

Serverless LLM Platform a Technology Architecture Blueprint for Function-Style LLM Deployment

An end-to-end design for function-style LLM deployment, shared inference, isolation, elasticity, and operations

Author: **Jagpreet Singh Sasan**

Date: June 2026

Architecture thesis: Serverless at the API and accounting boundary; persistent, shared, model-aware execution cells beneath it.

Executive Summary

Developers want large language models delivered the way modern compute is delivered: deploy a function, call it, pay per use, and never think about machines. But applying the ordinary serverless mental model — a fresh process or virtual machine per invocation — to LLM inference is a category error. An LLM's weights are enormous, shared, long-lived state. Spinning up a new model server per call optimizes the cheap part (the function sandbox) while repeatedly paying the most expensive part (moving tens of gigabytes of weights into accelerator memory).

This blueprint resolves that tension with a single organizing idea: **be serverless at the API and accounting boundary, but run persistent, shared, model-aware execution cells beneath it**. Untrusted function logic runs in a small CPU sandbox that never owns an accelerator; it sends one validated, high-level generation request to a resident inference cell that multiplexes many tenants, owns the model weights and key-value (KV) memory, performs batching, and streams tokens back.

Key insights:

- **Performance is a system-level objective, not a single number.** The platform should continuously reduce invocation overhead and time-to-first-token (TTFT) while publishing realistic behavior per model, workload, and residency state.
- **Model residency is the strongest performance lever.** MicroVM readiness matters, but weight loading, queueing, prompt prefill, output length, and accelerator contention usually dominate.
- **Service classes must be explicit products.** A warm interactive tier, an elastic tier, and a batch tier should be declared and priced — not emerge as hidden behavior.
- **The request path must be short and model-aware.** The gateway routes by *predicted completion behavior*, not a generic request count.
- **The first version should be deliberately narrow.** One model family, one accelerator type, one region, declarative functions, and stateless invocations are enough to prove the core thesis.

Recommended actions: Build a workload simulator before the distributed system; prove a single-host hot inference slice; layer shared inference, the function product, multi-host placement, untrusted-code isolation, and production durability in sequence — each phase gated by measurable exit criteria.

Business impact: A residency-and-scheduling platform exposed as functions delivers higher accelerator density (thousands of function versions sharing one base-model cell), predictable interactive latency under

load, execution-accurate metering that prevents invisible losses, and a cost structure customers can reason about — turning scarce accelerator capacity into a multiplexed, measurable service.

1. Industry Context & Problem Statement

1.1 The shift to LLM inference as a utility

LLM capability is increasingly consumed as an API rather than operated as bespoke infrastructure. The market expectation, set by a decade of serverless compute, is simple: deploy a unit of logic, invoke it over HTTP or events, pay for what you use, and let the platform handle placement, scaling, and operations. Bringing that experience to LLM inference is commercially attractive but technically hard, because the economics and latency of LLM serving are governed by physics that ordinary serverless never had to confront.

1.2 Why naive serverless fails for LLMs

The defining property of an LLM endpoint is that its weights are **enormous shared state**. This breaks the assumptions behind process-per-invocation serverless:

- **Cold starts are dominated by weight movement, not sandbox boot.** Loading model weights into accelerator memory dwarfs the cost of starting a CPU sandbox. Optimizing VM start time while reloading weights every call optimizes the wrong term.
- **Requests are not comparable.** Two invocations of the same function can differ by orders of magnitude in input length, output length, and cache reuse. Counting requests tells you almost nothing about load.
- **Utilization and tail latency are in tension.** Driving accelerators toward 100% utilization causes explosive queueing and ruinous p99 latency. Interactive serving needs headroom.
- **Isolation is expensive if done at the device level.** Attaching an accelerator to a per-call VM, snapshotting device memory, and remoting every low-level operation across a VM boundary introduces lifecycle and synchronization costs that destroy the latency budget.

1.3 The gap

The result is a gap between the **developer experience** the market wants (functions, pay-per-use, no capacity planning) and the **execution reality** of LLMs (huge resident weights, shared accelerators, batch-sensitive throughput). Closing it requires neither a turnkey serverless product nor a generic model server, but a purpose-built platform whose originality lies in how functions, models, memory, scheduling, isolation, and lifecycle work together — while foundational technologies (the OS, KVM, drivers, transport security, durable storage) are deliberately reused rather than reinvented.

2. Key Insights / Findings

The architecture is grounded in an analytical model of where latency and cost actually originate. These are the findings that drive every downstream decision.

