Inference Engines: how generative AI is changing computers .. again!

Understand how the operating system is failing us, and what the next generation of compute looks like

Abdelrahman Hosny, PhD
Sr. Silicon Alliances Manager
Canonical
abdelrahman.hosny@canonical.com

Contents

Contents 2
Executive summary 4
Introduction: the new compute paradigm 5
The rise of the inference economy 5
Defining the stack 6
Application layer 6
Frameworks and runtimes 6
The operating system 7
The silicon 7
The missing layer: model-aware orchestration 7
The memory wall 8
How AI inference works: a high-resolution view 8
Predictive vs. generative models 8
Tokens: the atomic unit of generation 10
The two-phase execution 10
The prefill phase 11
The decode phase 12
The math of generation 13
The OS bottleneck: why legacy kernels fail AI 15
The "logic + data" conflict 15
VRAM fragmentation 15
The global interpreter lock (GIL) and host overhead 16
Scheduling mismatch 17
The rise of inference engines 18
Breaking the research-production gap 18
Architecture: layers and components 18
Core techniques in inference engines 20
Quantization and Mixed Precision 20
PagedAttention 20
Continuous batching 20
Speculative Decoding 20
Tensor and Pipeline Parallelism 21
Token Streaming and Early Exit 21
Toward inference-native infrastructure 21
Inference as a first-class citizen 21
Inference snaps 22
The inference system call 23
Business value of the optimized stack 25
Unit economics 25

Latency arbitrage 25
The MLOps lifecycle 25
Own your AI infrastructure 26
Data center AI deployment: Canonical private AI infrastructure 26
Edge and IoT: Inference snaps 26
Conclusion 27
References 28

Executive summary

Generative AI is driving a structural shift in computing by exposing a deep mismatch between modern model workloads and the legacy systems that run them. The prevailing software and hardware stacks were designed for deterministic tasks: file operations, transactional databases, and other short-lived processes; not for probabilistic, memory-dense, and continuously executing inference loops. As large language and multimodal models transition from experimental tools to production infrastructure, this mismatch turns what was once a secondary optimization concern into a primary constraint on performance, reliability, and cost. The consequence is that the operating system – not the model or even the accelerator – increasingly becomes the bottleneck.

Inference workloads behave differently from traditional applications. They maintain a persistent state, move massive tensors across memory boundaries, and depend on fine-grained scheduling decisions that legacy kernels were never built to understand. Treating inference as just another process leads to VRAM fragmentation, inefficient batching, unnecessary host-device synchronization, and poor utilization of specialized hardware. What appears as “model slowness” is often systemic friction embedded in system calls that predate machine learning entirely.

Inference engines emerge as a corrective layer that narrows the gap between research performance and production reality. By introducing model-aware scheduling, paged attention, and continuous batching, these engines align execution with the statistical and iterative nature of generative computation. Accelerators become shared, stateful resources instead of isolated peripherals, allowing higher throughput, lower latency, and more predictable scaling under real-world demand. This transition is not purely technical; it reshapes economics. Optimized inference stacks alter unit cost curves, enable latency arbitrage across geographies and device classes, and compress the MLOps lifecycle by reducing divergence between experimentation and deployment environments. The distinction between data center, edge, and local compute begins to blur as each tier converges on the same requirement: infrastructure designed for inference as a native operation rather than a tolerated workload.

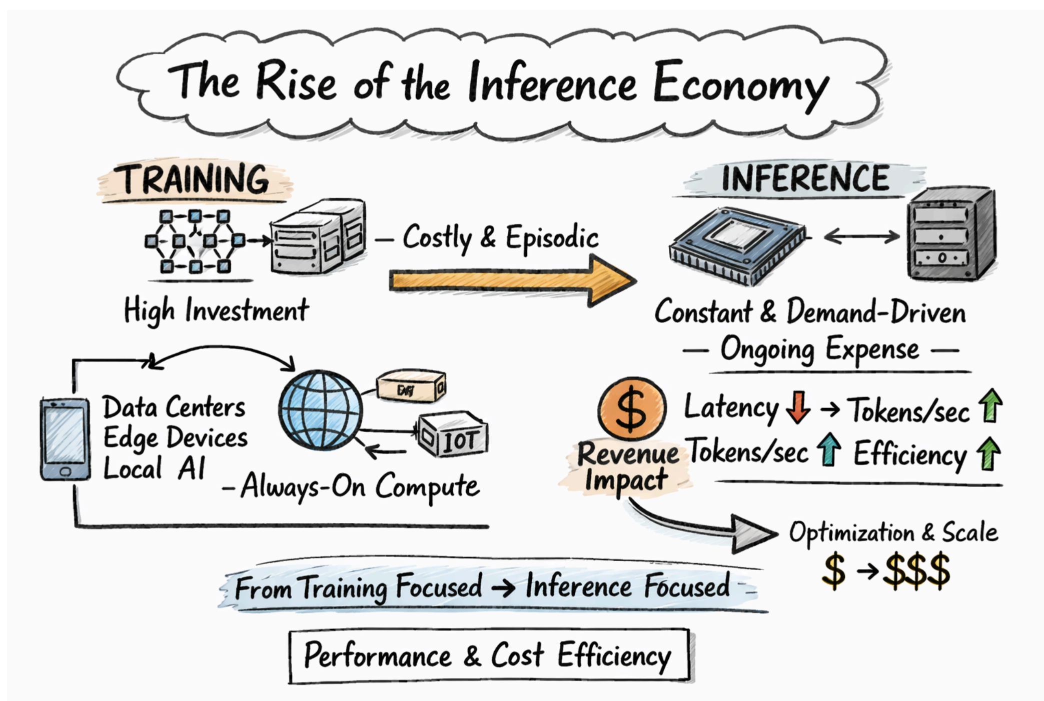
This paper argues that the next generation of computing will be defined less by number of tensor cores and more by how systems manage model execution, memory locality, and probabilistic generation as core principles. Inference stops being an application concern and becomes a force that influences hardware design, operating systems, developer tooling, and ultimately business value itself.

Introduction: the new compute paradigm

The rise of the inference economy

For most of the history of machine learning, economic value was concentrated in training. Organizations invested heavily in data collection, model design, and large-scale training runs, while inference (the act of using a trained model) was treated as a marginal cost. That assumption no longer holds. As generative and foundation models move into daily operational use, inference becomes the dominant and continuous expense, transforming from an afterthought into the primary driver of infrastructure design and spending.

Inference differs from training in both frequency and economic profile. Training is episodic and capital-intensive; whereas inference is persistent and demand-driven. A single successful model may be trained a handful of times but executed millions or billions of times across its lifecycle. Each interaction (e.g. text generation, image synthesis, recommendation, or classification) consumes compute, memory bandwidth, and energy. The aggregate effect shifts cost centers away from occasional large clusters toward always-on, latency-sensitive fleets of accelerators and memory-optimized systems. This shift produces a new economic layer where performance per watt, tokens per second, and memory efficiency become business metrics rather than purely technical benchmarks. Organizations begin to optimize not only for model accuracy but for throughput stability, tail latency, and hardware utilization rates. Small percentage gains in batching efficiency or memory reuse translate directly into substantial operating margin improvements at scale. Inference efficiency becomes a competitive advantage, not merely an engineering preference.



