

# INFERENTIAL

Library Architecture Document

---

Internal Design, Module Structure, Interfaces, and Data Flow

---

inferential.sh • March 2026 • v0.1

# 1. Package Structure

Inferential is a single Python package with namespaced submodules. Each submodule has a clear responsibility and minimal coupling to other modules. All inter-module communication flows through defined Protocol interfaces.

## 1.1 Directory Layout

```
inferential/
├── __init__.py          # Public API: Server, Robot, Scheduler, etc.
├── _version.py          # Version string
├── transport/
│   ├── __init__.py      # Transport protocol + factory
│   ├── zmq_transport.py  # ZMQ DEALER/ROUTER async implementation
│   └── messages.py      # IncomingObservation, OutgoingResponse dataclasses
├── observation/
│   ├── __init__.py      # ObservationAssembler
│   ├── assembler.py     # Decode, validate, track staleness
│   └── slots.py         # Slot validation and numpy reconstruction
├── scheduler/
│   ├── __init__.py      # Scheduler ABC + registry
│   ├── base.py          # Scheduler abstract base class
│   ├── round_robin.py   # Round-robin strategy
│   ├── deadline_aware.py # Cadence-based deadline scoring
│   ├── batch_optimized.py # Batching by model with flush timer
│   ├── priority_tiered.py # Strict tiers with FIFO within tier
│   └── request.py       # InferenceRequest dataclass
├── tracking/
│   ├── __init__.py      # ResponseTracker, CadenceTracker
│   ├── response.py      # Response lifecycle state machine
│   ├── cadence.py       # EWMA cadence learning
│   └── robots.py        # Per-robot state tracking
├── dispatch/
│   ├── __init__.py      # Dispatcher
│   ├── dispatcher.py    # Ray Serve deployment handle routing
│   └── health.py        # Endpoint health tracking (reactive)
├── metrics/
│   ├── __init__.py      # MetricsCollector
│   ├── collector.py     # Collects, aggregates, and exposes metrics
│   ├── store.py         # In-memory ring buffer per metric series
│   └── callbacks.py     # User-registered metric callbacks
├── config/
│   ├── __init__.py      # InferentialConfig
│   └── schema.py        # Pydantic models for YAML validation
```

```
|   └─ watcher.py           # File watcher for hot-reload
|
| └─ proto/
|   └─ inferential.proto    # Protobuf schema definition
|   └─ inferential_pb2.py   # Generated protobuf Python code
|
| └─ server.py              # Server class (main entry point)
| └─ client.py              # Robot SDK (client entry point)
```

## 1.2 Dependency Graph

Modules depend downward only. No circular dependencies. The server module orchestrates all submodules. The client module (SDK) is fully independent and only depends on transport and proto.

```
server.py
├─ config/          (reads YAML, validates, watches)
├─ transport/       (ZMQ recv/send)
├─ observation/     (decode, validate envelopes)
├─ scheduler/       (priority queue, pluggable strategies)
├─ tracking/        (response lifecycle, cadence learning)
├─ dispatch/        (Ray Serve deployment handles)
└─ metrics/         (collect, aggregate, callbacks)

client.py (Robot SDK – standalone, no server deps)
├─ transport/       (ZMQ DEALER send/recv)
└─ proto/           (protobuf encode/decode)
```

## 2. Concurrency Model

The server runs on a single Python asyncio event loop. All I/O (ZMQ, Ray calls) is non-blocking. CPU-bound work (protobuf decode, scheduler scoring) is microseconds per call and does not warrant threading or multiprocessing.

### 2.1 Event Loop Structure

```
async def run(config):
    transport = ZmqTransport(config.transport.bind)
    assembler = ObservationAssembler(config.observations)
    scheduler = create_scheduler(config.scheduling)
    tracker = ResponseTracker(config.response_tracking)
    dispatcher = Dispatcher(config.ray)
    metrics = MetricsCollector(config.metrics)

    await asyncio.gather(
        receive_loop(transport, assembler, scheduler, tracker, metrics),
        dispatch_loop(scheduler, dispatcher, tracker, transport, metrics),
        tick_loop(scheduler, tracker),
        config_watch_loop(config_path),
    )
```

### 2.2 Coroutine Responsibilities

Coroutine	Responsibility	Blocking?
receive_loop	await ZMQ recv, decode envelope, submit to scheduler	No — zmq.asyncio returns Futures
dispatch_loop	Pull from scheduler, call Ray handle, send response via ZMQ	No — Ray handles are async, ZMQ send is async
tick_loop	Periodic scheduler tick (flush batches, expire stale), cadence cleanup	No — pure CPU, microseconds
config_watch_loop	Reserved for future use: watch for runtime parameter changes	No — currently unused, included for extensibility

### 2.3 GIL Analysis

The GIL is not a concern for this workload. ZMQ recv/send releases the GIL during I/O. Ray deployment handle calls are async and release the GIL. Protobuf decode and scheduler scoring are microseconds of CPU work. The system would need hundreds of robots at very high frequency before the GIL becomes measurable. At that scale, the solution is running multiple Inferential server instances behind a load balancer, not threading.

