CENDRYVA WHITEPAPER

Sub-5ms Inference at Scale: Why Rust Belongs in Production ML Infrastructure

Tyshawn Despenza, Cendryva · May 24, 2026 · v1.0

AUDIENCE

Manufacturing automation teams, logistics platforms, robotics teams, ML infrastructure teams, SRE leaders, edge computing teams

Abstract

Production machine learning is increasingly moving into latency-sensitive physical operations: warehouse routing during pick operations, machine-vision inspection on production lines, autonomous vehicle dispatch, robotic control loops, predictive maintenance, yard management, safety monitoring, and edge quality checks. In these environments, inference is not a batch analytics job. It is part of the product or operational control plane.

Sub-5ms inference requires more than a fast model. It requires predictable runtime behavior, low allocation overhead, tight memory control, efficient concurrency, careful model format choices, observability at the tail, and deployment patterns that keep hot paths small.

Rust is a strong fit for this layer because it combines native performance, memory efficiency, compile-time safety, concurrency primitives, and a practical systems ecosystem without requiring a garbage-collected runtime in the request path. This paper explains where Rust fits in production ML infrastructure, how ONNX-based inference can be shaped for low latency, and what operational practices are required to make sub-5ms inference credible at scale.

Executive Summary

Low-latency ML systems fail differently from offline ML systems. A model can be accurate and still unusable if inference introduces unpredictable tail latency, memory pressure, cold starts, allocation spikes, or operational fragility.

For real-time ML, teams need to optimize:

- model artifact size
- feature preparation latency
- runtime initialization

- memory allocation behavior
- concurrency and queueing
- cache locality
- network hops
- serialization overhead
- observability and tracing
- rollback and deployment safety

Rust belongs in the production inference layer because it lets teams build small, efficient, predictable services that sit close to the request path while integrating with portable model runtimes such as ONNX Runtime.

Sub-5ms inference is not a universal requirement. Many workflows can tolerate 50ms, 500ms, or asynchronous processing. But when the model participates in real-time decisions, the infrastructure must be designed for predictability, not just average speed.

Cendryva is built for teams that need this fast path to remain observable and governable. It combines low-latency inference patterns with model version traceability, decision logs, drift monitoring, threshold classification, and rollback controls so manufacturing and logistics teams can move quickly without losing operational accountability.

Why Latency Matters Beyond Speed

Latency-sensitive ML systems often operate inside larger workflows. A 5ms model call may sit inside a 40ms checkout request, a 20ms security gateway decision, a 16ms visual inspection frame budget, or a robotic control loop with strict timing constraints. The model does not get the whole latency budget.

The relevant metric is rarely average latency. Teams need to care about:

- p50 latency for normal experience
- p95 and p99 latency for tail behavior
- cold-start time
- throughput under burst
- queue wait time
- timeout rate
- feature preparation time
- end-to-end decision latency

A system with 2ms average inference and 80ms p99 latency may be worse than a system with 4ms average and 6ms p99 if the application cares about predictable response.

Industry Focus: Manufacturing, Logistics, and Edge Automation

Sub-5ms inference is most valuable when model output must be produced before a physical or operational system can continue.

Logistics and Fleet Operations

Routing decisions, ETA updates, load balancing, dock assignment, and exception detection may need to run continuously as vehicles, orders, weather, and facility conditions change. Low-latency inference keeps the optimization loop responsive.

Manufacturing and Quality Inspection

Machine-vision models may inspect products on a moving line. The inference service must keep up with frame cadence and reject or route items without slowing production.

Edge Robotics and Industrial IoT

Robots, drones, cameras, and industrial devices may require local inference because network latency, connectivity, or safety constraints make remote scoring unsuitable.

The common thread is not the vertical. It is the operational shape: a model result is needed quickly, repeatedly, and reliably.

For these teams, Cendryva's value is the combination of speed and control. A line manager, fulfillment lead, or robotics engineer needs to know which model version routed an item, flagged a defect, or changed a robot behavior, and whether that decision happened inside acceptable latency and quality bounds.

Why Rust Fits the Inference Layer

Rust is not a replacement for Python-based model development, notebook exploration, or training pipelines. It is a strong choice for the serving layer where predictability, resource control, and safety matter.