The inference economy also alters how value is captured and distributed. Revenue correlates with responsiveness and availability rather than solely with model sophistication. Services that reduce latency or increase concurrency unlock new classes of applications such as real-time copilots, embedded assistants, and autonomous decision systems. These applications were previously infeasible under higher cost or delay constraints. Infrastructure vendors, software platforms, and model providers increasingly compete on their ability to deliver predictable, low-variance inference at scale.

As inference workloads proliferate across data centers, edge devices, and local hardware, the boundary between product features and compute infrastructure erodes. Compute becomes an operational utility tightly coupled with user experience and unit economics. The rise of the inference economy signals a transition from model-centric thinking to execution-centric thinking. The sustained act of generation, not the singular act of training, defines both technological direction and financial outcomes.

Defining the stack

So, what defines this new inference stack? A good answer lies in the traditional path of user requests from input to silicon, and how abstractions align or fracture at each layer. Modern computing stacks were layered to separate concerns: applications issue requests, operating systems arbitrate resources, hardware executes instructions, and silicon enforces the physical limits of speed, energy, and density. This model worked because workloads were short-lived, predictable, and largely stateless. Inference breaks those assumptions.

Application layer

In the new inference model, the user request for inference sits at the top – this can be a prompt, an image, a stream of tokens, or a real-time decision. These requests are conversational or continuous sessions, rather than discrete transactions. They carry hidden state, accumulate context, and demand bounded latency rather than eventual completion. The application layer therefore behaves less like a program and more like an interactive pipeline whose performance is judged per token, per millisecond, and per watt.

> Inference requests are not discrete transactions but conversational or continuous sessions. They carry hidden state, accumulate context, and demand bounded latency rather than eventual completion.

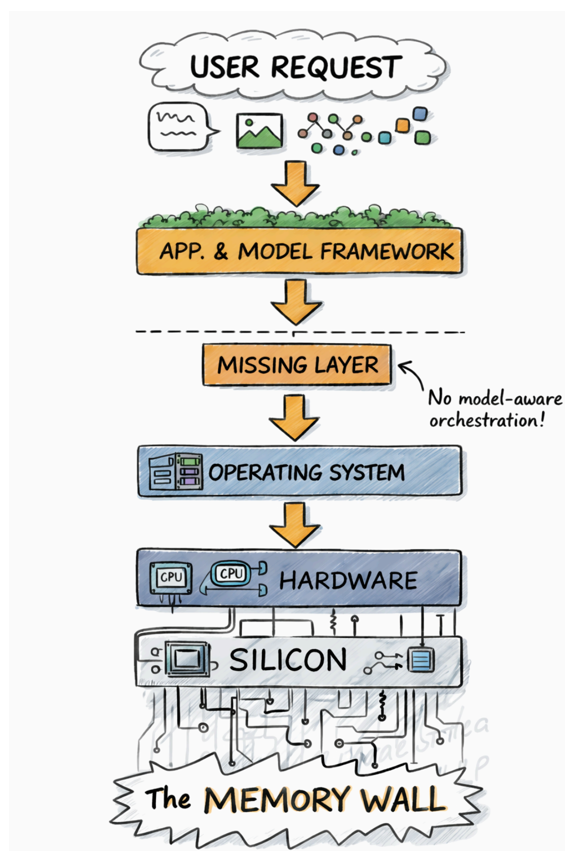
Frameworks and runtimes

Beneath the application layer lies the software stack: frameworks, runtimes, drivers, and libraries that translate high-level model operations into executable kernels. This layer was historically optimized for throughput in training jobs and batch analytics. Its abstractions assume large, uniform tensors, infrequent synchronization, and predictable memory layouts. During inference, those assumptions invert. Tensor sizes fluctuate, synchronization becomes frequent, and memory locality dominates performance. The software stack begins to leak

hardware details upward, forcing developers to reason about device placement, precision formats, and batching strategies that were meant to remain invisible.

The operating system

The next boundary is the operating system. It schedules processes, manages memory, and virtualizes devices, but it does so with a process-centric worldview. GPUs and accelerators are treated as peripherals attached to a host, not as primary compute domains with persistent state. Context switches, page faults, and driver handoffs introduce micro-latencies that accumulate across millions of token generations. The kernel remains logically correct while becoming economically inefficient, because its scheduling logic was never designed for workloads that are both long-running and latency-critical.



The silicon

Below the OS sits the hardware layer: CPUs, GPUs, NPUs, interconnects, and memory hierarchies. These components are extraordinarily capable, yet their efficiency depends on coherent orchestration. Compute units wait on memory, memory waits on bandwidth, and bandwidth waits on scheduling decisions made several layers above. Hardware exposes parallelism; it does not guarantee its utilization. The distance between theoretical peak performance and sustained inference throughput is largely a coordination problem rather than a silicon limitation.

At the base is silicon itself: transistors, fabrication nodes, power envelopes, and thermal constraints. Physics imposes hard ceilings on clock speed and energy density, which shifts optimization pressure toward data movement rather than raw arithmetic. The cost of moving bits increasingly exceeds the cost of multiplying them. This inversion quietly reorders priorities across the entire stack.

The missing layer: model-aware orchestration

What becomes visible in this vertical traversal is an absent layer: a model-aware orchestration plane that understands inference as a first-class workload. Between user intent and the operating system there is no standardized mechanism that preserves session state, batches heterogeneous requests intelligently, or schedules accelerators with knowledge of token generation dynamics. Frameworks approximate this role, and drivers expose fragments of it,

but neither owns the full lifecycle from prompt to last token. The result is fragmentation of responsibility and diffusion of efficiency. This missing layer is the inference engine in conceptual form; not merely a library, but an execution substrate that bridges semantic intent and physical resources. Its necessity emerges from the gap between probabilistic generation at the top and deterministic scheduling at the bottom. Without it, each layer optimizes locally while the system underperforms globally.

> The cost of moving bits increasingly exceeds the cost of multiplying them.

The tension becomes most acute at the boundary where computation meets memory. Persistent model weights, expanding context windows, and irregular access patterns collide with hierarchies designed for locality and predictability. The stack, as currently defined, lacks a coordinating intelligence that treats memory movement as the primary cost center. This absence leads directly to the next constraint: the memory wall.

The memory wall

As requests descend from applications through the operating system into hardware, the dominant cost shifts from arithmetic to data movement. Systems spend more time loading weights, moving tensors, and synchronizing buffers than performing the calculations that define inference.

Generative workloads intensify this imbalance. Large persistent models, growing context windows, and irregular access patterns fragment memory and saturate bandwidth. Accelerators remain theoretically powerful yet frequently idle, stalled by the delay of fetching data instead of executing instructions. The memory wall is therefore not a single hardware limit but a structural consequence of mismatched abstractions. Without a layer that understands model structure and session state, memory management remains generic while workloads are not, placing a hard ceiling on performance that raw silicon improvements alone cannot overcome.

How AI inference works: a high-resolution view

Predictive vs. generative models

AI inference is often described as “running a model,” but the behavior of that run depends heavily on the type of model involved. The most important distinction is between predictive models and generative models. Both use learned parameters to transform input data into output, yet their execution patterns, latency profiles, and resource demands differ in fundamental ways.

Predictive models answer a bounded question. Given an input, they return a finite label, score, or probability distribution. Examples include image classification, fraud detection, and

recommendation ranking. The computation is typically a single forward pass through the network: data enters, flows layer by layer, and then produces a result. The interaction is short-lived and stateless from the system's perspective. Once the output is produced, the process ends and the memory footprint can be released.

