

INFERENCE

Distributed Inference Runtime for Robotics

Technical Specification v0.1

inferential.sh

March 2026

1. Executive Summary

Inferential is a Python library that provides robotics-aware inference orchestration on top of Ray Serve. It bridges the gap between robots that need low-latency action chunks and GPU-hosted VLA (Vision-Language-Action) models managed by Ray.

The core problem: as robotics teams scale from 1 robot to 5, 10, or 50, they need intelligent scheduling of inference requests across shared GPU resources. Every team currently solves this with ad-hoc scripts that lack observability, deadline awareness, and robustness.

Inferential adds the robotics-specific layer that Ray does not provide: ZMQ transport for persistent robot connections, multimodal observation assembly with temporal alignment, action chunk lifecycle management, and deadline-aware scheduling that prevents robot starvation.

1.1 What Ray Handles

- Model hosting, GPU allocation, and replica management
- Autoscaling based on load
- Multi-node cluster management
- Request batching at the model level
- Prometheus-format metrics export

1.2 What Inferential Handles

- ZMQ transport layer for low-latency robot communication
- Observation decoding, validation, and temporal alignment
- Action chunk lifecycle (pending, issued, executing, completed, invalidated)
- Deadline-aware scheduling based on robot urgency and starvation
- Robot state tracking (connected, active, latency budgets)
- Robotics-specific metrics with in-memory aggregation and user callbacks
- Python SDK for robot integration

2. Architecture

Inferential is a thin Python layer that sits between robots (communicating via ZMQ) and Ray Serve (hosting model deployments). It runs inside the same Ray cluster, enabling zero-copy tensor passing through Ray's shared memory object store.

2.1 System Overview

The data flow is as follows: Robots send multimodal observations over ZMQ to the Inferential core. The core decodes the observation envelope, validates slots, and submits the request to the scheduler. The scheduler orders requests by urgency and starvation. The dispatcher calls Ray Serve deployment handles directly, passing numpy arrays through shared memory. Action chunks return through the same path.

2.2 Data Flow

Robot → ZMQ → [Inferential Core] → Ray Deployment Handle → [Ray Serve Model]
Robot ← ZMQ ← [Inferential Core] ← Ray Deployment Handle ← [Ray Serve Model]

2.3 Deployment Topology

All components run inside the same Python process within the Ray cluster. Ray Serve deployments run as Ray actors on GPUs. The ZMQ socket binds on a configurable port for robot connections. Metrics are collected in-memory and exposed via callbacks — users pipe them to whatever observability stack they use (Prometheus, W&B, logging, etc.).

Component	Language	Role
Inferential Core	Python (asyncio)	ZMQ transport, scheduling, response tracking, observation assembly, metrics
Ray Serve	Python	Model hosting, GPU management, replicas, batching, autoscaling
Robot SDK	Python	Client library (Connection + Model) for robot integration

3. Protocol Specification

3.1 Wire Format

Communication between robots and the core uses ZMQ DEALER/ROUTER sockets with multipart messages. The message format separates the metadata envelope (protobuf) from the tensor payload (raw bytes) to enable zero-copy handling of large payloads.

ZMQ Multipart Message (Robot to Core)

Frame	Content	Format
0	Identity (auto-managed by ROUTER)	Bytes
1	Empty delimiter	Empty bytes
2	Observation envelope	Protobuf-encoded
3	Tensor payload	Raw bytes (numpy tobytes)

ZMQ Multipart Message (Core to Robot)

Frame	Content	Format
0	Identity (routes to correct robot)	Bytes
1	Empty delimiter	Empty bytes
2	ModelResponse envelope	Protobuf-encoded
3	Action tensor payload	Raw bytes

3.2 Protobuf Schema

The protobuf envelope carries metadata for routing, scheduling, and observability. The core parses the envelope but never deserializes the tensor payload — it passes raw bytes through to Ray, where they become numpy arrays via shared memory.