2.1 Decompose latency before optimizing it

Time-to-first-token is a sum of distinct stages, each independently optimizable:

$$\begin{aligned} \text{TTFT} \approx & \text{edge_parse} + \text{authentication} + \text{routing_and_reservation} + \text{sandbox_ready} \\ & + \text{prompt_assembly} + \text{tokenization} + \text{model_readiness} + \text{scheduler_queue} \\ & + \text{prefill} + \text{first_decode_and_sample} + \text{first_stream_flush} \end{aligned}$$

Total generation time then follows: $\text{total} \approx \text{TTFT} + (\text{output_tokens} - 1) \times \text{mean_inter_token_time}$.

The insight: **track TTFT, inter-token latency, and useful completed tokens per accelerator-second separately**. Maximizing raw tokens/second while allowing long queue waits can improve a throughput dashboard and degrade the actual product.

2.2 Residency dominates cold-start cost

Cold-start time follows the full activation path — artifact lookup, storage reads, host transfer, device transfer, allocation, initialization, verification, warmup:

$$\begin{aligned} \text{transfer_component} &= \text{packed_weight_bytes} / \text{sustained_path_bandwidth} \\ \text{activation_time} &= \text{transfer_component} + \text{allocation} + \text{initialization} + \text{verification} + \\ &\quad \text{warmup} \end{aligned}$$

Because every autoregressive token traverses every layer, demand-paging weights is rarely sufficient — tiering helps with *activation and parking*, not steady-state virtual memory for an oversized model. The design response is a **residency hierarchy plus prepositioning and warm pools**, not cleverness in the hot loop.

2.3 Memory capacity, not just compute, is a first-class limit

For a conventional attention model, KV memory per stored token is approximately:

$$\begin{aligned} \text{kv_bytes_per_token} &\approx 2 \times \text{layer_count} \times \text{kv_head_count} \times \text{head_dimension} \times \\ &\quad \text{bytes_per_KV_element} \end{aligned}$$

This makes **maximum concurrent context a hard capacity constraint even when weights fit easily**. Grouped/multi-query attention reduces `kv_head_count`; numeric compression reduces element bytes. Capacity must be modeled as curves (prefill tokens/s vs. prompt length; decode tokens/s vs. active sequences and KV length), never a single tokens/second figure.

2.4 Predict work, not request count

Routing and admission should estimate input tokens, expected output (learned per function), prefix-cache hit length, prefill and per-token cost for the model/hardware/batch state, adapter readiness, queued work, and cold-activation time. The candidate cell is chosen by an interpretable score:

$$\begin{aligned} \text{score}(c) = & \text{predicted_TTFT}(c) + \text{deadline_miss_penalty}(c) + \text{fairness_penalty}(c) \\ & + \text{failure_risk}(c) - \text{cache_locality_credit}(c) \end{aligned}$$

A simple, understandable predictor beats a clever opaque one until enough trace data exists to fit coefficients.

2.5 Five conclusions that govern the design

1. Performance is a system-level objective, reported realistically per model, workload, and residency state.
2. Model residency is one of the strongest performance controls.
3. Different service classes need different operating policies, exposed as explicit products.
4. The request path stays short and model-aware, routing by predicted completion behavior.
5. The first viable version is deliberately narrow.

3. Framework / Approach

The platform is best understood as a **model-residency and token-scheduling system exposed as functions**. This section describes its structure.

3.1 The core idea: serverless boundary above, shared cells below

Serverless behavior lives at the API and accounting boundary. Beneath it, persistent shared inference cells own the expensive state. Untrusted function logic runs in a CPU sandbox that never touches an accelerator and communicates with the inference cell only through one validated, high-level generation request.

SERVERLESS BOUNDARY — per-invocation, ephemeral

Function logic
declarative bytecode or CPU microVM

↓ *one validated GenerateRequest*

SHARED EXECUTION SUBSTRATE — persistent, multi-tenant

Inference cell
owns weights · KV memory · batching · sampling

↑ *streamed tokens flow back to the function sandbox and on to the caller*

Figure 1 — The defining split: ephemeral function context above a persistent, shared, model-aware execution cell.

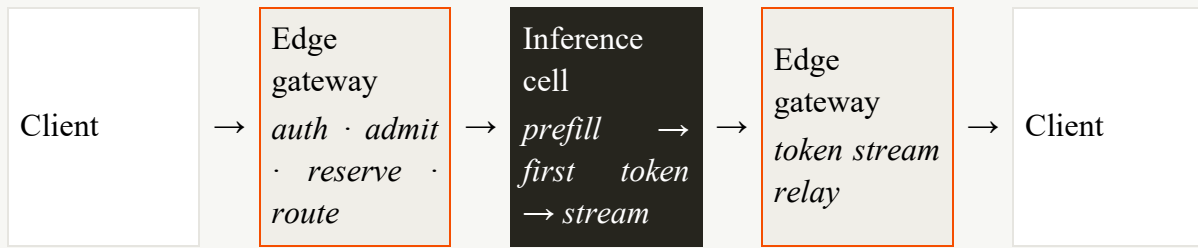
3.2 Planes and platform services

The system separates a **data plane** (regional edge gateways, node supervisors, prompt sandboxes and KVM pools, inference cells, direct streaming, local caches, metering emitters) from a **control plane** (function/model catalog, deployment compiler, immutable version and alias manager, placement/residency planner, quota/policy/key management, regional replica directory, audit and billing configuration).

