

Inference at Scale

Chris Qin, Co-founder & CEO, Xinference

Engineering Lessons from a Decade of Distributed Systems · 2026

Table of contents

Introduction	3
Why Static Configuration Fails at Scale	4
Plan, Then Re-Plan: How Dynamic Tiling Works	6
Decisions Best Made at Runtime	8
From DataFrames to Tokens: The Same Problem in LLM Inference	10
A Production Inference Checklist	13
For the technical leader	13
For the engineer	13
About the author	15
Inference glossary	16
Next step	17

The gap that decides whether an AI product survives.

In 2026, the quality gap between leading AI models is narrower than it has ever been. The gap that decides whether an AI product survives is somewhere else: in the latency users feel, the reliability operations teams answer for, and the cost line the CFO reads. All three live in inference: the unglamorous work of running models in production.

Most teams discover the same thing in the same order. Getting a model to run is a weekend project. Getting it to run fast, predictably, and at a cost that scales with the business is an engineering discipline. The symptoms are familiar: a service that performs beautifully in testing and falls over under real traffic, GPUs that sit half idle while users wait, infrastructure bills that grow faster than usage.

This book makes a specific argument: these are not new problems. They are the problems distributed data systems spent the last decade solving, wearing new clothes. I spent that decade building distributed computing frameworks: first Mars, an open-source engine for large-scale tensor and dataframe computation, then Xorbits, a distributed data science framework published at ICDE 2024, and now Xinference, an inference platform for production AI workloads. Across all three, one principle kept proving itself: systems that adapt while they run beat systems that are configured in advance. Every major failure we studied came from a decision made too early, and every major speedup came from deferring that decision until the system could measure reality.

The chapters that follow trace this principle from data engineering, where it is backed by published benchmarks, to LLM inference, where it now decides serving economics. Chapters 1 to 3 walk through the evidence: why statically configured systems fail at scale, how runtime adaptation works mechanically, and what it means to remove configuration burden from users. Chapter 4 maps the same structure onto inference: batching, KV cache management, and engine selection. Chapter 5 condenses it into a checklist you can apply to your own stack this week.

This book is written for two readers at once: the engineer who runs models in production and wants the mechanism, and the technical leader who has to choose infrastructure and wants the decision logic. Both can finish it in under an hour. Neither needs to take anything on faith: every number in the data chapters traces to published, peer-reviewed work.

Why Static Configuration Fails at Scale

Scaling failures rarely announce themselves in advance. A pipeline that runs cleanly on a sample dataset dies at ten times the volume, and the error message, out of memory, worker killed, job hung, points at a symptom, not a cause. For a technical leader, these incidents look random. They are not. Most of them share a single root cause: the system decided how to divide its work before it could see the work.

Here is the mechanism. Any distributed framework must split data into partitions and place those partitions on machines. The natural place to make that decision is up front, when the job is planned: look at the size of the input files, divide by the number of workers, and build the execution plan. Nearly every mainstream framework does some version of this. The plan is built once, before execution, from the only information available at that time: the size of the source data.

The problem is that the size of data in motion does not stay the size of data at rest. A filter can shrink a table by 99 percent. A join can multiply it. A group-by can concentrate most of the output on a handful of keys: the skew problem every data engineer eventually meets. After a few of these operations, the intermediate data flowing through the pipeline bears no resemblance to the input the plan was built from. Partitions sized for the input become wildly wrong for the intermediates. One worker ends up holding far more than its share, exhausts its memory, and the job dies: usually hours in.

The Xorbits research team measured how widespread this failure mode is. They took TPC-H, an industry-standard suite of 22 analytical queries, implemented it with the pandas API, and ran it on the major distributed Python frameworks at three dataset scales. At the smallest scale, almost everything worked. At the largest, roughly a third of a terabyte, the picture inverted: Modin failed all 22 queries, Dask failed 5, and PySpark failed 4, with out-of-memory errors and hangs dominating the failure reasons. Plain pandas, bound to a single machine, failed 17 of the 22 queries at the middle scale and all 22 at the largest.

The same study surfaced a second, quieter failure mode: migration cost. None of the distributed frameworks fully reproduces the pandas API they imitate, Dask and Spark both acknowledge the gaps in their own documentation, so code that ran on one machine does not simply run on the cluster. Engineers lose hours, sometimes days, hunting workarounds for missing operations, or restructuring logic around partition rules the original library never had. Dask, for example, requires users to specify chunk sizes explicitly, and certain linear-algebra routines only accept particular chunk shapes; configure them wrongly and the job throws exceptions rather than results. This cost never appears in a benchmark, but every team that has scaled a data workload has paid it.