## 2.4 ZMQ Async Integration

pyzmq 17+ supports standard asyncio event loops natively. No custom event loop replacement needed. The ZMQ ROUTER socket is created via `zmq.asyncio.Context` and all recv/send operations return standard asyncio Futures.

```
import zmq
import zmq.asyncio

ctx = zmq.asyncio.Context()
socket = ctx.socket(zmq.ROUTER)
socket.bind("tcp://*:5555")

# Non-blocking, returns Future
frames = await socket.recv_multipart()
```

## 2.5 ZMQ Reconnection and Resilience

ZMQ provides automatic reconnection at the transport level. No application-level reconnect logic is needed in either the server or the SDK. The DEALER socket (robot side) automatically retries connection with exponential backoff. Messages are buffered during disconnection up to the high water mark. When the server comes back, the connection re-establishes transparently.

### Server-Side Socket Options (ROUTER)

```
socket = ctx.socket(zmq.ROUTER)
socket.setsockopt(zmq.ROUTER_HANDOVER, 1)      # Allow reconnect with same identity
socket.setsockopt(zmq.ROUTER_MANDATORY, 1)     # Fail fast on unroutable messages
socket.setsockopt(zmq.RCVHWM, 1000)            # Receive high water mark
socket.setsockopt(zmq.SNDHWM, 1000)            # Send high water mark
socket.setsockopt(zmq.LINGER, 0)               # Don't block on shutdown
socket.bind(bind_address)
```

ROUTER\_HANDOVER is critical: when a robot restarts and reconnects with the same identity, the server hands the connection to the new socket and disconnects the stale one. Without this, the second connection is rejected.

### Client-Side Socket Options (DEALER)

```
socket = ctx.socket(zmq.DEALER)
socket.identity = robot_id.encode()
socket.setsockopt(zmq.RECONNECT_IVL, 100)      # 100ms initial reconnect interval
socket.setsockopt(zmq.RECONNECT_IVL_MAX, 5000) # 5s max backoff (exponential)
socket.setsockopt(zmq.RCVHWM, 100)            # Receive high water mark
socket.setsockopt(zmq.SNDHWM, 100)            # Send high water mark
socket.setsockopt(zmq.LINGER, 0)               # Don't block on shutdown
socket.connect(f"tcp://{server}")
```

The DEALER will automatically reconnect with exponential backoff starting at 100ms and doubling up to 5 seconds. During disconnection, outgoing messages buffer up to the send HWM. This means the robot SDK can keep calling `observe()` during a brief server outage without errors — messages queue locally and flush when the connection resumes.

### Disconnect Detection

ZMQ does not provide an explicit disconnect callback on the ROUTER side. The server cannot know directly that a robot disconnected. Instead, the cadence tracker serves as implicit heartbeat detection: if a robot stops sending observations beyond its learned cadence multiplied by the overdue threshold, the server infers disconnection and emits a disconnect event. No separate heartbeat protocol is needed — the observation stream IS the heartbeat.

```
# Cadence tracker detects disconnect
# If robot's learned cadence is 1.6s and overdue_multiplier is 1.5:
# After 2.4s of silence, robot is marked overdue
# After a configurable timeout (e.g., 10s), robot is considered disconnected

if tracker.is_overdue(robot_id):
    metrics.record("robot_overdue", 1.0, {"robot": robot_id})

if tracker.time_since_last(robot_id) > disconnect_timeout:
    emit_event("robot_disconnected", robot_id)
```

## 3. Module Interfaces (Protocols)

Each module exposes a Protocol (Python typing.Protocol) that defines its interface. This enables testing with mocks, swapping implementations, and clear contracts between modules.

