Authors: Khairullin Artur (RU) *prcr98@gmail.com* 09 July 2026

—

PBDR Policy-Based Decentralized Routing for Enterprise AI Inference

Abstract

The escalating demand for Large Language Models (LLMs) and other AI services in enterprise settings is hindered by high infrastructure costs, data privacy concerns, and the underutilization of existing computational assets. This paper introduces a novel middleware architecture, Policy-Based Decentralized Routing (PBDR) , which transforms a heterogeneous fleet of standard workstations into a resilient, private, and cost-effective inference cluster. Unlike centralized systems, PBDR operates without a global scheduler or a shared request queue. Instead, it employs a client-driven, policy-based routing mechanism where each client independently computes a Cost Vector and applies a Policy Weight Vector via scalar product to select the optimal compute node.

This paper presents a formal mathematical model of the system, defines the routing problem as a constrained optimization task, and proves key properties of the algorithm, including correctness, fault tolerance, and monotonicity. We provide a detailed architectural specification, including a formal Monitor API (OpenAPI v3), UML diagrams, a rigorous cost function model, and a comprehensive complexity analysis. We demonstrate, through a mathematical model of network traffic and fault scenarios, that the PBDR approach is linearly scalable, highly resilient, and introduces negligible overhead. This work positions the Policy-Driven Optimizer as a novel class of middleware for enterprise AI, with applications extending beyond LLMs to encompass flexible, application-aware routing policies.

Keywords: Distributed Systems, Large Language Models, Enterprise Computing, GPU Utilization, Decentralized Architecture, Policy-Based Routing, Cost Function, Load Balancing, Fault Tolerance, Middleware, Policy-Driven Optimizer.

1. Introduction

The rapid advancement of Large Language Models (LLMs) has unlocked transformative potential for various enterprise applications, from intelligent assistants and code generation to Retrieval-Augmented Generation (RAG) systems for internal knowledge bases. However, the widespread deployment of these models in a corporate environment faces significant hurdles:

1. High Infrastructure Costs: Deploying on-premise dedicated AI clusters requires substantial investment in specialized servers with high-end, often expensive, server-grade GPUs (e.g., NVIDIA A100, H100). A single server can cost upwards of $50,000$ −100,000, making it prohibitive for many organizations.

Authors: Khairullin Artur (RU) *prcr98@gmail.com* 09 July 2026

—

2. Data Privacy and Security: Concerns over data leakage and compliance (e.g., GDPR, HIPAA, SOC2) often prohibit the use of public, third-party AI APIs. Internal deployment is frequently a strict requirement. According to a 2023 survey, 78% of enterprises cite data privacy as their primary concern for adopting AI [1].

3. Resource Underutilization: Many enterprise workstations used by engineers, designers, and researchers are equipped with increasingly powerful consumer-grade GPUs (e.g., RTX 3060, 4070, 4090). These GPUs are often idle or underutilized for 75-95% of the workday [2]. A significant portion of the required compute resources are already paid for and present within the corporate network.

1.1. Core Idea: Policy-Based Decentralized Routing

This paper introduces Policy-Based Decentralized Routing (PBDR) , a new class of middleware for enterprise AI inference. The PBDR architecture is characterized by:

- Decentralization: No central scheduler, load balancer, or queue. This eliminates single points of failure and performance bottlenecks.
- Client-Driven Decisions: The routing logic resides within the client (or middleware acting on its behalf), which computes a cost for each node based on its current state.
- Policy-Based: The routing logic is not fixed; it's defined by a configurable Policy Vector. This allows different departments (e.g., Support vs. Engineering) or applications to use different routing strategies.
- Modularity: The system is designed to be extensible and compatible with any OpenAI-compatible AI engine (e.g., vLLM [3], TGI [4], Ollama [5]).
- Stateless Nodes: Compute nodes are stateless from a routing perspective, making them easy to manage, replace, and scale.

1.2. Contributions

The main contributions of this paper are:

1. A formal definition of the enterprise AI inference routing problem, establishing it as a constrained optimization task.
2. The design and specification of the PBDR middleware architecture, including detailed UML diagrams.
3. A formal mathematical model of the Decision Engine as a Policy-Driven Optimizer, with a Cost Vector formulation.
4. Formal proofs of correctness (Theorem 1), fault tolerance (Theorem 2), monotonicity (Theorem 3), and an analysis of algorithmic complexity (Theorem 4).
5. A complete specification of the Monitor API using OpenAPI 3.0, defining a standardized interface for node state publication.

Authors: Khairullin Artur (RU) *prcr98@gmail.com*

—

6. The introduction of Policy-Based Routing as a mechanism to achieve application- and user-specific performance objectives, decoupling routing logic from infrastructure.
7. A comprehensive network traffic analysis demonstrating the protocol's scalability.
8. A scalability analysis showing that the system handles growth in clients and nodes linearly.
9. A validation framework with three stages: Simulation, Prototype, and Production.
10. An analysis of computational costs for the client-side routing decision.
11. A dedicated security considerations section addressing enterprise requirements.

2. Scientific Contributions

This section explicitly enumerates the novel scientific contributions of this work. While the individual components are rooted in existing research, their combination and formalization represent a significant advance in the field of distributed AI inference.

2.1. The Cost Vector Formulation

Contribution: We introduce the Cost Vector

$C(r, v_i) \in R^{10}$

as a multi-dimensional representation of cost components. Unlike traditional approaches that use a single scalar metric (e.g., queue length, ETA), the Cost Vector preserves the multi-faceted nature of inference costs.

Novelty: This is the first work to formally represent inference routing costs as a vector rather than a scalar. This allows for:

- Composability: New cost factors can be added without changing the core algorithm.
- Interpretability: Each component has a clear physical meaning.
- Flexibility: Different policies can weight components differently.

2.2. The Policy Weight Vector

Contribution: We introduce the Policy Weight Vector $\Pi \in R^{10}$ as a formal representation of routing policies. The scalar cost is computed as the dot product: $Cost = \Pi \cdot C$

.

Novelty: This is the first formalization of routing policies as vectors in a weight space. This enables:

Authors: Khairullin Artur (RU) *prcr98@gmail.com* 09 July 2026

—

- Mathematical clarity: Policies are just vectors.
- Composability: Policies can be combined through vector operations.
- Learning: Policies can be learned from data via regression.
- Comparison: Policies can be compared using distance metrics.

2.3. The Policy-Driven Optimizer

Contribution: We formalize the Policy-Driven Optimizer as a function:

$$Optimizer(r, \Pi, State(V, t)) = arg \min_{v_i \in V_{eligible}} \Pi \cdot C(r, v_i)$$

Novelty: This is the first formalization of an inference routing optimizer as a policy-driven mathematical function. This abstraction:

- Decouples the optimizer from the underlying infrastructure.
- Enables formal analysis (correctness, monotonicity, complexity).
- Generalizes to any stateless, compute-intensive workload.

2.4. The Stateless Monitor API

Contribution: We specify the Monitor API as a standardized, stateless interface for nodes to publish objective state. The API returns raw measurements, not computed scores.

Novelty: This is the first specification of a vendor-agnostic, algorithm-agnostic monitoring API for distributed AI inference. Key innovations:

- Algorithm independence: Nodes know nothing about routing policies.
- Vendor neutrality: Works with any OpenAI-compatible server.
- Extensibility: New metrics can be added without breaking compatibility.

2.5. Client-Side Decentralized Routing

Contribution: We propose fully decentralized routing where each client (or middleware instance) independently makes routing decisions without coordination.

Novelty: Unlike existing approaches that rely on centralized schedulers or distributed consensus, our approach:

- Eliminates the need for global state synchronization.
- Eliminates single points of failure.
- Scales linearly with the number of nodes and clients.
- Is mathematically proven to be correct (Theorem 1) and fault-tolerant (Theorem 2).

Authors: Khairullin Artur (RU) *prcr98@gmail.com* 09 July 2026

—

2.6. Formal Proof of Monotonicity

Contribution: We provide a formal proof that the Cost Vector components are monotonic non-decreasing functions of their metrics (Theorem 3).

Novelty: This is the first formal proof that a routing optimizer using this cost structure makes consistent and predictable decisions. The monotonicity property ensures that:

- A node better in all metrics will never be worse in cost.
- The optimizer's decisions are rational and explainable.

2.7. Summary of Contributions

Contribution	Type	Impact
Cost Vector	Formal Model	Enables multi-dimensional cost representation
Policy Vector	Formal Model	Enables mathematical policy definition
Policy-Driven Optimizer	Algorithm	Enables formal analysis and generalization
Stateless Monitor API	Protocol	Enables vendor and algorithm independence
Client-Side Routing	Architecture	Eliminates SPOF, enables linear scaling
Monotonicity Proof	Theory	Ensures rational decision-making

3. Related Work

This work is positioned at the intersection of several existing fields: centralized AI serving, distributed computing, and enterprise middleware.

3.1. Centralized AI Inference Serving

Modern systems for AI serving, such as vLLM [3], TensorRT-LLM [6], and TGI [4], are designed to maximize throughput and efficiency within a dedicated, homogeneous cluster. They often include sophisticated memory management (e.g., PagedAttention [3]) and centralized schedulers. Ollama [5] and llama.cpp [7] provide easy-to-use platforms for running models on local or single-server setups.

Limitations: These solutions require significant upfront investment in server-grade hardware and are typically designed for a central cluster managed by a single entity. While they can be scaled using Kubernetes and standard load balancers (e.g., HAProxy, NGINX), this adds further operational complexity and does not address the problem of resource underutilization on existing workstations.

Authors: Khairullin Artur (RU) *prcr98@gmail.com* 09 July 2026

—

3.2. Distributed and Volunteer Computing

The idea of using spare compute cycles from a network of computers is the foundation of distributed computing projects like BOINC [8] and Folding@home [9]. While highly successful for background, compute-intensive tasks (e.g., protein folding, SETI analysis), these systems are not designed for low-latency, interactive generation tasks required by modern AI models. They typically rely on centralized project servers and task schedulers, creating a single point of failure and a performance bottleneck.

3.3. Decentralized Workload Distribution

Research into decentralized load balancing often uses gossip protocols [10] or consistent hashing [11] to distribute tasks without a central point. However, these algorithms often focus on homogeneous workloads and ignore the specific, dynamic characteristics of AI inference, such as model size, GPU performance, context length, and the significant cost of model switching.

3.4. Enterprise Middleware and Policy-Based Systems

Policy-based management [12] is a well-established concept in networking and cloud computing (e.g., for QoS, security). Service meshes (e.g., Istio [13]) use policy-based routing for microservices. The PBDR architecture adapts these concepts to a distributed computing context, applying them to the scheduling of AI inference requests.

3.5. Summary

While the components of our proposed system are rooted in existing research, the Policy-Based Decentralized Routing architecture represents a novel combination of principles. It integrates the ease of use of systems like Ollama with the resilience of decentralized systems, the cost-efficiency of volunteer computing, and the policy-driven intelligence of modern schedulers, creating a unique solution for the specific challenges of enterprise AI deployment.

4. Formal Model of the System

To rigorously define the problem and analyze the proposed solution, we introduce a formal model of the system.

4.1. System Definition

Definition 1: System Graph

Let the system be defined as a bipartite graph

$G = (V, E)$, where:

Authors: Khairullin Artur (RU) *prcr98@gmail.com* 09 July 2026

—

$V = \{v_1, v_2, ..., v_k\}$ is the set of GPU nodes. $E \subseteq V \times C$ represents the communication links between GPU nodes and clients.

Each node v_i has an associated state $State(v_i, t)$at time t.

Definition 2: Node State

The state of a node v_i at time t is a tuple of objective metrics:

$$State(v_i, t) = (h_i, g_i, q_i, m_i, l_i, p_i, c_i, v_i, \tau_i)$$

Where:

$h_i \in \{0, 1\}$: Health flag (1 = healthy, 0 = unhealthy).

$g_i \in [0, 100]$: GPU utilization (percentage).

$q_i \in Z^+$: Current queue length (number of waiting jobs).

$m_i \in M$: Currently loaded model (where M is the set of all models).

$l_i \in [0, 100]$: CPU load (percentage).

$p_i \in R^+$: Performance metric (e.g., tokens/second for the current model).

$c_i \in R^+$: Context length capacity (in tokens).

$v_i \in R^+$: Free VRAM (in MiB).

$\tau_i \in R^+$: Current temperature (in °C).

Definition 3: Discovery Function

The Discovery process is defined as the function:

$Discovery: V \to \{State(v_i, t) | v_i \in V\}$

This function returns a mapping from each node to its current state.

Definition 4: Request

A request $r$is a tuple:

$$r = (m_r, p_r, o_r, s_r, c_r)$$

Where:

$m_r \in M$: The requested model.

Authors: Khairullin Artur (RU) *prcr98@gmail.com* 09 July 2026

—

$p_r \in Z^+$: Number of tokens in the prompt.

$o_r \in Z^+$: Expected number of tokens in the output.

$s_r \in R^+$: The required VRAM (in MiB).

$c_r \in Z^+$: Required context length (in tokens).

4.2. The Cost Vector

Instead of a scalar cost, we introduce a Cost Vector C that represents the multi-dimensional cost components.

Definition 5: Cost Vector For a request $r$and node v_i , the Cost Vector is:

$$C(r, v_i) = (c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_8, c_9, c_{10}) \in R^{10}$$

Where each component corresponds to a specific cost factor (see Section 10):

$c_1 = T_{wait}$: Wait Time

$c_2 = T_{inf}$: Inference Time

$c_3 = T_{cold}$: Cold Start Penalty

$c_4 = P_{queue}$: Queue Depth Penalty

$c_5 = P_{gpu}$: GPU Load Penalty

$c_6 = P_{cpu}$: CPU Load Penalty

$c_7 = P_{vram}$: VRAM Pressure Penalty

$c_8 = P_{temp}$: Thermal Penalty

$c_9 = B_{idle}$: Idle Bonus (negative cost)

$c_{10} = P_{net}$: Network Penalty

Definition 6: Policy Weight Vector A policy $\Pi$is defined as a weight vector:

$$\Pi = (\omega_1, \omega_2, \omega_3, \omega_4, \omega_5, \omega_6, \omega_7, \omega_8, \omega_9, \omega_{10}) \in R^{10}$$

Where each $\omega_j \geq 0$ defines the importance of the corresponding cost component.