A critical property: **the data plane keeps serving deployed functions even when the control plane is temporarily unavailable**, operating from signed, versioned configuration snapshots. The logical services (edged, catalogd, artifactd, placed, noded, sandboxd, inferd, meterd, telemetryd) are boundaries, not a mandatory process count — the first single-host build combines several to avoid unnecessary hops. (See Appendix C for the service reference.)

3.3 The hot invocation path

The hot path is deliberately short — no central queue, no service-mesh sidecars, no generic event bus, no control-plane lookup:



Edge reserves capacity on the selected cell (short-lived nonce) before sending the request body — preventing burst overcommitment. No central queue or service-mesh sidecar is on this path.

Figure 2 — client → edge → selected cell → edge → client. The gateway reserves capacity on a candidate cell (short-lived reservation nonce) before sending the body, preventing burst overcommitment.

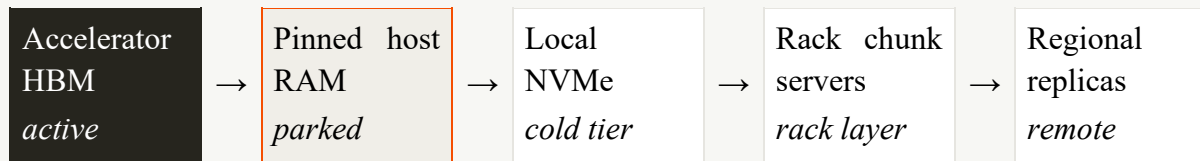
The cell checks a tenant-safe prefix cache, reserves KV pages, enters the request into its iteration scheduler, runs prefill in bounded chunks so a long prompt cannot block all decodes, flushes the first token immediately, and constrains sampling with an output-schema automaton that can stop as soon as a valid object is complete.

3.4 The inference cell

The inference cell is the heart of the platform and the largest differentiated effort. Its building blocks:

- **Offline model packer** — converts raw artifacts at registration into a canonical execution package: tensors in execution order, layer-aligned independently-checksummed chunks, accelerator-friendly tiling, kernel-selection metadata, partition plans per topology, and a deterministic known-input/known-output verification vector. A single damaged layer chunk can be refetched without recopying the whole model.
- **Fixed-graph execution runtime** — for a narrow first model family, with preallocated command/memory structures and no general heap allocation in the per-token loop. Specialized kernels are implemented only where profiling shows a meaningful advantage; generality follows measured performance.
- **Weight memory hierarchy** — immutable, content-addressed weight pages tiered across accelerator HBM → pinned host RAM → local NVMe → rack chunk servers → regional replicas, with large preallocated arenas to avoid fragmentation and atomic, checksum-gated readiness so partially loaded cells never receive traffic.
- **Paged KV memory** — each sequence owns a list of fixed-size KV pages rather than one contiguous max-length allocation, enabling incremental growth, cheap cancellation, immutable-prefix sharing, predictable quotas, and eviction/migration of cold sessions. Pages carry tenant ownership and privacy class; high-security tenants can require zeroing before reuse.
- **Prefix cache** — a radix-like token tree keyed by model + tokenizer + adapter set + exact token IDs + attention options + privacy namespace. Cross-tenant reuse is disabled by default because cache timing can leak information.
- **Deadline-aware continuous batching** — at each iteration the scheduler protects approaching inter-token deadlines, fills remaining compute with bounded prefill chunks, selects by earliest deadline adjusted for tenant fairness credits, reserves KV pages before launching, and updates service-time estimates from real timings. Its adaptive window: wait_more only if predicted_batch_saving > added_wait + deadline_risk_penalty.

3.5 The residency hierarchy and retirement



Retirement: active HBM → idle HBM → parked host RAM → local NVMe → remote chunks. Scale-up is quick and predictive; scale-down has hysteresis (a mistaken eviction pays the full reload cost).

Figure 3 — The weight residency hierarchy. Retirement walks active HBM → idle HBM → parked host RAM → local NVMe only → remote chunks only. The *instant* minimum replica stops at idle HBM; *interactive* may park under pressure; *elastic* and *batch* may reach zero. Scale-up is quick and predictive; scale-down has hysteresis because a mistaken eviction pays the full reload penalty.

