and training datasets) asynchronously and separately to a distributed storage system, anticipating that the destination replica may become active (**C2**).

3.1 NotebookOS Overview

Figure 3 depicts the architecture of NotebookOS. NotebookOS consists of five core components, described below.

NotebookOS Clients. Like traditional Jupyter environments, NotebookOS clients interface with NotebookOS by sending notebook operations via HTTP or WebSocket messages to NotebookOS's Jupyter Server (Step ❶ of Figure 3).

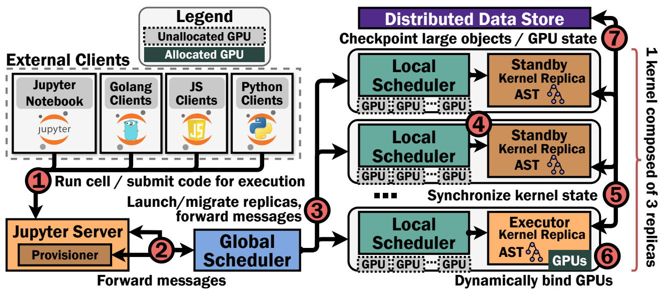


Figure 3. Architecture overview of NotebookOS.

Jupyter Server. The Jupyter Server provides the core services, serving APIs and REST endpoints used by Jupyter web apps like Jupyter Notebook [58] and JupyterLab [57].

Global Scheduler. The Global Scheduler is responsible for creating distributed kernels and initiating the provisioning of their kernel replicas (an operation that is delegated to another component of NotebookOS, the Local Scheduler). It performs a majority of the book-keeping required by NotebookOS, including managing the allocation of compute resources (i.e., CPUs, host memory, GPUs) to Distributed Kernel replicas, handling failures, kernel replica migration, and auto-scaling.

Additionally, the Global Scheduler is responsible for routing messages from Jupyter clients to the appropriate Distributed Kernel replicas. Each Jupyter message, sent from a client and forwarded by the Jupyter Server (Step ❷), contains information such as the unique identifier of the target kernel. The Global Scheduler inspects this information and then routes the message to the replicas of the target kernel. Note that messages are first forwarded to the target kernel replica's Local Scheduler (Step ❸).

Local Scheduler. NotebookOS deploys a Local Scheduler on each GPU server. It forwards messages from the Global Scheduler to the target kernel replica running on the Local Scheduler's server. It is also responsible for provisioning and managing the containers in which the kernel replicas run. It also ensures proper cleanup upon kernel termination. Upon receiving a message from the Global Scheduler, the Local Scheduler routes it to the target kernel replica (Step ❹).

Distributed Kernel. NotebookOS's Distributed Kernel consists of three replicas, which use the Raft SMR protocol to replicate the CPU-memory state of the IPython process (Step ❺ and §3.2.4). NotebookOS designs a lightweight executor election protocol to elect a proper replica with sufficient GPU resources for running GPU tasks (Step ❻ and §3.2.2). A replication factor of 5 incurs substantially higher memory, storage, and network cost without delivering significant performance benefit, while a replication factor of 2 is unsupported by the Raft protocol.

Distributed Data Store. NotebookOS offloads large object (e.g., model parameters) storage and replication to a pluggable Distributed Data Store (Step ❼). These objects are asynchronously replicated when NotebookOS detects GPU overload and switches to a new executor replica or migrates

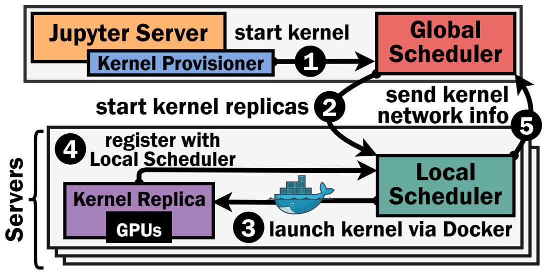


Figure 4. Process of creating a new kernel within NotebookOS's Docker Compose and Docker Swarm modes.

the current one to a different server (§3.2.4). NotebookOS supports Redis [89], AWS S3 [7], and HDFS [13, 76, 95].

3.2 Distributed Notebook Kernels

3.2.1 Distributed Kernel Creation. As described in §3.1, NotebookOS provides a customized Kernel Provisioner to integrate NotebookOS directly with Jupyter. When a user creates a new Distributed Kernel, the Jupyter Server will create a new instance of NotebookOS's Kernel Provisioner, which issues a `StartKernel` RPC request to the Global Scheduler (see Step ❶ in Figure 4).

Upon receiving the `StartKernel` RPC request, the Global Scheduler identifies three candidate GPU servers to host the replicas of the new kernel and then issues a `StartKernelReplica` RPC request to the Local Scheduler running on each of the candidate servers (Step ❷). The Global Scheduler eventually receives connection information about the new cluster of kernel replicas from the associated Local Schedulers and returns this information back to the Jupyter Server, which indicates that the new kernel was successfully created.

The Global Scheduler begins the process of identifying viable candidate GPU servers by examining the *resource request* argument of the `StartKernelReplica` RPC. The *resource request*, configured by the user, specifies the required resources for its notebook IDLT tasks, including CPUs (in millicpus, where 1 millicpu is equal to $1/1000^{th}$ of a vCPU), memory (in megabytes), GPUs, and VRAM (in gigabytes).

The Global Scheduler iterates over the servers in the cluster, checking if each server has sufficient capacity to serve a replica of the kernel. If a server meets the requirements, this server becomes a candidate. When more than three viable candidates ($N > 3$) are found, a pluggable policy is used to select the target hosts. By default, the *least-loaded* hosts—those with the fewest actively used GPUs—are chosen.

Note that, when scheduling a new Distributed Kernel, resources are not *exclusively committed* to its replicas. Instead, the kernel replicas "subscribe" to the requested resources (Figure 3). Each server in the NotebookOS cluster maintains a *subscription ratio* with a configurable high watermark that prevents excessive over-subscription. This mechanism is designed to minimize resource contention among replicas of different Distributed Kernels scheduled on the same server.

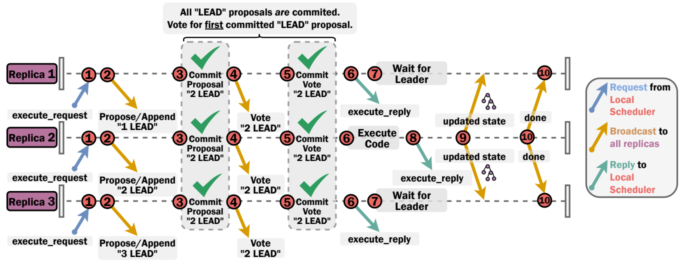


Figure 5. NotebookOS's executor election protocol.

Upon receiving a `StartKernelReplica` RPC request, the Local Scheduler will provision a new container for the kernel replica (Step ❸), In Step ❹, the provisioned replica begins running by initiating a registration procedure with its Local Scheduler. During this procedure, the Local Scheduler informs the Global Scheduler that the replica has been created successfully. Finally, in Step ❺, the Local Scheduler concludes this process by returning the replica's connection information to the Global Scheduler.

After registering with their Local Schedulers, the three kernel replica containers establish peer-to-peer (P2P) connections with one another to run the Raft protocol for transparent kernel SMR. Once the Raft cluster is established, the kernels notify their Local Schedulers, which notify the Global Scheduler. The Global Scheduler then returns from its `StartKernel` RPC handler, and the kernel is officially created.

If the Global Scheduler cannot identify three candidate servers to host the replicas of the new Distributed Kernel, then it will invoke a pluggable handler that is set based upon the configured scheduling policy. This invocation will initiate a scale-out operation to provision however many additional servers are required, as detailed in §3.4.2.

3.2.2 Distributed Kernel Raft Protocol. The three replicas of a NotebookOS Distributed Kernel connect to one another directly, forming a peer-to-peer (P2P) network. The replicas use Raft [83] to perform SMR. The use of Raft is motivated by the need for high-availability and durable sessions, even for CPU-only notebooks: existing platforms like Google Colab reclaim idle sessions [32], resulting in the loss of computed state. While recent progress on GPU state checkpoint/restore offers efficient mechanisms for GPU state snapshotting [68, 103], coarse-grained CPU state checkpoint/restore [19] is often unnecessarily heavyweight. NotebookOS simplifies fault tolerance, replica migration, failover, and the prevention of duplicate cell execution by leveraging the strong consistency guarantees provided by Raft. To this end, NotebookOS adopts a hybrid approach: it performs online replication for small CPU state and uses a distributed data store for asynchronous persistence of large objects.

Kernel replicas perform an *executor replica election protocol* to designate an "executor replica" each time a user executes a notebook cell. This protocol is designed to be efficient and fault tolerant: progress occurs even when messages between replicas—or from each replica's respective Local Scheduler—are dropped or delayed. The executor replica is responsible for executing user-submitted code while the other replicas remain idle until the next cell execution request.

Figure 5 provides an overview of the executor election protocol. When a user submits a cell request, the client sends an `execute_request` message to the Jupyter Server. This message, which encodes the target kernel's unique ID, is then forwarded to the Global Scheduler. The Global Scheduler then broadcasts a copy of this message to each Local Scheduler managing a server where a kernel replica container is running (see Step ❶ in Figure 5). When the Global Scheduler has sufficient resource information, it directly selects the executor replica for a given kernel and bypasses the Raft-based `LEAD`/`YIELD` election phase entirely. In this case, it will convert the `execute_request` message into a `yield_request`, signaling to the recipient kernel replicas that they should not attempt to participate in the election process and instead defer execution to the designated replica. This message conversion typically occurs when the Global or Local Scheduler determines that a particular server lacks the necessary resources for that server's kernel replica to execute code.

Upon receiving an `execute` or `yield` message, a kernel immediately appends a `LEAD` or `YIELD` proposal to its Raft log (Step ❷). In Step ❸, the kernel replicas wait for all Step ❷'s proposals to be committed by Raft to the Raft log. Proposal commitment is handled by the Raft protocol. Kernel replicas will simply take note of any `YIELD` proposals that are committed, as the election will "fail" if all kernel replicas propose `YIELD`. This case is described in §3.2.3.