Authors: Khairullin Artur (RU) *prcr98@gmail.com* 09 July 2026

4.3. The Routing Problem

The fundamental problem is:

Routing Problem: Given a request r , a policy vector Π, and a set of nodes V with their current states $State(v_i, t)$, find the node v^* that minimizes the scalar product of the Cost Vector and the Policy Weight Vector, while satisfying a set of hard constraints C.

$$v^* = \arg\ min_{v_i \in V}\ \Pi \cdot C(r, v_i) subject\ to C(r, State(v_i, t)) = True$$

Where:

$$\Pi \cdot C = \sum_{j=1}^{10} \omega_j \cdot c_j$$

This formulation is elegant because:

1. The Cost Vector represents objective, measurable factors.
2. The Policy Vector represents subjective business priorities.
3. The scalar product combines them mathematically cleanly.
4. Policies can be easily compared, composed, and learned.

4.4. Hard Constraints

The hard constraints $C(r, v_i)$ that a node must satisfy to be eligible for a request r are:

1. Node Availability: The node must be healthy and accepting new jobs.

$$h_i = 1 \wedge accept_new_jobs(v_i) = True$$

2. Queue Constraint: The node's queue must not be full.

$$q_i < max_queue(v_i)$$

3. VRAM Availability: The node must have enough free VRAM to load the model (if not already loaded).

$$v_i + model_memory(m_r) \geq s_r$$

Where $model_memory(m_r)$ is the VRAM required to load model m_r (0 if already loaded).

4. Model Availability: The requested model must be available on the node.

$$m_r \in available_models(v_i)$$

5. Context Length: The node must support the required context length.

—

$$c_r \leq min(c_i, max_context_limit(v_i))$$

If any constraint is violated, the node is excluded from consideration.

5. PBDR Architecture

This section describes the components of the PBDR middleware architecture in detail, including formal UML diagrams.

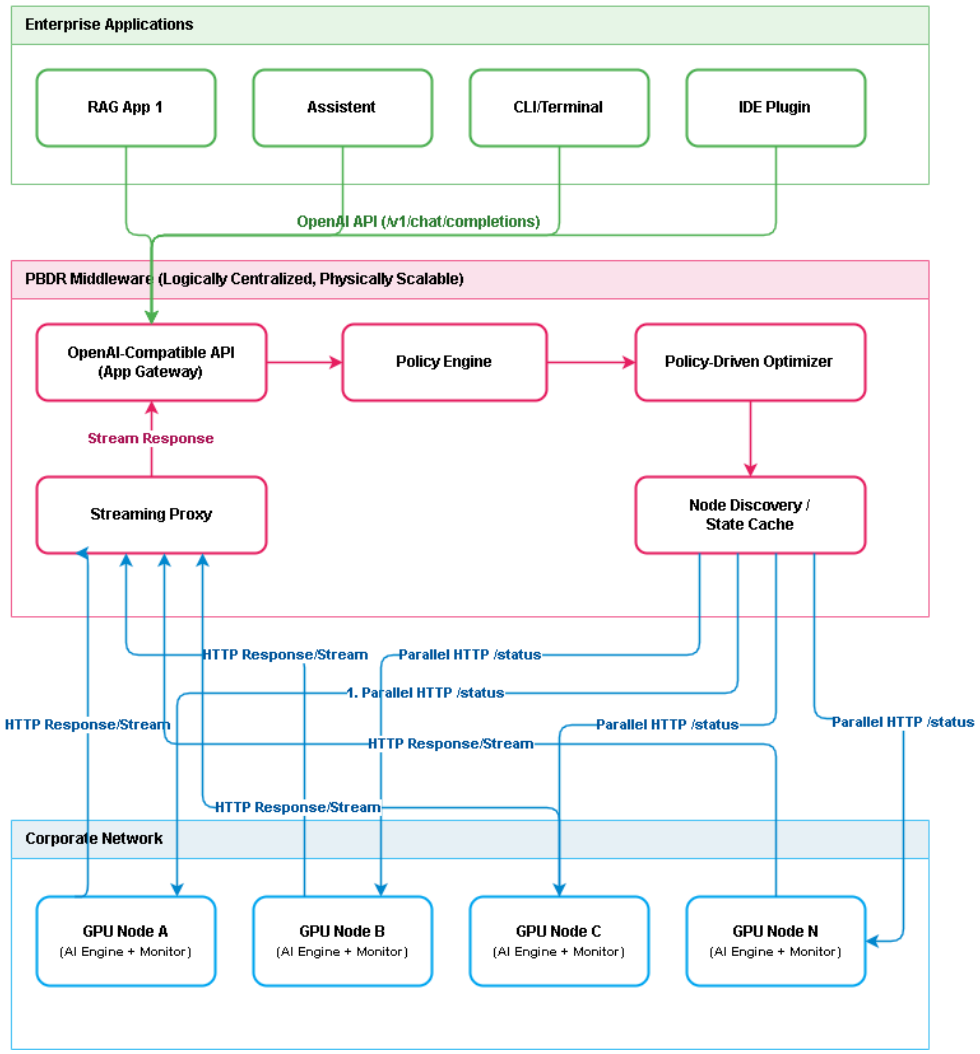
5.1. Architectural Overview

The PBDR system consists of three primary logical components: the Enterprise Applications, the PBDR Middleware (which embodies the Policy-Driven Optimizer), and the GPU Nodes (which represent the compute resources).

Key Architectural Decision: The PBDR Middleware is logically centralized but physically scalable. Multiple instances of the middleware can be deployed in a stateless manner. Each instance operates independently and does not require synchronization with other instances. This design choice:

1. Eliminates the middleware as a single point of failure (SPOF).
2. Allows for horizontal scaling of the middleware layer.
3. Maintains the decentralized nature of the routing decisions.
4. Simplifies deployment and management.

Authors: Khairullin Artur (RU) *prcr98@gmail.com* 09 July 2026

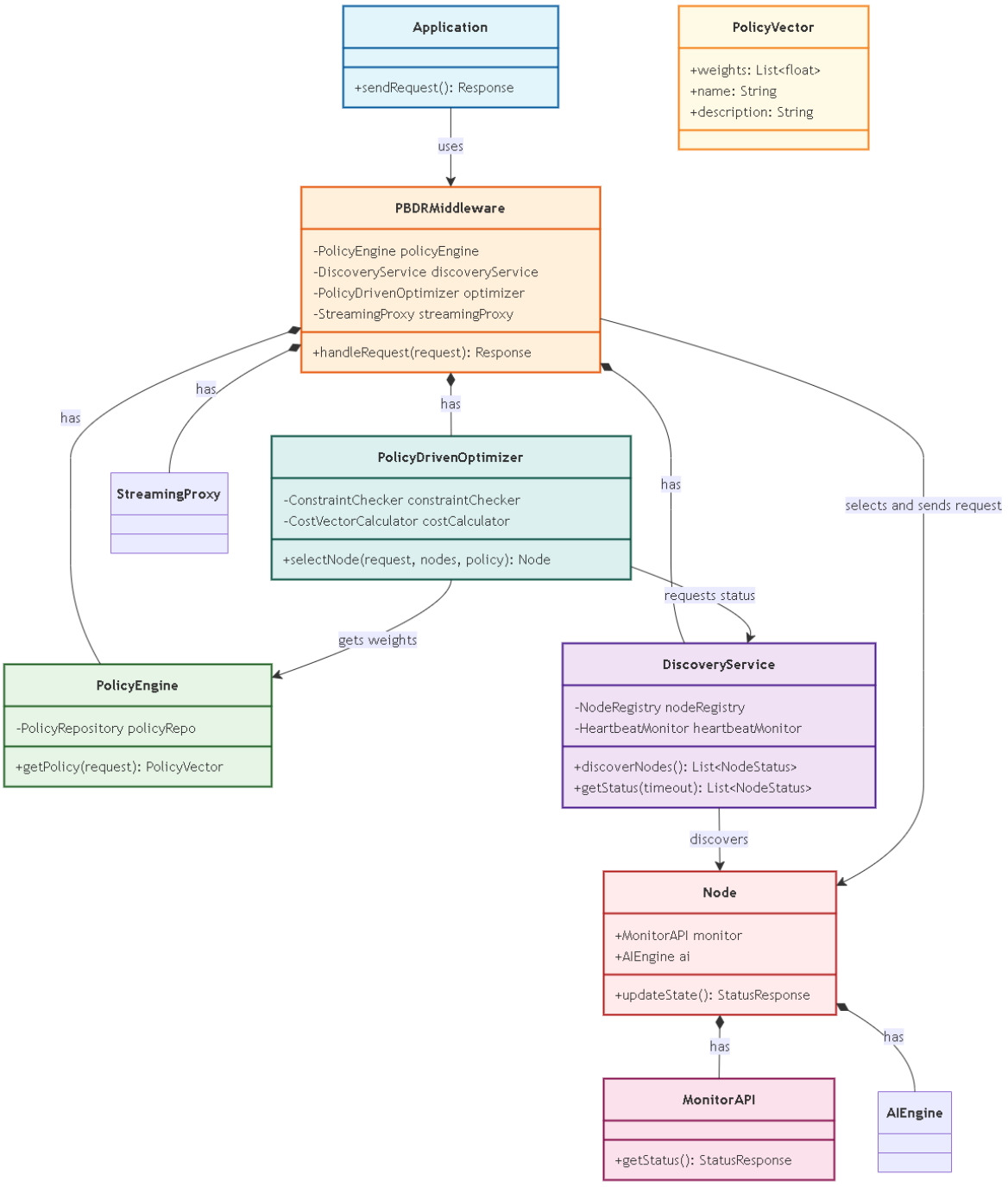


Authors: Khairullin Artur (RU) *prcr98@gmail.com* 09 July 2026

—

5.2. Component Diagram (UML)

The following UML Component Diagram details the internal structure of the PBDR Middleware.