3.6 KVM and isolation: where it belongs

KVM is valuable for **isolating untrusted user code**, not for pretending the shared model is ephemeral. The microVM receives only a few paravirtual devices (read-only function artifact, I/O shared-memory rings, monotonic clock, random bytes, optional declared capabilities) — no general network and no accelerator by default. The host validates every high-level inference descriptor, applies quotas, and maps only bounded buffers, keeping the device-parser attack surface tiny. Snapshots are created after the runtime initializes but before any tenant secrets or data are present, then cloned copy-on-write per invocation.

4. Use Cases / Applications

4.1 Function types

The platform offers three execution shapes in increasing order of flexibility and cost:

Function type	What it is	When to use
Declarative LLM function	Input schema, prompt program, model, decoding policy, output schema — compiled to small prompt bytecode; no guest OS on the hot path	The default and fastest product; most endpoints fit here
Sandboxed function	User code pre/post-processes around a high-level generate capability, in a networkless capability-limited CPU microVM	Custom preprocessing of untrusted logic
Dedicated inference function	A tenant receives one or more dedicated accelerator cells	Custom model code, unusual kernels, strict isolation, guaranteed capacity

Most useful endpoints — classification, extraction, rewriting, summarization, routing, and structured generation — fit the **declarative** form. Starting with arbitrary code would add an enormous security and cold-start surface before the inference system is proven.

4.2 Latency classes as products

Class	Residency promise	Queue policy	Intended use
instant	Model and function prefix kept hot; reserved headroom	Strict deadline, admission control	User-facing interactive paths
interactive	Model normally resident in region; may re-place during low demand	Short bounded queue	Chat and synchronous APIs
elastic	May park in host memory or local storage; cold start is visible	Longer queue allowed	Infrequent endpoints
batch	No warm promise; maximize goodput	Deadline / completion-window scheduling	Offline jobs

4.3 A representative function

A declarative function reads cleanly and compiles to immutable prompt bytecode, a schema-constrained sampler, pre-tokenized static segments, and a resource envelope:

```
function review_analyzer version 7 {
  trigger http POST /reviews/analyze
  class interactive
  model text_base_8b@42
  input { review: utf8(max_bytes=12000) }
  prompt {
    system "Analyze sentiment. Return only the declared record."
    user input.review
  }
  generate { max_output_tokens 160 randomness 0.1 stop_on_schema_complete true }
  output {
    sentiment: enum(positive, neutral, negative)
    confidence: float(0, 1)
    summary: utf8(max_bytes=800)
  }
  limits { request_timeout policy_default max_cost_units 300 network none }
}
```

4.4 Event-driven and scheduled invocation

Synchronous HTTP and asynchronous events need different admission semantics and are not forced through one generic queue. Events use a **partitioned append-only log**: producers append (with source, partition key, sequence, payload digest, optional not-before time), replicas persist before acknowledging, a trigger binding owns a per-partition cursor, and a dispatcher derives an idempotency key from function version plus event identity. Failures create bounded retry schedules and ultimately a named poison-event stream; ordering is guaranteed only within a partition. Scheduled triggers are durable catalog records whose single dispatch is granted by a conditional append to the metadata log, with explicit daylight-saving and missed-fire behavior. Handlers remain idempotent because a scheduled dispatch can execute more than once after failure.

5. Solution Perspective

5.1 Capabilities and differentiators

- **Shared resident model cells + declarative functions** — thousands of function versions sharing model/version batch together; static prefixes and adapters preserve customization; only lightweight invocation context is ephemeral.
- **Model-aware routing and capacity reservation** — route to minimum predicted TTFT, reserve before sending the body, reject fast with machine-readable reasons rather than leaving requests in unbounded queues.
- **Paged KV + prefix cache + continuous batching** — the density and latency engine, with deadline-aware scheduling and tenant fairness credits.
- **Adapter multiplexing** — a resident base model plus small per-function low-rank deltas in an adapter page pool; kernels apply the selected delta without building a new full model.
- **Execution-authoritative metering** — the inference cell, which knows what actually ran and emitted, is the billing authority.
- **Isolation tiers matched to risk** — from declarative bytecode (best latency) to dedicated hardware (strict isolation).

5.2 Technology boundary — reuse vs. build

The engineering effort is concentrated where it differentiates. Foundational layers are reused behind narrow interfaces and replaced only when measurements prove they block required behavior.