### 3.1 Transport Protocol

```
class Transport(Protocol):
    async def recv(self) -> IncomingObservation: ...
    async def send(self, response: OutgoingResponse) -> None: ...
    async def close(self) -> None: ...
```

IncomingObservation and OutgoingResponse are frozen dataclasses:

```
@dataclass(frozen=True)
class IncomingObservation:
    identity: bytes          # ZMQ ROUTER identity frame
    envelope: bytes         # Protobuf-encoded Observation
    payload: bytes          # Raw tensor bytes
    received_at: float      # time.monotonic() on receipt

@dataclass(frozen=True)
class OutgoingResponse:
    identity: bytes         # Route back to correct robot
    envelope: bytes         # Protobuf-encoded ModelResponse
    payload: bytes          # Raw result tensor bytes
```

### 3.2 Observation Assembler Protocol

```
class Assembler(Protocol):
    def decode(
        self,
        envelope: bytes,
        payload: bytes,
        received_at: float,
    ) -> DecodedObservation: ...

@dataclass
class DecodedObservation:
    robot_id: str
    model_id: str
    timestamp_ns: int
    urgency: float
    steps_remaining: int | None  # Optional, from robot
    slots: list[SlotInfo]
    envelope: bytes             # Pass through unchanged
    payload: bytes              # Pass through unchanged
    received_at: float
    staleness_ms: float
```

### 3.3 Scheduler Protocol

```
class Scheduler(ABC):
    @abstractmethod
    def submit(self, request: InferenceRequest) -> None:
        """Enqueue a request. Raises QueueFullError if at capacity."""

    @abstractmethod
    def next_batch(self) -> list[InferenceRequest]:
        """Return next batch to dispatch. Empty list if none ready."""

    @abstractmethod
    def tick(self) -> None:
        """Periodic call for time-driven logic (batch flush, expiry)."""

    def on_robot_connected(self, robot_id: str) -> None:
        """Called when a new robot sends its first observation."""
        pass

    def on_robot_disconnected(self, robot_id: str) -> None:
        """Called when a robot is detected as disconnected."""
        pass

    @abstractmethod
    def status(self) -> SchedulerStatus: ...

    @abstractmethod
    def queue_len(self) -> int: ...
```

### 3.4 Response Tracker Protocol

```
class Tracker(Protocol):
    def on_request(self, robot_id: str) -> None:
        """Record that robot sent an inference request."""

    def on_response_sent(self, robot_id: str, response_id: str,
                        latency_ms: float) -> None:
        """Record that response was sent to robot."""

    def on_invalidated(self, robot_id: str, reason: str) -> None:
        """Record that a response was invalidated."""

    def time_until_next(self, robot_id: str) -> float:
        """Predicted seconds until robot needs next inference."""

    def learned_cadence(self, robot_id: str) -> float | None:
        """Learned cadence in seconds. None if insufficient data."""

    def is_overdue(self, robot_id: str) -> bool:
        """Whether robot has exceeded its expected cadence."""
```



### 3.5 Dispatcher Protocol

```
class Dispatcher(Protocol):
    async def dispatch(
        self,
        requests: list[InferenceRequest],
    ) -> list[DispatchResult]: ...

    def get_handle(self, model_id: str) -> DeploymentHandle | None: ...

    def endpoint_health(self, model_id: str) -> EndpointHealth: ...

@dataclass
class DispatchResult:
    robot_id: str
    response_id: str
    envelope: bytes          # Protobuf-encoded ModelResponse
    payload: bytes          # Raw result tensor bytes
    latency_ms: float
    success: bool
    error: str | None = None
```

### 3.6 Metrics Collector Protocol

```
class Metrics(Protocol):
    def record(self, name: str, value: float,
               labels: dict[str, str] | None = None) -> None:
        """Record a metric data point."""

    def get_latest(self, name: str,
                  labels: dict[str, str] | None = None) -> float | None:
        """Get most recent value for a metric."""

    def get_stats(self, name: str,
                  labels: dict[str, str] | None = None,
                  window_seconds: int = 300) -> MetricStats | None:
        """Get aggregated stats (avg, p50, p95, p99) over window."""

    def snapshot(self) -> MetricSnapshot:
        """Full snapshot for dashboard/export."""

    def on_metric(self, callback: Callable) -> None:
        """Register a callback invoked on every metric recording."""
```

## 4. Data Flow

This section traces a single observation from robot to model and back, identifying every function call, data transformation, and decision point.

