

# Machine Learning Fleet Efficiency: Analyzing and Optimizing Large-Scale Google TPU Systems with ML Productivity Goodput

*A Preprint*

Arissa Wongpanich<sup>†</sup> Tayo Oguntebi<sup>†</sup> Jose Baiocchi Paredes<sup>†</sup>  
 Yu Emma Wang<sup>†</sup> Phitchaya Mangpo Phothilimthana<sup>†</sup> Ritwika Mitra<sup>†</sup>  
 Zongwei Zhou<sup>†</sup> Naveen Kumar<sup>†</sup> Vijay Janapa Reddi<sup>†‡</sup>

<sup>†</sup>Google

<sup>‡</sup>Harvard University

## Abstract

Machine learning (ML) infrastructures operating at warehouse scale present unique performance characterization challenges beyond traditional high-performance computing metrics. This paper introduces a systematic framework for analyzing ML fleet efficiency, demonstrated on Google’s production TPU infrastructure comprising thousands of accelerators running diverse workloads. Our fleet-wide analysis reveals performance dependencies spanning the entire ML system stack, from hardware to model architecture, data pipelines, frameworks, compilers, and schedulers. We identify critical gaps in conventional utilization-based performance metrics and propose “ML Productivity Goodput” (MPG) to capture fleet-wide efficiency across heterogeneous ML environments. Using MPG, we provide different methods for identifying and addressing bottlenecks throughout the ML system stack. When applied to Google’s TPU workloads that are used in production, our results show that the methodology yields substantial performance improvements, and establishes a basis for efficient management of large-scale ML computing infrastructure.

## 1 Introduction

As machine learning (ML) models become larger and more complex, production fleets must deploy highly parallel training [61] and low-latency inference [49] applications at unprecedented scale. Much like how the boom of internet-scale services [22] prompted the development of the warehouse-scale computer (WSC), [8, 31], the current explosion of foundation ML models [5, 12, 58] is ushering in the era of the ML fleet. ML fleets, sometimes referred to as “AI hypercomputers” [59], represent a new paradigm in computer architecture, where domain-specific accelerators (DSAs) work in tandem with a modular ML system stack to achieve significant performance and energy efficiency improvements. They are characterized by their massive scale and ML-centric workloads, often requiring more compute power, more storage, and more complex networking than traditional WSCs. Despite the rapid development of these ML fleets, the challenges associated with building and operating them at scale remain significant and poorly understood in the literature.

This paper presents a playbook for instrumenting and optimizing large-scale ML fleets, using Google’s production Tensor Processing Unit (TPU) infrastructure for internal workloads as our case

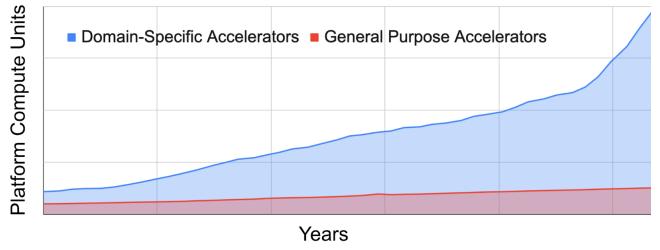


Figure 1: Five-year historical ML fleet breakdown by accelerator type. The rapid proliferation of domain-specific accelerators in response to ML-based workloads has presented novel challenges in optimizing ML fleets. Managing these domain-specific accelerators means effectively handling hardware and workload heterogeneity, as well as hardware-software co-design at scale.

study. In contrast to prior work, where previous Google TPU papers have predominantly focused on hardware architecture design and chip-level performance evaluations [27–29], our work provides the first in-depth perspective on the software stack that enables these domain-specific accelerators to operate efficiently at scale.

This shift from hardware-centric discussions to the practical aspects of operationalization is important in understanding how to leverage and manage ML infrastructure in real-world, production environments. Although our study centers on Google’s ML fleet, which is primarily composed of TPUs, the insights, methodologies, and best practices we present are broadly applicable to ML fleets consisting of other DSAs such as GPUs.

The contributions of this paper are listed as follows. ① First, we provide a methodology to dissect the ML fleet by segmenting it along the layers of a system stack, from the low-level hardware layer to the user-facing application layer. This analysis reveals three major challenges for ML fleet optimization: hardware heterogeneity, workload heterogeneity, and hardware/software co-design. Traditional optimization strategies, which often rely on stable hardware and workload characteristics, are inadequate for addressing these dynamic interactions. Architecture-centric metrics, such as TOPs/Watt or peak FLOPS, while valuable for specific hardware evaluations, fall short in capturing the overall complexity and efficiency of an ML fleet.

② Our second contribution addresses these issues. To view the fleet holistically and capture its performance across multiple layers of the ML system stack, we introduce the “ML Productivity Goodput” (MPG) metric. Unlike traditional metrics, MPG encompasses scheduling efficiency, runtime efficiency, and compiler / program

\*Corresponding author: arissa@google.com

**Table 1: Comparison of Machine Learning (ML) Fleet, Warehouse Scale Computer (WSC), and High-performance Computing (HPC).**

Category	Warehouse Scale Computer	High-Performance Computing	Machine Learning Fleet
Workload types	Diverse web services (search, email, social networking, media streaming)	Scientific simulations, graph computations, solvers	Training of ML models, real-time serving, bulk inference
Fleet composition	More stable, as most user demand has reached a steady state or known patterns	A large portion of demand is predetermined, as it is driven by scientific missions	Rapidly changing due to newly emerging ML models and increasing user demand
Hardware heterogeneity	General-purpose CPUs	CPUs, GPUs, other ASICs	CPUs, GPUs, TPUs, FPGAs, other ASICs.
Hardware/Software co-design	Hardware is workload-agnostic	Hardware is chosen for specialized applications	ASICs often co-designed with workloads in mind

$$\text{ML Productivity Goodput (MPG)} = \text{Scheduling Goodput (SG)} \times \text{Runtime Goodput (RG)} \times \text{Program Goodput (PG)}$$

↓                      ↓                      ↓  
 How often does    When an ML    When an ML  
 an ML               application has    application is  
 application have    all necessary    making  
 all necessary       resources, how    progress, how  
 resources to       often is it       close is it to the  
 make progress?       making       ideal roofline?

**Figure 2: ML Productivity Goodput (MPG) and its components.**

efficiency. MPG is analogous to the “iron law” of computer performance [16], but adapted for ML fleets as shown in Figure 2. The MPG metric enables us to identify areas for improvement across the entire machine learning fleet by analyzing performance across different segments, such as accelerator type, model architecture, and workload phase (e.g., training vs. serving). By breaking down the metric into subcomponents—Scheduling Goodput (SG), Runtime Goodput (RG), and Program Goodput (PG)—and examining it along the axes of fleet characteristics, we can precisely pinpoint optimization opportunities.

③ Finally, we provide a procedure for leveraging the MPG metric and show strategies that we deployed in the real world for optimizing ML fleet performance with the help of this metric. We show how MPG helps us identify compiler optimizations to generate more efficient code for specific hardware configurations, and improve scheduling algorithms to better utilize heterogeneous resources within the fleet, along with a few other strategies. Even after optimizations are implemented in the fleet, the MPG metric can be used to validate and track fleet-wide improvements.

These contributions collectively provide a basis for understanding and optimizing large-scale ML infrastructure, thereby paving the way for more efficient, sustainable, and cost-effective deployment of ML workloads across diverse computing environments.

## 2 Background

We introduce the concept of an ML fleet, a specialized form of computing infrastructure designed specifically for ML workloads at scale. To help set the stage for an ML fleet, we compare ML fleets with two related large-scale computing paradigms: traditional warehouse-scale computers (WSC) [8] and high-performance computing (HPC) supercomputers [6]. An ML fleet can be considered an evolution of a WSC, adapted for high-performance ML workloads.

Table 1 summarizes the key features of each system type, highlighting their distinct characteristics. We examine these features to better understand how the unique demands of ML workloads have shaped the design and optimization strategies of ML fleets, setting them apart from their predecessors in large-scale computing.

## 2.1 Workloads

WSCs [8] traditionally focus on Internet-scale services, which serve web content, process user queries, and handle data storage and retrieval for billions of users. WSCs must manage diverse, often latency-sensitive tasks such as search indexing [11], social media feeds [44], and transaction processing services. The workloads are characterized by their burstiness and high concurrency, with millions of small, independent tasks that require fast response times.

High-performance computing, on the other hand, has focused on simulation models for research, such as climate modeling [40], and molecular dynamics for drug discovery [53]. These workloads involve complex, tightly-coupled computations that require both significant processing power and high-bandwidth, low-latency communication between nodes [56]. They operate on large datasets and require sustained performance over long periods of execution.

ML fleets combine the scale of WSC workloads with the computational intensity of HPC workloads. They consist of workloads which are focused on running deep neural networks [32], such as recommendation models [42, 68] for e-commerce and social media, computer vision tasks for autonomous vehicles [25] and medical imaging [18], and natural language processing (NLP) applications like large language models for chatbots and translation services [67]. ML workloads involve heavy matrix-matrix [19] or matrix-vector [9] operations and parallel processing that often require high computational throughput and memory bandwidth. ML workloads also tend to be more communication-bound rather than compute-bound [36], especially since large models are often sharded across many chips and require large datasets to be fed as input. ML fleets also face another key challenge, which is rapidly fluctuating user demand and huge tectonic shifts in ML model architectures, as discussed in Section 1. As a result, the designers of an ML fleet need to trade off specialization with fungibility to adapt quickly while still delivering peak performance.