Reuse (foundation)	Design (differentiated platform)
Commodity servers, CPUs, NICs, NVMe, accelerators	Function description, compiler, CLI, versions, aliases, deployment
Production OS; process/memory/FS/network facilities	Invocation contract, admission, model-aware routing, streaming
KVM and OS virtualization interfaces	Model registry, execution packaging, tokenizer, runtime, sampler
Accelerator firmware, drivers, device interfaces	Accelerator memory, paged KV, prefix reuse, adapters, iteration scheduling

Reuse (foundation)	Design (differentiated platform)
HTTP compatibility, encrypted transports, reviewed crypto	Cell lifecycle, placement, warm-residency policy, scaling, forecasting
Durable storage and replicated metadata primitives	Function/inference isolation boundary; metering, quota, observability, regional policy

5.3 Isolation tiers

Tier	Mechanism	Latency / utilization	Use
Declarative	Memory-safe prompt bytecode	Best	Most LLM functions
Public code	CPU KVM microVM + high-level inference RPC	Small added latency, shared GPU	Untrusted pre/post-processing
Trusted internal	Restricted host process	Fast, weaker boundary	One organization's trusted code
Dedicated	VM or physical host owns the accelerator cell	Expensive, lower sharing	Regulated / custom workloads

5.4 Architecture families considered

Eight families were evaluated (full detail in Appendix D). In brief: full model/VM-per-invocation (A) and one-replica-per-function (B) waste the expensive resource; **shared resident cells + declarative functions (C)** attack the actual latency and density constraints and is the recommended default; **CPU microVM + shared cells (D)** is the recommended second tier for general code; **dedicated cells (F)** are a premium product; and a **serverless-native model family (H)** offers the largest long-term advantage but requires model training and evaluation, so it is pursued only after the serving core produces trustworthy workload data. **Recommended evolution: C → D, with F as a premium product and H as research.**

6. Business Impact

Note on figures: This is an architecture blueprint, not a benchmarked product report. The platform's value is expressed through the metrics it is designed to move and the cost mechanisms it builds in; concrete numbers must come from benchmark curves on the chosen hardware. The point is that **every dimension below is explicitly measured and attributable by design.**

6.1 The economics: density and multiplexing

The central commercial advantage is **multiplexing scarce accelerators**. Because thousands of function versions can share one compatible base-model cell — preserving customization through static prefixes and

adapters — the platform amortizes the dominant cost (resident weights) across many tenants and functions. One persistent replica per function, by contrast, duplicates identical base weights, wastes HBM, and fragments batching.

6.2 KPIs the platform is engineered to move

KPI	Mechanism	Direction
Time-to-first-token (p50/p95/p99)	Residency, short data path, model-aware routing, prefix reuse	↓ Lower
Useful tokens per accelerator-second	Continuous batching, paged KV, deadline-aware scheduling	↑ Higher
Cost per completed invocation	Density via shared cells, adapter multiplexing, residency value scoring	↓ Lower
Tail latency (p99) under load	Admission control, warm capacity slack, fairness credits	↓ Controlled
Rejection predictability	Fast machine-readable rejection vs. unbounded queueing	↑ Predictable
Billing accuracy	Execution-authoritative usage records, deduplicated by attempt	↑ Exact reconciliation
Accelerator utilization without SLO breach	Work-based autoscaling at a safe sub-saturation point	↑ Higher safely

6.3 Metering and pricing that prevent invisible losses

The inference cell emits an immutable usage record per attempt (input/cached/generated/emitted tokens, accelerator busy time and topology share, KV byte-time, activation attribution, sandbox CPU/memory, outcome and retry reason). Records are appended locally before acknowledgment, replicated to a usage ledger, and deduplicated by attempt ID. Public pricing can be token- and residency-based, but the platform **retains internal accelerator-time accounting** so inefficient model versions or adversarial decoding patterns never become silent losses. Possible pricing dimensions: uncached input tokens, discounted cached input tokens, output tokens, sandbox compute, reserved warm residency, dedicated capacity, and optional state/actions.