### 4.1 Receive Path (Robot to Scheduler)

```
# 1. ZMQ recv (transport/zmq_transport.py)
frames = await socket.recv_multipart()
# frames = [identity, b'', envelope_bytes, payload_bytes]

# 2. Wrap in dataclass (transport/messages.py)
incoming = IncomingObservation(
    identity=frames[0],
    envelope=frames[2],
    payload=frames[3],
    received_at=time.monotonic(),
)

# 3. Decode envelope only (observation/assembler.py)
decoded = assembler.decode(
    incoming.envelope, incoming.payload, incoming.received_at
)
# Validates: robot_id present, payload size matches slots,
#            slot temporal alignment, computes staleness
# Raises: ObservationError on invalid data

# 4. Update cadence tracker (tracking/cadence.py)
tracker.on_request(decoded.robot_id)

# 5. Record metrics (metrics/collector.py)
metrics.record("observation_staleness_ms", decoded.staleness_ms,
               {"robot": decoded.robot_id})
metrics.record("payload_size_bytes", len(incoming.payload),
               {"robot": decoded.robot_id})

# 6. Build inference request (scheduler/request.py)
request = InferenceRequest(
    robot_id=decoded.robot_id,
    model_id=decoded.model_id,
    identity=incoming.identity,
    envelope=incoming.envelope,
    payload=incoming.payload,
    received_at=incoming.received_at,
    urgency=decoded.urgency,
    steps_remaining=decoded.steps_remaining,
    time_until_next=tracker.time_until_next(decoded.robot_id),
    latency_budget_ms=config.get_robot_budget(decoded.robot_id),
    priority=config.get_robot_priority(decoded.robot_id),
)
```

```
# 7. Submit to scheduler (scheduler/base.py)
scheduler.submit(request)
# Raises: QueueFullError if at capacity
```

## 4.2 Dispatch Path (Scheduler to Ray to Robot)

```
# 1. Pull batch from scheduler
batch = scheduler.next_batch()
# Returns [] if nothing ready (batch scheduler may hold)

# 2. Dispatch to Ray (dispatch/dispatcher.py)
results = await dispatcher.dispatch(batch)
# For each request in batch:
#   a. Get Ray deployment handle by model_id
#   b. Reconstruct numpy arrays from raw bytes + slot info
#   c. Call handle.infer.remote(observation_dict)
#       -> Zero-copy via Ray shared memory (same node)
#   d. Await result
#   e. Serialize result to protobuf envelope + raw bytes
#   f. Measure inference latency

# 3. Process results
for result in results:
    if result.success:
        # 3a. Update response tracker
        tracker.on_response_sent(
            result.robot_id, result.response_id, result.latency_ms
        )

        # 3b. Record metrics
        metrics.record("inference_latency_ms", result.latency_ms,
                       {"robot": result.robot_id,
                        "model": request.model_id})

        # 3c. Send response to robot via ZMQ
        await transport.send(OutgoingResponse(
            identity=result.identity,
            envelope=result.envelope,
            payload=result.payload,
        ))
    else:
        # 3d. Log error, record failure metric
        metrics.record("dispatch_errors", 1.0,
                       {"robot": result.robot_id,
                        "error": result.error})
```

## 5. Scheduler Internals

### 5.1 InferenceRequest Dataclass

```
@dataclass
class InferenceRequest:
    robot_id: str
    model_id: str
    identity: bytes          # ZMQ routing identity
    envelope: bytes          # Protobuf (pass-through)
    payload: bytes           # Raw tensors (pass-through)
    received_at: float       # time.monotonic()
    urgency: float           # 0.0 to 1.0 from robot
    steps_remaining: int | None # Optional hint from robot
    time_until_next: float   # From cadence tracker (seconds)
    latency_budget_ms: float # From config
    priority: int            # From config (lower = higher)
    score: float = 0.0       # Computed by scheduler
```