Observation Message

```
message Observation {
  string robot_id = 1;
  uint64 timestamp_ns = 2;
  float urgency = 3;           // 0.0 (low) to 1.0 (critical)
  repeated Slot slots = 4;
  string model_id = 5;
  optional uint32 steps_remaining = 6; // optional hint from robot
  map<string, string> metadata = 7;
}
```

The `steps_remaining` field is optional. If the robot provides it (e.g., from a chunked model), the core uses it for more accurate scheduling. If omitted, the core infers urgency from learned cadence patterns. This makes the protocol model-agnostic.

Slot Message

```
message Slot {
  string key = 1;           // "camera/wrist", "proprioception"
  string dtype = 2;         // "float32", "uint8"
  repeated int64 shape = 3; // [256, 256, 3]
  uint64 byte_offset = 4;
  uint64 byte_length = 5;
  uint64 timestamp_ns = 6;  // per-slot for temporal alignment
  string encoding = 7;       // "raw", "jpeg"
}
```

ModelResponse Message

```
message ModelResponse {
  string robot_id = 1;
  string response_id = 2;
  uint64 timestamp_ns = 3;
  string dtype = 4;
  repeated int64 shape = 5;    // e.g. [16, 7] or [7]
  uint64 byte_offset = 6;
  uint64 byte_length = 7;
  float inference_latency_ms = 8;
  string model_id = 9;
  map<string, string> metadata = 10;
}
```

The ModelResponse is intentionally generic. It may contain an action chunk (16 steps), a single action (1 step), predicted future states from a world model, or a language response from a VLM. The core does not interpret the response — it routes it. The robot SDK interprets the semantics.

4. Core Components

4.1 Transport Layer (ZMQ)

The transport layer manages persistent ZMQ DEALER/ROUTER connections. The ROUTER socket binds on the server side, accepting connections from any number of DEALER sockets (one per robot). ZMQ automatically tracks client identities, enabling the core to route responses back to the correct robot.

- Binding address configurable via Server constructor (default: tcp://*:5555)
- Non-blocking receive with asyncio integration via pyzmq
- Robot identity tracked automatically by ZMQ ROUTER
- Supports concurrent connections from hundreds of robots
- ROUTER_HANDBOVER enabled: if a robot restarts and reconnects with the same identity, the server hands the connection to the new socket and disconnects the stale one
- Automatic reconnection on DEALER side: exponential backoff from 100ms (ZMQ_RECONNECT_IVL) to 5s max (ZMQ_RECONNECT_IVL_MAX), messages buffer locally during disconnection up to send HWM
- No explicit heartbeat protocol needed: the cadence tracker infers disconnection from absence of observations beyond the learned cadence interval

4.2 Observation Assembler

The observation assembler decodes protobuf envelopes, validates slot metadata against the actual payload size, checks temporal alignment between slots, and tracks observation freshness per robot.

- Decodes protobuf envelope (does not touch tensor payload)
- Validates payload size matches declared slot byte offsets and lengths
- Checks temporal alignment: warns if slots are misaligned beyond configurable threshold (default: 20ms)
- Computes staleness: time between observation capture and server receipt
- Tracks last-seen timestamp per robot for disconnect detection

4.3 Scheduler

The scheduler is a priority queue that orders inference requests by urgency. It is pluggable — users can select built-in strategies via config or provide custom scheduler classes via the SDK.

