MLSys · im

First-Principles Infrastructure Modeling for Machine Learning Systems

Vijay Janapa Reddi
Harvard University

mlsysbook.ai/mlsysim

Abstract

As machine learning models transition from laboratory curiosities to critical infrastructure, the systems required to sustain them have reached a point of extreme bimodal complexity. Developers must reason about constraints spanning from sub-milliwatt microcontrollers to multi-gigawatt datacenter fleets. Existing evaluation methodologies are polarized between hardware-dependent empirical profiling and cycle-accurate simulation, leaving a void for rapid, full-stack architectural reasoning. We present **MLSys · im** (**M**achine **L**earning **Sys**tems **I**nfrastructure **M**odeling), a first-principles analytical modeling framework that formalizes the "physics of systems" into a dimensionally-strict Python engine. MLSys · im introduces a 5-layer *Progressive Lowering* abstraction that decouples computational demand from silicon supply and environmental context. Strict enforcement of unit-level integrity at runtime eliminates the silent conversion errors that plague ad-hoc systems modeling. We codify a complete taxonomy of 22 "Systems Walls" (hard physical or logical constraints spanning compute ceilings, memory bandwidth, network topology, data pipelines, scaling laws, fleet reliability, economics, and sustainability) organized into six domains and resolved by a suite of 25 resolvers (20 analytical models, 2 analysis solvers, and 3 optimizers). Our evaluation demonstrates that MLSys · im enables sub-second design-space exploration, identifying binding constraints and synthesizing ideal hardware specifications across the entire ML systems lifecycle.

1 Introduction

Machine learning has become infrastructure (Sutton, 2019). Training a frontier model now requires orchestrating tens of thousands of accelerators across datacenter fabrics where memory ceilings, network bandwidth, power delivery, and regional carbon intensities interact in non-obvious ways (Dean et al., 2012; Shoeybi et al., 2019). The pace of this scaling is accelerating: frontier models have grown from billions to trillions of parameters in under five years, and the infrastructure cost of a single training run now rivals that of a small datacenter (DeepSeek-AI, 2025).

Yet the hardware required to develop intuition for these systems is prohibitively scarce: a student cannot requisition a 100,000-GPU cluster to explore how topology affects AllReduce latency, and a researcher cannot easily sweep parallelism strategies across hardware generations. This creates a growing *reasoning gap*. The systems are getting more complex, but the tools to think about them have not kept pace.

Consider a concrete example. A team deploying LLaMA-3 70B for interactive serving must answer: *How many H100 GPUs are needed to meet a 50 ms time-to-first-token SLA at 95th-percentile latency?* The answer depends on at least seven interacting constraints, each a "wall" (a hard bound imposed by physics, economics, or algorithmic scaling; see Table 1): the model's 70 billion parameters require ∼140 GB in FP16, exceeding a single GPU's 80 GB HBM capacity (Wall 2: Memory). Tensor-parallel sharding across two GPUs introduces NVLink synchronization overhead (Wall 14: Communication). Continuous batching with PagedAttention determines KV-cache memory utilization (Wall 5: Batching). The decode phase is memory-bandwidth-bound at 3.35 TB/s per device (Wall 4: Serving). Tail latency under load follows Erlang-C queueing dynamics (Wall 7: Tail Latency). And the fleet's total cost of ownership constrains what is economically viable (Wall 17: Capital). A similar multi-wall analysis applies to training: determining the optimal parallelism strategy for a 512-GPU run involves compute throughput, memory capacity, communication overhead, scaling laws, reliability, and economics simultaneously. No single equation suffices.

Existing tools are constrained along three axes (fidelity, speed, and scope), and no tool occupies the region where all three are adequate (Table 2). Profilers require physical silicon; cycle-accurate simulators like ASTRA-

sim 2.0 (Won et al., 2023) require hours per configuration; analytical tools like Calculon (Isaev et al., 2023) achieve speed but focus narrowly on LLM training, ignoring data pipelines, reliability, sustainability, and inference. Patterson and Hennessy faced an analogous tradeoff in computer architecture education and chose taxonomic completeness over cycle accuracy: MIPS exposed every architectural concept through a model simple enough to reason about yet faithful enough to develop correct intuition (Hennessy et al., 2024). MLSYS·IM makes the same choice for ML systems. Beyond coverage gaps, none of these tools enforce dimensional correctness. In ML systems, confusing GB with GiB, FLOP/s with FLOPs, or bandwidth with throughput can silently invalidate capacity-planning spreadsheets, yet no existing tool catches these errors at runtime.

MLSYS·IM (Machine Learning Systems Infrastructure Modeling) is an open-source, pure-Python analytical framework that formalizes back-of-the-envelope ML systems reasoning into a dimensionally strict, composable engine. The framework codifies 22 "Systems Walls" into 25 composable resolvers organized across six domains: Node, Data, Algorithm, Fleet, Operations, and Analysis. It separates computational *demand* from silicon *supply* and environmental *context* through a 5-layer progressive lowering architecture, enforces SI unit correctness at runtime via the `pint` library, and produces full-stack analysis in milliseconds on any laptop.

Each of the resolvers computes structural physics from first-principles equations and hardware datasheet constants, then bridges the gap to measured performance through a single, explicit efficiency coefficient η that absorbs second-order effects (kernel launch overhead, straggler variance, framework dispatch costs). This is the same abstraction that makes the Roofline model useful: replacing peak bandwidth with *achievable* bandwidth sacrifices cycle-level precision for the ability to reason correctly about which constraint binds and why (Williams et al., 2009).

MLSYS·IM serves three communities. *Students* build quantitative intuition through interactive labs where changing a single parameter reveals which wall binds and why. *Instructors* run live classroom demonstrations that produce concrete numbers in real time, replacing hand-waving with Roofline diagrams. *Researchers* perform rapid what-if analysis: comparing parallelism strategies, evaluating procurement decisions, or projecting carbon footprints before committing resources. Designed as the analytical companion to the *Machine Learning Systems* textbook (Reddi et al., 2025a), MLSYS·IM is fully open-source (MIT license) with deterministic, hardware-free execution, ensuring that every result in this paper is independently reproducible. This matters for educational equity: a student at a community college with a Chromebook can run the same quantitative exercises and verify the same binding constraints as one with access to a university GPU cluster. We validate against published empirical anchors spanning five taxonomy domains, matching values within 7.1% error while sweeping over 1,000 configurations in under one second.

This paper makes the following contributions:

C1. A Taxonomy of 22 Systems Walls (Table 1), each grounded in a published equation and resolved by a dedicated resolver (model or solver). The taxonomy provides a complete, structured vocabulary for reasoning about the constraints that bind ML system performance (Section 4).

C2. Demand–Supply Separation with Dimensional Strictness. A 5-layer progressive lowering abstraction formally decouples computational demand from silicon supply. Every physical quantity carries SI units at runtime, transforming dimensional analysis from a manual discipline into a machine-checked invariant (Section 3).

C3. Composable Resolver Algebra. 25 resolvers (20 models, 2 solvers, and 3 optimizers) compose through chaining: each is a pure function $f(\text{config}) \rightarrow \text{metrics}$, producing a three-level evaluation (Feasibility, Performance, Macro) that identifies binding constraints. The algebra includes an inverse-Roofline *synthesis* solver that derives minimum hardware specifications from SLA requirements (Sections 5 and 7).

C4. Accessible Full-Stack Reasoning without Hardware. MLSYS·IM runs on any laptop without GPUs, clusters, or cloud credits. It powers fully autogradable, deterministic labs and compiles directly to WebAssembly via Marimo notebooks, providing an interactive, browser-based systems engineering environment (Section 7).

The paper builds the framework in layers. We first survey the modeling landscape and identify the void (Section 2), then present the architecture: three design principles (Section 3.1) that materialize as a 5-layer progressive lowering stack (Section 3.2) whose type system enables the 22-wall taxonomy (Section 4), which in turn defines the solver algebra (Section 5). We validate against published benchmarks spanning five domains (Section 6), demonstrate the framework through student, instructor, and researcher use cases (Section 7), surface common misconceptions the framework is designed to expose (Section 8), and discuss limitations and future work (Section 9).

Table 1: **The 22 ML Systems Walls.** Each wall represents a physical or logical constraint resolved by a dedicated resolver (model or solver). Walls 1–2 (Compute and Memory) share the `SingleNodeModel`. Together with 3 optimizers (parallelism, batching, placement), the framework provides 25 resolvers across 22 walls. Domains progress from local node resources through data movement and algorithmic scaling to fleet coordination, operations, and cross-cutting analysis. Each wall is formalized in Section 4.

#	Wall	Resolver	Bounded	Core Equation	Ref.
Node (Single-Accelerator Resources)					
1	Compute	SingleNode	Peak FLOP/s	$T = \text{OPs}/(\text{Peak} \times \eta)$	Williams et al. 2009
2	Memory	SingleNode	HBM BW + cap.	$T = \lvert W\rvert / BW_{\text{HBM}}$	Williams et al. 2009
3	Software	Efficiency	Achieved MFU	$\eta = \text{FLOPS}_{\text{ach}}/\text{Peak}$	Chowdhery et al. 2023
4	Serving	Serving	Prefill vs. dec.	$T_{\text{pf}} = 2PS/(F\eta)$; $T_{\text{dec}} = \lvert W\rvert/BW$	Pope et al. 2023
5	Batching	Cont. Batch	KV-cache frag.	$\text{KV} = 2LHD\lceil S/p\rceil pBb$	Kwon et al. 2023
6	Streaming	WeightStream	Injection BW	$T = \max(\lvert W_\ell\rvert/BW, 2P_\ell B/F\eta)$	Lie 2022
7	Tail Latency	TailLatency	P99 queueing	Erlang-C M/M/c	Dean and Barroso 2013
Data (Movement & Pipelines)					
8	Ingestion	Data	Storage I/O	$\rho = BW_{\text{demand}}/BW_{\text{supply}}$	Mohan et al. 2021
9	Transform.	Transform.	CPU preproc.	$T = B/R_{\text{cpu}}$	Murray et al. 2021
10	Locality	Topology	Bisection BW	$BW_{\text{eff}} = BW_{\text{link}} \cdot \beta/\text{osub}$	Leiserson 1985
Algorithm (Scaling & Compression)					
11	Complexity	Scaling	Scaling laws	$C = 6PD$; $P^* = \sqrt{C/120}$	Hoffmann et al. 2022
12	Reasoning	Inf. Scaling	Inf.-time comp.	$T = K \times T_{\text{step}}$	Snell et al. 2025
13	Fidelity	Compression	Acc.–efficiency	$r = b_{\text{base}}/b$; $r = 1/(1-s)$	Han et al. 2016
Fleet (Multi-Node Coordination)					
14	Communic.	Distributed	AllReduce	$T = 2\frac{N-1}{N}\frac{M}{B_{\text{link}}} + 2(N-1)\alpha$	Shoeybi et al. 2019
15	Fragility	Reliability	Cluster MTBF	$\text{MTBF}_{\text{cl}} = \text{MTBF}_{\text{node}}/N$	Daly 2006
16	Multi-tenant	Orchestration	Queue wait	$T_{\text{wait}} = \rho/[2\mu(1-\rho)]$	Little 1961
Operations (Economics, Sustainability & Safety)					
17	Capital	Economics	TCO	$\text{TCO} = \text{CapEx} + \text{OpEx}$	Barroso et al. 2018
18	Sustain.	Sustainability	Carbon + water	$\text{CO}_2 = E \times \text{PUE} \times \text{CI}$	Patterson et al. 2021
19	Checkpoint	Checkpoint	I/O burst penalty	$\text{penalty} = T_{\text{write}}/T_{\text{interval}}$	Eisenman et al. 2022
20	Safety	Resp. Eng.	DP-SGD overhead	$\sigma \propto 1/\varepsilon$	Abadi et al. 2016
Analysis (Cross-Cutting Diagnostics)					
21	Sensitivity	Sensitivity	Binding constr.	$\partial T/\partial x_i$	Williams et al. 2009
22	Synthesis	Synthesis	Inverse spec	$BW_{\text{req}} = \lvert W\rvert/T_{\text{target}}$	Kwon et al. 2023
Optimizers (Design-Space Search — no dedicated wall)					
	Parallelism	Parallelism	Max MFU	$\max_c \eta(c)$ over TP×PP×DP	Shoeybi et al. 2019
	Batching	Batching	SLA-aware B^*	max B s.t. $P_{99} \le T_{\text{SLA}}$	Kwon et al. 2023
	Placement	Placement	Cost–perf.	min cost s.t. $T \le T_{\text{SLA}}$	—

2 Related Work

Tools for modeling and evaluating ML systems span a wide spectrum of fidelity, scope, and intended audience. We organize prior work into four categories and position MLSys·im relative to each: (1) cycle-level simulators that model hardware at the microarchitectural level, (2) accelerator design tools that evaluate individual chip architectures, (3) analytical and co-design tools that trade fidelity for speed, and (4) pedagogical simulators that prioritize conceptual clarity. Table 2 provides a quantitative summary across these categories.

2.1 Cycle-Level Simulators

Cycle-accurate simulators provide the highest fidelity by modeling hardware behavior at the microarchitectural level. gem5 (Binkert et al., 2011) is the canonical general-purpose architecture simulator, capable of modeling CPUs and GPUs down to individual pipeline stages. While invaluable for processor design, gem5 lacks ML-

specific abstractions (it has no notion of a transformer layer, a training step, or a parallelism strategy), and simulating even a single forward pass of a modern model can require hours of wall-clock time.

ASTRA-sim 2.0 (Won et al., 2023) addresses the ML gap by providing a hierarchical network simulator purpose-built for distributed training. It models collective communication patterns such as AllReduce across realistic network topologies, producing high-fidelity estimates of communication overhead. SimAI (Wang et al., 2025) extends this approach with a full-stack training simulator that integrates NS3-based network modeling with kernel computation traces, achieving 98% alignment with real-world results on 1024-node A100 clusters. Both inherit the fundamental cost of high fidelity: simulating one training step of a large model at cluster scale can take minutes to hours, making iterative design-space exploration impractical. Their scope is also limited to communication and compute; neither models economics, sustainability, data pipelines, or reliability.

MLSYS · IM occupies a different point on the fidelity–speed spectrum. Where ASTRA-sim answers "how many microseconds does this AllReduce take on this exact topology," MLSYS · IM answers "which of 22 possible bottlenecks binds this system, and how does changing hardware shift the binding constraint?" The two classes are complementary: MLSYS · IM narrows the design space, and high-fidelity simulators validate specific points within it.

2.2 Accelerator Design Tools

A second class of tools targets the design and evaluation of individual accelerator architectures. Timeloop (Parashar et al., 2019) provides a systematic methodology for evaluating DNN accelerator dataflows, modeling how data tiles map onto spatial architectures and estimating latency and energy for each mapping. Accelergy (Wu et al., 2019), its companion framework, supplies the energy estimation primitives that Timeloop consumes. Together, they form a powerful toolkit for accelerator architects exploring the design space of novel silicon. LLMCompass (Zhang et al., 2024) brings this approach to LLM inference, combining an automated mapper with an area-based cost model to explore compute, memory bandwidth, and buffer configurations, achieving 4% error for end-to-end LLM inference on A100 nodes within minutes.

These tools operate at the operator and tile level, modeling how a single convolution or matrix multiplication executes on a specific microarchitecture. They do not reason about system-level concerns: how multiple accelerators communicate across a network fabric, how the data pipeline feeds those accelerators, or what the total cost of ownership looks like at fleet scale. MLSYS · IM operates one abstraction level higher. It consumes the *outputs* of accelerator-level analysis (peak FLOP/s, memory bandwidth, TDP) as inputs to its hardware registry and reasons about how those specifications interact with workload demands, network topologies, and infrastructure constraints.

