System-Aware Multi-Agent Orchestration: Co-Optimizing Reasoning and Latency via Hierarchical RL

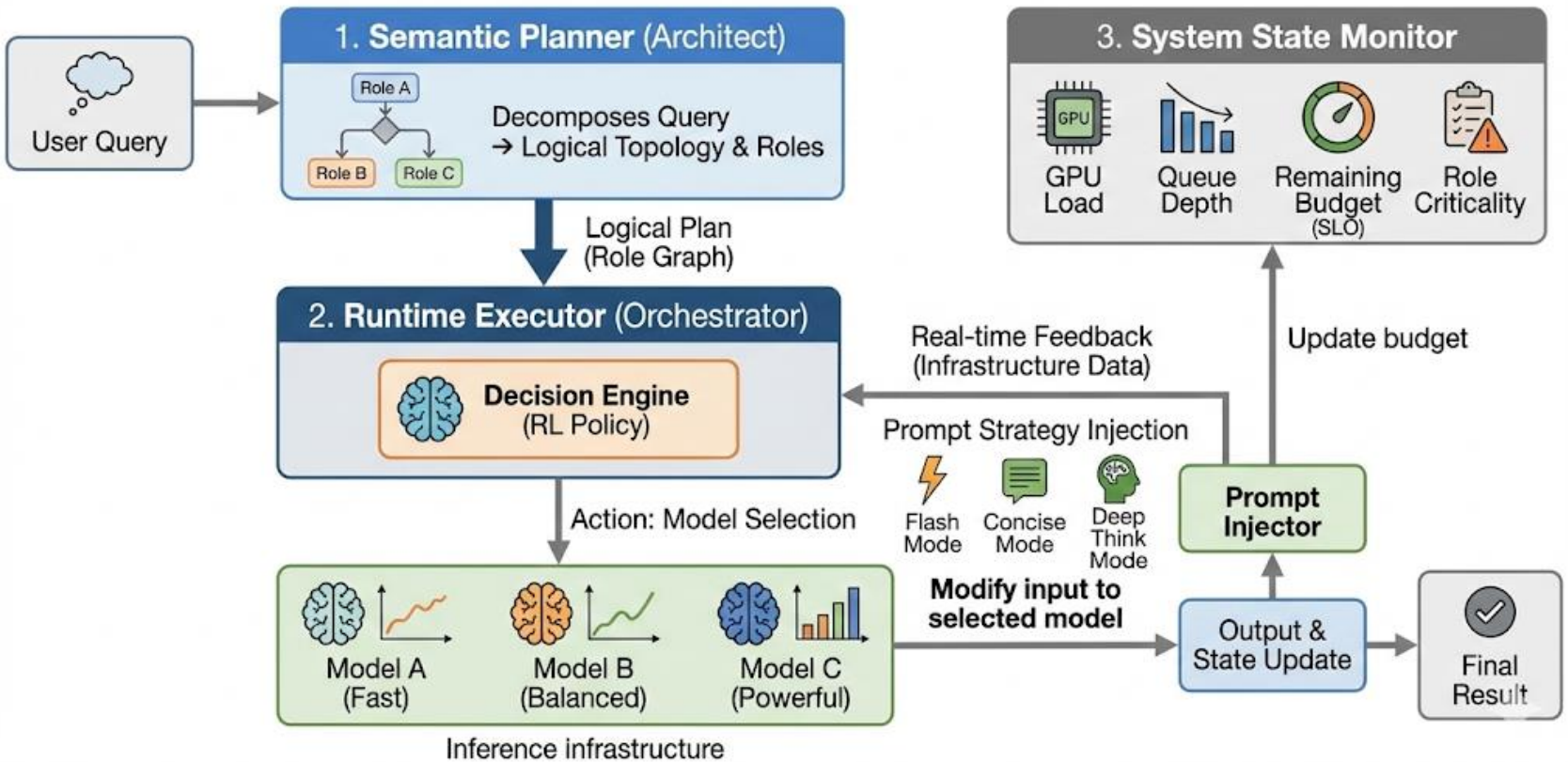
- **The Limitations of "Client-Side" Orchestration**
  - Current Multi-Agent routers (e.g., MasRouter) operate as *Black-Box API Clients*.
  - They optimize for *Static Constraints*: Price per Token ($) or Benchmark Quality.
  - **The Blind Spot:** They lack visibility into real-time infrastructure states, treating overloaded and idle models identically.