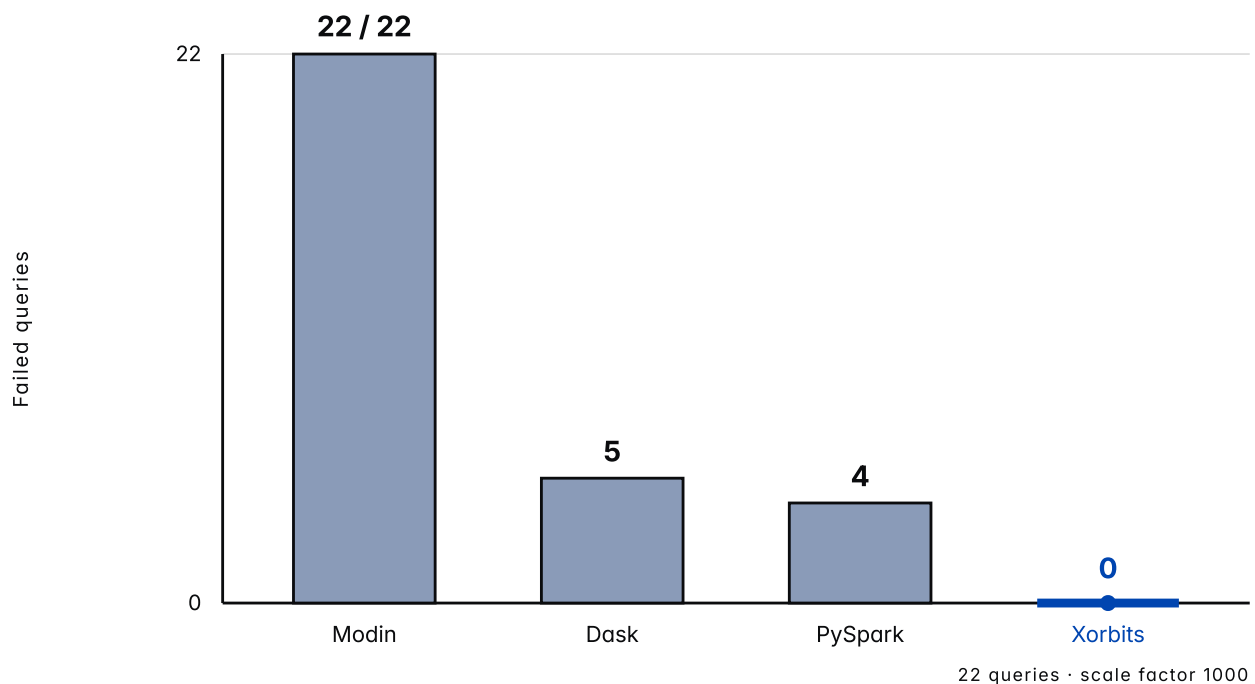


Figure 1. Failed TPC-H queries at scale factor 1000, out of 22. Out-of-memory errors and hangs dominate. Source: Lu et al., ICDE 2024.

One case from the same study shows the cause in isolation. A workload joined a 3.2-megabyte customer table with a 34-gigabyte transaction table: an everyday shape for enterprise data, and a textbook skew scenario. Frameworks that had fixed their partitioning before execution funnelled the join through what was effectively a single CPU core while the rest of the cluster idled. A system that could repartition based on what it observed at runtime completed the same workload 29 times faster than one baseline and 37 times faster than the other. Nothing about the hardware differed. The entire gap came from when the partitioning decision was made.

It is worth being precise about why this matters beyond data engineering, because the pattern is the subject of this whole book. A static plan encodes a prediction. At small scale, predictions are cheap to get wrong: everything fits in memory anyway. At production scale, the cost of a wrong prediction is concentrated on a single machine, and that machine fails. The failure is built in at planning time; the traffic merely reveals it.

“The failure is built in at planning time; the traffic merely reveals it.”

The practical takeaway is a question. For any system you operate or evaluate, a data engine, a query planner, an inference server, ask: when does it decide how to allocate resources, and what does it do when reality diverges from that plan? If the answer is “it decides up front and hopes,” you now know exactly where it will break, even if you cannot yet know when.

Plan, Then Re-Plan: How Dynamic Tiling Works

Planning is not the mistake. No system can place work on a cluster without a plan. The mistake is treating the plan as final: refusing to revise it once execution starts producing better information than the planner ever had. This chapter walks through how Xorbits builds revision into the plan itself, because the mechanism, published by our team at ICDE 2024, is the clearest worked example of the principle this book is about.

Xorbits compiles a user's program through three levels of graph. The first is a logical plan: each pandas or NumPy call becomes a node, with edges describing data dependencies: no notion of partitioning yet. The second is a chunk-level physical plan: data is divided into chunks, and complex operations expand into map, combine, and reduce stages over those chunks. The third is the execution plan: adjacent chunk operations are fused into subtasks, and each subtask is assigned to a specific worker. If this sounds like a database query planner, it should: the difference is that it operates on general Python data science code, not SQL.