In simplified pseudocode, predictive inference resembles a function call:

```
None
function predict(input):
    features = preprocess(input)
    output = model.forward(features)
    return output
```

The cost profile here is dominated by matrix multiplications and memory reads for model weights. Latency is measured once per request, and throughput scales linearly with batching.

In contrast, generative models operate quite differently. Instead of returning a single answer, they produce a sequence over time, such as tokens, pixels, or audio frames. Each step depends on the previous output as well as the original input. The model is not merely queried; it is engaged in an iterative loop. This transforms inference from a discrete event into a session with persistent state and evolving memory requirements.

A simplified generative loop looks like this:

```
None
function generate(prompt, max_tokens):
    state = initialize(prompt)
    for t in range(max_tokens):
        next_token = model.forward(state)
        state = update(state, next_token)
        yield next_token
```

Two consequences follow. First, latency becomes cumulative rather than singular; performance is evaluated per token instead of per request. Second, memory usage grows with context length because the model must repeatedly reference prior tokens. What was once a stateless prediction becomes a stateful dialogue between input, model weights, and intermediate activations.

From a systems perspective, predictive inference is bursty and transactional, while generative inference is continuous and conversational. Predictive workloads stress compute units briefly and release them. Generative workloads hold accelerators for extended periods, demand

stable memory locality, and amplify scheduling inefficiencies. This difference is why generative AI reshapes infrastructure requirements rather than merely increasing utilization. It changes the temporal and structural nature of inference itself.

Tokens: the atomic unit of generation

Before looking at how generative workload executes, it helps to understand the basic unit that generative models operate on: tokens.

Large language models do not process text as words or sentences. They process tokens, which are small pieces of text created by a tokenizer. A token may represent a full word, part of a word, punctuation, or even whitespace depending on the vocabulary used by the model.

For example, the phrase: "Ubuntu: Linux for human beings" might be split into tokens such as:

```
None
["Ubuntu", ":", "Linux", "for", "human", "beings"]
```

Each token is converted into a numerical vector (an embedding) that the model processes through its neural network layers.

Two properties of tokens are important for understanding inference. **First, compute cost scales with tokens**, which is why many AI services price usage per token rather than per request. **Second, generative models produce output one token at a time**, feeding each new token back into the model to predict the next one. But why is generative inference sequential, and not parallel? Let's dive into how generative workload executes in the following sections.

The two-phase execution

Most modern generative models, especially large language models built on the Transformer architecture, do not generate output in a single pass. Inference is divided into two distinct execution phases: prefill and decode. These phases have different computational shapes, memory behaviors, and performance bottlenecks, which is why treating them as a single workload leads to inefficient scheduling and poor utilization.

At a high level, the Transformer is a stack of attention and feed-forward neural network layers that repeatedly transform token embeddings. The critical mechanism is self-attention, where each token attends to every previous token in the sequence. This dependency structure is what creates the two-phase split.

The prefill phase

The prefill phase processes the initial input sequence; often called the prompt or context. All tokens are known upfront, and the model computes their representations simultaneously. This phase is highly parallel and compute-intensive, resembling a dense matrix workload similar to training but without gradient updates.

The dominant operation is attention over the full input length. If the prompt has `N` tokens, attention complexity is approximately `O(N²)` because each token interacts with every other token in the sequence. A simplified view:

```
None
function prefill(prompt_tokens):
    embeddings = embed(prompt_tokens)
    for layer in transformer_layers:
        embeddings = attention(embeddings)   # full sequence
        embeddings = feed_forward(embeddings)
    kv_cache = extract_keys_values(embeddings)
    return kv_cache
```

Three key properties define prefill:

1. Parallelism: All tokens are processed at once
2. Compute-bound: GPU/accelerator arithmetic units are heavily utilized
3. Memory initialization: Key-value (KV) caches are created, storing intermediate representations for later reuse

In equation form, a single attention layer computes:

```
None
Attention(Q, K, V) = softmax(QKᵀ / √d) V
```

- `Q` (Query): This is what you are looking for (the current word being processed)
- `K` (Key): This is the label or index of all the information in the database (all other words in the sentence)
- `V` (Value): This is the actual information contained in those words.
- `d`: The dimension of the keys.

During prefill, `Q`, `K`, and `V` are matrices covering the entire prompt length, making the operation large but efficient for vectorized hardware.

The decode phase

The decode phase begins once the first new token must be generated. From this point onward, inference becomes iterative. Each new token is produced one step at a time, conditioned on all prior tokens and the cached context from prefill.

Unlike prefill, decode is sequential. Only one token (or a very small batch) is processed per step. The computation shifts from large parallel matrix operations to repeated small attention queries against an ever-growing memory of prior tokens. A simplified loop:

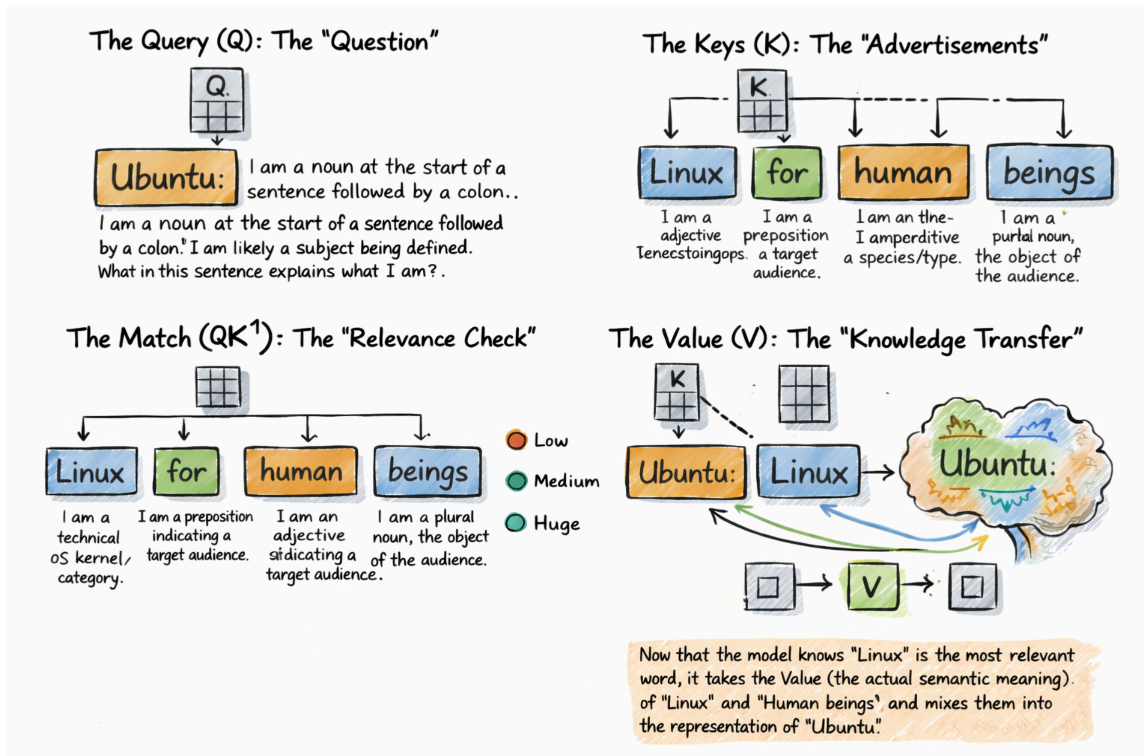
```
None
function decode(kv_cache, max_tokens):
    token = start_token
    for t in range(max_tokens):
        q = project(token)
        context = attend(q, kv_cache)   # query vs cached keys/values
        token = sample(model_head(context))
        kv_cache.append(token)
        yield token
```

Characteristics of decode:

1. Sequential dependency: Each step waits for the previous token
2. Memory-bound: Frequent reads from the KV cache dominate cost
3. Latency-sensitive: Performance is measured in tokens per second, not batch throughput

In practical terms, prefill saturates compute units, while decode stresses memory bandwidth and cache locality. The hardware profile flips: arithmetic intensity drops, data movement rises. This asymmetry is why inference systems that perform well during prompt ingestion can still feel slow during generation. The two phases are not merely implementation details; they are distinct workload classes that require different optimization strategies across scheduling, memory management, and accelerator utilization.