2.3 Analytical and Co-Design Tools

Closest in spirit to MLSYS · IM are analytical tools that sacrifice microarchitectural detail for speed. Calculon (Isaev et al., 2023) is an analytical co-design tool for large language model training. It models training time as a function of hardware specifications, parallelism strategies, and model architecture, achieving execution speeds comparable to MLSYS · IM. However, Calculon's scope is narrow by design, targeting transformer-based LLM training exclusively, with no support for CNNs, mixture-of-experts architectures, or inference workloads. It does not model data pipelines, reliability, sustainability, economics, or safety considerations, and it lacks dimensional enforcement. Lumos (Liang et al., 2025) takes a trace-driven approach, using profiled kernel traces to predict LLM training performance at scale with 3.3% average error on up to 512 H100 GPUs. While Lumos achieves higher single-point accuracy than MLSYS · IM, it requires empirical traces from the target hardware, limiting its use in what-if exploration across hypothetical configurations. The DeepSeek-V3 systems paper (DeepSeek-AI, 2025) exemplifies the kind of hardware-aware co-design analysis that MLSYS · IM targets: it demonstrates how FP8 mixed-precision training, MoE sparsity, and multi-plane network topology interact to achieve frontier-model training at a fraction of conventional cost, a multi-wall optimization that spans Walls 1, 3, 13, 14, and 17 in our taxonomy.

Paleo (Qi et al., 2017) pioneered the analytical approach, decomposing DNN training time into computation and communication components across data- and model-parallel configurations. FlexFlow (Jia et al., 2019) optimizes parallelism strategies through simulation-guided search, and Habitat (Yu et al., 2021) provides cross-hardware extrapolation of training performance using execution-time scaling curves. While these tools advance specific aspects of analytical modeling, they predate or do not address the full scope of modern concerns (inference serving, fleet economics, sustainability). Vidur (Agrawal et al., 2024b) extends analytical modeling to LLM *inference*, using operator-level profiling to build a fine-grained runtime estimator validated at less than 5% error across multiple LLMs and scheduling policies. GenZ (Bambhaniya et al., 2024) provides an analytical framework for LLM inference platform de-

Table 2: **Comparison of ML systems modeling tools.** Scope indicates the range of system aspects modeled. Speed reflects wall-clock time for a single evaluation. Walls indicates the number of the 22 systems walls (Table 1) each tool addresses. Phase indicates whether training, inference, or both are modeled. Dist. indicates support for multi-node distributed analysis.

Tool	Approach	Scope	Speed	Walls	Phase	Dist.
Cycle-level						
gem5	Cycle-accurate	CPU/GPU microarchitecture	Hours	1–2	Both	–
ASTRA-sim 2.0	Cycle-accurate	Network collectives, topology	Hours	1–2	Train	✓
SimAI	Trace-driven	Full-stack distributed training	Minutes	2–3	Train	✓
Accelerator design						
Timeloop + Accelergy	Analytical	Accelerator dataflow & energy	Minutes	1–2	Infer	–
LLMCompass	Analytical	LLM inference HW design space	Minutes	2–3	Infer	–
Analytical & co-design						
Paleo	Analytical	DNN training compute & comm.	Seconds	2	Train	✓
Calculon	Analytical	LLM training performance	Seconds	2–3	Train	✓
Lumos	Trace-driven	LLM training perf. modeling	Seconds	2–3	Train	✓
Vidur	Empirical	LLM inference scheduling	Seconds	3–4	Infer	–
GenZ	Analytical	LLM inference platform design	Seconds	2–3	Infer	–
LLM-Viewer	Analytical	LLM inference memory/latency	Seconds	1–2	Infer	–
Sustainability						
LLMCarbon	Analytical	LLM carbon footprint (op. + embodied)	Seconds	1	Both	–
CodeCarbon	Empirical	Runtime energy & carbon tracking	Seconds	1	Both	–
MLSys · im	**Analytical**	**Full-stack: compute, memory, network, data, scaling, reliability, econ., sustainability, safety**	**Sub-sec.**	**22**	**Both**	✓

sign that models multi-dimensional network topologies and serving optimizations. DistServe (Zhong et al., 2024) and Sarathi-Serve (Agrawal et al., 2024a) advance LLM serving through prefill-decode disaggregation and chunked-prefill scheduling respectively, demonstrating that the two-phase inference model (Equation (6)) requires increasingly sophisticated scheduling to achieve high goodput. All of these tools focus on inference performance in isolation; they do not model training, data pipelines, economics, or sustainability.

LLM-Viewer (Yuan et al., 2024) and llm-analysis (Li, 2023) provide lightweight memory and latency estimation for transformer inference. These tools are useful for single-model profiling but do not extend to fleet-level reasoning, multi-tenant scheduling, or cross-domain constraint analysis.

A parallel line of work targets sustainability and fleet efficiency. LLMCarbon (Faiz et al., 2024) projects end-to-end carbon footprints (operational and embodied) for dense and MoE LLMs, validated within 8% of Google's published figures. CodeCarbon (Lottick et al., 2019) provides empirical energy tracking at runtime via hardware power monitors. Wongpanich et al. (2025) introduce *ML Productivity Goodput* (MPG) as a fleet-level efficiency metric for warehouse-scale TPU clusters, demonstrating that traditional utilization metrics are insufficient for characterizing ML fleet performance across model, data, framework, compiler, and scheduling layers. These tools each address one or two domains and do not model the full cross-stack interactions that determine *why* a workload consumes so much energy.

MLSys · im generalizes the analytical approach to the full ML systems stack. Where each tool above models one or two domains, MLSys · im composes resolvers spanning all six (Table 1): compute, memory, serving, data pipelines, scaling laws, fleet coordination, economics, sustainability, and responsible engineering. It also enforces dimensional strictness at runtime (Section 3.1.2), transforming unit consistency from a manual discipline into a machine-checked invariant.

2.4 Pedagogical Precedents

MLSys · im draws direct inspiration from the tradition of pedagogical simulators in systems education. Patterson and Hennessy's MIPS/SPIM simulator (Patterson and Hennessy, 2014) taught generations of students computer architecture not by replicating a production processor, but by providing a simplified model that made architectural concepts tangible through rapid experimentation. Similarly, xv6 (Cox et al., 2011) and

MINIX (Tanenbaum and Woodhull, 2006) teach operating systems by stripping away production complexity to reveal core abstractions.

MLSYS·IM follows this pedagogical philosophy for the ML systems domain. Production ML infrastructure (spanning millions of lines of code across frameworks, compilers, schedulers, and orchestrators) is too complex for students to reason about directly. MLSYS·IM provides a controlled environment in which students can sweep hardware configurations, vary parallelism strategies, and observe how binding constraints shift, all in rapid iteration cycles. Its integration with the companion textbook (Reddi et al., 2025a) provides structured laboratory exercises with autogradable assessments, a capability absent from every research-oriented tool surveyed in Section 2.

3 Architecture

Figure 1 presents the end-to-end framework. This section unpacks the design in two parts: four principles that constrain every subsequent decision (Section 3.1), and the five-layer progressive lowering stack with a dimensionally strict type system that realizes them (Section 3.2).

3.1 Design Philosophy

The central question MLSYS·IM addresses is: *Where does the complexity of an ML system come from?* We argue that complexity arises from the non-linear interaction of constraints across six distinct domains (node resources, data movement, algorithms, fleet coordination, operations, and cross-cutting analysis) and that an effective modeling tool must formalize *all six* simultaneously. Four design principles govern MLSYS·IM's approach.

3.1.1 Analytical Speed over Cycle Accuracy

Analytical models that execute in sub-second time enable iterative design-space exploration that cycle-accurate simulators cannot support at comparable speed. By absorbing microarchitectural detail (cache hit rates, warp scheduling) into a single efficiency parameter η, MLSYS·IM achieves sub-second execution per solve. This enables sweeps over thousands of hardware–model–topology combinations in seconds, the kind of rapid "what-if" exploration that Hennessy et al. (2024) identify as essential for quantitative architectural reasoning.

To maintain rigor despite analytical simplification, MLSYS·IM enforces a **"No Magic Numbers" invariant**. Every hardware constant references a datasheet URL and verification date. The H100's peak FP16 throughput is not a bare `989` floating-point literal; it is `989 * TFLOPs / second`, sourced from NVIDIA's published datasheet (NVIDIA Corporation, 2023). This provenance discipline ensures that analytical speed does not come at the cost of reproducibility.

3.1.2 Dimensional Strictness as an Invariant

Dimensional consistency is a pervasive challenge in systems modeling. Mixing gigabits with gigabytes, or omitting the refractive index of fiber in latency calculations, are canonical failure modes that silently corrupt results. MLSYS·IM treats dimensional correctness not as a feature but as a **runtime invariant**, analogous to memory safety in Rust. The framework wraps every physical quantity using the `pint` unit library. If a user attempts to add FLOP/s to GB/s, the framework raises a deterministic `DimensionalityError` before any computation proceeds:

```python
from mlsysim.core.constants import Q_
rate = Q_("989 TFLOPs/s")     # Compute throughput
bw   = Q_("3.35 TB/s")        # Memory bandwidth
rate + bw  # → DimensionalityError
(rate / bw).to("flop/byte")  # → 295 flop/byte (ridge point)
```

Listing 1: **Dimensional Strictness.** Prevents silent unit errors at the API level.

This design eliminates the single most common class of bugs in back-of-the-envelope systems analysis: silent unit conversion errors.[1] Every constant in `constants.py` carries dimensioned units through all downstream computations (e.g., `H100_MEM_BW = 3.35 * TB/s`), preventing unit mismatches at the type level.

3.1.3 Taxonomic Completeness

We define a modeling framework as "complete" only when every fundamental bottleneck to scaling has a mathematical resolver. MLSYS·IM codifies 22 such bottlenecks, which we call *Systems Walls*, organized into six domains: Node (single-accelerator resources), Data (movement and pipelines), Algorithm (scaling and compression), Fleet (multi-node coordination), Operations (economics, sustainability, and safety), and Analysis (cross-cutting diagnostics). Each wall maps to a dedicated resolver with a formal equation grounded in the systems literature. Section 4 presents the full taxonomy with formal definitions, and Section 5 describes how solvers compose.

3.1.4 Demand–Supply Separation

MLSYS·IM enforces a strict separation between *what* a model computes and *where* it runs. A

[1] The most infamous unit-conversion failure is the Mars Climate Orbiter, lost in 1999 because ground software produced thrust data in pound-force seconds while the spacecraft expected newton seconds (Stephenson et al., 1999).

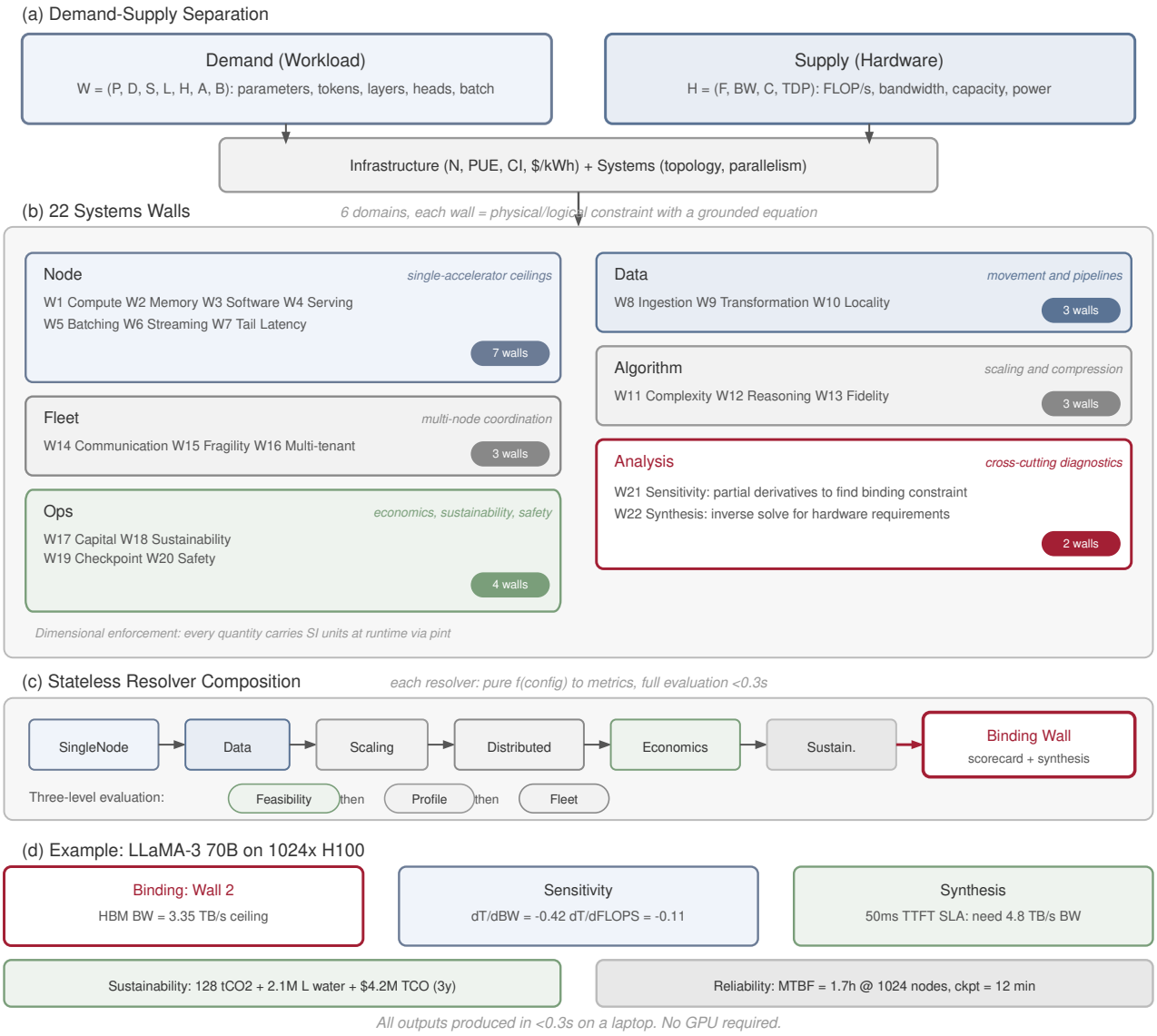

Figure 1: **MLSys · im Framework Overview.** (a) Demand–supply separation decouples workload specifications from hardware capabilities and environmental context. (b) All 22 Systems Walls organized into six domains (Node, Data, Algorithm, Fleet, Operations, Analysis), each grounded in a published equation. (c) Stateless resolver composition chains 25 resolvers to identify binding constraints through a three-level evaluation. (d) Example outputs for LLaMA-3 70B on 1024× H100, produced in <0.3 s on a laptop.

`TransformerWorkload` describes computational demand (parameters, layers, FLOPs, arithmetic intensity) without reference to any specific accelerator. A `HardwareNode` describes physical supply (peak throughput, memory bandwidth, TDP) without reference to any specific model. This decoupling, inspired by the compiler IR philosophy of progressive lowering (Hennessy et al., 2024), enables hardware–software co-design: the same GPT-3 workload can be evaluated against an H100, a TPU v5p, or a hypothetical future accelerator in a single parametric sweep, with all dimensional conversions handled automatically.

3.2 The Progressive Lowering Stack

MLSys · im implements these design principles through a five-layer *Progressive Lowering* stack (Figure 2). Layers A–D are independent input layers that describe demand, supply, context, and topology respectively; they do not depend on one another. Layer E (Resolvers) consumes any combination of layers A–D as needed; a single-node analysis requires only A+B, while a fleet-wide carbon estimate draws on A+B+C+D.