### 5.2 Deadline-Aware Scheduler

The default scheduler computes a composite score and uses a max-heap (heapq with negated scores). Higher score means serve first.

```
class DeadlineAwareScheduler(Scheduler):
    def __init__(self, config: SchedulingConfig):
        self.queue: list[InferenceRequest] = [] # heapq
        self.weights = config.deadline_aware
        self.max_queue = config.max_queue_size

    def submit(self, request: InferenceRequest) -> None:
        if len(self.queue) >= self.max_queue:
            raise QueueFullError(request.robot_id)
        request.score = self._compute_score(request)
        heapq.heappush(self.queue, request) # __lt__ uses -score

    def _compute_score(self, r: InferenceRequest) -> float:
        s = 0.0

        # Cadence urgency: negative time_until_next = overdue
        cadence_urgency = max(0, -r.time_until_next) / 0.1 # normalized
        s += cadence_urgency * self.weights.cadence_weight

        # Robot-reported urgency
        s += r.urgency * self.weights.urgency_weight

        # Steps remaining (if provided)
        if r.steps_remaining is not None:
            if r.steps_remaining == 0:
                s += self.weights.steps_weight
```

```
        elif r.steps_remaining <= 2:
            s += self.weights.steps_weight * 0.7
        elif r.steps_remaining <= 4:
            s += self.weights.steps_weight * 0.4

    # Static priority
    s += (10 - min(r.priority, 10)) * self.weights.priority_weight / 10

    # Queue age (anti-starvation)
    age_ms = (time.monotonic() - r.received_at) * 1000
    s += min(age_ms / r.latency_budget_ms, 1.0) * self.weights.age_weight

    return s

def next_batch(self) -> list[InferenceRequest]:
    if self.queue:
        return [heapq.heappop(self.queue)]
    return []
```

### 5.3 Batch-Optimized Scheduler

The batch scheduler accumulates requests for the same model and flushes when the batch is full or the timer expires. This maximizes GPU throughput.

```
class BatchOptimizedScheduler(Scheduler):
    def __init__(self, config):
        self.batches: dict[str, list[InferenceRequest]] = {} # model_id ->
requests
        self.max_batch = config.batch_optimized.max_batch_size
        self.max_wait_ms = config.batch_optimized.max_wait_ms
        self.batch_start: dict[str, float] = {}

    def submit(self, request):
        model = request.model_id
        if model not in self.batches:
            self.batches[model] = []
            self.batch_start[model] = time.monotonic()
        self.batches[model].append(request)

    def next_batch(self) -> list[InferenceRequest]:
        now = time.monotonic()
        for model, requests in self.batches.items():
            batch_full = len(requests) >= self.max_batch
            timed_out = (now - self.batch_start[model]) * 1000 >= self.max_wait_ms
            if batch_full or timed_out:
                batch = requests[:self.max_batch]
                self.batches[model] = requests[self.max_batch:]
                if not self.batches[model]:
                    del self.batches[model]
                    del self.batch_start[model]
        return batch
```

```
return []
```

## 6. Dispatch Internals

### 6.1 Ray Serve Integration

The dispatcher maintains a map of `model_id` to Ray Serve deployment handles. Handles are obtained once at startup (or lazily on first request) and reused. Since the Inferential server runs inside the Ray cluster, deployment handle calls pass tensors through shared memory with zero serialization.

```
class RayDispatcher:
    def __init__(self, ray_config):
        self.handles: dict[str, DeploymentHandle] = {}
        self.health: dict[str, EndpointHealth] = {}

    def _get_handle(self, model_id: str) -> DeploymentHandle:
        if model_id not in self.handles:
            self.handles[model_id] = serve.get_deployment_handle(model_id)
        return self.handles[model_id]

    async def dispatch(self, requests: list[InferenceRequest]
        ) -> list[DispatchResult]:
        results = []
        # Group by model for potential batching at Ray level
        by_model = defaultdict(list)
        for req in requests:
            by_model[req.model_id].append(req)

        for model_id, model_requests in by_model.items():
            handle = self._get_handle(model_id)
            for req in model_requests:
                start = time.monotonic()
                try:
                    obs_dict = self._reconstruct_numpy(req)
                    result = await handle.infer.remote(obs_dict)
                    latency = (time.monotonic() - start) * 1000
                    self._update_health(model_id, latency, True)
                    results.append(self._build_result(
                        req, result, latency
                    ))
                except Exception as e:
                    latency = (time.monotonic() - start) * 1000
                    self._update_health(model_id, latency, False)
                    results.append(DispatchResult(
                        robot_id=req.robot_id,
                        response_id=uuid4().hex,
                        envelope=b'', payload=b'',
                        latency_ms=latency,
                        success=False, error=str(e),
                    ))
        return results
```

## 6.2 Numpy Reconstruction

The dispatcher reconstructs numpy arrays from raw bytes using slot metadata from the protobuf envelope. This is the one copy between ZMQ and Ray.

```
def _reconstruct_numpy(self, req: InferenceRequest) -> dict:
    obs = Observation() # protobuf
    obs.ParseFromString(req.envelope)
    result = {}
    for slot in obs.slots:
        dtype = np.dtype(slot.dtype)
        data = req.payload[slot.byte_offset:slot.byte_offset + slot.byte_length]
        array = np.frombuffer(data, dtype=dtype).reshape(slot.shape)
        result[slot.key] = array
    if obs.metadata.get("language"):
        result["language"] = obs.metadata["language"]
    return result
```