- **The System Provider's Perspective**
  - Unlike API users, System Providers possess *Granular Observability* into the inference backend:
    * *Real-Time Load:* Distinguishing which specific model replicas are **Idle** (responsive) versus **Busy** (queued).
    * *SLO Governance:* Direct awareness of the Service Level Objective (SLO) deadlines for each request.
    * *Resource States:* Visibility into GPU utilization and KV-cache saturation.
  - **The Opportunity:** Shifting orchestration to the system layer allows for dynamic routing that co-optimizes semantic goals with the physical reality of the hardware.

- **Gap 1: Load-Blind Model Selection (Routing Level)**
  - Standard routers rely on static profiles, ignoring dynamic contention.
  - **The Paradox:** A "High-Quality" model (e.g., Llama-3-70B) may be idle and responsive, while a "Low-Cost" model (e.g., Llama-3-8B) is thrashing under heavy queues.
  - *Result:* Suboptimal routing decisions that cause latency spikes despite theoretical cost efficiency.

- **Gap 2: Rigid, Budget-Agnostic Reasoning (Cognitive Level)**
  - The agent's thinking process is decoupled from System Constraints.
  - **Idle System Failure:** Critical roles (e.g., Coder) fail to recognize when ample budget exists, missing opportunities to "think deeper" for higher quality.
  - **Busy System Failure:** During contention, agents fail to adapt strategies (e.g., switching to *Concise Mode*), causing budget exhaustion via unnecessary verbosity.
  - *Conclusion:* We lack a feedback loop where Infrastructure State modulates the *depth* of Agent Reasoning.

System-Aware Multi-Agent Orchestration



We propose **System-Aware Multi-Agent Orchestration**, a framework that transforms agents from static entities into adaptive workers. Instead of optimizing only for Model Quality, the system co-optimizes for **Inference Velocity** by adapting the agent's behavior to the current infrastructure reality.

1. Hierarchical Control Architecture

The orchestration is decomposed into two levels of abstraction to separate semantic planning from runtime execution:

- **Level 1: The Planner (Semantic Architect)**
  - *Role:* Decomposes the user query into a logical topology (e.g., Chain, Graph) and assigns specific Roles.
  - *Focus:* Optimizes for pure semantic correctness. It executes once at the initialization of the request ($T = 0$).
- **Level 2: The Executor (Runtime Orchestrator)**
  - *Role:* Navigates the trade-off between Quality and Speed at every individual agent step ($t = 1 \dots K$).
  - *Action Space:* It jointly selects the optimal **Model** AND the **Prompt Strategy** based on real-time feedback.

2. The Feedback Mechanism: Behavioral Prompt Injection

We introduce a novel control channel where the System State directly modulates the Agent's output complexity. The decision logic dynamically weighs **System Load** (Queues), **Role Importance**, and the **Remaining Budget** (SLO).

- **Adaptive Strategies:**
  - **Flash Mode:** "Answer immediately, no explanation." (Triggered by Low Budget / Simple Role).
  - **Concise Mode:** "Be brief, focus on result." (Triggered by Medium Budget / Support Role).
  - **Deep Think Mode:** "Reason step-by-step." (Triggered by High Budget / Critical Role).

We formalize the orchestration problem as a sequential decision process. While previous approaches treat model selection as a static trade-off between quality and price, we introduce time-bound system constraints into the optimization loop.

1. Optimization Objective

Baseline (Standard MAS Router):
Existing multi-agent routers formulate the problem as a Contextual Bandit or simple RL task. They optimize a scalar objective combining Semantic Quality (Q) and Financial Cost (C) based on static API pricing.

$$\pi_{base}^{*} = \underset{\pi}{\operatorname{argmax}} \, E_{(s,a)\sim\pi} \left[Q(s,a) - \alpha \cdot C_{price}(a) \right]$$

Limitation: The cost function C_{price} is static. It does not account for runtime latency, making it impossible to guarantee time-bound delivery during system contention.