The math of generation



After the prefill phase, the model has read the entire prompt and built an internal map of relationships between the words. In the illustrated example above, it now understands that “Ubuntu” is not just a random token but something closely related to “Linux” and described by “human beings.” This understanding lives as a dense numerical vector inside the model: an abstract representation of meaning rather than a dictionary definition.

During the decode phase, the model repeatedly asks a single question: given everything I now know, what word is most likely to come next? Each step produces a new probability distribution over the entire vocabulary. The system is not retrieving a stored sentence; it is ranking all possible next words based on how well they fit the current context.

In the sketch, this is the moment where the enriched “Ubuntu” representation (already influenced by “Linux” and “human beings”) is projected outward into possible continuations. Words like “Linux” receive a very high probability because they strongly match the semantic relationship implied by the colon. Functional words such as “for” or punctuation receive lower probability because they add less definitional value at that position.

What we get after prefill and decode is not a finished paragraph but a continuously updated probability field. Each generated token slightly reshapes that field, which then influences the next decision. The apparent fluency of language comes from this rapid loop of contextual weighting and selection. In visual terms, the sketch shows arrows flowing from “Ubuntu” toward the most semantically aligned words, illustrating that generation is the act of

choosing the next token from a mathematically ranked landscape of possibilities rather than pulling a predefined phrase from memory.

To put it all together, one inference iteration looks like:

```
None
# Prompt (prefill): "Ubuntu: Linux for human beings"
tokens = tokenize(prompt)

# PREFILL: read all prompt tokens, build attention relationships, store KV
cache
kv_cache = []
for layer in transformer_layers:
    tokens, layer_kv = layer.forward_full_sequence(tokens)   # full attention
over prompt
    kv_cache.append(layer_kv)

# DECODE: repeatedly pick the next token using the cached context
while not stop_condition:
    # compute a contextual hidden state for the "current position"
    h = forward_one_step(transformer_layers, kv_cache, tokens[-1])

    # turn that hidden state into a probability distribution over the
vocabulary
    probs = vocab_distribution(h)   # "ranking all possible next words"

    # choose next token (greedy or sampling)
    next_token = choose(probs)

    # append to sequence and update caches for the next step
    tokens.append(next_token)
    kv_cache = update_kv_cache(kv_cache, next_token)

output_text = detokenize(tokens)
```

What emerges from this process is a system that is mathematically elegant yet operationally demanding: every generated token requires coordinated compute, precise memory access, and tightly timed scheduling across multiple layers of hardware and software. The model itself is rarely the sole constraint; the real friction appears in how the surrounding system feeds data, preserves context, and arbitrates resources at millisecond scale. As generation shifts from isolated predictions to continuous sessions, the limits of general-purpose operating systems become visible. The next key challenge is understanding why the underlying kernels that run our models struggle to keep up – and this sets the scene for our next discussion, where we examine the operating system bottleneck and why legacy designs fail modern AI workloads.

The OS bottleneck: why legacy kernels fail AI

The "logic + data" conflict

Modern operating systems were designed around a clear separation of concerns: the CPU executes logic, memory stores data, and file descriptors abstract persistent state such as disks, sockets, and devices. This model emerged from decades of workloads where programs were primarily instruction-driven: a compiler produced binary logic, the OS scheduled that logic on a processor, and data was fetched as needed from memory or storage.

The dominant assumption of this traditional model was that code is active and data is passive. Traditional software fits this model well; after all, a web server, database query, or compression utility is mostly “logic-only” from the scheduler’s point of view. The working data is comparatively small, locality is predictable, and the lifetime of memory allocations is short. The operating system can efficiently context-switch processes, page memory in and out, and treat files or devices as interchangeable resources because the program’s identity is defined by its instructions rather than by the volume or structure of its data.

AI inference upsets this balance. The model’s parameters – often billions of weights – are no longer passive inputs but an integral part of execution. The “program” is effectively the combination of logic and a massive, persistent data structure that must remain resident in accelerator memory to be useful. Every token generation step reuses the same weight matrices and context caches. Data is no longer incidental; it is the execution substrate itself. This creates a structural conflict with operating systems that still assume logic is cheap to move and data is cheap to reload. Page swapping, memory overcommit, and generic device abstraction introduce stalls because they treat model weights like ordinary buffers rather than long-lived computational state. The kernel can efficiently juggle small code segments, but it struggles when the identity of a process is tied to tens of gigabytes of continuously accessed tensors.

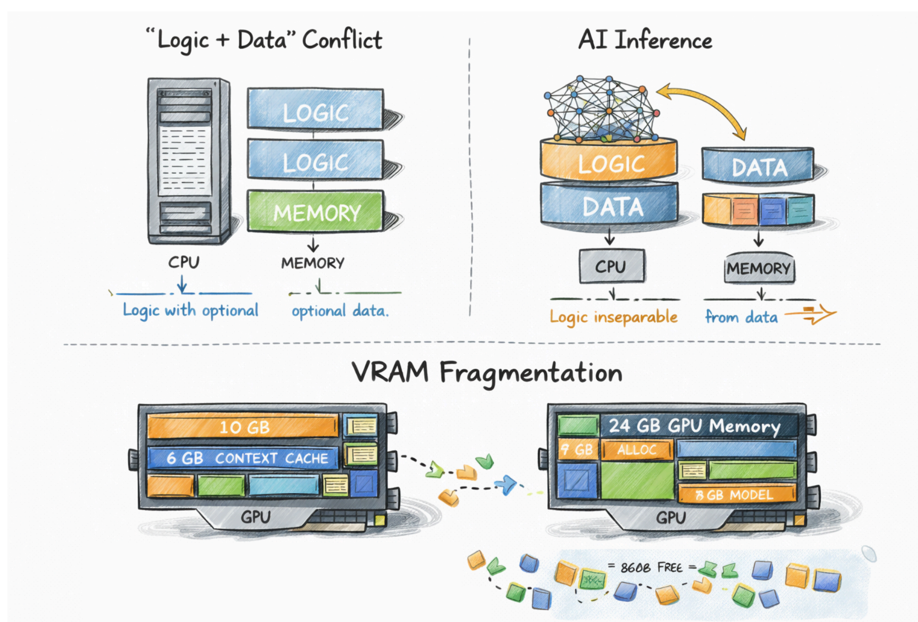
In effect, legacy kernels are optimized for logic with optional data, while AI inference is logic inseparable from data. The scheduler sees a process; the hardware sees a persistent numerical universe that must not be disturbed. The gap between these two perspectives is the root of the logic-plus-data conflict, and it is the first signal that the abstractions underpinning general-purpose operating systems no longer align with the realities of generative workloads.