## 6.3 Endpoint Health Tracking

The dispatcher tracks per-endpoint health using exponential moving average of latency and error rate. If an endpoint's latency exceeds a threshold, the dispatcher logs a warning. Future work: multiple endpoints per model with load-based routing.

```
@dataclass
class EndpointHealth:
    model_id: str
    avg_latency_ms: float = 0.0
    error_rate: float = 0.0
    total_requests: int = 0
    last_error: str | None = None
```



## 7. Cadence Tracker Deep Dive

The cadence tracker is the core innovation that makes Inferential model-agnostic. It learns each robot's natural inference request pattern and predicts when the next request will arrive.

### 7.1 EWMA Algorithm

```
class CadenceTracker:
    def __init__(self, alpha: float = 0.3, overdue_multiplier: float = 1.5):
        self.alpha = alpha
        self.overdue_multiplier = overdue_multiplier
        self._cadence: dict[str, float] = {}
        self._last_request: dict[str, float] = {}
        self._sample_count: dict[str, int] = defaultdict(int)

    def on_request(self, robot_id: str) -> None:
        now = time.monotonic()
        if robot_id in self._last_request:
            interval = now - self._last_request[robot_id]
            if robot_id in self._cadence:
                self._cadence[robot_id] = (
                    self.alpha * interval
                    + (1 - self.alpha) * self._cadence[robot_id]
                )
            else:
                self._cadence[robot_id] = interval
            self._sample_count[robot_id] += 1
        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

    def is_overdue(self, robot_id: str) -> bool:
        if robot_id not in self._cadence:
            return False
        elapsed = time.monotonic() - self._last_request[robot_id]
        return elapsed > self._cadence[robot_id] * self.overdue_multiplier

    def learned_cadence(self, robot_id: str) -> float | None:
        return self._cadence.get(robot_id)

    def confidence(self, robot_id: str) -> float:
        """0.0 to 1.0 based on sample count. >10 samples = high confidence."""
        count = self._sample_count.get(robot_id, 0)
        return min(count / 10.0, 1.0)
```

## 7.2 Behavior by Model Type

Model Type	Cadence	Samples to Converge	Notes
Chunked VLA (16 steps @ 10Hz)	~1.6s	3–5 requests	Stable, predictable cadence
Single-step (OpenVLA @ 10Hz)	~100ms	10–15 requests	Fast cadence, converges in <2 seconds
Single-step (50Hz)	~20ms	15–20 requests	Very fast, alpha=0.2 recommended for stability
VLM queries (irregular)	Variable	Never fully converges	Confidence stays low, scheduler treats as urgent

## 7.3 Edge Cases

- First request: No cadence data. `time_until_next` returns 0.0 (assume urgent). Scheduler uses `latency_budget_ms` from config as the fallback priority signal.
- Robot goes idle then resumes: Large gap detected. EWMA smooths it but the next few predictions may be inaccurate. Confidence drops. Scheduler falls back to urgency and priority signals.
- Robot switches models: Cadence may change (e.g., chunked to single-step). EWMA adapts within 5–10 samples. During transition, both cadence and robot-reported urgency inform scheduling.
- Robot disconnects: Last request timestamp goes stale. `is_overdue` returns True. Server emits disconnect event after configurable threshold.

## 8. Configuration Internals

### 8.1 Validation with Pydantic

All configuration is validated at startup using Pydantic v2 models. Invalid config raises immediately with clear error messages. No silent defaults for required fields.

```
class TransportConfig(BaseModel):
    type: Literal["zmq"] = "zmq"
    bind: str = "tcp://*:5555"
    recv_hwm: int = Field(default=1000, ge=1)
    send_hwm: int = Field(default=1000, ge=1)

class SchedulingConfig(BaseModel):
    strategy: str = "deadline_aware"
    max_queue_size: int = Field(default=1000, ge=1)
    deadline_aware: DeadlineAwareConfig | None = None
    batch_optimized: BatchConfig | None = None
    priority_tiered: TieredConfig | None = None

class RobotDefaults(BaseModel):
    latency_budget_ms: float = Field(default=50.0, gt=0)
    priority: int = Field(default=1, ge=0)

class RobotEntry(BaseModel):
    id: str
    model: str
    latency_budget_ms: float | None = None
    priority: int | None = None

class RobotsConfig(BaseModel):
    defaults: RobotDefaults = RobotDefaults()
    known: list[RobotEntry] = []
    accept_unknown: bool = True

class InferentialConfig(BaseModel):
    ray: dict = {} # Pass-through to Ray
    transport: TransportConfig = TransportConfig()
    scheduling: SchedulingConfig = SchedulingConfig()
    robots: RobotsConfig = RobotsConfig()
    response_tracking: ResponseTrackingConfig = ResponseTrackingConfig()
    observations: ObservationConfig = ObservationConfig()
    metrics: MetricsConfig = MetricsConfig()
```