Ours (System-Aware Formulation):
We reformulate the problem as a **Constrained Markov Decision Process (CMDP)**. The objective is to maximize Semantic Quality subject to a dynamic Latency Budget (B_t).

$$\max_{\pi} E \left[Q(s,a) \right] \quad \text{s.t.} \quad E \left[L_{sys}(s,a) \right] \leq B_t$$

Where $L_{sys}(s,a)$ is the real-time system latency (queue + inference time) for taking action a in state s.

2. The Augmented State Space

To solve the CMDP, we augment the agent's observation space. Unlike standard routers that observe only the semantic query, our state S_t at step t captures the physical reality of the infrastructure.

$$S_t = [\mathbf{h}_{sem} \oplus \mathbf{v}_{sys} \oplus b_t]$$

- $\mathbf{h}_{sem}$: The semantic embedding of the current user query and agent role history.
- $\mathbf{v}_{sys}$: A vector representing real-time **System State**, including queue depth and GPU utilization across available model replicas $\mathcal{M}$.
- b_t: The **Remaining Latency Budget**, calculated as $b_t = \text{SLO}_{total} - T_{elapsed}$.

3. The Joint Action Space

Standard routers optimize over a single dimension (Model Selection). We introduce a hierarchical action space that jointly optimizes the hardware resource and the cognitive load.

$$a_t = (m_t, \rho_t)$$

- $m_t \in \mathcal{M}$: The selected LLM (e.g., Llama-3-70B vs. Llama-3-8B).
- $\rho_t \in \mathcal{P}$: The **Prompting Strategy** (Behavioral Injection).
  - $\mathcal{P} = \{\text{Flash}, \text{Concise}, \text{DeepThink}\}$.

4. Reward Function

The reward function R_t guides the policy to balance quality against the budget constraint.

$$R_t = Q_{sem}(s_t, a_t) - \lambda \cdot \mathcal{P}_{latency}(L_{sys}, b_t)$$

- Q_{sem}: A score representing the semantic quality of the agent's response (e.g., derived from self-reflection or ground truth).
- $\mathcal{P}_{latency}$: A penalty function that activates when the expected latency L_{sys} approaches or exceeds the remaining budget b_t.

Updates : 19th January

Q1: How to determine the SLO/Budget initially for the whole workflow ?

Q2: Which opensource models to use for our project ?

(Based on the GPU resources we have. Also to make a diverse set of models so that different LLM's are good at different roles)

RL System State

System State Estimation & Predictive Modeling

To decouple complex queuing dynamics from the reinforcement learning loop, we employ a **Two-Stage Architecture**. Instead of feeding raw, noisy system metrics directly to the RL agent, we use supervised estimators to predict explicit latency values. The RL agent then optimizes based on these synthesized predictions.

The Latency Estimator ($\mathcal{F}_{lat}$)

This MLP regressor estimates the hardware response time for a specific Model m given the current system load.

Input Feature Vector ($\mathbf{x}_{lat}$):
For a candidate model m, the input is a concatenation of real-time infrastructure metrics and request metadata:

$$\mathbf{x}_{lat} = [q_{wait}, q_{run}, u_{kv}, \bar{\mu}_{ttft}, L_{in}, \mathbf{e}_{role}, \mathbf{e}_{model}]$$

- q_{wait}: Current Waiting Queue Length (Integer).
- q_{run}: Current Running/Batch Queue Length (Integer).
- u_{kv}: KV-Cache Utilization $(0.0 - 1.0)$.
- $\bar{\mu}_{ttft}$: Exponential Moving Average (EMA) of recent Time-to-First-Token (ms).
- L_{in}: Prompt Token Count (Integer).
- $\mathbf{e}_{role}, \mathbf{e}_{model}$: Learned embeddings for Role ID and Model ID.

Output Targets ($\mathbf{y}_{lat}$):

$$(\hat{t}_{ttft}, \hat{t}_{tpot}) = \mathcal{F}_{lat}(\mathbf{x}_{lat})$$

The Output Length Estimator ($\mathcal{F}_{len}$)

This estimator predicts the verbosity of the agent's response, which is crucial for calculating total generation time. It captures the variance introduced by different Prompt Strategies (e.g., "Flash" vs. "DeepThink").

