Lizard LLM / Caterpillar

APPLIED AI ENGINEERING

From model file to reliable product

A personal engineering portfolio
Local inference, native execution, and the systems around them.

Gerry Walter

Senior Applied AI Engineer

Second edition / September 2026

Technical case study with implementation references and illustrated workflows.

INTRODUCTION

The engineering behind local AI

A local model becomes a product only when a user can select it, run it, understand its behavior, and recover from failure. This book explains the decisions I made around that gap, from model identity and memory fit to native execution and measured improvement.

Why I built this system

I wanted local inference to be a usable workflow rather than a collection of fragile launch commands. Different Windows machines expose different memory budgets, GPU capabilities, and performance limits. A model recommendation is only useful when it leads to a supported execution path.

That requirement connected product engineering to systems work: artifact verification, provider policy, native process control, numerical graph construction, quantized kernels, and resource accounting. A correct kernel was necessary but not sufficient.

The resulting portfolio is about the boundaries I owned and the decisions evidence changed. It retains successful work, corrected comparisons, and experiments that did not justify a new default.

How to read this book

Start with the product and ownership chapters. The engine chapters explain the process and graph boundaries. Memory and performance chapters examine fit, quantization, device execution, cache state, and speculation. The final sections connect serving and recovery to testing and evidence.

Every chapter has an illustration, a side caption, and two text sections. The front contents follow reading order; the back index follows topics. Reference identifiers point to source files, historical records, or explicitly external reports.

This is a new edition, not a new benchmark campaign. Calculations are labeled, historical measurements retain their conditions, and external GPU results are not presented as Lizard or Caterpillar measurements.

Presentation perspective

My contribution is the connection between model capability and reliable local execution: identity that stays precise, plans that fit the machine, and results that can be explained and improved.

CONTENTS / 1 OF 2

A route through the book

Introduction / 02 Product, engines, execution, and evidence.

CONTENTS / 2 OF 2

A route through the book

The subject index on page 39 follows themes across chapters.

01 / PRODUCT AND OWNERSHIP

What Lizard is

Lizard turns a model file and a Windows computer into a usable local inference service. The product problem is broader than generating tokens: the model must fit, the selected runtime must support it, and the user must be able to understand and recover the execution state.

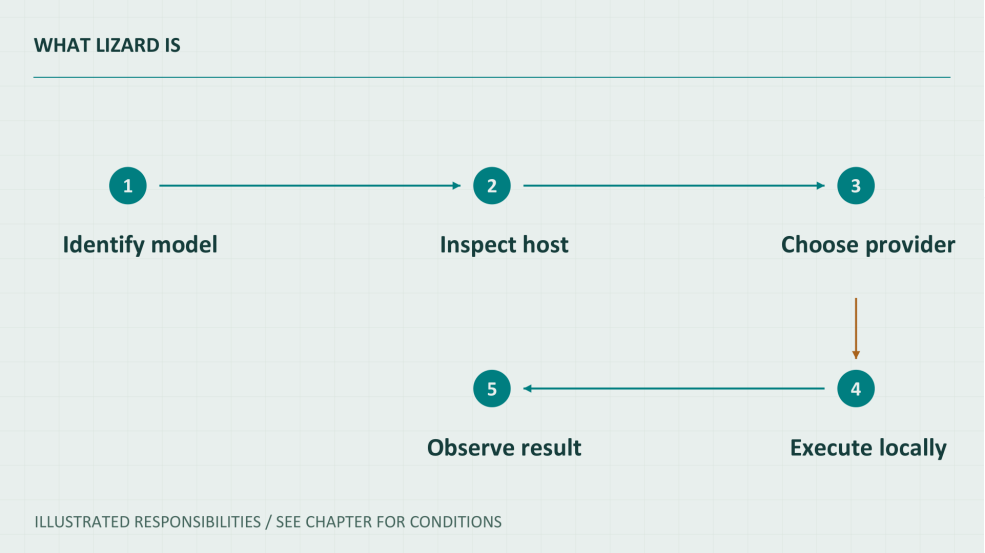

FIGURE 01

Product journey. Each handoff carries model identity, execution conditions, and a visible outcome.

The product I set out to build

The entry point is a practical question: what can this machine run reliably? A useful answer connects CPU and GPU capability, available memory, model identity, context length, and an executable provider. It cannot be inferred from parameter count or installed RAM alone.

I connected onboarding, verification, provider policy, local serving, and runtime telemetry around that question. A user should be able to download a model, activate it, inspect the selected execution path, and continue working through the same application contract when the implementation changes.

A result with context

The historical 3B optimization ledger records a shipped-default improvement from 4.18 to 9.67 tok/s on an i7-1165G7 reference machine. The corrected warm llama.cpp comparison remained faster at 12.50 tok/s. Both facts matter: the implementation improved substantially without establishing universal parity.

Lizard Native and Caterpillar represent different execution approaches under the product. Ostrich isolates scheduling research, while llama.cpp provides a compatibility implementation and comparison baseline. The portfolio follows those boundaries down to graph planning, quantized kernels, and memory behavior.

The evidence pages separate recorded measurements, calculated memory examples, and external reference runs. This makes the product story useful to a stakeholder and inspectable by a developer.

SOURCE NOTES R01, R07 | References on pages 35-36

02 / PRODUCT AND OWNERSHIP

Engineering ownership

My contribution is the connection between product behavior and numerical execution. I treat provider selection, model loading, scheduling, telemetry, and recovery as one engineering responsibility with explicit boundaries, rather than as unrelated features surrounding a fast kernel.

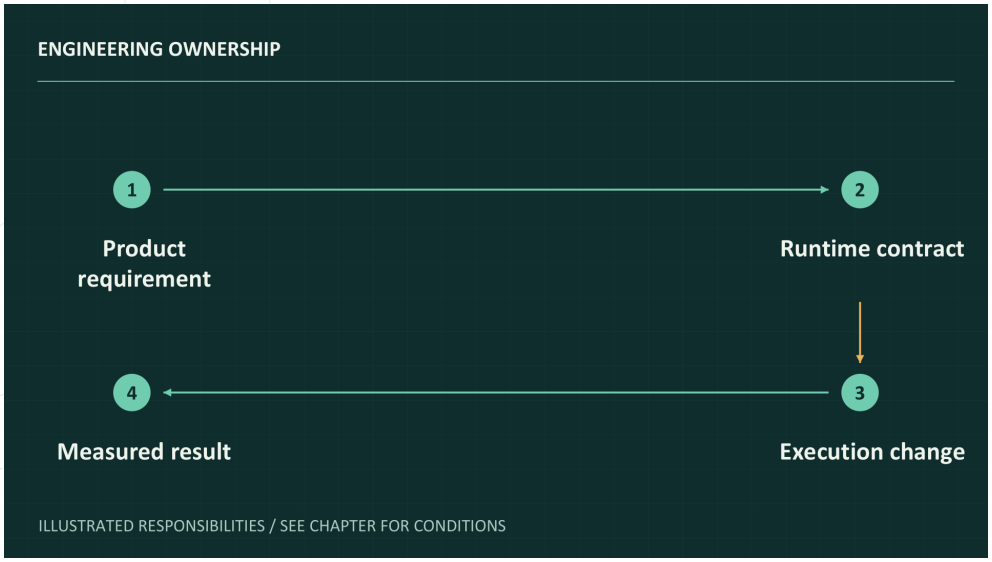

FIGURE 02

Ownership crosses layers. A performance change is complete only when its product behavior and evidence are understood.

Decisions I owned

At the product boundary I designed how a model becomes a verified, selectable runtime and how the application explains loading, warm state, stopping, and fallback. The same request contract needs to survive engine experiments without forcing every client to understand a native implementation.

At the execution boundary the work includes family-specific graph construction, reusable allocation, CPU quantization paths, Direct3D 12 resources, and KV state. These are connected decisions: changing a tensor layout affects loading, kernels, memory accounting, and the evidence needed before promotion.

Evidence changed the implementation

The performance work exposed a selector problem: integer kernels existed but the shipped configuration did not reach them. Fixing that route was more valuable than optimizing code that remained dormant. I kept a reference path and used paired measurements to decide which changes deserved a default.

The portfolio also records rejected work. More threads, a more advanced instruction path, or fewer dispatches did not automatically improve the tested workload. A negative result has value when its conditions and the resulting engineering decision remain attached.

In a presentation, I explain ownership through one traceable chain: requirement, boundary, implementation, measurement, and decision. That establishes what I built and why it changed.

SOURCE NOTES R01, R07, R08 | References on pages 35-36

03 / PRODUCT AND OWNERSHIP

Application and runtime layers

The architecture separates the application contract from the computation engine. This permits a stable local interface while providers differ in model support, scheduling, resource use, and operational behavior. The diagram is a responsibility map, not a claim that each layer is a separate process.

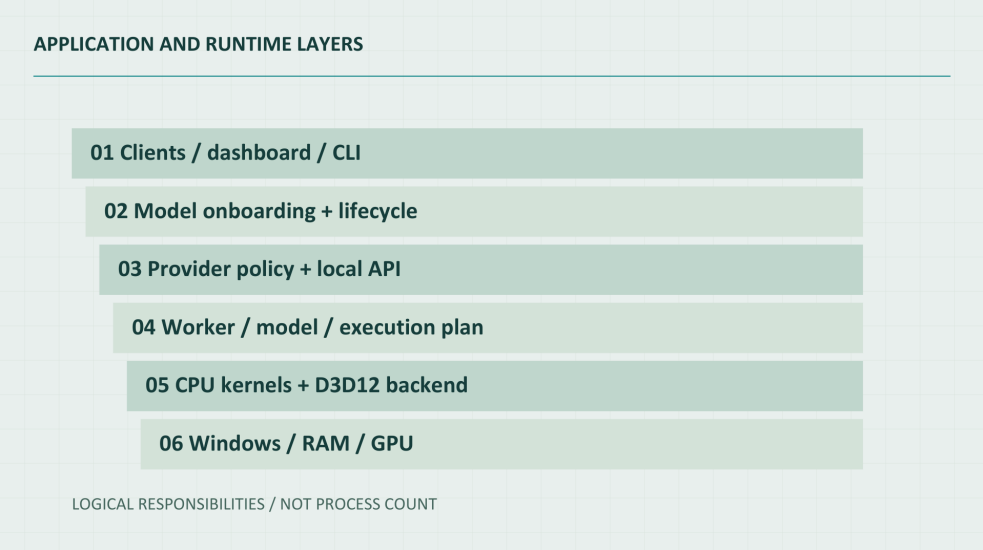

FIGURE 03

Logical application layers. Policy and lifecycle surround the runtime; numerical work remains below the provider boundary.

Above the engine

Desktop, dashboard, CLI, and API clients express user intent. Model onboarding handles catalog selection and artifact verification. Provider policy resolves an implementation and records why a requested route changed. Lifecycle code determines which model is selected, loaded, warm, or stopped.

The adapter translates a product operation into a worker request and interprets its result. It should preserve runtime identity and distinguish preparation, generation, failure, and cancellation. These fields are essential when two engines expose the same API but take different execution paths.

Below the contract

The runner interprets GGUF metadata, maps a supported family, creates a memory and execution plan, and performs prefill and decode. CPU and GPU backends own numerical operations and resource synchronization. The operating system supplies scheduling, memory budgets, and device capability.

An OpenAI-compatible endpoint is an integration contract, not proof of equivalence to every hosted API feature. Agent frameworks can use the local model through that boundary, while orchestration, tools, and retrieval remain responsibilities above the inference engine.

This separation provides a debugging route: first identify the failed boundary, then inspect the provider, plan, or kernel involved. A single generic model error obscures that route.

SOURCE NOTES R01, R02, R06 | References on pages 35-36

04 / PRODUCT AND OWNERSHIP

Two engines, one product

I retained multiple execution implementations because hardware and experimentation impose different constraints. The product can present one workflow without pretending that Lizard Native, Caterpillar, and the compatibility provider share an identical internal design or performance envelope.