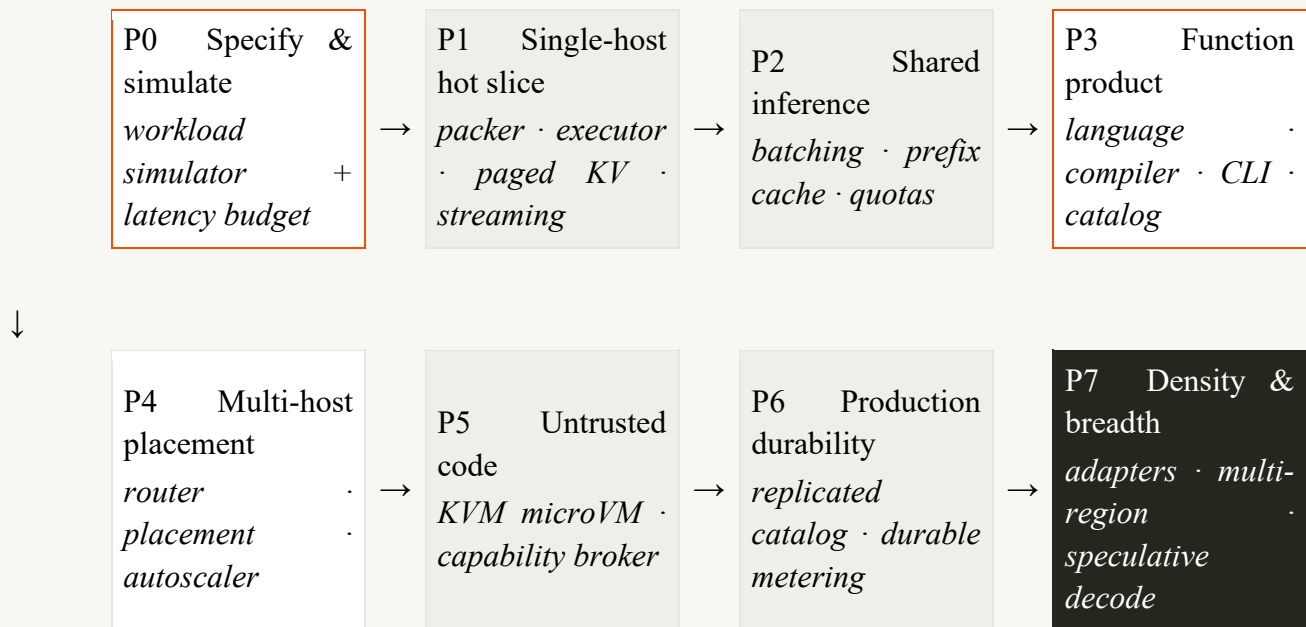
6.4 Predictable cost for customers

Explicit service classes and **warmth markets** (a tenant can purchase a residency floor or publish a warm schedule) let customers trade money for latency transparently, instead of the platform pretending all per-call

latency is equally cheap. This makes spend predictable and aligns price with the true cost driver — residency.

7. Roadmap / Recommendations

Adoption is staged, and every phase is gated by a measurable exit criterion. The guiding principle: **prove the single-host inference and scheduling thesis before building every distributed subsystem.**



Each phase ends only when its measurable exit criterion is demonstrated.

Figure 4 — The build plan. Each phase ends only when its exit criterion is demonstrated.

7.1 Quick wins — prove the core (Phases 0–2)

- **Phase 0 — Specify & simulate.** Build a discrete workload simulator (arrivals, prompt/output distributions, memory tiers, bandwidth, compute curves, cache reuse, failures) and compare residency, batching, and routing policies. *Exit: a written latency budget with evidence the proposed fleet/caching policy meets it under representative traces.*
- **Phase 1 — Single-host hot vertical slice.** One model family, one accelerator, canonical packer and tokenizer, fixed-graph executor, paged KV, single-request streaming, exact stage timings. *Exit: deterministic reference correctness, stable streaming, measured TTFT/token intervals by bucket.*
- **Phase 2 — Shared inference.** Continuous batching, chunked prefill, deadline-aware scheduler, cancellation/backpressure, prefix cache, quotas and fairness, load tests to saturation. *Exit: throughput scales with concurrency while interactive p99 stays within budget below target utilization.*

7.2 Near-term — the product and the fleet (Phases 3–4)

- **Phase 3 — Function product.** Function language, schemas, compiler, CLI, versions, aliases, deployment, declarative bytecode sandbox, immutable artifact store, single-writer catalog, structured-output automaton, developer traces, billing preview. *Exit: a user can deploy, invoke, stream, trace, promote, and roll back without touching host infrastructure.*
- **Phase 4 — Multi-host placement.** Node supervisor, replica directory, reservations, model-aware router, NVMe/host-memory caching, placement planner, work-based autoscaler, cold-start cohorting, readiness

barriers, host-failure handling. *Exit: the system survives node loss, avoids duplicate burst activations, and meets hot/warm/cold SLOs under trace replay.*

7.3 Long-term — hardening and advantage (Phases 5–7)

- **Phase 5 — Untrusted code.** Minimal KVM VMM over narrow OS primitives, tiny guest, copy-on-write snapshot pool, shared-memory rings, capability broker, syscall/device fuzzing, networkless default. *Exit: hostile guests cannot reach the host, other tenants, raw network, accelerator, or undeclared data.*
- **Phase 6 — Production durability.** Replicated catalog/metadata with verified invariants, durable artifact layer with repair, durable metering ledger, key rotation/audit/deletion/backup/recovery, failure injection at every stage. *Exit: RPO/RTO demonstrated, accounting reconciles, data-loss boundaries documented.*
- **Phase 7 — Density and breadth.** Adapter multiplexing, additional numeric versions, multi-accelerator cells, speculative decoding (by measured benefit), session affinity/KV leases, prefill/decode separation where justified, multi-region routing. *Exit: each feature improves a named customer metric without regressing tail latency or correctness.*