Once the first `LEAD` proposal is committed by Raft—for example, in Figure 5, after Step ❷, Raft commits "`2 LEAD`" before "`1 LEAD`" and "`3 LEAD`" (Step ❸)—the kernel replicas each vote for `Replica 2` by appending a `VOTE` proposal to the log (Step ❹). The replicas encode the ID of the replica (`Replica 2`) that proposed the `LEAD` proposal in the `VOTE` proposal. This ID is used by the replicas in Step ❺ to determine if they won the election. The kernel replica who won the executor election becomes the *executor replica* and proceeds to execute the user-submitted code, while the other replicas become the *standby replicas* and remain idle (Step ❻).

Once the executor replica finishes executing the cell task, it commits a notification to the Raft log to inform its peers that the execution has finished (Step ❼). This notification is thereby received in Step ❽. In Step ❾, all replicas send a Jupyter `execute_reply` message to their Local Scheduler, which forwards the message to the Global Scheduler. These messages are aggregated and merged together by the Global Scheduler before being forwarded back to the Jupyter Server and subsequently the user's client.

After responding to the user's code submission, the executor replica begins replicating any updated notebook state to the standby replicas. Small state objects are directly replicated using NotebookOS's Raft SMR protocol. Large objects (model parameters copied from GPU VRAM and training datasets) are asynchronously written to the Distributed Data Store. This process occurs entirely outside the user request's critical path, avoiding any impact on user experience (§3.2.4).

3.2.3 Handling Failed Executor Elections.

In the worst case, if all kernel replicas `YIELD`, the Global Scheduler initiates a *migration* of one of the kernel replicas to a server with sufficient resources. Specifically, the Global Scheduler selects a particular kernel replica for migration and instructs that replica to persist any important state to the Distributed Data Store. The kernel replica notifies the Global Scheduler once it has done so and is ready to migrate.

Meanwhile, the Global Scheduler selects a target server as the destination of the replica's migration. The selection criteria depends on the cluster's configured scheduling policy. In general, the target server must have sufficient idle resources to immediately and exclusively bind the required GPUs to the migrated kernel replica. If a suitable server is found, the Global Scheduler sends a `StartKernelReplica` RPC to the Local Scheduler on that server, instructing it to provision a new kernel replica container. If no viable servers are available, the migration is enqueued and periodically retried, several times if necessary, before ultimately being aborted if unsuccessful. In case of an aborted migration, an `execute_reply` message with an error is returned to the client.

Once the new kernel replica has started and has read the persisted state from remote storage, the Global Scheduler terminates the original kernel replica before instructing the remaining replicas to reconfigure their Raft cluster to replace the terminated kernel replica with the newly-created replica. Next, the new replica joins its Raft cluster and begins replaying its Raft log synchronize its state with its peers, after which the Raft cluster becomes operational again. Finally, the Global Scheduler resubmits the execution request to the migrated kernel replica, ensuring that it executes the user-submitted code while the other replicas yield.

Pre-warmed Container Pool. To reduce migration overhead, the Global Scheduler maintains a small pool of *pre-warmed containers* [28, 94], managed by a component called the *Container Prewarmer*. The Container Prewarmer uses a pluggable policy for provisioning an initial pool of warm containers, and another pluggable policy for maintaining the capacity of this pool of warm containers. By default, the Container Prewarmer ensures that each server has a specified, minimum number of pre-warmed containers available.

During kernel replica migrations, the Global Scheduler will query the Container Prewarmer to see if the selected target host has any pre-warmed containers available. If so, then a pre-warmed container will be used. These containers contain a pre-initialized Python runtime with commonly used dependencies, thereby eliminating the on-demand container provisioning overhead. Pre-warming containers with varied runtime dependencies is beyond the scope of this work;

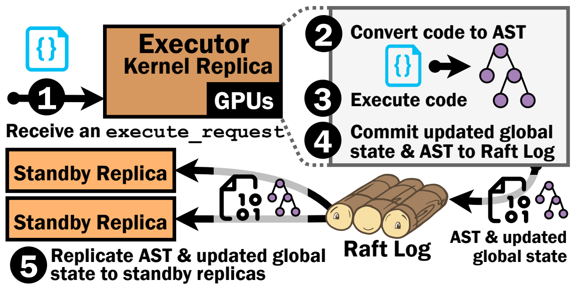

Executor Kernel Replica
GPUs
1
Receive an execute_request
2 Convert code to AST
3 Execute code
4 Commit updated global state & AST to Raft Log
Standby Replica
Standby Replica
Raft Log
AST & updated global state
5 Replicate AST & updated global state to standby replicas

Figure 6. Overview of the Raft-based state synchronization protocol used by NotebookOS's Distributed Kernel.

however, NotebookOS could leverage existing checkpoint/restore [81] or fork-based [100] solutions for this purpose.

3.2.4 Kernel State Replication. NotebookOS's Distributed Kernel requires an efficient mechanism to maintain consistency across the kernel replicas. To achieve this, the kernels leverages Python AST-based code analysis [42] to identify runtime state that needs to be replicated and synchronized via the Raft SMR. This procedure can detect and replicate Python-level state as well as state declared in native (C/C++) code, as this state is referenced in the kernel namespace (i.e., the set of user-defined variables stored within the memory of a kernel process). State of external processes or `libC` cannot be synchronized under the current implementation and is left as future work. An overview of this protocol is shown in Figure 6. As shown in Steps ❶ and ❷, code submitted for execution is first transformed into an AST by the executor replica before being executed in Step ❸. The executor replica then analyzes the Python AST to identify variables that must be synchronized with the executor's peers, such as global variables storing intermediate state that are likely to be referenced later when executing subsequent notebook cells. After the AST analysis, the executor replica proposes the AST and any relevant state to the Raft log in Step ❹. The Raft consensus protocol replicates the update to the distributed log across all replicas. Upon commitment of the newly appended log entry in Step ❺, the AST is applied by the standby replicas, and any changes to standby replicas' global state are applied to the replica-local variables.

Handling Large Objects. During state replication, the executor replica analyzes variable types from AST inspection. For large variables like models and datasets (hundreds of MBs to GBs), it avoids direct Raft log replication. Instead, it appends a *pointer* to the log while storing the actual data in the Distributed Data Store (see Step ❼ in Figure 3). NotebookOS supports AWS S3 [7], HDFS [95], and Redis [89]. Pointers in the Raft log encode data retrieval, while kernel replicas handle persistence and retrieval transparently. If a user submits an `execute_request` during state replication, it is enqueued until replication completes. Large object replication occurs asynchronously between kernel replicas, but high task IAT in IDLT workloads hides this latency from users. NotebookOS also employs a simple node-level cache to limit storage and memory costs.

3.2.5 Handling Failures. Each Distributed Kernel cluster is able to tolerate a *fail-stop* [93] failure of a single replica, as the Raft cluster of Distributed Kernel has 3 servers. If two or more replicas of a kernel were to fail, then this failure would be detected by the Global and Local Schedulers. NotebookOS's Distributed Kernel uses *heartbeat messages* to ensure that all components are active. If a heartbeat or another message, such as an `execute_request`, times out, the Global Scheduler deems that a kernel has failed. In this case, the kernel's replicas can be terminated and recreated, and the replica's can restore all state from remote storage.

3.3 GPU Management

NotebookOS performs dynamic GPU binding right before a replica begins executing user-submitted code so as to optimize resource management by enabling fine-grained, per-training-task allocations (see step 6 in Figure 3). NotebookOS's approach to dynamic GPU sharing is also designed to be *pluggable*, allowing integration with alternative approaches such as GaiaGPU [34]. We implement a simple approach in NotebookOS's prototype, with the intent of using a full-featured solution in production.

NotebookOS binds all GPUs available on a server to all its hosted kernel replica containers. Each time a notebook cell is submitted for execution by a client, the Global Scheduler embeds the device IDs of the GPUs allocated to the target kernel replica within the request metadata. NotebookOS automatically loads model parameters from the host's main memory onto all allocated GPUs using the `PyTorch` API on the critical path of execution requests. This process typically only takes up to a couple hundred milliseconds, so it does not impact performance or interactivity severely. When the executor replica finishes executing the user-submitted code, NotebookOS automatically copies the user's data from the GPUs to the server's host memory. The executor replica returns the result to the user only after the GPU operations finish and GPU state is copied to host memory.

3.4 Resource Scheduling

3.4.1 NotebookOS's Default Placement Policy. NotebookOS is designed to be highly modular. The system can support arbitrary resource scheduling policies, and implementing support for a new policy is accomplished by implementing a simple interface. NotebookOS's default kernel replica placement policy takes several different factors into account. *First*, NotebookOS considers the number of idle GPUs available on each GPU server. In particular, NotebookOS favors placing kernel replica containers on servers with more idle GPUs available.

Second, NotebookOS considers a metric referred to as the *subscription ratio* (SR) of a GPU server. The SR of a GPU server is defined as $\frac{S}{G \cdot R}$, where S stands for the number of *subscribed* GPUs, which is the sum of all the GPUs requested by kernel replicas scheduled on that server (including idle kernel replicas), G is the number of server GPUs (ranging from 1 to 8), and R is the number of replicas per distributed kernel ($R = 3$). R accounts for the fact that, at a given time, only one out of R replicas per distributed kernel will serve as the executor and actively use GPUs. Dividing by R adjusts for the redundancy introduced by NotebookOS's distributed kernel, ensuring that the SR captures the *effective* GPU subscription ratio rather than the total GPU subscription capacity of a single server. For example, if server H with 8 GPUs is serving 4 kernel containers each requiring 4 GPUs, then the number of subscribed GPUs S for H is $4 \times 4 = 16$. Consequently, H's SR is $\frac{16}{8 \cdot 3} = 0.667$. A cluster-wide SR of 1 or lower theoretically ensures that, during GPU code execution, any kernel replica k can acquire its required number of GPUs on a server that is hosting k, where k is one of the R replicas of some distributed kernel.

Third, NotebookOS implements a dynamic, cluster-wide limit on the SR. While the cluster size is a function of the number of GPUs used by actively-training kernel replicas (detailed in §3.4.2), the maximum permissible SR across the entire cluster is dynamically adjusted. This limit is calculated as $\frac{\sum S}{\sum G \cdot R}$, where $\sum S$ denotes the total number of subscribed GPUs for all kernel replicas across all servers, and $\sum G$ represents the total GPU count across all servers. If scheduling an additional kernel replica on a server would cause the server's SR to exceed this limit, the server is rejected in favor of another. The effect of this cluster-wide SR limit is illustrated in the evaluation results discussed in §5.3.4.