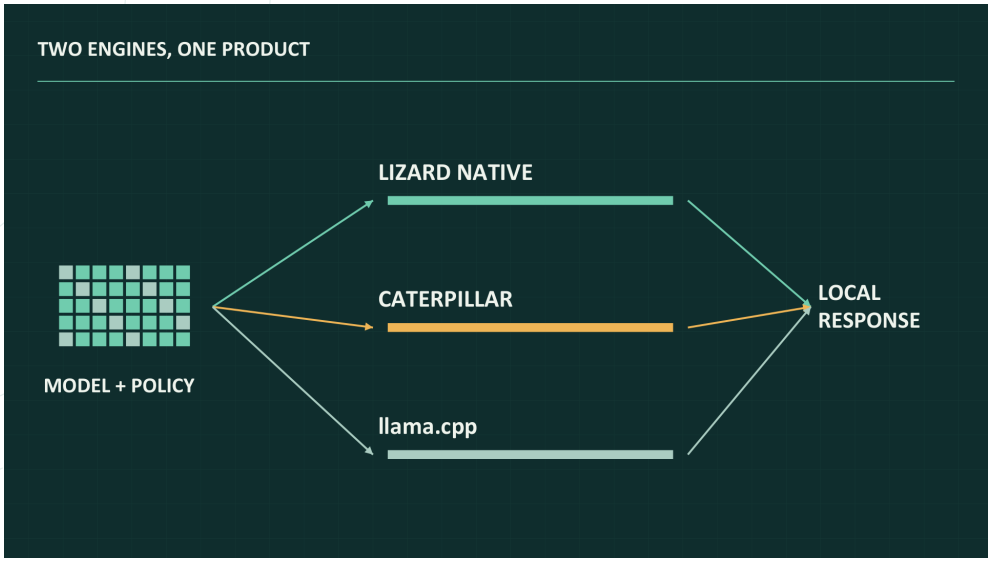

FIGURE 04

Alternative execution providers, not sequential stages. The policy checks support, fit, and configuration before choosing a lane.

The two native approaches

Lizard Native owns a native worker and Windows-oriented execution path. Resident GPU operation is valuable when the artifact, kernels, and complete working set fit the selected device. The local process boundary makes it possible to load, inspect, and replace the engine independently of the application.

Caterpillar makes the execution graph, allocation plan, and CPU/GPU spans explicit. It supports a standalone graph-plan workflow and quantized CPU execution alongside its device path. Shared graphics and a memory-constrained host can have a very different optimum from a discrete GPU.

Policy is more than a GPU test

The provider registry marks Caterpillar stable, Lizard Native experimental, and Ostrich experimental in the inspected snapshot. It also distinguishes the fresh-model default from the normalization fallback. A configured default and an implementation-status label describe different things.

Consequently, selection cannot be reduced to "GPU present means one engine." Family support, quantization, memory, requested configuration, and provider access all matter. llama.cpp remains an independent compatibility route and benchmark control rather than a hidden name for the owned engines.

The practical benefit is controlled substitution: a client keeps its request contract while the product records which implementation actually executed it and why.

SOURCE NOTES R01 | References on pages 35-36

05 / ENGINES AND EXECUTION

Lizard Native

Lizard Native is an owned runtime behind a standalone worker/server contract. It is the path for explicit native execution and Windows GPU work, but the product must still prove model compatibility, memory fit, and recovery for the actual configuration being requested.

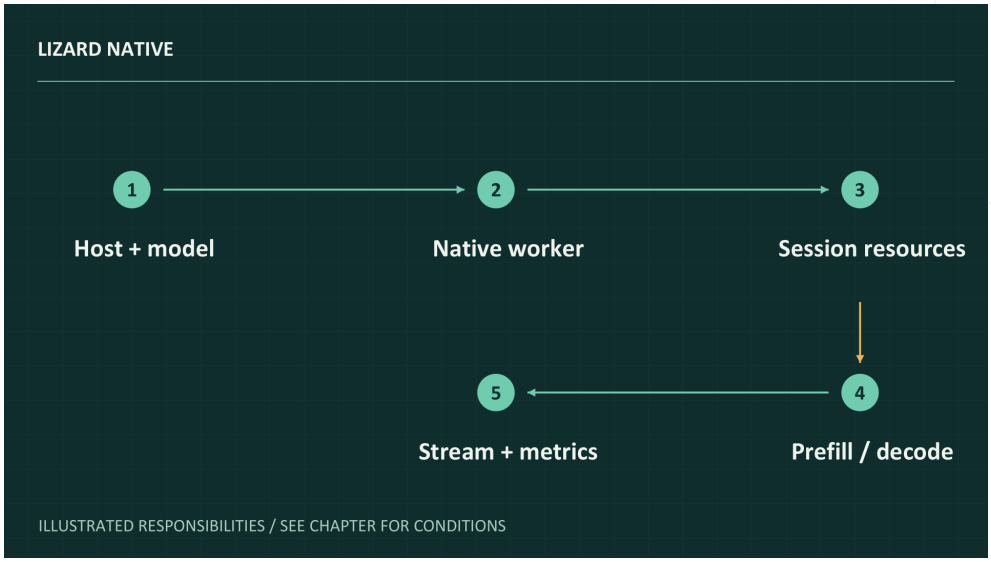

FIGURE 05

Native process boundary. Capability and performance are established for a specific loaded model and execution route.

A session owns resources

A native session brings together model identity, loaded weights, attention state, activation buffers, and request telemetry. The adapter prepares and starts that session instead of exposing device objects to every application feature. An unhealthy process can therefore be identified and replaced at a known boundary.

The execution decision begins with the host and model. A discrete GPU that can retain the complete working set may avoid repeated transfers. On a unified-memory machine, CPU and GPU contend for shared resources, so device availability alone does not establish that the GPU is the faster route.

Describe maturity precisely

The inspected registry labels this provider experimental even though it is the fresh-model configuration default. I retain that distinction rather than turning a product-facing role into a stronger maturity claim. The compatibility provider is a separate implementation, not an automatic guarantee that every failure is recoverable.

Earlier technical notes record device experiments and Gemma-shaped timing work. Those measurements belong to their named engine, workload, and revision. This edition does not copy a Caterpillar device result into the Lizard Native performance column merely because both use D3D12.

The engineering value is ownership of the native boundary: it makes resource use, provider differences, and replacement behavior measurable without changing the application interface.

SOURCE NOTES R01, R02 | References on pages 35-36

06 / ENGINES AND EXECUTION

Caterpillar

Caterpillar is the standalone C++ graph-plan engine. Its central design choice is to represent computation, storage, and backend scheduling explicitly, so execution can be inspected and tested before a model is promoted into a user-facing workflow.

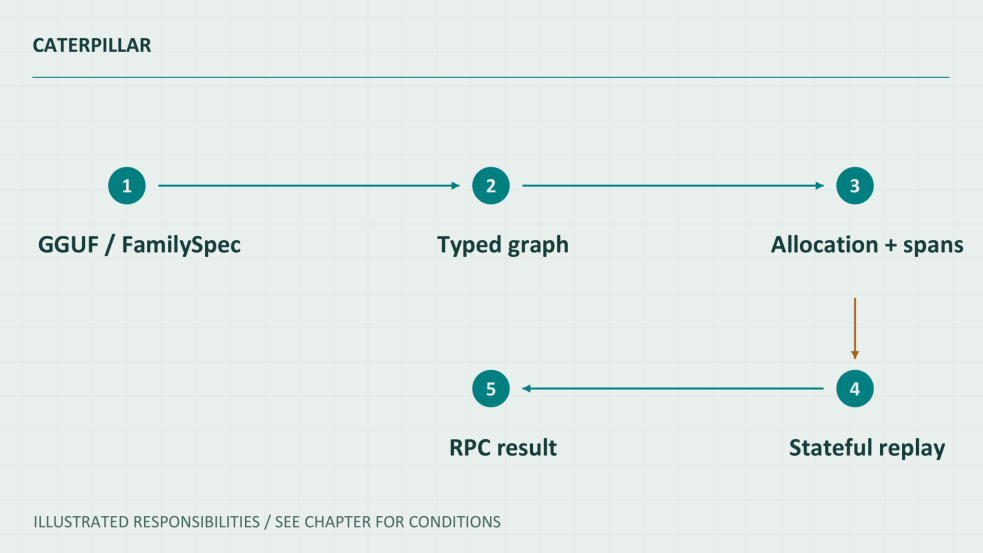

FIGURE 06

Caterpillar execution chain. Structural planning and changing request state are separate inputs to the runtime.

Compile once, replay with state

The path runs from GGUF metadata through FamilySpec and a typed graph to an allocation plan and backend spans. The plan is compiled for a model and configuration. Decode then supplies live inputs and past-token state rather than rebuilding the same structural work for each generated token.

The plan interface exposes reference replay, quantized bindings, dynamic position information, and a GPU executor. CPU and GPU spans meet through explicit bindings and arena storage. This creates concrete places to compare outputs, inspect allocation, and isolate scheduling changes.

A stable baseline for experiments

The registry currently marks Caterpillar stable. The historical handoff records 11 native CTests and 105 focused provider tests passing in its acceptance snapshot. Those are dated results, not a claim that every present configuration was retested during this editorial revision.

The recorded default-path improvement from 4.18 to 9.67 tok/s demonstrates a combination of selector correction, routing, and kernel work. A separate corrected warm reference remained faster. Retaining both outcomes keeps the optimization story tied to actual decisions.

Caterpillar provides a baseline against which a deeper experiment can be compared. Ostrich can explore a different schedule without silently changing the implementation being used as that baseline.

SOURCE NOTES R03, R07, R08 | References on pages 35-36

07 / ENGINES AND EXECUTION

Ostrich: isolated research

Ostrich is an experimental provider for scheduling and thread-affinity research. Separating it from Caterpillar gives performance hypotheses a place to be tested without turning an unqualified experiment into the stable product path.

FIGURE 07

Research promotion path. Historical comparison: 10.71 versus 10.14 tok/s; correctness alone did not justify a new default.

A concrete scheduling hypothesis

The recorded experiment attempted to reduce repeated worker wake-ups by dispatching at graph level. The intuition was reasonable: many small fork/join operations could consume time that a longer-lived parallel region might recover. But dispatch count is only one part of the execution cost.

On the documented Llama 3.2 3B Q4_K_M test, four threads and four interleaved rounds produced 10.71 tok/s for per-node dispatch and 10.14 tok/s for graph dispatch. The new path was about 5.3% slower despite token-identical output in that test.

Why the experiment stayed gated

The log identifies a scheduling problem: only matrix operations were distributed, while other operations ran on one worker and left the others waiting at a barrier. Reducing dispatch overhead did not compensate for serialization and synchronization. The graph-dispatch path remained off by default.

The registry requires experimental access for Ostrich. The code and historical log also distinguish new provider selection from already saved configurations, so a UI label alone is not an access-control proof. Request-facing policy remains the enforcement boundary.

The next useful change follows the observed bottleneck: distribute additional operation ranges, then repeat numerical, paired-performance, and recovery checks. The result rejects a particular implementation, not every possible graph-level scheduler.

SOURCE NOTES R01, R09 | References on pages 35-36

08 / ENGINES AND EXECUTION

A request in sequence

A local request crosses product policy, a process boundary, and numerical execution. The sequence matters because an early request acceptance, a loaded model, and a completed streamed response are different events with different failure and recovery behavior.

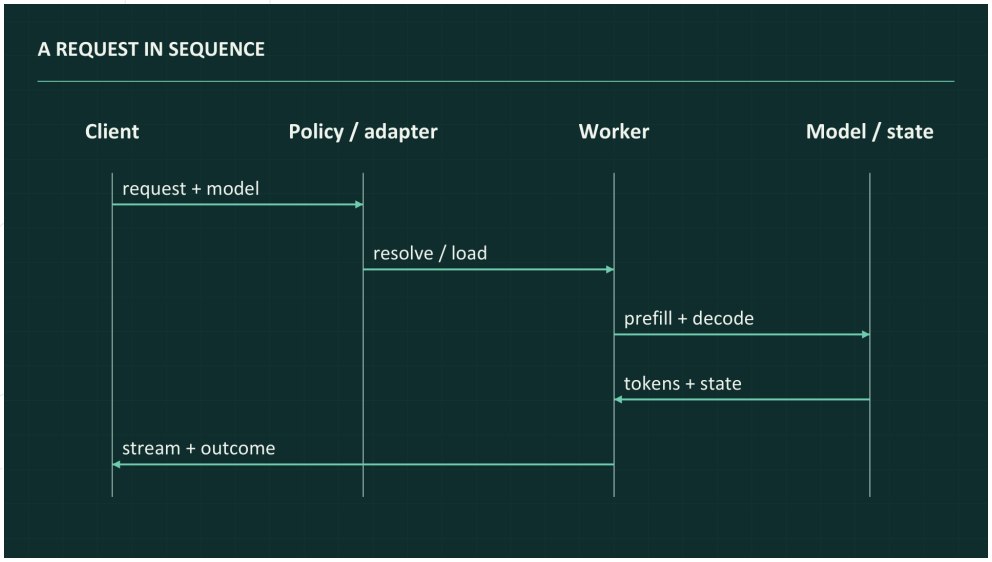

FIGURE 08

Request sequence. The arrows show ownership handoffs, not measured durations. Streaming and cancellation return through the same adapter boundary.

Before the first token

The client supplies a model and generation request. The application resolves model identity, checks the selected provider, and obtains a usable runtime. Verification and fit may lead to a different allowed plan, a compatibility route, or a refusal with a specific reason.