7.4 The smallest credible MVP

Deliberately narrower than the final vision: one region, one homogeneous accelerator fleet, one decoder-only small/medium model **always resident (no autoscaling yet)**, declarative functions only, HTTP + token streaming, schemas and constrained generation, static prompt pre-tokenization and prefix KV, continuous batching with chunked prefill, per-tenant limits, stage-level traces, and authoritative usage records. **Success criteria:** a developer deploys and invokes without seeing infrastructure; hot TTFT is dominated by model prefill, not platform overhead; concurrency improves goodput without uncontrolled p99 growth; every cancellation frees KV and stops compute; usage records reconcile exactly with traces; overload is rejected predictably. Only then add model lifecycle and true scale-from-zero — so every later cold-start mechanism can be measured against a known fast resident baseline.

Conclusion

The winning system is **not "a conventional function platform with a GPU attached."** It is a model-residency and token-scheduling platform exposed as functions. Its fast path is created by compiling most functions into safe prompt bytecode, doing static work at deployment, keeping the right weights and prefixes resident, routing straight to the cell most likely to produce the first token soonest, batching only within each request's deadline budget, isolating general user code in CPU microVMs while leaving shared inference outside, reserving resources before accepting work, and streaming immediately while accounting at the point of execution.

KVM belongs to the security architecture; shared inference cells provide the execution architecture; and scale-to-zero is a *selectable operating mode*, not a universal rule. Performance improves through sensible residency, short request paths, fair scheduling, controlled batching, and transparent service classes — never through a single absolute latency promise.

Call to action: Begin with the simulator and the single-host hot slice. Establish the latency budget, prove the inference and scheduling thesis on one model and one accelerator, and let measured workload data — not architectural ambition — pace the path to a multiplexed, multi-region, density-optimized platform.

Appendices

Appendix A — Glossary

- **TTFT (time-to-first-token)** — elapsed time from request arrival to the first streamed token.
- **Prefill** — processing the input prompt to populate KV state before decoding begins.
- **Decode** — autoregressive generation of output tokens, one iteration at a time.
- **KV cache / KV pages** — per-sequence key/value attention state; here managed as fixed-size paged allocations.
- **Prefix cache** — reusable KV for a shared leading token sequence, keyed by model, tokenizer, adapter, exact tokens, attention options, and privacy namespace.
- **Continuous batching** — forming and re-forming batches at each model iteration rather than per request.
- **Inference cell** — a persistent, shared execution unit owning weights, KV memory, batching, and sampling for a model version.
- **Adapter** — a small per-function low-rank weight delta applied over a resident base model.
- **MicroVM** — a minimal KVM virtual machine used to isolate untrusted CPU function code (no accelerator, no default network).
- **Residency** — where a model's weights currently live across the HBM → host RAM → NVMe → rack → region hierarchy.
- **Alias** — a mutable atomic pointer (e.g., production) to an immutable function/model version.
- **Goodput** — useful completed work per unit time (distinct from raw throughput).

Appendix B — Key formulas

- **Time-to-first-token:** $TTFT \approx \text{edge_parse} + \text{authentication} + \text{routing_and_reservation} + \text{sandbox_ready} + \text{prompt_assembly} + \text{tokenization} + \text{model_readiness} + \text{scheduler_queue} + \text{prefill} + \text{first_decode_and_sample} + \text{first_stream_flush}$
- **Total generation:** $\text{total} \approx TTFT + (\text{output_tokens} - 1) \times \text{mean_inter_token_time}$
- **Cold activation:** $\text{activation_time} = \text{packed_weight_bytes} / \text{sustained_path_bandwidth} + \text{allocation} + \text{initialization} + \text{verification} + \text{warmup}$
- **KV bytes per token:** $\approx 2 \times \text{layer_count} \times \text{kv_head_count} \times \text{head_dimension} \times \text{bytes_per_KV_element}$
- **Cell device-memory constraint:** $\text{weight_bytes} + \text{active_token_capacity} \times \text{kv_bytes_per_token} + \text{adapter_cache_bytes} + \text{execution_workspace_bytes} + \text{fragmentation_and_failure_margin} \leq \text{usable_device_memory}$
- **Replica provisioning:** $\text{replicas} = \text{ceil}(\text{offered_work} / \text{safe_cell_capacity}) + \text{failure_reserve}$
- **Routing score:** $\text{score}(c) = \text{predicted_TTFT}(c) + \text{deadline_miss_penalty}(c) + \text{fairness_penalty}(c) + \text{failure_risk}(c) - \text{cache_locality_credit}(c)$
- **Adaptive batching:** wait_more only if $\text{predicted_batch_saving} > \text{added_wait} + \text{deadline_risk_penalty}$
- **Quota reconciliation:** $\text{available_budget} \text{ -= worst_case_reservation at admission; available_budget += reservation - actual_charge at completion.}$