3.4.2 Scale-Out Operations.

Scale-out operations involve provisioning additional servers in a platform-dependent manner, and then waiting for the Local Schedulers that are started on the new servers to connect and register with the Global Scheduler. Scale-out operations occur in one of two scenarios. *First*, they are triggered in response to a failed attempt to place one or more replicas of a distributed kernel. Such a failure occurs when there are no viable candidate servers across the cluster to serve the kernel replicas. Upon triggering the scale-out operation, the placement of the corresponding kernel replicas is paused. Resources are immediately reserved for the paused kernel replicas on newly provisioned servers, i.e., before the servers are fully added to the NotebookOS server cluster. Once the servers are ready, NotebookOS resumes placing the replicas on them.

Second, scale-out operations can also be triggered by NotebookOS's auto-scaling policy. NotebookOS's auto-scaler runs on a configurable interval, monitoring cluster resource utilization and determining whether servers should be added or removed. To handle bursts of training requests, NotebookOS maintains a *scaling buffer* of "extra" servers. To decide if additional servers should be added, the auto-scaler first examines the total number of GPUs actively committed ($\sum C$) to kernel replicas (that are actively executing GPU code) on servers across the NotebookOS cluster, where C is the number of actively-utilized GPUs on a server. The expected cluster capacity ($\sum G'$) is defined as $\sum G' = f \cdot \sum C$, where f is a scalar multiplier that controls how aggressively NotebookOS scales. If the current cluster capacity is smaller than $\sum G'$, additional GPU servers are provisioned. Intuitively, if the number of actively-utilized GPUs is low, then the auto-scaler will not try to add additional servers. If the number of actively-utilized is GPUs is high and many existing servers are at capacity, then NotebookOS's auto-scaler triggers scale-out. We set f to 1.05, as we have found empirically that this enables an appropriate degree of auto-scaling during testing.

If cluster resource usage is too low, NotebookOS's auto-scaler attempts to release 1-2 idle servers at a time (where idle servers are those with no active training kernel replicas.) The auto-scaler determines if scaling-in is appropriate using a similar approach to scaling-out: if $\sum G'$ is less than the number of currently-provisioned GPU servers within the cluster, then NotebookOS gradually releases servers 1 to 2 at a time until this condition is no longer met.

4 NotebookOS Implementation

To demonstrate NotebookOS's efficacy, we implemented a fully functional prototype Jupyter Notebook platform. NotebookOS maintains compatibility with all Jupyter clients by reusing the IPython messaging protocol [41]. It leverages Jupyter Server from the official `base-notebook` Docker image [10] and introduces custom components for NotebookOS-specific functionality. All custom components were implemented using the official Jupyter Server API extension methods [56], building on the default Jupyter Server.

NotebookOS is platform-agnostic. We developed and evaluated NotebookOS atop Docker Swarm [23] and Docker Compose [22]; Kubernetes [61] is also supported. To simplify deployment, experimentation, and reproducibility, we provide a set of Ansible [3] playbooks to automate setup, installation, and deployment on these container orchestration platforms. We have implemented NotebookOS in approximately 282k lines of code over roughly two person-years. Refer to Appendix C for additional implementation details.

5 Evaluation

In this section, we present our evaluation of NotebookOS.

5.1 Experimental Setup & Methodology

5.1.1 Baselines.

We implement three representative baseline policies within NotebookOS itself, each of which represents a unique category of alternative solutions.

Reservation emulates the behavior of current notebook platforms like Adobe research cluster and Google Colab. Reservation creates one long-running kernel container for each user session that remains active for the entire duration

Table 1. Models and datasets used in the evaluation along with their associated application domains.

App domain	Dataset	Model
Computer Vision	CIFAR-10, CIFAR-100, Tiny ImageNet	VGG-16, ResNet-18, Inception v3
Natural Language Processing	IMDb Large Movie Reviews, CoLA	BERT, GPT-2
Speech Recognition	LibriSpeech	Deep Speech 2

of the user session. Fixed resources, including GPUs, are exclusively allocated to each long-running kernel container. **Batch** is a baseline representing the class of batch GPU cluster schedulers [35, 66, 72, 88, 106] designed for long-running GPU training workloads. Batch provisions a kernel replica container each time a user submits code and a job request for execution (e.g., to a slurm scheduler). The new container serves the training request before terminating. As such, Batch approximates on-demand resource scaling. While various GPU schedulers differ in their design, implementation, and scheduling/placement algorithms, they all share several significant sources of overhead in the context of *notebook scheduling* for IDLT workloads. We implemented Batch using a first-come, first-serve (FCFS) job scheduling and GPU allocation policy within NotebookOS to approximate the performance of these GPU schedulers.

NotebookOS (LCP) (large container pool) is an alternative NotebookOS baseline that sacrifices (some) interactivity in favor of reduced resource cost. NotebookOS (LCP) serves as a tool for exploring the trade-off between interactivity and resource efficiency. NotebookOS (LCP) employs a larger pool of pre-warmed containers than NotebookOS's default configuration, which uses three kernel replicas and a significantly smaller pre-warmed container pool used during replica migrations. When a cell task arrives, NotebookOS selects a warm container[1] from the pool to serve the request. After execution, the container is returned to the pool rather than being terminated, as in NotebookOS's default policy. Addressing security and privacy concerns related to container sharing is out of scope for this paper but can leverage existing research [38, 64, 109].

5.1.2 Setup. We conducted our evaluation on AWS EC2 [26]. We ran each baseline on a cluster of 30 GPU EC2 VMs, each equipped with 8 GPUs to match the Adobe research cluster production setups. We developed a workload driver and dashboard to automate workload deployment and execution on NotebookOS. Following [33], we integrated support for a diverse set of DL models and datasets across several different application domains (Table 1). The workload driver randomly assigns each client an application domain, after which a random dataset and model are assigned. Each cell task request submitted by the client will train its assigned model on the assigned dataset, which are retrieved by kernel replicas from an AWS S3 bucket.

[1]We analyzed the Python dependencies of notebook containers from our workload and observed substantial overlaps in common dependencies.

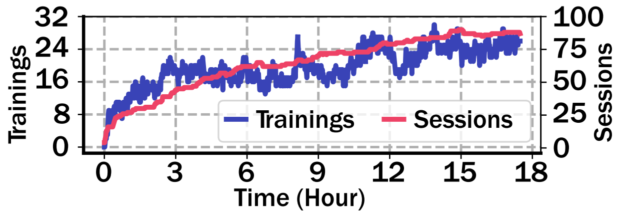


Figure 7. The number of active user-submitted training tasks and active user sessions during the 17.5-hour AdobeTrace.

We evaluate NotebookOS using a 17.5-hour excerpt of the Adobe research cluster workload trace. To further elucidate its performance and monetary cost implications of NotebookOS, we also implemented a robust, detailed simulator that we used to run the full Adobe research cluster trace. The simulator implemented NotebookOS's default scheduling/placement policy along with the baseline policies described in §5.1.1. The results of our simulation study on NotebookOS are presented in §5.5.

5.2 Prototype Evaluation

This section presents the results of our prototype evaluation.

5.3 Active Sessions & Training Events

Figure 7 presents a timeline plot showing the number of active sessions during the execution of the 17.5-hour workload trace excerpt on NotebookOS. This series is plotted on the secondary (i.e., right) y-axis. The number of active sessions increases from 0 at the beginning of the trace excerpt to 87 by the end of the excerpt. The maximum number of active sessions at any given time throughout the 17.5-hour AdobeTrace excerpt is 90.

Figure 7 also presents a timeline plot showing the number of user-submitted training events being processed during the execution of the 17.5-hour AdobeTrace excerpt on NotebookOS. This series is plotted on the primary (i.e., left) y-axis. Initially there are no active, user-submitted trainings. At the end of the trace excerpt, there are 26 active user-submitted trainings. The mean and medium number of active, user-submitted trainings are 19.5 and 19, respectively. The maximum number of active user-submitted trainings at any given time throughout the summer AdobeTrace is 141 (see Figure 20 in appendix A). The maximum number of active user-submitted trainings at any given time throughout the 17.5-hour AdobeTrace excerpt is 34.

5.3.1 Resource Usage & Efficiency. Batch (Figure 8 left), achieves significantly improved resource utilization compared to Reservation because Batch only provisions containers and allocates resources in response to training requests. NotebookOS achieves higher GPU utilization than

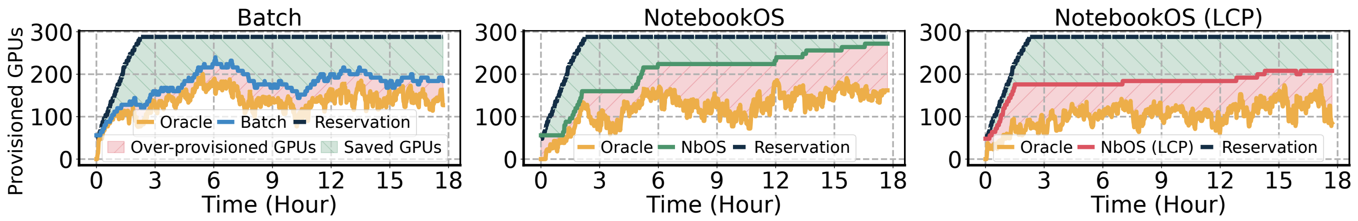
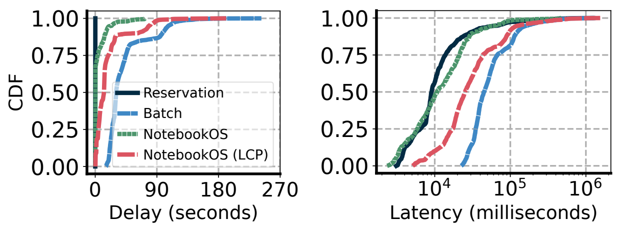


Figure 8. Provisioned GPUs timelines. Curves for Batch (left), NotebookOS (middle), and NotebookOS (LCP) (right) represent the number of GPUs provisioned under each policy, while "oracle" represents an optimal policy that provisions the exact number of GPUs required to serve training requests. The GPUs saved relative to Reservation is shown in the green-shaded region, while the number of GPUs over-provisioned compared to "oracle" are represented by the orange-shaded region.



(a) Interactivity delay. (b) Task completion time (TCT).