No Garbage Collector in the Hot Path

Garbage-collected languages can be highly productive and fast enough for many services, but garbage collection introduces runtime behavior that can complicate strict tail-latency targets. Rust does not require a garbage collector, giving engineers explicit control over allocation patterns and object lifetimes.

Memory and Thread Safety

Rust's ownership model and type system support memory safety and thread safety at compile time in safe Rust. For infrastructure that handles concurrent requests, shared buffers, model handles, and low-level runtime bindings, these guarantees reduce classes of production bugs.

Efficient Concurrency

Inference services often need to handle many concurrent requests while limiting contention around model sessions, CPU cores, and queues. Rust's concurrency ecosystem supports high-throughput asynchronous services, worker pools, bounded channels, and explicit backpressure.

Small, Deployable Binaries

Rust services can be packaged as compact binaries or container images, which helps reduce cold-start time, simplify deployment, and make edge distribution more practical.

Good Boundary Language

Rust works well at system boundaries: network services, native libraries, embedded environments, C APIs, and performance-sensitive data transformations. This makes it a practical bridge between model runtimes, telemetry pipelines, and application services.

ONNX as the Artifact Boundary

The serving layer should not be tightly coupled to every training framework. ONNX provides a portable model representation that can be exported from common ML workflows and executed by optimized runtimes.

In a production inference architecture, ONNX can serve as the contract between:

- training pipelines
- validation workflows
- model registry
- promotion controls
- runtime serving
- rollback procedures

ONNX Runtime provides graph optimizations and multiple execution providers. Teams can choose optimization levels, validate optimized artifacts, and align runtime configuration with target hardware. The important point is to treat the model artifact as a versioned, testable production object rather than a loose file copied from training.

Anatomy of a Low-Latency Inference Service

```
flowchart LR
  Request[Request] --> Parse[Parse and validate]
  Parse --> Feature[Feature assembly]
  Feature --> Cache[Hot feature cache]
  Cache --> Tensor[Tensor conversion]
  Tensor --> Runtime[ONNX runtime session]
  Runtime --> Policy[Policy and threshold checks]
  Policy --> Response[Response]
  Policy --> Telemetry[Metrics, traces, decision logs]
```

The fastest model runtime cannot compensate for slow feature assembly or bloated serialization. Teams should optimize the full decision path:

- parse only required fields
- avoid unbounded request payloads
- precompute or cache common features
- reuse buffers where safe
- minimize tensor conversion overhead
- keep model sessions warm
- avoid unnecessary network calls
- apply policy checks locally when possible
- emit telemetry asynchronously when safe

Tail Latency Engineering

Sub-5ms systems are usually lost at the tail, not the average. Common sources of tail latency include:

- dynamic allocation spikes
- lock contention
- thread oversubscription
- cold caches
- model session initialization
- network lookups during feature assembly
- large payload parsing
- noisy neighbors in shared infrastructure
- autoscaling lag
- synchronous telemetry export

- blocking file or secret lookups

Mitigations include:

- prewarming runtime sessions
- bounded queues and backpressure
- per-core worker design
- memory pool or buffer reuse
- strict request size limits
- local feature caches
- async telemetry export
- CPU pinning for critical workloads
- load shedding under saturation
- fallback policies for degraded dependencies

The goal is not only to make inference fast. It is to make slow behavior rare, measurable, and controlled.

Deployment Patterns

In-Process Library

The fastest path is often in-process inference embedded directly inside the application. This avoids network overhead but couples the model runtime to the application release cycle.

Best for:

- edge devices
- robotics
- embedded scoring
- ultra-low latency gateways

Tradeoff: tighter coupling and harder model rollout isolation.

Sidecar Service

A sidecar inference service runs next to the application instance. This keeps network hops local while separating model runtime concerns from the application process.

Best for:

- services that need local low-latency scoring
- polyglot applications
- controlled model rollout

Tradeoff: more deployment complexity and per-pod resource planning.

Dedicated Inference Service

A centralized inference service is easier to operate and scale independently, but adds network overhead and shared-service saturation risks.