Layer A: Workloads (Demand). A workload is a hardware-agnostic description of computational demand. MLSys · im provides five concrete workload

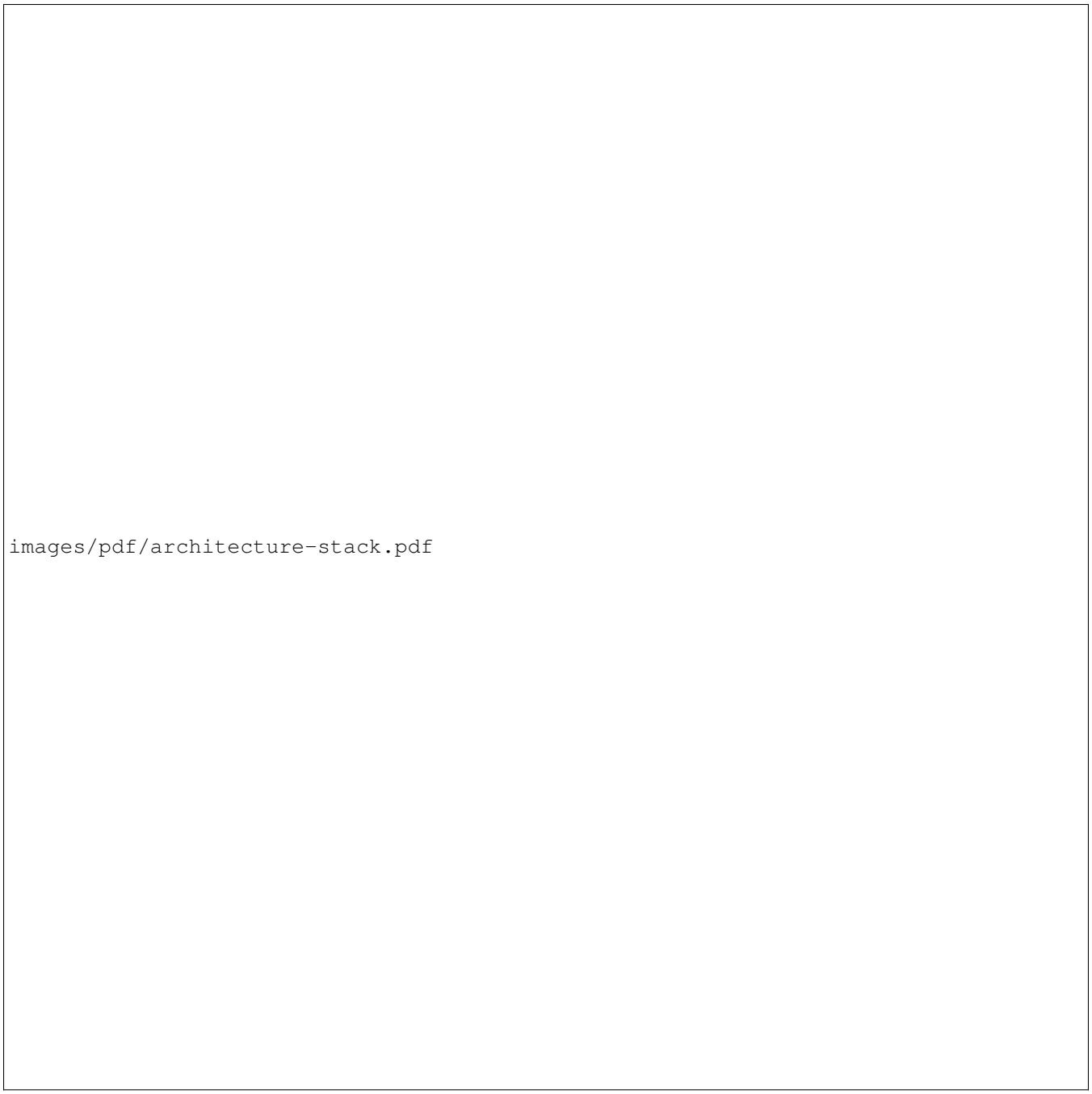

Figure 2: **The MLSYS · IM 5-Layer Architecture.** Layers A–D provide typed inputs: workload demand, hardware supply, infrastructure context, and network topology. Layer A lowers through `lower()` to a hardware-agnostic Computation Graph. Layer E's 25 stateless resolvers consume these inputs and produce a three-level SystemEvaluation scorecard. The red arrow marks the binding constraint identified by the solver chain.

types (Table 3), each exposing a `lower()` method that produces a `ComputationGraph`: an intermediate representation containing total operations, weight bytes, and arithmetic intensity in `flop/byte`. This IR is the contract between demand and supply: it captures *what* must be computed without prescribing *how*.

Layer B: Hardware (Supply). A `HardwareNode` composes four subsystems: `ComputeCore` (peak FLOP/s with a precision-keyed dictionary for FP16, TF32, FP8, INT8), `MemoryHierarchy` (capacity and bandwidth), optional `StorageHierarchy`, and optional `IOInterconnect`. Each node also carries TDP, unit cost, and a kernel dispatch tax. The

Table 3: **Supported Workload Types.** Each workload lowers to a `ComputationGraph` with total FLOPs, weight bytes, and arithmetic intensity.

Workload	Key Parameters	Scaling
Transformer	P, L, H, D, seq. length	$2P$ FLOPs/token
CNN	P, inference FLOPs	Fixed per image
Sparse (MoE)	Total vs. active P, experts	Active P for FLOPs
SSM (Mamba)	P, state dim, D	$O(1)$ state cache
Diffusion	P, denoising steps T	$T\times$ FLOPs/step

`ridge_point()` method computes the Roofline inflection $R = F_{\text{peak}}/BW_{\text{mem}}$ in `flop/byte` (Williams et al., 2009), enabling immediate classification of any lowered workload as compute-bound or memory-bound.

Layer C: Infrastructure (Context). `GridProfile` objects encode regional environmental parameters: carbon intensity (gCO_2/kWh), Power Usage Effectiveness (PUE), and Water Usage Effectiveness (WUE). A `Datacenter` composes a grid profile with rack-level power density constraints. This layer converts raw energy consumption into carbon footprint and water usage, following the methodology of Patterson et al. (2021).

Layer D: Systems (Topology). A `Fleet` composes `Nodes` (accelerator type, count per node, intra-node bandwidth) with a `NetworkFabric` (topology, inter-node bandwidth, latency, oversubscription ratio).[2] This layer enables distributed analysis: the `DistributedModel` decomposes workloads using 4D parallelism (data-parallel DP $\times$ tensor-parallel TP $\times$ pipeline-parallel PP $\times$ expert-parallel EP) and calculates hierarchical AllReduce costs, pipeline bubble fractions, and scaling efficiency.

Layer E: Resolvers (Analysis). Twenty-five stateless resolvers (20 models, 2 solvers, and 3 optimizers) consume demand and supply to produce dimensioned performance metrics. The solver formalism (stateless composition, chaining semantics, and three-level evaluation) is detailed in Section 5.

Reproducible analysis requires reproducible inputs. MLSys·IM organizes vetted specifications into four curated registries collectively called the *MLSys Zoo*: the **Silicon Zoo** (hardware accelerators), the **Model Zoo** (workload architectures), the **Fleet Zoo** (cluster topologies), and the **Infrastructure Zoo** (regional grid profiles). Each entry is a fully typed object whose constants are sourced from datasheets with provenance metadata including source URLs and verification dates. For example, `Hardware.Cloud.H100` returns a `HardwareNode` with all physical quantities dimensioned via `pint`.

The Silicon Zoo spans six orders of magnitude from sub-milliwatt microcontrollers (`Hardware.Tiny.ESP32_S3`, 512 KiB SRAM) to wafer-scale engines (`Hardware.Cloud.CerebrasCS3`, 44 GB on-wafer SRAM, 125 PFLOP/s (Lie, 2022)). The `list(sort_by=)` class method enables programmatic comparison, and users extend any zoo by instantiating new typed objects (Listing 2), ensuring custom entries participate in the same dimensionally strict pipeline as vetted ones.

3.3 The Type System

MLSys·IM's type system is built on Pydantic `BaseModel` classes with `pint Quantity` fields, providing both schema validation and dimensional enforcement at construction time. The composition hierarchy is deliberately shallow: `HardwareNode` aggregates `ComputeCore` and `MemoryHierarchy` as direct fields, not through deep inheritance. This design makes the relationship between a hardware specification and its physical quantities immediately legible:

```python
from mlsysim.hardware.types import *
from mlsysim.core.constants import Q_
node = HardwareNode(
    name="Custom Accelerator",
    release_year=2025,
    compute=ComputeCore(
        peak_flops=Q_("500 TFLOPs/s"),
        precision_flops={"fp8": Q_("1000 TFLOPs/s")
    }),
    memory=MemoryHierarchy(
        capacity=Q_("96 GiB"),
        bandwidth=Q_("4 TB/s")),
    tdp=Q_("500 W"))
print(node.ridge_point())  # → 125 flop/byte
```

Listing 2: **Custom Hardware Node.** Composing a hardware specification with dimensional types.

The `ComputationGraph` IR bridges the demand–supply gap. When a solver calls `workload.lower()`, the workload computes its total operations, weight bytes, and arithmetic intensity, all in dimensioned quantities. For Mixture-of-Experts models, `SparseTransformerWorkload.lower()` uses *active* parameters for FLOPs but *total* parameters for memory footprint, correctly modeling the fundamental decoupling between compute cost and capacity requirements in sparse architectures (Shazeer et al., 2017).

The complete evaluation produces a `SystemEvaluation` scorecard, a single object containing every metric from every resolver, cross-

[2] Throughout this paper, *node* refers to a physical compute node (a server containing one or more accelerators connected via PCIe or NVLink), not a vertex in a graph or network topology.

referenced by wall number. Students can inspect any individual wall or view the aggregate to understand how constraints interact across the full stack.

3.4 Extensibility

The layered architecture is designed for extension at every level. New workload types (e.g., a `RetrievalAugmentedWorkload` for RAG pipelines) require only implementing the `lower()` method to produce a `ComputationGraph`; all existing resolvers then apply without modification. New hardware entries are added to the Silicon Zoo as declarative `HardwareNode` specifications (Listing 2), with no resolver changes needed. New resolvers can be introduced for emerging constraints by implementing the appropriate tier interface: a Tier 1 Model (e.g., a `PrivacyModel` for federated learning overhead), a Tier 2 Solver, or a Tier 3 Optimizer.

By accepting typed inputs and returning dimensioned outputs, the type system enforces correctness at every boundary, ensuring that custom extensions compose safely with existing components. This design ensures that MLSys·IM can track the rapidly evolving ML systems landscape without requiring architectural changes to the core framework.

4 Taxonomy of ML Systems Walls

The ML systems literature is rich with specialized models for specific bottlenecks, from the original Roofline model (Williams et al., 2009) and Chinchilla scaling laws (Hoffmann et al., 2022) to PagedAttention batching limits (Kwon et al., 2023) and datacenter sustainability accounting (Patterson et al., 2021). However, these constraints are typically studied in isolation. We synthesize this disjointed literature into a unified taxonomy of 20 distinct "Walls" plus 2 cross-cutting diagnostic tools (Sensitivity and Synthesis). We borrow the term from the computer architecture tradition (the "memory wall" (Williams et al., 2009), the "power wall" (Hennessy et al., 2024)), where a *wall* denotes a hard physical or logical constraint that bounds system performance and cannot be circumvented by software optimization alone. Codifying these previously disparate equations into a single, composable framework, each wall is resolved by a dedicated resolver (model or solver) that accepts typed inputs and produces dimensionally correct bounds. This integration of theoretical and empirical constraints into a single executable engine has not been done before. The complete taxonomy is summarized in Table 1; the subsections that follow formalize each domain.

Table 4: **Notation.** Symbols used throughout; all quantities carry SI units at runtime via `pint`.

Symbol	Unit	Description
Model & Workload		
P, P_ℓ	params	Total / per-layer parameter count
$\lvert W\rvert$, $\lvert W_\ell\rvert$	bytes	Bytes read per step / per layer
b_{prec}	B/param	Precision (e.g., 2 for FP16)
L, H, D	–	Layers, attention heads, head dim
S	tokens	Sequence length
B	samples	Batch size
K	–	Reasoning steps
C	FLOPs	Training compute ($6PD$)
I	FLOP/B	Arithmetic intensity
Hardware & Infrastructure		
$\text{Peak}_{\text{FLOPS}}$	FLOP/s	Peak accelerator throughput
BW_{HBM}	B/s	HBM bandwidth
BW_{inject}	B/s	Injection BW (wafer-scale)
BW_{link}	B/s	Per-link network bandwidth
N, G	–	Nodes in fleet, GPUs per node
α	s	Per-hop network latency
Efficiency & Utilization		
η	–	HW utilization ($\approx$ MFU)
η_{overlap}	–	Compute–comm overlap
ρ	–	Utilization ratio (queue/data)
β, β_{opt}	–	Bisection BW frac, optimizer multiplier
Parallelism		
TP, PP, DP, EP	–	Tensor, pipeline, data, expert parallel
V, M_{micro}	–	Virtual stages, microbatches
Sustainability		
PUE, WUE	–, L/kWh	Power / Water Usage Effectiveness
CI	gCO_2/kWh	Regional carbon intensity
Key Derived Quantities		
I^*	FLOP/B	Ridge point (Peak/BW_{HBM})
B^*, P^*, D^*	varies	Optimal batch, model, dataset size
τ_{opt}	s	Optimal checkpoint interval

4.1 Node (Single-Accelerator Resources)

The Node walls define what a single accelerator can achieve in isolation. They are the innermost constraints and the first a practitioner should evaluate.

Wall 1: The Compute Wall. Every accelerator has a hard throughput ceiling determined by the number of arithmetic units and the clock frequency. An H100, for example, provides a peak of 989 TFLOP/s at FP16 with Tensor Cores, establishing an upper bound that no software optimization can exceed. The `SingleNodeModel` resolves this wall via Roofline analysis (Williams et al., 2009):

$$T_{\text{compute}} = \frac{\text{OPs}}{\text{Peak}_{\text{FLOPS}} \times \eta} \tag{1}$$

where $\eta \in (0, 1]$ is the hardware utilization efficiency and OPs is the total operation count. Throughout this paper, η denotes the ratio of sustained to peak throughput; the related metric *Model FLOPS Utilization* (MFU) measures only model-useful FLOPs and excludes over-

head such as activation recomputation. For first-order analysis, we treat $\eta \approx$ MFU; the distinction matters only when recomputation or non-model compute is significant. When this wall binds, the only remedy is faster silicon or fewer operations. **Assumptions:** Peak FLOPS is a hard ceiling; η is workload-dependent and must be specified or estimated from benchmark data.

Wall 2: The Memory Wall. High-bandwidth memory (HBM) imposes two ceilings: capacity (the model must fit) and bandwidth (weights must stream to compute units fast enough). An H100 reads HBM at 3.35 TB/s, yet its 989 TFLOP/s demand data at a rate that exceeds this bandwidth for any workload below ∼295 flop/byte, making most LLM inference memory-bound, not compute-bound. During training, techniques like **Low-Rank Adaptation (LoRA)** and **Activation Recomputation** fundamentally alter the capacity constraint by trading compute or parameter trainability for drastically reduced memory footprints. The same `SingleNodeModel` computes (Williams et al., 2009):

$$T_{\text{memory}} = \frac{|W|}{BW_{\text{HBM}}} \tag{2}$$

where $|W|$ is the total bytes read per inference step. The realized execution time is the maximum of the two bounds:

$$T = \max(T_{\text{compute}},\ T_{\text{memory}}) \tag{3}$$

The crossover between compute-bound and memory-bound regimes occurs at the *ridge point*, the arithmetic intensity at which the two ceilings intersect:

$$I^* = \frac{\text{Peak}_{\text{FLOPS}}}{BW_{\text{HBM}}} \quad \text{(flop/byte)} \tag{4}$$

Workloads with arithmetic intensity $I < I^*$ are memory-bound; those with $I > I^*$ are compute-bound. **Assumptions:** Peak FLOPS and HBM bandwidth are hard ceilings; MFU accounts for software inefficiency via η.

Wall 3: The Software Wall. The gap between peak and achieved FLOP/s is typically larger than the gap between hardware generations. Most naïve implementations achieve only 30% of peak throughput; the remaining 70% is lost to redundant memory traffic, low warp occupancy, and unfused operations. The `EfficiencyModel` models this as a multiplicative efficiency factor (Chowdhery et al., 2023):

$$\eta = \frac{\text{FLOPS}_{\text{achieved}}}{\text{Peak}_{\text{FLOPS}}} \tag{5}$$

where $\eta \in (0, 1]$ modulates the Roofline ceiling, reducing the effective peak from $\text{Peak}_{\text{FLOPS}}$ to $\eta \times \text{Peak}_{\text{FLOPS}}$. FlashAttention (Dao et al., 2022), for example, achieves a 2.5× speedup over standard attention by fusing memory-bound operations into a single kernel pass, effectively raising η from ∼0.3 to ∼0.75 for attention layers. When this wall binds, better kernels, not bigger chips, are the remedy. **Assumption:** η is a single scalar that aggregates all software inefficiencies; in practice, different operations (GEMM vs. attention vs. normalization) achieve different utilization on the same silicon. To avoid circularity when η is unknown, the `EfficiencyModel` provides default ranges derived from published benchmarks: $\eta \approx 0.30$–0.45 for large-scale training (Chowdhery et al., 2023; Llama Team, AI @ Meta, 2024), $\eta \approx 0.50$–0.60 for highly optimized GEMM-heavy workloads, and $\eta < 0.10$ for memory-bound inference decode. Students can use these defaults as starting points and refine as they gather profiling data.