## 2.2 Hardware

The hardware composition of these systems also reflects the demands of its users. WSCs usually rely on general-purpose commodity CPUs [8], balancing diverse computational needs for web services, and scaling massively to accommodate varying user demands [50]. HPC supercomputers, designed for complex scientific calculations, utilize specialized vector processors [45] and accelerators to achieve extreme performance for tightly-coupled computations. An ML fleet must utilize hardware that can provide both the scalability of a WSC and the performance intensity of a HPC supercomputer. They are mostly comprised of accelerators such as GPUs, TPUs, and/or other ML application-specific integrated circuits (ASICs)

that are optimized for the compute-intensive parallel processing demands of machine learning tasks. They also make significant use of general-purpose chips like CPUs for hosting or scheduling tasks.

Memory and storage architectures are also highly tailored to their specific workloads' needs. ML fleets typically employ high-throughput, local memory systems coupled with ML-optimized SSD storage, which facilitates fast data access and parallelism for iterative learning algorithms [34]. WSCs, on the other hand, utilize distributed, commodity-based memory and storage systems, prioritizing cost-effectiveness and redundancy for handling diverse, large-scale web services [8]. HPC supercomputers feature low-latency, parallel-optimized memory architectures and parallel file systems, enabling efficient processing of massive scientific datasets. These distinctions in memory and storage design influence each system's data handling capabilities: WSCs ensure robust, scalable data management for varied web applications; HPC systems focus on minimizing latency for complex, data-intensive scientific computations; and ML fleets optimize for repeated, high-throughput access to training data and weight updates.

### 2.3 System Efficiency

Network infrastructure and job scheduling in ML fleets, WSCs, and HPC supercomputers are also optimized for their respective workloads. WSCs employ distribution-optimized networks to handle diverse, geographically dispersed web traffic, using cloud-based or custom schedulers [13] designed for varied, often short-lived tasks. HPC supercomputers, however, feature ultra-low latency networks crucial for tightly-coupled parallel computations, alongside specialized job schedulers like SLURM [66] that manage complex, long-running scientific workloads. ML fleets prioritize high-bandwidth networks [69] to facilitate rapid data movement for distributed training, coupled with job schedulers like Borg [62], Kubernetes [51] and MAST [14] that efficiently manage GPU/TPU resources. These differences in network and scheduling approaches directly impact each system's ability to handle its target applications.

Efficiency considerations and workloads vary across ML fleets, WSCs, and HPC supercomputers. ML fleets exhibit workload dependent energy efficiency, with high initial costs due to model training, but improved efficiency with tasks like bulk inference. WSCs prioritize high energy efficiency at scale and low per-unit costs, despite high aggregate expenses, handling diverse web services [15] and cloud computing workloads. Meanwhile, HPC supercomputers focus on maximum performance for tightly-coupled scientific simulations, often at the expense of energy efficiency [30].

These distinctions in efficiency and workload optimization reflect each system's primary purpose: ML fleets are designed for flexible, scalable AI computation; WSCs for cost-effective, large-scale web services; and HPC systems for pushing the boundaries of computational performance in scientific research. The trade-offs between cost, energy efficiency, and performance in each system are carefully balanced to best serve their intended applications. In particular, if we wish to measure the performance of the ML fleet and identify opportunities for further optimization, we must study its characteristics across all levels of the system stack.

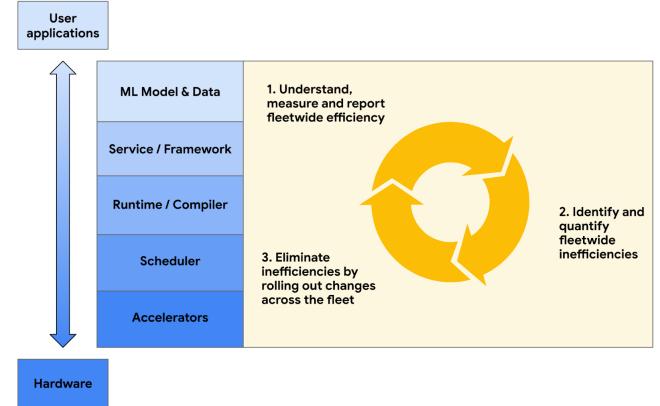


Figure 3: The ML fleet system stack of a production system at Google. The multi-layered architecture of a fleet is complex; each layer is a critical component in the ML system, with interactions between layers affecting overall performance and efficiency. Segmenting the fleet based on these layers provides actionable metrics which can be used to improve performance.

## 3 Anatomy of an ML Fleet

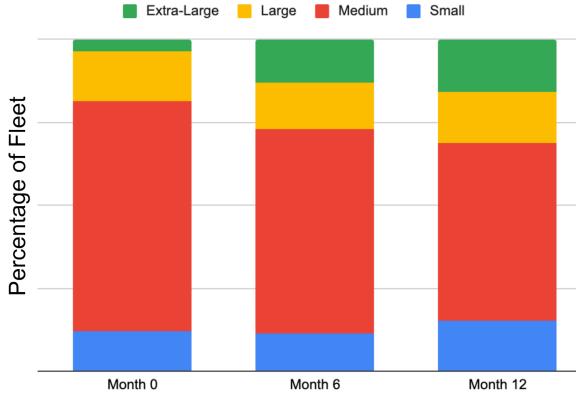
We begin by dissecting Google's user-facing production ML fleet, analyzing its components from hardware foundation to application level. Figure 3 shows the layers comprising the system stack that mediate user access. We supplement our discussion with data from a snapshot of Google's TPU fleet for internal workloads, providing concrete examples of challenges these systems face.

### 3.1 Accelerators

ML fleets are distinguished from other types of large-scale systems by their accelerator-centric architecture. The ML computing landscape is dominated by domain-specific hardware, such as GPUs and other ASICs. In order to tailor to the vector and matrix intensive operations that underpin ML workloads, there has been a Cambrian explosion of ML hardware accelerators [23], with new accelerators being deployed at an unprecedented rate compared to traditional WSC fleets [27]. Figure 1 illustrates this dynamism, showing dramatic shifts in our ML fleet's hardware makeup for internal workloads over just a few years.

ML fleets typically incorporate a diverse array of hardware including CPUs, GPUs, TPUs, and other accelerators, each fulfilling specific roles. For example, CPUs may be responsible for scheduling, GPUs for training tasks, and edge accelerators [65] for deployment and serving. The challenge lies in effectively orchestrating these heterogeneous accelerators to maximize their individual strengths—a complexity rarely encountered in general compute fleets.

Moreover, the heterogeneity extends beyond just accelerator type. Even within a single class of hardware accelerators, there are many different versions of the hardware, adding another layer of complexity to fleet management. Each hardware generation introduces unique features that require significant optimizations to extract peak ML workload efficiency. One notable example is the integration of the SparseCore (SC) in TPUs [28], which was designed to significantly boost performance for embedding-heavy models. Subsequent large-embedding model teams would likely then consider the hardware specifications of the SparseCore when



**Figure 4:** A sample breakdown of Google’s ML fleet for internal workloads, segmenting on workload topology size (the number of accelerators requested by a given job). Progressive snapshots over the course of one year illustrate the ML fleet’s growing share of jobs using an “extra-large” number of accelerators. This demonstrates how an ML fleet scheduler must be able to adapt to changing conditions, as the evolution of job sizes and topologies in response to shifting ML workloads presents unique challenges for the entire fleet.

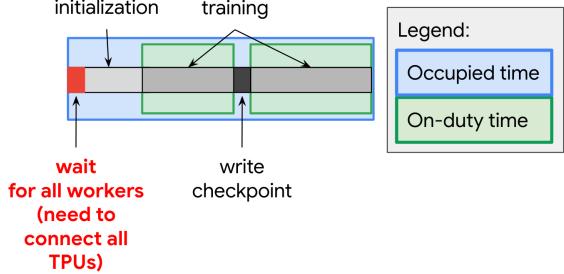
designing their embedding configurations. Model design configurations such as embedding dimension, vocabulary size, valence, and others might also be co-designed to optimize performance on the hardware platform. This shows how hardware-software co-design is becoming increasingly important in improving the efficiency of these diverse accelerators, forming a symbiotic relationship where the computational needs of future workloads affect the next generation of hardware, and the hardware capabilities inform the types of workloads that the ML fleet is best equipped to handle [55].

### 3.2 Scheduler

The scheduler directly manages the hardware in a fleet by coordinating the allocation of resources. For the case study presented in this paper, it coordinates TPU allocations for Google’s internal-facing ML workloads. There are two interconnected challenges that a scheduler must address when allocating hardware for an ML fleet: (1) optimizing performance across various hardware types, and (2) balancing utilization with stability and fault tolerance.