VRAM fragmentation

VRAM fragmentation occurs when accelerator memory is technically “available” but unusable because it is split into small, non-contiguous blocks. Imagine a 24 GB GPU running several inference sessions. One model loads 10 GB of weights, another reserves 6 GB for a context cache, and multiple short-lived requests allocate and free smaller buffers over time. After

hours of activity, the GPU might report 8 GB free, yet no single block is large enough to load a new 7 GB model shard.

The problem is not total capacity but layout. Generative inference repeatedly allocates and releases variable-sized tensors: KV caches, attention buffers, temporary activations; which leaves “holes” scattered across memory. Unlike CPU RAM, VRAM cannot be easily compacted or swapped without stalling execution. The result is stranded memory: enough in sum, insufficient in shape, forcing model eviction or request failure even when headline utilization appears low. This fragmentation is equally relevant for edge devices as well as datacenters.



The global interpreter lock (GIL) and host overhead

A large portion of modern AI infrastructure is orchestrated in Python. Even when the heavy numerical kernels execute in C++ or CUDA, the control plane (i.e. model loading, batching, request routing, cache management, and tensor movement) often runs inside the Python interpreter. Historically, this introduced a structural constraint known as the Global Interpreter Lock (GIL), which allowed only one thread to execute Python bytecode at a time within a single process. For highly concurrent inference servers, this meant coordination logic could become serialized even while accelerators sat partially idle.

This situation is changing, but not yet universally. With PEP 703, Python introduced a no-GIL build that became available starting with Python 3.13 which continues to mature through Python 3.14. It is now technically possible to run Python without the GIL, but the ecosystem is in a transitional phase: many libraries, extensions, and deployment environments still assume the traditional interpreter model, and not all production stacks have migrated. As a result, real-world inference systems often operate in mixed conditions where parts of the stack are GIL-free while others remain effectively single-threaded.

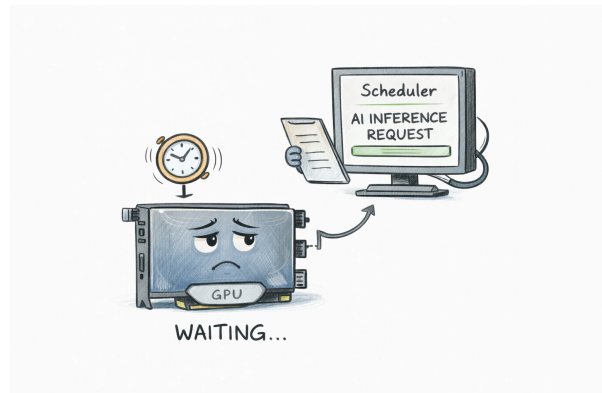
Even without the GIL, the broader issue of host overhead persists. Every inference request still traverses a CPU-side path: tokenization, batching decisions, networking, logging, memory bookkeeping, and synchronization with device drivers. These operations are individually lightweight but occur at high frequency and tight latency budgets. Removing the interpreter lock increases parallelism, yet it does not eliminate queue contention, object allocation costs, or cross-device data movement. The bottleneck shifts rather than disappears.

The practical outcome is that accelerators frequently wait on the host's ability to coordinate work, not on their own arithmetic limits. Whether constrained by the legacy GIL or by residual control-plane overhead in no-GIL environments, the imbalance reveals a deeper pattern: inference performance is often gated by orchestration efficiency as much as by model math.

Scheduling mismatch

At this stage, the limiting factor in AI inference is rarely raw compute power. Modern GPUs advertise enormous TFLOP numbers, yet real-world utilization often falls far below those theoretical peaks. The gap is not caused by insufficient arithmetic capability but by how work is admitted, ordered, and interleaved. In other words, the bottleneck shifts from silicon throughput to the scheduler.

The consequence is underutilization exposed as scheduler fairness. GPUs may context-switch too often, batch sizes may fluctuate unpredictably, and memory residency may be broken by generic eviction policies. A scheduler optimized for egalitarian CPU workloads inadvertently fragments accelerator time and disrupts token generation pipelines. The system reports high device availability while delivering inconsistent tokens-per-second and elevated tail latency.



The mismatch reveals a structural truth: inference performance is governed less by peak TFLOPS and more by who decides what runs next, where, and for how long. Without model-aware scheduling that understands session persistence, KV-cache locality, and phase asymmetry, hardware capability remains theoretical. The ceiling is no longer transistor density or clock speed; it is the intelligence of the layer that sequences work across the stack.

The rise of inference engines

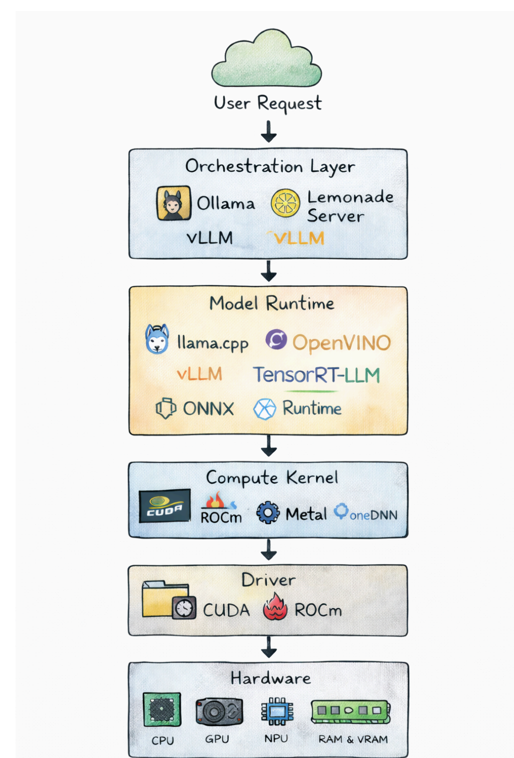
Breaking the research-production gap

Large models often demonstrate impressive performance in research environments but degrade once deployed in production. In the lab, inference runs in isolation: fixed prompts, stable batch sizes, clean memory, and dedicated hardware. Production conditions are the opposite; requests arrive unpredictably, context lengths vary, latency targets are strict, and memory becomes fragmented. The performance gap is not primarily a model issue; it is an execution-layer issue.

Inference engines emerge as the runtime layer that closes this gap. They sit between application frameworks and the operating system, keeping models resident in memory, coordinating dynamic batching, and managing shared caches across concurrent sessions. Instead of treating each request as a separate process, these engines treat inference as a continuous, stateful workload aligned with accelerator behavior. The effect is that optimizations once limited to benchmarking become standard operational primitives. Performance shifts from theoretical peak numbers to sustained throughput and stable latency under real traffic. The model remains the same; the difference is the system that runs it.

Architecture: layers and components

Before discussing optimizations, it is necessary to clarify what an inference engine actually is. An inference engine is not a single binary or library, but a layered runtime stack that connects user requests to accelerator hardware. Each layer has a different responsibility: executing math, managing memory, coordinating sessions, or packaging models. Performance emerges from how well these layers align, and not from any one component in isolation.



At the base is hardware and silicon: CPUs, GPUs, system RAM, and VRAM. Above this sits the device driver, which exposes hardware capabilities to the operating system and enforces resource boundaries. On top of the driver lives the compute kernel layer such as CUDA (NVIDIA), ROCm (AMD), Metal (Apple), or Vulkan. This layer provides the primitives for launching parallel kernels, allocating device memory, and moving tensors between host and accelerator. It is the level where numerical operations are actually dispatched.

Above the compute layer is the model runtime. This is where projects like llama.cpp, TensorRT-LLM, ONNX Runtime, or vLLM operate. A runtime loads model weights, executes transformer layers, manages attention caches, and interfaces directly with CUDA or Metal.