Figure 9. CDFs of (a) *interactivity delays* and (b) *task completion times (TCT)* across different scheduling policies.

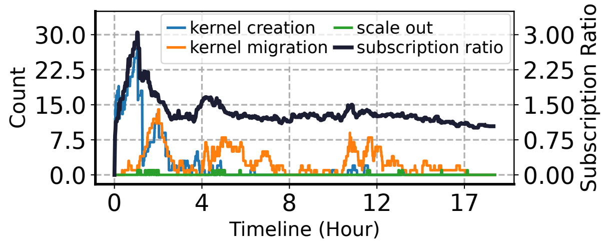


Figure 10. Timeline of major events occurred while executing the 17.5-hour workload on NotebookOS, with NotebookOS's subscription ratio plotted on the secondary Y-axis.

Reservation (Figure 8, middle), saving 1,187.66 (NotebookOS) and 1,662.53 (NotebookOS (LCP)) GPU hours. However, because NotebookOS maintains three long-running replicas per kernel as well as a small buffer of "extra" GPU servers for request bursts, NotebookOS provisions more servers than Batch, which only allocates GPUs during cell executions. NotebookOS (LCP) (Figure 8, right) improves resource efficiency, provisioning 23.52% fewer GPUs than NotebookOS but still 18.18% more than Batch. Note that the oracle curve shows the number of GPUs required to serve all active training requests, rather than aggregated GPU bandwidth utilization, which is shown in Figure 2(d). NotebookOS provisions slightly more GPUs than Batch/LCP to account for its use of replicated kernels and to handle request bursts.

5.3.2 Interactivity. Figure 9(a) plots a CDF of the per-task interactivity delays incurred by the different scheduling policies. The interactivity delay is the interval between the instant that a client submits a Jupyter "`execute_request`" message to a kernel and the instant that the kernel begins executing the user-submitted code included within the message. Long interactivity delays are perceptible to clients and degrade the user experience.

The Reservation baseline binds GPUs to a user's notebook kernel for its full lifetime, ensuring high interactivity. However, IDLT workloads often under-utilize GPU resources, as users frequently debug, edit code, or pause while GPUs remain idle but reserved.

Effect of Multiple Kernel Replicas. NotebookOS commits GPUs to a kernel replica immediately upon receiving an execution request 89.6% of the time. Similarly, 89.45% of the time, NotebookOS reused the same executor replica for consecutive execution request. NotebookOS's use of replicated kernels maximizes the chances that at least one replica has available resources when a training request arrives, achieving nearly the same interactivity as the `reservation` baseline.

5.3.3 Task Completion Time (TCT). TCT measures the interval between cell submission and the instant the cell's execution is completed. As shown in Figure 9(b), NotebookOS achieves a TCT distribution comparable to Reservation, with slightly higher TCTs from the 38^{th} to the 90^{th} percentile. This is because some sessions were not able to find available GPUs on their servers (due to oversubscription and server's full GPU usage) and thus are migrated to different servers. When NotebookOS's pre-warmed containers were exhausted, new containers were created on demand, incurring long cold startup delays. This can also be observed from the tail of NotebookOS's interactivity delays in Figure 9(a). NotebookOS (LCP) sees much longer TCTs as a submitted cell request triggered a warming-up operation to download model parameters and datasets. FCFS yields the highest TCTs due to on-demand notebook kernel provisioning and mandatory pre- and post-processing data I/O (model parameters and datasets) for each submission.

5.3.4 Subscription Ratio. The relationship between the cluster-wide subscription ratio (SR, described in §3.4) and the triggering of scale-out events by NotebookOS's auto-scaling policy (§3.4.2) are clearly shown in Figure 10. Specifically, the SR increases sharply at the beginning of the workload when

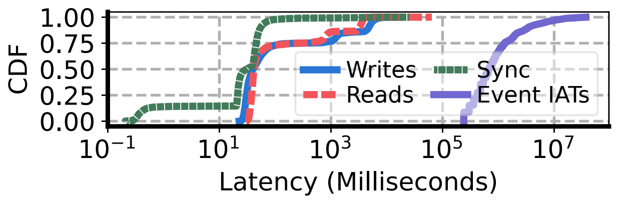


Figure 11. CDF of the latency of reading and writing large objects and synchronizing smaller object state via the Raft synchronization protocol. Note that the x-axis is log-scale.

many new Distributed Kernels are created. To accommodate these kernel replicas, NotebookOS triggers scale-out events. As new servers are provisioned, the cluster-wide SR falls. Around hour two, there is another burst of kernel replica creations, leading to another increase in the SR. Subsequently, additional servers are provisioned to serve the new kernel replicas shortly after hour four, leading to another drop in the SR. A similar pattern occurs before hour twelve. Similarly, there is an increase in the frequency of kernel migrations when the SR begins to climb, reflecting growing GPU resource saturation and potential contention when training requests arrive. Notably, a spike in migrations coincides with the first spike in the SR, followed by additional migrations after hours four and twelve of workload execution.

5.4 Object Synchronization Overhead

Figure 11 presents a timeline plot showing the number of active sessions during the execution on NotebookOS of the 17.5-hour AdobeTrace excerpt. The "Sync" curve corresponds to the end-to-end latency of synchronizing small objects via the Raft protocol (§3.2.4). The "Sync" latency is typically extremely short: the 90th, 95th, and 99th percentile values are 54.79ms, 66.69ms, and 268.25ms, respectively. The "Reads" and "Writes" curves correspond to the latencies observed when reading and writing large objects from and to the Distributed Data Store, respectively. These operations are reasonably short, as 99% of read and write operations complete within roughly 3.95 and 7.07 seconds, respectively.

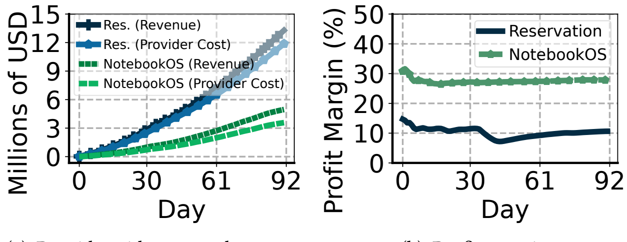
As discussed in §2.3, the 50th and 75th percentile of task IATs are 300 seconds (5 minutes) and 480 seconds (8 minutes) for the AdobeTrace, respectively. The shortest event IAT within the AdobeTrace is 240 seconds, or 4 minutes. Notably, the overhead of writing and reading large objects to and from the Distributed Data Store is completely contained within event IATs. As a result, this overhead is almost completely hidden from users in IDLT workloads such as the Adobe research cluster workload.

5.5 Simulation Study

This section present the results of our simulation study.

5.5.1 Monetary Cost. To understand the cost implications of NotebookOS, we compare it with Reservation in terms of provider costs—specifically, the operational cost of provisioning EC2 resources, as done by production notebook platforms like Adobe research cluster—and the revenue generated by the notebook service provider.

We implemented a simple billing model, assuming the provider covers AWS EC2 VM costs. Users pay 1.15× the provider's rate, proportional to resource usage. Standby Distributed Kernel replicas are charged 12.5% of the base rate. For example, if a provider pays \$10/hour for an 8-GPU EC2 VM, each standby replica is billed \$1.44/hour ($10 \times 1.15 \times 0.125$). When a replica runs a training task using 4 GPUs, the charge increases to \$5.75/hour ($10 \times 1.15 \times 0.5$). Reservation follows the same 1.15× multiplier for on-demand GPU usage. As shown in Figure 12(a), NotebookOS achieves a provider-side \$-cost reduction of up to 69.87% compared to Reservation by the end of the trace, thanks to its GPU resource savings. NotebookOS also offer a higher profit margin (Figure 12(b)) by significantly lowering provider costs and modestly increasing revenue through minimal charges on idle replicas. The increased profit comes with the added benefit of guaranteed programming interactivity.



(a) Provider-side cost and revenue. (b) Profit margin.

Figure 12. Timeline of provider cost (provisioned EC2 resources), revenue, and profit margin.

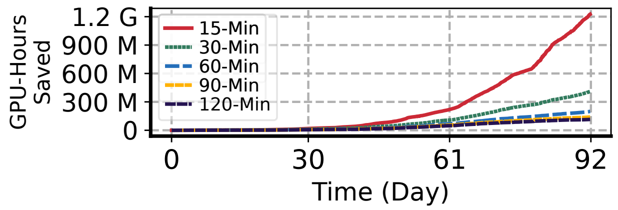


Figure 13. The number of GPU-hours saved by NotebookOS by avoiding unnecessary re-execution of notebook cells following idle reclamations of user sessions.

5.5.2 GPU Hours Saved. Figure 13 illustrates the number of GPU hours saved by NotebookOS by avoiding unnecessary re-execution of notebook cells following idle session reclamations across the entire AdobeTrace. Each of the five lines in the plot represents a different configuration of the idle reclamation interval—the duration for which a Jupyter Notebook session may remain idle before its associated kernel is reclaimed. In the absence of NotebookOS's state replication and persistence mechanisms, reclaiming idle sessions would result in the loss of in-memory state, requiring cell re-execution and additional GPU usage when users return.

By preserving session state, NotebookOS prevents this redundant computation and significantly reduces GPU waste.

6 Related Work

Notebook Platforms. Commercial notebook services such as Google Colab [17], NaaS [71], Lentiq [63], and JupyterHub [53] allow users to reserve GPUs for however long they like with great flexibility but low GPU utilization. These platforms add keep-alive and timeout strategies so that the platform can automatically terminate a notebook instance if the user does not interact with it for a configurable period of time. The keep-alive strategy can help save unused GPU resources to some extent but does not fundamentally address the low GPU utilization issue. Azure Machine Learning [9], Open OnDemand [84, 85], among others [74, 91, 92], offer "batch-connect" notebook services to allow users to submit GPU jobs through a batch scheduler. This approach introduces significant runtime cold startup cost and batch scheduling delays during request bursts, undermining the interactive nature of notebook programming.

GPU Cluster Schedulers. State-of-the-art batch GPU cluster schedulers [35, 66, 72, 88, 106] aim to balance the competing demands of long-running GPU training jobs but overlook responsiveness and interactivity, making them poorly suited for notebook IDLT workloads. Gandiva [106] is a GPU cluster scheduler that leverages domain-specific knowledge and an introspective scheduler to dynamically optimize BDLT workloads. Tiresias [35] is a GPU cluster manager that minimizes job completion time by prioritizing jobs with partial information and tasks with the least service received. Themis [66] employs a 2-tiered, "finish-time fair" approach where workloads bid for GPUs in an auction, balancing short-term efficiency with long-term fairness.