Wall 4: The Serving Wall. Autoregressive LLM inference exhibits two distinct phases with fundamentally different Roofline characteristics (Pope et al., 2023). For a 70B model on two tensor-parallel H100s, a 512-token prefill takes ∼72 ms (compute-bound) while each decode token costs ∼21 ms (memory-bound). Prefill processes all input tokens in parallel (∼7 066 tokens/s), while decode generates one token per memory-bandwidth pass (∼48 tokens/s)—a ∼148× throughput gap. The `ServingModel` decomposes end-to-end inference latency as:

$$T_{\text{prefill}} = \frac{2P \cdot S_{\text{in}}}{\text{Peak}_{\text{FLOPS}} \times \eta} \quad \text{(compute-bound)} \tag{6}$$

$$T_{\text{decode}} = \frac{|W|}{BW_{\text{HBM}}} \quad \text{(memory-bound)} \tag{7}$$

where S_{in} is the input sequence length, P is the parameter count, and $|W|$ includes both model weights and KV-cache reads (which grow with batch size and context length: $|W| = |W_{\text{model}}| + |W_{\text{KV}}|$). The prefill phase processes all input tokens in parallel and is compute-bound; the decode phase generates one token at a time and is memory-bandwidth-bound. The solver incorporates modern paradigms including **Prompt Caching** (prefix caching (Zheng et al., 2024), which reduces TTFT by skipping prefill for previously computed KV-cache entries), **Speculative Decoding** (probability-weighted verification using a smaller draft model (Leviathan et al., 2023)), and **Disaggregated Serving** (phase splitting onto different hardware with KV-cache network transfer (Patel et al., 2024)). This duality explains why batching strategies that improve prefill throughput may have no effect on decode latency, because the two phases are bound by different resources. **Assumptions:** Prefill is compute-bound for sequence lengths $S_{\text{in}} \gg 1$; decode is memory-bandwidth-bound at batch size 1. At large batch sizes, decode transitions toward compute-bound; the solver models this crossover via the Roofline.

Wall 5: The Batching Wall. Serving throughput depends on how many requests share the accelerator simultaneously, but memory-bound decode means that each additional request in the batch adds KV-cache pressure without reducing per-token latency. Static batching wastes memory through external fragmentation: each request reserves a contiguous KV-cache block sized for maximum sequence length, even if most requests finish early. The `ContinuousBatchingModel` models iteration-level scheduling with non-contiguous allocation via PagedAttention (Kwon et al., 2023):

$$\text{KV}_{\text{paged}} = 2 \times L \times H \times D \times \lceil S/p \rceil \times p \times B \times b \quad (8)$$

where L is layers, H is KV heads, D is head dimension, S is sequence length, p is page size in tokens, B is batch size, and b is bytes per element. Internal fragmentation is bounded by the last page, eliminating the 40–50% external fragmentation of contiguous allocation. **Assumptions:** Decode is memory-bandwidth-bound for batch $\geq$ 1; static batching baseline assumes 50% fragmentation waste.

Wall 6: The Streaming Wall. Wafer-scale architectures (Lie, 2022) (e.g., Cerebras CS-3) invert the conventional memory hierarchy: activations reside on-wafer in SRAM while model weights stream from external MemoryX nodes, shifting the bottleneck from HBM bandwidth to injection interconnect bandwidth. The `WeightStreamingModel` models this as:

$$T_{\text{layer}} = \max\left(\frac{|W_\ell|}{BW_{\text{inject}}}, \frac{2P_\ell \times B}{\text{Peak} \times \eta}\right) \quad (9)$$

where $|W_\ell|$ is the layer weight size in bytes, $P_\ell = |W_\ell|/b_{\text{prec}}$ is the parameter count (with b_{prec} bytes per element), and the factor of 2 accounts for the multiply-accumulate FLOPs per parameter. The two terms inside the max represent the injection time (weight delivery) and the compute time (matrix arithmetic) for one layer. When B is small, weight injection dominates and the compute engine sits idle; when B is large, compute dominates and the injection link sits idle. Setting the two terms equal and substituting $|W_\ell| = P_\ell \times b_{\text{prec}}$ yields the optimal batch size $B^* = (b_{\text{prec}} \times \text{Peak} \times \eta)/(2 \times BW_{\text{inject}})$. This result depends only on numerical precision, peak compute, and injection bandwidth, independent of layer size. This is the unique operating point where injection and compute perfectly overlap, maximizing utilization of both resources. **Assumptions:** Layer weights dominate the injection payload; 10% overhead is reserved for working memory; perfect within-layer pipelining is assumed.

Wall 7: The Tail Latency Wall. At scale, P99 latency governs user experience, not the median. A single slow replica in a fan-out of 100 services dominates end-to-end response time. The `TailLatencyModel` models inference replicas as an M/M/c queue using the Erlang-C formula (Dean and Barroso, 2013):

$$\mathbb{P}[\text{wait}] = \frac{(c\rho)^c/c! \cdot (1-\rho)^{-1}}{\sum_{k=0}^{c-1}(c\rho)^k/k! + (c\rho)^c/c! \cdot (1-\rho)^{-1}} \quad (10)$$

where c is the number of replicas, $\rho = \lambda/(c\mu)$ is per-server utilization, and λ, μ are arrival and service rates. P99 latency grows non-linearly as $\rho \to 1$: for a 10-replica deployment, P99 at 80% utilization is $\sim$3$\times$ the service time; at 95%, it rises to $\sim$10$\times$, making the distinction between the two the difference between meeting and violating a latency SLA. **Assumption:** The M/M/c model is intentionally optimistic; real traffic is bursty and service times are heavy-tailed, so M/M/c provides a lower bound on tail latency. If the system is unstable under M/M/c, it will certainly be unstable under real traffic.

4.2 Data (Movement & Pipelines)

The Data walls govern how data moves *to* the accelerator. A node that is locally unconstrained can still starve if the data pipeline cannot keep pace.

Wall 8: The Ingestion Wall. Storage I/O must supply training samples at the rate the accelerator consumes them. This wall binds most often in vision tasks, where raw images are large: an 8-GPU DGX A100 training ResNet-50 at batch 2,048 demands $\sim$2 GB/s of compressed image reads per step, easily saturating a single NVMe SSD and requiring striped or cached storage. The `DataModel` computes the demand–supply ratio (Mohan et al., 2021):

$$\rho_{\text{data}} = \frac{BW_{\text{demand}}}{BW_{\text{supply}}} \quad (11)$$

When $\rho_{\text{data}} > 1$, the accelerator stalls waiting for data. BW_{demand} is the product of batch size, sample size, and step rate; BW_{supply} is the effective throughput of the storage subsystem after accounting for read amplification and caching. For LLM training on tokenized text, BW_{demand} is typically negligible ($<$1 MB/s); this wall is primarily relevant for vision, audio, and video workloads. **Assumption:** Storage bandwidth is the bottleneck, not network I/O (single-node training).

Wall 9: The Transformation Wall. JPEG decoding, tokenization, and augmentation execute on CPU cores, not on the accelerator. When the CPU preprocessing pipeline cannot keep pace, the accelerator stalls even if storage bandwidth is abundant. On ImageNet, a single CPU worker typically decodes and augments $\sim$850 images/s; 64 workers therefore cap the

pipeline at ∼54,400 images/s, which an 8-GPU DGX A100 can exceed at large batch sizes (see Case S2, Section 7). The `TransformationModel` quantifies this bottleneck (Murray et al., 2021):

$$T_{\text{transform}} = \frac{B}{R_{\text{cpu}}} \tag{12}$$

where B is the batch size (in samples) and R_{cpu} is the aggregate CPU preprocessing rate (in samples/s), computed as $R_{\text{cpu}} = N_{\text{workers}} \times r_{\text{worker}}$ where r_{worker} is the per-worker throughput after all transformations (decode, augment, normalize). **Assumption:** Preprocessing is CPU-bound and scales linearly with worker count up to core saturation.

Wall 10: The Locality Wall. Network topology determines the effective bandwidth available between any two nodes in the cluster. The `TopologyModel` models this through the *bisection bandwidth fraction* β (Leiserson, 1985), which varies by topology: Fat-Tree provides full bisection ($\beta = 1.0$), Dragonfly achieves $\beta \approx 0.85$, and 3D Torus yields $\beta \approx 0.67$. The effective inter-node bandwidth is:

$$BW_{\text{eff}} = \frac{BW_{\text{link}} \times \beta}{\text{oversubscription}} \tag{13}$$

This wall becomes binding when collective communication patterns demand bandwidth that the topology cannot supply at scale. For example, a 3D Torus cluster with 400 Gb/s links delivers an effective bisection bandwidth of $400 \times 0.67 = 268$ Gb/s per node; a Fat-Tree with the same links provides the full 400 Gb/s ($\beta = 1$), a 1.5× advantage that compounds across multi-hop AllReduce patterns. We place the Locality Wall in the Data domain rather than Fleet because it models the *physical topology constraint* on data movement (bisection bandwidth, oversubscription), whereas the Communication Wall (Wall 14, Fleet) models the *algorithmic cost* of specific collectives (AllReduce, All-to-All) that run atop that topology. The two walls interact but address distinct levels of abstraction. **Assumption:** β values are topology-specific constants; real networks may exhibit dynamic congestion not captured by this static model.

4.3 Algorithm (Scaling & Compression)

The Algorithm walls arise not from hardware but from the mathematics of learning itself. They determine how much computation a workload *requires*, independent of the silicon that executes it.

Wall 11: The Complexity Wall. Chinchilla scaling laws (Hoffmann et al., 2022) establish that training compute scales jointly with model size P and dataset size D: doubling the parameters requires approximately doubling the tokens to remain compute-optimal. The `ScalingModel` implements:

$$C = 6PD \quad \text{(total training FLOPs)} \tag{14}$$

$$D^* \approx 20P \quad \text{(compute-optimal tokens)} \tag{15}$$

$$P^* = \sqrt{\frac{C}{120}} \quad \text{(optimal model size for budget } C\text{)} \tag{16}$$

These relations allow students to reason backward from a compute budget to the largest model that can be trained optimally, or forward from a model size to the minimum viable training cluster. For example, a budget of 10^{24} FLOPs yields $P^* \approx 91$B parameters with $D^* \approx 1.8$T tokens; Chinchilla (70B, 1.4T tokens) was trained near this optimal frontier (Hoffmann et al., 2022). **Assumption:** Scaling law coefficients are fitted to published training runs; extrapolation beyond the fitted range is flagged.

Wall 12: The Reasoning Wall. Inference-time compute scaling introduces a cost that grows linearly with the number of reasoning steps K. The `InferenceScalingModel` models this as (Snell et al., 2025):

$$T_{\text{reason}} = K \times T_{\text{step}}(P, S_{\text{context}}) \tag{17}$$

where T_{step} is the per-step latency, itself a function of model size P and context length S_{context}. In practice, K varies dramatically by strategy: simple chain-of-thought uses $K \approx 3$–8, best-of-N sampling uses $K = N$ (typically 8–64), and tree-search methods like Monte Carlo Tree Search can require $K > 100$ (Snell et al., 2025). A K=8 chain-of-thought query costs ∼8× more compute than a single-pass answer, fundamentally altering serving economics (see Case S1, Section 7). **Assumption:** Each reasoning step is an independent decode sequence; KV-cache is not shared across steps.

Wall 13: The Fidelity Wall. Compression trades model fidelity for efficiency: quantization reduces precision while pruning removes weights entirely. The accuracy–efficiency frontier is task- and architecture-dependent. The `CompressionModel` quantifies the two primary mechanisms (Han et al., 2016; Gholami et al., 2021):

$$r_{\text{quant}} = \frac{b_{\text{base}}}{b_{\text{target}}} \quad \text{(quantization ratio, e.g., } 16/4 = 4\times\text{)} \tag{18}$$

$$r_{\text{prune}} = \frac{1}{1-s} \quad \text{(memory reduction at sparsity } s\text{)} \tag{19}$$

Here b_{base} is the baseline precision (typically 16 for FP16/BF16 models, not 32) and b_{target} is the quantized precision. Critically, quantization reduces memory reads

but not FLOPs, shifting the arithmetic intensity rightward on the Roofline by a factor of r_{quant} and potentially crossing the ridge point from memory-bound to compute-bound. For pruning, r_{prune} gives the *memory reduction* ratio; actual compute speedup depends on sparsity structure. Unstructured sparsity yields no acceleration on current GPUs, while 2:4 structured sparsity (NVIDIA Corporation, 2023) provides a $2\times$ throughput gain via Sparse Tensor Cores. Post-training quantization methods such as GPTQ (Frantar et al., 2023) and AWQ (Lin et al., 2024) demonstrate that 4-bit quantization can preserve most accuracy for large language models, making $r_{\text{quant}} = 4\times$ a practical operating point. The accuracy impact Δ_{acc} is modeled as a configurable function, since the fidelity–compression frontier varies by architecture and task. **Assumptions:** Accuracy degradation follows empirical curves from Gholami et al. (2021); pruning compute speedup requires structured sparsity with hardware support.

4.4 Fleet (Multi-Node Coordination)

The Fleet walls arise when systems scale beyond a single node, requiring coordination across multiple accelerators connected by network fabric.

Wall 14: The Communication Wall. Distributed training requires gradient synchronization across N nodes, and the cost of that synchronization grows with both message size and node count. The `DistributedModel` models the dominant collective operations. For Ring AllReduce (Shoeybi et al., 2019) and its **ZeRO/FSDP** partitioned equivalents (Reduce-Scatter and All-Gather):

$$T_{\text{ring}} = \frac{2(N-1)}{N} \cdot \frac{M}{B_{\text{link}}} + 2(N-1) \cdot \alpha \tag{20}$$

where M is the message size, B_{link} is the per-link bandwidth, and α is the per-hop latency. Modern clusters use **hierarchical AllReduce** to exploit the bandwidth asymmetry between intra-node interconnect (e.g., NVLink at 900 GB/s) and inter-node fabric (e.g., InfiniBand at 50 GB/s per port, an $18\times$ gap). The `DistributedModel` implements a two-level model:

$$T_{\text{hier}} = T_{\text{intra}}(BW_{\text{NVLink}}, G) + T_{\text{inter}}(BW_{\text{IB}}, N) \tag{21}$$

where G is GPUs per node and N is the node count. Each level applies the ring formula (Equation (20)) at its respective bandwidth, making the inter-node phase the dominant cost at scale. In practice, modern frameworks also use **Compute/Communication Overlap**, hiding network latency behind backward pass computation. The `DistributedModel` models this with an overlap efficiency parameter $\eta_{\text{overlap}} \in [0, 1]$ (default 0.85, reflecting typical Megatron-LM behavior), yielding an exposed communication cost of $(1 - \eta_{\text{overlap}}) \cdot T_{\text{comm}}$. Pipeline parallelism (Narayanan et al., 2021) introduces a bubble overhead:

$$B_{\text{pipeline}} = \frac{P_{\text{stages}} - 1}{V \cdot M_{\text{micro}} + P_{\text{stages}} - 1} \tag{22}$$

where V is the number of virtual pipeline stages and M_{micro} is the number of microbatches. For Mixture-of-Experts All-to-All dispatch:

$$T_{\text{a2a}} = \frac{(N-1)}{N} \cdot \frac{M}{B_{\text{link}}} + (N-1) \cdot \alpha \tag{23}$$

Wall 15: The Fragility Wall. Component failures are inevitable at scale. If each node has a mean time between failures of $\text{MTBF}_{\text{node}}$, then a cluster of N nodes has (Daly, 2006):

$$\text{MTBF}_{\text{cluster}} = \frac{\text{MTBF}_{\text{node}}}{N} \tag{24}$$

The probability of at least one failure during a training run of duration T is:

$$P(\geq 1 \text{ failure}) = 1 - e^{-T/\text{MTBF}_{\text{cluster}}} \tag{25}$$

The Young-Daly formula (Young, 1974; Daly, 2006) gives the optimal checkpoint interval:

$$\tau_{\text{opt}} = \sqrt{2\delta \cdot \text{MTBF}_{\text{cluster}}} \tag{26}$$

where δ is the time to write one checkpoint. The `ReliabilityModel` uses these relations to estimate the fraction of compute lost to checkpointing and recovery. **Assumption:** Failures are independent and exponentially distributed (memoryless).