- **llama.cpp** is a lightweight C/C++ runtime optimized for portability and local execution. It can run efficiently on CPUs and consumer GPUs, with minimal dependencies and strong support for quantized models [1].
- **ONNX** Runtime focuses on cross-platform compatibility, allowing the same model graph to run across many hardware backends [2].
- **OpenVINO** targets Intel hardware in particular; CPUs, integrated GPUs, and NPUs [3]. It provides graph optimizations and precision tuning tailored for edge and enterprise deployments where power efficiency and hardware heterogeneity matter.
- **vLLM** and **TensorRT-LLM** are designed for data-center workloads, emphasizing multi-GPU scaling, continuous batching, and deep memory orchestration [4, 5].

Above the runtime sits the orchestration or serving layer, which turns a raw executor into a usable system. This layer manages model downloads, lifecycle control, multi-request batching, API exposure, and routing across devices. Tools like Ollama and Lemonade Server live here.

- **Ollama** packages runtimes, model files, and configuration into a local developer-friendly service. It simplifies running models on laptops or workstations by abstracting away driver and runtime details [6].
- **Lemonade Server** and similar platforms focus more on managed or distributed serving, handling concurrency, scaling, and operational policies across machines or clusters [7].
- **llm-d** acts as an inference control plane, coordinating where and how inference runs across multiple runtimes or nodes, maintaining session continuity and routing decisions rather than executing model math itself [8].

Another critical element is the model file format. In the llama.cpp ecosystem, this is commonly **GGUF** (GPT-Generated Unified Format) [9]. A GGUF file is more than weights; it bundles quantized tensors, tokenizer data, and architecture metadata into a structure optimized for fast loading and memory mapping. This allows partial loading and efficient startup, especially on resource-constrained systems. Other ecosystems use formats such as SafeTensors or ONNX, each tuned for different runtimes and deployment styles. Viewed vertically, the stack becomes:

User Request → Orchestration Layer (Ollama, Lemonade, vLLM) → Model Runtime (llama.cpp, vLLM, TensorRT-LLM) → Compute Kernel (CUDA / ROCm / Metal) → Driver → Hardware

An inference engine is the coordinated behavior across these layers. The runtime executes the model, the orchestration layer manages concurrency and lifecycle, the compute kernel drives the hardware, and the model format determines how efficiently everything loads and runs. The effectiveness of the system depends less on any single layer and more on how tightly these layers are integrated.

Core techniques in inference engines

Inference engines did not emerge because of one optimization, but because a few key ideas changed how memory and concurrency are handled during generation. Two of the most impactful breakthroughs are PagedAttention and Continuous Batching. Both focus less on faster math and more on smarter coordination of memory and requests.

Quantization and Mixed Precision

Quantization reduces the numerical precision of model weights and activations from 32-bit or 16-bit floats down to 8-bit or even 4-bit representations. Mixed precision combines different levels of accuracy where needed. The practical effect is smaller memory footprint, faster data movement, and the ability to fit larger models on the same hardware, often with minimal impact on quality.

PagedAttention

PagedAttention is a memory management technique for the model's context cache (the key–value tensors used in attention). Instead of allocating one large, fixed block of GPU memory per session, it splits memory into smaller "pages" that can be reused and rearranged as needed. In simple terms, it treats GPU memory more like virtual memory in an operating system. When a user stops generating text or a session shrinks, its unused pages can be reassigned to another request instead of remaining locked. This reduces wasted space and prevents situations where memory is technically free but unusable due to fragmentation. The result is higher model density per GPU and fewer out-of-memory failures.

Continuous batching

Continuous batching changes how requests share the GPU over time. Static batching waits for a full group of requests, runs them together, then finishes. Continuous batching instead allows new requests to join an already running batch at each decode step. The intuition is similar to a highway on-ramp rather than a closed convoy. Instead of stopping traffic to form a new group, the system merges incoming requests into the flow. This keeps the accelerator busy, smoothes latency spikes, and increases overall tokens-per-second without forcing strict synchronization between users.

Speculative Decoding

Speculative decoding uses a smaller, faster model to "guess" several future tokens ahead of the main model. The large model then verifies or corrects those guesses in batches. When the guesses are correct, multiple tokens are produced in one step instead of one at a time. This shifts generation from strictly sequential to partially parallel, significantly increasing tokens-per-second without sacrificing output quality.

Tensor and Pipeline Parallelism

Tensor and pipeline parallelism split a single model across multiple GPUs or accelerator units. Instead of loading a full model replica on each device, layers or weight matrices are distributed. This enables models that exceed the memory capacity of a single GPU and allows horizontal scaling without simple duplication. It turns multiple accelerators into one coordinated execution surface rather than isolated workers.

Token Streaming and Early Exit

Token streaming sends generated tokens to the user as soon as they are produced instead of waiting for the full response. Early-exit techniques allow intermediate layers to produce acceptable outputs for simpler queries. Together, these approaches reduce perceived latency and improve responsiveness, which is often more important to user experience than raw throughput metrics.

Toward inference-native infrastructure

Inference as a first-class citizen

Today's computing infrastructure still treats AI inference as an application layered on top of general-purpose abstractions. In this traditional conceptualization, the operating system sees a process, the scheduler sees CPU time slices, and the memory manager sees pages and buffers. None of these primitives understand tokens, context windows, KV caches, or session persistence. As a result, inference is forced to adapt to primitives that were never designed for probabilistic, long-lived, accelerator-centric workloads.

Making inference a first-class citizen means elevating it from "just another process" to a recognized execution primitive, similar to how networking, filesystems, and virtualization became native concepts in earlier eras. Instead of mapping model execution onto generic process and memory semantics, the system would expose inference-aware constructs directly: persistent model residency, shared accelerator pools, token-level scheduling, and memory regions optimized for attention caches rather than anonymous pages. In practical terms, this shift changes what the kernel and platform optimize for:

- GPU and NPU memory managed with awareness of long-lived tensors rather than short-lived buffers.
- Scheduling decisions that account for session continuity and decode phases instead of only CPU fairness.
- Device drivers and runtimes exposing stable residency guarantees rather than best-effort allocation.

The infrastructure begins to recognize that moving model weights or evicting KV caches is not equivalent to swapping ordinary memory; it is closer to tearing down an active service.

This transition mirrors earlier computing evolutions. Networking became first-class when sockets and TCP stacks moved into the kernel. Virtualization became first-class when hypervisors were embedded into hardware and firmware. Inference as a first-class citizen implies a similar realignment: hardware, drivers, operating systems, and runtimes converge around the assumption that model execution is continuous, stateful, and economically central rather than sporadic and optional.

The result is infrastructure that optimizes for sustained generation rather than burst computation. Instead of chasing peak TFLOPS, the system prioritizes memory locality, concurrency stability, and predictable latency under load. Inference stops being an application pattern and becomes an organizing principle for how compute resources are allocated and coordinated across the entire stack.

Inference snaps

We cannot move from today's general-purpose operating systems to a world where inference is a true first-class citizen overnight. The transition is architectural, not cosmetic. It requires answering foundational questions that current kernels were never built to address. What system calls would an operating system expose for model residency or token streaming? How should memory managers behave when requests are long-lived and stateful rather than short and disposable? Should the OS maintain explicit inference state for a user program, similar to how it tracks sockets or file descriptors? How does scheduling change when a "process" is actually a persistent model session that must not be evicted without economic consequence? These are not implementation details; they are research questions that touch kernel design, driver contracts, and hardware abstraction.