Serverless. Optimized snapshot-loading reduces cold start costs for CPU [4, 6, 81, 100] and GPU [27, 39, 111] serverless functions. Locality-driven keep-alive [28, 65, 101, 109] optimize container caching for minimizing cold start costs. NotebookOS's pre-warmed container pool can leverage these methods to further cut GPU startup cost.

GPU Virtualization. Existing solutions for GPU virtualization and sharing leverage NVIDIA MIG [79] / MPS [80] as well as user-space CUDA API interposition and remoting techniques for GPU kernel performance isolation [105, 112], memory isolation [110], and parallel executions within a single remote GPU [25]. Currently, NotebookOS supports only coarse-grained, exclusive GPU time-sharing via kernel replication and executor replica election but does not support fine-grained, concurrent, spatial GPU sharing. Enabling fine-grained GPU sharing is part of our future work.

Consensus Protocols and Leader Election Algorithms. Raft [83] is a widely-adopted consensus algorithm designed for understandability while providing strong consistency, featuring leader elections with randomized timeouts (to prevent ties) and log replication via heartbeats. NotebookOS's state synchronization and executor selection protocols are implemented atop Raft, as they require more complex logic than what is provided directly by Raft's leader elections. The use of Raft provides robust, off-the-shelf consistency and fault tolerance support. Alternative consensus protocols, such as Viewstamped Replication [82], EPaxos [69], and Zab [45], offer tradeoffs in latency, leader dependence, and throughput. The Bully algorithm [30] targets crash-recovery systems and elects the node with the highest ID, while the Ring algorithm [15, 62] organizes nodes into a logical ring in which communication occurs only between adjacent neighbors. Systems like ZooKeeper [40] and Chubby [12] offer practical leader election mechanisms, informing the design of resilient control planes in NotebookOS.

7 Limitations and Future Work

NotebookOS does not yet support GPU sharing or fractional allocations. Future work will incorporate GPU virtualization using tools such as HAMi [36]. While multi-server training is also not supported, it can be enabled through integration with distributed training frameworks like Ray [70]. These capabilities are largely orthogonal to NotebookOS's core abstractions, and NotebookOS's mechanisms for scheduling and execution would remain largely unchanged. Lastly, NotebookOS could further optimize CPU-only notebook cells by enabling elastic burst-parallel executions through the integration of stateful serverless scheduling techniques [14, 96].

8 Conclusion

The key insight of this paper is that notebooks are long-running but with fragmented executions, rendering intermittent, sporadic, and often low GPU utilization. We built NotebookOS, a GPU-efficient notebook platform designed to meet the unique notebook workload requirements and prioritizes interactivity. NotebookOS oversubscribes and multiplexes GPUs using a novel replicated kernel design. NotebookOS dynamically allocates GPUs only for notebook cell executions that involve GPU training. We evaluated NotebookOS using production notebook workloads. Results show that NotebookOS reduces cluster GPU resource cost significantly compared to existing notebook platforms. Integrating NotebookOS in production is part of our future work.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Rodrigo Bruno, for their valuable feedback and comments, which improved the paper. This research was supported in part by NSF grants OAC-2411009 and CNS-2322860 (NSF CAREER Award). We also acknowledge support from NSF CloudBank for providing AWS credits, and thank Adobe for their generous research gift. Benjamin Carver was supported by a Presidential Scholarship from George Mason University.

References

[1] Adobe. https://www.adobe.com/.

[2] Adobe Experience Platform: A supercharged engine, finely tuned to make customer experiences hum. https://business.adobe.com/products/experience-platform/adobe-experience-platform.html.

[3] Ansible. https://www.ansible.com/.

[4] Lixiang Ao, George Porter, and Geoffrey M. Voelker. Faasnap: Faas made fast using snapshot-based vms. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 730–746, New York, NY, USA, 2022. Association for Computing Machinery.

[5] Awesome Jupyter: A curated list of awesome Jupyter projects, libraries and resources. https://github.com/markusschanta/awesome-jupyter.

[6] Improving startup performance with Lambda SnapStart. https://docs.aws.amazon.com/lambda/latest/dg/snapstart.html.

[7] AWS S3. https://aws.amazon.com/s3/.

[8] Amazon SageMaker Studio. https://aws.amazon.com/sagemaker-ai/studio/.

[9] Run Jupyter notebooks in your workspace. https://learn.microsoft.com/en-us/azure/machine-learning/how-to-run-jupyter-notebooks?view=azureml-api-2.

[10] DockerHub: jupyter/base-notebook. https://hub.docker.com/r/jupyter/base-notebook/.

[11] Bloomberg BQuant (BQNT). https://mingze-gao.com/posts/bloomberg-bquant/.

[12] Mike Burrows. The chubby lock service for Loosely-Coupled distributed systems. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI 06)*, Seattle, WA, November 2006. USENIX Association.

[13] Benjamin Carver, Runzhou Han, Jingyaun Zhang, Mai Zheng, and Yue Cheng. λfs: A scalable and elastic distributed file system metadata service using serverless functions. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2023, New York, NY, USA, 2023. Association for Computing Machinery.

[14] Benjamin Carver, Jingyuan Zhang, Ao Wang, Ali Anwar, Panruo Wu, and Yue Cheng. Wukong: A scalable and locality-enhanced framework for serverless parallel computing. In *ACM Symposium on Cloud Computing 2020 (SoCC'20)*, 2020.

[15] Ernest Chang and Rosemary Roberts. An improved algorithm for decentralized extrema-finding in circular configurations of processes. *Commun. ACM*, 22(5):281–283, May 1979.

[16] CoCalc: Collaborative Calculation and Data Science. https://cocalc.com/.

[17] Google Colaboratory. https://github.com/googlecolab.

[18] The easiest way to collaborate on Jupyter. https://youtu.be/ecl48qKyREE?si=cRsCDTMsV1hstf76.

[19] CRIU: A project to implement checkpoint/restore functionality for Linux. https://criu.org/Main_Page.

[20] Data Driven Science & Engineering: Machine Learning, Dynamical Systems, and Control. https://github.com/dynamicslab/databook_python.

[21] Data center operators must lower risk as GPU costs rise: Future-proof your data center (presented by DDC solutions) | GTC 25 2025 | NVIDIA on-demand.

[22] Docker Compose. https://docs.docker.com/compose/.

[23] Docker Swarm. https://docs.docker.com/engine/swarm/.

[24] Dozzle: Realtime log viewer for containers. Supports Docker, Swarm and K8s. https://github.com/amir20/dozzle.

[25] José Duato, Antonio J. Peña, Federico Silla, Rafael Mayo, and Enrique S. Quintana-Ortí. rcuda: Reducing the number of gpu-based accelerators in high performance clusters. In *2010 International Conference on High Performance Computing & Simulation*, pages 224–231, 2010.

[26] Amazon EC2. https://aws.amazon.com/ec2/.

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

[28] Alexander Fuerst and Prateek Sharma. Faascache: keeping serverless computing alive with greedy-dual caching. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 386–400, New York, NY, USA, 2021. Association for Computing Machinery.

[29] The past, present and future of the Jupyter. https://youtu.be/G6guUzERa9w?si=DESPWvrp4qxcCiPS.

[30] Garcia-Molina. Elections in a distributed computing system. *IEEE Transactions on Computers*, C-31(1):48–59, 1982.

[31] Google Cloud Vertex AI Notebooks. https://cloud.google.com/vertex-ai-notebooks.

[32] Google Colaboratory: Frequently Asked Questions . https://research.google.com/colaboratory/faq.html.

[33] Diandian Gu, Yihao Zhao, Yinmin Zhong, Yifan Xiong, Zhenhua Han, Peng Cheng, Fan Yang, Gang Huang, Xin Jin, and Xuanzhe Liu. Elasticflow: An elastic serverless training platform for distributed deep learning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 266–280, New York, NY, USA, 2023. Association for Computing Machinery.

[34] Jing Gu, Shengbo Song, Ying Li, and Hanmei Luo. Gaiagpu: Sharing gpus in container clouds. In *2018 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Ubiquitous Computing & Communications, Big Data & Cloud Computing, Social Computing & Networking, Sustainable Computing & Communications (ISPA/IUCC/BDCloud/SocialCom/SustainCom)*, pages 469–476, 2018.

[35] Juncheng Gu, Mosharaf Chowdhury, Kang G. Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. Tiresias: A GPU cluster manager for distributed deep learning. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 485–500, Boston, MA, feb 2019. USENIX Association.

[36] HAMi: Heterogeneous AI Computing Virtualization Middleware. https://github.com/Project-HAMi/HAMi.

[37] Daniel Howley. These 169 industries are being hit by the global chip shortage.

[38] Jialiang Huang, MingXing Zhang, Teng Ma, Zheng Liu, Sixing Lin, Kang Chen, Jinlei Jiang, Xia Liao, Yingdi Shan, Ning Zhang, Mengting Lu, Tao Ma, Haifeng Gong, and YongWei Wu. Trenv: Transparently share serverless execution environments across different functions and nodes. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, SOSP '24, page 421–437, New York, NY, USA, 2024. Association for Computing Machinery.

[39] Zhuobin Huang, Xingda Wei, Yingyi Hao, Rong Chen, Mingcong Han, Jinyu Gu, and Haibo Chen. Parallelgpuos: A concurrent os-level gpu checkpoint and restore system using validated speculation, 2024.

[40] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: Wait-free coordination for internet-scale systems. In *2010 USENIX Annual Technical Conference (USENIX ATC 10)*. USENIX Association, jun 2010.

[41] IPython Messaging Protocol. https://jupyter-client.readthedocs.io/en/latest/messaging.html.

[42] Python ast: Abstract Syntax Tree. https://docs.python.org/3/library/ast.html.

[43] Suhas Jayaram Subramanya, Daiyaan Arfeen, Shouxu Lin, Aurick Qiao, Zhihao Jia, and Gregory R. Ganger. Sia: Heterogeneity-aware, goodput-optimized ml-cluster scheduling. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 642–657, New York, NY, USA, 2023. Association for Computing Machinery.

[44] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, unjie Qian, Wencong Xiao, and Fan Yang. Analysis of large-scale multi-tenant gpu clusters for dnn training workloads. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, page 947–960, USA, 2019. USENIX Association.