Best for:

- moderate latency budgets
- many applications using the same model
- centralized governance

Tradeoff: request routing, multi-tenancy, and queueing must be carefully managed.

Edge Inference

Edge inference places model execution on devices, gateways, cameras, or facility-local servers.

Best for:

- intermittent connectivity
- privacy-sensitive local processing
- robotics and industrial control
- physical-world latency constraints

Tradeoff: distribution, update, hardware variation, and observability become harder.

Observability for Fast Inference

Fast systems still need deep visibility. OpenTelemetry provides a vendor-neutral framework for traces, metrics, and logs, which can help teams correlate inference behavior with upstream requests and downstream actions.

Inference observability should capture:

- model name and version
- runtime version
- request latency
- feature preparation latency
- tensor conversion latency
- inference latency
- policy check latency

- queue wait time
- cache hit rate
- timeout and fallback rate
- CPU and memory usage
- p50, p95, p99, and max latency
- decision identifiers for audit correlation

Telemetry should be designed so it does not become the bottleneck. High-cardinality labels, synchronous exporters, and excessive payload logging can destroy the latency budget.

Autoscaling and Capacity

Autoscaling inference services requires more than CPU utilization. Kubernetes Horizontal Pod Autoscaling can scale workloads based on resource metrics and custom metrics, but latency-sensitive inference often needs signals such as queue depth, request rate, concurrency, and p99 latency.

Capacity planning should include:

- warm capacity for expected bursts
- maximum queue depth
- per-model concurrency limits
- model loading time
- cold-start behavior
- hardware-specific throughput
- saturation tests
- graceful degradation policies

For strict latency budgets, waiting for autoscaling after a spike may be too slow. Teams may need prewarmed capacity, predictive scaling, edge-local inference, or admission control.

Model Optimization Workflow

A practical optimization sequence:

1. Define the end-to-end latency budget.
2. Measure baseline latency by stage.
3. Reduce feature assembly cost.
4. Export the model to ONNX.
5. Validate numerical equivalence against the source model.
6. Apply graph optimizations appropriate for target hardware.

7. Quantize or simplify only when accuracy and safety remain acceptable.
8. Benchmark with representative payloads and concurrency.
9. Measure p95 and p99 under burst.
10. Add rollback and fallback behavior.
11. Re-test after every model and runtime change.

Optimization without validation is dangerous. A faster model that changes decisions unpredictably is not an improvement.

Failure Modes

Low-latency inference services should be designed for controlled failure:

- model artifact fails validation
- runtime fails to initialize
- feature source is unavailable
- request payload is malformed
- inference exceeds timeout
- queue is saturated
- telemetry backend is unavailable
- model output violates policy guardrails
- downstream service cannot accept the decision

For each failure, teams need explicit behavior:

- reject
- retry
- fall back to rules
- use previous known-good model
- route to human review
- degrade to asynchronous processing
- shed load

The wrong answer is silent partial failure.

How Cendryva Applies This Pattern

Cendryva's inference architecture treats low-latency serving as one part of a larger operational system:

- ONNX-based model portability

- versioned model registry
- promotion and rollback controls
- Rust-oriented serving for latency-sensitive paths
- telemetry for latency, errors, and throughput
- decision logs for audit and review
- drift monitoring for post-deployment behavior
- threshold classification for operational response

This connects the fast path to governance. A system that produces an answer in 3ms but cannot explain which version produced it is not production-ready for critical operations.

Implementation Checklist

Teams building sub-5ms inference should define:

- end-to-end latency budget
- p95 and p99 targets
- model artifact format
- runtime optimization settings
- feature assembly strategy
- warmup behavior
- concurrency model
- queue limits and backpressure
- timeout and fallback behavior
- telemetry budget
- model validation and rollback path
- edge versus centralized deployment tradeoffs
- hardware-specific benchmarks
- load test and saturation test cadence

Conclusion

Sub-5ms inference is a systems problem. Model architecture matters, but so do feature pipelines, runtime behavior, memory allocation, concurrency, deployment topology, telemetry, and failure handling.

Rust is well suited for the production inference layer because it provides native performance, memory efficiency, and strong safety properties without requiring a garbage-collected runtime in the hot path.