### 8.2 No External Configuration

Inferential is a library and does not own configuration. There is no YAML file, no config watcher, no hot-reload mechanism. All parameters are passed through Python constructors and method calls. Users who want Hydra, YAML, or any other config system integrate it themselves by passing parsed

values to the library API. Runtime parameter changes (e.g., swapping scheduler) are made through method calls on the Server instance.

## 9. Error Handling

Errors are raised upfront at startup and configuration time. During runtime, errors are logged and the system continues operating. A single malformed observation from one robot must not crash the server or affect other robots.

### 9.1 Error Categories

Category	Behavior	Example
Parameter error	Raise immediately at Server/Connection construction.	Invalid bind address, empty model list, unknown scheduler name
Observation error	Log warning, drop observation, continue.	Malformed protobuf, payload size mismatch, missing robot_id
Scheduler error	Raise QueueFullError to receive loop. Observation dropped, metric recorded.	Queue at max capacity
Dispatch error	Log error, record failure metric, continue with other requests.	Ray endpoint unreachable, model error, timeout
Transport error	Log error, attempt recovery. Fatal if bind fails at startup.	ZMQ socket error, send failure to disconnected robot

### 9.2 Custom Exceptions

```

class InferentialError(Exception):
    """Base exception for all Inferential errors."""

class ObservationError(InferentialError):
    """Malformed or invalid observation."""

class QueueFullError(InferentialError):
    """Scheduler queue at capacity."""
    def __init__(self, robot_id: str):
        self.robot_id = robot_id
        super().__init__(f"Queue full, dropping request from {robot_id}")

class DispatchError(InferentialError):
    """Failed to dispatch to Ray Serve."""

class TransportError(InferentialError):
    """ZMQ transport failure."""

```

## 10. Testing Architecture

Every module is testable in isolation via Protocol-based interfaces. No test requires a running Ray cluster or ZMQ socket.

### 10.1 Mock Implementations

```
class MockTransport:
    """In-memory transport for testing. No ZMQ."""
    def __init__(self):
        self.inbox: asyncio.Queue[IncomingObservation] = asyncio.Queue()
        self.outbox: list[OutgoingResponse] = []

    async def recv(self) -> IncomingObservation:
        return await self.inbox.get()

    async def send(self, response: OutgoingResponse) -> None:
        self.outbox.append(response)

class MockDispatcher:
    """Returns predetermined results. No Ray."""
    def __init__(self, latency_ms: float = 50.0):
        self.latency_ms = latency_ms
        self.dispatched: list[InferenceRequest] = []

    async def dispatch(self, requests):
        self.dispatched.extend(requests)
        await asyncio.sleep(self.latency_ms / 1000)
        return [DispatchResult(
            robot_id=r.robot_id,
            response_id=uuid4().hex,
            envelope=b'mock', payload=b'mock',
            latency_ms=self.latency_ms,
            success=True,
        ) for r in requests]
```

### 10.2 Test Matrix

Module	Test Type	Dependencies
transport/	Unit: message framing, identity tracking	MockTransport (no ZMQ)
observation/	Unit: protobuf decode, validation, staleness	None (pure functions)
scheduler/	Unit: scoring, ordering, backpressure, all strategies	None (pure data structures)
tracking/	Unit: cadence EWMA, convergence, edge cases	None (pure math)

Module	Test Type	Dependencies
dispatch/	Unit: numpy reconstruction, health tracking	MockDispatcher (no Ray)
metrics/	Unit: ring buffer, aggregation, callbacks	None (in-memory)
config/	Unit: YAML parse, validation, hot-reload	None (file system)
server.py	Integration: full pipeline with mocks	MockTransport + MockDispatcher
client.py	Unit: observe/get_result encoding	MockTransport (no ZMQ)
End-to-end	Integration: real ZMQ + real Ray + simulated robots	Full stack (CI only)