The adapter then loads or reuses the worker session. Prompt tokens enter prefill and create the attention state required for generation. Reusing a process does not automatically mean that the prompt prefix or every request-specific buffer can also be reused.

During and after generation

Decode produces tokens incrementally and returns telemetry through the adapter to the client. Stop sequences, output limits, cancellation, and worker health constrain the operation. Completion must preserve the distinction between a finished response and an interrupted stream.

The diagram is a conceptual happy path. An unsupported artifact exits before numerical execution; an allocation failure belongs to load or planning; a device failure belongs to the runtime. Those locations determine whether retry, a smaller context, or another provider is appropriate.

A useful trace connects request identity, artifact, provider, fit reason, lifecycle, and final outcome. That lets a developer investigate a slow or failed turn without guessing which implementation produced it.

SOURCE NOTES R01, R02, R06 | References on pages 35-36

09 / ENGINES AND EXECUTION

Model identity and GGUF

A GGUF file is more than a bag of low-bit weights. The loader must establish architecture, tensor layout, dimensions, and supported operations before the runtime commits a large working set or interprets a tensor through the wrong kernel.

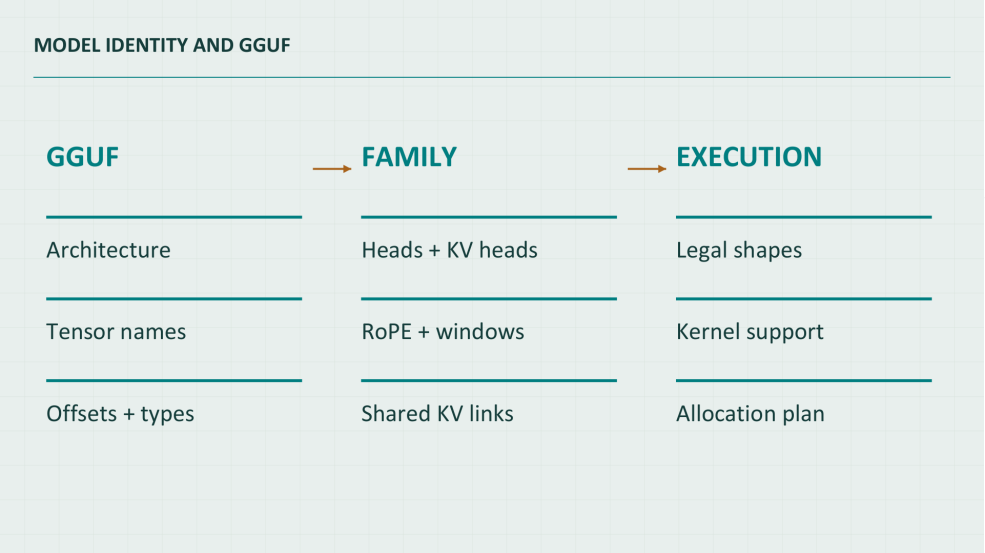

FIGURE 09

Identity before allocation. Metadata and tensor records determine the executable contract; the filename is not sufficient.

What loading must establish

The relevant metadata includes hidden and feed-forward dimensions, attention and KV head counts, head dimension, and rotary-position configuration. Tensor records add names, types, shapes, and offsets. Together these describe what computation is legal and where its data is located.

Family-specific behavior matters. Attention windows, shared KV relationships, biases, and positional settings cannot safely be inferred from a familiar model filename. FamilySpec exposes per-layer information as well as common dimensions so the graph can represent those differences explicitly.

Fail before expensive work

The loader and graph validation form an early diagnostic boundary. Missing tensors, unsupported quantization, or incompatible shapes should be identified before a long allocation or an opaque numerical failure. The application can then report a precise condition or select a compatible implementation.

Mapped access and staged access use the same artifact identity but have different memory costs. A mapped tensor offset describes storage location; it does not prove that the tensor is resident on the GPU or already in the operating system page cache.

For a reproducible benchmark, I keep the exact artifact and quantization alongside the runtime revision. Comparing two files with similar display names is not a controlled comparison.

SOURCE NOTES R03 | References on pages 35-36

10 / ENGINES AND EXECUTION

From family to decoder graph

The decoder graph translates model-family semantics into typed operations and tensor dependencies. It is a numerical execution graph, not an agent workflow graph: its nodes represent computation rather than planning, tool use, or a conversational role.

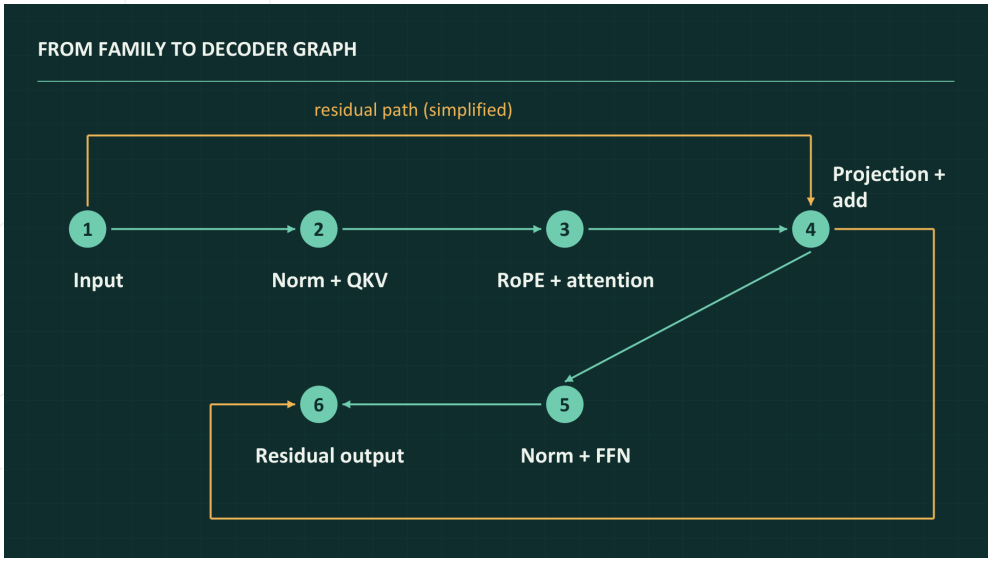

FIGURE 10

Simplified decoder block. Residual connections preserve the input around attention and feed-forward transformations.

Represent the actual computation

A Llama-style block combines normalization, Q/K/V projections, rotary positions, attention, output projection, residual addition, and a feed-forward path. Family-specific construction determines which tensors and parameters belong to each operation and how outputs feed the next stage.

The graph records runtime inputs separately from parameters and KV bindings. Prefill and incremental decode can then share the structural contract while carrying different live sequence lengths and past-token state. Missing dependencies and illegal shapes become testable graph errors.

Why the representation matters

An explicit graph provides information for lifetime analysis and backend planning. It also gives the reference implementation and optimized replay a common description to evaluate. That is more useful than comparing two unrelated code paths whose input interpretation may already differ.

The simplified diagram shows one decoder block, including its residual route. It omits family-specific extensions; it should not be mistaken for a complete Gemma or Qwen graph. The source FamilySpec and tensor directory determine the concrete model.

This boundary lets a developer ask whether a failure comes from model interpretation, graph construction, or execution. Optimizing a kernel cannot repair a graph that binds the wrong tensor or advances the wrong position.

SOURCE NOTES R03 | References on pages 35-36

11 / MEMORY AND PERFORMANCE

Memory fit is a product decision

The fit planner answers whether this model can run under the requested context and current host conditions. It uses staged weights, KV storage, activation memory, reserves, and live budgets; installed memory is not the same as memory available to this request.

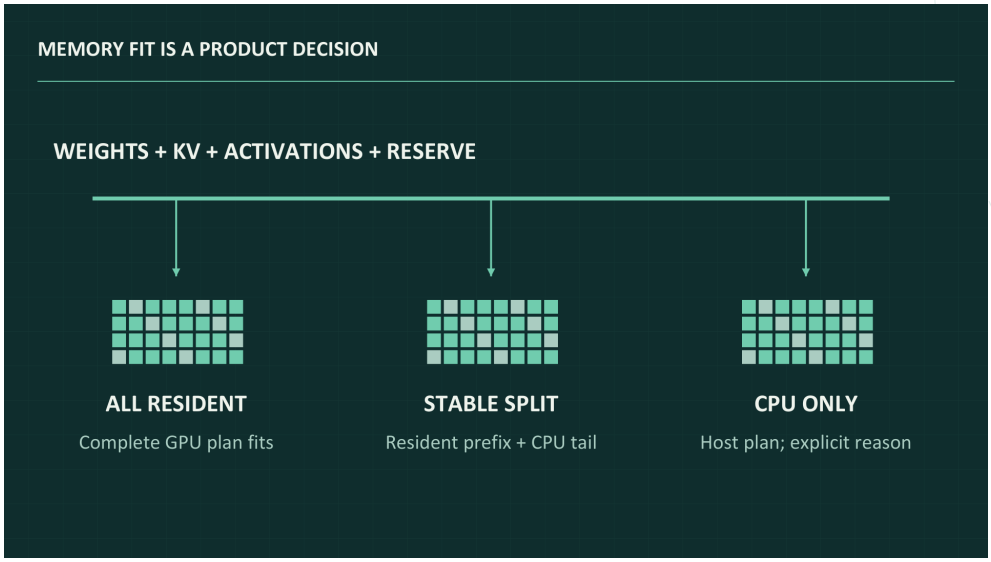

FIGURE 11

The three source-level fit outcomes. Unsupported artifacts or unusable plans are handled by the surrounding validation and provider policy.

Choose a stable plan

The source exposes three fit decisions: all resident, stable split, and CPU only. A stable split retains a complete layer prefix on the GPU while the remainder executes on the CPU. The intention is to avoid repeated per-token eviction and upload of the same weights.

The plan records its reason, budget source, adapter, resident layer count, staged bytes, KV storage, and possible CPU mirror. Separate host-envelope fields classify comfortable, tight, or oversized conditions using available and total RAM. Unprobed memory has an explicit reason rather than fabricated capacity.

Fit is not a speed guarantee

A model may fit yet run poorly because of bandwidth, synchronization, shared-memory contention, or unsupported kernels. Conversely, a smaller context or a CPU plan may offer more dependable behavior than a device plan at its memory limit.

Fallback or rejection is a surrounding product response, not a fourth value in the inspected FitDecision enum. Keeping those concepts separate makes the implementation easier to explain: the planner describes a route, and policy determines whether the request can proceed.

For the user, the best result includes a reason and an actionable alternative. For the engineer, it includes enough byte accounting to explain why a different model, context, or host changes the decision.

SOURCE NOTES R04 | References on pages 35-36

12 / MEMORY AND PERFORMANCE

The compiled graph plan

Caterpillar separates the logical graph from allocation and backend scheduling. Building those decisions once gives decode a replayable plan with dynamic inputs instead of forcing structural scheduling and allocation to be rediscovered for every token.

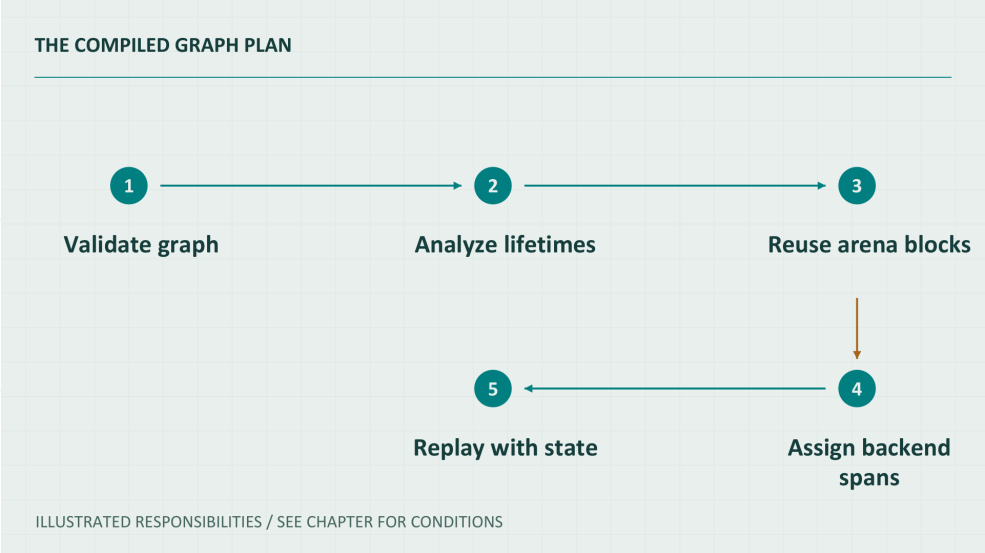

FIGURE 12

Compile-time structure and replay-time state. A plan is reusable only while its model and configuration contract remains valid.