Wall 16: The Multi-tenant Wall. Production GPU clusters are rarely dedicated to a single job. Shared clusters introduce queueing delays that grow hyperbolically as utilization approaches 1.0, creating a tension between maximizing hardware utilization and minimizing researcher wait times. The `OrchestrationModel` models job wait times using an M/D/1 queue (Little, 1961):

$$T_{\text{wait}} = \frac{\rho}{2\mu(1-\rho)} \tag{27}$$

where $\rho = \lambda/\mu$ is the cluster utilization, λ is the job arrival rate, and μ is the service rate. As $\rho \to 1$, wait times diverge hyperbolically: a cluster at 90% utilization has $5\times$ the queue depth of one at 50%. **Assumption:** Job durations are approximately deterministic, which is reasonable for large training runs with predictable step times.

4.5 Operations (Cost, Carbon & Safety)

The Operations walls capture constraints that are not about *how fast* a system runs but *whether it should run at all*: economic viability, environmental impact, checkpoint overhead, and responsible deployment.

Wall 17: The Capital Wall. Performance analysis is incomplete without economic constraints. A 1 024-GPU H100 cluster costs ∼$30.7M in CapEx alone; amortized over three years, the hardware cost of a single 30-day training run exceeds $800K before adding electricity. The `EconomicsModel` computes total cost of ownership (Barroso et al., 2018):

$$\text{TCO} = \text{CapEx} + \text{OpEx}_{\text{energy}} + \text{OpEx}_{\text{maint}} \tag{28}$$

where $\text{OpEx}_{\text{energy}} = E_{\text{total}} \times P_{\text{kWh}}$ converts total energy consumption to dollar cost at the regional electricity price. At scale, CapEx dominates: electricity typically accounts for less than 10% of total run cost, making hardware utilization (MFU) the primary lever for cost efficiency. **Assumption:** Linear amortization over a 3–5 year hardware lifetime.

Wall 18: The Sustainability Wall. Regional grid carbon intensity varies by more than an order of magnitude, making geography a first-order systems design variable. The `SustainabilityModel` converts energy into environmental impact (Patterson et al., 2021):

$$E_{\text{total}} = E_{\text{IT}} \times \text{PUE} \tag{29}$$

$$\text{CO}_2 = E_{\text{total}} \times \text{CI}_{\text{region}} \quad (\text{gCO}_2/\text{kWh}) \tag{30}$$

$$\text{H}_2\text{O} = E_{\text{total}} \times \text{WUE} \quad (\text{L/kWh}) \tag{31}$$

where PUE is the power usage effectiveness of the datacenter, $\text{CI}_{\text{region}}$ is the carbon intensity of the local grid, and WUE is the water usage effectiveness. **Assumption:** Grid carbon intensity is a static regional constant; temporal variation (e.g., renewable intermittency) is not modeled. Energy-proportional power follows Barroso and Hölzle (2007): idle power is 30% of TDP, with the remaining 70% scaling linearly with MFU.

Wall 19: The Checkpoint Wall. Long-running training jobs must periodically save model state (weights and optimizer states) to persistent storage, incurring an I/O penalty that directly reduces effective MFU. The `CheckpointModel` models the I/O burst penalty (Eisenman et al., 2022):

$$\text{MFU}_{\text{penalty}} = \frac{T_{\text{write}}}{T_{\text{interval}}} = \frac{|W| \times \beta_{\text{opt}} / BW_{\text{storage}}}{T_{\text{ckpt\_interval}}} \tag{32}$$

where $|W|$ is the model weight size in bytes, and β_{opt} is the optimizer state multiplier, the ratio of total checkpoint bytes to model weight bytes. For mixed-precision Adam with FP16 weights, $\beta_{\text{opt}} \approx 7$ (FP32 master weights at 4 bytes + FP32 momentum at 4 bytes + FP32 variance at 4 bytes, plus FP16 model weights at 2 bytes, totaling 14 bytes per parameter vs. 2 bytes for the FP16 model alone). Gradients are ephemeral and not checkpointed. For a 70B-parameter model, the checkpoint size is 70B × 14 B/param ≈ 1 TB, making storage bandwidth the binding constraint during I/O bursts.

Wall 20: The Safety Wall. Privacy and fairness guarantees impose quantifiable computational overhead. The `ResponsibleEngineeringModel` models the cost of differential privacy via DP-SGD (Abadi et al., 2016), where the noise multiplier scales inversely with the privacy budget:

$$\sigma \propto \frac{1}{\varepsilon} \tag{33}$$

At moderate privacy ($\varepsilon = 1$), DP-SGD typically incurs a ∼3× training slowdown due to per-sample gradient clipping and noise addition; at strong privacy ($\varepsilon = 0.1$), the overhead can exceed 10×. The solver models this as a compute multiplier: $T_{\text{DP}} = T_{\text{base}} \times f(\varepsilon)$, where $f(\varepsilon)$ is empirically calibrated from published DP-SGD benchmarks. Fairness constraints impose a complementary data requirement: sufficient representation of minority subgroups demands additional training data proportional to $O(1/p_{\text{min}})$ where p_{min} is the smallest subgroup prevalence. Unlike the other walls, Wall 20 is more qualitative than quantitative—the overhead depends heavily on task, model architecture, and privacy mechanism. The solver reports the compute multiplier as a range rather than a point estimate. **Assumption:** Privacy budget ε is a hard constraint; the solver reports the compute multiplier, not the privacy guarantee.

4.6 Analysis (Cross-Cutting Diagnostics)

The preceding 20 walls each model a specific physical or logical constraint. The final two entries are *diagnostic tools* rather than walls in the strict sense: they operate *across* the taxonomy rather than within a single domain, providing analysis capabilities that span all walls. The Sensitivity tool identifies which wall is binding, and the Synthesis tool derives minimum hardware from SLA requirements.

Wall 21: The Sensitivity Wall. Optimization is effective only when directed at the binding constraint; improving a non-bottleneck parameter yields no measurable gain. The `SensitivitySolver` identifies the binding constraint by computing partial derivatives of end-to-end latency with respect to each hardware parameter (Williams et al., 2009):

$$\frac{\partial T}{\partial BW_{\text{mem}}}, \quad \frac{\partial T}{\partial \text{Peak}_{\text{FLOPS}}}, \quad \frac{\partial T}{\partial BW_{\text{net}}}, \quad \ldots \tag{34}$$

The parameter with the largest sensitivity is the binding constraint, that is, the single upgrade that would yield

the greatest performance improvement. For LLaMA-70B inference on an A100, the solver reports $\partial T/\partial BW_{\text{mem}} = -0.88$ versus $\partial T/\partial \text{Peak} = -0.06$, confirming that a 10% bandwidth upgrade yields 14× more latency reduction than a 10% FLOPS upgrade. This transforms "where should I invest?" from intuition into calculation. **Assumption:** Finite-difference approximation with 1% perturbation; second-order effects are ignored. The binding constraint is identified as the parameter with the largest normalized gradient $|\partial T/\partial x_i| \cdot (x_i/T)$.

Wall 22: The Synthesis Wall. The `SynthesisSolver` addresses the inverse problem: given a service-level objective (e.g., 50 ms inter-token latency), it derives the minimum hardware specifications required to satisfy it (Kwon et al., 2023):

$$BW_{\text{required}} = \frac{|W|}{T_{\text{target}}} \tag{35}$$

$$\text{FLOPS}_{\text{required}} = \frac{\text{OPs}}{T_{\text{target}} \times \eta} \tag{36}$$

This enables hardware-software co-design: engineers specify an SLA and the solver derives the minimum hardware that satisfies it. For LLaMA-70B at a 50 ms inter-token latency target, the solver yields $BW_{\text{required}} = 140\ \text{GB}/50\ \text{ms} = 2\,800\ \text{GB/s}$—1.4× the A100's 2 TB/s, confirming that this SLA requires at least two tensor-parallel A100s. **Assumption:** Hardware parameters are independently adjustable; co-design coupling between FLOPS and bandwidth is not modeled.

5 The 3-Tier Resolver Architecture

The walls define *what* constrains a system. This section formalizes *how* resolvers compose to produce end-to-end system evaluations. To clarify the mathematical intent of each component, MLSYS·IM organizes its analytical tools into a strict 3-tier taxonomy: **Models** (evaluate), **Solvers** (diagnose), and **Optimizers** (search).

5.1 Tier 1: Analytical Models

Analytical models act as the "physics engine." They perform forward evaluation ($Y = f(X)$) to determine the physical and logical consequences of a specific system configuration. For example, the `ServingModel` calculates the exact time-to-first-token for a given LLM and GPU pair. Models are purely deterministic and make no decisions; they comprise the first 20 resolvers in our taxonomy.

5.2 Tier 2: Analysis Solvers

Analysis solvers act as the "math engine." They perform algebraic inversion or calculus ($X = f^{-1}(Y)$ or ∇f) to find the exact parameter required to hit a specific target. For example, the `SynthesisSolver` takes a target latency SLA and works backward to derive the minimum memory bandwidth required.

5.3 Tier 3: Optimizers

Optimizers act as the "engineering engine." They perform constrained design-space search ($\max f(X)$ s.t. $g(X) \leq c$) to find the best configuration among many valid options. Unlike Models and Solvers, which map directly to individual walls, Optimizers operate across the entire taxonomy to navigate complex constraint spaces. For example, the `ParallelismOptimizer` sweeps all valid 3D tensor/pipeline/data parallel splits to maximize Model FLOPs Utilization (MFU) on a given cluster, while the `BatchingOptimizer` searches for the maximum batch size that satisfies a P99 queueing latency SLA.

5.4 Stateless Composition and Chaining

Every resolver in MLSYS·IM is a pure function: it accepts a typed configuration, performs analytical computation, and returns a typed result object (a Pydantic `BaseModel` with dimensioned fields). Resolvers maintain no hidden state between invocations. Because they share a common type system, the output of one tier feeds naturally into the next. A full-stack analysis composes resolvers in sequence to resolve complex design questions. For example, determining the financial cost of training an optimally-sized model on a frontier cluster requires chaining algorithmic scaling, distributed execution, and macro-economics:

$$\text{Scaling} \xrightarrow{\mathcal{R}_1} \text{Distributed} \xrightarrow{\mathcal{R}_2} \text{Economics} \xrightarrow{\mathcal{R}_3} \text{Sustainability} \tag{37}$$

Listing 3 demonstrates this exact chain in MLSYS·IM. The `ScalingModel` calculates the optimal model size for a given compute budget ($\mathcal{R}_1$). The `DistributedModel` takes that workload and computes the real-world execution time on an 8,192-GPU fleet, factoring in 3D parallelism overhead ($\mathcal{R}_2$). Finally, the `EconomicsModel` converts that execution time into a Total Cost of Ownership ($\mathcal{R}_3$).

Each link in the chain preserves dimensional correctness: units propagate through the computation, and any mismatch raises an immediate error rather than producing a silently wrong result.

5.5 The SystemEvaluation Scorecard

MLSYS·IM provides a `Scenario.evaluate()` entry point that orchestrates solver composition automatically through a three-level evaluation:

```python
import mlsysim
from mlsysim import ScalingModel, DistributedModel, EconomicsModel

# 1. Algorithm: Find optimal parameters for a fixed compute budget
budget = mlsysim.Q_("4e24 FLOP")  # ~100K H100-days at 50% MFU
optimal = ScalingModel().solve(compute_budget=budget)

# Instantiate the demand (Layer A: Workload)
model = mlsysim.TransformerWorkload(
    name="Frontier-Model",
    parameters=optimal.optimal_parameters,
    layers=80, hidden_dim=8192, heads=64
)

# 2. Fleet: Evaluate on a massive 8K GPU cluster (Layer D: Supply/Topology)
fleet = mlsysim.Systems.Clusters.Frontier_8K
perf = DistributedModel().solve(
    model, fleet,
    batch_size=4096, tp_size=8, pp_size=4
)

# 3. The Capital: Calculate TCO for the resulting training time
duration_days = perf.step_latency_total * optimal.optimal_tokens / 4096
tco = EconomicsModel().solve(fleet, duration_days=duration_days.to('day').magnitude)

print(f"Scaling Efficiency: {perf.scaling_efficiency:.1%}")
print(f"Total Job Cost: ${tco.tco_usd:,.2f}")
```

Listing 3: **Solver Composition.** Bridging algorithmic scaling, distributed execution, and fleet economics in a single executable chain.

Level 1: Feasibility. Does the model fit? Can the data pipeline keep pace? The framework checks memory capacity against model size, ingestion bandwidth against training throughput, and reports any wall where demand exceeds supply.

Level 2: Performance. What are the achievable latency, throughput, and utilization? The Roofline analysis (Equation (3)), communication modeling (Equation (20)), and pipeline bubble (Equation (22)) combine to produce end-to-end training step time.

Level 3: Macro. What does it cost, and what does it emit? TCO (Equation (28)), carbon (Equation (29)), and responsibility overhead (Equation (33)) are computed from the performance results.

The three levels are evaluated in order; a feasibility failure at Level 1 short-circuits the evaluation and reports the binding constraint. This ordering reflects the dependency structure: communication optimization is irrelevant if the model exceeds available memory. The complete implementation details and key assumptions for each solver are documented alongside their respective walls in Section 4.

6 Validation

An analytical framework earns trust through transparent confrontation with empirical ground truth. We validate MLSys·IM along two axes: *accuracy* against published benchmarks, and *speed* relative to alternative modeling tools. Crucially, our goal in validation is not to "curve-fit" the model to perfectly match empirical results by introducing arbitrary fudge factors. Rather, it is to demonstrate that a first-principles application of physics and generalized efficiency constants places the prediction squarely in the correct ballpark. As we argue in Section 9, a model that is off by 10% because it structurally ignores L2 cache misses is functioning exactly as intended: it trades cycle-level precision for architectural intuition. The anchors below illustrate this philosophy.

6.1 Empirical Anchors

We anchor MLSys·IM predictions against seven published benchmarks spanning single-node training, distributed training, inference, scaling laws, sustainability, and automated design-space optimization.

Anchor 1: MLPerf ResNet-50 on DGX A100 (Single-Node Training). For ResNet-50 training on a DGX A100 node (8× A100 GPUs with NVLink) at batch size 2048, MLSys·IM predicts a throughput of approximately 38,500 samples/s using the `SingleNodeModel` with hardware utilization $\eta = 0.19$ and 8-way data parallelism within the node. The low η reflects that ResNet-50's small convolution kernels cannot saturate the A100's tensor cores the way large GEMM-dominated Transformer layers can; published MLPerf submissions show ∼58 TFLOP/s per A100 out of 312 TFLOP/s peak. The MLPerf Training v4.0 NVIDIA closed-division submission reports 38,200 samples/s for this 8-GPU configuration (Mattson et al., 2020), yielding a prediction error of 0.9%. Per-GPU throughput is ∼4,800 samples/s, consistent with the A100's Roofline ceiling for ResNet-50's arithmetic intensity. This validates the Roofline-based throughput model (Williams et al., 2009) at the core of MLSys·IM's single-node solver.