## 10.3 Simulation Harness

A built-in simulation module enables end-to-end testing and benchmarking without physical robots. It spawns N simulated robot clients that send observations at configurable cadences and consume responses.

```
from inferential.testing import SimulatedFleet

fleet = SimulatedFleet(
    server="localhost:5555",
    robots=[
        {"id": "arm-1", "model": "pi0", "cadence_hz": 0.6}, # chunked
        {"id": "arm-2", "model": "pi0", "cadence_hz": 0.6},
        {"id": "arm-3", "model": "octo", "cadence_hz": 10}, # single-step
    ],
)
await fleet.run(duration_seconds=60)
fleet.print_stats() # latency, throughput, starvation events
```

## 11. Public API Surface

The public API is intentionally minimal. Users interact with `Server`, `Connection`, and `register_policy`. Everything else is internal.

### 11.1 Exports from `inferential/__init__.py`

```
# inferential/__init__.py

from inferential.server import Server
from inferential.client import Connection
from inferential.policy import register_policy
from inferential.exceptions import (
    InferentialError,
    ObservationError,
    QueueFullError,
    DispatchError,
    TransportError,
)

__all__ = [
    "Server",
    "Connection",
    "register_policy",
    "InferentialError",
    "ObservationError",
    "QueueFullError",
    "DispatchError",
    "TransportError",
]
```

### 11.2 Server API

```
class Server:
    def __init__(
        self,
        bind: str = "tcp://*:5555",
        models: list[str] = None,      # Ray Serve deployment names
    ):
        """
        Initialize server. Validates parameters immediately.
        Raises InferentialError on invalid parameters.
        """

    def use_scheduler(
        self,
        name: str,                    # "deadline_aware", "round_robin", etc.
        policy: str | None = None,    # Scoring policy name
        **kwargs,                    # Strategy-specific params
    ) -> None:
```



```
        """Select scheduler strategy and optional scoring policy."""

    async def run(self) -> None:
        """Start the event loop. Blocks until shutdown."""

    async def shutdown(self) -> None:
        """Graceful shutdown. Drain queue, close transport."""

    @property
    def metrics(self) -> MetricsCollector:
        """Access metrics for queries and callbacks."""

    @property
    def connected_robots(self) -> list[str]:
        """List of currently connected robot IDs."""
```

### 11.3 Client SDK API (Connection + Model)

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.

```
class Connection:
    def __init__(
        self,
        server: str,          # "host:port"
        robot_id: str,
        reconnect_ivl_ms: int = 100,    # Initial reconnect interval
        reconnect_max_ms: int = 5000,   # Max reconnect backoff
    ):
        """
        Connect to Inferential server via ZMQ DEALER.
        ZMQ handles reconnection automatically with exponential
        backoff. Messages buffer locally during disconnection.
        """

    def model(
        self,
        model_id: str,
        latency_budget_ms: float = 50.0,
        priority: int = 1,
    ) -> "Model":
        """Create a model handle bound to this connection."""

    def close(self) -> None:
        """Close ZMQ connection."""

class Model:
    """Lightweight handle for a specific model endpoint."""
```

```
def observe(self, **slots) -> None:
    """
    Send observation to this model. Keyword args are slot
    names mapped to numpy arrays.
    Special keys: 'language' (str), 'urgency' (float),
                  'steps_remaining' (int).
    Non-blocking. Raises ObservationError on invalid data.
    """

def get_result(self, timeout_ms: int = 0) -> dict | None:
    """
    Get model result. Returns dict of numpy arrays, or None
    if no result available (non-blocking when timeout_ms=0).
    """
```

## Multi-Model Usage

```
import inferential as infr

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

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

# Independent observations
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)
```

## Reconnection Behavior

ZMQ DEALER handles reconnection automatically. If the server goes down, the SDK buffers outgoing messages and retries connection with exponential backoff (100ms initial, doubling to 5s max). When the server returns, the connection re-establishes and buffered messages flush. The robot code does not need to handle disconnection explicitly.

## 11.4 Policy Registration

```
from inferential import register_policy

# 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")
```

## 11.5 Metric Callbacks

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

# Register a callback for every metric event
@server.on_metric
def log_metric(name: str, value: float, labels: dict):
    print(f"{name} = {value} | {labels}")

# Or query metrics directly
stats = server.metrics.get_stats(
    "inference_latency_ms",
    labels={"robot": "arm-1"},
    window_seconds=300,
)
print(f"p99 latency: {stats.p99}ms")
```

— End of Architecture Document —