Figure 4 illustrates these challenges. It shows the allocation of workloads in Google’s internal-facing ML fleet with different chip requirements over time, categorized into sizes based on the total number of TPU chips in the required topology. In this categorization, workloads with size “small” refer to jobs that request a single TPU or a handful of TPUs, while workloads with size “extra-large” refer to jobs that request the largest number of TPUs (often requiring multiple pods, as described in Kumar et al. [34]). Figure 4 demonstrates that over the course of just one year, the allocation distribution can shift dramatically, reflecting the changing nature of ML workloads in the fleet. As large-scale ML models become more prevalent in an ML fleet, an increasing number of workloads will require correspondingly larger meshes of connected accelerators.

Optimizing the scheduling of jobs while meeting these resource requirements is difficult because it presents an NP-hard bin-packing problem. Each workload may specify a different accelerator type, chip topology, and location requirement and needs to be scheduled



**Figure 5:** An ML workload requires all requested TPUs to be allocated before the task can start. In this example of a training workload, forward progress is saved via checkpoints. Delays during workload initialization and checkpoint writing, which are part of the Runtime and Framework layers, can reduce overall system efficiency.

according to fleet constraints in a way that reduces overall fragmentation of the fleet. Since workloads are constantly being started and completed, the machine availability of the fleet is constantly changing, requiring a robust defragmentation algorithm. In addition, latency requirements may require accelerators for a workload to be grouped together near certain locations or data cells, adding another constraint to the scheduling optimization problem.

The utilization of fleet resources must also be balanced with stability and fault tolerance. For example, to reduce disruptions, some machines may intentionally remain underutilized so that higher priority jobs may be more easily scheduled when needed. While high utilization is desirable for cost-efficiency, pushing hardware to its limits can lead to thermal issues, increased failure rates, and unpredictable performance. In large-scale ML fleets, hardware failures are inevitable, and the scheduler must be robust enough to handle these failures gracefully, redistributing workloads and ensuring job continuity without significant performance degradation.

### 3.3 Runtime/Compiler

The runtime and compiler layers form an important component in the ML fleet system stack. They are responsible for bridging the gap between high-level ML models and the underlying hardware accelerators. The runtime layer focuses on the execution environment of ML programs. It handles important tasks such as program setup, data feeding, result management, and checkpoint creation, as illustrated in Figure 5. Depending on the system design, it either triggers just-in-time compilation of user-written code into accelerator-specific instructions or invokes pre-compiled operation kernels from vendor-specific libraries. The runtime layer can also manage the distribution strategy of code execution, as with notable runtimes like Pathways [7].

Figure 6 shows the growth of Pathways-based workloads in our production fleet. It highlights the demand for runtimes that support efficient distributed execution for ML workloads. It also emphasizes the rapidly shifting distribution of workload runtimes in a fleet.

The compiler layer, working with the runtime, transforms high-level ML model code into executable code optimized for specific accelerators. It operates on graph intermediate representations, applying both platform-independent and platform-dependent optimizations. The output is a program tailored to the target accelerator,



**Figure 6:** The prevalence of fleet-wide workloads using the Pathways runtime over a sample of one year, illustrating the rapid shift of fleet-wide runtimes to accommodate changing workloads. Pathways adoption has increased rapidly, as it provides better support for distributed execution and data processing.

such as a specific version of a TPU. Domain-specific compilers, like XLA (Accelerated Linear Algebra) [1], have significantly improved the performance of ML workloads. For instance, in MLPerf BERT benchmarks [38], XLA demonstrated a remarkable 7 $\times$  performance boost and 5 $\times$  batch size improvement [34] over previous records, emphasizing the potential of specialized compilation techniques. We note that there are many types of accelerators, some of which do not require an explicit compiler for code generation.

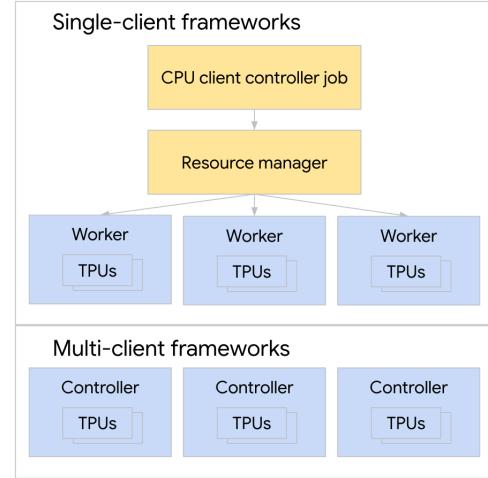
Compiler optimization in ML fleets faces unique challenges due to the rapid evolution of hardware accelerators, requiring frequent updating of optimization strategies to leverage the specific features of each new hardware generation. Moreover, the impact of optimizations can be difficult to generalize, as an optimization that improves one workload may degrade another due to differences in computation or communication patterns. This emphasizes the need for a balanced approach to optimization, considering both platform-independent techniques for flexibility and platform-specific optimizations for maximum performance.

### 3.4 Framework

The framework layer sits on top of the runtime/compiler. It is the interface between ML practitioners and the underlying complex hardware and software infrastructure. This layer encompasses various ML frameworks and libraries, such as TensorFlow [3], JAX [20], and PyTorch [47], each offering unique features and optimizations.

The framework layer provides high-level abstractions and APIs that allow developers to build and deploy ML models efficiently. These frameworks are responsible for translating user-written code into representations that can be understood and optimized by lower-level layers such as compilers and runtimes. This translation process bridges the gap between user intent and system execution.

One of the key responsibilities of ML frameworks is defining the structure of distributed ML applications. For example, TensorFlow’s Distribution Strategy [2] provides a framework for distributing training across multiple devices or machines. These can have single-client or multi-client architectures, depending on workload needs, as shown in Figure 7. These frameworks must also map ML primitives to hardware-specific designs to achieve optimal performance. This is important for specialized hardware like TPUs, which are designed for bulk-synchronous training. Frameworks like JAX



**Figure 7:** Comparing single-client frameworks with multi-client frameworks.

are more targeted towards ML workloads, with features that facilitate ease of interpretability when analyzing ML performance, such as high-level tracing for just-in-time compilation. In the ML fleet, JAX usage has increased over time, most likely due to these features and the emergence of more ML-heavy workloads [20].

Also, ML frameworks often provide auxiliary services to improve efficiency of the entire ML fleet. For instance, TensorFlow’s `tf.data` [41] service optimizes the performance of the data pipeline. These features, while abstracted from the user, can impact the overall system efficiency, as shown in Figure 5. Underneath these high-level frameworks lies a foundation of general-purpose libraries and datacenter services. Frameworks like TensorFlow utilize libraries such as gRPC [57], protobuf [60], and tcmalloc [52] for various low-level operations, and interface with datacenter services for storage (e.g., Colossus [21] [24]) and monitoring (e.g., Monarch [4]).

As the primary point of interaction for users, the framework layer serves as a key bridge in the ML system stack, as it not only abstracts underlying complexities but also plays an important role in determining the overall efficiency and capabilities of the fleet. Frameworks must balance the need for user-friendly APIs with the need to leverage underlying hardware-specific optimizations, while also managing the complexities of distributed computing, data pipeline optimization, and integration with lower-level services.

### 3.5 ML Model & Data

To characterize the ML Fleet at the highest level of the stack, we want to know: What types of workloads are we spending most of our compute cycles on? This is a critical question because it drives nearly every design decision we make for the ML Fleet at every level of the stack, from the hardware (how many training vs. inference vs. other chips) to the software (JAX vs. other frameworks, runtime distribution strategies, and compiler optimizations). Workload heterogeneity analysis is useful for understanding what kind of models are prevalent in the production fleet, especially since different workloads stress the hardware in different ways. This understanding can drive decisions about which accelerators to deploy and how many, or which compiler optimizations to carry out.

$$\text{ML Productivity Goodput (MPG)} = \frac{\text{Scheduling Goodput (SG)}}{\text{capacity}} \times \frac{\text{Runtime Goodput (RG)}}{\text{productive}} \times \frac{\text{Program Goodput (PG)}}{\text{predicted}}$$

Figure 8: ML Productivity Goodput (MPG) and its components.

In practice, we observe that the model and data layer of the ML fleet stack are the most affected by fluctuating user demands. User requirements such as the model architecture, size of the training dataset, or even use of different numerical formats in the training model can impact the efficiency of the job, which can have a cascading effect on the efficiency of the overall ML fleet.

While ML workloads share some computational patterns, particularly in their use of matrix operations and data-intensive processing, the specific architectures and resource requirements can vary significantly. As new model architectures and learning tasks emerge, they prompt rapid shifts in workload composition, leading to fluctuations in resource demands across various model types.

In a production ML fleet, there are varying proportions of workloads dedicated to each phase of the ML model life cycle; training, bulk inference, and real-time serving. Thus, the fleet must be flexible enough to handle the requirements of each of these phases; for example, training workloads may be compute intensive while real-time serving workloads may focus on minimizing latency.

## 4 ML Productivity Goodput

The optimization of ML fleet efficiency is a complex, cyclical challenge, as illustrated in Figure 3. The first challenge is measuring, understanding, and reporting fleetwide efficiency, establishing a baseline for current performance. Second, we must identify and quantify fleet-wide inefficiencies, pinpointing areas that require improvement. Third, we must eliminate these inefficiencies by implementing changes across the fleet, which in turn leads back to the first stage as we measure the impact of these changes. This cycle ensures ongoing optimization and adaptation to the ever-evolving landscape of ML workloads and hardware. To keep pace with this lifecycle, we require a metric that not only quantifies current performance but also guides optimization efforts across the fleet.