Input Feature Vector ($\mathbf{x}_{len}$):

$$\mathbf{x}_{len} = [\mathbf{h}_{prompt}, \mathbf{e}_{role}, \mathbf{e}_{strat}, \mathbf{e}_{model}]$$

- $\mathbf{h}_{prompt}$: Semantic embedding of the input prompt.
- $\mathbf{e}_{strat}$: Embedding of the selected Prompt Strategy ($\rho \in \{\text{Flash}, \text{Concise}, \text{DeepThink}\}$).

Output Target ($\hat{L}_{out}$):

$$\hat{L}_{out} = \mathcal{F}_{len}(\mathbf{x}_{len})$$

Where $\hat{L}_{out}$ is the predicted number of generated tokens.

Synthesized RL State Space

Before the RL planning step, we pre-compute the **Total Expected Latency** ($\hat{T}_{total}$) for every candidate action (m, ρ) using the outputs from the estimators:

Latency Calculation:

$$\hat{T}_{total}(m, \rho) = \hat{t}_{ttft} + \left(\hat{L}_{out}(\rho) \times \hat{t}_{tpot} \right)$$

RL State Vector (S_{rl}):

The Reinforcement Learning agent observes a "synthesized" state containing these clean time predictions rather than raw queues:

$$S_{rl} = [\mathbf{h}_{sem} \oplus B_{rem} \oplus \mathbf{v}_{estimates}]$$

- $\mathbf{h}_{sem}$: Semantic Query Embedding.
- B_{rem}: Remaining Latency Budget (calculated as SLO $- T_{elapsed}$).
- $\mathbf{v}_{estimates}$: A vector containing $\hat{T}_{total}$ for all available candidate models.

Dataset Generation for capturing system state

Dataset Generation Strategy

To train the system predictors, we employ a two-phase data collection strategy. We will strictly use the **MBPP Dataset** for the initial pilot experiments. Once the pipeline is validated, we will expand to additional datasets.

Phase 1: Intermediate Prompt Collection

Objective: Capture the specific, **Intermediate Agent Prompts** received by individual agents during the workflow execution.

Methodology:

1. **Base Framework:** We utilize the **MASRouter Baseline** codebase on the MBPP dataset.
2. **Topology:** We rely on the dynamic topology generation of MASRouter.
3. **Randomized History:** To ensure the prompt chains themselves are diverse (since Agent A's output becomes Agent B's input), we apply random strategies during this generation phase.
4. **Output Artifact:** We save a CSV file (`intermediate_prompts.csv`) containing:
    - `prompt_text`: The exact text received by the agent.
    - `role_id`: The role assigned (e.g., Coder).

Phase 2: Hardware Metric Collection

Objective: Map these prompts to concrete hardware execution metrics. We test **all strategies** against every harvested prompt to maximize data density.

Methodology: We adopt a similar load generation methodology as our previous Hardware-Aware Router project to stress-test individual models.

1. **Background Load Generation:**
    - We sample from the `intermediate_prompts.csv` pool to generate realistic background traffic.
    - We vary the concurrency ($N = 1, 10, 50$) to create diverse Queue Depth and KV-Cache states.
2. **Strategy Iteration & Replay:**
    - For **every** prompt row collected in Phase 1, we iterate through all three strategies: $\rho \in \{\text{Flash}, \text{Concise}, \text{DeepThink}\}$.
    - We append the specific instruction for that strategy to the prompt.
    - We measure the exact execution metrics for that specific (Prompt + Strategy) combination.

Final Dataset Structure

The final dataset combines the system state, the injected strategy (from Phase 2), and the target model ID.

Category	Feature (Column Name)	Source
System State	`queue_waiting`	Load Generator
	`queue_running`	Load Generator
	`kv_utilization`	Load Generator
Request Info	`model_id`	Configuration (Target Model)
	`role_id`	Phase 1 CSV
	`strategy_id`	Phase 2 Injection (Iterated)
	`total_input_tokens`	Phase 1 + Strategy Text
Targets	`actual_ttft` (ms)	Measured
	`actual_tpot` (ms)	Measured
	`output_tokens` (Count)	Measured