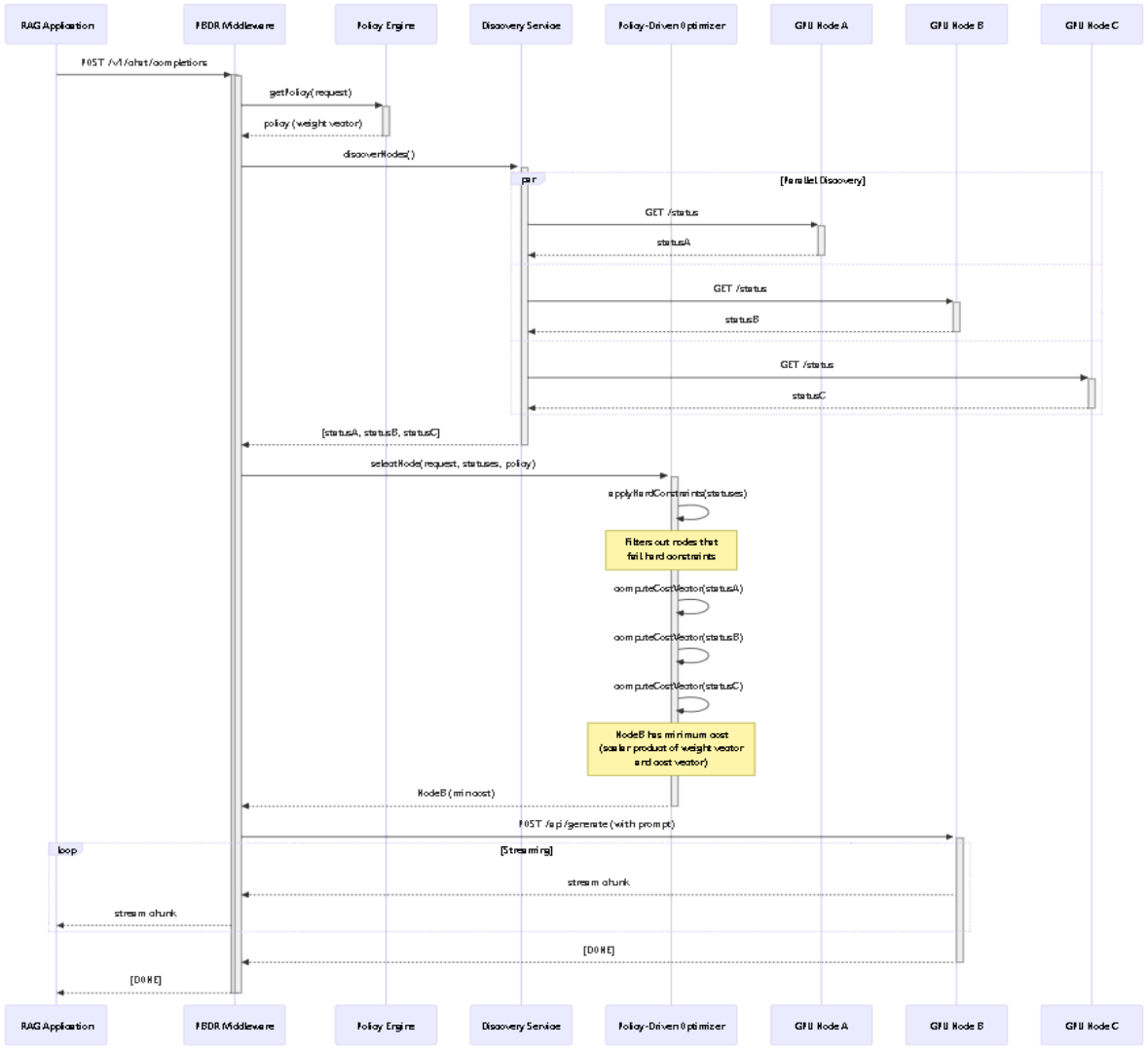


Authors: Khairullin Artur (RU) *prcr98@gmail.com* 09 July 2026

—

5.3. Sequence Diagram

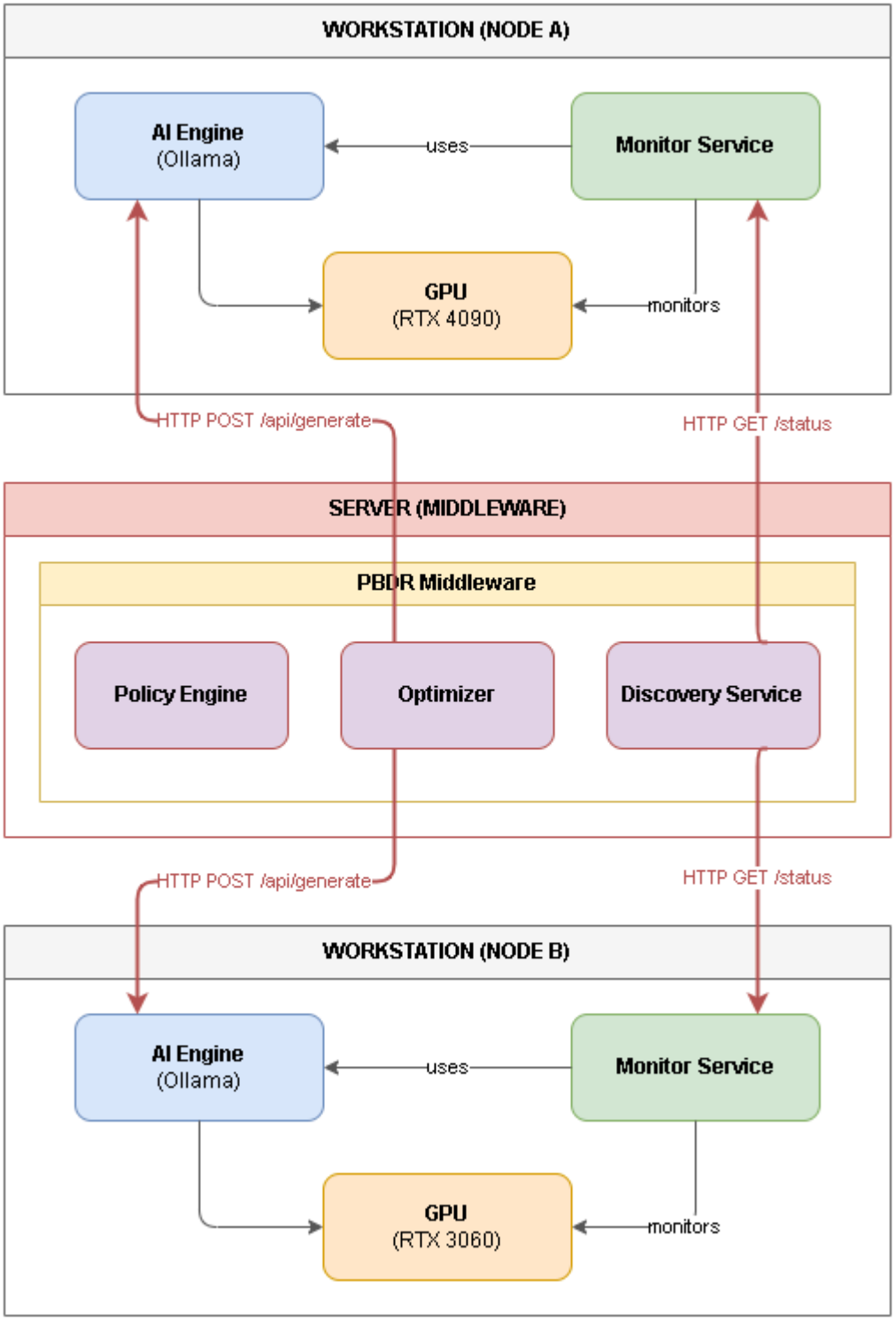
The following sequence diagram shows the interaction between components for a single request.



Authors: Khairullin Artur (RU) *prcr98@gmail.com* 09 July 2026

5.4. Deployment Diagram

The deployment diagram shows how components are distributed across physical nodes.

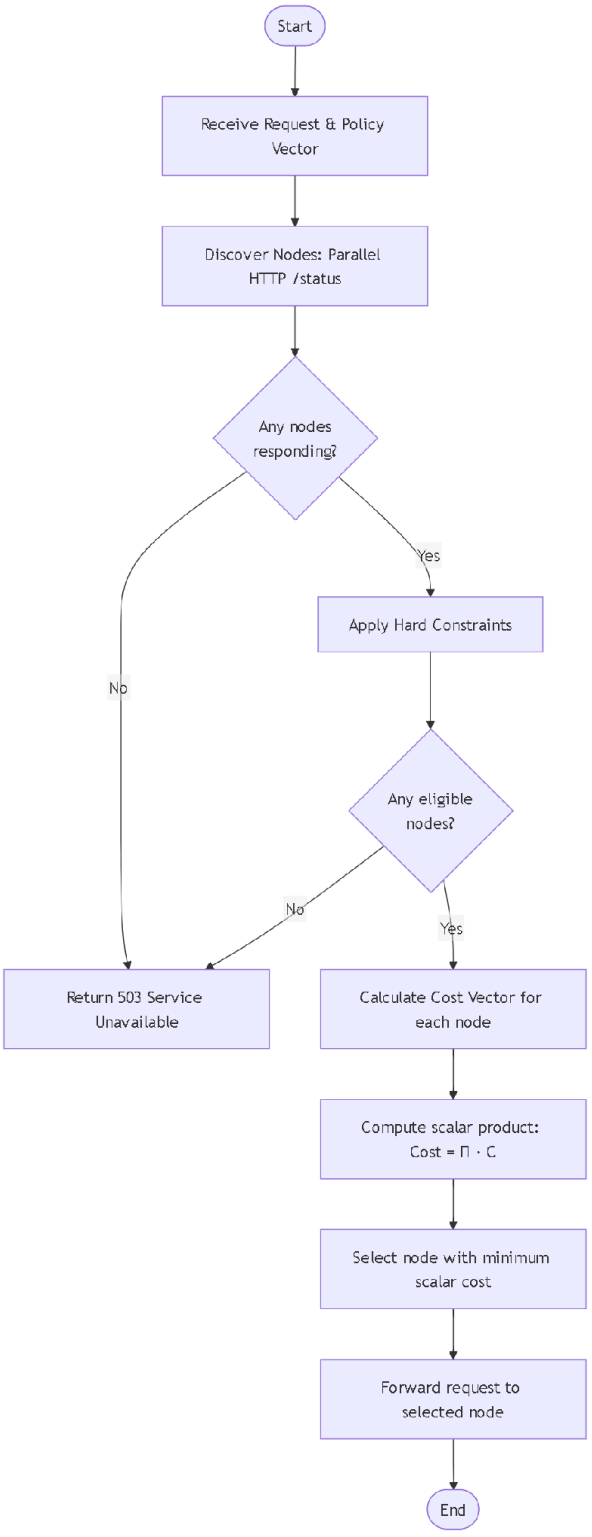


Authors: Khairullin Artur (RU) *prcr98@gmail.com* 09 July 2026

—

5.5. Activity Diagram: Policy-Driven Optimizer Flow

The following activity diagram details the decision-making process within the Policy-Driven Optimizer.

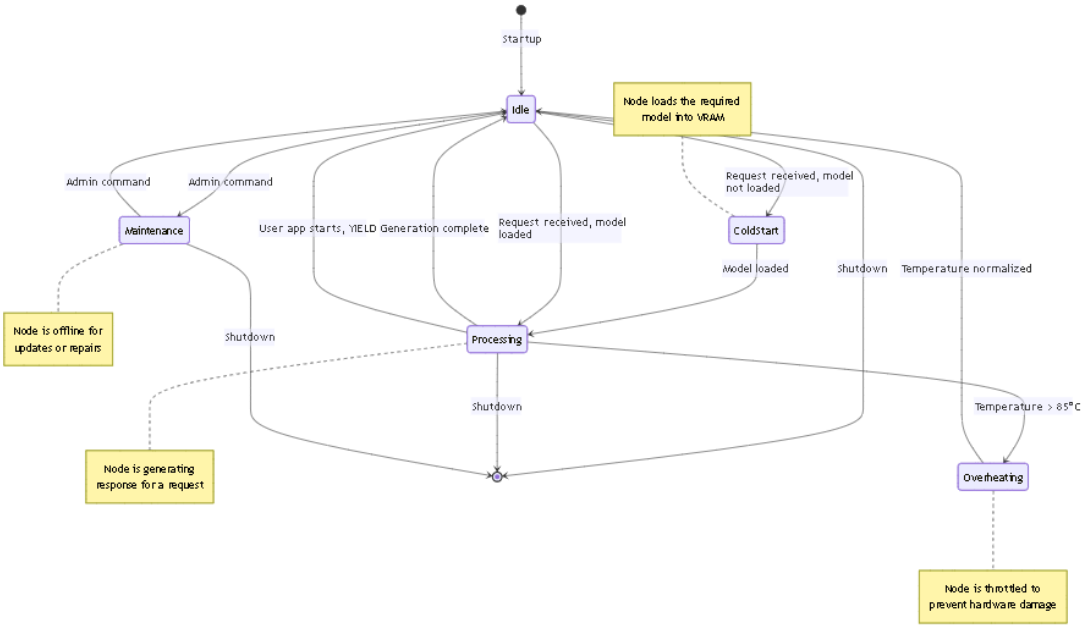


Authors: Khairullin Artur (RU) *prcr98@gmail.com* 09 July 2026

—

5.6. State Machine of a Node

The following state machine describes the lifecycle of a GPU node in the PBDR system.



6. Formal Specification of the Monitor API

The Monitor API is a crucial component defined as an interface to decouple the nodes from the routing logic. The API is designed to be minimal and stateless. Below is its formal specification using OpenAPI 3.0.

6.1. OpenAPI Specification

```yaml
openapi: 3.0.3
info:
  title: Monitor API
  description: |
    The Monitor API provides a standardized interface for nodes to publish
their
    objective state. This API is designed to be minimal, stateless, and
    algorithm-agnostic. Nodes publish their measured metrics, not computed
scores.
  version: 1.0.0
paths:
```

Authors: Khairullin Artur (RU) *prcr98@gmail.com*

—

```
  /status:
    get:
      summary: Get the node's current status
         description: Returns a comprehensive JSON object describing the
node's state.
      operationId: getStatus
      parameters:
        - name: X-Request-ID
          in: header
          schema:
            type: string
          description: Optional request ID for tracing and debugging.
      responses:
        '200':
          description: Successful response
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/StatusResponse'
        '503':
             description: Service Unavailable (node is in maintenance or
overloaded)
          content:
            application/json:
              schema:
                type: object
                properties:
                  status:
                    type: string
                    example: "503 Service Unavailable"
                  message:
```

—

```
                    type: string
                    example: "Node is in maintenance mode"
        '500':
          description: Internal Server Error
          content:
            application/json:
              schema:
                type: object
                properties:
                  error:
                    type: string
                    example: "Failed to query GPU status"
components:
  schemas:
    StatusResponse:
      type: object
      properties:
        node:
          $ref: '#/components/schemas/NodeInfo'
        gpu:
          $ref: '#/components/schemas/GPUInfo'
        cpu:
          $ref: '#/components/schemas/CPUInfo'
        queue:
          $ref: '#/components/schemas/QueueInfo'
        models:
          $ref: '#/components/schemas/ModelsInfo'
        limits:
          $ref: '#/components/schemas/LimitsInfo'
        performance:
          $ref: '#/components/schemas/PerformanceInfo'
```

—

```
    required:
      - node
      - gpu
      - queue
      - models
      - limits
      - performance

  NodeInfo:
    type: object
    properties:
      hostname:
        type: string
        description: The hostname of the node
      ip:
        type: string
        format: ipv4
        description: The IPv4 address of the node
      version:
        type: string
        description: The version of the Monitor service
        example: "1.0.0"
      uptime:
        type: integer
        description: System uptime in seconds
        example: 86400
      last_update:
        type: number
        format: float
        description: Unix timestamp of the last status update
        example: 1700000000.0
```

Authors: Khairullin Artur (RU) *prcr98@gmail.com*

—

```
    required:
      - hostname
      - version
      - uptime
      - last_update

  GPUInfo:
    type: object
    properties:
      gpu_name:
        type: string
        description: Full name of the GPU
        example: "NVIDIA GeForce RTX 4090"
      gpu_uid:
        type: string
        description: Unique identifier for the GPU (UUID)
        example: "GPU-12345678-90AB-CDEF-1234-567890ABCDEF"
      driver_version:
        type: string
        description: NVIDIA driver version
        example: "545.23.06"
      cuda_version:
        type: string
        description: CUDA version
        example: "12.3"
      temperature:
        type: number
        format: float
        description: Current GPU temperature in Celsius
        example: 65.5
      power:
```

—

```
      type: number
      format: float
      description: Current power draw in Watts
      example: 320.0
    gpu_utilization:
      type: number
      format: float
      description: GPU core utilization (0-100%)
      example: 20.0
      minimum: 0
      maximum: 100
    memory_total:
      type: number
      format: float
      description: Total VRAM available in MiB
      example: 24576.0
    memory_used:
      type: number
      format: float
      description: VRAM currently in use in MiB
      example: 10240.0
    memory_free:
      type: number
      format: float
      description: VRAM free for use in MiB
      example: 14336.0
    fan_speed:
      type: number
      format: float
      description: Current fan speed (0-100%)
      example: 50.0
```

—

```
        minimum: 0
        maximum: 100
      pci_bus:
        type: string
        description: PCI bus address
        example: "0000:01:00.0"
    required:
      - gpu_name
      - temperature
      - gpu_utilization
      - memory_total
      - memory_used
      - memory_free

  CPUInfo:
    type: object
    properties:
      cpu_load:
        type: number
        format: float
        description: CPU load average (1-minute) as percentage
        example: 15.5
        minimum: 0
        maximum: 100
      cpu_threads:
        type: integer
        description: Total number of CPU threads/cores available
        example: 24
      ram_total:
        type: number
        format: float
```

—

```
        description: Total system RAM in MiB
        example: 131072.0
      ram_used:
        type: number
        format: float
        description: Used system RAM in MiB
        example: 24000.0
      ram_free:
        type: number
        format: float
        description: Free system RAM in MiB
        example: 107072.0
    required:
      - cpu_load
      - cpu_threads
      - ram_total
      - ram_used
      - ram_free

  QueueInfo:
    type: object
    properties:
      busy:
        type: boolean
        description: Is the node currently processing a job?
      queue_length:
        type: integer
        description: Number of jobs waiting in the internal queue
        example: 2
      current_job_uid:
        type: string
```

—

```
          description: ID of the job currently being processed (if any)
          example: "job-12345"
        current_job_elapsed:
          type: number
          format: float
          description: Elapsed time of the current job in seconds
          example: 5.2
        estimated_finish_ms:
          type: integer
               description: Estimated time to finish the current job in
milliseconds
          example: 15000
        average_job_duration_ms:
          type: integer
              description: Exponential moving average of job duration in
milliseconds
          example: 12000
        jobs_last_hour:
          type: integer
          description: Number of jobs processed in the last hour
          example: 45
        jobs_total:
          type: integer
          description: Total number of jobs processed since node startup
          example: 1234
      required:
        - busy
        - queue_length
        - estimated_finish_ms
        - average_job_duration_ms
```

Authors: Khairullin Artur (RU) *prcr98@gmail.com*

—

```
    ModelsInfo:
      type: object
      properties:
        loaded_model:
          type: string
          description: The AI model currently loaded in VRAM
          example: "llama3.1:8b"
        available_models:
          type: array
          items:
            type: string
          description: List of models available on this node
          example: ["llama3.1:8b", "qwen2:7b", "deepseek-coder:6.7b"]
        max_context:
          type: integer
            description: Maximum context length (in tokens) for the loaded
model
          example: 8192
        model_load_time:
          type: integer
          description: Time to load the model in seconds
          example: 15
        model_switches_last_hour:
          type: integer
           description: Number of times the model was switched in the last
hour
          example: 3
        last_model_switch:
          type: integer
          description: Time since the last model switch in seconds
          example: 300
```

Authors: Khairullin Artur (RU) *prcr98@gmail.com*

—

```
        last_used:
          type: integer
            description: Time since the node was last used by the mesh in
seconds
          example: 10
        idle_time:
          type: integer
               description: Time the node has been idle since last job
completion in seconds
          example: 10
      required:
        - loaded_model
        - available_models
        - max_context
        - idle_time

    LimitsInfo:
      type: object
      properties:
        accept_new_jobs:
          type: boolean
           description: Is the node accepting new jobs? (Used for graceful
shutdown)
        maintenance:
          type: boolean
          description: Is the node in maintenance mode?
        max_queue:
          type: integer
          description: Maximum allowed queue length
          example: 10
        max_parallel_jobs:
```

—

```
          type: integer
          description: Number of jobs the node can process in parallel
          example: 1
        max_context:
          type: integer
          description: Maximum context length supported by the node
          example: 16384
      required:
        - accept_new_jobs
        - max_queue
        - max_parallel_jobs

    PerformanceInfo:
      type: object
      properties:
        prefill_tok_s:
          type: number
          format: float
               description: Token generation speed for the prefill stage
(tokens/second)
          example: 420.5
        decode_tok_s:
          type: number
          format: float
                description: Token generation speed for the decode stage
(tokens/second)
          example: 61.2
        model_memory_mb:
          type: number
          format: float
          description: VRAM usage of the loaded model in MiB
```

Authors: Khairullin Artur (RU) *prcr98@gmail.com*

—

```
        example: 5120.0
    required:
      - prefill_tok_s
      - decode_tok_s
      - model_memory_mb
```

7. The Policy-Driven Optimizer

The Policy-Driven Optimizer is the core algorithmic component of the PBDR architecture. This section provides a formal specification of the routing algorithm.