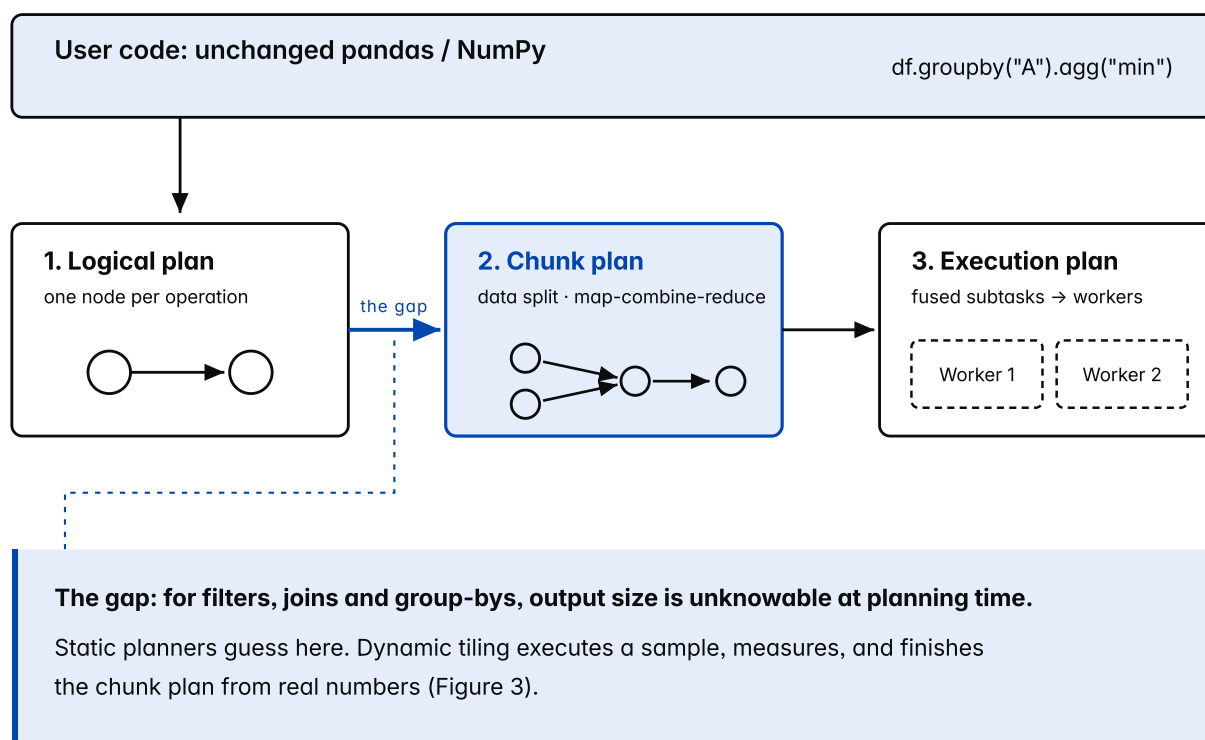


Figure 2. From unchanged pandas code to scheduled subtasks. The output sizes of filters, joins, and group-bys cannot be known at planning time: the gap dynamic tiling closes.

The compilation pipeline has a gap that cannot be closed ahead of time. Operators fall into two classes. For some, matrix multiplication is the classic case, the output shape is computable from the input shape, so the planner can size every downstream chunk on paper. But for the operators that dominate real workloads, filters, group-bys, joins, deduplication, the output size depends on the data's content, not just its shape. No amount of cleverness at planning time can tell you how many rows survive a filter. The information simply does not exist yet.

The Xorbits answer is to stop pretending it does. When the planner reaches an operator whose output it cannot predict, it pauses. It sends the first few chunks of that operator to the cluster for real execution, collects the resulting metadata, actual sizes, shapes, column types, and stores it in a metadata service. Then planning resumes, and the remaining chunks are tiled using measured numbers instead of guesses. The team calls this dynamic tiling: the planner and the executor take turns, each feeding the other, for as many rounds as the pipeline requires.