Three related representations

The graph records operations and dependencies. Lifetime analysis identifies when an intermediate value is no longer needed, allowing its arena storage to be reused. Backend assignment groups contiguous operations into CPU and GPU spans whose boundaries require explicit data exchange.

The Plan interface exposes compilation, replay, allocation, and spans. ReplayDynamics supplies live past-token and position information for capacity-sized plans. The GPU executor consumes a span and communicates through external bindings and the CPU arena.

Where the benefit comes from

The design targets allocation churn and repeated orchestration work independently of the arithmetic kernel. It also provides a compact debugging surface: inspect the graph, inspect storage reuse, then inspect the span that produced an incorrect value.

Reuse must respect value lifetime. A buffer that becomes free in one execution shape may still be needed in another. Full and incremental replay tests therefore matter alongside standalone kernel checks, especially when cache position or shared-state behavior changes.

Compilation is not a promise of speed on every host. The measured bottleneck may still be matrix bandwidth, and a new backend partition may add transfers. A plan is valuable because its decisions are explicit and measurable, not because compilation itself guarantees an improvement.

SOURCE NOTES R03, R04 | References on pages 35-36

13 / MEMORY AND PERFORMANCE

CPU quantized kernels

Low-bit execution is a data-layout problem as much as an instruction problem. The useful path keeps weights in a compact representation, uses a compatible dot-product kernel, and avoids spending the saved bandwidth on unnecessary conversion or an unreachable optimization.

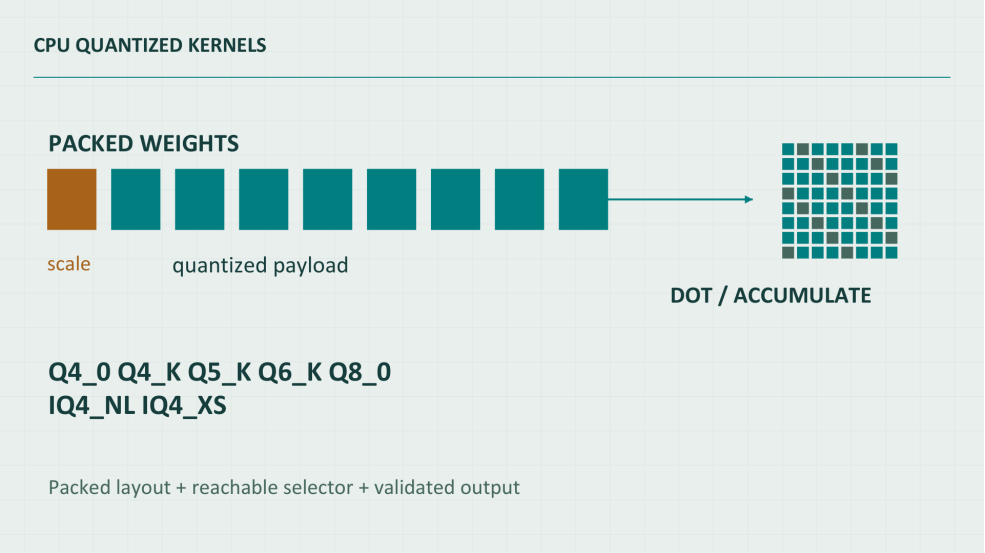

FIGURE 13

Quantized execution path. A packed layout only helps when the dispatched kernel understands it and the measurement confirms the gain.

Optimize the path that runs

The historical ledger exposed that the integer-dot family was gated behind a configuration flag that the shipped path did not enable. Promoting the reachable route changed the interpretation of earlier R8 results: a layout that looked expensive in a float fallback became useful when its intended kernel actually ran.

Subsequent work included a lane-wise Q4_K payload, a Q6_K four-row tile, smaller block storage, and prefetch where the paired measurement reproduced. Tensor tests covered formats including Q4_0, Q4_K, Q5_K, Q6_K, Q8_0, IQ4_NL, and IQ4_XS.

Correctness and reachability

Scalar, vectorized float, and quantized integer routes do not necessarily have identical numerical error. A tolerance suitable for two float implementations may be wrong for a quantized activation path. Test the intended equivalence and record which selector and format were active.

More instructions are not automatically better. The reference host favored physical-core routing over higher thread counts, and the VNNI default did not win the recorded comparison. Those observations are specific to the workload and machine rather than universal CPU rules.

A useful optimization record includes layout, active kernel, bytes moved, correctness result, and a paired throughput comparison. Without reachability, a faster kernel remains unused source code.

SOURCE NOTES R07, R08 | References on pages 35-36

14 / MEMORY AND PERFORMANCE

D3D12 execution

Direct3D 12 gives the native Windows path explicit control over compute resources and synchronization. The important unit is the full resource pipeline: a fast shader alone cannot compensate for repeated uploads, invalid resource state, an over-budget plan, or an unhandled device failure.

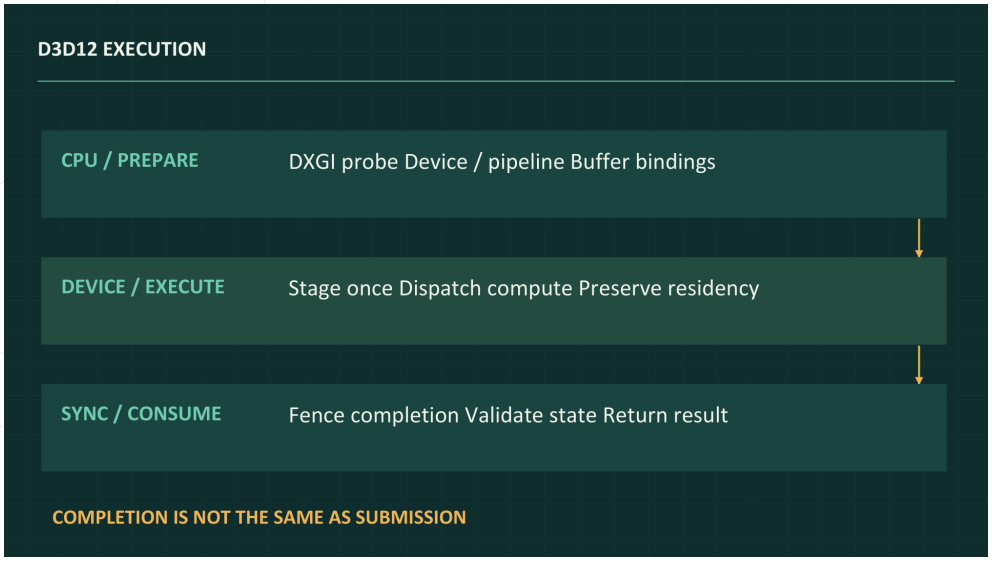

FIGURE 14

Device execution with an explicit completion boundary. Staging and synchronization must be measured alongside kernel time.

From adapter to dispatch

DXGI probing identifies the adapter and memory budget. Device creation and pipeline setup establish executable compute work. Root bindings connect buffers to shaders; command lists schedule dispatches, and fences establish when results can safely be consumed or resources reused.

The backend implements numerical operations needed by the model, including quantized matrix work, normalization, positional transforms, KV operations, and attention. Capability varies by operation and format. An available GPU does not imply that every requested quantization has a supported device path.

Measure beyond shader time

I distinguish setup, staging, queue submission, device execution, synchronization, and readback. A warm decode measurement can exclude one-time loading while a user-facing latency measurement must still explain it. Both are valid when their definitions are explicit.

D3D12 provides a Windows-native alternative to a CUDA-only implementation. That choice increases control and portability across Windows devices, but requires resource management, shader support, and debugging work that an established backend might otherwise supply.

The historical performance decision on the reference UMA host favored CPU routing. Keeping that result is essential: the purpose of an owned GPU path is to establish where it helps, not to force every supported machine through it.

SOURCE NOTES R04, R05, R07 | References on pages 35-36

15 / MEMORY AND PERFORMANCE

Residency and synchronization

A runtime can lose most of its performance outside the arithmetic operation. Weight residency, span boundaries, buffer reuse, and fence waits determine whether an apparently efficient graph repeatedly pays for data movement and coordination.

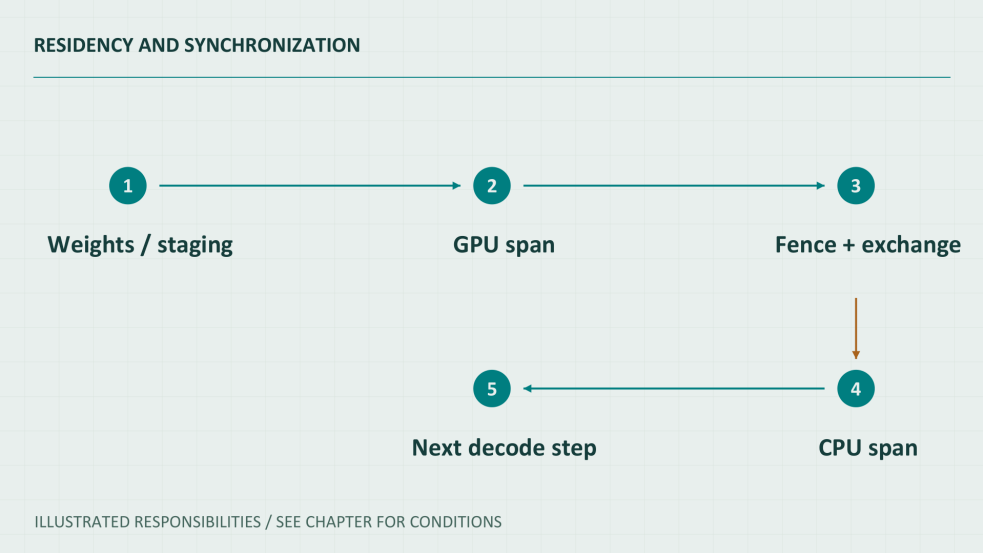

FIGURE 15

A conceptual mixed-backend boundary. The actual plan determines where transfers occur and which buffers remain resident.

Keep the working set deliberate

Resident weights stay available across decode steps. KV grows with the sequence, while activation storage can be reused according to lifetime. Staging buffers and any verifier mirror add another memory category. The fit calculation must account for each rather than comparing a file size with VRAM.

A stable CPU/GPU split makes the boundary predictable. Moving complete layer ranges is different from repeatedly evicting and reloading individual weights per token. The latter can convert an arithmetic problem into a bandwidth and synchronization bottleneck.

Make completion observable

Command submission means that work was queued, not that the result is ready. Fences and resource-state transitions protect the producer-consumer relationship. Buffer reuse must wait for the operation that owns the previous contents, including cancellation and device-error paths.

Telemetry should expose staged bytes, residency, transfers, submissions, and waits where the backend provides them. A lower submission count is only a useful optimization when it also preserves output and improves the measured request. The Ostrich example illustrates why a local count cannot replace end-to-end evidence.

The engineering objective is not to eliminate every boundary. It is to make the necessary boundaries stable, correct, and visible enough to explain their cost.

SOURCE NOTES R03, R04, R05 | References on pages 35-36

16 / MEMORY AND PERFORMANCE

KV cache: calculate the bytes

Context length creates a predictable attention-storage cost. A first estimate is 2 x layers x tokens x KV heads x head dimension x bytes per element. Packed cache formats require their actual block size, including scale metadata, rather than an idealized bit count.

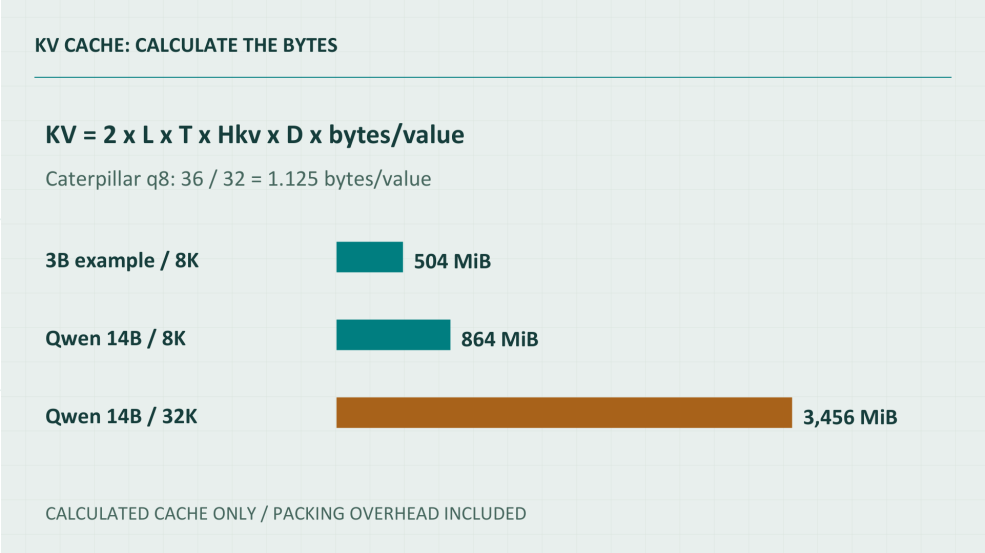

FIGURE 16

Calculated cache-only examples. q8 uses the local 36-byte block representation. Neither line represents total RAM consumption.