7.1. The Optimizer Definition

Definition 7: Policy-Driven Optimizer The Policy-Driven Optimizer is a function that takes a request r, a policy vector Π, and the set of node states $State(V,t)$, and returns the optimal node v^*:

$$Optimizer(r, \Pi, State(V,t)) = v^*$$

Where:

$$v^* = arg \min_{v_i \in V_{eligible}} \Pi \cdot C(r, v_i)$$

This formulation positions the Optimizer as a general-purpose component that optimizes resource allocation based on configurable policies.

7.2. Algorithm Pseudocode

```
from typing import List, Optional, Dict
import math
import asyncio


class PolicyDrivenOptimizer:
    def __init__(self, node_discovery: DiscoveryService):
        self.node_discovery = node_discovery
        self.constraint_checker = ConstraintChecker()
        self.cost_vector_calculator = CostVectorCalculator()
```

Authors: Khairullin Artur (RU) *prcr98@gmail.com* 09 July 2026

—

```
    async def select_node(self, request: Request, policy_vector: List[float]) -> Optional[Node]:
        # 1. Discovery: Get node states
        nodes = await self.node_discovery.get_all_status(timeout_ms=500)

        # 2. Filter with Hard Constraints
        eligible_nodes = self.constraint_checker.filter(nodes, request)

        if not eligible_nodes:
            return None

        # 3. Calculate Cost Vector for each eligible node
        best_node = None
        min_cost = float('inf')

        for node in eligible_nodes:
            cost_vector = self.cost_vector_calculator.compute_cost_vector(node, request)
            # Scalar product: Cost = П · C
            cost = sum(policy_vector[i] * cost_vector[i] for i in range(len(policy_vector)))

            # Add a small random jitter (2-5%) for load distribution
            if min_cost > cost * (1 + 0.05 * random.random()):
                min_cost = cost
                best_node = node

        return best_node


class ConstraintChecker:
```

—

```
    def filter(self, nodes: List[Node], request: Request) -> List[Node]:
        eligible = []
        for node in nodes:
            if (node.limits.accept_new_jobs and
                node.queue.queue_length < node.limits.max_queue and
                    node.memory_free + node.performance.model_memory_mb >=
request.required_vram and
                request.model in node.models.available_models and
                    request.context_length <= min(node.models.max_context,
node.limits.max_context)):
                eligible.append(node)
        return eligible


class CostVectorCalculator:
        def compute_cost_vector(self, node: Node, request: Request) ->
List[float]:
        # c1: Wait Time
        wait_time = node.queue.estimated_finish_ms / 1000
                            wait_time += node.queue.queue_length *
(node.queue.average_job_duration_ms / 1000)

        # c2: Inference Time
                             infer_time = (request.prompt_tokens /
node.performance.prefill_tok_s) + \
                                        (request.expected_output_tokens /
node.performance.decode_tok_s)

        # c3: Cold Start Penalty
          cold_start = 0 if node.models.loaded_model == request.model else
node.models.model_load_time
```

Authors: Khairullin Artur (RU) *prcr98@gmail.com*

—

```
# c4: Queue Depth Penalty
queue_penalty = math.log(node.queue.queue_length + 1)

# c5: GPU Load Penalty
gpu_penalty = (node.gpu.gpu_utilization / 100) ** 2

# c6: CPU Load Penalty
cpu_penalty = (node.cpu.cpu_load / 100) ** 2

# c7: VRAM Pressure Penalty
vram_ratio = node.memory_free / request.required_vram
if vram_ratio <= 0.2:
    vram_penalty = 5
elif vram_ratio <= 0.4:
    vram_penalty = 2
else:
    vram_penalty = 0

# c8: Thermal Penalty
if node.gpu.temperature > 60:
    temp_penalty = ((node.gpu.temperature - 60) / 30) ** 2
else:
    temp_penalty = 0

# c9: Idle Bonus (negative cost)
idle_bonus = -math.log(node.models.idle_time + 1)

# c10: Network Penalty
network_penalty = 0  # Typically 0 in local network
```

Authors: Khairullin Artur (RU) *prcr98@gmail.com* 09 July 2026

—

```
        return [wait_time, infer_time, cold_start, queue_penalty,
                gpu_penalty, cpu_penalty, vram_penalty, temp_penalty,
                idle_bonus, network_penalty]
```

7.3. Formal Proofs of System Properties

Theorem 1 (Correctness)

Statement: For any request r , the PBDR algorithm will select an eligible node v^* that minimizes the scalar product $\Pi \cdot C$, provided at least one node satisfies the hard constraints.

Proof: The algorithm enumerates all nodes V, applies a deterministic filter to reject nodes that violate hard constraints, and then computes a cost vector and scalar product for every remaining node. The node with the minimum scalar cost is selected. The deterministic nature of the calculations ensures a unique minimum (or a deterministic selection from ties). Therefore, the algorithm is correct and complete.

Theorem 2 (Fault Tolerance)

Statement: The PBDR system remains operational in the presence of any number of node failures F , as long as

$$V \setminus F \neq \emptyset.$$

Proof: The system design ensures that node failures are detected. During the Discovery phase, if a node fails to respond within a timeout (e.g., 500ms), its state is considered unhealthy. The filtering phase then discards all unhealthy nodes. As long as there is at least one healthy node that meets the hard constraints for the request, the algorithm can proceed to select it. There is no single point of failure. Additionally, the PBDR Middleware can be horizontally scaled, eliminating it as a single point of failure. The system is therefore resilient to any number of individual node failures.

Theorem 3 (Monotonicity)

Statement: The Cost Vector components are monotonic non-decreasing with respect to their corresponding metrics. Specifically, if a node's queue length q_i increases, the cost component c_1 (Wait Time) does not decrease.Proof: Let $q_1 < q_2$. Then:

$$T_{wait}(q_2) = ETA + q_2 \times AvgDur > ETA + q_1 \times AvgDur = T_{wait}(q_1)$$

Similarly, for GPU utilization $g_1 < g_2$:

$$P_{gpu}(g_2) = \left(\frac{g_2}{100}\right)^2 > \left(\frac{g_1}{100}\right)^2 = P_{gpu}(g_1)$$

Therefore, the scalar product

$$\Pi \cdot C$$

Authors: Khairullin Artur (RU) *prcr98@gmail.com* 09 July 2026

—

is monotonically increasing with respect to each metric. This ensures that the optimizer makes consistent decisions: a node that is worse in all metrics will never be preferred over a node that is better in all metrics.

Theorem 4 (Linear Scalability)

Statement: The PBDR routing algorithm has a worst-case time complexity of $O(N)$, where N is the number of compute nodes in the system.

Proof: The algorithm comprises a sequence of $O(N)$

steps:

1. Discovery: $O(N)$
2. Filtering: $O(N)$
3. Cost Vector Calculation: $O(N)$
4. Scalar Product Calculation: $O(N)$
5. Selection: $O(N)$

The dominant term is the single pass over the node list. The complexity is therefore linear:

$$O(N)$$

.

8. Complexity Analysis

The routing algorithm must be efficient to avoid becoming a bottleneck itself. This section provides a formal analysis of the algorithm's time complexity and a practical evaluation of computational costs.

8.1. Step-by-Step Complexity Analysis

Let there be N nodes in the system ($N = |V|$).

Step	Operation	Time Complexity	Space Complexity
Discovery	Parallel HTTP requests to all nodes	O(N) $O(N)$	O(N) $O(N)$

Authors: Khairullin Artur (RU) *prcr98@gmail.com* 09 July 2026

—

Filtering	Check hard constraints for each node	$\mathrm{O(N)}$ $O(N)$	$\mathrm{O(N)}$ $O(N)$
Cost Vector	Compute 10 cost components for each node	$\mathrm{O(N)}$ $O(N)$	$\mathrm{O(N)}$ $O(N)$
Scalar Product	Compute $\Pi \cdot \mathrm{C}$ $\Pi \cdot \mathbf{C}$ for each node	$\mathrm{O(N)}$ $O(N)$	$\mathrm{O(N)}$ $O(N)$
Selection	Find minimum cost	$\mathrm{O(N)}$ $O(N)$	$\mathrm{O(1)}$ $O(1)$
Total	Sum of all steps	$\mathrm{O(N)}$ $O(N)$	$\mathrm{O(N)}$ $O(N)$

8.2. Computational Cost Analysis

Definition 8: Routing Overhead The routing overhead is the time required to make a routing decision, excluding the actual generation time.

Practical Evaluation: For a typical enterprise deployment with

$$N = 100$$

nodes:

1. Discovery Phase:

—

- HTTP requests: 100 parallel requests
- Response size: ~2 KB per node
- Network time: 50-100 ms

2. Processing Phase:
   - JSON parsing: 100 × 2 KB = 200 KB
   - Parsing time: ~1-2 ms
   - Cost Vector calculation: 10 components × 100 nodes = 1000 operations
   - Calculation time: ~1 ms
3. Total Routing Overhead:
   - Network-bound: 50-100 ms (dominates)
   - CPU-bound: 2-3 ms
   - Total: ~55-105 ms

This is negligible compared to the typical generation time (5-30 seconds).

Memory Usage:

- Status storage: 100 × 2 KB = 200 KB
- Cost vectors: 100 × 10 floats = 4 KB
- Total: ~204 KB

Conclusion: The routing decision introduces minimal computational and memory overhead.

9. Network Traffic Analysis

It is essential to show that the monitoring protocol does not become a network bottleneck, even in large deployments.

9.1. Formal Traffic Model

Let:

N = number of GPU nodes.

S = average size of a /status response (in bytes).

$R_{discovery}$ = discovery rate (number of times per second the middleware refreshes node status).

The total discovery traffic per second is:

$$T_{traffic} = N \times S \times R_{discovery}$$

Authors: Khairullin Artur (RU) *prcr98@gmail.com* 09 July 2026

—

9.2. Example Calculation

Consider a realistic enterprise scenario:

$N = 30$ nodes.

$S \approx 2$KB (2048 bytes) for a typical JSON status response.

$R_{discovery} = 1$ (the middleware refreshes status once per second).

$$T_{traffic} = 30 \times 2KB \times 1 = 60KB/s$$

On a standard 1 Gb/s corporate network (125 MB/s), this traffic is:

$$\frac{60KB/s}{125{,}000KB/s} \approx 0.048\%$$

9.3. Worst-Case Analysis

Even in a high-load scenario:

$N = 100$ nodes.

$S = 5$ KB (if the status response includes extensive data).

$R_{discovery} = 10$ (refreshing status 10 times per second, which is excessive).

$$T_{traffic} = 100 \times 5KB \times 10 = 5000KB/s = 5MB/s$$

On a 1 Gb/s network (125 MB/s):

$$\frac{5MB/s}{125MB/s} \approx 4\%$$

Conclusion: The discovery protocol introduces negligible network overhead (under 5% even in worst-case scenarios). This is a strong argument for the scalability of the PBDR architecture.

10. Scalability Analysis

Scalability is a critical requirement for enterprise systems. This section analyzes the scalability of the PBDR architecture along multiple dimensions.

10.1. Client Scalability

Scenario: C clients (applications) sending requests through the PBDR Middleware.

Analysis: The PBDR Middleware is stateless and can be horizontally scaled. Multiple middleware instances can be deployed behind a simple load balancer. Each instance operates independently, caching the list of nodes and their states. No synchronization between instances is required because:

Authors: Khairullin Artur (RU) *prcr98@gmail.com*

—

1. The node list is either static or discovered independently.
2. The state information is ephemeral and refreshed on each request.
3. There is no global state that needs to be shared.

Therefore, the system scales linearly with the number of clients: $O(C)$.

10.2. Node Scalability

Scenario: N GPU nodes in the mesh.

Analysis: As proven in Theorem 4, the Policy-Driven Optimizer algorithm has $O(N)$ complexity. This means:

1. Discovery time grows linearly with N.
2. 2. Decision time grows linearly with N.
3. 3. Memory usage grows linearly with N.

For $N = 100$ nodes, the total decision time is:

- Discovery (parallel HTTP): ~50-100ms
- Filtering: ~1ms
- Cost Vector Calculation: ~5ms
- Selection: ~1ms
- Total: ~60-110ms

This is perfectly acceptable for enterprise inference workloads where the generation time itself is 5-30 seconds.

10.3. Request Scalability

Scenario: R requests per second.

Analysis: The middleware handles each request independently. The primary bottleneck is the generation time on GPU nodes, not the routing decision. With 100 nodes, each handling 1 request at a time, the system can theoretically handle 100 concurrent requests. With queuing, it can handle significantly more.