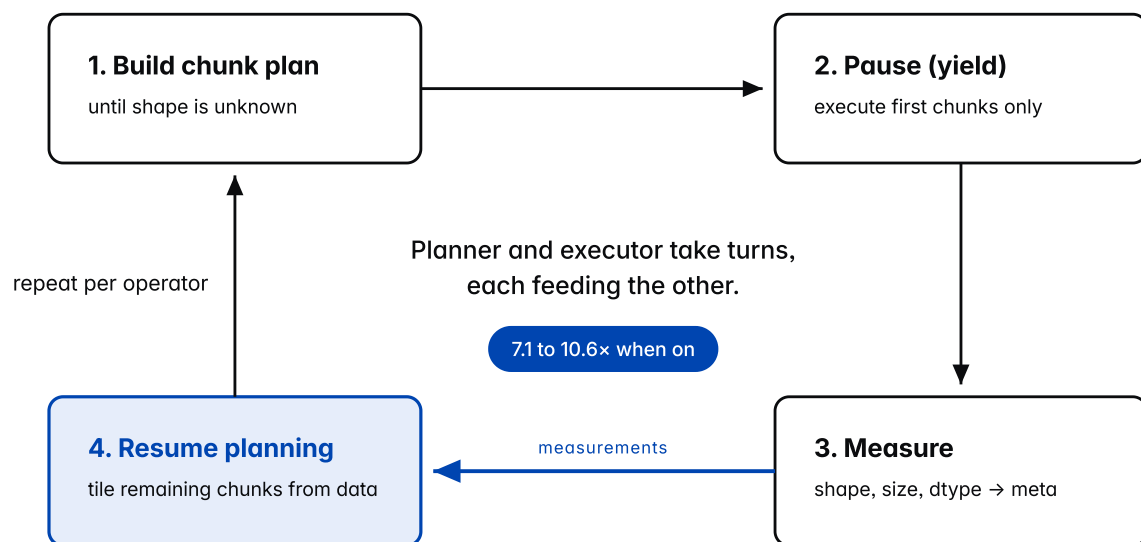


Figure 3. Dynamic tiling alternates between planning and execution. Disabling this loop slows join-heavy queries by factors of 7.1 and 10.6 (ICDE 2024 ablation).

The implementation rests on a mundane piece of Python: the yield statement, which lets a function suspend mid-execution and resume exactly where it stopped. An operator's tiling routine runs until it needs metadata that does not exist yet, yields to trigger execution of the sample chunks, and resumes with the measurements in hand. The language feature is simple; the system around it is not. Making the suspension safe required dedicated services for task management, metadata, scheduling, and intermediate storage, so that a half-built plan and a partially executed graph can coexist without corruption. That engineering investment is the real cost of runtime adaptation: and the benchmarks say it pays.

Across data science workloads, this design delivered a 2.66-times average speedup over the fastest competing framework on each task. The ablation results isolate the contribution: on two join-heavy analytical queries, switching dynamic tiling off slowed execution by factors of 7.1 and 10.6 respectively. And the design has been proven outside the lab: production Xorbits clusters have run at up to 5,000 CPU cores, processing workloads in e-commerce recommendation and financial risk analysis.

One supporting system deserves its own mention, because Chapter 4 will need it: the intermediate store. Every chunk an operator produces has to live somewhere until its consumers run, and at scale that “somewhere” becomes the bottleneck. Xorbits manages intermediates through a dedicated storage service organised as a hierarchy: shared memory first, spilling to disk when data outgrows it, with the option of a remote filesystem when intermediates exceed the cluster’s combined memory. Each chunk is addressed by a key, so workers read and write without knowing where the data physically sits, and zero-copy techniques keep transfers between processes from eating the gains the planner won. The principle is the same one driving the planner: the system observes how large intermediates actually are and moves them through the hierarchy accordingly, rather than asking anyone to provision for a guess.

Readers who know databases will recognise a relative of adaptive query execution, where engines like Spark SQL re-optimize plans using runtime statistics. The distinction is where the adaptation lives. Bolting statistics onto a SQL optimiser adjusts a plan within a constrained language. Pushing measurement-driven re-planning into the core of a general Python framework, while preserving the exact pandas and NumPy APIs, which have no concept of partitions, meant the entire mechanism had to be invisible. Users write single-machine code; the system measures and re-plans behind the curtain.

That invisibility is the practical lesson. Whenever a tool asks you to set a partition count, a chunk size, or a parallelism factor by hand, recognise what is happening: the system is exporting its planning problem to you, the person with the least runtime information of anyone in the loop. You are being asked to predict, at configuration time, what the system could simply measure at execution time. Sometimes that trade is acceptable. It should never go unnoticed.

“You are being asked to predict, at configuration time, what the system could simply measure at execution time.”



Chapter 3

Decisions Best Made at Runtime

Walk through the post-mortems of most production data incidents and you will find a configuration value at the bottom: a partition count, a memory limit, a parallelism setting that someone chose once, for a workload that has since changed shape. The previous chapter argued that systems should measure instead of predict. This chapter shows what that looks like as a design habit rather than a single feature, through three decisions Xorbits moved from configuration time to runtime.