The implementation changes the estimate

In the inspected Caterpillar code, q8_0 stores 32 values in 36 bytes; q4_0 uses 20 bytes and q3_wht uses 16. The q8 effective cost is therefore 1.125 bytes per value. For f16 it is two bytes. Alignment and model-specific sharing can change the final allocation.

For the stated 3B example with 28 layers, eight KV heads, head dimension 128, and 8,192 positions, q8 storage is 504 MiB. That is about 528.5 decimal MB. The number is a cache calculation, not the total process or model working set.

A larger-model example

Qwen2.5-Coder-14B specifies 48 layers and eight KV heads; 5,120 hidden dimensions across 40 query heads give head dimension 128. At 8,192 positions, Caterpillar-style q8 storage is 864 MiB. At 32,768 positions it is 3,456 MiB, before weights, activations, reserve, or any mirror.

The previous 768 MiB estimate omitted the q8 block overhead. Keeping the formula and units visible makes the correction verifiable and prevents a capacity estimate from being mistaken for a measured allocation.

For shared-KV or sliding-window families, use the actual layer relationships and retained positions. A dense, all-layer formula is an example, not a universal cache model.

SOURCE NOTES R03, R04, W01 | References on pages 35-36

17 / MEMORY AND PERFORMANCE

Attention and prefix reuse

KV storage lets incremental decode reuse prior attention state instead of recomputing every earlier token. Correct reuse depends on model identity, positions, retained context, and the exact prefix. A cache hit that changes interpretation is a correctness failure, not an optimization.

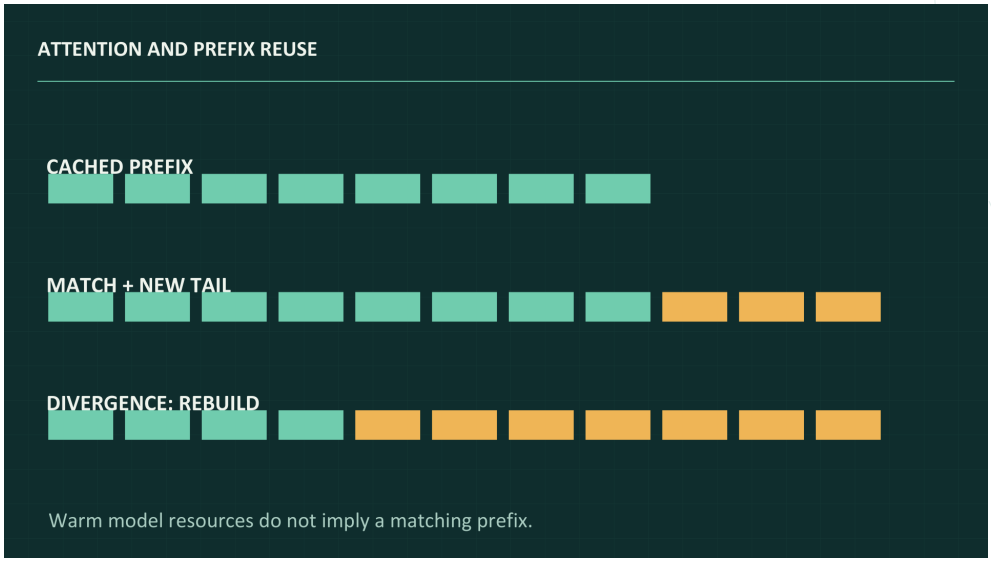

FIGURE 17

Cache reuse is conditional. A warm process and a valid prefix cache are separate states with separate tests.

How attention consumes state

Query heads attend to prior keys and values using the model family mapping. Grouped-query attention reduces the number of stored KV heads relative to query heads. Causal windows and per-layer relationships determine which history is visible to a particular operation.

Streaming online softmax can process attention without materializing the entire score matrix. Quantized cache modes trade storage against reconstruction work and numerical effects. Their value should be tested at meaningful context lengths rather than inferred from bytes saved alone.

Reuse only a verified prefix

When a prompt extends a known prefix, the runtime can ingest only its new tail. A diverged prefix requires the correct rebuild or invalidation behavior. Cross-request continuation must preserve the same token and position semantics as a single uninterrupted generation.

This creates a precise test family: matching prefix, changed prefix, context boundary, reset, split generation, and shared-KV behavior. A short successful answer is not sufficient to prove that a later continuation uses the right state.

The product should distinguish a warm model from a reusable prompt prefix. The first concerns loaded resources; the second concerns request-specific attention history. Conflating them produces misleading latency expectations and difficult state bugs.

SOURCE NOTES R03, R08 | References on pages 35-36

18 / MEMORY AND PERFORMANCE

Speculative decode

Caterpillar can draft candidate tokens from recent token history and verify them with the target model. This avoids requiring a second draft model, but its value depends on actual matches, verification cost, and the target execution route.

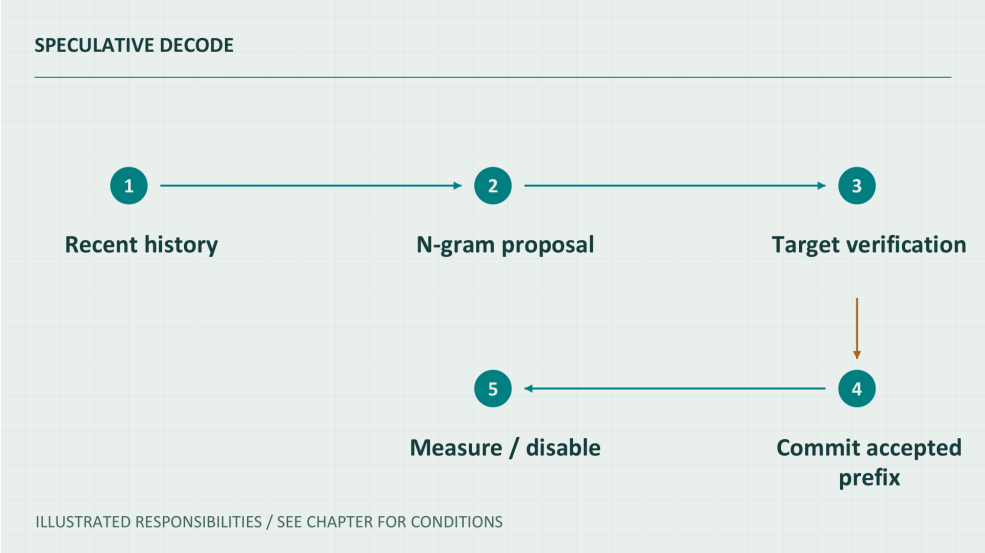

FIGURE 18

Recorded pattern test: 5.13 to 8.86 tok/s. Target verification remains authoritative; the result is not an Ollama comparison.

Draft, verify, commit

The documented n-gram method searches for the longest matching suffix and prefers a recent continuation. The target verifies a bounded proposal. Only the accepted prefix is committed to the resident state; rejected candidate rows must not become part of the next authoritative context.

The historical deterministic test proposed seven tokens and accepted all seven across two batches. Verification consumed 858.68 ms, while total generation changed from 1,947.90 to 1,128.91 ms for ten tokens. Recorded throughput rose from 5.13 to 8.86 tok/s with identical output.

The controller must measure the tradeoff

Those results describe a repetitive 3B pattern test, not arbitrary prose or larger-model performance. Acceptance is necessary but not sufficient: even a correct draft loses if verification and synchronization cost more than the target steps it replaces.

The documented auto mode requires the integer-dot verifier and compares observed verification cost per committed step with GPU replay time. A slower result disables further speculation and records a reason. Forced and disabled modes support controlled comparisons.

For a larger model, I would keep the same acceptance and latency accounting rather than scaling the old uplift by parameter count. The relevant result is verified useful tokens per elapsed second under the actual cache and execution configuration.

SOURCE NOTES R08 | References on pages 35-36

19 / PRODUCT RELIABILITY

Serving and agent integration

The local endpoint makes inference available to applications without requiring them to manage native resources directly. It is also the boundary between Lizard and an agent framework: generating tokens is not the same responsibility as planning actions or executing tools.

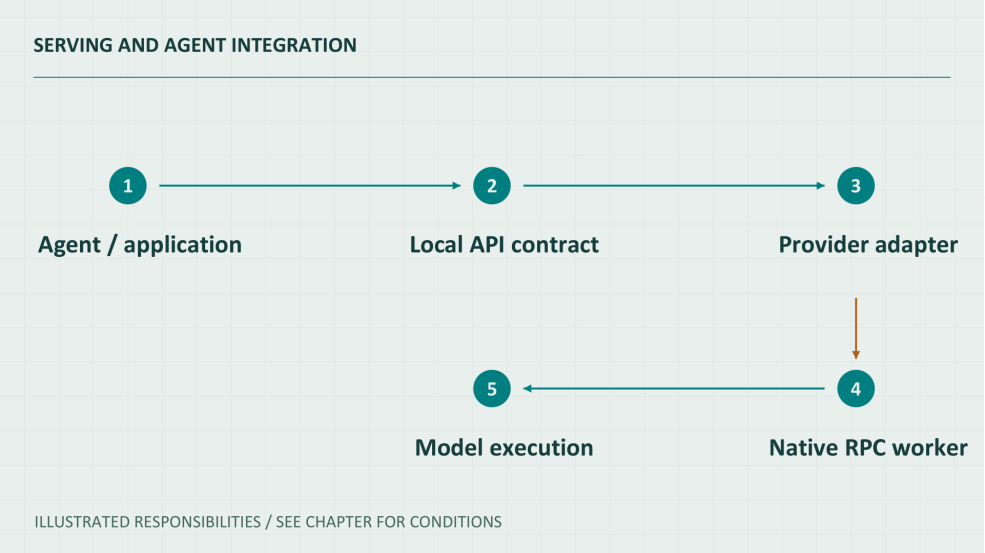

FIGURE 19

Agent orchestration is above the inference boundary. The diagram describes integration responsibility, not an embedded LangGraph implementation.

A bounded worker contract

The Caterpillar server uses a line-oriented RPC boundary with a 16 MiB input-line limit. The process owns loaded model and session state. The adapter exposes product operations such as preparation, load, generation, streaming, unload, and health.

The boundary makes process restart and failure classification possible, but it also requires careful framing and cancellation. A partial stream, malformed message, exhausted context, and worker exit are different outcomes. Each must preserve enough identity for the caller to decide what can be retried.

Where LangChain and LangGraph belong

A compatible local model endpoint can sit beneath LangChain integrations or a LangGraph application. Those layers can assemble prompts, retrieve material, maintain workflow state, and invoke tools. Lizard remains responsible for artifact compatibility, fit, execution, lifecycle, and inference telemetry.

The numerical graph inside Caterpillar is not a LangGraph agent graph. Sharing the word graph does not make their nodes or state interchangeable. One schedules tensor computation; the other can coordinate application decisions and actions.

This distinction gives a clear applied-AI presentation: I engineered the local execution substrate. Agent behavior above it must be designed and validated by the consuming application rather than implied by an endpoint label.

SOURCE NOTES R01, R06 | References on pages 35-36

20 / PRODUCT RELIABILITY

Lifecycle and recovery

A reliable local product needs distinct model and request states. Downloaded, verified, activated, loaded, warm, and stopped describe different facts. A failed request should not silently erase those distinctions or leave the interface presenting a worker that no longer exists.

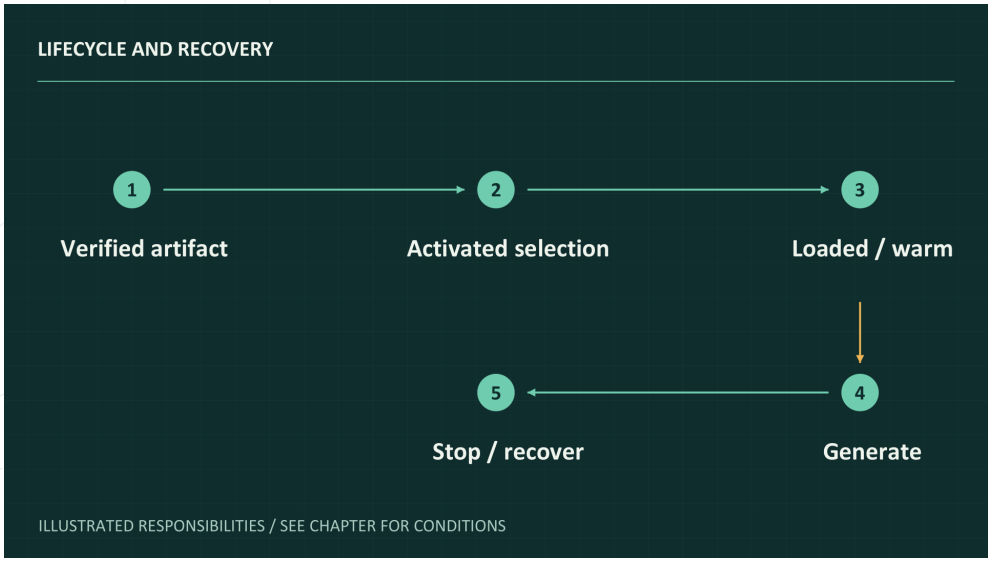

FIGURE 20

Conceptual lifecycle transitions. Selection, resource residency, and request completion are separate state dimensions.