Appendix C — Platform service reference

Service	Responsibility
edged	External protocol, authentication cache, admission, model-aware routing, streaming
catalogd	Functions, models, versions, aliases, policies, schemas
artifactd	Content-addressed chunks, replication, checksums, local cache fill
placed	Replica placement, warm-residency decisions, capacity reservation
noded	Host inventory, process/VM lifecycle, memory and accelerator ownership
sandboxd	Prompt-bytecode execution and KVM snapshot cloning
inferd	Tokenization, batching, graph execution, KV memory, sampling, streaming
meterd	Durable usage records, quota reconciliation, billing aggregation
telemetryd	Trace spans, counters, logs, histograms, diagnostic queries

Appendix D — Architecture families considered

Family	Summary	Verdict
A. Full model + VM per invocation	Strong isolation, simple accounting; catastrophic weight-load latency/utilization	Viable only for a dedicated warm customer VM
B. One persistent replica per function	Easy ownership; duplicates identical base weights, wastes HBM, fragments batching	Only for truly custom full models
C. Shared resident cells + declarative functions	Functions batch together; prefixes/adapters preserve customization	Recommended default
D. CPU microVM per function + shared cells	General code with strong host isolation, accelerator still shared	Recommended second tier
E. Disaggregated accelerator service	Centralizes scarce hardware; network variance and KV movement hurt latency	Supported within a rack; prefer same-host placement
F. Dedicated accelerator	Strict isolation, custom	Premium product

Family	Summary	Verdict
VM or host	kernels; residency fee, lower multiplexing	
G. Model-provider wrapper	Fast prototype; delegates inference behavior, batching, isolation, economics	Validates the API only
H. Serverless-native model family	Shared trunk, adapter/soft-prefix compiler, locality-trained experts, early exits	Long-term research, largest upside

Appendix E — Key trade-offs and recommended choices

Decision	Recommendation
Meaning of serverless	Shared persistent inference cells; ephemeral function context
Scale to zero	Only elastic/batch; retain minimum replicas for latency SLOs
Untrusted code	Declarative bytecode first; CPU KVM for general code
GPU isolation	Shared high-level inference service; dedicated cells as premium
GPU remoting	One high-level validated generation request (not per operation)
Queue	Direct host reservation for sync traffic; durable log only for events/batch
Routing	Predicted TTFT / deadline risk with cache locality (not least-requests)
Batching	Deadline-aware adaptive iteration batching
Model loading	HBM/host/NVMe hierarchy plus predictive placement
State	Stateless default; durable state and ephemeral KV handles clearly separated
Billing	Public token/residency pricing; internal accelerator-time accounting
First implementation	One model family, one device, declarative functions, one region

Appendix F — Reliability, overload, and risks

Failure boundaries. Before output is visible, an invocation may retry on another compatible cell if deadline and idempotency policy allow; after output is visible, failure is reported as a partial stream and resumed via buffered frames or explicit continuation. Stateless handlers are at-least-once with idempotency keys; usage

accounting is eventually committed exactly once by deduplicating the immutable attempt record. Transparent retries after streaming are avoided — they create inconsistent text and duplicate effects.

Overload degradation order. (1) stop batch admissions; (2) stop elastic admissions; (3) reduce cache leases and speculative work; (4) preserve accepted interactive streams; (5) reject new interactive work quickly rather than letting requests time out.

Risks that can sink the project. Supporting every model architecture early; treating cold start as a VM problem; maximizing utilization at the expense of p99; adding arbitrary user code first; building every distributed subsystem at once; using request count as load; transparent retries after streaming; unsafe cross-tenant prefix caching; rebuilding foundational security/consensus unnecessarily; silent model fallbacks; no accounting at execution; and promising one latency number.

Appendix G — What to benchmark and break

Workload dimensions: input sizes (1 / 32 / 256 / 2K / 8K / max tokens); output sizes (1 / 32 / 256 / max); concurrency from idle to deliberate overload; arrival shapes (steady, burst, flash crowd, periodic); cache states (no hit, partial prefix, full static prefix, session reuse); residency states (HBM, host RAM, local NVMe, rack, regional cold); adapters (hot, cold, high-cardinality churn); client behaviors (fast/slow reader, disconnect, reconnect, cancel); failures (edge, host, cell, storage, catalog, network partition, full disk); and a noisy tenant mixed with many latency-sensitive tenants.

Invariants to test: no invocation exceeds declared limits; no tenant observes another's pages/prefixes/frames/identifiers; one reservation cannot be consumed twice; a cell never advertises ready with unverified weights; alias resolution is stable for an accepted invocation; visible sequence numbers never go backward or duplicate; usage attempts are neither lost nor double-billed; a control-plane outage cannot grant new privilege or quota.