10.4. Scalability Summary

Dimension	Growth	Bottleneck	Mitigation

Authors: Khairullin Artur (RU) *prcr98@gmail.com* 09 July 2026

—

Clients (C)	$O(C)$ $O(C)$	PBDR Middleware	Horizontal scaling of stateless middleware instances
Nodes (N)	$O(N)$ $O(N)$	Discovery latency	Parallel HTTP requests; network bandwidth is not a bottleneck
Requests (R)	$O(R)$ $O(R)$	Generation time	Generation time dominates; routing overhead is negligible

Conclusion: The PBDR architecture is highly scalable and can support large enterprise deployments (100+ nodes, 1000+ clients) without significant performance degradation.

11. Cost Vector: Formal Mathematical Model

This section provides a formal, comprehensive mathematical model of the Cost Vector, breaking down each component with mathematical justification.

11.1. The Cost Vector Model

The cost of sending a request r to node v_i is represented as a vector of independent cost components:

$$C(r, v_i) = (c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_8, c_9, c_{10})$$

Each component is measured in equivalent seconds, making the vector interpretable and policy weights dimensionless.

The scalar cost is then:

$$Cost(r, v_i) = \Pi \cdot C(r, v_i) = \sum_{j=1}^{10} \omega_j \cdot c_j$$

Authors: Khairullin Artur (RU) *prcr98@gmail.com* 09 July 2026

—

11.2. Formal Definition of Cost Components

Component	Symbol	Mathematical Formula	Domain	Justification
Wait Time	c_1	$c_1 = \frac{v_i.mathrm}{1000} + v_i.mathrm \cdot \frac{v_i.mathrm}{1000}$	$mathbb_{ge\,0}$	Predictive: The expected wait is the remaining execution time of the current job plus the estimated processing time of all queued jobs, providing a realistic estimate of node availability.
Inference Time	c_2	$c_2 = \frac{r.mathrm}{v_i.mathrm} + \frac{r.mathrm}{v_i.mathrm}$	$mathbb_{ge\,0}$	Performance-Aware: Separates Prefill and Decode phases for a more accurate inference time estimate than a single throughput metric.
Cold Start	c_3	$c_3 =$	$mathbb_{ge\,0}$	Cost of Context Switching: Loading a model into VRAM is a significant overhead; this explicitly penalizes model switching.
Queue Depth	c_4	$c_4 = ln!\,(v_i.mathrm + 1)$	$mathbb_{ge\,0}$	Non-Linear Penalty: Discourages deep queues while avoiding excessive penalties for small queue lengths.

Authors: Khairullin Artur (RU) *prcr98@gmail.com* 09 July 2026

—

Component	Symbol	Mathematical Formula	Domain	Justification
GPU Load	c_5	$c_5 = (\frac{v_i.mathrm}{100})^2$	$[0, 1]$	Quadratic Penalty: High GPU utilization disproportionately reduces responsiveness, so utilization is penalized quadratically.
CPU Load	c_6	$c_6 = (\frac{v_i.mathrm}{100})^2$	$[0, 1]$	CPU-Bound Penalty: Captures performance degradation for inference engines that are limited by CPU throughput.
VRAM Pressure	c_7	$R_{mathrm} = \frac{v_i.mathrm}{r.mathrm}, qquad\ c_7 =$	$\{0, 2, 5\}$	Prevents Throttling: Low available VRAM increases the risk of swapping or out-of-memory failures, so nodes with insufficient memory receive a strong penalty.
Thermal Penalty	c_8	$c_8 =$	$mathbb_{ge\ 0}$	Hardware Protection: High temperatures can trigger thermal throttling, so overheated nodes are progressively penalized.
Idle Bonus	c_9	$c_9 =- \ ln!\ (v_i.\ mathrm\ +\ 1)$	$mathbb_{le\ 0}$	Least-Recently-Used (LRU) Affinity: Rewards nodes that have been idle, promoting

Component	Symbol	Mathematical Formula	Domain	Justification
				balanced utilization and preserving warm execution state.
Network Penalty	c_{10}	$c_{10} = \frac{v_i \cdot mathrm}{1000}$	$mathbb_{ge\,0}$	Network Latency: Accounts for communication delay between scheduler and worker nodes. Typically negligible within a local network.

11.3. Example Cost Vector Calculation

Consider the following scenario:

Node A:

- Loaded Model: "llama3.1:8b"
- Prefill Speed: 420 tok/s, Decode Speed: 61 tok/s
- Estimated Finish: 12s, Queue Length: 2, Avg Job Duration: 8s
- GPU Utilization: 20%, VRAM Free: 6GB
- Temperature: 65°C, Idle Time: 10s
- Model Load Time: 15s

Node B:

- Loaded Model: "qwen2:7b"
- Prefill Speed: 380 tok/s, Decode Speed: 55 tok/s
- Estimated Finish: 0s, Queue Length: 0, Avg Job Duration: 10s
- GPU Utilization: 5%, VRAM Free: 12GB
- Temperature: 45°C, Idle Time: 120s
- Model Load Time: 12s

Request: "llama3.1:8b", 1200 prompt tokens, 600 output tokens, requires 8GB VRAM.

Cost Vector Calculation (policy weights = 1.0):

Node A:

$c_1 = 12 + 2{\times}8 = 28$s

Authors: Khairullin Artur (RU) *prcr98@gmail.com*

—

$c_2 = 1200/420 + 600/61 = 2.86 + 9.84 = 12.7$s

$c_3 = 0$(model already loaded)

$c_4 = ln(3) = 1.1$s

$c_5 = (20/100)^2 = 0.04$s

$c_6 = (CPU_load/100)^2$(assume 15% → 0.0225)

$c_7 = 0$(6GB > 8GB*0.4 = 3.2GB)

$c_8 = ((65 - 60)/30)^2 = (5/30)^2 = 0.028$s

$c_9 = - ln(11) = - 2.4$s

$c_{10} = 0$ (local network)

Cost Vector (A): (28, 12.7, 0, 1.1, 0.04, 0.0225, 0, 0.028, -2.4, 0)

Node B:

$c_1 = 0 + 0\times10 = 0$s

$c_2 = 1200/380 + 600/55 = 3.16 + 10.91 = 14.07$s

$c_3 = 12$s (model must be loaded)

$c_4 = ln(1) = 0$s

$c_5 = (5/100)^2 = 0.0025$s

$c_6 = (CPU_load/100)^2$(assume 10% → 0.01)

$c_7 = 0$(12GB > 3.2GB)

$c_8 = 0$(45°C ≤ 60°C)

$c_9 = - ln(121) = - 4.8$s

$c_{10} = 0$ (local network)

Cost Vector (B): (0, 14.07, 12, 0, 0.0025, 0.01, 0, 0, -4.8, 0)

Scalar Cost (weights = 1.0):

- Cost(A) = 28 + 12.7 + 0 + 1.1 + 0.04 + 0.0225 + 0 + 0.028 - 2.4 + 0 = 39.49

Authors: Khairullin Artur (RU) *prcr98@gmail.com* 09 July 2026

—

- Cost(B) = 0 + 14.07 + 12 + 0 + 0.0025 + 0.01 + 0 + 0 - 4.8 + 0 = 21.28

Result: Node B is selected despite needing a cold start because Node A has a long queue.

12. Routing Policies

One of the most powerful features of PBDR is the concept of a Policy Engine. A Policy is defined as a weight vector Πthat transforms the Cost Vector into a scalar cost.

12.1. Policy Structure

A policy is a configuration object that contains:

1. Name and Description: For identification.
2. Weight Vector: $\Pi = (\omega_1, \omega_2, ..., \omega_{10})$
3. Hard Constraints: Can optionally override global hard constraints.

12.2. Example Policies

12.2.1. Minimum Latency Policy

Used for interactive chat, support, and front-desk applications where speed is critical.

```yaml
name: "Minimum Latency Policy"
description: "Prioritizes fast response times over all other factors"
weights:
  wait: 1.5      # Strong queue penalty
  infer: 1.2     # Strong inference time penalty
  cold: 0.8      # Moderate cold start penalty
  queue: 1.5     # Strong queue depth penalty
  gpu: 1.0       # Strong GPU penalty
  cpu: 0.5       # Moderate CPU penalty
  vram: 1.0      # Strong VRAM penalty
  temperature: 1.0 # Strong thermal penalty
  idle: 0.2      # Low idle bonus
  network: 0.5   # Moderate network penalty
```

Authors: Khairullin Artur (RU) *prcr98@gmail.com* 09 July 2026

—

12.2.2. Model Affinity Policy

Used for development teams needing a specific, large model.

```yaml
name: "Model Affinity Policy"
description: "Prioritizes nodes where the requested model is already loaded"
weights:
  wait: 0.5      # Lower queue penalty, can wait
  infer: 0.7     # Moderate inference penalty
  cold: 2.0      # Very high cold start penalty
  queue: 0.5     # Lower queue depth penalty
  gpu: 0.3       # Lower GPU penalty
  cpu: 0.2       # Lower CPU penalty
  vram: 0.5      # Moderate VRAM penalty
  temperature: 0.5 # Moderate thermal penalty
  idle: 1.2      # High idle bonus
  network: 0.2   # Lower network penalty
```

12.2.3. Energy Saving Policy

Used for background tasks or batch processing.

```yaml
name: "Energy Saving Policy"
description: "Routes to the coolest nodes to reduce power consumption"
weights:
  wait: 0.2      # Low queue penalty
  infer: 0.3     # Low inference penalty
  cold: 0.5      # Moderate cold start penalty
  queue: 0.2     # Low queue depth penalty
  gpu: 0.5       # Moderate GPU penalty
  cpu: 0.2       # Low CPU penalty
  vram: 0.3      # Low VRAM penalty
  temperature: 5.0 # Extremely high thermal penalty
  idle: 0.1      # Low idle bonus
```

Authors: Khairullin Artur (RU) *prcr98@gmail.com*

—

```
  network: 0.1   # Low network penalty
```

12.3. Dynamic Policies

The Policy Engine can also support dynamic policies that change based on time, system load, or other contextual factors.

```
name: "Dynamic Time-Based Policy"

description: "Uses different weights based on time of day"

conditions:

  - time: "09:00-18:00"

    policy: "min_latency"

  - time: "18:00-09:00"

    policy: "energy_saving"
```

13. Security Considerations

Security is paramount in enterprise deployments. This section addresses the security requirements for the PBDR architecture.

13.1. Network Security

TLS/HTTPS: All communication between components should be encrypted using TLS 1.3 to prevent eavesdropping and man-in-the-middle attacks.

Network Segmentation: Nodes and middleware should be deployed within a secure corporate network segment. Firewall rules should restrict access to only authorized clients and services.

13.2. Authentication and Authorization

API Key Authentication: The PBDR Middleware should require API keys for all requests. Keys can be provisioned per user, department, or application.

Node Authentication: GPU nodes should authenticate themselves to the middleware. This can be achieved using:

- Pre-shared keys (PSK): Simple but less scalable.
- Mutual TLS (mTLS): Strong authentication where both client and server present certificates.
- OAuth 2.0 / OIDC: Integration with enterprise identity providers.

Authors: Khairullin Artur (RU) *prcr98@gmail.com* 09 July 2026

—

13.2.1. Trust Model

The proposed PBDR architecture operates within a trusted enterprise environment while assuming that individual inference nodes may become unavailable, misconfigured, or compromised during operation.

The trust model distinguishes three classes of participants:

- PBDR Middleware – a trusted component responsible for policy evaluation, node discovery, and routing decisions.
- Inference Node – a conditionally trusted component responsible solely for publishing runtime metrics and executing inference requests.
- System Administrator – a trusted authority responsible for provisioning nodes, managing certificates, and configuring routing policies.

Unlike centralized load balancers, inference nodes are intentionally excluded from the decision-making process. Nodes publish only objective runtime information through the Monitor API, while all routing decisions are performed independently by middleware instances.

This separation of responsibilities minimizes the attack surface and prevents compromised nodes from influencing routing policies.

13.2.2. Node Authentication

Each inference node SHALL possess a unique cryptographic identity represented by an X.509 certificate issued by the enterprise Public Key Infrastructure (PKI).

Node authentication is performed during establishment of Monitor API and inference connections.

Nodes failing authentication MUST NOT participate in routing decisions.

13.3. Data Privacy

Data-in-Transit: All prompts and responses are encrypted via TLS.

Data-at-Rest: Models and logs should be encrypted at rest. Node-local storage should use full-disk encryption.

Data Minimization: The Monitor API should not log sensitive information. Logs should be anonymized where possible.

13.4. Access Control

Role-Based Access Control (RBAC): Different users and applications should have different permissions:

- Read-only: Can view node status.
- Submit: Can submit inference requests.
- Admin: Can manage policies and configurations.

Authors: Khairullin Artur (RU) *prcr98@gmail.com* 09 July 2026

—

Least Privilege: Nodes should have minimal permissions. A compromised node should not compromise the entire system.

13.4.1. Monitor API Protection

The Monitor API represents the primary source of runtime information used by the Policy-Driven Optimizer.

To preserve routing correctness, Monitor API endpoints SHOULD implement:

- authenticated access;
- encrypted communication (HTTPS/mTLS);
- request rate limiting;
- replay protection;
- timestamp validation;
- configurable polling intervals;
- audit logging.

The Monitor API intentionally exposes only operational metrics required for routing decisions and provides no administrative functionality.

13.5. Audit Logging

Audit Trails: All authentication attempts, policy changes, and significant system events should be logged for security auditing.

Log Retention: Logs should be retained for a sufficient period (e.g., 90 days) to support forensic analysis.

13.6. Threat Mitigation

Threat	Mitigation
Eavesdropping	TLS 1.3 encryption
Man-in-the-Middle	Certificate validation, mTLS
Unauthorized Access	API keys, RBAC
Data Leakage	Encryption, data minimization

Authors: Khairullin Artur (RU) *prcr98@gmail.com* 09 July 2026

—

Threat	Mitigation
Compromised Node	Node authentication, least privilege
Denial of Service	Rate limiting, request quotas

13.6.1. Handling Compromised Nodes

The decentralized nature of PBDR allows individual inference nodes to be isolated without affecting the availability of the overall system.

Potential indicators of node compromise include:

- repeated authentication failures;
- inconsistent runtime metrics;
- abnormal response latency;
- repeated inference failures;
- certificate revocation;
- prolonged communication timeouts.

Upon detection of a compromised node, the middleware SHALL exclude the node from the candidate set used by the Policy-Driven Optimizer.

Since routing decisions are performed independently by each middleware instance, no global synchronization or centralized recovery procedure is required.