[45] Flavio P. Junqueira, Benjamin C. Reed, and Marco Serafini. Zab: High-performance broadcast for primary-backup systems. In *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks*, DSN '11, page 245–256, USA, 2011. IEEE Computer Society.

[46] Available Jupyter Kernels. https://github.com/jupyter/jupyter/wiki/Jupyter-kernels.

[47] Jupyter on AWS: A secure, scalable, and collaborative Jupyter experience on AWS. https://aws.amazon.com/jupyter/.

[48] Jupyter Community Workshop (2019). https://jupyter-workshop-2019.lbl.gov/.

[49] JupyterCon: Expand your knowledge, explore new possibilities, and connect with fellow pioneers. https://www.jupytercon.com/.

[50] Jupyter Enterprise Gateway documentation: Culling idle kernels. https://github.com/ds2-lab/LambdaFS-Benchmark-Utility.

[51] Teaching and Learning with Jupyter. https://jupyter4edu.github.io/jupyter-edu-book/.

[52] Jupyter Education Projects. https://docs.jupyter.org/en/latest/projects/education.html.

[53] Jupter Hub: A multi-user version of the notebook designed for companies, classrooms and research labs. https://jupyter.org/hub.

[54] IJulia.jl. https://github.com/JuliaLang/IJulia.jl.

[55] IRKernel. https://github.com/IRkernel/IRkernel.

[56] Jupyter Kernel Provisioning. https://jupyter-client.readthedocs.io/en/latest/provisioning.html.

[57] Jupyterlab: Real time collaboration. https://jupyterlab.readthedocs.io/en/latest/user/rtc.html.

[58] Jupyter Notebook. https://jupyter.org/.

[59] JupyterLab: Real Time Collaboration. https://jupyterlab.readthedocs.io/en/latest/user/rtc.html.

[60] Anton Korinek and Jai Vipra. Concentrating intelligence: Scaling and market structure in artificial intelligence. Working Paper 33139, National Bureau of Economic Research, November 2024.

[61] kubernetes: Production-Grade Container Orchestration. https://kubernetes.io/.

[62] Gérard Le Lann. Distributed systems - towards a formal approach. In Bruce Gilchrist, editor, *Information Processing, Proceedings of the 7th IFIP Congress 1977, Toronto, Canada, August 8-12, 1977*, pages 155–160. North-Holland, 1977.

[63] Jupyter as a Service on Lentiq. https://lentiq.com/jupyter-as-a-service.

[64] Zijun Li, Linsong Guo, Quan Chen, Jiagan Cheng, Chuhao Xu, Deze Zeng, Zhuo Song, Tao Ma, Yong Yang, Chao Li, and Minyi Guo. Help rather than recycle: Alleviating cold startup in serverless computing through Inter-Function container sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 69–84, Carlsbad, CA, July 2022. USENIX Association.

[65] Qichang Liu, Yue Cheng, Haiying Shen, Ao Wang, and Bharathan Balaji. Concurrency-informed orchestration for serverless functions. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '25, page 147–161, New York, NY, USA, 2025. Association for Computing Machinery.

[66] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. Themis: Fair and efficient GPU cluster scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 289–304, Santa Clara, CA, feb 2020. USENIX Association.

[67] Notebooks at Microsoft. https://visualstudio.microsoft.com/vs/features/notebooks-at-microsoft/.

[68] Modal: GPU Memory Snapshots: Supercharging Sub-second Startup. https://modal.com/blog/gpu-mem-snapshots.

[69] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 358–372, New York, NY, USA, 2013. Association for Computing Machinery.

[70] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging AI applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 561–577, Carlsbad, CA, oct 2018. USENIX Association.

[71] NaaS: Jupyter Notebooks as a Service. https://home.naas.ai/.

[72] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. Heterogeneity-Aware cluster scheduling policies for deep learning workloads. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 481–498. USENIX Association, November 2020.

[73] Accelerating Discovery for NASA Cryosphere Communities with JupyterHub. https://youtu.be/9itR-_EXa5c?si=cod7BLXGljeSPT_8.

[74] Batch Processing Jupyter Notebooks with Papermill. https://ncar.github.io/esds/posts/2022/batch-processing-notebooks-with-papermill/.

[75] Beyond Interactive: Notebook Innovation at Netflix. https://netflixtechblog.com/notebook-innovation-591ee3221233.

[76] Salman Niazi, Mahmoud Ismail, Seif Haridi, Jim Dowling, Steffen Grohsschmiedt, and Mikael Ronström. Hopsfs: Scaling hierarchical file system metadata using newsql databases. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 89–104, Santa Clara, CA, feb 2017. USENIX Association.

[77] NotebookOS Dashboard GitHub Repository. https://github.com/ds2-lab/NotebookOS-Dashboard.

[78] NotebookOS GitHub Repository. https://github.com/ds2-lab/NotebookOS/.

[79] NVIDIA GPU Multi-Instance GPU. https://www.nvidia.com/en-us/technologies/multi-instance-gpu/.

[80] NVIDIA GPU Multi-Process Service. https://docs.nvidia.com/deploy/mps/index.html.

[81] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. SOCK: Rapid task provisioning with serverless-optimized containers. In *USENIX ATC 18*, Boston, MA, 2018. USENIX Association.

[82] Brian M. Oki and Barbara H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, PODC '88, page 8–17, New York, NY, USA, 1988. Association for Computing Machinery.

[83] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, Philadelphia, PA, jun 2014. USENIX Association.

[84] Open OnDemand. https://openondemand.org/.

[85] Batch Connect - OSC Jupyter. https://github.com/OSC/bc_osc_jupyter.

[86] PayPal Notebooks: Data science and machine learning at scale, powered by Jupyter. https://conferences.oreilly.com/jupyter/jup-ny/public/schedule/detail/68405.html.

[87] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[88] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R. Ganger, and Eric P. Xing. Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 1–18. USENIX Association, July 2021.
[89] Redis. https://redis.io/.
[90] https://www.reportsinsights.com ReportsInsights. Graphic Processors Market 2022 | Size | Forecast By 2030.
[91] Amazon SageMaker Notebooks: Fully managed notebooks for exploring data and building ML models. https://aws.amazon.com/sagemaker/notebooks/.
[92] SageMaker Studio Lab: Learn and experiment with machine learning. https://studiolab.sagemaker.aws/.
[93] Richard D. Schlichting and Fred B. Schneider. Fail-stop processors: an approach to designing fault-tolerant computing systems. *ACM Trans. Comput. Syst.*, 1(3):222–238, August 1983.
[94] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 205–218. USENIX Association, July 2020.
[95] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.
[96] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Jose M. Faleiro, Joseph E. Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. Cloudburst: Stateful Functions-as-a-Service. *Proceedings of the VLDB Endowment*, 13(12):2438–2452, 2020.
[97] Foteini Strati, Paul Elvinger, Tolga Kerimoglu, and Ana Klimovic. Ml training with cloud gpu shortages: Is cross-region the answer? In *Proceedings of the 4th Workshop on Machine Learning and Systems*, EuroMLSys '24, page 107–116, New York, NY, USA, 2024. Association for Computing Machinery.
[98] Mark Sweney. Global shortage in computer chips 'reaches crisis point'. https://www.theguardian.com/business/2021/mar/21/global-shortage-in-computer-chips-reaches-crisis-point.
[99] Traefik: The Cloud Native Application Proxy. https://traefik.io/traefik.
[100] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. Benchmarking, analysis, and optimization of serverless function snapshots. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 559–572, New York, NY, USA, 2021. Association for Computing Machinery.
[101] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking behind the curtains of serverless platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 133–146, Boston, MA, 2018. USENIX Association.
[102] Zhuang Wang, Zhen Jia, Shuai Zheng, Zhen Zhang, Xinwei Fu, T. S. Eugene Ng, and Yida Wang. Gemini: Fast failure recovery in distributed training with in-memory checkpoints. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 364–381, New York, NY, USA, 2023. Association for Computing Machinery.
[103] Xingda Wei, Zhuobin Huang, Tianle Sun, Yingyi Hao, Rong Chen, Mingcong Han, Jinyu Gu, and Haibo Chen. Phoenixos: Concurrent os-level gpu checkpoint and restore with validated speculation. In *Proceedings of the ACM SIGOPS 31th Symposium on Operating Systems Principles*, 2025.
[104] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. MLaaS in the wild: Workload analysis and scheduling in Large-Scale heterogeneous GPU clusters. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 945–960, Renton, WA, apr 2022. USENIX Association.
[105] Bingyang Wu, Zili Zhang, Zhihao Bai, Xuanzhe Liu, and Xin Jin. Transparent GPU sharing in container clouds for deep learning workloads. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 69–85, Boston, MA, apr 2023. USENIX Association.
[106] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou. Gandiva: Introspective cluster scheduling for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 595–610, Carlsbad, CA, oct 2018. USENIX Association.
[107] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. AntMan: Dynamic scaling on GPU clusters for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 533–548. USENIX Association, nov 2020.
[108] Zongheng Yang, Zhanghao Wu, Michael Luo, Wei-Lin Chiang, Romil Bhardwaj, Woosuk Kwon, Siyuan Zhuang, Frank Sifei Luan, Gautam Mittal, Scott Shenker, and Ion Stoica. SkyPilot: An intercloud broker for sky computing. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 437–455, Boston, MA, April 2023. USENIX Association.
[109] Hanfei Yu, Rohan Basu Roy, Christian Fontenot, Devesh Tiwari, Jian Li, Hong Zhang, Hao Wang, and Seung-Jong Park. Rainbowcake: Mitigating cold-starts in serverless with layer-wise container caching and sharing. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS '24, page 335–350, New York, NY, USA, 2024. Association for Computing Machinery.
[110] Minchen Yu, Ao Wang, Dong Chen, Haoxuan Yu, Xiaonan Luo, Zhuohao Li, Wei Wang, Ruichuan Chen, Dapeng Nie, and Haoran Yang. Faaswap: Slo-aware, gpu-efficient serverless inference via model swapping, 2024.
[111] Minchen Yu, Rui Yang, Chaobo Jia, Zhaoyuan Su, Sheng Yao, Tingfeng Lan, Yuchen Yang, Yue Cheng, Wei Wang, Ao Wang, and Ruichuan Chen. λscale: Enabling fast scaling for serverless large language model inference, 2025.
[112] Wei Zhao, Anand Jayarajan, and Gennady Pekhimenko. Tally: Non-intrusive performance isolation for concurrent deep learning workloads. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS '25, page 1052–1068, New York, NY, USA, 2025. Association for Computing Machinery.