Built-in Strategies

Strategy	Behavior	Best For
round_robin	Fixed rotation across robots	Identical robots, identical tasks
deadline_aware	Scores by urgency, starvation, priority, queue age	Mixed workloads, varying criticality
batch_optimized	Holds requests briefly to batch by model	High throughput, many robots per model
priority_tiered	Strict priority classes, FIFO within each tier	Safety-critical robots must

Strategy	Behavior	Best For
		preempt others

Scheduler Interface

All schedulers implement a simple Python interface. Custom schedulers are registered via the SDK and selectable in the YAML config.

```
class Scheduler(ABC):
    def submit(self, request: InferenceRequest) -> None: ...
    def next_batch(self) -> list[InferenceRequest]: ...
    def tick(self) -> None: # periodic timer call
    def on_robot_connected(self, robot_id: str) -> None: ...
    def on_robot_disconnected(self, robot_id: str) -> None: ...
    def status(self) -> dict: ...
    def queue_len(self) -> int: ...
```

Custom Scheduling Policy

Users customize scheduling behavior by writing a policy function — a pure function that takes a request and returns a priority score. No base class needed, no queue management. The scheduler calls the policy for every request and orders by score.

```
from inferential import register_policy

@register_policy("my_policy")
def my_policy(request) -> float:
    score = 0.0
    score += max(0, -request.time_until_next) * 40
    score += request.urgency * 30
    score += (10 - request.priority) * 5
    return score

server.use_scheduler("deadline_aware", policy="my_policy")
```

Deadline-Aware Scoring (Default)

The default `deadline_aware` scheduler computes a composite score for each request. Higher score means higher priority. The score combines learned cadence signals with optional robot-reported hints.

Factor	Weight	Description
Cadence Urgency	45	<code>time_until_next</code> from learned cadence (negative = overdue, highest urgency)
Robot-Reported Urgency	25	Optional urgency signal from robot SDK (0.0 to 1.0)
Steps Remaining	15	Optional hint from chunked models (0 steps = critical). Ignored if not provided.

Factor	Weight	Description
Priority	10	Static priority from config (lower number = higher priority)
Queue Age	15	Time in queue relative to latency budget (prevents starvation)

For single-step models, cadence urgency dominates since the robot is always near its next expected request. For chunked models, `steps_remaining` provides a more precise signal when available. The scheduler gracefully handles any combination of provided and omitted signals.

4.4 Response Tracker

The response tracker replaces a chunk-specific manager with a model-agnostic tracking system. It monitors the lifecycle of every inference response and learns each robot's request cadence to inform scheduling decisions. This design works identically for chunked models (Diffusion Policy, ACT), single-step models (OpenVLA, RT-2), and any future model architecture.

Response States

State	Description
Pending	Inference requested, waiting for model result
Issued	Response computed and sent to robot
Consumed	Robot has requested a new inference (previous response fully used)
Invalidated	Response discarded due to observation drift or new command
Expired	Robot did not request a follow-up within expected cadence window

Cadence Learning

Instead of requiring chunk length or control frequency in configuration, the core observes each robot's natural request pattern and learns its cadence. This makes the system model-agnostic — it adapts automatically to chunked models, single-step models, or anything in between.

The cadence tracker maintains an exponentially weighted moving average (EWMA) of the interval between consecutive inference requests per robot. The EWMA smooths out jitter while adapting to genuine cadence changes (e.g., a robot switching from a chunked model to a single-step model).

```
class CadenceTracker:
    def __init__(self, alpha=0.3):
        self.alpha = alpha # EWMA smoothing factor
        self.cadence = {} # robot_id -> estimated interval (seconds)
        self.last_request = {}

    def on_request(self, robot_id: str):
        now = time.monotonic()
        if robot_id in self.last_request:
            interval = now - self.last_request[robot_id]
            if robot_id in self.cadence:
                # EWMA update
                self.cadence[robot_id] = (
                    self.alpha * interval +
                    (1 - self.alpha) * self.cadence[robot_id]
                )
            else:
                self.cadence[robot_id] = interval
        self.last_request[robot_id] = now

    def time_until_next(self, robot_id: str) -> float:
        if robot_id not in self.cadence:
            return 0.0 # unknown cadence, assume urgent
```

```
elapsed = time.monotonic() - self.last_request[robot_id]
return self.cadence[robot_id] - elapsed
```

Scheduling Integration

The scheduler uses `time_until_next` as the primary urgency signal. A negative value means the robot is overdue for inference — it should be prioritized. A large positive value means the robot has time remaining and can wait.