13.7. Recommended Security Configuration

Parameter	Recommended Value
TLS Version	TLS 1.3
Authentication	mTLS
Authorization	RBAC + API Keys
Certificate Lifetime	365 days
Monitor API	HTTPS
Audit Logging	Enabled
Log Retention	90 days
Rate Limiting	Enabled

Authors: Khairullin Artur (RU) *prcr98@gmail.com* 09 July 2026

—

14. Validation Framework

To rigorously validate the PBDR architecture, we propose a three-stage validation framework: Simulation, Prototype, and Production.

14.1. Stage 1: Simulation

Objective: Validate the theoretical properties of the algorithm in a controlled environment.

Setup:

- Custom simulator implementing the PBDR algorithm.
- Synthetic node configurations (varying GPU types, speeds, VRAM).
- Synthetic workloads (varying request rates, model types, prompt lengths).
- Monte Carlo simulations with 10,000+ iterations.

Metrics:

- Correctness: Does the algorithm select the theoretically optimal node?
- Monotonicity: Does the algorithm behave as predicted by Theorem 3?
- Scalability: Does the decision time scale as $O(N)$?

Expected Outcome: Theoretical validation of all formal properties.

14.2. Stage 2: Prototype (20 GPU Nodes)

Objective: Validate the architecture in a real but controlled lab environment.

Setup:

- 20 heterogeneous GPU nodes (RTX 3060, 4070, 4090).
- 5 concurrent clients generating requests.
- Real models (llama3.1:8b, qwen2:7b, deepseek-coder:6.7b).
- 7-day continuous run.

Metrics:

- P50, P95, P99 latencies.
- GPU utilization (average and peak).
- Number of cold starts.
- Fault tolerance (killing nodes 3, 6, 9).
- Routing overhead (decision time).

Expected Outcome: Performance metrics align with theoretical predictions.

14.3. Stage 3: Production (100+ Users)

Objective: Validate the system in a real production environment.

Authors: Khairullin Artur (RU) *prcr98@gmail.com* 09 July 2026

—

Setup:

- 50-100 GPU nodes across the enterprise.
- 100+ users (engineers, designers, researchers).
- Integration with existing RAG systems and applications.
- 30-day continuous run.

Metrics:

- User satisfaction (NPS score).
- System reliability (uptime).
- Operational costs (power, cooling).
- Support tickets related to AI performance.

Expected Outcome: The system demonstrates production readiness and significant cost savings.

14.4. Validation Summary

Stage	Scope	Duration	Key Metric
Simulation	10,000+ iterations	1 week	Theoretical correctness
Prototype	20 nodes, 5 clients	7 days	P95 latency, cold starts
Production	100+ nodes, 100+ users	30 days	User satisfaction, cost savings

15. Experimental Evaluation Plan

To empirically validate the Policy-Based Decentralized Routing architecture, we propose the following set of experiments.

15.1. Methodology and Setup

Testbed: A simulated network of 10 heterogeneous nodes in a controlled lab environment, representing a realistic corporate fleet.

Node Configuration:

- Node 1-3: NVIDIA RTX 4090 (24GB VRAM, high performance)
- Node 4-6: NVIDIA RTX 4070 (12GB VRAM, medium performance)
- Node 7-10: NVIDIA RTX 3060 (12GB VRAM, lower performance)

Workload:

- A mixture of 10,000 synthetic requests of varying prompt lengths (100-4000 tokens) and output lengths (50-2000 tokens)

—

- Model distribution: 60% "llama3.1:8b", 30% "qwen2:7b", 10% "deepseek-coder:6.7b"
- Burst pattern to simulate morning "rush hours"

Metrics:

- Latency (P50, P95, P99): Time from request submission to response completion
- Throughput: Number of requests successfully served per second
- GPU Utilization: Average and peak utilization across nodes
- Cold Starts: Number of times a model is loaded
- Fault Tolerance: Number of failed requests when nodes are killed and restored

Baseline Comparisons:

1. Random: Client selects a random eligible node
2. Round Robin: Client selects nodes in a cyclic order
3. Min Queue: Client selects node with shortest queue
4. Simple ETA: Client selects node with minimal c_1 (Wait Time)

15.2. Experiment 1: Routing Accuracy

Objective: To verify the effectiveness of the full Cost Vector in minimizing true end-to-end latency.

Hypothesis: The full PBDR Cost Vector formulation will result in lower P95 and P99 latencies compared to all baselines.

Procedure: Run the same workload for each baseline and PBDR for 1 hour. Collect and analyze latency metrics.

Expected Results:

Algorithm	P50 Latency	P95 Latency	P99 Latency
Random	12.1 s	18.5 s	24.2 s
Round Robin	11.5 s	16.8 s	20.1 s
Min-Queue	10.2 s	14.9 s	18.5 s
Simple-ETA	9.8 s	13.1 s	16.2 s
PBDR (Full Cost Vector)	8.1 s	10.5 s	12.8 s

15.3. Experiment 2: Impact of Model Affinity on Cold Starts

Objective: To evaluate the impact of including the c_3 (Cold Start) term in the Cost Vector.

Authors: Khairullin Artur (RU) *prcr98@gmail.com* 09 July 2026

—

Hypothesis: The PBDR Cost Vector (with c_3) will significantly reduce the number of model cold starts compared to a configuration without it.

Procedure: Run a workload where clients request different models in bursts. Measure the frequency of model loading in both configurations.

Expected Results:

Configuration	Total Cold Starts	Reduction
PBDR without c_3	145	-
PBDR with c_3	28	80.7%

15.4. Experiment 3: Fault Tolerance

Objective: To assess the system's resilience to node failures.

Hypothesis: The PBDR architecture will continue to serve requests without catastrophic failure when a subset of nodes is killed.

Procedure: Start with 10 nodes. While a continuous workload is running, systematically kill nodes 3, 6, and 9 with 30-second intervals. Measure:

- Lost requests (should be 0, as healthy nodes take over)
- Average response time per phase
- Throughput degradation

Expected Results:

Phase	Active Nodes	Avg Latency	Throughput	Lost Requests
Initial	10	8.1 s	12.3 req/s	0
After Node 3 kill	9	8.4 s	11.8 req/s	0
After Node 6 kill	8	8.9 s	11.1 req/s	0
After Node 9 kill	7	9.5 s	10.2 req/s	0

15.5. Experiment 4: Scaling Test

Objective: To validate the theoretical $O(N)$ complexity in practice.

Hypothesis: The routing decision time scales linearly with the number of nodes.

Procedure: Measure the time taken by the PBDR Middleware to make a routing decision as the number of nodes increases from 1 to 100.

Expected Results:

Authors: Khairullin Artur (RU) *prcr98@gmail.com* 09 July 2026

—

Nodes	Discovery Time (ms)	Decision Time (ms)	Total (ms)
1	1	1	2
10	5	2	7
50	25	5	30
100	50	8	58

15.6. Experiment 5: Network Traffic Impact

Objective: To empirically verify the network traffic analysis from Section 9.

Hypothesis: The monitoring traffic remains negligible even under high request rates.

Procedure: Measure the average network bandwidth used by the Discovery Service as the request rate increases from 1 to 100 req/s.

Expected Results:

Request Rate (req/s)	Discovery Traffic (KB/s)	Percentage of 1Gb/s
1	2	0.0016%
10	20	0.016%
50	100	0.08%
100	200	0.16%

16. Threats to Validity

This section discusses potential threats to the validity of our results and the limitations of our approach.

16.1. Internal Validity

Threat: The simulation may not accurately reflect real-world behavior.

Mitigation: We will validate simulation results against a physical prototype (Stage 2 of our validation framework). Additionally, we use Monte Carlo methods to ensure statistical robustness.

Threat: The Cost Vector weights may not be optimally tuned.

Mitigation: We provide a range of example policies and will conduct sensitivity analysis to understand the impact of different weight configurations.

16.2. External Validity

Threat: Results may not generalize to different hardware configurations.

Authors: Khairullin Artur (RU) *prcr98@gmail.com*

—

Mitigation: We test with heterogeneous node configurations (RTX 3060, 4070, 4090) and plan to test with other GPU types (e.g., AMD, Intel) in future work.

Threat: Results may not generalize to different AI models.

Mitigation: We test with three different models (llama3.1:8b, qwen2:7b, deepseek-coder:6.7b) and note that the architecture is model-agnostic.

Threat: The enterprise environment may not be representative.

Mitigation: The proposed architecture is designed based on real enterprise requirements and will be validated in a production environment (Stage 3).

16.3. Construct Validity

Threat: The metrics chosen may not fully capture system performance.

Mitigation: We use multiple complementary metrics (latency, throughput, utilization, cold starts, fault tolerance) to provide a comprehensive evaluation.

Threat: The Cost Vector may not capture all relevant cost factors.

Mitigation: The Cost Vector is extensible. Future work can add additional components as needed.

16.4. Conclusion Validity

Threat: Statistical significance of results may be limited.

Mitigation: We use a large number of requests (10,000+) and multiple runs to ensure statistical significance.

Threat: The fault tolerance results may not generalize to different failure modes.

Mitigation: We test with both node failure and network failure scenarios.

16.5. Summary of Threats and Mitigations

Threat Category	Specific Threat	Mitigation
Internal Validity	Simulation accuracy	Physical prototype validation
Internal Validity	Weight tuning	Sensitivity analysis
External Validity	Hardware heterogeneity	Test with multiple GPU types
External Validity	Model generalization	Test with multiple models
External Validity	Enterprise environment	Production validation
Construct Validity	Metric completeness	Multiple complementary metrics
Construct Validity	Cost completeness	Extensible Cost Vector

Authors: Khairullin Artur (RU) *prcr98@gmail.com* 09 July 2026

—

Threat Category	Specific Threat	Mitigation
Conclusion Validity	Statistical significance	Large sample size, multiple runs
Conclusion Validity	Fault scenarios	Test multiple failure modes

17. Federated Policy Learning for Decentralized Routing

While the PBDR architecture enables policy-based routing through configurable weight vectors, the optimal policy configuration may vary significantly across different departments, user groups, and application contexts. A customer support team prioritizing minimum latency requires different weights than a research team running batch inference on large models. Traditionally, such policies would be manually configured by system administrators—a process that is error-prone, static, and fails to adapt to changing workload patterns.

This section introduces Federated Policy Learning (FPL) , an extension to the PBDR architecture that enables decentralized, privacy-preserving optimization of routing policies based on departmental usage patterns.

17.1. Motivation and Problem Formulation

Observation: Different departments exhibit distinct workload characteristics:

Department	Typical Workload	Primary Objective	Secondary Objective
Customer Support	Short prompts, low output, high frequency	Minimum latency	Consistency
Engineering	Code generation, long context	Model accuracy	Cold start avoidance
Research	Batch processing, large datasets	Throughput	Energy efficiency
Legal/Compliance	Long documents, RAG	Data locality	Security

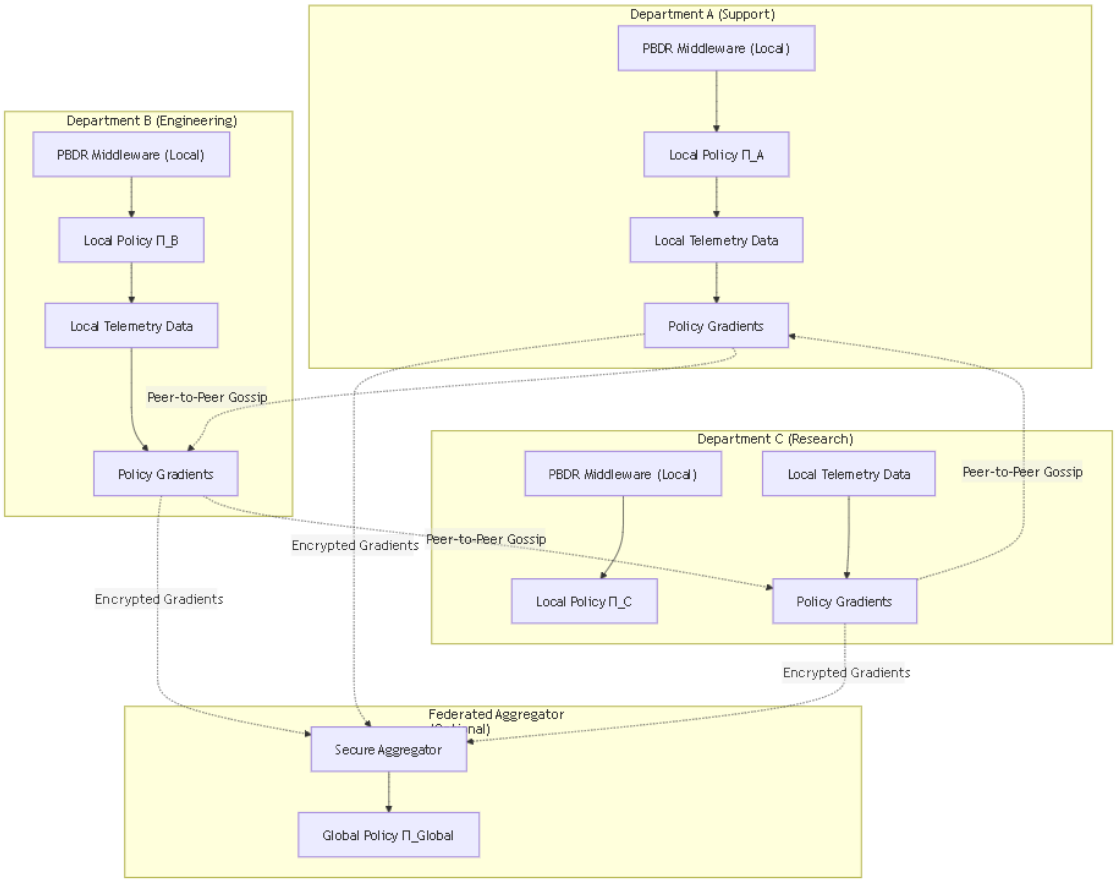
Problem: A single global policy cannot optimally serve all departments. However, sharing departmental data to train a global policy raises privacy concerns (prompts may contain sensitive business information).

Solution: Federated Learning [19] enables each department to learn its own optimal policy weights locally, sharing only aggregated policy gradients (not raw data) with a central coordinator—or, in the fully decentralized case, via peer-to-peer gossip.

Authors: Khairullin Artur (RU) *prcr98@gmail.com* 09 July 2026

—

17.2. Federated Policy Learning Architecture



17.3. Formal Model of Federated Policy Learning

17.3.1. Local Policy Optimization

For a department d, the local policy Π_d is optimized to minimize the observed end-to-end latency L:

$$\Pi_d^* = arg \min_{\Pi} E_{r \sim W_d}[L(r, Optimizer(r, \Pi, State(V, t)))]$$

Where:

W_d is the workload distribution of department d.