A Artifact Appendix

A.1 Abstract

This section presents supplementary materials and guidance intended to support the reproducibility of NotebookOS and its experimental evaluation. Provided resources include the source code, dataset/model names, config files, and tools used in the experimental framework. Comprehensive instructions for deployment and carrying out the experiments are available within the linked GitHub repositories [77, 78].

A.2 Artifact Check-List (Meta-Information)

- **Model:** `VGG-16, ResNet-18, Inception v3, BERT, GPT-2, Deep Speech 2.`
- **Data set:** CIFAR-10, CIFAR-100, Tiny ImageNet, IMDb Large Movie Reviews, CoLA, LibriSpeech.
- **Run-time environment:** AWS, Linux, Ubuntu, WSL 2.
- **Hardware:** AWS EC2 virtual machines.
- **Metrics:** Interactivity delay, task completion time, number of GPUs provisioned, number of GPUs actively used, profit, profit margin, GPU-hours saved.
- **Output:** Numerical statistics.
- **Experiments:** Execution of real-world workload trace excerpt on prototype implementation, extension of real-world experiments using simulation study.
- **How much disk space is required (approximately)?:** 10's of GB across multiple virtual machines.
- **How much time is needed to prepare workflow (approximately)?:** Several hours.
- **How much time is needed to complete experiments (approximately)?:** Several hours.
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** Pending.
- **Archived (provide DOI)?:** 10.5281/zenodo.15832098.

A.3 Description

A.3.1 How to Access.

NotebookOS can be deployed using a set of Ansible [3] playbooks and shell scripts that we have provided. These tools support deployment across an arbitrary number of AWS EC2 virtual machines, allowing users to customize the scale of their infrastructure. The latest versions of the Ansible playbooks are available in the NotebookOS GitHub repository.

A.3.2 Hardware and Software Dependencies.

NotebookOS was developed, tested, and evaluated on WSL 2 (Windows Subsystem for Linux 2) and Ubuntu on AWS EC2. The WSL 2 environment used Ubuntu 22.04.5 LTS (Jammy) with Docker 27.2.0, Go 1.22.9 (`linux/amd64`), Python 3.12.6, Protoc 27.2, and CUDA 12.7 (via NVIDIA driver v566.36, NVML v565.77, and NVIDIA-SMI v565.77.01). The host OS was Windows 10 v22H2 (OS build 19045.5487), with WSL 2 kernel version 5.15.167.4-1 and WSLg version 1.0.65. The AWS EC2 environment ran Ubuntu 24.04.1 LTS (Noble) with Docker 27.3.1 (build ce12230) with matching versions of Golang, Python, and Protoc. Currently, the `glibc` version in the Global and Local Scheduler Docker containers is `ldd (Debian GLIBC 2.36-9+deb12u7) 2.36`; within the Jupyter Docker container, it is `ldd (Ubuntu GLIBC 2.35-0ubuntu3.8) 2.35`.

Remote Storage. NotebookOS persists intermediate workload data to remote storage and currently supports AWS S3 (recommended), Redis and HDFS. For S3, one must create an AWS IAM role with access to the necessary AWS S3 buckets. When provisioning the AWS EC2 virtual machines, be sure that they are assigned an AWS IAM role with access to the AWS S3 buckets intended for use by the distributed kernels.

A.3.3 Models and Datasets.

To ensure a thorough evaluation of NotebookOS and the various other baselines, we integrated support for a variety of different deep learning models and datasets across several different application domains. Table 1 displays the complete set of models and datasets.

A.4 Installation

NotebookOS is deployed on AWS EC2 instances using provided Ansible playbooks (in `setup/ansible` [78]). After provisioning VMs with IAM roles granting S3 access (see the AWS documentation), the playbooks automate dependency installation, patching, and experiment orchestration across the cluster. The source code, as well as additional documentation, is available in the following GitHub repositories:

1. NotebookOS source code (primary artifact).
2. Administrative dashboard and workload orchestrator for NotebookOS.

A.4.1 Multi-Node Deployments.

The recommended way to deploy NotebookOS across multiple AWS EC2 virtual machines is using the provided Ansible playbooks (located in the `setup/ansible` directory). Before running any playbooks, there are a few configuration-related steps that must be performed. First, create a file called "`all.yaml`" in the `setup/ansible/group_vars` directory. The "`all.template.yaml`" file is provided as a starting point. There are five configuration parameters that must be specified explicitly:

- **`ansible_ssh_private_key_file`**: Path to a private SSH key on the computer used to run the Ansible playbook(s). Used to enable access to the other VMs in the NotebookOS cluster.
- **`private_key_to_upload`**: Path to a private SSH key on the computer used to run the Ansible playbook(s). This SSH key will be uploaded to the VMs in the NotebookOS cluster to enable SSH connectivity between them. This is useful because you may want to run some scripts or Ansible playbooks from one of the VMs once NotebookOS has been deployed.
- **`public_key_to_upload`**: Path to a public SSH key on the computer used to run the Ansible playbook(s). This SSH key will be uploaded to the VMs in the NotebookOS cluster to enable SSH connectivity between them. This is useful because you may want to run some

scripts or Ansible playbooks from one of the VMs once NotebookOS has been deployed.

- **gitbranch**: the branch of the NotebookOS GitHub repository to use when deploying NotebookOS. You can default to using "main".
- **git_personal_access_token**: GitHub personal access token (PAT) with read access to the NotebookOS GitHub repository (or the fork of the NotebookOS source code that is being used).

To use Docker images built manually (rather than the provided images), there are several configuration parameters that must be changed. Please consult the documentation in the GitHub repository for additional details.

Once the "`all.yaml`" file has been created in the correct directory (i.e., "`setup/ansible/group_vars`"), you can begin deploying NotebookOS. First, run the playbook to create the Docker Swarm cluster:

```
$ ansible-playbook -i inventory_file.ini create\_docker
    \_swarm\_cluster.yaml --tags ``swarm''
```

Next, deploy the Traefik [99] Docker Stack onto the Docker Swarm cluster. Traefik is an open source reverse proxy and ingress controller that NotebookOS uses to route external web traffic to the appropriate internal component. Traefik can be deployed onto the Docker Swarm cluster by executing the following command:

```
$ ansible-playbook -i inventory_file.ini redeploy\
    _traefik\_docker\_stack.yaml
```

Finally, deploy the NotebookOS Docker Stack onto the Docker Swarm cluster:

```
$ ansible-playbook -i inventory_file.ini
    deploy_distributed_notebook\_docker\_stack.yaml
```

If desired, verbose Ansible logging can be enabled by setting the `ANSIBLE_STDOUT_CALLBACK` environment variable to "`debug`". For example:

```
$ ANSIBLE_STDOUT_CALLBACK=debug ansible-playbook -i
    inventory_file.ini deploy\_distributed\_notebook\
    _docker\_stack.yaml
```

A.4.2 Single-Node and Development Deployments.

For development, NotebookOS supports Docker Compose [22]. Run `setup/install.sh` to install host dependencies. A template for the Docker Compose `yml` file is provided in the `deploy/docker-WSL2/` directory; generate it via the "generate-docker-compose-file.sh" script. Helper scripts and usage details are documented in the directory's README. Once the `docker-compose.yml` file is generated, you can deploy NotebookOS via Docker Compose using the following command:

```
$ docker compose up -d --build --scale daemon=4
```

In order for NotebookOS to operate correctly with 3 replicas per distributed kernel, the minimum value of the "`–scale daemon=`" argument is 3; however, the recommended minimum is 4 to enable kernel replica migrations to occur.

When deploying NotebookOS for development, we recommend specifying a few additional configuration parameters. First, the system expects that the core dumps will be written to a `/cores` directory. We recommend mounting a Docker volume so that data written to the `/cores` directory are persisted and can be accessed later. We also recommend deploying Dozzle [24] for easier log monitoring.

```
$ docker run --name dozzle -d \
  -v /var/run/docker.sock:/var/run/docker.sock:ro \
  -p 7744:8080 amir20/dozzle:latest
```

A.4.3 Running the Dashboard Locally or Independently.

After cloning the dashboard repository, execute the following command from within the `driver-frontend` directory:

```
$ npm install && npm run start:dev
```

Note that unless the above command is modified, the frontend server will become the active process for the terminal.

Next, to run the backend server, execute the following command from within the `driver-backend` directory:

```
$ make run-server
```

Note that unless the avove command is modified, the backend server will become the active process for the terminal.

A.4.4 AWS Configuration.

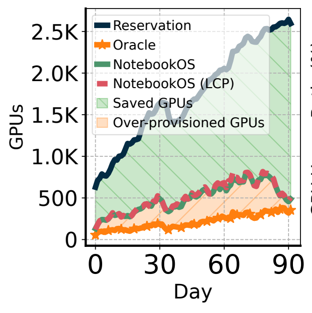
To deploy NotebookOS on AWS, designate one internet-accessible EC2 VM with a public IPv4 address as the "primary VM". This VM, which should be named "`Jupyter NaaS Leader`", hosts NotebookOS's Traefik service, which routes external traffic (e.g., to the workload dashboard). Use the `generate_inventory_file_aws.py` file in the `setup/ansible/` directory to generate an `inventory.ini` from `inventory_file.template.ini`. The script requires the primary VM to be named as above, and all other VMs in the NotebookOS cluster should be tagged with key "`swarm`" and value "`follower`". It assumes Ubuntu VMs with the "`ubuntu`" root username.

A.5 Experiment Workflow

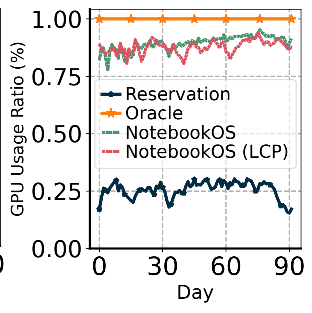
The experiments for NotebookOS and the other baselines are primarily orchestrated using NotebookOS's administrative dashboard [77]. The dashboard provides a browser-based interface to view and manage NotebookOS itself, as well as workloads running on NotebookOS.