Anchor 2: vLLM Llama-2-70B on H100 (Inference). For autoregressive decoding of Llama-2-70B (FP16, batch size 1), the model weights total 140 GB, requiring at minimum two tensor-parallel H100s (each with 80 GB HBM3). MLSys·IM's first-order estimate divides total weights by aggregate bandwidth: $140\ \text{GB}/(2 \times 3.35\ \text{TB/s}) =$

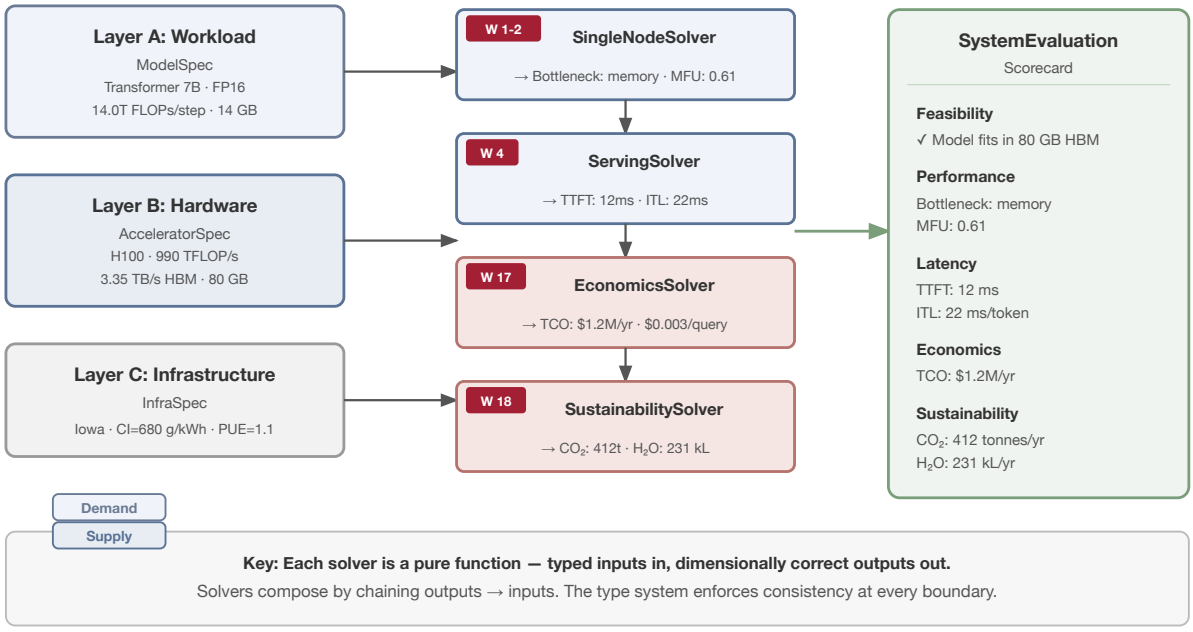

Figure 3: **Resolver Composition.** Three input layers feed four resolvers. Each resolver is a pure function: typed inputs in, dimensionally correct outputs out. The scorecard aggregates three evaluation levels: Feasibility, Performance, and Macro (economics, sustainability, safety).

20.9 ms for the weight-read phase alone. Adding KV-cache reads, attention computation, NVLink synchronization, and framework scheduling overhead, the predicted end-to-end ITL is approximately 43 ms (NVIDIA Corporation, 2023). Published vLLM benchmarks for this configuration report ITL values in the 40–50 ms range (Kwon et al., 2023), confirming that decode-phase LLM inference is memory-bandwidth-bound and that the overhead multiplier (∼2× over the pure bandwidth floor) is consistent across deployments.

Anchor 3: Llama 3 Training at 16K H100s (Distributed Training). Meta's Llama 3 training report (Llama Team, AI @ Meta, 2024) documents achieving 38–43% MFU on 16,384 H100 GPUs with 4D parallelism (DP×TP×PP×CP). We configure MLSys·IM's `DistributedModel` with an equivalent fleet (2,048 nodes × 8 H100s, 400 Gb/s InfiniBand per node, TP=8, PP=4, DP=512) training a 405B-parameter model with system-level efficiency $\eta = 0.42$. This η captures kernel utilization, stragglers, load imbalance, checkpointing pauses, and thermal throttling—effects the analytical communication model does not represent. After accounting for pipeline bubble overhead (Equation (22)), hierarchical AllReduce cost (Equation (21)), and compute–communication overlap ($\eta_{\text{overlap}} = 0.85$), MLSys·IM predicts a scaling efficiency of ∼95%, yielding an aggregate MFU of $0.95 \times 0.42 = 40.0\%$, within the reported 38–43% range. This validates the distributed training model at production scale.

Anchor 4: PaLM Scaling Efficiency (Communication Overhead). Google's PaLM report (Chowdhery et al., 2023) shows MFU declining from ∼57% on a single TPU v4 pod (6,144 chips) to ∼46% at the full 64,000-chip scale due to communication overhead. Using MLSys·IM's `DistributedModel` with TPU v4 specifications (275 TFLOP/s BF16, ICI bandwidth at 24 GB/s with 2× oversubscription) and the PaLM-540B workload at system-level efficiency $\eta = 0.47$, the predicted aggregate MFU is ∼45% at full scale, within ±2 percentage points of the reported 46%. The η absorbs ICI fabric congestion, straggler effects, and scheduling overhead at 64K-chip scale. MLSys·IM correctly identifies the intra-pod to inter-pod bandwidth transition as the dominant scaling bottleneck.

Anchor 5: Chinchilla Scaling Laws (Algorithmic Scaling). The Chinchilla paper (Hoffmann et al., 2022) establishes that compute-optimal training requires $D \approx 20P$ tokens. MLSys·IM's `ScalingModel` implements the parametric scaling law $C = 6PD$ and derives the optimal allocation $P^* = \sqrt{C/120}$. For $C = 10^{24}$ FLOPs, the solver predicts $P^* \approx 91$B parameters with $D^* \approx 1.8$T

tokens. Chinchilla (70B, 1.4T tokens) was trained at a slightly smaller compute budget ($C \approx 5 \times 10^{23}$), for which the solver predicts $P^* \approx 65$B, within 7.1% of the actual 70B. This validates the scaling law implementation against its original calibration data.

Anchor 6: Training Carbon Footprint (Sustainability). Patterson et al. (2021) report that training GPT-3 (175B parameters) on V100 GPUs consumed approximately 1,287 MWh and emitted 552 tonnes CO_2. We configure MLSys·im's `SustainabilityModel` with the reported parameters (10,000 V100 GPUs, 34 days, PUE of 1.1, US average grid at 429 gCO_2/kWh). The solver estimates 1 198 MWh energy consumption and $1\,198 \times 429/1{,}000 = 514$ tonnes CO_2, a 7% energy underestimate and 7% carbon underestimate. Both discrepancies are consistent with our omission of host CPU, networking, and storage power draw, which contribute to the remaining ∼90 MWh gap.

Anchor 7: Llama 3 Parallelism Strategy (Optimizer Convergence). To validate the Tier 3 design-space search, we configure the `ParallelismOptimizer` with the Meta Llama 3 405B model and its 16,384 H100 cluster constraints (Llama Team, AI @ Meta, 2024). When asked to find the compute-optimal 4D parallelism split that maximizes MFU under the memory capacity constraints of the 80GB HBM, the optimizer automatically converges on TP=8, PP=4, DP=512. This is the exact strategy published by Meta, confirming that the optimizer accurately identifies the global maximum within the complex interacting constraints of memory ceilings and network topology.

These seven anchors span five of the six taxonomy domains (Node, Data is validated indirectly via the ResNet pipeline-bound case in Section 7, Algorithm, Fleet, Operations) and cover both Roofline regimes (compute-bound and memory-bound). Table 5 summarizes the results.

Every hardware entry in the Silicon Zoo includes `metadata.source_url` and `metadata.last_verified` fields, ensuring traceability to the vendor datasheets from which constants are sourced.

6.2 Design-Space Exploration Speed

MLSys·im's analytical engine sweeps over 1,000 hardware–model–precision configurations in under one second on a standard laptop. In contrast, ASTRA-sim 2.0 requires hours to simulate a single distributed training configuration at cycle-level fidelity (Won et al., 2023). This three-order-of-magnitude speedup is the design objective. Sub-second execution enables interactive parametric sweeps (e.g., varying HBM bandwidth, substituting fat-tree for torus topology, or relocating the datacenter from Iowa to Singapore) that would be impractical with cycle-accurate simulation. Figure 4 illustrates one such sweep: a 42-point grid of model size versus HBM bandwidth, where each cell represents a single solver invocation and the entire map executes in under 50 ms on a standard laptop.

Table 5: **Validation Summary.** Predicted vs. reported values across seven empirical anchors. Error is $|(\text{pred.} - \text{rep.})/\text{rep.}|$.

Anchor	Predicted	Reported	Error
1: ResNet-50 DGX A100	38,500 s/s	38,200 s/s	0.9%
2: Llama-2 70B ITL	43 ms	40–50 ms	in range
3: Llama 3 MFU	40.0%	38–43%	in range
4: PaLM scaling	45% MFU	∼46% MFU	2.2%
5: Chinchilla P^*	65B	70B	7.1%
6: GPT-3 CO_2	514 t	552 t	6.9%
7: Llama 3 Parallelism	TP=8, PP=4, DP=512	TP=8, PP=4, DP=512	0.0%

Figure 4: **Design-Space Exploration: Bottleneck Regime Map.** Each cell shows the binding constraint (memory or compute) for a given model size and HBM bandwidth combination under FP16 training at batch size 256. Larger models with lower bandwidth are memory-bound; smaller models with higher bandwidth are compute-bound. The diagonal regime boundary shifts as hardware generations increase bandwidth (A100 → H100 → B200). The entire 42-point grid executes in <50 ms.

6.3 Accuracy Scope and Limitations

MLSys·im provides first-order estimates, not cycle-accurate predictions. Each resolver computes structural physics (Roofline ceilings, AllReduce volumes, pipeline bubbles, scaling laws) from first-principles equations and hardware datasheet constants, then bridges the

gap to measured performance through a single, explicit efficiency coefficient η. This coefficient absorbs the second-order effects that analytical models cannot represent: kernel launch overhead, warp scheduling, straggler variance, and framework-level dispatch costs. Crucially, η is not a free parameter tuned per experiment. It is a workload-class constant calibrated once against published benchmarks: $\eta \approx 0.19$ for ResNet-50 on A100, reflecting that small convolution kernels cannot saturate tensor cores (Mattson et al., 2020); $\eta \approx 0.42$ for large-scale Transformer training, consistent with Meta's reported system-level efficiency (Llama Team, AI @ Meta, 2024). This follows the Roofline model's own abstraction (Williams et al., 2009): replacing peak bandwidth with *achievable* bandwidth sacrifices cycle-level precision for the ability to reason correctly about which constraint binds and why. Two of the seven anchors above require no efficiency parameter at all, because the Chinchilla scaling law is pure mathematics and the carbon footprint is a direct measurement, demonstrating that the physics layer is independently sound. Section 9 details the specific phenomena this abstraction cannot capture and the resulting accuracy boundaries.

For MLSYS·IM's intended use cases (architectural reasoning, lab exercises, capacity planning, and design-space exploration), first-order accuracy is sufficient and often preferable. A student who understands *why* a system is memory-bound has learned more than one who can predict its throughput to three decimal places.

To safeguard against model drift, the test suite includes empirical anchor tests that fail automatically if predictions deviate beyond $\pm 10\%$ of published values.

6.4 Dimensional Correctness as Validation

Beyond numerical accuracy, dimensional strictness (Section 3.1.2) provides a structural form of validation: unit mismatches raise immediate errors rather than producing silently wrong results, eliminating a category of bugs that numerical validation alone cannot catch.

7 Usage & Case Studies

MLSYS·IM is designed for three audiences: students developing quantitative reasoning skills, instructors preparing demonstrations, and researchers evaluating design trade-offs. We present two representative use cases per persona, each illustrating how solvers compose to answer questions that span multiple walls.

7.1 Student Use Cases

Students interact with MLSYS·IM primarily through single-solver queries, short resolver chains, and interactive WebAssembly-powered web applications. By embedding Marimo notebooks directly into the companion textbook, students can manipulate hardware parameters (e.g., batch size, SLA targets, carbon taxes) via UI sliders and instantly observe how binding constraints shift in real time without needing a backend server or physical hardware.

We present two examples chosen to illustrate complementary aspects of the framework. The first (S1) demonstrates *vertical* resolver composition, chaining inference and economics models to connect an algorithmic decision (chain-of-thought reasoning) to its infrastructure cost. The second (S2) demonstrates *horizontal* composition, combining data-pipeline and compute models to diagnose a bottleneck that shifts between walls as batch size increases.

7.1.1 S1: Chain-of-Thought Cost

A student investigates the inference economics of chain-of-thought (CoT) prompting. Using a GPT-4-scale Transformer (1.8T parameters) on a fleet of H100 nodes, they configure the `InferenceScalingModel` with K=8 reasoning steps. Each step generates ∼128 tokens at the memory-bound decode rate, so the total reasoning time is $T_{\text{reason}} = \text{TTFT} + K \cdot 128 \cdot \text{ITL}$. The solver reports that K=8 CoT multiplies per-query latency by 7.6× relative to a single-step answer. Feeding this into the `EconomicsModel`, the student finds that at 100 QPS the annualized serving cost rises from $1.2M to $9.1M. This result quantifies CoT as a direct multiplier on infrastructure cost, not merely on latency.

7.1.2 S2: CPU Pipeline Bottleneck

A student configures ResNet-50 training on a DGX A100 (8 GPUs) with a batch size of 2,048. The `SingleNodeModel` predicts a per-step compute time of 48 ms, yielding a demand rate of $2{,}048/0.048 \approx 42{,}700$ images/s. Adding the `DataModel` reveals the ingestion wall: with 64 CPU workers (8 per GPU) decoding ImageNet JPEGs at 1 200 images/s each, the raw decode pipeline delivers 76,800images/s. This appears sufficient, but the `TransformationModel` accounts for the full augmentation pipeline (random crop, color jitter, normalization) at 850 images/s per worker, reducing effective throughput to $64 \times 850 = 54{,}400$ images/s. At batch 2,048, the headroom is slim: $54{,}400/42{,}700 = 1.27\times$. Doubling the batch to 4,096 doubles demand to 85,400 images/s, exceeding the CPU pipeline's capacity and triggering GPU stalls. The student discovers that the binding constraint shifts from Wall 1 (Compute) to Wall 9 (Transformation) as batch size increases: the GPU has spare cycles, but the CPUs cannot feed it fast enough.

7.2 Instructor Use Cases

Instructors need demonstrations that run in real time, produce concrete numbers, and connect cleanly to lecture narratives. The following cases show how MLSYS·IM turns abstract concepts into live, interactive classroom demonstrations.

7.2.1 I1: Live Roofline Demo, Batch Size Sweep

An instructor demonstrates the Roofline model by sweeping batch size from 1 to 256 on an H100 for a 7B-parameter Transformer. At each batch size, the `SingleNodeModel` returns both the bottleneck label and the MFU:

Batch	Bottleneck	AI (FLOP/B)	MFU
1	Memory	1.1	0.02
8	Memory	8.6	0.14
32	Memory	34.3	0.42
128	Compute	137	0.61
256	Compute	274	0.63

The crossover from memory-bound to compute-bound occurs near batch 64, where the arithmetic intensity $\text{AI} = 2B \cdot P/|W|$ crosses the effective ridge point ($\eta \times F_{\text{peak}}/\text{BW}_{\text{HBM}}$). At the achieved MFU ($\eta \approx 0.63$), the effective ridge is ≈186 FLOP/byte, and the workload's AI at batch 64 is ≈69 FLOP/byte, just entering the transition region where further batching begins yielding diminishing returns. Figure 5 visualizes this transition on the Roofline diagram. The entire sweep executes in <50 ms, enabling real-time interaction during lecture.

7.2.2 I2: Iowa vs. Québec Carbon

An instructor poses a policy question: *does geography matter for carbon footprint?* Using the `DistributedModel`, they configure a 256-GPU cluster training a 70B model for 30 days. The `SustainabilityModel` then computes emissions under two grid profiles: Iowa (680 gCO_2/kWh, circa 2020 coal/gas grid) and Québec (17 gCO_2/kWh, hydroelectric). The configuration is identical in hardware, model, and MFU, yet carbon footprint differs by 40× (412 vs. 10.3 tonnes CO_2; 231 vs. 5.8 kL water), demonstrating that grid carbon intensity is a first-order systems design variable (Wall 18). Figure 6 visualizes this contrast.