Combined with ONNX model portability, careful observability, and disciplined rollout controls, Rust can help teams build inference systems that are fast, predictable, and governable.

The real objective is not speed for its own sake. It is real-time decision infrastructure that can be trusted when the surrounding operation cannot afford to wait.

Scope and Limitations

This is a vendor-authored paper published by Cendryva. It explains why Rust and ONNX-based serving fit the low-latency inference layer and how that pattern is reflected in Cendryva's architecture. It is not a vendor-neutral benchmark and it is not a head-to-head comparison of runtimes.

In scope. Architectural reasoning for low-latency inference, the role of Rust in the serving layer, ONNX as an artifact boundary, tail-latency engineering practices, deployment topologies (in-process, sidecar, dedicated, edge), observability for the fast path, autoscaling considerations, and failure-mode design.

Out of scope. Specific GPU kernel tuning, custom CUDA implementations, training infrastructure design, distributed training, model architecture selection, model fairness and bias audit techniques, and certification of safety-critical systems (for example ISO 26262 ASIL grades, IEC 61508 SIL grades, DO-178C avionics). Hardware procurement and accelerator selection are also out of scope.

This is not a published benchmark. The latency figures and the "sub-5ms" framing in this paper are illustrative engineering targets that motivate the architectural choices described, not measurements from a published, reproducible benchmark harness with documented hardware, model, payload, concurrency, and methodology. This is a known gap. A follow-up benchmark report with a reproducible harness (model artifacts, payload generator, hardware matrix, p50 / p95 / p99 / max reporting, and HdrHistogram-style coordinated-omission correction) is planned for a future version of this paper. Until that report is published, readers should treat all latency numbers here as targets to design toward and to validate themselves on their own hardware and workload.

Time-bounded items. Tooling, runtime versions, and Kubernetes APIs referenced here change quickly. Re-verify current behavior of ONNX Runtime, Tokio, Triton, io_uring, eBPF tooling, and HPA APIs at the time of implementation. Edge hardware capability also moves quickly.

Empirical claims. Beyond the latency-target caveat above, claims about the predictability advantages of compiled, non-GC runtimes, the value of bounded queues and warm sessions, and the cost of synchronous telemetry on the hot path are well-established in the systems literature but are presented here in summary form. The cited references provide the underlying detail.

Jurisdiction. The patterns are technology-oriented and largely jurisdiction-neutral. When inference participates in regulated decisions (for example credit, employment, healthcare, transportation safety), additional regulatory and ethical obligations apply that are out of scope for this paper.

References and Further Reading

Model serving and runtime

- ONNX project. *ONNX Specification*. https://onnx.ai/
- Microsoft. *ONNX Runtime documentation*. https://onnxruntime.ai/docs/
- NVIDIA. *Triton Inference Server architecture and documentation*. https://developer.nvidia.com/triton-inference-server

Tail latency and measurement discipline

- Dean, J. and Barroso, L. A. *The Tail at Scale*. Communications of the ACM, 56(2), 2013. https://research.google/pubs/the-tail-at-scale/
- Tene, G. *How NOT to Measure Latency (How to Measure Latency Like a Professional)*. Conference talk, multiple venues. Reference for coordinated omission and HdrHistogram methodology.

Rust performance and async ecosystem

- Rust Project. *The Rust Programming Language*. https://www.rust-lang.org/
- Rust Project. *The Rust Performance Book*. https://nnethercote.github.io/perf-book/
- Tokio Project. *Tokio asynchronous runtime documentation*. https://tokio.rs/

Linux performance and observability primitives

- Axboe, J. *Efficient IO with io_uring*. Kernel documentation and design notes. https://kernel.dk/io_uring.pdf
- Linux Kernel. *perf wiki and documentation*. https://perf.wiki.kernel.org/
- *eBPF documentation and ecosystem*. https://ebpf.io/

Deployment and scaling

- Kubernetes Project. *Horizontal Pod Autoscaling*. https://kubernetes.io/docs/concepts/workloads/autoscaling/horizontal-pod-autoscale/
- OpenTelemetry project. *OpenTelemetry specification and semantic conventions*. https://opentelemetry.io/docs/