Because of this, the path forward is incremental. Instead of rewriting the operating system first, we introduce controlled layers that allow us to observe, standardize, and learn how inference behaves across heterogeneous hardware. Here at Canonical, Inference Snaps represent one of those early steps. They are not a new kernel primitive; they are a packaging and execution model that begins to teach the operating system what inference workloads look like in practice.

An inference snap bundles the model, runtime, dependencies, and silicon targeting logic into a single, reproducible unit. The goal is consistency across CPUs, GPUs, and NPUs without requiring the user to manually assemble drivers, libraries, or environment variables. The operating system starts to see inference not as a loose collection of binaries, but as a well-defined workload artifact with predictable resource needs.

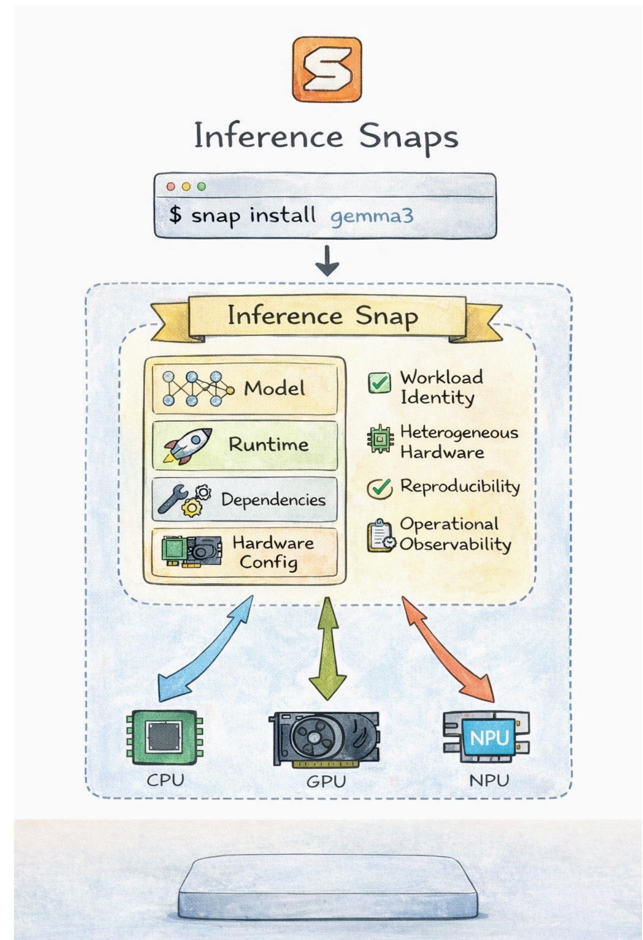
Consider a simple command: `snap install gemma3`

Under the hood, this does far more than installing a typical application. A traditional software install places executables on disk and resolves shared libraries. An inference snap, by contrast, pulls a model artifact, a runtime tuned for the host architecture, accelerator bindings, and

confinement policies that define how memory and devices are accessed. It configures GPU or NPU access, ensures compatible driver interfaces, and exposes a stable execution surface regardless of whether the machine runs an integrated GPU, a discrete accelerator, or only a CPU. The user experiences a single command; the system performs coordinated hardware and runtime alignment.

This approach introduces several core ideas that move the platform closer to inference-native behavior:

- Workload identity: The model and runtime become a named, versioned unit rather than scattered files and scripts
- Hardware awareness: The packaging layer understands heterogeneous silicon and selects compatible execution paths automatically
- Reproducibility: The same snap behaves consistently across environments, reducing drift between development and production
- Resource confinement: Memory and device access are explicitly declared, giving the OS clearer signals about long-lived accelerator usage
- Operational observability: Because inference runs inside a defined artifact, telemetry and performance characteristics become easier to measure and standardize



Inference snaps do not replace future kernel-level primitives, but they create the conditions to design them intelligently. They provide a bridge between today's application-centric operating systems and tomorrow's inference-aware infrastructure by making inference visible, structured, and portable. In effect, they bring the workload one layer closer to the silicon while the operating system learns how to meet it halfway.

The inference system call

If inference truly becomes a first-class workload, the operating system of the future might eventually expose primitives that acknowledge it directly. Today, kernels provide system calls such as `open()`, `fork()`, or `malloc()`; small, well-defined entry points that express fundamental intentions: open a file, create a process, allocate memory. These calls do not describe applications; they describe capabilities the system recognizes as universal.

Inference, in its current form, has no such native expression. It is assembled from many smaller calls: memory allocations, device handles, network sockets, and driver invocations. None of which understand that the user is asking the machine to generate tokens or maintain a model session. The idea of an "inference system call" is therefore not a concrete engineering proposal but a thought experiment about how operating systems might evolve if probabilistic computation becomes as common as file I/O.

One could imagine a hypothetical call[1] shaped less like launching a process and more like declaring intent:

```None
run_inference(model_handle, context_region, constraints)
```

In this imaginary world, the call would not run the model itself. Instead, it would signal to the kernel that a long-lived, stateful generation session is beginning. The operating system could then coordinate accelerator residency, memory locality, and scheduling continuity as a single semantic unit rather than as a collection of unrelated buffers and threads. The request becomes declarative: "sustain this model session with these latency and memory bounds"; rather than procedural.

Such a construct would blur boundaries that are currently rigid. Memory might be allocated as "attention-persistent" rather than anonymous. GPU or NPU access could be reserved with continuity guarantees rather than best-effort handles. The scheduler might treat the session more like a network socket, durable and stateful, than like a short CPU burst. None of this implies a specific API or kernel patch; it illustrates how the language of operating systems could shift if inference becomes a universal computing pattern.

The value of this imaginary system call is not in its syntax but in the mental model it introduces. Just as `open()` once abstracted disks and files into a unified concept, an inference primitive would abstract heterogeneous accelerators and model runtimes into a coherent execution intent. Whether such a call ever exists in code is secondary. The exercise highlights a broader trajectory: **operating systems may eventually need vocabulary that treats probabilistic generation not as an exotic workload, but as a normal, named capability of the machine itself.**

[1] This is the opinion of the author and does not necessarily represent the Ubuntu roadmap.

Business value of the optimized stack

To fuel further development in inference compute, the economics have to make sense. Generative AI is no longer a niche research curiosity. It is woven into modern workflows, agents, copilots, and digital services that millions rely on daily. The cost of inference directly impacts whether these systems can operate sustainably at scale; efficient execution is not a technical luxury but a business imperative.

Unit economics

Inference has become the dominant operational cost for AI services because every interaction consumes compute continuously rather than only during training. Between late 2022 and 2024, providers like OpenAI and Google cut the price of processing 1 million tokens from roughly $12 down below $2 for similar quality outputs, reflecting steep declines in per-token cost as hardware and software improve [10]. Cloud pricing has similarly dropped rapidly, with equivalent GPT-4 inference now costing about $0.40 per million tokens versus about $20 in late 2022 [11]. In this environment, small efficiency gains in throughput or memory use can translate into meaningful margin improvements, making optimized inference stacks a direct contributor to profitability.

Latency arbitrage

Fast, predictable inference unlocks latency arbitrage, which is the ability to serve real-time, agentic workloads that competitors cannot. As token costs decline and usage grows, markets have segmented into bulk offline compute and responsive online inference. Latency-optimized execution enables services such as real-time personalization, customer support bots, and autonomous decision agents that are intolerant of unpredictable stalls or high tail latency. Systems that reduce variance and improve tokens-per-second while keeping operational costs low capture value that slower, less efficient stacks cannot.