The first is choosing an aggregation strategy. Distributed group-by can be executed two ways. Tree-reduce funnels partial results up a hierarchy of combiners: fast and simple, but only safe when the aggregated data is small, because everything eventually converges on few nodes. Shuffle-reduce redistributes data across all workers by key: it scales to any size, but pays a heavy communication cost. The right choice depends on how much the data actually aggregates, which is precisely the kind of fact you cannot know in advance: ten billion rows might collapse to fifty groups or fan out to a billion. Most systems resolve this with a fixed rule or a user setting. Xorbits resolves it empirically: it executes the first few chunks, measures how much the data shrinks, and picks tree-reduce when the measured result is small and shuffle-reduce when it is not. Per operation, per workload, with no human in the loop.

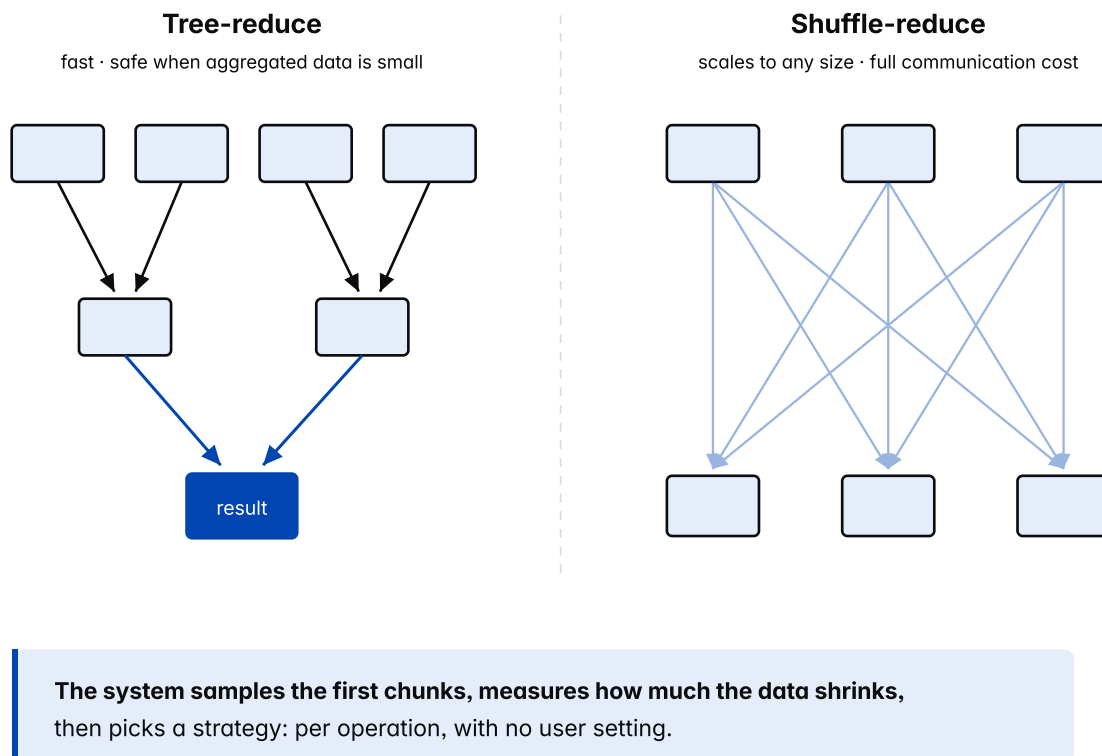


Figure 4. Tree-reduce versus shuffle-reduce. Xorbits measures how much the data actually aggregates and selects the strategy per operation.

The second decision is graph granularity. Chunked execution can produce thousands of tiny tasks, and at that point the overhead of scheduling the graph rivals the work itself. Using the same runtime measurements, Xorbits merges small chunks during the combine stage until they reach a sensible size, keeping the execution graph compact without anyone tuning it. Related compile-time optimisations multiply the effect: fusing adjacent graph nodes into single subtasks sped up two benchmark queries by factors of 3.8 and 2.0, and fusing at the operator level added a further 16 percent.

The third is shape correction. Some numerical operators only work on particular matrix layouts: QR decomposition, the workhorse behind least-squares and many ML routines, wants tall-and-skinny chunks. Frameworks that expose chunking ask users to re-chunk by hand and throw exceptions when they get it wrong. Xorbits computes the required chunk geometry from the operator's constraints and re-partitions automatically, so code written against NumPy runs unmodified.

Placement follows the same philosophy. Once the graph is fused into subtasks, each must be assigned to a worker, and Xorbits makes that assignment with two complementary strategies rather than a fixed mapping: a breadth-first pass spreads the graph's entry points across the cluster so no machine starts overloaded, and a locality-aware pass then keeps each subtask close to the data its predecessors produced, cutting transfer overhead. Scheduling even respects hardware boundaries below the machine level, treating each NUMA node or GPU as its own placement target. None of this is exotic individually; the point is the accumulation: partitioning, strategy selection, granularity, shape, and placement are all decisions the system makes from observation, not configuration.