Separate selection from execution

The provider registry defines lifecycle labels including downloaded, verified, benchmarked, activated, loaded, warm, and stopped. Activation expresses product selection; loaded and warm concern available runtime resources. A stopped process does not mean the model artifact has been deleted.

I use these distinctions to explain transitions and failure behavior. Loading can fail on compatibility or memory; generation can be interrupted after a valid load. The recovery action depends on the failed stage, not merely on whether the UI currently displays a red status.

Retry with a defined scope

An unsupported family suggests another implementation or artifact. A fit failure may justify a smaller context or different plan. A dead worker requires process recovery. A device fault may require abandoning GPU state rather than continuing with buffers whose completion is unknown.

A streamed partial answer is also an externally visible result. A retry should be identifiable as a new attempt, and the consuming application should decide whether to retain or replace the incomplete content. Quietly joining two attempts can misrepresent one model response.

The product value is predictable recovery: users retain their artifact and intent while the application explains which resource or request state needs to be rebuilt.

SOURCE NOTES R01, R06 | References on pages 35-36

21 / PRODUCT RELIABILITY

Telemetry that explains the result

A token rate becomes useful when the interface also explains what ran and under which conditions. The benchmark and runtime surfaces should connect model identity, provider, lifecycle, timing, memory, and provenance rather than presenting one isolated winner.

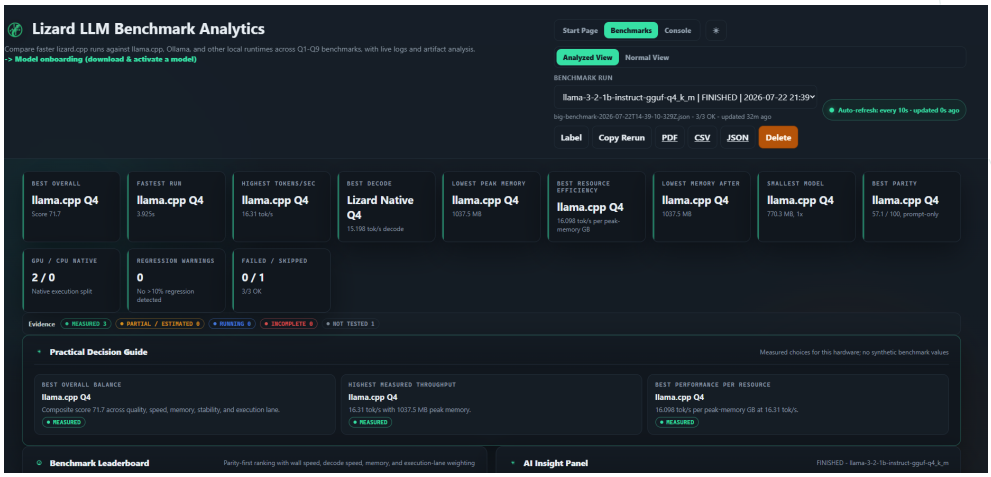

FIGURE 21

Original application benchmark surface. The image illustrates observability; the curated measurements on later pages have their own provenance.

Fields with engineering meaning

Prefill describes prompt processing; decode describes generated tokens. Load and staging can dominate a cold interaction while disappearing from a warm execution-only comparison. Total elapsed time therefore needs a defined interval instead of being treated as another name for decode latency.

Memory has similar distinctions: installed RAM, available RAM, VRAM budget, staged weights, live KV, and process working set answer different questions. A fit reason links those fields to the actual provider and plan chosen for the request.

Read the screenshot as a surface

The original benchmark screenshot is retained as product evidence. It shows how execution lanes and evidence states are exposed. Its displayed values are not combined with separate historical ledgers to manufacture a single composite run.

For an investigation, I start with artifact and runtime identity, then ask whether the measurement includes loading and how many output tokens completed. A missing value is not zero; a running row is not a final result; a fallback lane must not inherit the requested provider label.

That discipline benefits both audiences. A stakeholder can understand why an operation is slow or incomplete, while a developer can find the process, model, and timing interval that produced the observation.

SOURCE NOTES R01, R10 | References on pages 35-36

22 / PRODUCT RELIABILITY

A benchmark is a contract

A fair comparison specifies the artifact, workload, host, runtime revision, and timing definition before it ranks results. I distinguish optimization experiments, provider comparisons, and user-facing latency because they answer different questions.

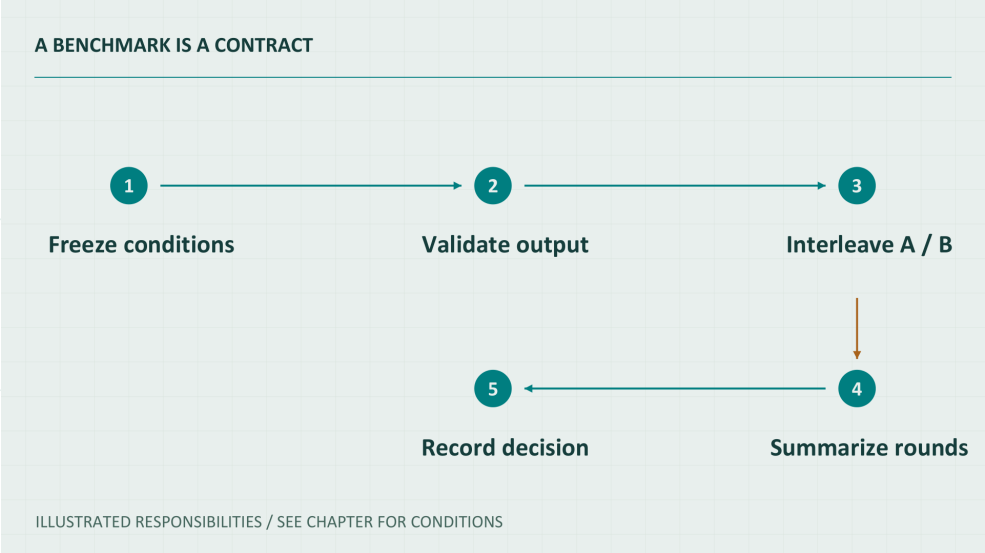

FIGURE 22

Measurement workflow. A result includes its conditions and acceptance rule, not only a tokens-per-second value.

Control the comparison

Use the same GGUF, prompt, output-token budget, context, cache format, concurrency, and warm-up policy. Record CPU threads, GPU offload, build options, driver, memory pressure, and thermal conditions. A provider name alone does not describe the executable configuration.

Interleave candidate and control runs so time-dependent drift is less likely to be mistaken for an improvement. Preserve individual rounds, summarize them with an appropriate statistic, and retain the output-quality or token-parity condition that the experiment was intended to satisfy.

Do not merge incompatible metrics

The historical reference was corrected after a cold page-cache measurement had been compared with warm candidate runs. That changed the performance conclusion. A strong portfolio keeps the corrected comparison rather than selecting the denominator that makes the candidate look fastest.

Prefill throughput, decode throughput, time to first token, and total request latency are related but not interchangeable. A short pattern test with speculation is also not a substitute for sustained generation across representative prompts.

The next pages separate local evidence from external reports and memory calculations. They can inform one engineering decision, but they must not be pooled into one provider leaderboard.

SOURCE NOTES R07, R08, R09 | References on pages 35-36

23 / EVIDENCE AND DECISIONS

Local performance evidence

These rows preserve traceable local results as separate experiments. They are not a simultaneous leaderboard. Provider identity, test intent, and comparison method remain visible so an optimization result cannot be mistaken for an unrelated Ollama or GPU measurement.

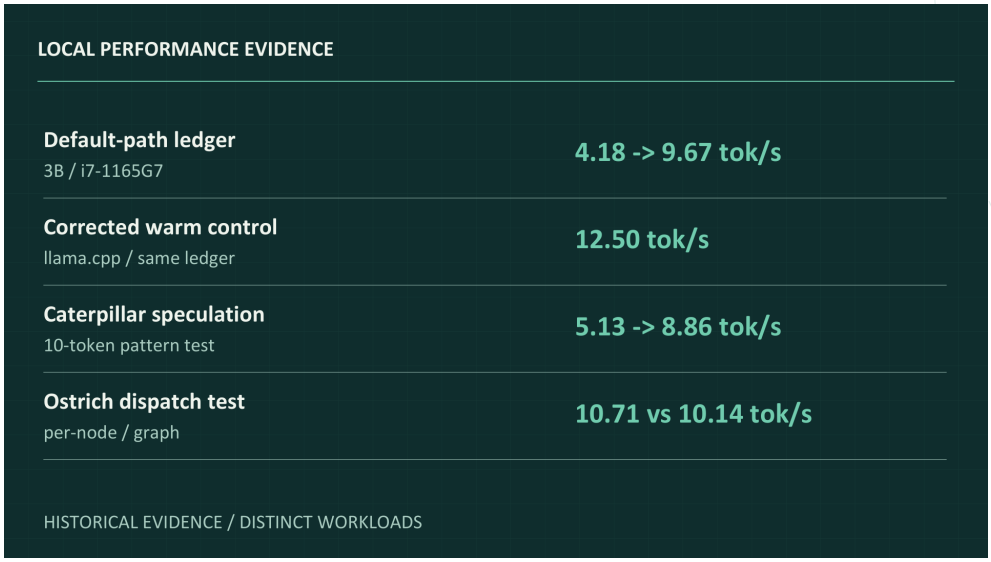

FIGURE 23

Historical local experiments. Values are retained with their own comparison and workload; no unmeasured provider cells are filled.

What the recorded rows establish

The default-path ledger records 4.18 to 9.67 tok/s for the reference 3B workload and adopts a corrected 12.50 tok/s warm llama.cpp bar. The derived improvement is 2.31 times the initial default, while the final value is about 77.4% of the corrected reference.

The deterministic speculation test is a different workload. Its 5.13 and 8.86 tok/s values compare Caterpillar without and with speculation. The original portfolio table incorrectly placed the first value under Ollama; that attribution is corrected here.

What they do not establish

The Ostrich rows compare dispatch implementations under another documented harness: 64 output tokens, 2,048 context, four threads, and four interleaved rounds. They explain a deferred scheduling decision rather than contradicting the earlier Caterpillar ledger.

These records support an engineering history on named small-model workloads. They do not prove present performance across every model or machine. No fresh inference benchmark was executed to produce this second edition.

For a new comparison, the product should expose Lizard Native, Caterpillar, Ollama, and llama.cpp separately when those lanes are actually measured. An empty comparator should never borrow a value from another experiment.

SOURCE NOTES R07, R08, R09 | References on pages 35-36

24 / EVIDENCE AND DECISIONS

The performance story

The most important optimization was making existing work reachable. The recorded progression combines selector fixes, route selection, layout changes, and targeted kernel improvements. Its final interpretation also depends on the corrected warm comparison.

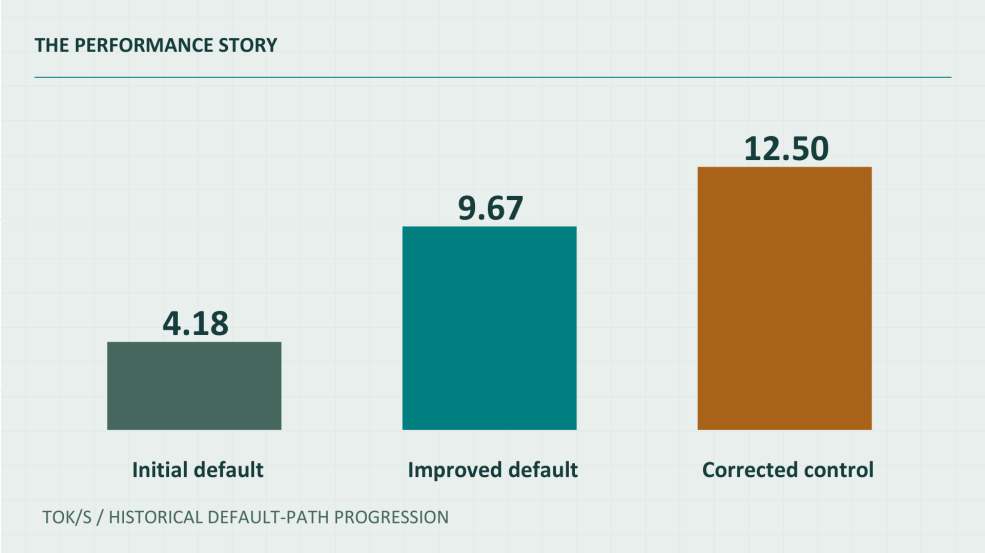

FIGURE 24

Recorded default-path endpoints and corrected warm reference. The bars summarize a historical progression, not one simultaneous three-lane run.