If the robot also provides the optional `steps_remaining` field in the observation envelope, the scheduler incorporates it as a secondary signal. This enables even more accurate scheduling for chunked models, while still working correctly for single-step models that omit the field.

Signal	Source	Priority
<code>time_until_next</code> (learned cadence)	Core (automatic)	Primary — always available, model-agnostic
<code>steps_remaining</code> (robot-reported)	Robot SDK (optional)	Secondary — more precise for chunked models
<code>urgency</code> (robot-reported)	Robot SDK (optional)	Override — robot can signal emergency conditions
<code>priority</code> (config)	YAML config	Static — safety tiers, business logic

Model-Agnostic Behavior

The response tracker and cadence learning work identically across all model types:

- Chunked models (Diffusion Policy, ACT): cadence is ~1–2 seconds. Robot may optionally report `steps_remaining` for finer scheduling.
- Single-step models (OpenVLA, RT-2): cadence is ~50–100ms. Robot requests inference every timestep. No `steps_remaining` needed.
- VLM/LLM queries: cadence is irregular. Robot queries when needed. Core detects the irregular pattern and avoids penalizing for missed cadence.
- Mixed fleet: robots using different model types share the same GPU through the same scheduler, each tracked independently by their learned cadence.
- Model switches: if a robot changes models mid-session, the cadence tracker adapts within a few requests via the EWMA update.

4.5 Dispatcher

The dispatcher takes requests from the scheduler and calls Ray Serve deployment handles. It routes by model ID — each model is a separate Ray Serve deployment. Within a model, Ray handles replica selection internally.

- Routes requests to the correct Ray Serve deployment handle based on `model_id`
- Passes numpy arrays directly through Ray's shared memory (zero-copy on same node)
- Receives model results back from Ray and packages them into `ModelResponse` envelopes
- Measures inference latency and reports to the response tracker
- Implements reactive health monitoring: tracks per-endpoint latency, avoids slow endpoints
- Optionally accepts predictive signals from backends (model swap notification, queue depth)

- Model-agnostic: dispatches identically regardless of whether the model returns action chunks, single actions, predictions, or language

5. Configuration

Inferential is a library, not a framework. It does not own configuration. All parameters are passed through Python constructors and method calls. Users who want YAML, Hydra, or any other config system wire it themselves.

5.1 Server Configuration (Python API)

```
import inferential as infr

# All configuration via constructor and methods
server = infr.Server(
    bind="tcp://*:5555",
    models=["pi0", "octo"],
)

# Scheduler selection
server.use_scheduler("deadline_aware",
    cadence_weight=45,
    urgency_weight=25,
    steps_weight=15,
    priority_weight=10,
    age_weight=15,
)

# Metrics callbacks
@server.on_metric
def handle(name, value, labels):
    wandb.log({f"{labels.get('robot', 'fleet')}/{name}": value})

# Start
import asyncio
asyncio.run(server.run())
```

5.2 Client Configuration (Python API)

```
import inferential as infr

# Connection: transport-level config
conn = infr.Connection(
    server="192.168.1.100:5555",
    robot_id="arm-1",
    reconnect_ivl_ms=100,          # Initial reconnect interval
    reconnect_max_ms=5000,       # Max reconnect backoff
)

# Model: per-model config
```

```
pi0 = conn.model("pi0",
    latency_budget_ms=30,
    priority=1,
)

vlm = conn.model("scene_vlm",
    latency_budget_ms=200,
    priority=2,
)
```

5.3 Integration with External Config Systems

Users who prefer declarative configuration use their own config system. Inferential accepts parameters, it does not read files.

```
# Example: Hydra integration (user's code, not Inferential's)
import hydra
from omegaconf import DictConfig

@hydra.main(config_path="conf", config_name="config")
def main(cfg: DictConfig):
    server = infr.Server(
        bind=cfg.transport.bind,
        models=cfg.models,
    )
    server.use_scheduler(cfg.scheduler.name, **cfg.scheduler.params)
    asyncio.run(server.run())
```

5.4 Runtime State

All runtime state is ephemeral. The library holds no persistent state. On restart, robots reconnect via ZMQ automatic reconnection, cadence trackers rebuild from observed request patterns, and the scheduler starts fresh. No state is lost because no state needed to be remembered.