In this section, we present an in-depth discussion of MPG, a new metric for quantifying ML fleet efficiency. We refer to this as the iron law of performance for ML fleets, drawing a corollary to the iron law of processor performance [17]. MPG, defined in Figure 8, is a means for measuring efficiency gains and guiding exploration of optimization strategies across various fleet components.

### 4.1 Metric Features

Ideally, the MPG metric must be a clearly defined and accurate measure of forward progress; improvements in the metric must also reflect real improvements in the efficiency of the fleet. This metric must be capable of overcoming two significant challenges.

- (1) **It must capture the dynamic nature of ML fleets:** The fleet is constantly fluctuating due to variables such as changes

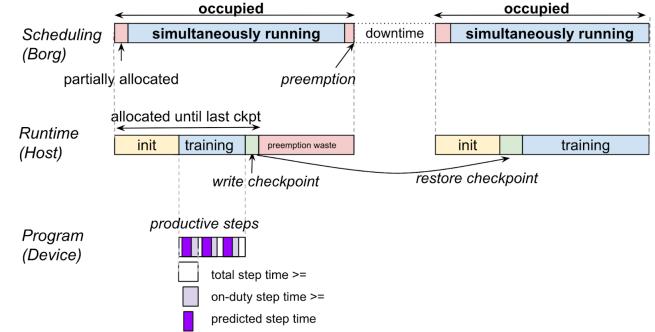


Figure 9: Breakdown of a ML workload using ML Productivity Goodput.

in workload composition, updates to the code stack, and evolving hardware. To effectively improve efficiency, we must ensure that any change in the metric is explainable despite these fluctuating variables.

- (2) **It must explain the trade-offs between individual and aggregate efficiency.** At a fleetwide scale, jobs must be scheduled in concert with one another to ensure maximum aggregate efficiency of the fleet. However, individual jobs may have certain service-level requirements, meaning that this metric must be decomposable based on workload characteristics.

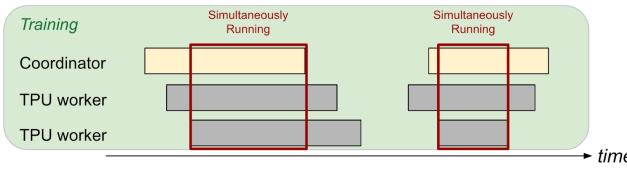
## 4.2 A New Approach: ML Productivity Goodput

ML Productivity Goodput (MPG) is designed to address the myriad challenges discussed in Section 3, as well as to overcome the limitations of existing approaches. Just as the Iron Law of Processor Performance [17] breaks down CPU performance into  $\frac{\text{instructions}}{\text{program}} \times \frac{\text{cycles}}{\text{instruction}} \times \frac{\text{time}}{\text{cycle}}$ , the MPG metric decomposes ML fleet efficiency into scheduling, runtime, and program components (see Figure 8).

This multi-layered structure, as shown in Figure 9, offers several advantages over traditional metrics. First, it allows precise identification of performance bottlenecks or improvements at specific layers of the stack, facilitates a more granular analysis of efficiency trends over time, and mitigates the risk of misleading interpretations that can arise from aggregated data. Second, by decoupling these submetrics, we enable more targeted optimization efforts and gain deeper insights into the complex interactions within the ML fleet. Finally, this approach not only enhances our ability to measure current performance but also provides a framework for guiding improvements, discussed in Section 5.

**Scheduling Goodput:** How often does an application have all necessary resources to make progress?

Scheduling Goodput (SG) quantifies the efficiency of resource allocation in an ML fleet. It measures the fraction of time that an application has all the required resources simultaneously available to make progress. This metric can be lower than traditional Occupancy metrics, particularly in distributed, bulk-synchronous applications where all required chips must be available concurrently. The numerator of SG is calculated as the simultaneous uptime of all tasks in a distributed ML application that must be connected to make synchronous progress, as shown in Figure 10. This is referred to



**Figure 10:** The scheduling goodput for training workloads measures the percentage of time when all of the TPU workers are available to work at the same time. In other words, it measures the portion of time that all of the necessary resources are available to make progress.

as “allocated chip-time” or “all-allocated” time. The denominator is fleet capacity, expressed as chip-time. This provides a full view of how effectively the scheduling layer is using the fleet’s resources.

Scheduling Goodput offers insights into potential inefficiencies in resource allocation, such as fragmentation of available resources, delays in coordinating multiple chips for distributed applications, and mismatches between application requirements and available resources. By optimizing SG, we can improve the overall efficiency of resource utilization in the ML fleet, ensuring that applications have the necessary resources to make consistent progress.

**Runtim Goodput:** *Of the time that an application has all necessary resources, how often is it making progress?*

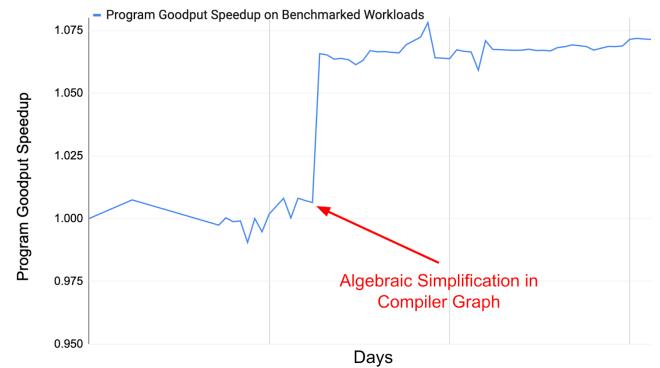
Runtim Goodput (RG) measures the efficiency of the orchestration layers in managing the execution of ML applications once resources are allocated. This metric focuses on the actual productive time of an application, accounting for various overheads in the runtime environment. The orchestration layer is responsible for critical tasks such as initializing chips, connecting them into slices for bulk-synchronous progress, loading and compiling programs, feeding data to these programs, and ensuring that training progress is regularly saved through checkpoints. The numerator of RG is the productive chip-time of the application’s progress that has been saved in checkpoints; work done between the last checkpoint and failure (or preemption) doesn’t count as “productive” time and is therefore not included in RG. The denominator of RG is the allocated chip-time defined as the numerator of SG.

Runtim Goodput can help with identifying bottlenecks in the runtime environment, such as slow data loading, inefficient checkpointing, or suboptimal program compilation. It can guide the efforts to streamline the execution pipeline and improve the overall throughput of ML workloads.

**Program Goodput:** *Of the time that an application is making progress, how close is it to the ideal roofline?*

Program Goodput (PG) assesses the efficiency of the application code itself, measuring how effectively it utilizes the available computational resources. While a traditional roofline performance model [64] might seem suitable for this purpose, it falls short in capturing the true efficiency of modern ML workloads. The traditional roofline model is highly sensitive to compiler decisions, such as how ML operators are fused or rematerialized [10], or which operands are placed in which memory space. It rewards individual ops that are close to peak utilization, but penalizes correct optimizations that result in computation graphs where the utilization may be lower, but overall execution time is shorter.

To overcome these limitations, we use a compute-based roofline model that compares the ideal execution time of the workload



**Figure 11:** This figure demonstrates the effect of an XLA algebraic simplification optimization on Program Goodput (PG) across a benchmark of the top 150 fleet workloads. Looking at the PG in this way allows us to bisect which code changes improved or regressed overall fleet efficiency.

against its actual execution time. The ideal predicted execution time, which is the numerator of PG, can be computed from intrinsic properties of the machine learning model being run. By analyzing the shape of the unoptimized high-level operations (HLO) graph, we can estimate how many floating point operations (FLOPs) the program would require at its theoretical peak performance. Since we are analyzing the computation graph before any compiler optimizations, this prediction is agnostic to compiler decisions.

The denominator of PG is the actual execution time. The PG metric can thus be interpreted as a percentage reflecting how well optimized the ML program is, with a score of 100% indicating perfect performance matching the theoretical peak.

## 5 Improving Fleetwide Efficiency

We show how ML Productivity Goodput is a robust quantifier of ML Fleet performance through optimization examples from Google’s ML Fleet in production. We present a breakdown of the MPG components using segmented fleet data and demonstrate how this procedure can help identify potential optimization techniques. Additionally, we showcase the effects of deploying these optimizations and how MPG helps verify and track performance improvements.

Looking at the aggregated MPG of the fleet does not necessarily help ML practitioners identify what kinds of improvements will make the largest impact on the fleet; this is where the decomposability of the metric comes into play, as shown in Table 2. By breaking MPG into its three components; Scheduling Goodput, Runtime Goodput, and Program Goodput, we can diagnose fleetwide issues and identify the types of optimizations that would most improve fleet efficiency. Furthermore, we can segment the fleet using the characteristics described in Section 3 in order to identify issues with specific workload types and propose model-level optimizations.

### 5.1 Scheduling Goodput Optimizations

The optimization of Scheduling Goodput (SG) can be presented as a bin packing problem, as described in subsection 3.2. Users launch workloads with varying TPU topology sizes, and the scheduling algorithm must determine how to best fit these workloads into the existing fleet of allocated chips. This process presents numerous

Table 2: Optimizing different components of ML Productivity Goodput.