Follow the causal chain

The integer-dot family existed behind a flag that was not active in shipped configurations. Promoting the route changed both throughput and the interpretation of R8 layout costs. A later routing change made the default follow the useful CPU path instead of choosing a slower GPU path on the reference host.

Once the intended path ran, targeted work reduced memory traffic and accumulation overhead. The ledger records the Q6_K four-row kernel, a lane-wise Q4_K payload, and smaller headers among the retained changes. Prefetch remained conditional on reproducible benefit.

Use the profiler to choose the next step

Stage profiling attributed 91.8% of replay time to matrix multiplication. Attention, positional transforms, activation products, additions, and normalization occupied the remainder. That finding focused further optimization on the matrix path rather than on an attractive but small bookkeeping change.

The progression chart deliberately keeps the corrected llama.cpp reference visible. It does not turn the individual stage gains into an additive percentage, because the stages interact and some measurements used different thermal conditions.

The practical lesson is specific: verify reachability, measure the selected route, reduce the dominant cost, then repeat the comparison with a consistent reference. That sequence explains the result more convincingly than a claim of universal speed leadership.

SOURCE NOTES R07 | References on pages 35-36

25 / EVIDENCE AND DECISIONS

Rejected work is part of the result

A senior engineering portfolio should explain why a plausible change did not ship. The rejected and deferred experiments show how correctness, paired measurement, and maintainability constrain optimization instead of allowing every experiment to become a product claim.

FIGURE 25

Historical decisions. The result constrains a particular implementation and host, not the entire technology category.

Keep the negative result local

On the reference UMA machine, forcing GPU routing was slower than the selected CPU route. Higher thread counts also failed to beat the physical-core configuration. These are host-specific outcomes, not reasons to reject GPUs or additional threads on every system.

The VNNI default and some prefetch configurations did not clear the measured bar. A parameter-cache result initially looked promising but did not reproduce in interleaved comparisons, so its throughput claim was withdrawn. The record preserves that correction.

Turn a result into a decision

An experiment record should state the hypothesis, active path, workload, output check, raw rounds, summary, and decision. Deferred means a path may remain available behind a gate; rejected means it did not justify the proposed product change. Neither should be presented as a shipped improvement.

Ostrich provides a related example: fewer dispatches introduced enough serialization and barrier cost to lose overall. The next hypothesis follows that observation rather than repeating the claim that a graph scheduler must be faster.

This approach protects users from unstable defaults and keeps the research backlog concrete. A later engineer can reproduce the condition or design a different experiment without first reconstructing why the earlier change was abandoned.

SOURCE NOTES R07, R09 | References on pages 35-36

26 / EVIDENCE AND DECISIONS

Beyond the reference laptop

Larger memory and VRAM budgets permit larger model configurations, but they do not establish a native-engine token rate. The useful planning distinction is between artifact capacity, working-set fit, architecture support, and measured execution speed.

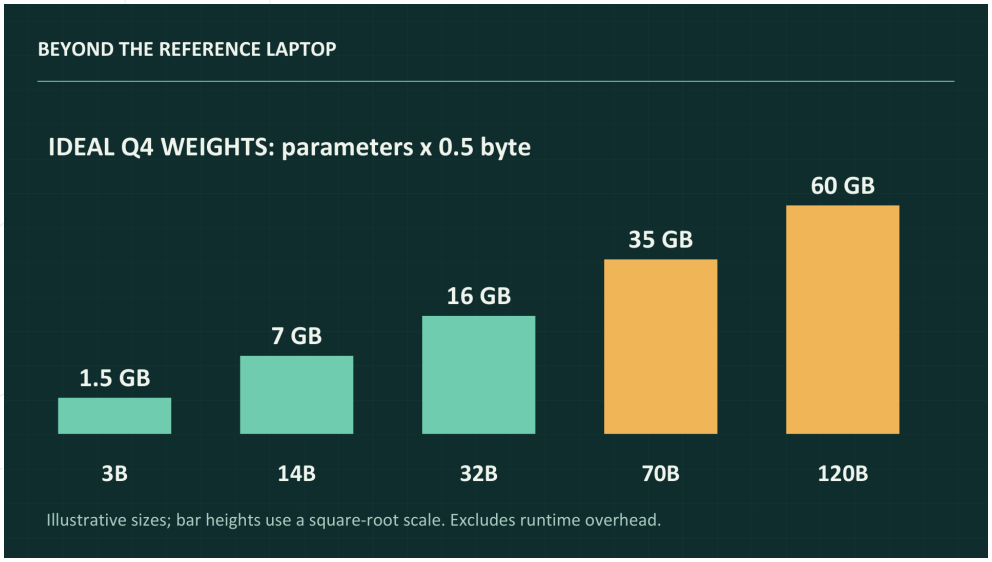

FIGURE 26

Calculated ideal Q4 payload only, in decimal GB. Real artifacts and complete runtime working sets are larger.

Size the artifact and context

An idealized four-bit weight payload is parameter count multiplied by half a byte. This gives lower-bound sizes of 1.5 GB for 3B, 7 GB for 14B, 16 GB for 32B, 35 GB for 70B, and 60 GB for 120B. Real GGUF artifacts include mixed precision, scales, and metadata.

Context adds KV and activation requirements. A machine with 64 GB system RAM does not have 64 GB VRAM, and a 32 GB card cannot devote every byte to weights. Current use, reserve, staging, and any extra model components reduce the usable budget.

Capacity is one admission gate

A dense model and a mixture-of-experts model with similar total parameter count can have very different compute requirements. Active experts affect per-token work, while the complete weight set still has to be stored or deliberately placed. Model family support remains an independent requirement.

For Lizard Native and Caterpillar, promotion should therefore name the exact GGUF and supported path rather than promise every model below a RAM threshold. The compatibility provider can broaden product coverage where it supports the artifact, but that is not native-engine validation.

The external examples on the next page show what other runtime configurations have reported at larger scales. They inform experiment design; they are not transferred into a Lizard tokens-per-second column.

SOURCE NOTES R04, W01, W02, W03 | References on pages 35-36

27 / EVIDENCE AND DECISIONS

External large-model references

Larger-model examples are useful when they retain their original runtime and hardware. This page uses first-hand published reports, clearly separated from local Lizard evidence. The values are external observations, not predictions for the owned native engines.

EXTERNAL LARGE-MODEL REFERENCES

Gemma 31B / RTX 5090 64k context / W02	**71.84 tok/s**
Gemma 31B / RTX 5090 192k context / W02	**72.09 tok/s**
Nemotron 120B / RTX 3090 128 GB RAM / W03	**16.6 tok/s**

EXTERNAL REPORTS / NOT NATIVE LIZARD MEASUREMENTS

FIGURE 27

First-hand external reports, not Lizard benchmarks. Exact sources and methodological limits are linked on the reference page.

What the external reports show

Dan Billings reports Gemma-4-31B QAT UD-Q4_K_XL on an RTX 5090 using llama.cpp, full offload, Flash Attention, and mixed q8_0/q5_1 KV. The published sweep gives 71.84 tok/s at 64k context and 72.09 tok/s at 192k. It reports one repetition, so these are not multi-round confidence estimates. [W02]

In the llama.cpp project discussion, a user reports Nemotron-3-Super-120B-A12B UD-Q3_K_XL at 16.6 tok/s with a 4,096-token depth and 128 generated tokens. The host is a Core Ultra 265K with 128 GB DDR5 and an RTX 3090; the configuration explicitly places selected tensors on CPU. [W03]

How I would use these references

The examples demonstrate different execution conditions: a large dense model on a high-VRAM card and a larger MoE with deliberate mixed placement. The external runtime, cache formats, and architecture paths differ from Caterpillar. Those differences are part of the evidence, not incidental details.

A native validation would pin an equivalent artifact and workload, establish numerical behavior, inspect actual placement, and measure prefill and decode separately. It would also test load, stop, continuation, and memory pressure before declaring a usable product configuration.

This keeps the upper-model story concrete without inventing a conversion factor from CUDA throughput to D3D12 or from an external result to a Lizard measurement.

SOURCE NOTES W02, W03 | References on pages 35-36

28 / EVIDENCE AND DECISIONS

Testing and evaluation

Each failure surface needs a focused test before a full model benchmark becomes meaningful. Numerical checks, graph replay, fit and cache contracts, provider lifecycle, and user-facing behavior answer different questions and should remain visible as distinct coverage areas.

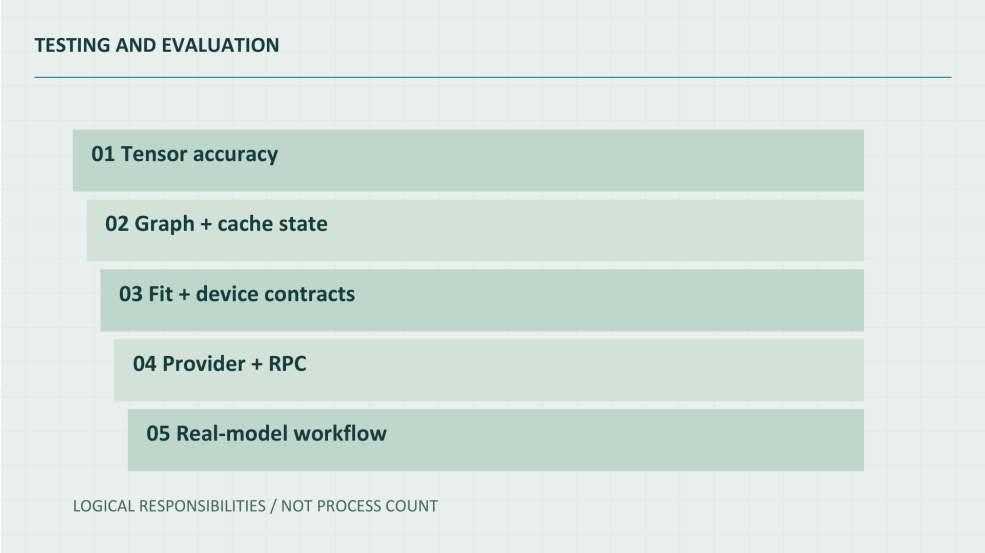

FIGURE 28

Layered testing responsibilities. Historical suite counts are evidence of a specific acceptance snapshot, not a live test dashboard.

Test the numerical and state contracts

Tensor checks compare supported scalar, vectorized, and quantized paths under the intended tolerance. Graph tests exercise topology, shape validation, lifetime reuse, and full versus incremental replay. Cache tests must include growth, prefix changes, shared-state behavior, and continuation across RPC calls.

Fit tests verify byte accounting and reasoned plan selection. Device tests compare supported operations with the reference path and exercise capability or failure boundaries. A passing CPU test does not establish the behavior of a shader path that was not executed.

Test the product around the engine

Adapter tests cover provider normalization, load and unload behavior, streaming, health, and fallback fields. System checks connect those operations to the real worker, including cancellation and recovery. A test must make clear whether it used a stub, a reference model, or actual hardware.

The historical acceptance record reports 11 of 11 native CTests and 105 of 105 focused provider tests. This edition preserves those dated counts while making no claim that the suites were rerun for the document revision.

For evaluation, pair performance with useful output and sustained behavior. A fast short completion is not enough if longer context corrupts state, memory growth exhausts the host, or the process cannot stop cleanly.

SOURCE NOTES R03, R06, R08 | References on pages 35-36

29 / EVIDENCE AND DECISIONS

The engineering delivery sequence

The development workflow can be presented as staged engineering gates. The waterfall diagram summarizes order and dependencies, while measurement and failure feed back into earlier decisions. It is not a claim that implementation proceeded without iteration.