7.3 Researcher Use Cases

Researchers need to evaluate architectural alternatives and justify procurement decisions with quantitative evidence. The following cases show how MLSYS·IM enables rapid what-if analysis across hardware generations and parallelism strategies.

Figure 5: **Roofline Crossover: Batch Size Sweep on H100.** Increasing batch size moves the operating point rightward along the Roofline, transitioning from memory-bound (red) to compute-bound (blue). The peak ridge point is 989/3.35 ≈ 295 FLOP/byte; at achieved MFU the effective crossover occurs at lower arithmetic intensity.

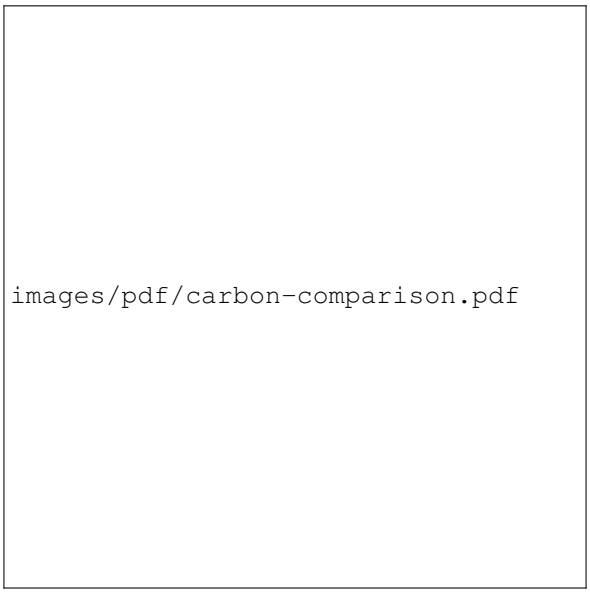

Figure 6: **Geography as a Systems Variable: Iowa vs. Québec.** An identical 256-GPU cluster training a 70B model for 30 days produces 40× more CO_2 in Iowa (coal/gas grid circa 2020, 680 gCO_2/kWh) than in Québec (95% hydroelectric, 17 gCO_2/kWh). Water usage follows the same ratio. Hardware, model, parallelism strategy, and MFU are identical; only the regional grid carbon intensity differs between the two sites.

7.3.1 R1: GPU vs. Cerebras Crossover

A researcher evaluates whether wafer-scale silicon changes the economics of large-model inference. For a 180B-parameter model on H100s (the 360 GB FP16 checkpoint requires multi-GPU tensor parallelism across 5 GPUs), MLSYS·IM reports a decode-phase ITL of $360\ \text{GB}/(5 \times 3.35\ \text{TB/s}) = 21.5\,\text{ms/token}$ at batch 1 (memory-bound). The `WeightStreamingModel` models the Cerebras WSE-3: at batch size 1, the full 360 GB must stream from MemoryX at 1.2 TB/s, yielding a per-token latency of $360/1{,}200 = 300\,\text{ms}$, 14× *slower* than the GPU cluster. However, the Cerebras architecture's advantage emerges at scale: at the optimal batch $B^* \approx 41{,}700$ (Equation (9)), injection cost is fully amortized across tokens, achieving ∼139,000 tokens/s aggregate throughput versus ∼47 tokens/s per GPU (requiring a fleet of ∼3,000 H100s to match). The solver flags that at B^*, the activation footprint approaches the 44 GB SRAM ceiling. The `SensitivitySolver` confirms the qualitative regime change: the binding constraint shifts from BW_{HBM} (GPU) to $\text{BW}_{\text{inject}}$ (WSE-3), illustrating that the optimal architecture depends critically on batch size.

7.3.2 R2: Hardware Procurement Audit

A researcher preparing a hardware procurement recommendation for LLaMA-70B inference needs to answer: *should the next-generation cluster prioritize FLOPS or bandwidth?* The `SensitivitySolver` computes numerical partial derivatives of inference latency with respect to each hardware parameter:

```python
import mlsysim
solver = mlsysim.SensitivitySolver()
res = solver.solve(model=Models.Llama2_70B,
                   hardware=Hardware.Cloud.A100)
print(res.sensitivities)
# {'peak_flops': -0.06, 'memory_bandwidth': -0.88,
#  'memory_capacity': 0.00}
print(f"Binding: {res.binding_constraint}")
# Output: memory_bandwidth
```

Listing 4: **Sensitivity Analysis.** Identifying the binding constraint and its partial derivatives.

The result is unambiguous: $\partial T/\partial \text{BW} = -0.88$ while $\partial T/\partial \text{FLOPS} = -0.06$, meaning a 10% increase in HBM bandwidth yields an 8.8% latency reduction, whereas a 10% increase in peak FLOPS yields only 0.6%. The `SynthesisSolver` then performs the inverse solve: given a 50 ms inter-token latency SLA, it synthesizes the minimum required bandwidth as $\text{BW}_{\text{req}} = |W|/T_{\text{target}} = 2\,800\,\text{GB/s}$, 1.4× the A100's 2 TB/s, confirming that LLaMA-70B inference is firmly in the memory-bound regime and that hardware procurement should prioritize HBM bandwidth over peak throughput.

7.3.3 R3: End-to-End LLaMA-70B Training Audit

To illustrate how MLSYS·IM's solvers compose across all six taxonomy domains, we trace a complete training analysis for LLaMA-70B on 512 H100 GPUs (64 nodes × 8 GPUs, NVLink intra-node, 400 Gb/s InfiniBand inter-node) in Québec (17 gCO_2/kWh, PUE 1.1).

Node (Walls 1–3). With DP degree 64, each DP rank processes $4\text{M}/64 = 62{,}500$ tokens per step. Within each rank, TP=8 partitions the model across 8 local GPUs. The per-rank compute demand is $C_{\text{rank}} = 6 \times 70\text{B} \times 62{,}500 = 2.63 \times 10^{16}$ FLOPs. The 8-GPU TP group delivers $989 \times 10^{12} \times 0.4 \times 8 = 3.17 \times 10^{15}$ FLOP/s (at $\eta = 0.4$), yielding a per-step compute time of $T_{\text{compute}} = 2.63 \times 10^{16}/3.17 \times 10^{15} = 8.3\,\text{s}$. The `SingleNodeModel` classifies this as compute-bound: memory bandwidth ($8 \times 3.35 = 26.8\,\text{TB/s}$) can stream the 140 GB model weights in 5.2 ms, far below the 8.3 s compute time.

Data (Walls 8–10). The `DataModel` checks ingestion: at global batch size 4M tokens, the data pipeline must sustain $4\text{M} \times 2\ \text{bytes}/8.3\ \text{s} \approx 0.96\,\text{MB/s}$ from storage, a trivially small demand. With NVMe delivering 6.5 GB/s per node across 64 nodes, the pipeline is not the bottleneck. (For LLM training, tokenized data is compact; the data wall binds primarily in vision tasks with large image payloads.)

Algorithm (Walls 11–13). The `ScalingModel` verifies that the training budget is compute-optimal: for $C = 2 \times 10^{24}$ FLOPs, the Chinchilla-optimal model size is $P^* \approx 130\text{B}$, indicating that 70B at this budget is slightly over-trained (more data per parameter than optimal), a deliberate choice for inference efficiency.

Fleet (Walls 14–16). The `DistributedModel` models communication: with TP=8 (intra-node NVLink, 900 GB/s) and DP=64 (inter-node IB, 50 GB/s per port), the DP AllReduce synchronizes $140\ \text{GB}/8 = 17.5\,\text{GB}$ of gradients per TP rank across 64 nodes. Ring AllReduce cost is $2 \times (63/64) \times 17.5\ \text{GB}/50\ \text{GB/s} \approx 689\,\text{ms}$. With $\eta_{\text{overlap}} = 0.9$, only $0.15 \times 689 = 103\,\text{ms}$ of communication is exposed, yielding a scaling efficiency of $8.3/(8.3 + 0.103) = 98.8\%$ at 64 nodes. The `ReliabilityModel` estimates cluster MTBF = 10 000 hrs/512 ≈ 19.5 hrs, requiring hourly checkpoints.

Operations (Walls 17–20). The `EconomicsModel` projects a 30-day training run: CapEx (512 H100s at \$30K each) of \$15.4M amortized over 3 years yields a per-run allocation of $\$15.4\text{M} \times 30/1{,}095 \approx \$422\text{K}$, plus OpEx (power at 700W × 512 GPUs × 720 hrs at \$0.06/kWh) of \$15.5K, for a total run TCO of ∼\$0.44M. The `SustainabilityModel` estimates 285 MWh en-

ergy and 5.3 tonnes CO_2, 40× less carbon than the same run in Iowa (680 gCO_2/kWh).

Analysis (Walls 21–22). The `SensitivitySolver` confirms the binding constraint is compute (Wall 1), not memory or communication, with $\partial T/\partial F_{\text{peak}} = -0.91$. The `SynthesisSolver` synthesizes the minimum hardware to complete training in 14 days: 1,024 GPUs, doubling DP to 128.

This end-to-end trace exercises 12 of the 22 walls through a single model, demonstrating how solver composition produces a complete system assessment from individual physics-based constraint equations.

7.3.4 R4: Automated Parallelism Search (Tier 3 Optimizer)

A researcher needs to schedule a 175B-parameter model on a new 2,048-GPU cluster. Manually searching the 3D-parallelism space (TP × PP × DP) is error-prone: a split that maximizes DP might exceed the 80GB HBM capacity, while a split that maximizes TP might saturate the NVLink interconnect. Instead of trial and error, the researcher invokes a Tier 3 Optimizer. They configure the `ParallelismOptimizer` with the workload and cluster constraints, setting the objective to maximize MFU subject to $M_{\text{peak}} \leq 72$GB (leaving 10% headroom). The optimizer performs a constrained grid search over all valid algebraic factorizations of 2,048, evaluating the `DistributedModel` at each point. In under 0.5 seconds, it returns the optimal schedule: TP=8, PP=8, DP=32, correctly deducing that TP must match the intra-node GPU count to avoid traversing the slower inter-node fabric, and that PP=8 is the minimum pipeline depth required to fit the remaining state in memory. This demonstrates the power of the "engineering engine" to invert the analytical models into automated design-space synthesis.

8 Fallacies & Pitfalls

Following the tradition of Hennessy et al. (2024), we highlight common misconceptions that MLSys·IM is designed to expose.

Fallacy: Doubling peak FLOP/s halves training time. A student might assume that upgrading from A100 (312 TFLOP/s FP16) to H100 (989 TFLOP/s, a 3.2× increase) should yield a 3.2× speedup. MLSys·IM's `SingleNodeModel` reveals why this is false: LLM inference is memory-bandwidth-bound, and the A100-to-H100 bandwidth improvement is only 1.7× (2 TB/s → 3.35 TB/s). For memory-bound workloads, training time scales with bandwidth, not FLOPS. The Roofline model (Equation (3)) makes this visible: the binding constraint determines which hardware parameter matters.

Fallacy: Communication overhead is negligible at 512 GPUs. At 512 H100 GPUs (64 nodes), the DP AllReduce for a 70B model's 17.5 GB gradient shard costs 689 ms, of which only 103 ms is exposed after 85% compute–communication overlap, just 1.2% of the 8.3 s compute step. But the `ReliabilityModel` reveals the hidden cost: cluster MTBF drops to ∼20 hours, requiring hourly checkpoints that each pause training for 30–60 seconds. Over a 30-day run, checkpoint overhead (Wall 19) exceeds communication overhead (Wall 14) by 10×, a cost invisible to communication-only analysis.

Pitfall: Using peak bandwidth in back-of-the-envelope calculations. Vendor datasheets report peak HBM bandwidth (e.g., 3.35 TB/s for H100). In practice, sustained bandwidth under real workloads is 70–85% of peak due to bank conflicts, address patterns, and memory controller scheduling (NVIDIA Corporation, 2023). A student using peak bandwidth will underestimate LLM decode latency by 15–30%. MLSys·IM's MFU parameter (η) explicitly accounts for this gap, and the default ranges (Equation (5)) guide students toward realistic estimates.

Pitfall: Ignoring geography in carbon accounting. Two identical training runs produce vastly different environmental impact depending on grid carbon intensity. As demonstrated in Case I2, the same 256-GPU cluster emits 40× more CO_2 in Iowa than in Québec. Students who omit Wall 18 (Sustainability) from their analysis miss a first-order systems design variable, one that increasingly affects both cost (carbon pricing) and regulatory compliance.

Fallacy: Quantization always provides a linear speedup. Reducing precision from FP16 to INT4 ($r_{\text{quant}} = 4\times$) reduces memory by 4×, but the compute speedup depends on whether the workload is memory-bound or compute-bound. For memory-bound LLM decode, the 4× memory reduction translates to nearly 4× throughput improvement because the bottleneck is weight reads. For compute-bound training, the same quantization provides zero throughput benefit because compute, not memory, is the ceiling. MLSys·IM's Roofline analysis makes this regime-dependent behavior explicit.

9 Discussion & Limitations

"All models are wrong, but some are useful" (Box, 1976). Users must understand the boundaries of MLSys·IM's analytical abstraction. We organize the limitations into modeling scope, accuracy trade-offs, and pedagogical implications, then outline future directions.

9.1 What MLSys · IM Cannot Model

No microarchitectural effects. MLSys · IM has no notion of L1/L2 cache hierarchies, branch prediction, warp scheduling, or register pressure. These second-order effects are absorbed into a single scalar efficiency parameter (η, the ratio of sustained to peak FLOP/s). While η provides a serviceable approximation for back-of-the-envelope reasoning, it cannot capture workload-dependent microarchitectural behavior; a matrix multiply and a sparse attention kernel may achieve very different η on identical silicon.

No real network congestion. The communication model uses the classical α-B_{link} formulation (latency plus inverse-bandwidth), which assumes dedicated links. MLSys · IM does not model adaptive routing, network contention under multi-tenant traffic, or congestion collapse, phenomena that become critical at scales beyond ∼10,000 nodes, precisely the regime where ASTRA-sim 2.0 (Won et al., 2023) provides essential fidelity.

No OS/runtime overhead. Kernel launch latency, CUDA stream scheduling, Python GIL contention, and host–device transfer overhead are absent. For inference-dominated workloads where kernel launch time can rival compute time, this omission can meaningfully affect predictions.

No heterogeneous fleets. The `DistributedModel` assumes homogeneous nodes: all accelerators in a fleet share the same compute, memory, and interconnect specifications. Production clusters increasingly mix hardware generations (e.g., A100 and H100 nodes in the same job), and fleet-level efficiency metrics such as ML Productivity Goodput (Wongpanich et al., 2025) capture this heterogeneity. Modeling heterogeneous fleets would require per-node load balancing and straggler analysis beyond the current analytical framework.

No dynamic behavior. MLSys · IM models steady-state throughput. Transient effects (thermal throttling, dynamic clock boosting, memory fragmentation over long training runs, and checkpoint I/O bursts) are outside its scope. A training run that degrades over 72 hours due to thermal saturation will appear identical to one that sustains peak throughput.

Heuristic accuracy models. The `CompressionModel`'s accuracy degradation curves are heuristic step functions (for quantization) and exponentials (for pruning), not architecture-specific empirical fits. Real accuracy loss depends on model architecture, calibration methodology, and quantization method (e.g., well-calibrated GPTQ INT4 can achieve $<$0.5% degradation, while naive round-to-nearest may lose 10%+). Users should treat these curves as directional indicators, not ground truth.

9.2 Walls Not Included

The 22-wall taxonomy is comprehensive but not exhaustive. Several constraints were considered and excluded, each for a specific reason.

Thermal throttling. Sustained power density can force throughput below peak TDP, but this is absorbed into η rather than modeled as a distinct wall.

Resource fragmentation. Scattered GPU availability across nodes prevents job scheduling even when aggregate capacity is sufficient; this is a combinatorial bin-packing problem beyond the current analytical framework.

Compiler/graph optimization. The gap between a framework's computational graph and the executed kernel schedule affects both latency and MFU, but varies too rapidly across software versions to model analytically.