ML Fleet Stack Layer	Program Goodput ×	Runtime Goodput ×	Scheduling Goodput =	Workload ML Productivity Goodput
<b>Compiler:</b>				
On-duty step time decreases	<b>Increases</b>	Decreases if device-bound Decreases if host-bound	Decreases if device-bound No change if host-bound	<b>Increases if device-bound</b> No change if host-bound
<b>Runtime:</b>				
Off-duty time or preemption waste decreases	No change	<b>Increases</b>	Decreases	<b>Increases</b>
<b>Scheduler:</b>				
Partially-allocated time decreases	No change	No change	<b>Increases</b>	<b>Increases</b>

challenges, primarily due to the wide variety of job sizes in the fleet, ranging from single-chip to multipod configurations [34].

A significant complication in job scheduling is that it requires more than mere availability in the fleet. The topology of the available hardware must also satisfy the topology requirements of the workload, which is sometimes impossible without first pre-empting other jobs. Consequently, suboptimal scheduling can have a cascading effect on other components of ML Productivity Goodput.

We can identify availability issues in the ML Fleet by looking at the SG for jobs with different chip allocation requirements, as shown in Figure 12. The data shows that the overall SG is already close to optimal, due to defragmentation techniques and scheduling optimizations. However, it is interesting to note which jobs tend to have the highest SG: extra-large jobs which require the greatest number of chips or possibly multiple TPU pods, as well as smaller jobs which require only a single chip or a few chips.

This is likely due to the way the scheduler deals with evictions; evicting extremely large jobs would have a severe negative impact on the overall ML Productivity Goodput score due to their huge startup overhead. Once the extra-large job is running, it is also immensely dependent on checkpointing and data sharding, affecting the Runtime and Program Goodput components as well. In short, evicting extra-large jobs from the hardware would present a cascading series of failures, strongly incentivizing the scheduler to reduce churn for these jobs and evict medium-sized jobs instead.

On the other hand, extremely small jobs usually do not get prematurely evicted since they are more likely to finish quickly, and if pre-empted, it is usually quicker to find topologically matching availability. With extremely small jobs, the scheduler has more flexibility to intelligently allocate the workloads to optimal compute cells in order to defragment the overall ML fleet availability. It is also important to note that for workloads of all sizes, the SG is greater than 95% due to the particular pre-emption preferences of the scheduler. The pre-emption preferences of the scheduler can be tuned, e.g. to require a SG of greater than 95% for medium-sized jobs, but this could reduce the SG for other segments of the fleet.

## 5.2 Runtime Goodput Optimizations

Outside of device time, host overhead and pre-emptions can be major bottlenecks for some workloads, and can be tracked by measuring Runtime Goodput. For example, training jobs usually use input pipelines to ingest and transform input data, which could be bottlenecks for certain models that ingest large amounts of data. Some solutions have been proposed to reduce host overhead, such as Plumber [33], a tool to find bottlenecks in ML input pipelines.

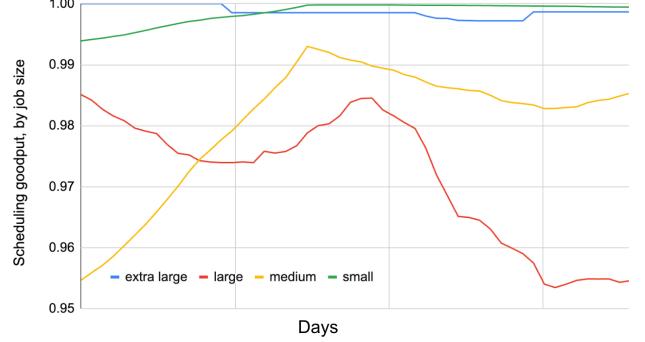
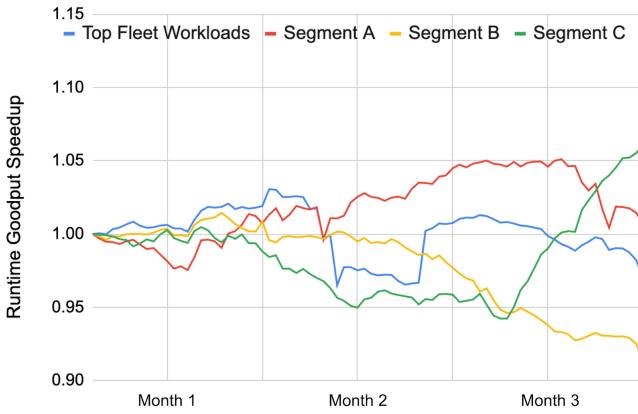


Figure 12: Scheduling goodput by job size. Extra-large and small jobs tend to have better scheduling goodput due to the scheduler’s preemption algorithm.

We can improve the RG of the ML fleet with asynchronous strategies such as sharding the dataflow graph, as proposed by Pathways [7]. The segmented analysis of RG shows that the particular workloads on Pathways tend to have higher RG scores over time, validating the benefits of Pathways for our particular ML Fleet. Also, techniques such as asynchronous checkpointing [39, 43] can reduce the time spent fetching previous model training checkpoints where the accelerators temporarily pause training and are completely idle.

Other strategies, such as ahead-of-time compilation, where programs are compiled on less expensive hardware such as CPUs and then executed on TPUs, can also improve RG. By offloading compilation to a less expensive chip and storing the results in a compilation cache, we can reduce the total runtime of more specialized accelerators. These techniques are often implemented in common ML frameworks, such as TensorFlow [46] and JAX [26].

To demonstrate the benefits of these framework-specific optimizations, we can examine the RG of the ML Fleet at the framework level of the system stack described in Figure 3. This can help us understand which frameworks or runtime strategies may be better suited for which workloads. Figure 13 shows RG scores from a sample of Google’s ML fleet for internal workloads, segmented based on characteristics such as model architecture, product area or workload phase (training, real-time serving, or bulk inference), and compared to a baseline of top fleet workloads. Although the segments in Figure 13 are not explicitly identified, we demonstrate how segmenting RG based on workload characteristics (Segment A, Segment B, and Segment C) can reveal trends that would otherwise be hidden by aggregate fleet metrics (represented by the "Top Fleet Workloads" segment). For example, training workloads running JAX with Pathways may tend to have a higher RG, possibly due



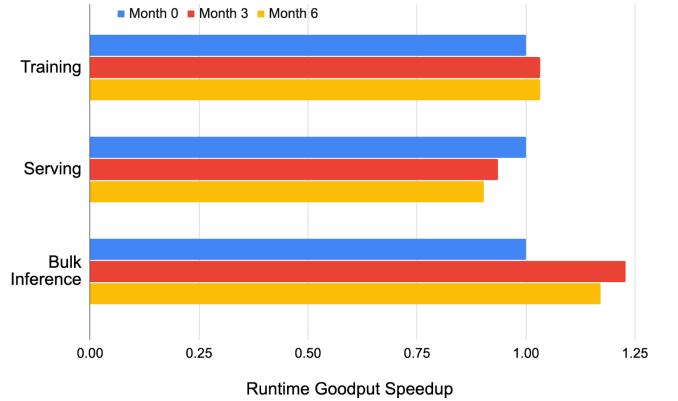
**Figure 13:** Runtime goodput speedups over the course of one quarter, segmented by fleet workload types. Speedup is normalized to the top N workloads in the fleet, measured at the beginning of the quarter.

to the fact that Pathways is single-client [7] and therefore better optimized for training than multi-client frameworks.

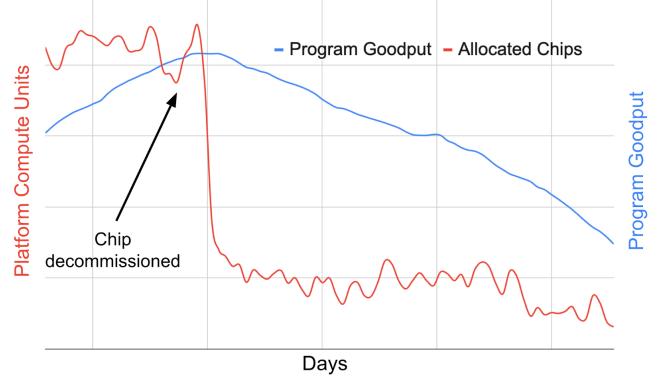
Examining the data along a different axis, training versus real-time serving versus bulk inference, can also be helpful. Using a sample from Google’s ML fleet, Figure 14 illustrates that training workloads tend to have a higher RG than serving workloads. This is most likely due to the inherently different nature of serving and training. Typically, training workloads have more constant computational demands, while real-time serving can fluctuate based on user demand. The slight decrease in serving RG can be attributed to transitory demands on the fleet, but it remains relatively stable in comparison to the bulk inference segment. The huge fluctuation in RG for bulk inference highlights the changing nature of production fleet demands. Previously, the bulk inference segment of the fleet was dominated by workloads running on a single core, where each chip contained a replica of the model, resulting in more easily accessible checkpoints/data and less accelerator wait time. However, as we move to larger models, the weights must be sharded across multiple chips, resulting in more expensive data reads. Additionally, the rise of expert-based models [54] has made bulk inference runtime much more complex to optimize, as some machines must wait for others for distillation of weight updates in a student-teacher model. This has resulted in a temporary decrease of RG for the bulk inference segment between “Month 3” and “Month 6” of Figure 14. This example illustrates how analyzing disaggregated RG can allow ML fleet architects to make informed decisions about their runtime stack by pinpointing segments that may be more susceptible to shifting fleet demands.