6. Python SDK

The SDK separates connection management from model interaction. A Connection is one ZMQ socket per robot. A Model is a lightweight handle for a specific endpoint, created from the Connection. Multiple models share one connection. No YAML configuration — everything is Python API.

6.1 Basic Usage

```
import inferential as infr
import numpy as np

# One connection per robot
conn = infr.Connection(server="192.168.1.100:5555", robot_id="arm-1")

# Create a model handle
pi0 = conn.model("pi0", latency_budget_ms=30, priority=1)

# Send observation
pi0.observe(
    camera_wrist=np.array(...),      # (256, 256, 3) uint8
    proprioception=np.array(...),    # (7,) float32
    language="pick up the red block"
)

# Get result (non-blocking)
result = pi0.get_result()
if result is not None:
    actions = result["actions"]
    arm.execute(actions)
```

6.2 Multi-Model Usage

```
import inferential as infr

conn = infr.Connection(server="192.168.1.100:5555", robot_id="arm-1")

# Multiple models through the same connection
pi0 = conn.model("pi0", latency_budget_ms=30, priority=1)
vlm = conn.model("scene_vlm", latency_budget_ms=200, priority=2)

# Different observations to different models
vlm.observe(camera=frame, language="what's on the table?")
scene = vlm.get_result(timeout_ms=500)

pi0.observe(joints=positions, language=scene["answer"])
```

```
actions = pi0.get_result(timeout_ms=100)
```

6.3 Server Setup

```
import inferential as infr

# All Python, no YAML
server = infr.Server(bind="tcp://*:5555", models=["pi0", "octo"])
server.use_scheduler("deadline_aware")

# Metrics via callbacks
@server.on_metric
def handle(name, value, labels):
    print(f"{name}: {value} | {labels}")

# Start
import asyncio
asyncio.run(server.run())
```

6.4 Custom Policy

```
from inferential import register_policy

# A policy is a pure function: request in, score out
@register_policy("my_policy")
def my_policy(request) -> float:
    score = 0.0
    score += max(0, -request.time_until_next) * 40
    score += request.urgency * 30
    score += (10 - request.priority) * 5
    return score

# Use it
server.use_scheduler("deadline_aware", policy="my_policy")
```

7. Tensor Data Flow

A key design goal is minimal copies. Since Inferential runs inside the Ray cluster, tensors pass through shared memory from the ZMQ boundary to the model.

7.1 Copy Analysis

Step	Operation	Copies
1	Robot captures sensor data	0 (exists in memory)
2	SDK serializes to ZMQ buffer	1 (numpy.tobytes)
3	Core receives from ZMQ	1 (ZMQ buffer to numpy)
4	Core passes to Ray via deployment handle	0 (shared memory)
5	Model converts numpy to torch tensor	0 (torch.from_numpy shares memory)
6	Model runs inference on GPU	GPU transfer (unavoidable)
7	Result returns through Ray	0 (shared memory)
8	Core sends over ZMQ to robot	1 (numpy.tobytes)
9	SDK receives action chunk	1 (ZMQ buffer to numpy)

Total: 4 copies, all at the ZMQ network boundary between robot and server. Everything inside the server is zero-copy through Ray's Plasma object store.