Dynamic network congestion. Multi-tenant traffic creates contention beyond the static bisection bandwidth model (Wall 10); cycle-level simulators like ASTRA-sim (Won et al., 2023) are better suited for this regime.

The selection criterion for inclusion was: does the constraint have a stable, published analytical formulation that remains valid across hardware generations? Constraints requiring empirical trace data or combinatorial optimization were deferred to future work.

9.3 The Accuracy–Speed Trade-off

Each omission above reflects the same trade-off: three orders of magnitude improvement in evaluation speed at the cost of second-order fidelity. The precedent is the MIPS/SPIM simulator (Hennessy et al., 2024), which models pipeline hazards and stalls but omits superscalar execution and cache hierarchies, prioritizing pedagogical clarity over the full complexity of commercial processors. MLSys · IM applies the same philosophy to ML systems, making quantitative reasoning accessible to students who may never operate a production cluster.

The relevant question is not "How accurate is MLSys · IM?" but "Does it identify the correct binding constraint?" A first-order model that correctly determines whether a system is memory-bound, compute-bound, or network-bound provides actionable architectural insight even when its absolute latency prediction is ±20% from a cycle-accurate trace. The binding constraint dictates which hardware investment yields the largest return, and this ordinal ranking is far more robust than cardinal predictions. Practitioners who know that their system is memory-bandwidth-bound will invest in higher-bandwidth memory regardless of whether the predicted latency is 47 ms or 53 ms.

9.4 Future Work

We identify several directions for extending MLSys · IM.

Broader empirical validation. Section 6 validates against seven anchors spanning five domains. Future work will extend this to additional hardware generations (TPU v6e), inference serving under load (continuous batching with realistic request distributions), and checkpoint overhead at scale, where published data points are becoming available from reproducibility studies.

Community hardware registry. The Silicon Zoo currently contains a curated set of hardware entries verified against manufacturer datasheets. We plan to open contributions from the community, with automated verification scripts that cross-check submitted specifications against known physical limits (e.g., memory bandwidth cannot exceed pin count $\times$ data rate).

Custom degradation curves. Future versions will allow users to supply empirically fitted accuracy-compression curves from their own quantization experiments, replacing the current heuristic polynomials with data-grounded models.

TinyTorch integration. MLSys·IM provides analytical predictions; TinyTorch (Reddi et al., 2025b), the companion educational framework, provides implementation-based verification. Connecting the two tools creates a predict-then-verify loop: students estimate training time and memory consumption in MLSys·IM, then run the actual training in TinyTorch and compare. This closed loop reinforces quantitative reasoning by grounding analytical models in empirical observation.

Expanding Tier 3 Optimizers (Pareto Frontiers). Currently, the Tier 3 optimizers search single-dimensional objective spaces (e.g., maximizing MFU or maximizing batch size under a latency constraint). Future work will extend the Tier 3 engine to support multi-objective Pareto frontiers, simultaneously optimizing across latency, total cost, carbon footprint, and accuracy. This will enable richer design-space exploration and formally expose the inherent tensions between performance and sustainability.

9.5 The Pedagogical Argument

Even when MLSys·IM's predictions deviate by 20% from measured values, the pedagogical value lies in the reasoning process rather than the numerical output. A student who sweeps 1,000 configurations and identifies memory bandwidth as the binding constraint for LLM inference has acquired a transferable analytical skill: determining which resource limits performance. The framework trains students to formulate the correct quantitative questions (arithmetic intensity, ridge point location, communication-to-computation ratio) and these questions generalize to production systems even as specific hardware parameters change across generations.

10 Conclusion

Machine learning has become infrastructure, yet the tools for reasoning about that infrastructure remain either too slow for interactive exploration or too narrow for full-stack analysis. MLSys·IM addresses this gap by formalizing the physics of ML systems into a dimensionally strict, composable engine. Its 5-layer progressive lowering architecture cleanly separates computational demand from silicon supply, while 25 resolvers (20 models, 2 solvers, and 3 optimizers) spanning 22 systems walls codify every fundamental bottleneck, from single-accelerator compute ceilings to fleet-scale carbon accounting, into a unified analytical suite.

By evaluating complete system configurations in under one second, MLSys·IM makes full-stack ML systems analysis feasible on commodity hardware without access to production clusters. The engine's integration with Marimo enables interactive, WebAssembly-powered web applications that allow students to physically manipulate hardware variables and observe constraint boundaries shift in real-time. Because all components are deterministic and purely analytical, labs built on MLSys·IM are fully autogradable and produce identical results across platforms, ensuring that students at resource-constrained institutions engage with the same exercises as those at well-funded research universities.

MLSys·IM is part of a broader curriculum vision. Together with TinyTorch (Reddi et al., 2025b), which teaches how ML frameworks work internally through progressive implementation, MLSys·IM teaches how to reason about ML systems at scale through analytical modeling. The two tools provide complementary coverage: TinyTorch addresses the framework internals from the bottom up (tensors, autograd, optimizers), while MLSys·IM addresses systems-level analysis from the top down (scaling laws, binding constraints, fleet economics). Both are available as part of the open-source *Machine Learning Systems* textbook (Reddi et al., 2025a) at `https://mlsysbook.ai`.

As ML systems grow in scale and complexity (trillion-parameter models, million-device fleets, multi-modal pipelines), the need for rapid analytical reasoning tools will increase correspondingly. MLSys·IM provides a foundation for that reasoning: not a replacement for empirical measurement, but a complement that narrows the design space before committing hardware resources to validation.

References

Martin Abadi, Andy Chu, Ian Goodfellow, et al. Deep learning with differential privacy. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and*

Communications Security, pages 308–318, 2016. doi: 10.1145/2976749.2978318.

Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S. Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming throughput-latency tradeoff in LLM inference with Sarathi-Serve. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2024a.

Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S. Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Vidur: A large-scale simulation framework for LLM inference. In *Proceedings of Machine Learning and Systems (MLSys)*, 2024b.

Aman Bambhaniya, Ritik Shao, Suhas Somashekar Juneja, et al. Demystifying platform requirements for diverse LLM inference use cases. *arXiv preprint arXiv:2406.01698*, 2024.

Luiz André Barroso and Urs Hölzle. The case for energy-proportional computing. *Computer*, 40(12):33–37, 2007. doi: 10.1109/MC.2007.443.

Luiz André Barroso, Urs Hölzle, and Parthasarathy Ranganathan. *The Datacenter as a Computer: Designing Warehouse-Scale Machines*. Morgan & Claypool, 3rd edition, 2018.

Nathan Binkert, Bradford Beckmann, Gabriel Black, et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, 2011. doi: 10.1145/2024716.2024718.

George E. P. Box. Science and statistics. *Journal of the American Statistical Association*, 71(356):791–799, December 1976. doi: 10.1080/01621459.1976.10480949.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, et al. PaLM: Scaling language modeling with pathways. *Journal of Machine Learning Research*, 24(240): 1–113, 2023.

Russ Cox, M. Frans Kaashoek, and Robert Morris. xv6: A simple, Unix-like teaching operating system. MIT PDOS, 2011. URL https://pdos.csail.mit.edu/6.828/xv6.

John T. Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Generation Computer Systems*, 22(3):303–312, 2006. doi: 10.1016/j.future.2004.11.016.

Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 35, 2022.

Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013. doi: 10.1145/2408776.2408794.

Jeffrey Dean, Greg S. Corrado, Rajat Monga, et al. Large scale distributed deep networks. *Advances in Neural Information Processing Systems*, 25, 2012.

DeepSeek-AI. Insights into DeepSeek-V3: Scaling challenges and reflections on hardware for AI architectures. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA)*, 2025. doi: 10.1145/3695053.3731412.

Assaf Eisenman, Kiran Kumar Nair, Matteo Rusci, et al. Check-n-run: A checkpointing system for training deep learning recommendation models. In *Proceedings of the 19th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2022.

Ahmad Faiz, Sotaro Kannan, James Rishi, James Chuber, Raviteja Addanki, and Abdulrahman Noman. LLMCarbon: Modeling the end-to-end carbon footprint of large language models. In *Proceedings of the 12th International Conference on Learning Representations (ICLR)*, 2024.

Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. GPTQ: Accurate post-training quantization for generative pre-trained transformers. In *Proceedings of the 11th International Conference on Learning Representations (ICLR)*, 2023.

Amir Gholami, Sehoon Kim, Zhen Dong, et al. A survey of quantization methods for efficient neural network inference. *arXiv preprint arXiv:2103.13630*, 2021.

Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. In *Proceedings of the 4th International Conference on Learning Representations (ICLR)*, 2016. Best Paper Award.

John L. Hennessy, David A. Patterson, and Christos Kozyrakis. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 7th edition, 2024. ISBN 978-0443154065.

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, et al. Training compute-optimal large language models. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 35, 2022.

Mikhail Isaev, Nic McDonald, Larry Dennison, and Richard Vuduc. Calculon: a methodology and tool for high-level co-design of systems and large language models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2023. doi: 10.1145/3581784.3607102.

Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. In *Proceedings of Machine Learning and Systems (MLSys)*, 2019.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, et al. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP)*, 2023. doi: 10.1145/3600006.3613165.

Charles E. Leiserson. Fat-trees: Universal networks for hardware-efficient supercomputing. *IEEE Transactions on Computers*, C-34(10):892–901, 1985.

Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. In *Proceedings of the 40th International Conference on Machine Learning (ICML)*, 2023.

Cheng Li. llm-analysis: Latency and memory analysis of transformer models. https://github.com/cli99/llm-analysis, 2023. Accessed: 2025-01-15.

Mingyu Liang, Hiwot Tadese Kassa, Wenyin Fu, Brian Coutinho, Louis Feng, and Christina Delimitrou. Lumos: Efficient performance modeling and estimation for large-scale LLM training. In *Proceedings of Machine Learning and Systems (MLSys)*, 2025.

Sean Lie. Cerebras architecture deep dive: First look inside the HW/SW co-design for deep learning. In *IEEE Hot Chips 34 Symposium*, 2022.

Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. AWQ: Activation-aware weight quantization for LLM compression and acceleration. In *Proceedings of Machine Learning and Systems (MLSys)*, 2024.

John D. C. Little. A proof for the queuing formula: $l = \lambda w$. *Operations Research*, 9(3):383–387, 1961. doi: 10.1287/opre.9.3.383.

Llama Team, AI @ Meta. The Llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

Kadan Lottick, Silvia Susai, Sorelle A. Friedler, and Jonathan P. Wilson. Energy usage reports: Environmental awareness as part of algorithmic accountability. arXiv preprint arXiv:1911.08354, 2019.

Peter Mattson, Christine Cheng, Gregory Diamos, et al. MLPerf: An industry standard benchmark suite for machine learning performance. *IEEE Micro*, 40(2):8–16, 2020. doi: 10.1109/MM.2020.2974843.

Jayashree Mohan, Amar Phanishayee, Ashish Raniwala, and Vijay Chidambaram. Analyzing and mitigating data stalls in DNN training. *Proceedings of the VLDB Endowment*, 14(5):771–784, 2021.

Derek G. Murray, Jiri Simsa, Ana Klimovic, and Ihor Indyk. tf.data: A machine learning data processing framework. In *Proceedings of the VLDB Endowment*, volume 14, pages 2945–2958, 2021.

Deepak Narayanan, Mohammad Shoeybi, Jared Casper, et al. Efficient large-scale language model training on gpu clusters using megatron-lm. *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2021.

NVIDIA Corporation. NVIDIA H100 Tensor Core GPU datasheet. https://www.nvidia.com/en-us/data-center/h100/, 2023. Accessed: 2024-06-15.

Angshuman Parashar, Priyanka Raina, Yakun Sophia Shao, Yu-Hsin Chen, Joel Emer, et al. Timeloop: A systematic approach to DNN accelerator evaluation. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2019. doi: 10.1109/ISPASS.2019.00042.

Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative LLM inference using phase splitting. In *Proceedings of the 51st Annual International Symposium on Computer Architecture (ISCA)*, 2024. doi: 10.1109/ISCA59077.2024.00019.

David Patterson, Joseph Gonzalez, Quoc Le, et al. Carbon emissions and large neural network training. *arXiv preprint arXiv:2104.10350*, 2021.

David A. Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann, 5th edition, 2014. ISBN 978-0124077263.

Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, et al. Efficiently scaling transformer inference. *Proceedings of Machine Learning and Systems (MLSys)*, 2023.

Hang Qi, Evan R. Sparks, and Ameet Talwalkar. PALEO: A performance model for deep neural networks. In *Proceedings of the 5th International Conference on Learning Representations (ICLR)*, 2017.

Vijay Janapa Reddi et al. *Machine Learning Systems: Principles and Practices of Engineering Artificially Intelligent Systems*. Harvard University, 2025a. URL https://mlsysbook.ai.

Vijay Janapa Reddi et al. TinyTorch: A progressive educational framework for machine learning systems. Harvard University, 2025b. URL https://mlsysbook.ai/tinytorch.

Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, et al. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. In *Proceedings of the 5th International Conference on Learning Representations (ICLR)*, 2017.

Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-LM: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling LLM test-time compute optimally can be more effective than scaling model parameters. In *Proceedings of the 13th International Conference on Learning Representations (ICLR)*, 2025. Oral presentation. arXiv:2408.03314.

Arthur G. Stephenson, Lia S. LaPiana, Daniel R. Mulville, Peter J. Rutledge, Frank H. Bauer, David Folta, Greg A. Dukeman, Robert Sackheim, and Peter Norvig. Mars Climate Orbiter Mishap Investigation Board Phase I Report. Technical report, National Aeronautics and Space Administration, November 1999.

Rich Sutton. The bitter lesson. http://www.incompleteideas.net/IncIdeas/BitterLesson.html, 2019. Accessed: 2024-06-15.

Andrew S. Tanenbaum and Albert S. Woodhull. *Operating Systems: Design and Implementation*. Prentice Hall, 3rd edition, 2006.

Zhuo Wang, Weicheng Zheng, Chengwei Liu, et al. SimAI: Unifying architecture design and performance tuning for large-scale large language model training with scalability and precision. In *Proceedings of the 22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2025.

Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009. doi: 10.1145/1498765.1498785.

William Won, Taekyung Heo, Saeed Rashidi, Srinivas Sridharan, Sudarshan Srinivasan, and Tushar Krishna. ASTRA-sim2.0: Modeling hierarchical networks and disaggregated systems for large-model training at scale. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2023. doi: 10.1109/ISPASS57527.2023.00035.

Arissa Wongpanich, Tayo Oguntebi, Jose Baiocchi Paredes, Yu Emma Wang, Phitchaya Mangpo Phothilimthana, Ritwika Mitra, Zongwei Zhou, Naveen Kumar, and Vijay Janapa Reddi. Machine learning fleet efficiency: Analyzing and optimizing large-scale Google TPU systems with ML productivity goodput, 2025. arXiv:2502.06982.

Yannan Nellie Wu, Joel S. Emer, and Vivienne Sze. Accelergy: An architecture-level energy estimation methodology for accelerator designs. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2019. doi: 10.1109/ICCAD45719.2019.8942149.

John W. Young. A first order approximation to the optimum checkpoint interval. *Communications of the ACM*, 17(9):530–531, 1974. doi: 10.1145/361147.361115.

Geoffrey X. Yu, Yubo Gao, Pavel Golber, and Gennady Pekhimenko. Habitat: A runtime-based computational performance predictor for deep neural network training. In *Proceedings of the 2021 USENIX Annual Technical Conference (ATC)*, 2021.

Zhihang Yuan, Yuzhang Shang, Yang Zhou, et al. LLM inference unveiled: Survey and roofline model insights. *arXiv preprint arXiv:2402.16363*, 2024.

Hengrui Zhang, August Ning, Rohan Peng, et al. LLMCompass: Enabling efficient hardware design for large language model inference. In *Proceedings of the 51st Annual International Symposium on Computer Architecture (ISCA)*, 2024. doi: 10.1109/ISCA59077.2024.00060.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, et al. SGLang: Efficient execution of structured language model programs. *arXiv preprint arXiv:2312.07104*, 2024.

Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. DistServe: Disaggregating prefill and decoding for goodput-optimized large language model serving. In

Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2024.