### 5.3 Program Goodput Optimizations

We present various techniques and strategies that have been employed at Google over recent years for improving Program Goodput in our ML fleets, ranging from parallelization methods to compiler optimizations. Recall that PG measures the effective utilization of computational resources. As ML models grow in size and complexity, optimizing PG becomes increasingly important to make efficient use of hardware and reduce compute times. With PG instrumentation, we have been able to pinpoint which segments of the fleet require further optimization at the compiler or ML model level.



**Figure 14:** Runtime Goodput trends for a notional slice of a sample ML fleet over a period of six months, segmented by workload phase.



**Figure 15:** Tracking the Program Goodput (PG) versus allocation trends for a particular domain-specific chip in an ML fleet. Looking at the disaggregated segments of MPG can help reveal fleetwide trends and interactions between different layers in the ML fleet stack, informing future design decisions.

**Overlapping communication and computation.** To identify potential system optimizations that can improve fleet efficiency, we can look at the PG of workloads segmented by performance characteristics. In other words, how many of the workloads in the fleet are compute-bound versus communication-bound? By segmenting the PG in this way, it is possible for us to identify that many high-cost workloads are communication-bound.

To address this issue at the high-level operation (HLO) level, a technique that overlaps communication with computation was developed and deployed in our production fleet (described in Wang et al. [63]). This technique decomposes communication collectives, along with the dependent computation operations, into a sequence of finer-grained operations to hide data transfer latency so that better system utilization is achieved. This approach improved the overall system throughput by up to 1.38 $\times$  and achieves 72% FLOPS utilization on 1024 TPU chips for a large language model with 500 billion parameters [63].

**Compiler autotuning.** At the fleet level, we have also developed and deployed optimizations that improve code-generation quality and can be generalized to any workload in the fleet. XTAT [48] is an autotuner for production ML compilers that tunes multiple compiler stages, including tensor layouts, operator fusion decisions, tile sizes and code generation parameters. Evaluated over 150 ML

training and inference models on TPUs, XTAT offers speedups over the heavily-optimized XLA compiler in the fleet.

**Example: Quantifying the impact of an XLA optimization on the TPU fleet.** It is rare for any single optimization to have a significant impact on overall fleet-wide PG. But we can track the impact of these optimizations by looking at the change in PG for a fixed set of benchmarked workloads or segment of the production fleet over time. For example, looking at a benchmark of the top 150 most costly workloads in the fleet, Figure 11 pinpoints the effect of a code change that was submitted to the XLA compiler - in this case, an algebraic simplification in the compiler graph. The increase in PG for the benchmark of 150 workloads suggests that the positive impact of this optimization can be generalized to the fleet.

It is also helpful to look at PG fluctuations across hardware segments of the fleet. Figure 15 shows a notional example where looking at segmented PG can uncover insights that would otherwise be hidden by looking at aggregate metrics. In this case, the segmented data suggests that when a new ML accelerator chip is introduced to the fleet, the workloads running on that chip may initially have a low PG, since the model / compiler code has not been fully tailored for that chip yet. As user adoption increases and accelerator-specific software optimizations are rolled out to the fleet, PG gets closer to theoretical peak efficiency. In other words, accelerator maturity tends to yield greater PG over time. As the chip nears the end of its lifecycle (represented by decreasing allocation in the fleet, and shown in Figure 15 by the “Chip decommissioned” label), the PG decreases due to lower chip usage and natural workload/compiler drift. This shows the importance of co-design across all layers of the ML fleet to make sure that both the software (compiler) and hardware (chips) are optimized for the latest workloads.

## 6 Discussion

To put things into perspective for MPG, we considered more traditional approaches to measuring ML fleet efficiency such as Capacity, Occupancy, and Duty Cycle. Figure 16 illustrates how we may have traditionally tended to think about performance metrics [31, 35, 37]. However, our experience reveals that these traditional metrics can sometimes fall short in providing a holistic view, given the unique challenges we have discussed in Section 2 and Section 3. While we cannot provide specific measured details for proprietary reasons, we can share the key insights and lessons learned from our evaluation of these metrics in a production environment.

**High Capacity equates to high resource availability.** While capacity can tell us how many individual accelerators may be available in the fleet at a given time, it does not take into account the topological shape of those accelerators. For example, an ML training workload requesting thousands of chips in a certain physical mesh shape may never be scheduled if the only available accelerators are fragmented across different clusters or data centers. Other factors, such as the geographical location of data storage cells and accelerators, are not included in the capacity metric, even though they significantly affect job scheduling. Therefore, high capacity by itself as a metric does not necessarily effectively translate to high availability for workloads, and we should instead opt for scheduling efficiency as a more robust metric.

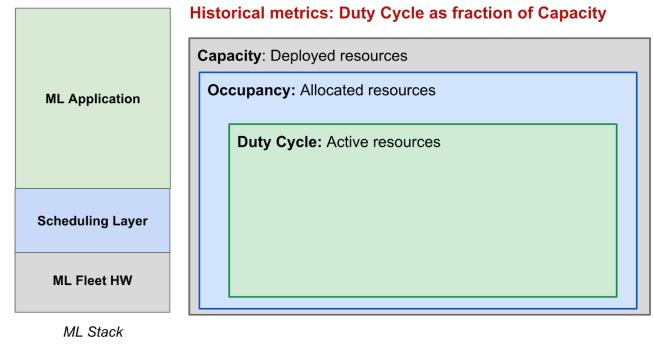


Figure 16: Traditional utilization-based metrics. We replace these using goodput as a measure of fleet efficiency rather than utilization.

**High Occupancy guarantees productivity.** Occupancy is defined as the fraction of accelerators allocated to jobs and is often measured by the scheduler (e.g. Borg [62]). Occupancy is traditionally seen as a key efficiency indicator, but it can be misleading as it masks inefficiencies in the system stack. For example, an accelerator might be successfully allocated but stuck in I/O wait or running poorly optimized code, thus resulting in a high occupancy but very little actual progress being made towards the workload task. This is important for long-running tasks such as ML model training, where frequent pre-emptions may hinder checkpoint progress but still result in a nominally high occupancy. The traditional occupancy metric therefore does not distinguish between productive and unproductive use of allocated resources.

**Duty Cycle accurately represents useful work.** Duty Cycle measures whether an accelerator is in use, not how much of its compute capacity is used. When looking at an ML workload running on a TPU, duty cycle does not provide any signal on how much the matrix-multiply units (MXUs) are utilized [28]. It is agnostic of the program-level efficiency and does not take into account the effectiveness of the operations being performed. An accelerator could have a high duty cycle while executing unnecessary or redundant computations. So, we require a more sophisticated metric.

**Overarching Misconception: Utilization == Productivity.** The thread among these metrics is the assumption that keeping accelerators busy equates to productive work. However, this overlooks critical factors. (1) *Quality of Computations:* None of these metrics assess whether the operations being performed are actually contributing to the desired output. (2) *Workload Efficiency:* They do not consider whether the workloads are optimally designed for the hardware. (3) *System-level Bottlenecks:* Focusing solely on accelerator usage ignores potential bottlenecks in data loading, memory access, or inter-accelerator communication. (4) *Forward Progress:* The traditional metrics provide no insight into how much useful work is being accomplished towards completing an actual ML task, thus motivating the need for a goodput-based metric, MPG.

## 7 Conclusion

Our study presents an analysis of ML fleet efficiency using Google’s TPU-based fleet. In Section 2 and Section 3, we provide an anatomy of the ML system stack and highlight the unique challenges of performance optimization for ML fleets. These challenges include

rapid model and hardware evolution and orchestration of hardware/software co-design. To address these issues, we introduce the ML Productivity Goodput (MPG) metric in Section 4. The MPG metric decomposes ML fleet efficiency into three key components: Scheduling, Runtime, and Program Goodputs.

The composable nature of the MPG metric allows us to dissect efficiency trends over time and precisely identify performance bottlenecks at specific layers of the stack. Our work demonstrates that this modular approach helps us develop and deploy more targeted optimization efforts and quantify their impact in an interpretable way. In Section 5, we provide examples of real efficiency improvements from leveraging MPG with Google’s own TPU fleet. The results demonstrate that the methodology presented in this paper can be generally applied to large-scale ML fleets across industry.

## 8 Acknowledgements

Our work is a product of the collaboration of many teams at Google, including the XLA team, the ML Metrics team, and the TPU Performance team. We are grateful to Daniel Herrington, Victor Cai, Kongtao Chen, Yiru Sun, Yinquan Hao, Aleksey Orehov, Peter Ma, and Jie Sun for their support with metrics instrumentation and Amit Sabne and Shibo Wang for support with XLA optimizations. We also thank Sushma Prasad, Martin Maas, Dan Zhang, David Patterson, Michael Burrows, Charles Chang, Peter Mattson, and Michael Isard for their valuable review and feedback on this publication.