The MLOps lifecycle

Inference economics also reshapes the lifecycle from research to production. Generative workloads cross the boundary between development and operations continuously, and traditional silos between experimentation, staging, and serving inflate costs. Inference-aware stacks that preserve models, caches, and memory across environments reduce the divergence between prototype and production, cutting deployment friction and operational variance. This, in turn, accelerates innovation and lowers total cost of ownership across the MLOps pipeline, a compelling metric for enterprises integrating AI deeply into business processes. Taken together, these economic forces explain why optimized inference is not merely a matter of faster math. Efficient, predictable, and cost-aware inference directly influences business viability as generative AI becomes embedded in consumer products and enterprise systems at massive scale. Continued investment in infrastructure that aligns performance with economics is therefore critical for sustaining the future of compute.

Own your AI infrastructure

Data center AI deployment: Canonical private AI infrastructure

As discussed in the previous sections, generative workloads are reshaping the entire compute stack. This shift also raises a strategic question for organizations adopting AI at scale: **who owns the infrastructure that runs inference?**

For many teams, the long-term value of AI depends on controlling the execution environment. Running inference in your own data center, or across hybrid cloud environments, allows tighter control over latency, cost, security, and hardware utilization. It also enables organizations to optimize the stack end-to-end, aligning runtimes, accelerators, and orchestration layers with their specific workloads rather than relying solely on opaque external platforms. Open source infrastructure plays a key role in this transition. By combining operating systems, inference runtimes, and orchestration frameworks built on open standards, organizations can build AI platforms that evolve alongside the rapidly changing inference ecosystem.

If your team is evaluating how to deploy and operate AI infrastructure at scale, we strongly recommend you refer to [our guide to open source AI infrastructure].

Edge and IoT: Inference snaps

While data centers remain vital to large-scale inference, many AI workloads are moving closer to where data is generated. Industrial systems, robotics, gateways, and IoT devices increasingly require local inference to reduce latency, improve reliability, and preserve data privacy.

This is where inference snaps, introduced earlier in this paper, become particularly valuable. By packaging models, runtimes, and dependencies into a single reproducible artifact, inference snaps allow AI workloads to be deployed consistently across heterogeneous edge hardware, from CPUs to GPUs to NPUs, without rebuilding the entire runtime environment for each device. Inference snaps therefore extend the same principles discussed throughout this paper: portable runtimes, hardware awareness, and reproducible execution; into the edge computing world.

If your organization is exploring AI deployments across edge or IoT environments, we welcome the opportunity to discuss how inference snaps and Ubuntu-based infrastructure can help build reliable, inference-native systems.

[Get in touch to learn more]

Conclusion

Computing has gone through several eras where software forced hardware and operating systems to reinvent themselves: graphical interfaces drove GPUs, the internet reshaped networking stacks, virtualization redefined data centers, and mobile computing altered power and silicon design. Generative AI represents another such inflection point. The change is not merely in applications, but in the fundamental nature of execution. Computers are being reshaped around the continuous act of inference rather than the discrete act of computation.

This paper traced how that shift unfolds across the stack. At the workload level, generative models differ from traditional predictive systems by being persistent, stateful, and memory-intensive. The "memory wall" emerges not as a hardware defect but as a consequence of primitives that were never designed for token-by-token generation. The operating system has become a visible bottleneck. Legacy kernels optimize for short CPU bursts and passive data, while inference binds logic and massive persistent tensors together. VRAM fragmentation, host overhead, interpreter locks, and scheduling mismatches show that GPUs are often idle not because they lack power, but because the surrounding system cannot coordinate work efficiently. The constraint shifts from TFLOPS to orchestration intelligence.

Inference engines arise as the corrective layer. They close the research-to-production gap by introducing memory paging strategies, continuous batching, cache reuse, and model-aware scheduling. These breakthroughs demonstrate that efficiency gains come from smarter coordination rather than faster silicon alone. The ecosystem evolves into layered runtimes, orchestration services, and standardized model formats that make inference portable across heterogeneous hardware. The trajectory then points toward inference-native infrastructure. Treating inference as a first-class citizen implies new operating-system semantics, packaging approaches such as inference snaps, and even speculative ideas like an inference system call. These are early signals of a broader realignment in which model execution becomes a recognized primitive rather than an improvised pattern built from generic process and memory calls.

Finally, the economic dimension reinforces the technical one. As generative and agentic systems integrate into everyday tools and services, sustainable unit economics, predictable latency, and streamlined MLOps lifecycles become decisive. Optimized inference is no longer an optimization exercise; it is the condition for viable products and scalable compute.

"Generative AI is changing computers … again" captures this convergence. Generative AI is not simply running on existing machines; it is redefining what those machines must be optimized for. The rise of inference engines marks the moment where execution, memory, operating systems, and business value begin to reorganize around a single reality: continuous, probabilistic generation is becoming a native capability of the computer itself.

References

[1] *llama.cpp: LLM inference in C/C++*. (2023). llama.cpp. Retrieved February 15, 2026, from https://github.com/ggml-org/llama.cpp

[2] *ONNX Runtime*. (2017). ONNX Runtime. Retrieved February 15, 2026, from https://github.com/microsoft/onnxruntime

[3] *OpenVino: OpenVINO™ is an open source toolkit for optimizing and deploying AI inference*. (2018). OpenVINO. Retrieved February 15, 2026, from https://github.com/openvinotoolkit/openvino

[4] *vLLM: A high-throughput and memory-efficient inference and serving engine for LLMs*. (2023). vLLM. Retrieved February 15, 2026, from https://github.com/vllm-project/vllm

[5] *TensorRT-LLM: TensorRT LLM provides users with an easy-to-use Python API to define Large Language Models (LLMs) and supports state-of-the-art optimizations to perform inference efficiently on NVIDIA GPUs. TensorRT LLM also contains ...* (2023). NVIDIA TensorRT-LLM. Retrieved February 15, 2026, from https://github.com/NVIDIA/TensorRT-LLM

[6] *Ollama - Start building with open models*. (2023). Ollama. Retrieved February 15, 2026, from https://github.com/ollama/ollama

[7] *Lemonade Server*. (2025). Lemonade: Local AI for Text, Images, and Speech. Retrieved February 15, 2026, from https://github.com/lemonade-sdk/lemonade

[8] *llm-d: Achieve state of the art inference performance with modern accelerators on Kubernetes*. (2025). llm-d. Retrieved February 15, 2026, from https://github.com/llm-d/llm-d

[9] *GGUF*. (2023). GGUF - GPT-Generated Unified Format. Retrieved February 15, 2026, from https://github.com/ggml-org/ggml/blob/master/docs/gguf.md

[10] Crosley, B. (2026, 2 9). *Inference Unit Economics: The True Cost Per Million Tokens*. Inference Unit Economics: The True Cost Per Million Tokens. Retrieved 2 14, 2026, from https://introl.com/blog/inference-unit-economics-true-cost-per-million-tokens-guide

[11] Spark, Z. (2025). *The LLM Cost Paradox: How “Cheaper” AI Models Are Breaking Budgets*. The LLM Cost Paradox: How “Cheaper” AI Models Are Breaking Budgets. Retrieved 2 14, 2026, from https://www.ikangai.com/the-llm-cost-paradox-how-cheaper-ai-models-are-breaking-budgets/