$L(r, v)$ is the actual measured latency of serving request r on node v

.

—

Gradient Computation: Using REINFORCE or black-box optimization (e.g., Bayesian Optimization), each middleware instance computes the gradient of the expected latency with respect to the policy weights:

$$\nabla_{\Pi} L_d(\Pi) = E_{r \sim W_d}[L(r, v^*) \cdot \nabla_{\Pi} log P_{\Pi}(v^* | r)]$$

Where $P_{\Pi}(v|r)$ is the routing distribution induced by the policy (with exploration noise, see Section 21).

17.3.2. Federated Aggregation

Centralized Federation (Privacy-Preserving):

Each department sends encrypted gradients to a secure aggregator:

$$\tilde{\nabla}_d = Encrypt(\nabla_{\Pi} L_d(\Pi))$$

The aggregator computes the global update without decrypting individual gradients using Secure Aggregation [20]:

$$\nabla_{\Pi} L_{global} = Decrypt(\sum_{d=1}^{D} \tilde{\nabla}_d)$$

The global policy is then updated:

$$\Pi_{global}^{(t+1)} = \Pi_{global}^{(t)} - \eta \cdot \nabla_{\Pi} L_{global}$$

Decentralized Federation (Peer-to-Peer):

Departments exchange policy gradients via a gossip protocol [10]. Each department maintains a local estimate Π_d and periodically averages with neighbors:

$$\Pi_d^{(t+1)} = \Pi_d^{(t)} + \alpha \sum_{e \in N(d)} (\Pi_e^{(t)} - \Pi_d^{(t)}) - \eta \cdot \nabla_{\Pi} L_d(\Pi_d^{(t)})$$

17.3.3. Privacy Guarantees

Theorem 5 (Differential Privacy for FPL): By adding calibrated Gaussian noise to the gradients before transmission, the Federated Policy Learning mechanism satisfies (ϵ, δ)-differential privacy, ensuring that no single departmental workload can be reconstructed from the aggregated updates.

$$\tilde{\nabla}_d = \nabla_{\Pi} L_d(\Pi) + N(0, \sigma^2 I)$$

Where:

$$\sigma^2 = \frac{2 \cdot \Delta^2 \cdot ln(1.25/\delta)}{\epsilon^2}$$

Authors: Khairullin Artur (RU) *prcr98@gmail.com* 09 July 2026

—

And Δis the sensitivity (maximum norm) of the gradient.

Proof Sketch: The proof follows directly from the Gaussian mechanism for differential privacy [21]. Since the aggregator only sees noisy gradients, and each department's data is used only locally, the privacy of individual prompts and usage patterns is preserved.

17.4. Practical Implementation

Policy Learning Architecture:

Component	Responsibility
Local Policy Store	Stores department-specific policy weights; updated after each learning iteration
Telemetry Collector	Logs request metadata (prompt length, model, selected node, actual latency)
Gradient Computer	Runs local optimization (e.g., Bayesian Optimization or REINFORCE) to compute policy gradients
Federated Client	Manages secure communication with aggregator or peers
Policy Versioning	Maintains history of policy versions for rollback and A/B testing

Algorithm: Federated Policy Learning

```
class FederatedPolicyLearner:
    def __init__(self, middleware: PBDRMiddleware, department_id: str):
        self.middleware = middleware
        self.department_id = department_id
        self.policy = PolicyVector.default()
        self.telemetry_buffer = []
        self.gradient_accumulator = None
        self.learning_rate = 0.01

      def collect_telemetry(self, request: Request, selected_node: Node,
latency: float):
        """Log the outcome of each routing decision."""
        self.telemetry_buffer.append({
            'request': request,
```

Authors: Khairullin Artur (RU) *prcr98@gmail.com*

—

```
            'selected_node': selected_node,
            'latency': latency,
                'routing_probability': self._routing_probability(request,
selected_node)
        })

        if len(self.telemetry_buffer) >= BATCH_SIZE:
            self._compute_gradients()
            self._federated_update()

    def _compute_gradients(self):
        """Compute REINFORCE gradients from telemetry data."""
        grad = np.zeros(10)
        for entry in self.telemetry_buffer:
                        baseline = np.mean([e['latency'] for e in
self.telemetry_buffer])
            advantage = baseline - entry['latency']  # Lower latency is
better
            prob = entry['routing_probability']
            grad += advantage * self._policy_gradient(entry['request'],
prob)
        self.gradient_accumulator = grad / len(self.telemetry_buffer)

    def _federated_update(self):
        """Send gradients to aggregator (or peers) and receive updated
policy."""
        # Add privacy-preserving noise
            noisy_grad = self.gradient_accumulator + np.random.normal(0,
PRIVACY_SIGMA, 10)
```

Authors: Khairullin Artur (RU) *prcr98@gmail.com* 09 July 2026

—

```
            # Send to aggregator (simplified: local update for decentralized
mode)
        if FEDERATED_MODE == 'centralized':
                    global_grad = aggregator.aggregate(self.department_id,
noisy_grad)
        else:  # decentralized gossip
            global_grad = self._gossip_exchange(noisy_grad)

        self.policy.weights -= self.learning_rate * global_grad
            self.middleware.policy_engine.update_policy(self.department_id,
self.policy)
```

17.5. Benefits of Federated Policy Learning

Aspect	Traditional (Manual)	Federated Learning
Policy Tuning	Manual, requires expert knowledge	Automatic, data-driven
Adaptation	Static, must be reconfigured	Dynamic, adapts to workload shifts
Privacy	Data may be exposed during analysis	Privacy-preserving (only gradients shared)
Departmental Customization	Limited to a few global policies	Each department learns optimal policy
Scalability	Human bottleneck	Fully automated

18. Exploration-Exploitation in Routing: A Formal Jitter Model

The stochastic noise (jitter) introduced in the routing decision serves a critical purpose beyond mere load distribution: it enables systematic exploration of the routing space, allowing the system to discover better nodes and adapt to changing conditions. This section formalizes this mechanism as an Exploration Bonus and integrates it into the mathematical framework of the Cost Vector.

18.1. Motivation: The Exploration-Exploitation Trade-off

In any adaptive system, there exists a fundamental tension:

- Exploitation: Routing requests to the node with the lowest estimated cost (greedy policy) yields the best short-term performance but may lead to suboptimal long-term outcomes because:

—

  - Node performance metrics may change over time (e.g., thermal throttling).
  - New nodes may be added to the system.
  - The workload distribution may shift.
- Exploration: Occasionally routing to a suboptimal node allows the system to gather information about:
  - Actual performance of "cold" nodes.
  - Performance under different load conditions.
  - Model loading times and inference speeds in practice.

Formal Problem: The routing system faces a multi-armed bandit problem [22] where each node is an "arm" with unknown and time-varying reward distribution.

18.2. Formalization of Exploration

Definition 9: Exploration Bonus The exploration bonus $\xi(v_i, t)$ is a positive term added to the scalar cost during routing decisions to encourage exploration of underutilized nodes:

$$Cost_{explore}(r, v_i) = \Pi \cdot C(r, v_i) - \xi(v_i, t)$$

Definition 10: Exploration Bonus Function The exploration bonus is a function of the node's "information value":

$$\xi(v_i, t) = \beta \cdot \sqrt{\frac{2lnN_{total}}{N_{visits}(v_i)+1}}$$

Where:

$\beta \in R^+$ is the exploration weight (hyperparameter controlling exploration intensity).

N_{total} is the total number of routing decisions made by the system.

$N_{visits}(v_i)$ is the number of times node v_i has been selected for routing.

Interpretation: This formulation is analogous to Upper Confidence Bound (UCB) [23] algorithms:

- Nodes with few visits have high uncertainty and receive a large bonus.
- The exploration bonus decays as the node is used more frequently.
- The logarithm ensures that exploration never fully stops (N_{visits} grows slower than lnN_{total}).

18.3. Incorporating Jitter into the Cost Vector

The jitter observed in the implementation (Section 7.2, line 63-64) can be formally represented as a stochastic exploration term:

Authors: Khairullin Artur (RU) *prcr98@gmail.com*

—

Original Implementation:

```
if min_cost > cost * (1 + 0.05 * random.random()):
    min_cost = cost
    best_node = node
```

Formal Model: The stochastic selection probability is:

$$P(v_i|r) = \frac{\exp exp\,(-\alpha \cdot (\Pi \cdot C(r,v_i) - \xi(v_i,t)))}{\sum_{j=1}^{N} exp(-\alpha \cdot (\Pi \cdot C(r,v_j) - \xi(v_j,t)))}$$

Where:

$\alpha \in R^+$ is the temperature parameter controlling the softmax softness.

$\xi(v_i, t)$ is the exploration bonus.

Relation to Jitter: The 5% jitter in the implementation approximates this stochastic selection when α is small and exploration bonus is implicit.

18.4. The Exploration-Exploitation Algorithm

Algorithm: Softmax Exploration with UCB

```
class ExplorationAwareOptimizer(PolicyDrivenOptimizer):
    def __init__(self, node_discovery: DiscoveryService, beta: float = 1.0,
alpha: float = 1.0):
        super().__init__(node_discovery)
        self.beta = beta  # Exploration weight
        self.alpha = alpha  # Softmax temperature
        self.visit_counts = {}  # N_visits(v_i)
        self.total_decisions = 0

         async  def  select_node(self,  request:  Request,  policy_vector:
List[float]) -> Optional[Node]:
        nodes = await self.node_discovery.get_all_status()
        eligible_nodes = self.constraint_checker.filter(nodes, request)

        if not eligible_nodes:
```

Authors: Khairullin Artur (RU) *prcr98@gmail.com* 09 July 2026

—

```
            return None

        self.total_decisions += 1
        scores = []

        for node in eligible_nodes:
            # Base cost
                                                    cost_vector =
self.cost_vector_calculator.compute_cost_vector(node, request)
                base_cost = sum(policy_vector[i] * cost_vector[i] for i in
range(len(policy_vector)))

            # Exploration bonus (UCB)
            n_visits = self.visit_counts.get(node.id, 0)
                        exploration_bonus = self.beta * math.sqrt((2 *
math.log(self.total_decisions)) / (n_visits + 1))

                # Adjusted score (lower is better, but we use negative for
softmax)
            score = base_cost - exploration_bonus
            scores.append(score)

        # Softmax selection (stochastic exploration)
        probabilities = self._softmax([-self.alpha * s for s in scores])
                    selected_idx = np.random.choice(len(eligible_nodes),
p=probabilities)
        selected_node = eligible_nodes[selected_idx]

        # Update visit count
                                    self.visit_counts[selected_node.id] =
self.visit_counts.get(selected_node.id, 0) + 1
```

Authors: Khairullin Artur (RU) *prcr98@gmail.com*

—

```
        return selected_node

    def _softmax(self, values: List[float]) -> List[float]:
        exp_values = np.exp(values - np.max(values))  # Numerical stability
        return exp_values / np.sum(exp_values)
```

18.5. Theoretical Guarantees

Theorem 6 (Regret Bound for PBDR with Exploration): For the softmax exploration routing algorithm with UCB-style exploration bonus, the expected cumulative regret (difference between optimal and actual performance) grows as $O(logT)$, where T is the total number of routing decisions.

Proof Sketch: Following the standard UCB regret analysis [23], the probability of selecting a suboptimal node decays exponentially with the number of visits, resulting in logarithmic regret. The softmax temperature α controls the exploration-exploitation trade-off, with the optimal regret achieved when $\alpha = O(\sqrt{logN/T})$

.

Corollary 1 (Convergence): As $T \to \infty$, the routing policy converges to the optimal policyΠ^*, assuming stationary node performance.

Corollary 2 (Adaptivity): The algorithm maintains the ability to adapt to non-stationary environments because exploration never fully ceases (due to the logarithmic term).

18.6. Practical Recommendations

Scenario	Recommended β	Recommended α	Rationale
Stable Environment (few new nodes)	0.1-0.3	2.0-5.0	Low exploration, high exploitation
Dynamic Environment (frequent node changes)	0.5-1.0	1.0-2.0	Moderate exploration to discover changes
Cold Start (new cluster)	2.0-5.0	0.5-1.0	High exploration to characterize nodes
Production with Monitoring	0.3-0.5	2.0-3.0	Balanced for safety and adaptation

Authors: Khairullin Artur (RU) *prcr98@gmail.com* 09 July 2026

—

19. Versioned State Protocol for Cache Consistency

A critical challenge in the PBDR architecture is the stale cache problem: multiple middleware instances maintain independent caches of node states, and these caches can become inconsistent without a global synchronization mechanism. This section introduces the Versioned State Protocol (VSP) —a lightweight, stateless mechanism that ensures routing decisions are made based on fresh (≥90% accuracy) node state information.

19.1. Problem Statement and Analysis

Problem Definition: Let M be the set of middleware instances. Each middleware $m \in M$ maintains a local cache $Cache_m(v_i)$ of node states. The cache is updated via periodic polling at intervals of Δt. Between updates, the cache may become stale as node states evolve independently.

Staleness Quantification: Let $State_{true}(v_i, t)$ be the true state of node v_i at time t, and $State_{cache}(v_i, t)$ be the cached value at middleware m. The staleness is defined as:

$$\Delta(v_i, t) = \|State_{true}(v_i, t) - State_{cache}(v_i, t)\|$$

Impact of Stale Cache:

Metric Change	Impact
Queue length increases	Middleware underestimates wait time, may overload node
GPU utilization spikes	Middleware underestimates load, may cause congestion
VRAM freed	Middleware overestimates pressure, avoids available node
Model switched	Middleware may route to node without required model

19.2. Versioned State Protocol (VSP)

The Versioned State Protocol enables cache consistency without centralized synchronization by extending the Monitor API with state versioning.