That last point generalises into the number worth remembering from this chapter. In an API-compatibility benchmark of thirty common dataframe operations, Xorbits and Modin both scored 96.7 percent, while Dask reached 46.7 percent and PySpark 36.7 percent. Compatibility is usually discussed as a convenience feature. It is better understood as a direct consequence of runtime decision-making: every parameter the system decides for itself is a parameter the API never has to expose, and every parameter not exposed is single-machine code that runs on a cluster unchanged. The frameworks with low coverage are not careless: their architectures need the user to supply information that Xorbits measures.

Count the knobs. For the reader evaluating infrastructure, this suggests a blunt but effective metric: count the knobs. Every tuning parameter a system requires is a prediction someone on your team must make and maintain, and each one is a deferred failure waiting for the workload to drift. When you evaluate a platform, ask the vendor which decisions the system makes by itself, what those decisions are based on, and what happens when it gets one wrong. The quality of the answers tells you more about production behaviour than any benchmark slide.



Chapter 4

From DataFrames to Tokens: The Same Problem in LLM Inference

LLM inference presents itself as a new discipline with new vocabulary: tokens, KV caches, time-to-first-token, speculative decoding. Strip the vocabulary away and the resource problem underneath has the same structure as the one the last three chapters dissected. This chapter draws the mapping explicitly, because once you see it, much of what makes serving hard becomes predictable: and so do the solutions.

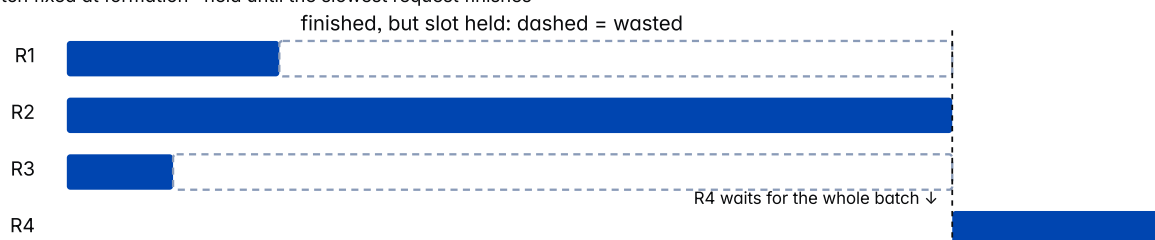
Start with the defining uncertainty. When an inference request arrives, the server does not know how long the response will be. One prompt yields ten tokens; the next, structurally identical, yields four thousand. Output size depends on content, not on input shape: exactly the property that made filters and joins impossible to plan ahead of time in Chapter 2. An LLM request is a non-static operator. Any serving design that pretends otherwise has rediscovered static tiling.

The earliest serving stacks did exactly that. Static batching collects a group of requests, pads every sequence to the length of the longest, and runs the batch to completion: no admissions, no departures, until the slowest request finishes. The plan is fixed when the batch forms. The result is the GPU equivalent of the skewed join in Chapter 1: short requests sit finished but held, padding burns compute on empty positions, and utilisation collapses as the variance in output lengths grows. The hardware is fine. The timing of the decision is the failure.

Continuous batching is dynamic tiling at millisecond scale. Instead of fixing batch composition once, the scheduler revisits it at every generation step: requests that finish leave immediately, waiting requests are admitted into the freed slots, and the batch in flight is perpetually re-formed around measured reality: who has actually finished, how much memory is actually free. It is the same move as pausing the planner to measure sample chunks, executed thousands of times per second. The throughput gains the industry attributes to modern inference engines come substantially from this one relocation of a decision, from batch-formation time to runtime.

Static batching

batch fixed at formation · held until the slowest request finishes



Continuous batching

batch recomposed at every generation step

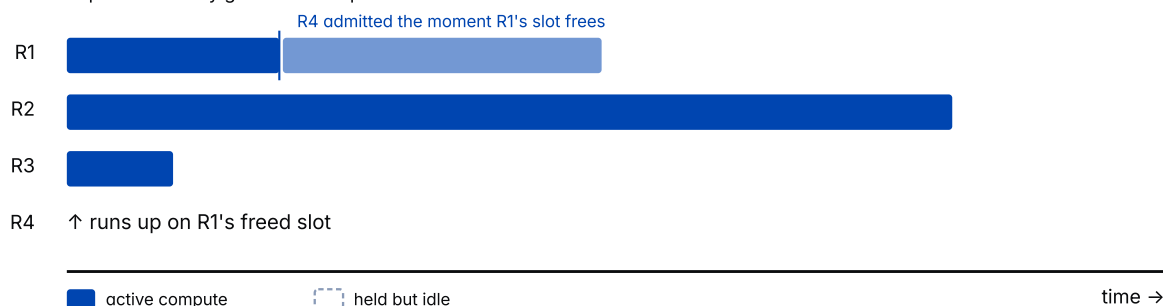


Figure 5. The same four requests under static and continuous batching. Continuous batching recomposes the batch at every generation step: dynamic tiling at millisecond scale.