## References

- [1] [n. d.] XLA. <https://openxla.org/xla>.
- [2] Martin Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mane, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viegas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. arXiv:1603.04467 [cs.DC] <https://arxiv.org/abs/1603.04467>
- [3] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A system for large-scale machine learning. arXiv:1605.08695 [cs.DC] <https://arxiv.org/abs/1605.08695>
- [4] Colin Adams, Luis Alonso, Ben Atkin, John P. Banning, Sumeer Bhola, Rick Buskens, Ming Chen, Xi Chen, Yoo Chung, Qin Jia, Nick Sakharov, George T. Talbot, Adam Jacob Tart, and Nick Taylor (Eds.). 2020. *Monarch: Google’s Planet-Scale In-Memory Time Series Database*.
- [5] Anthropic AI. 2024. The Claude 3 Model Family: Opus, Sonnet, Haiku.
- [6] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. 2009. A view of the parallel computing landscape. *Commun. ACM* 52, 10 (Oct. 2009), 56–67. <https://doi.org/10.1145/1562764.1562783>
- [7] Paul Barham, Aankanksa Chowdhery, Jeff Dean, Sanjay Ghemawat, Steven Hand, Dan Hurt, Michael Isard, Hyeontaek Lim, Ruoming Pang, Sudip Roy, Brennan Saeta, Parker Schuh, Ryan Sepassi, Laurent El Shafey, Chandramohan A. Thekkath, and Yonghui Wu. 2022. Pathways: Asynchronous Distributed Dataflow for ML. *arXiv preprint arXiv:2203.12533* (2022). <https://arxiv.org/abs/2203.12533>
- [8] Luiz André Barroso and Urs Höglund. 2009. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. <http://dx.doi.org/10.2200/S00193ED1V01Y200905CAC006>
- [9] Nathan Bell and Michael Garland. 2008. *Efficient sparse matrix-vector multiplication on CUDA*. Technical Report. Nvidia Technical Report NVR-2008-004, Nvidia Corporation.
- [10] Preston Briggs, Keith D. Cooper, and Linda Torczon. 1992. Rematerialization. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation* (San Francisco, California, USA) (PLDI ’92). Association for Computing Machinery, New York, NY, USA, 311–321. <https://doi.org/10.1145/143095.143143>
- [11] Sergey Brin and Lawrence Page. 1998. The Anatomy of a Large-Scale Hypertextual Web Search Engine. *Computer Networks* 30 (1998), 107–117. <http://www-db.stanford.edu/~backrub/google.html>
- [12] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. arXiv:2005.14165 [cs.CL] <https://arxiv.org/abs/2005.14165>
- [13] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. 2016. Borg, Omega, and Kubernetes. *Commun. ACM* 59, 5 (apr 2016), 50–57. <https://doi.org/10.1145/2890784>
- [14] Arnab Choudhury, Yang Wang, Tuomas Pelkonen, Kutta Srinivasan, Abha Jain, Shenghao Lin, Delia David, Siavash Soleimanifar, Michael Chen, Abhishek Yadav, Ritesh Tijoriwala, Denis Samoylov, and Chunqiang Tang. 2024. MAST: Global Scheduling of ML Training across Geo-Distributed Datacenters at Hyperscale. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 563–580. <https://www.usenix.org/conference/osdi24/presentation/choudhury>
- [15] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Dale Woodford, Yasushi Saito, Christopher Taylor, Michal Szymaniak, and Ruth Wang. 2012. Spanner: Google’s Globally-Distributed Database. In *OSDI*.
- [16] Lieven Eeckhout. 2010. *Computer architecture performance evaluation methods*. Morgan & Claypool Publishers.
- [17] Joel S. Emer and Douglas W. Clark. 1984. A Characterization of Processor Performance in the vax-11/780. *SIGARCH Comput. Archit. News* 12, 3 (jan 1984), 301–310. <https://doi.org/10.1145/773453.808199>
- [18] Andre Esteve, Katherine Chou, Serena Yeung, Nikhil Naik, Ali Madani, Ali Mottaghi, Yun Liu, Eric Topol, Jeff Dean, and Richard Socher. 2021. Deep learning-enabled medical computer vision. *NPJ digital medicine* 4, 1 (2021), 5.
- [19] Kayvon Fatahalian, Jeremy Sugerman, and Pat Hanrahan. 2004. Understanding the efficiency of GPU algorithms for matrix-matrix multiplication. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*. 133–137.
- [20] Roy Frostig, Matthew Johnson, and Chris Leary. 2018. Compiling machine learning programs via high-level tracing. <https://mlsys.org/Conferences/doc/2018/146.pdf>
- [21] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google File System. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*. Bolton Landing, NY, 20–43.
- [22] James Hamilton. 2007. On Designing and Deploying Internet-Scale Services. In *21st Large Installation System Administration Conference (LISA 07)*. USENIX Association, Dallas, TX. <https://www.usenix.org/conference/lisa-07/designing-and-deploying-internet-scale-services>
- [23] John L. Hennessy and David A. Patterson. 2019. A new golden age for computer architecture. *Commun. ACM* 62, 2 (jan 2019), 48–60. <https://doi.org/10.1145/3282307>
- [24] Dean Hildebrand and Denis Serenyi. 2021. Colossus under the hood: a peek into Google’s scalable storage system . <https://cloud.google.com/blog/products/storage-data-transfer/a-peek-behind-colossus-googles-file-system>
- [25] Joel Janai, Fatma Güney, Aseem Behl, Andreas Geiger, et al. 2020. Computer vision for autonomous vehicles: Problems, datasets and state of the art. *Foundations and Trends® in Computer Graphics and Vision* 12, 1–3 (2020), 1–308.
- [26] JAX. [n. d.] Ahead-of-time lowering and compilation. <https://jax.readthedocs.io/en/latest/aot.html>
- [27] Norman P. Jouppi, Doe Hyun Yoon, Matthew Ashcraft, Mark Gotscho, Thomas B. Jablin, George Kurian, James Laudon, Sheng Li, Peter Ma, Xiaoyu Ma, Thomas Norrie, Nishant Patil, Sushma Prasad, Cliff Young, Zongwei Zhou, and David Patterson. 2021. Ten Lessons From Three Generations Shaped Google’s TPUs: Industrial Product. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 1–14. <https://doi.org/10.1109/ISCA52012.2021.00010>
- [28] Norman P. Jouppi, George Kurian, Sheng Li, Peter Ma, Rahul Nagarajan, Lifeng Nai, Nishant Patil, Suvinay Subramanian, Andy Swing, Brian Towles, Cliff Young, Xiang Zhou, Zongwei Zhou, and David Patterson. 2023. TPUs v4: An Optically Reconfigurable Supercomputer for Machine Learning with Hardware Support for Embeddings. arXiv:2304.01433 [cs.AR] <https://arxiv.org/abs/2304.01433>
- [29] Norman P. Jouppi, Cliff Young, Nishant Patil, David A. Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle,

- Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagnann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Amir Salek, Enad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datcenter Performance Analysis of a Tensor Processing Unit. *CoRR* abs/1704.04760 (2017). arXiv:1704.04760 <http://arxiv.org/abs/1704.04760>
- [30] Shoaib Kamil, John Shalf, and Erich Strohmaier. 2008. Power efficiency in high performance computing. In *2008 IEEE International Symposium on Parallel and Distributed Processing*, 1–8. <https://doi.org/10.1109/IPDPS.2008.4536223>
- [31] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. 2015. Profiling a warehouse-scale computer. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, 158–169.
- [32] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems*, F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger (Eds.), Vol. 25. Curran Associates, Inc. [https://proceedings.neurips.cc/paper\\_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf](https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf)
- [33] Michael Kuchnik, Ana Klimovic, Jiri Simsa, Virginia Smith, and George Amvrosiadis. 2022. Plumber: Diagnosing and Removing Performance Bottlenecks in Machine Learning Data Pipelines. *Proceedings of Machine Learning and Systems* 4 (2022), 33–51.
- [34] Sameer Kumar, Yu Wang, Cliff Young, James Bradbury, Naveen Kumar, Dehao Chen, and Andy Swing. 2021. Exploring the limits of Concurrency in ML Training on Google TPUs. *Proceedings of Machine Learning and Systems* 3 (2021), 81–92.
- [35] Jie Li, George Michelogiannakis, Brandon Cook, Dulanya Cooray, and Yong Chen. 2023. Analyzing resource utilization in an HPC system: A case study of NERSC’s Perlmutter. In *International Conference on High Performance Computing*. Springer, 297–316.
- [36] Mu Li, David G Andersen, Alexander J Smola, and Kai Yu. 2014. Communication efficient distributed machine learning with the parameter server. *Advances in Neural Information Processing Systems* 27 (2014).
- [37] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. 2011. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proceedings of the 44th annual IEEE/ACM International Symposium on Microarchitecture*, 248–259.
- [38] Peter Mattson, Christine Cheng, Gregory Diamos, Cody Coleman, Paulius Micikevicius, David Patterson, Hanlin Tang, Gu-Yeon Wei, Peter Bailis, Victor Bittorf, et al. 2020. Mlperf training benchmark. *Proceedings of Machine Learning and Systems* 2 (2020), 336–349.
- [39] Avinash Maurya, Robert Underwood, M. Mustafa Rafique, Franck Cappello, and Bogdan Niclae. 2024. DataStates-LLM: Lazy Asynchronous Checkpointing for Large Language Models. In *Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing* (Pisa, Italy) (HPDC ’24). Association for Computing Machinery, New York, NY, USA, 227–239. <https://doi.org/10.1145/36225549.3658685>
- [40] D Menemenlis, C Hill, A Adcroft, J-M Campin, B Cheng, B Ciotti, I Fukumori, P Heimbach, C Henze, A Kohl, et al. 2005. NASA supercomputer improves prospects for ocean climate research. *Eos, Transactions American Geophysical Union* 86, 9 (2005), 89–96.
- [41] Derek G. Murray, Jiri Simsa, Ana Klimovic, and Ihor Indyk. 2021. tf.data: A Machine Learning Data Processing Framework. arXiv:2101.12127 [cs.LG] <https://arxiv.org/abs/2101.12127>
- [42] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Ilia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. 2019. Deep Learning Recommendation Model for Personalization and Recommendation Systems. arXiv:1906.00091 [cs.IR] <https://arxiv.org/abs/1906.00091>
- [43] Bogdan Niclae, Jiali Li, Justin M. Wozniak, George Bosilca, Matthieu Dorier, and Franck Cappello. 2020. DeepFreeze: Towards Scalable Asynchronous Checkpointing of Deep Learning Models. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, 172–181. <https://doi.org/10.1109/CCGrid49817.2020.00076>
- [44] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. 2013. Scaling Memcache at Facebook. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. USENIX Association, Lombard, IL, 385–398. <https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/nishtala>
- [45] Tetsuya Odajima, Yuetsu Kodama, Miwako Tsuji, Motohiko Matsuda, Yutaka Maruyama, and Mitsuhsisa Sato. 2020. Preliminary Performance Evaluation of the Fujitsu A64FX Using HPC Applications. In *2020 IEEE International Conference on Cluster Computing (CLUSTER)*, 523–530. <https://doi.org/10.1109/CLUSTER49012.2020.00075>
- [46] OpenXLA. [n. d.]. Using AOT compilation. <https://openxla.org/xla/tf2xla/tfcompile>.
- [47] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. arXiv:1912.01703 [cs.LG] <https://arxiv.org/abs/1912.01703>
- [48] Phitchaya Mangpob Phothilimthana, Amit Sabne, Nikhil Sarda, Karthik Srinivas Murthy, Yanqi Zhou, Christof Angermueller, Mike Burrows, Sudip Roy, Ketan Mandke, Reza Farahani, et al. 2021. A Flexible Approach to Autotuning Multi-Pass Machine Learning Compilers. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 1–16.
- [49] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. 2023. Efficiently Scaling Transformer Inference. In *Proceedings of Machine Learning and Systems*, D. Song, M. Carbin, and T. Chen (Eds.), Vol. 5. Curran, 606–624. [https://proceedings.mlsys.org/paper\\_files/paper/2023/file/c4be71ab8d24cd8fb45e3d06dbfc2a780-Paper-mlsys2023.pdf](https://proceedings.mlsys.org/paper_files/paper/2023/file/c4be71ab8d24cd8fb45e3d06dbfc2a780-Paper-mlsys2023.pdf)
- [50] Parthasarathy Ranganathan and Urs Holzle. 2024. Twenty Five Years of Warehouse-Scale Computing . *IEEE Micro* 44, 05 (Sept. 2024), 11–22. <https://doi.org/10.1109/MM.2024.3409469>
- [51] David K. Rensin. 2015. *Kubernetes - Scheduling the Future at Cloud Scale*. 1005 Gravenstein Highway North Sebastopol, CA 95472. All pages. <http://www.oreilly.com/webops-perf/free/kubernetes.csp>
- [52] Paul Menage Sanjay Ghemawat. [n. d.]. TCMalloc : Thread-Caching Malloc . <https://goog-perftools.sourceforge.net/doc/tcmalloc.html>.
- [53] Roland Schulz, Benjamin Lindner, Loukas Petridis, and Jeremy C Smith. 2009. Scaling of multimillion-atom biological molecular dynamics simulation on a petascale supercomputer. *Journal of Chemical Theory and Computation* 5, 10 (2009), 2798–2808.
- [54] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. 2017. Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer. arXiv:1701.06538 [cs.LG] <https://arxiv.org/abs/1701.06538>
- [55] Zhan Shi, Chirag Sakhuja, Milad Hashemi, Kevin Swersky, and Calvin Lin. 2020. Learned Hardware/Software Co-Design of Neural Accelerators. *CoRR* abs/2010.02075 (2020). arXiv:2010.02075 <https://arxiv.org/abs/2010.02075>
- [56] Vladimir Stegailov, Ekaterina Dlinnova, Timur Ismagilov, Mikhail Khalilov, Nikolay Kondratyuk, Dmitry Makagon, Alexander Semenov, Alexei Simonov, Grigory Smirnov, and Alexey Timofeev. 2019. Angara interconnect makes GPU-based Desmos supercomputer an efficient tool for molecular dynamics calculations. *The International Journal of High Performance Computing Applications* 33, 3 (2019), 507–521.
- [57] Varun Talwar. 2016. gRPC: a true internet-scale RPC framework is now 1.0 and ready for production deployments . <https://cloud.google.com/blog/products/gcp/grpc-a-true-internet-scale-rpc-framework-is-now-1-and-ready-for-production-deployments>.
- [58] Gemini Team. 2024. Gemini: A Family of Highly Capable Multimodal Models. arXiv:2312.11805 [cs.CL] <https://arxiv.org/abs/2312.11805>
- [59] Amin Vahdat and Mark Lohmeyer. 2023. Enabling next-generation AI workloads: Announcing TPU v5p and AI Hypercomputer. <https://cloud.google.com/blog/products/ai-machine-learning/introducing-cloud-tpu-v5p-and-ai-hypercomputer>.
- [60] Kenton Varda. 2008. Protocol Buffers: Google’s Data Interchange Format . <https://opensource.googleblog.com/2008/07/protocol-buffers-googles-data.html>.
- [61] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems*, I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc. [https://proceedings.neurips.cc/paper\\_files/paper/2017/file/3f5ee243547dee91fb0d053c1c4a845aa-Paper.pdf](https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fb0d053c1c4a845aa-Paper.pdf)
- [62] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)*. Bordeaux, France.
- [63] Shibo Wang, Jinliang Wei, Amit Sabne, Andy Davis, Berkin Ilbeyi, Blake Hechman, Dehao Chen, Karthik Srinivas Murthy, Marcello Maggioni, Qiao Zhang, et al. 2022. Overlap Communication with Dependent Computation via Decomposition in Large Deep Learning Models. In *Proceedings of the 28th ACM International*

- Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1.* 93–106.
- [64] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM* 52, 4 (apr 2009), 65–76. <https://doi.org/10.1145/1498765.1498785>
- [65] Amir Yazdanbakhsh, Kiran Seshadri, Berkin Akin, James Laudon, and Ravi Narayanaswami. 2021. An Evaluation of Edge TPU Accelerators for Convolutional Neural Networks. *CoRR* abs/2102.10423 (2021). arXiv:2102.10423 <https://arxiv.org/abs/2102.10423>
- [66] Andy B. Yoo, Morris A. Jette, and Mark Grondona. 2003. SLURM: Simple Linux Utility for Resource Management. In *Job Scheduling Strategies for Parallel Processing*, Dror Feitelson, Larry Rudolph, and Uwe Schliegelshohn (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 44–60.
- [67] Mark Zhao, Niket Agarwal, Aarti Basant, Bugra Gedik, Satadru Pan, Mustafa Ozdal, Rakesh Komuravelli, Jerry Pan, Tianshu Bao, Haowei Lu, Sundaram Narayanan, Jack Langman, Kevin Wilfong, Harsha Rastogi, Carole-Jean Wu, Christos Kozyrakis, and Parik Pol. 2021. Understanding and Co-designing the Data Ingestion Pipeline for Industry-Scale RecSys Training. *CoRR* abs/2108.09373 (2021). arXiv:2108.09373 <https://arxiv.org/abs/2108.09373>
- [68] Mark Zhao, Niket Agarwal, Aarti Basant, Bugra Gedik, Satadru Pan, Mustafa Ozdal, Rakesh Komuravelli, Jerry Pan, Tianshu Bao, Haowei Lu, Sundaram Narayanan, Jack Langman, Kevin Wilfong, Harsha Rastogi, Carole-Jean Wu, Christos Kozyrakis, and Parik Pol. 2022. Understanding data storage and ingestion for large-scale deep recommendation model training: industrial product. In *Proceedings of the 49th Annual International Symposium on Computer Architecture (New York, New York) (ISCA ’22)*. Association for Computing Machinery, New York, NY, USA, 1042–1057. <https://doi.org/10.1145/3470496.3533044>
- [69] Yazhou Zu, Alireza Ghaffarkhah, Hoang-Vu Dang, Brian Towles, Steven Hand, Safeen Huda, Adekunle Bello, Alexander Kolbasov, Arash Rezaei, Dayou Du, Steve Lacy, Hang Wang, Aaron Wisner, Chris Lewis, and Henri Bahini. 2024. Resiliency at Scale: Managing Google’s TPUs Machine Learning Supercomputer. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. USENIX Association, Santa Clara, CA, 761–774. <https://www.usenix.org/conference/nsdi24/presentation/zu>