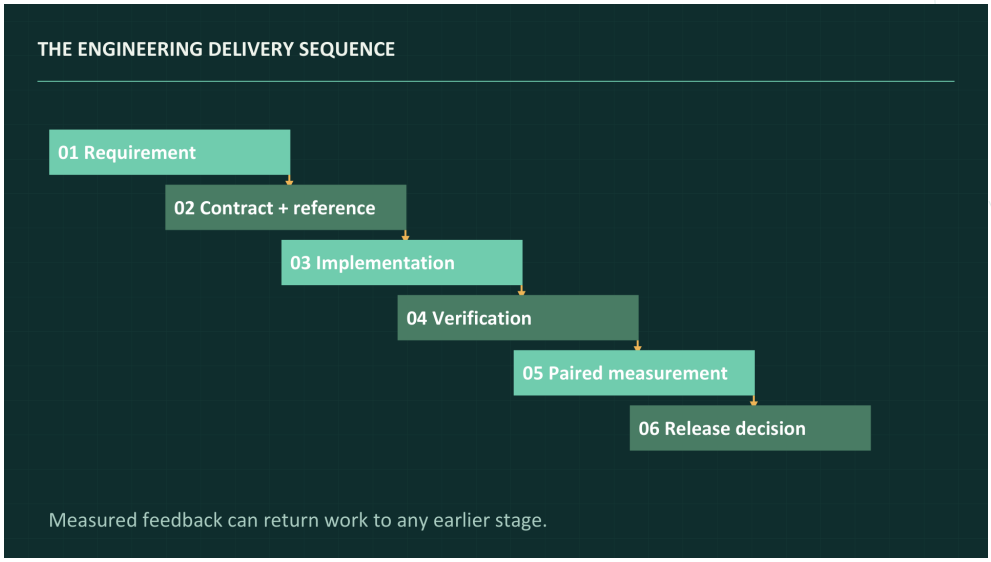

FIGURE 29

Staged delivery with feedback. Each step produces an artifact or decision that the next step can inspect.

Define before optimizing

Start with the user-facing objective and a bounded runtime contract. Establish artifact identity and a reference output path before introducing a new layout or backend. Then make resource accounting and failure behavior explicit enough to test without relying on a polished interface.

Implement a narrow executable path and verify it locally. Expand from tensor and graph checks to a real-model request. Only after that should a candidate optimization be compared under named conditions. This prevents a numerical or state bug from being celebrated as a throughput improvement.

Promote with a reason

A promotion decision combines correctness, repeatable performance, resource stability, and recovery. Some changes improve observability or memory safety without increasing tokens per second; their value should be stated accurately rather than forced into a speed claim.

When the measured result fails the gate, keep the hypothesis and conditions. The implementation may be deferred behind an experimental provider or replaced by a different approach. Stable defaults need a stronger standard than a promising isolated run.

The resulting portfolio narrative is practical: I can show what the requirement forced me to build, how I tested it, why I changed direction, and what evidence justified the final product behavior.

SOURCE NOTES R07, R08, R09 | References on pages 35-36

30 / EVIDENCE AND DECISIONS

Presenting the engineering case

The strongest presentation connects one user problem to the implementation and the evidence behind it. Lizard demonstrates applied AI through the work required to select, execute, observe, and recover a local model on real hardware.

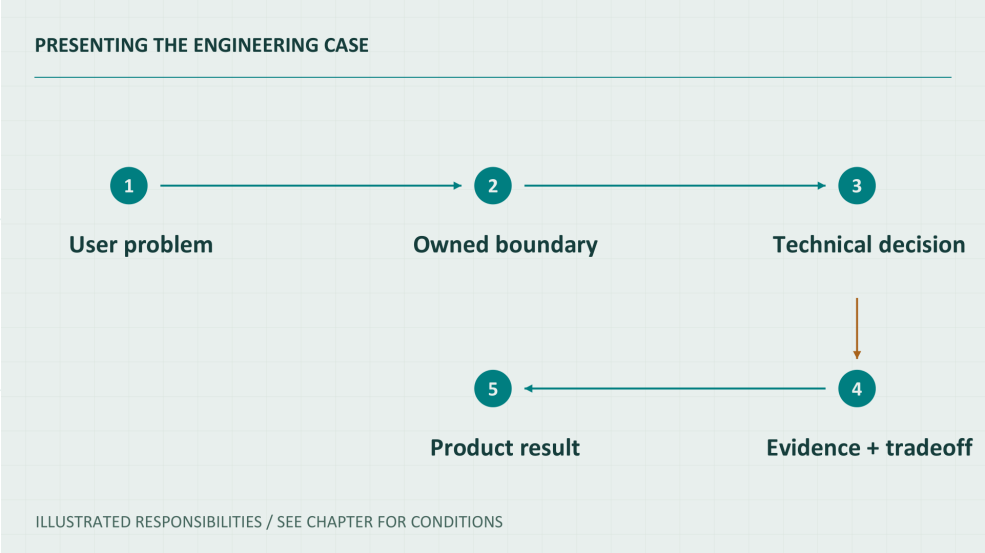

FIGURE 30

Presentation structure. Keep ownership and evidence connected so each technical choice has a visible product consequence.

A clear story for stakeholders

Begin with the user: local model execution should not require guessing whether a file fits or interpreting an unexplained runtime failure. Describe the product contract, the visible fit and lifecycle decisions, and the ability to keep working through a compatible execution route.

Then introduce the engines as deliberate boundaries. Lizard Native provides an owned native path, Caterpillar makes graph execution explicit, and Ostrich isolates deeper research. The compatibility implementation keeps coverage and comparison separate from the performance ambitions of the owned engines.

Depth for a developer audience

Choose one traceable technical example. The unreachable integer route connects configuration to layout and measured throughput. The KV calculation connects context to exact block storage. The Ostrich result connects scheduling intuition to a measured regression and a concrete next hypothesis.

Conclude with evidence and limitations together. Show the local improvement and its corrected comparison, then explain what would be required to qualify a larger model. This is more persuasive than mixing external GPU results into an unmeasured native-engine leaderboard.

The demonstrated scope spans C++ execution, Windows compute, memory planning, numerical validation, JavaScript provider integration, and product reliability. The common thread is making model capability usable and inspectable under real machine constraints.

SOURCE NOTES R01, R07, R09 | References on pages 35-36

REFERENCE

Implementation and reading notes

Source paths are relative to the Lizard repository. Inspected 12 September 2026. Code establishes implementation behavior; dated logs establish the conditions of historical measurements.

R01 / Provider policy and lifecycle

src/lib/runtime-providers.js

Provider maturity, fresh-model default, normalization fallback, lifecycle labels, and experimental access.

R02 / Native worker boundary

native/lizard-native-runner/; src/lib/runtime-providers.js

Owned native runner and product-facing provider identity. Do not conflate its measurements with Caterpillar.

R03 / Caterpillar graph and model

native/caterpillar-runner/src/{family_spec,graph,plan,gguf_loader}.{h,cpp}

Family metadata, typed dependencies, allocation, dynamic replay, and CPU/GPU span contracts.

R04 / Fit and packed KV accounting

native/caterpillar-runner/src/fit.h; kv_cache.cpp

Three fit decisions, live memory fields, host envelopes, and exact cache row sizes.

R05 / Windows compute backend

native/caterpillar-runner/src/backend_d3d12.h; gpu_model.h

Device execution interfaces, resource ownership, and GPU integration with the runtime.

R06 / Local process protocol

native/caterpillar-runner/src/server.cpp; server.h

Bounded line protocol, model/session operations, and metrics boundary. Input line limit: 16 MiB.

R07 / Historical optimization ledger

2026-08-04 iteration ledger, 06_ITERATION_LEDGER.md

Reference i7-1165G7, Llama 3.2 3B Q4_K_M. Records default 4.18 to 9.67 tok/s, corrected warm 12.50 bar, and 91.8% matmul share.

R08 / Historical acceptance record

.agent/caterpillar/04_Implementation_Handoff.md

July acceptance sections: speculative pattern test, 11/11 CTests, 105/105 provider tests. Historical, not rerun for this edition.

R09 / Ostrich experiment record

docs/ostrich/experiment-log.md

Graph dispatch comparison, workload and rounds, deferred decision, and access-policy qualifications.

R10 / Original portfolio and visuals

.agents/portfolio/Lizard_LLM_Senior_Applied_AI_Engineering_Portfolio.md; assets/

Editorial starting point and existing product imagery. Original PDF preserved; revised claims use the stronger evidence above.

REFERENCE

External sources and evidence rules

External reports and publisher metadata were checked for this edition. They inform model sizing and experiment design; they do not establish native-engine performance. Links below open the source pages.

W01 / Qwen model configuration

https://huggingface.co/Qwen/Qwen2.5-Coder-14B/blob/main/config.json

Publisher configuration: 48 layers, 40 query heads, eight KV heads, hidden size 5,120. Used only for the labeled cache calculation.

W02 / Gemma 31B: first-hand external sweep

https://www.megamote.com/blog/gemma-4-31b-q8-q5-kv-cache

Dan Billings, 28 June 2026. RTX 5090, llama.cpp, QAT UD-Q4_K_XL, full offload, Flash Attention, q8_0/q5_1 KV. Table reports one repetition; no native Lizard comparison.

W03 / Nemotron 120B: first-hand external report

https://github.com/ggml-org/llama.cpp/discussions/21112

Use discussion #21112, wbste comments of 28 March 2026. Windows build 8563 / 1f5d15e66; RTX 3090, Core Ultra 265K, 128 GB DDR5; mixed tensor placement. External user report, not independently rerun.

SOURCES AND EVIDENCE NOTES

How to interpret the numbers

Historical: recorded at the cited revision and workload. Calculated: derived from a stated formula and units. External: reported by another operator using the named hardware and runtime.

The revised benchmark chapter corrects the old 5.13 tok/s Ollama attribution. That value belongs to the Caterpillar non-speculative pattern test. Unsupported native performance targets are not filled with external scores.

Visual provenance

Cover and closing reuse the existing Lizard identity artwork. Figure 21 retains the original application benchmark screenshot. New technical plates are source-based editorial PNG renderings, not photographs or measured instrument output.

The plates simplify the cited implementation to explain one idea at a time. They are not exhaustive class diagrams or proof that every illustrated route is enabled. Read their captions and source notes together with the chapter text.

REFERENCE / 1 OF 2

Glossary

Definitions for the technical and product decisions discussed in this portfolio.

Activation arena

Reusable storage for intermediate values whose lifetimes are determined by the execution plan.

Artifact identity

The exact model file and its identifying metadata, rather than only its display name.

Backend span

A contiguous group of graph operations assigned to one execution backend.

Caterpillar

The standalone graph-plan engine used as a stable provider in the inspected registry.

D3D12 / DXGI

Windows compute and device-management interfaces used for GPU execution and memory probing.

Decode

Generating output tokens after the prompt has been processed.

Fallback

A deliberate alternative route selected with a reason when the requested path cannot be used.

FamilySpec

The model-family contract that maps metadata and per-layer behavior to a typed graph.

Fence

A synchronization value used to establish completion of queued device work.

Fit plan

A model and context placement decision based on capability and available resource budgets.

GGUF

A model container with metadata, tensor records, and weight data, including quantized formats.

GQA

Grouped-query attention: multiple query heads share a smaller number of key/value heads.

KV cache

Stored attention keys and values used to continue a sequence without rebuilding all past state.

LangChain / LangGraph

Application-level model integration and workflow tools; distinct from the numerical graph in an inference engine.

Lizard Native

The owned native worker/provider path, labeled experimental in the inspected registry.

MiB / MB

Binary mebibytes are 1,048,576 bytes; decimal megabytes are 1,000,000 bytes.

REFERENCE / 2 OF 2

Glossary

Definitions for the technical and product decisions discussed in this portfolio.

MoE

Mixture of experts: a model selects expert computation per token while retaining a larger total weight set.

Ostrich

An experimental provider used for scheduling and affinity research outside the stable baseline.

Prefill

Processing prompt tokens and constructing the state used by subsequent generation.

Provider

A selectable implementation behind the local application and runtime contract.

Quantization

Representing weights or cached values with compact numerical blocks and reconstruction rules.

Residency

Keeping resources in the memory location required for execution across repeated operations.

RoPE

Rotary positional encoding applied to query and key representations.

RPC

Remote procedure call; here, a structured request crossing the local worker-process boundary.

Speculative decode

Proposing tokens cheaply and committing only the portion validated by the target model.

Stable split

A planned GPU layer prefix and CPU remainder intended to avoid repeated weight thrashing.

Tok/s

Tokens per second; requires a definition of prompt or generation work and the timed interval.

UMA

Unified memory architecture, where CPU and GPU share physical memory and its bandwidth.

Warm state

Reusable loaded runtime resources; not automatically a valid request-specific prefix cache.

Working set

Memory needed for weights, cache, intermediates, staging, and other live execution resources.

REFERENCE

Subject index

Numbers refer to printed PDF pages, including the cover. Each number links to its own destination. The glossary is on pages 37-38.

CLOSING NOTE

From capability to a reliable product

Lizard connects a model file to an execution decision, a measured result, and a recoverable local workflow. The engineering value lies in that connection: precise identity, deliberate resource use, explicit runtime boundaries, and evidence that can change the next decision.

Gerry Walter

Senior Applied AI Engineer

A portfolio of local inference, native execution, performance engineering, and product reliability.