The memory side of serving maps just as cleanly. During generation, every request accumulates a KV cache: attention state that grows token by token, for a length nobody knew at admission. This is the intermediate-result problem in GPU form: as with the join whose output dwarfed its input, the data produced mid-flight, not the data at rest, is what exhausts memory. The serving stack's responses mirror the data stack's. Paged cache allocation is fine-grained chunking; eviction and offloading policies are the storage hierarchy of Chapter 2's intermediate store, now spanning GPU and host memory; admission control informed by live cache pressure is the planner consulting the metadata service before committing.

There is a third layer the benchmarks of earlier chapters did not need: heterogeneity. A production AI platform serves many models, chat, embedding, speech, multimodal, across many runtimes and hardware tiers, and the right engine and configuration differ per model, per GPU, per workload. This is engine selection as a runtime decision, structurally identical to choosing tree-reduce versus shuffle-reduce: a choice that systems too often push to users as configuration.

It is worth pausing on why these scheduling details carry direct financial weight in a way dataframe scheduling rarely did. Inference capacity is bought by the GPU-hour, and a GPU-hour costs the same whether the device generates tokens or holds padding. Every percentage point of utilisation lost to static batch composition, stranded cache memory, or a mis-configured engine is a percentage point added to the cost of every token served. This is why the timing-of-decisions question moved from an architectural preference to a line item: two teams serving identical models on identical hardware can see materially different unit economics, and the difference is almost entirely in how their serving layer makes the runtime decisions this chapter describes.

This mapping is not retrospective pattern-matching; it is how we actually built. At Xinference, the inference platform runs on Xoscar, the same actor framework that powered our distributed data engine: the supervisor-and-worker architecture that scheduled dataframe subtasks now schedules model replicas and inference requests across NVIDIA GPU clusters. Concurrent requests are batched automatically by the platform rather than by the user's client code. Xinference supports vLLM, SGLang, llama.cpp, Transformers and MLX. TensorRT-LLM is available in Xinference Enterprise only. Users select the inference engine when launching a model. Changing the engine requires redeployment. A decade of watching static plans die taught us where to put the decisions.

Three questions for any serving stack. For the practitioner, the chapter reduces to three questions to ask of any serving stack, including your own. Does batch composition adapt at every step, or only when a batch forms? What happens when KV cache pressure exceeds GPU memory, is there a managed hierarchy, or a crash? And who chooses engine parameters per model, the system, from measurement, or an engineer, from a prediction? You already know, from three chapters of data, which answers age well.



Chapter 5

A Production Inference Checklist

Principles earn their keep when they compress into questions you can ask on Monday. This closing chapter condenses the book into two short lists: one for the leader choosing infrastructure, one for the engineer operating it. They are the questions I would ask of any serving stack, including ours.

For the technical leader

Five questions that separate systems that adapt from systems that hope.

- 01 **When are resource decisions made:** at deployment, at request arrival, or continuously during execution? Push for specifics; “autoscaling” is often a deployment-time rule wearing a runtime costume.

- 02 **What does the system measure** about live workloads, and which decisions consume those measurements? A metric that feeds only a dashboard is observability; a metric that feeds a scheduler is adaptation.

- 03 **How does it behave under variance:** skewed request lengths, bursty arrivals, one model starving another? Uniform-load benchmarks are marketing; variance is production.

- 04 **How many parameters must your team set** and maintain? Count them. Each is a prediction you are signing up to keep correct as workloads drift.

- 05 **What is the migration cost** in code changes? Chapter 3 showed that API compatibility reflects architecture: a platform that needs you to restructure your code is telling you which problems it has exported to you.

For the engineer

Five checks against the running system.

- 01 **Measure the variance of your output lengths**, not just the mean: variance is what static batching punishes, and your token-length histogram predicts your utilisation ceiling.

- 02 **Verify, with the scheduler’s own logs, that batching is continuous:** requests should depart the instant they finish and admissions should happen mid-flight; if both only occur at batch boundaries, you are running static batching with better branding.

- 03 **Know your KV cache policy precisely:** what fills it, what evicts, where evicted state goes, and what the latency cliff looks like when it overflows; if no one on the team can answer, the answer is currently “crash”.