7.2 Latency Budget

Operation	Time	Notes
ZMQ recv	~0.1ms	I/O, non-blocking
Protobuf decode	~0.01ms	Envelope only, not tensors
Scheduler	~0.01ms	Priority queue pop
Queue wait	0–50ms	Variable, depends on contention
Ray dispatch	~0.1ms	Shared memory, same node
Model inference	50–100ms	The actual model forward pass
Return path	~0.3ms	Ray return + ZMQ send
Total overhead	<1ms	Excluding queue wait and inference

8. Observability

Inferential provides a self-contained observability stack. It consumes Ray's built-in metrics (exposed as Prometheus-format text) and combines them with robotics-specific metrics. No external Prometheus or Grafana installation required.

8.1 Built-in Metrics

Metric	Source	Description
inference_latency_ms	Core	Per-robot, per-request inference time
learned_cadence_ms	Response Tracker	Learned request interval per robot
cadence_deviation	Response Tracker	How far actual interval deviates from learned cadence
time_until_next_ms	Response Tracker	Predicted time until robot needs next inference
observation_staleness_ms	Observation Assembler	Capture-to-receipt delay
queue_depth	Scheduler	Number of pending requests
scheduler_wait_ms	Scheduler	Time request spent in queue
responses_served	Response Tracker	Total responses per robot
invalidation_count	Response Tracker	Responses invalidated per robot
payload_size_bytes	Observation Assembler	Observation payload size
gpu_utilization	Ray (consumed)	GPU usage from Ray metrics
active_robots	Core	Number of connected robots

8.2 Metrics Architecture

Metrics are collected in-memory using ring buffers per metric series. The library provides two ways to access metrics: direct queries via `server.metrics` and callbacks that fire on every metric recording. Users pipe callbacks to whatever observability stack they use — Prometheus, W&B, stdout logging, or custom storage. Inferential does not ship a dashboard, database, or any external infrastructure.

8.3 Metric Callbacks

```
# Callback: fires on every metric recording
@server.on_metric
def handle(name, value, labels):
    wandb.log({f"{labels.get('robot', 'fleet')}/{name}": value})

# Direct query: get aggregated stats
stats = server.metrics.get_stats(
    "inference_latency_ms",
    labels={"robot": "arm-1"},
    window_seconds=300,
```

```
)  
print(f"p99: {stats.p99}ms, avg: {stats.avg}ms")  
  
# Snapshot: full fleet state for custom dashboards  
snapshot = server.metrics.snapshot()
```

9. Installation and Usage

9.1 Installation

```
pip install inferential
```

Dependencies: pyzmq, protobuf, numpy, ray[serve]. All installed automatically via pip.

9.2 Minimal Server

```
import inferential as infr
import asyncio

server = infr.Server(bind="tcp://*:5555", models=["pi0"])
server.use_scheduler("deadline_aware")
asyncio.run(server.run())
```

9.3 Minimal Robot Client

```
import inferential as infr
import numpy as np

conn = infr.Connection("192.168.1.100:5555", robot_id="arm-1")
pi0 = conn.model("pi0")

pi0.observe(camera=np.random.rand(256,256,3), joints=np.zeros(7))
result = pi0.get_result(timeout_ms=100)
```

9.4 Prerequisites

Inferential requires a running Ray Serve cluster with model deployments already configured. Ray setup is the user's responsibility. Inferential connects to existing Ray deployments via deployment handles.

```
# 1. Start Ray cluster
ray start --head

# 2. Deploy models to Ray Serve (user's code)
serve deploy ray_config.yaml

# 3. Run Inferential server (user's script)
python run_server.py
```

10. Roadmap

10.1 Phase 1 — Core Library (Weeks 1–3)

- ZMQ DEALER/ROUTER transport with asyncio integration
- Protobuf observation/response envelope schema (model-agnostic)
- Deadline-aware scheduler with pluggable interface and cadence-based scoring
- Response tracker with EWMA cadence learning
- Observation assembler with temporal alignment
- Ray Serve integration via deployment handles (zero-copy tensors)
- Python SDK (Robot class, observe, get_response)
- Basic YAML config parser