A.6 Evaluation and Expected Results

The results presented in Section 5 correspond to the execution of a real-world workload trace on NotebookOS. We are working to open-source our workload trace. Executing the same workload on NotebookOS multiple times will generate approximately the same results, with small differences resulting from scheduling decisions and other random factors.



(a) Cluster-wide allocatable GPUs.



(b) GPU usage ratio.

Figure 14. Simulated GPU usage timeline over a 90-day snippet of the AdobeTrace spanning June through August.

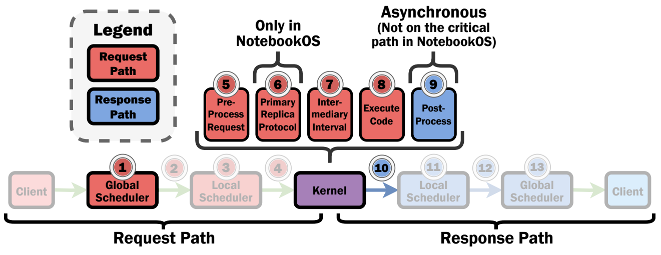


Figure 15. Workflow steps of notebook cell execution requests in NotebookOS and other baselines.

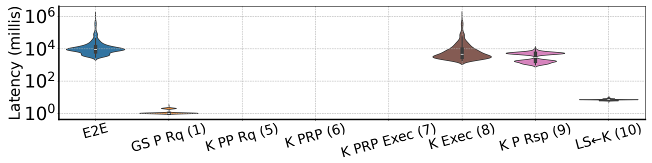


Figure 16. Detailed end-to-end latency breakdown of execute requests messages processed by Reservation.

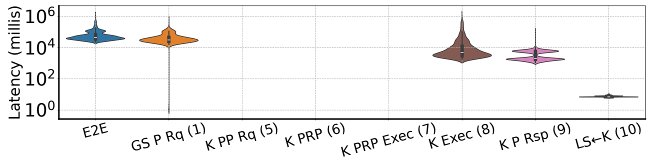


Figure 17. Detailed end-to-end latency breakdown of execute requests messages processed by Batch.

B Limited GPU Resource Availability

The demand for GPUs has surged in recent years, driven by the rapid growth of machine learning, deep learning, and high-performance computing applications [60, 90]. However, cloud providers such as Amazon Web Services (AWS), Google Cloud, and Microsoft Azure are struggling to meet this increasing demand due to limited GPU resources [21, 60, 97, 108]. This scarcity is further compounded by global supply chain challenges and the growing competition from sectors like gaming, cryptocurrency mining, and scientific research, all of which intensify the pressure on cloud infrastructure providers to allocate GPU resources efficiently [37, 60, 98]. The low availability of GPUs serves as further motivation of NotebookOS.

C Implementation

To demonstrate NotebookOS's efficacy, we implemented a fully functional prototype Jupyter Notebook platform. NotebookOS reuses the existing IPython messaging protocol used by standard Jupyter clients and kernels, ensuring compatibility with all Jupyter clients. NotebookOS uses the Jupyter Server from the official `base-notebook` Docker image. NotebookOS uses a custom kernel provisioner, referred to as the `GatewayProvisioner`, as well as a custom `SessionManager` and `KernelManager`. All custom components were implemented using the official Jupyter Server API extension methods, building on the default implementations provided by Jupyter Server. NotebookOS also implements a custom Jupyter Kernel Provisioner, `GatewayProvisioner` to integrate directly with standard/vanilla Jupyter deployments. Jupyter Kernel Provisioners enable third parties to manage the life-cycle of a kernel's runtime environment. NotebookOS's Global Scheduler and Local Schedulers communicate using gRPC, while kernel-replica-related messages use ZMQ.

Table 2. Lines of code required for the development and evaluation of NotebookOS by its components.

Component	LoC	Component	LoC
Simulator (Go/Python)	40,395	Prototype (Go/Python)	175,239
Cluster Dashboard (TypeScript)	23,038	Misc. scripts	4,322
Workload Orchestrator (Go)	43,043	**Total:**	285,037

We have implemented NotebookOS in approximately 282k lines of code over roughly two person-years. See Table 2 for a summary of our implementation efforts.

D Simulation Study: Resource Usage

Our simulation study of the 3-month trace confirms the significantly improved GPU efficiency over Reservation (Figure 14(b)). This is because NotebookOS oversubscribes server resources and consequently requires fewer servers to be provisioned. Figure 14(b) plots the ratio of allocatable GPUs that are actively utilized by kernel containers during the simulated 3-month workload. NotebookOS uses a significantly higher fraction of the available GPUs compared to the `reservation` baseline, again illustrating NotebookOS's resource efficiency.

E Detailed Latency & Overhead Breakdown

Figure 15 presents an overview of the individual steps along the critical path of execution requests within NotebookOS and other baselines. Some steps are unique to NotebookOS and do not exist in other baselines, and therefore, they have zero delay in other baselines.

Step ❶ corresponds to the processing performed by the Global Scheduler when an execution request is received. This may include on-demand docker container provisioning, queuing delays, and the overhead associated with making

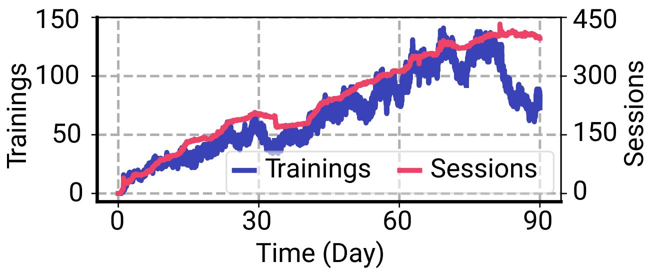


Figure 20. The number of active user-submitted trainings and active user sessions during the full summer portion (i.e., June, July, and August) of the Adobe research cluster workload trace.

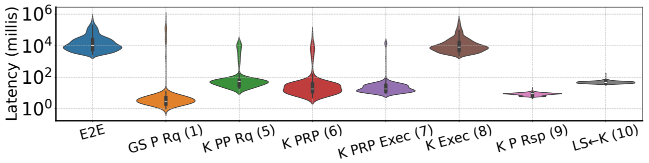


Figure 18. Detailed end-to-end latency breakdown of execute requests messages processed by NotebookOS.

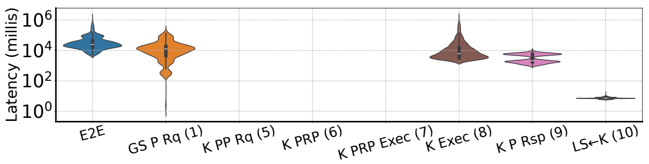


Figure 19. Detailed end-to-end latency breakdown of execute requests messages processed by NotebookOS (LCP).

placement decisions. Step ❷ is the latency of the network hop between the Global Scheduler and the Local Scheduler. Step ❸ is any processing performed by the Local Scheduler. Step ❹ is the latency of the network hop between the Local Scheduler and the kernel replica. Step ❺ includes pre-processing performed by the kernel replica, which may involve extracting metadata. Step ❻, which only occurs while using NotebookOS, is the executor replica selection protocol. Step ❼ includes the time between the selection of the executor replica and the beginning of code execution, and Step ❽ is the execution of the user-submitted cell task by the kernel replica. Step ❾ involves post-processing by the kernel replica, which in NotebookOS may include state synchronization via Raft SMR and writing large objects to the Distributed Data Store. In NotebookOS, this step is *asynchronous* and does not affect the user experience. The remaining steps largely just involve forwarding the response back to the client. Steps that appear more lightly-colored in Figure 15 are omitted from the detailed request latency figures discussed next. These steps are omitted because their latency is near zero for all baselines.

Figure 16 presents a detailed breakdown and overview of the steps along the critical path of execution requests as observed by the Reservation baseline. Execution requests processed by the Reservation baseline spend the most time in Step ❽, which corresponds to the kernel replica executing the user-submitted code. Step 9 also incurs some latency, as the kernel replica persists the updated state to remote storage during this step.

Figure 17 presents a detailed breakdown and overview of the steps along the critical path of execution requests as observed by the Batch baseline. Execution requests processed by the Batch baseline encounter significant delays during Step ❶, which includes queuing delays and on-demand docker container provisioning. The same is true for NotebookOS (LCP), though NotebookOS (LCP) observes shorter latencies during this step because many requests can simply be processed by an existing, pre-warmed container.

Figure 18 presents a detailed breakdown and overview of the steps along the critical path of execution requests as observed by NotebookOS. NotebookOS incurs slightly increased overhead in many steps compared to the other baselines; however, the small amount of additional overhead incurred in these steps does not outweigh the savings (in terms of interactivity and task completion time) enabled by NotebookOS. NotebookOS primarily incurs additional overhead when execution requests are being processed by the kernel replicas. This is because the executor replica selection protocol must be performed (Step ❻ in Figure 15). This protocol typically takes tens of milliseconds at most, however, and so it does not contribute significantly to the overall end-to-end latency.

E.1 Active Sessions & Training Events

Figure 20 presents a timeline plot showing the number of active sessions during the "summer portion" of the Adobe research cluster workload trace. Specifically, the number of active sessions is plotted on the secondary (i.e., right) y-axis. The summer portion includes data from the entirety of the months of June, July, and August. Initially, there are 0 active user sessions. By the end of June, July, and August, there are 206, 312, and 397 active user sessions, respectively. For June, the mean and median number of active user sessions are 115 and 130, respectively. For July, the mean and median number of active user sessions are 233 and 234, respectively. Finally, for August, the mean and median number of active user sessions are 379 and 385, respectively. The maximum number of active sessions at any given time throughout the summer Adobe research cluster workload trace is 433.

Figure 20 also presents a timeline plot showing the number of user-submitted training events being processed during the "summer portion" of the Adobe research cluster workload trace. Specifically, the number of active sessions is plotted on the primary (i.e., left) y-axis. By the end of June, July, and August, there are 58, 83, and 73 active user-submitted trainings, respectively. The mean and median number of active user-submitted trainings for the entire summer are 67.63 and 68, respectively. For June, the mean and median number of active user-submitted trainings are 31 and 30, respectively. For July, the mean and median number of active user-submitted trainings are 65 and 67, respectively. Finally, for August, the mean and median number of active user-submitted trainings are 105 and 108, respectively. The maximum number of active user-submitted trainings at any given time throughout the summer AdobeTrace is 141.