- 04 **Confirm engine and configuration are chosen per model** rather than platform-wide, because the settings that suit a 70-billion-parameter chat model will mistreat an embedding model on the same cluster.

- 05 **Insist on observability into the scheduler’s decisions** themselves: not just request latency, but why the system admitted, batched, and placed work the way it did. You cannot debug decisions you cannot see.

Underneath both lists sits the one sentence this book exists to argue. A decade of distributed systems, across dataframes, tensors, and now tokens, keeps teaching the same lesson: do not ask people to predict what systems can measure. Every static configuration is a prediction, every prediction decays, and the systems that win at scale are the ones built to notice. The benchmarks in Chapters 1 through 3 show what that principle is worth in published numbers. Your own production metrics, examined through the questions above, will show you what it is worth to you.

“Do not ask people to predict what systems can measure.”

Chris Qin

Co-founder & CEO, Xinference

Chris Qin has spent his career on one stubborn problem: making a cluster of machines behave like one honest machine. Long before that problem was fashionable, it was his daily work. As the BDFL and core developer of Mars, a tensor-based framework for large-scale computation, he learned what breaks when carefully made plans meet real data on real clusters.

He refused to accept that tuning partition counts by hand was the price of scale. That refusal became Xorbits, the distributed data-science engine, deployed in production at up to 5,000 CPU cores, whose dynamic tiling research this book draws on, published at ICDE 2024 with Chris as corresponding author (he publishes academically as Xuye Qin). The papers proved a principle he had watched hold for a decade: systems that adapt while they run beat systems that are configured before they start.

Then language models collided with production, and he watched the industry rediscover his old problem with new vocabulary. Unknown output sizes. Skewed workloads. GPUs idling behind static plans. So he built the third system: Xinference, the sovereign inference platform, headquartered in Australia with a strong presence in Singapore, serving enterprises across Asia-Pacific and beyond. It has earned more than nine thousand stars on GitHub, and runs production inference for banks, insurers, and manufacturers.

He remains what he has always been: an engineer who reads the scheduler logs himself, and who believes infrastructure should make decisions so its users never have to.

Inference Glossary

A short reference to the serving and distributed-execution terms used in this book.

Continuous batching	Recomposing the in-flight batch at every generation step: finished requests leave immediately and waiting ones are admitted into the freed slots, keeping the GPU full. The serving analogue of dynamic tiling.
Decode	The token-by-token phase of generation that follows prefill; its length is unknown when the request is admitted.
Dynamic tiling	Alternating planning and execution: build the plan until an output size is unknown, run a sample, measure, then resume planning with real numbers. The technique behind Xorbits.
Engine	A serving backend that executes a model, such as vLLM, SGLang, llama.cpp, Transformers or MLX. In Xinference, users select the engine when launching a model.
GPU-hour	The unit inference capacity is bought in. It costs the same whether the device generates tokens or holds padding: which is why utilisation is the cost lever.
KV cache	The attention key/value state a request accumulates token by token during generation. It grows with output length and consumes GPU memory.
Latency	Time to serve a request. For LLMs it splits into time-to-first-token and the interval between subsequent tokens.
Paged attention	Allocating the KV cache in fixed-size blocks rather than one contiguous region, cutting fragmentation and allowing flexible eviction.
Prefill	The phase that processes the input prompt in parallel before generation begins; it dominates time-to-first-token.
Shuffle-reduce	An aggregation strategy that redistributes data across all workers by key. Scales to any size but pays a full communication cost. See tree-reduce.
Skew	Uneven distribution of data or load across partitions or keys, concentrating work on one worker and causing memory blow-ups and stragglers.
Static batching	Forming a batch and running it to completion; finished requests hold their slots until the slowest finishes. Wastes compute when output lengths vary.
Throughput	Tokens or requests served per unit of time across the system; the denominator of serving cost.
Time-to-first-token	Latency from a request arriving to its first generated token; governed by prefill. Often abbreviated TTFT.
Token	The unit a language model reads and writes: roughly a word piece. Output length, measured in tokens, is not known in advance.
Tree-reduce	An aggregation strategy that funnels partial results up a hierarchy of combiners. Fast and cheap, but safe only when the aggregated data is small. See shuffle-reduce.

The decisions this book argues for are the ones our platform makes for you.

The techniques in these pages are not a reading list: they are the engineering inside the X inference platform. Continuous batching, managed KV cache behaviour and actor-based scheduling on NVIDIA infrastructure run inside the platform, on your cloud, on-premises, or anywhere in between. X inference supports vLLM, SGLang, llama.cpp, Transformers and MLX. TensorRT-LLM is available in X inference Enterprise only. Users select the inference engine when launching a model. Changing the engine requires redeployment.

See it run at x inference.co. For ongoing engineering analysis like this book, follow X inference on LinkedIn.