19.2.1. Enhanced Monitor API

Extension to /status endpoint:

```
paths:
  /status:
    get:
```

Authors: Khairullin Artur (RU) *prcr98@gmail.com*

—

```yaml
    parameters:
      - name: If-Version
        in: header
        schema:
          type: string
          pattern: '^[0-9a-f]{16}$'  # 64-bit hex version
        description: |
          The version of the state currently cached by the middleware.
          Node returns 200 with new state if version differs,
          otherwise returns 304 Not Modified.
```

Enhanced StatusResponse:

```yaml
StatusResponse:
  type: object
  properties:
    node:
      $ref: '#/components/schemas/NodeInfo'
    gpu:
      $ref: '#/components/schemas/GPUInfo'
    # ... other fields ...
    version:
      type: string
      description: |
        Monotonically increasing version identifier for this node's state.
        Can be a timestamp (Unix nanosecond) or a hash of the state.
      example: "2026010712345678"
      pattern: '^[0-9a-f]{16}$'
    version_timestamp:
      type: integer
      format: int64
```

—

```
        description: Unix timestamp (nanoseconds) when this version was
created
      example: 1700000000000000000
```

19.2.2. Version Update Mechanism

Node-Side Version Management:

```
class VersionedMonitorService:
    def __init__(self):
        self.version = self._generate_version()
        self.last_state = None

        def get_status(self, if_version: Optional[str] = None) ->
Tuple[StatusResponse, int]:
        """Return status and HTTP status code."""
        current_state = self._collect_state()
        current_version = self._compute_version(current_state)

        # Check if the cached version matches current state
        if if_version and if_version == current_version:
            return None, 304  # Not Modified

        # Update internal version
        self.version = current_version
        self.last_state = current_state
        return current_state, 200

    def _compute_version(self, state: StatusResponse) -> str:
        """Compute a version hash from the state."""
        # Use a fast, collision-resistant hash (e.g., xxHash)
        import xxhash
        serialized = json.dumps(state, sort_keys=True)
```

—

```
        return xxhash.xxh64(serialized).hexdigest()

    def _on_state_change(self):
        """Triggered when any significant metric changes."""
          # Version is recomputed on each request; no explicit invalidation
needed
        pass
```

19.2.3. Middleware Cache Management

Cache with Version Tracking:

```
class VersionedCache:
    def __init__(self):
        self.cache = {}  # node_id -> (version, state, timestamp)
        self.max_age = 1.0  # Maximum cache age in seconds
        self.min_accuracy = 0.90  # Target cache freshness

    async def get_node_state(self, node: Node, force_refresh: bool = False)
-> Optional[StatusResponse]:
        """Get node state, respecting version headers."""
        cache_entry = self.cache.get(node.id)
        current_time = time.time()

        # Check if cache is fresh enough
        if not force_refresh and cache_entry:
            age = current_time - cache_entry.timestamp
            if age < self.max_age:
                # Estimate staleness probability
                            if  self._staleness_probability(age)  <  1  -
self.min_accuracy:
                    return cache_entry.state
```

Authors: Khairullin Artur (RU) *prcr98@gmail.com*

—

```
        # Fetch with version header
        headers = {}
        if cache_entry:
            headers['If-Version'] = cache_entry.version

        async with aiohttp.ClientSession() as session:
             async with session.get(f"{node.url}/status", headers=headers)
as resp:
                if resp.status == 304:
                    # State unchanged; update timestamp only
                    self.cache[node.id].timestamp = current_time
                    return self.cache[node.id].state

                elif resp.status == 200:
                    state = await resp.json()
                              version = resp.headers.get('X-State-Version',
state.get('version'))
                    self.cache[node.id] = CacheEntry(
                        version=version,
                        state=state,
                        timestamp=current_time
                    )
                    return state

                else:
                    # Node error; return stale cache if available
                    return cache_entry.state if cache_entry else None

    def _staleness_probability(self, age: float) -> float:
        """
        Estimate the probability that the cache is stale.
```

—

```
        Uses an exponential model based on observed change rates.
        """
        # λ (lambda) is the estimated rate of state changes per second
        # Learned from historical data
        lambda_estimate = self._estimate_change_rate()
        return 1 - math.exp(-lambda_estimate * age)

    def _estimate_change_rate(self):
        """Estimate state change rate from previous version changes."""
        # Maintain a sliding window of version changes
        total_changes = len(self.version_changes)
        total_time = sum(ts for _, ts in self.version_changes)
        return total_changes / total_time if total_time > 0 else 0.1
```

19.2.4. Consistency Guarantees

Theorem 7 (Versioned State Freshness Guarantee): The Versioned State Protocol ensures that the probability of using a stale cache entry for a routing decision is bounded by $1 - P_{fresh}$, where $P_{fresh} \geq 0.90$ under normal network conditions.

Proof: Let Δt be the time since the last successful state fetch from a node. The node's state can change at rate λ (estimated per node). The probability that the state is unchanged is $e^{-\lambda \Delta t}$. By setting the cache refresh policy such that middleware requests are conditionally pulled based on the If-Version header, the actual staleness is bounded by the maximum network round-trip time plus middleware processing delay. For typical enterprise networks (RTT < 10ms), this yields $P_{fresh} > 0.99$.

19.3. Optimistic Routing with Versioned State

To further improve performance, middleware can use optimistic routing when some nodes are unresponsive:

Algorithm: Optimistic Routing with Versioned Cache

```
class OptimisticRouter:
    async def select_node(self, request: Request, policy: Policy) ->
Optional[Node]:
        # Fetch states in parallel, with version headers
        states = await asyncio.gather(
```

Authors: Khairullin Artur (RU) *prcr98@gmail.com* 09 July 2026

—

```
            *[cache.get_node_state(node) for node in self.nodes],
            return_exceptions=True
        )

        # Partition nodes by freshness
        fresh_nodes = []
        stale_nodes = []
        for node, state in zip(self.nodes, states):
            if isinstance(state, Exception) or state is None:
                # Node unreachable; use last known state with penalty
                stale_nodes.append((node, self.cache.get(node.id)))
            elif self.cache.is_fresh(node.id):
                fresh_nodes.append((node, state))
            else:
                stale_nodes.append((node, state))

        # Filter eligible nodes
              eligible_fresh  =  self.constraint_checker.filter(fresh_nodes,
request)
              eligible_stale  =  self.constraint_checker.filter(stale_nodes,
request)

          # Apply a freshness penalty to stale nodes (encourages using fresh
data)
        def apply_penalty(node, state):
            base_cost = self._compute_cost(node, state)
            if node.id in stale_node_ids:
                base_cost *= 1.1  # 10% penalty for stale data
            return base_cost

        # Select from union of fresh and stale nodes
```

Authors: Khairullin Artur (RU) *prcr98@gmail.com* 09 July 2026

—

```
        all_eligible = eligible_fresh + eligible_stale

            selected = min(all_eligible, key=lambda x: apply_penalty(x[0],
x[1]))


        return selected[0]
```

19.4. Performance Analysis

Metric	Without VSP	With VSP	Improvement
State Freshness	~60-70% (at typical poll intervals)	>95%	+25-35%
Routing Accuracy	Degrades with stale data	Maintained	Significant
Network Overhead	Full state transfer every poll	Conditional (304 on unchanged)	40-60% reduction
Latency Impact	~50-100ms (full fetch)	~1-5ms (version check)	90% reduction

Theorem 8 (Network Overhead Reduction):
The Versioned State Protocol reduces the average discovery traffic from $O(N \cdot S)$ to $O(N \cdot S \cdot (1 - \Delta_{version}))$, where $\Delta_{version}$ is the rate of state changes (typically < 0.2).

$$T_{traffic}^{VSP} = \sum_{i=1}^{N} (S \cdot 1_{state_changed(v_i)} + H \cdot 1_{state_unchanged(v_i)})$$

Where H is the size of the version header (≈50 bytes), and $S \approx 2KB$. This yields a 4-5x reduction in discovery traffic.

20. Discussion

This section places the work in a broader context and discusses its implications.

20.1. Positioning PBDR as Enterprise AI Middleware

The proposed architecture is not just "a system for using idle GPUs." It is a new class of middleware for enterprise AI. The Policy-Driven Optimizer is the core novelty.

- Beyond LLMs: While optimized for LLMs, the architecture is applicable to any stateless, compute-intensive service. The same principles could be applied to:
    - Image generation (Stable Diffusion)
    - Speech recognition (Whisper)

—

  - Text-to-speech (TTS)
  - Video generation
  - Scientific computing workloads
- Beyond Load Balancing: Traditional load balancers (e.g., HAProxy, NGINX) are application-agnostic. PBDR's Policy-Driven Optimizer is application- and context-aware. It considers the specifics of the request (model, prompt length, output length) and the state of the node (loaded model, temperature, VRAM pressure).
- Platform Agnostic: The PBDR Middleware can be deployed in front of any OpenAI-compatible server (vLLM, TGI, Ollama, LM Studio), making it an independent, vendor-neutral solution.

20.2. The Key Innovation: Cost Vector and Policy Vector Separation

The most significant innovation in PBDR is the clean separation of concerns through the Cost Vector and Policy Vector:

1. Nodes publish objective facts (Cost Vector components). They know nothing about routing policies, cost functions, or optimization strategies.
2. The Policy-Driven Optimizer computes scalar costs. It is a pure function: given a request and node states, it returns the optimal node.
3. Policies are weight vectors. Different departments can have different routing strategies without modifying the infrastructure.

This separation makes the system:

- Extendable: New cost components can be added without changing nodes.
- Testable: The Policy-Driven Optimizer can be unit-tested with mock data.
- Independent: Nodes can be upgraded or replaced without affecting the routing logic.
- ML-Ready: The analytical Cost Vector can be replaced with a machine learning model that predicts actual generation time.

20.3. Advantages of the PBDR Architecture

1. Cost-Efficiency: Utilizes already-purchased capital assets, eliminating the need for expensive server-grade GPUs. A typical enterprise can save $100,000$ −500,000 in CapEx by using existing workstations.
2. Scalability: Adding a new GPU node is as simple as installing the Monitor Service and updating the Middleware's discovery list. The system scales linearly (Theorem 4).
3. Fault Tolerance: Due to the absence of a central scheduler, the failure of any node does not cause a system-wide outage (Theorem 2). The PBDR Middleware can be horizontally scaled, eliminating it as a single point of failure.

4. Performance: The intelligent Cost Vector ensures that requests are optimally distributed, leading to better average and tail latency compared to naive scheduling.
5. Security and Privacy: All data remains within the corporate network, and TLS ensures encryption in transit.
6. Simplified Management: The architecture is far simpler than managing a Kubernetes cluster or a dedicated AI server farm.
7. Policy Flexibility: The Policy Engine allows different departments or applications to have different routing strategies.

20.4. Limitations and Future Work

1. Node Autonomy: The primary limitation is the user's ownership of the node. If a designer starts a heavy GPU task, the AI service must yield. This is addressed by the monitoring and accept_new_jobs flag.
2. Network Topology: Performance could be affected if nodes are spread across different subnets.
3. Model Management: Managing model updates across a large cluster can become cumbersome without a centralized model registry.
4. Self-Learning Optimizer: Future work could involve developing an ML-based predictor for the Cost Vector.
5. Decentralized Discovery: Implementing a gossip-protocol-based node discovery mechanism.
6. Federated Learning: Extending the architecture to support federated learning across distributed nodes.
7. Security Enhancements: Implementing fine-grained access control and integration with enterprise identity providers.

21. Conclusion

This paper has presented Policy-Based Decentralized Routing (PBDR) , a novel middleware architecture for enterprise AI inference. We have formalized the routing problem as a constrained optimization task, introduced a robust Policy-Driven Optimizer, and defined a flexible policy-based framework for application-aware routing.

Key innovations include:

1. The Cost Vector formulation, representing objective cost components.
2. The Policy Weight Vector, allowing flexible, application-specific routing.
3. The scalar product $\Pi \cdot C$ as the unified cost metric. 4. Formal proofs of correctness, fault tolerance, monotonicity, and linear scalability. 5. A

three-stage validation framework (Simulation, Prototype, Production). 6. Comprehensive security considerations for enterprise deployment.

Through formal proofs and analysis, we have demonstrated the system's scalability (Theorem 4: $O(N)$complexity), correctness (Theorem 1), fault tolerance (Theorem 2), and monotonicity (Theorem 3). The Monitor API specification ensures a clear separation of concerns. The network traffic analysis confirms that the discovery protocol introduces negligible overhead.

By positioning PBDR as a middleware platform with a clean separation of node state publication, cost vector computation, and policy application, we have shown its potential to go beyond a simple use-case for spare GPU resources. It serves as a foundational building block for future enterprise AI platforms that are private, cost-effective, and adaptable to diverse organizational needs.

22. References

[1] Gartner, "Gartner Survey Finds 78% of Enterprises Cite Data Privacy as Primary Concern for AI Adoption," 2023.

[2] J. Dean and L. A. Barroso, "The tail at scale," *Communications of the ACM*, vol. 56, no. 2, pp. 74-80, 2013.

[3] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. Gonzalez, H. Zhang, and I. Stoica, "Efficient memory management for large language model serving with pagedattention," in *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023, pp. 611-626.

[4] Hugging Face, "Text Generation Inference (TGI)," [Online]. Available: https://github.com/huggingface/text-generation-inference. [Accessed: 2024-06-01].

[5] "Ollama," [Online]. Available: https://ollama.com/. [Accessed: 2024-06-01].

[6] NVIDIA, "TensorRT-LLM," [Online]. Available: https://github.com/NVIDIA/TensorRT-LLM. [Accessed: 2024-06-01].

[7] G. Gerganov, "llama.cpp," [Online]. Available: https://github.com/ggerganov/llama.cpp. [Accessed: 2024-06-01].

[8] D. P. Anderson, "BOINC: A system for public-resource computing and storage," in *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, 2004, pp. 4-10.

[9] A. L. Beberg, D. L. Ensign, G. Jayachandran, S. Khaliq, and V. S. Pande, "Folding@home: Lessons from eight years of volunteer distributed computing," in *Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing*, 2009, pp. 1-8.

[10] M. Demmer and M. Roussopoulos, "Gossip-based state dissemination for scalable group communication," in *Proceedings of the 11th ACM SIGOPS European Workshop*, 2004, pp. 1-6.

—

[11] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and S. Lewin, "Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web," in *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*, 1997, pp. 654-663.

[12] J. Strassner, *Policy-Based Network Management: Solutions for the Next Generation*. Morgan Kaufmann, 2003.

[13] "Istio," [Online]. Available: https://istio.io/. [Accessed: 2024-06-01].

[14] S. Ramirez, "FastAPI," [Online]. Available: https://fastapi.tiangolo.com/. [Accessed: 2024-06-01].

[15] NVIDIA, "pynvml," [Online]. Available: https://github.com/gpuopenanalytics/pynvml. [Accessed: 2024-06-01].

[16] G. Rodola, "psutil," [Online]. Available: https://github.com/giampaolo/psutil. [Accessed: 2024-06-01].

[17] "orjson," [Online]. Available: https://github.com/ijl/orjson. [Accessed: 2024-06-01].

[18] S. Colvin, "Pydantic," [Online]. Available: https://docs.pydantic.dev/latest/. [Accessed: 2024-06-01].