10.2 Phase 2 — Observability (Weeks 4–5)

- Metrics emission from core (UDP fire-and-forget)
- In-memory metrics store with ring buffers and aggregation
- Metric callbacks for user-defined observability integration
- Direct query API (get_stats, get_latest, snapshot)
- WebSocket live updates
- Custom metrics via SDK decorator pattern

10.3 Phase 3 — Production Hardening (Weeks 6–8)

- Hot-reload config on file change (SIGHUP)
- Additional scheduler strategies (round_robin, batch_optimized, priority_tiered)
- Alerting with webhook/email/Slack notifications
- Simulated multi-robot demo with benchmarks
- Documentation and landing page on inferential.sh

10.4 Future

- Optional Prometheus metrics export endpoint for users who want it
- Multi-model pipeline orchestration (VLA + VLM + LLM chains)
- Edge-server compute split
- C++ SDK for ROS 2 native integration
- Adaptive scheduler (learns from runtime performance)

11. Key Design Decisions

11.1 Python over Rust

The original design called for a Rust core. This was changed when we decided to build on top of Ray Serve. Since Ray is Python, the inference backend is Python, and the primary SDK is Python, using Rust for the core would introduce an unnecessary language boundary. The ZMQ-to-Ray bridge requires inter-process communication (Unix sockets or gRPC) with Rust, but runs zero-copy in Python. The latency overhead of the core (<1ms) does not justify Rust's performance characteristics. Python enables faster development, direct Ray integration, and a lower barrier to contribution.

11.2 Building on Ray vs From Scratch

Ray Serve already provides model hosting, GPU management, replicas, autoscaling, batching, and multi-node support. Rebuilding these capabilities would take months and produce an inferior result. Inferential's value is the robotics-specific layer that Ray does not provide. By building on Ray, the project scope reduces from a distributed systems platform to a focused library of approximately 2000–3000 lines.

11.3 ZMQ over gRPC/HTTP

Robots need persistent, low-latency connections. HTTP is request-response with connection overhead. gRPC is better but still involves HTTP/2 framing. ZMQ DEALER/ROUTER provides persistent connections, sub-millisecond latency on LAN, built-in identity tracking for routing replies, and non-blocking I/O. It is the standard transport in robotics systems for good reason.

11.4 Protobuf Envelope with Raw Tensor Payload

Serializing tensors through protobuf would add unnecessary encode/decode overhead for large payloads (images, point clouds). Instead, the protobuf envelope carries only metadata (robot ID, model ID, slot descriptions, timestamps). The tensor data travels as raw bytes in a separate ZMQ frame. The core reads the envelope for routing decisions but never touches the tensor bytes. This hybrid approach gives structured metadata where needed and zero-overhead tensor passing where performance matters.

11.5 Config as Policy, State as Ephemeral

The YAML file defines how the system should behave (policies, defaults, thresholds). The runtime state (connected robots, active chunks, queue contents) is ephemeral. On restart, robots reconnect and the system rebuilds state from live connections. This eliminates the need for persistent state in the core and avoids config-state divergence problems.

11.6 Metrics as Callbacks, Not Infrastructure

Inferential collects metrics in-memory and exposes them via callbacks and direct queries. It does not ship a dashboard, database, or metrics service. Users integrate with their own observability stack through callbacks — the library provides the data, the user decides where it goes. This keeps the library lightweight, dependency-free, and composable with any existing monitoring setup.

11.7 Open Source Strategy

Inferential is designed to be open-sourced. The code serves as both a useful tool for the robotics community and a demonstration of systems engineering capability. If the project gains traction, it can be commercialized via an open-core model (free core, paid enterprise features). If edge compute makes the core obsolete, the project transitions gracefully to a community standard. The open source option is strategic leverage, not a default.

— End of Specification —