arXiv:2505.00472v1 [cs.AI] 1 May 2025

USERCENTRIX: AN AGENTIC MEMORY-AUGMENTED AI FRAMEWORK FOR SMART SPACES

Alaa Saleh
Center for Ubiquitous Computing
University of Oulu
Oulu, 90014, Finland
alaa.saleh@oulu.fi

Sasu Tarkoma
Department of Computer Science
University of Helsinki
Helsinki, 00100, Finland
sasu.tarkoma@helsinki.fi

Praveen Kumar Donta
Department of Computer and Systems Sciences
Stockholm University
Stockholm, 106 91, Sweden
praveen@dsv.su.se

Naser Hossein Motlagh
Department of Computer Science
University of Helsinki
Helsinki, 00100, Finland
naser.motlagh@helsinki.fi

Schahram Dustdar
Distributed Systems Group
TU Wien and ICREA Barcelona
Vienna, 1040, Austria
dustdar@dsg.tuwien.ac.at

Susanna Pirttikangas
Center for Ubiquitous Computing
University of Oulu
Oulu, 90014, Finland
susanna.pirttikangas@oulu.fi

Lauri Lovén
Center for Ubiquitous Computing
University of Oulu
Oulu, 90014, Finland
lauri.loven@oulu.fi

ABSTRACT

Agentic AI, with its autonomous and proactive decision-making, has transformed smart environments. By integrating Generative AI (GenAI) and multi-agent systems, modern AI frameworks can dynamically adapt to user preferences, optimize data management, and improve resource allocation. This paper introduces `UserCentrix`, an agentic memory-augmented AI framework designed to enhance smart spaces through dynamic, context-aware decision-making. This framework integrates personalized Large Language Model (LLM) agents that leverage user preferences and LLM memory management to deliver proactive and adaptive assistance. Furthermore, it incorporates a hybrid hierarchical control system, balancing centralized and distributed processing to optimize real-time responsiveness while maintaining global situational awareness. `UserCentrix` achieves resource-efficient AI interactions by embedding memory-augmented reasoning, cooperative agent negotiation, and adaptive orchestration strategies. Our key contributions include (i) a self-organizing framework with proactive scaling based on task urgency, (ii) a Value of Information (VoI)-driven decision-making process, (iii) a meta-reasoning personal LLM agent, and (iv) an intelligent multi-agent coordination system for seamless environment adaptation. Experimental results across various models confirm the effectiveness of our approach in enhancing response accuracy, system efficiency, and computational resource management in real-world application.

***K*eywords** Agentic AI, Follow-Me AI, Multi-agent Systems, Large Language Models, LLM Memory Management, Resource Management, Computing Continuum

1 Introduction

The rapid evolution and integration of Generative AI (GenAI) and multi-agent systems into smart environments have expanded the scope of user interaction, data management, and resource allocation [1, 2, 3]. In such environments, personalized and real-time adaptability is critical to ensuring both user satisfaction and energy efficiency. This promotes

the adoption of edge intelligence, which enhances the computing continuum by integrating AI to improve real-time processing capabilities. This approach involves embedding agentic AI for proactive planning, continuous learning, reasoning, and adaptation with a dynamic settings, distributed across the computing continuum [4, 5, 6, 7].

Smart environments equipped with the Follow-Me AI [8] system aim to dynamically adjust to users' needs by deploying AI agents that negotiate to create a responsive, personalized experience. The personal agent, embedded within the user's personal device, continuously monitors the user's preferences and plans. With negotiation, the smart building agents assess the building's current environmental conditions and adapt environmental settings to the user's requirements while optimizing energy usage across the building.

However, this framework needs personal LLM agents that can analyze user tasks, manage personal data, and adapt dynamically to new tasks by using past experiences to extract useful knowledge to provide more personalized assistance by managing personal contexts and preferences. Additionally, there is a need to monitor changes in controlled variables, based on user context. In addition, the static decision-making mechanisms of current publish/subscribe (pub/sub) systems often lack the flexibility and control mechanisms needed to handle the dynamic and resource-intensive demands of such multi-agent frameworks [9]. Furthermore, as more LLM-based agents, ensuring effective coordination and communication becomes increasingly intricate [10, 11]. Addressing these challenges necessitates auto-scaling mechanisms to manage resource-intensive demands, along with adaptive orchestration strategies [12, 13]. Furthermore, enhancing multi-LLM agent systems with advanced reasoning capabilities and iterative learning is essential for effectively structuring these systems [14].

Centralized systems typically rely on central nodes for processing and decision-making and relay responses, resulting slower response times due to the bottlenecks created by these central nodes. Additionally, centralized approaches can be prone to single points of failure, risking stability and system integrity, especially as the system scales [15]. In contrast, distributed systems decentralize processing tasks, allowing agents to make local decisions independently, resulting in quicker responses and higher efficiency for localized tasks. However, without regular and sufficient information exchange between agents, distributed systems often lack a cohesive global perspective. To gain a comprehensive view of the environment, distributed agents must exchange substantial amounts of information, leading to increased communication overhead and network congestion [15]. Balancing these trade-offs is essential in designing a system that can handle both local autonomy and global situational awareness effectively [16].

By integrating centralized and distributed control within a hierarchical framework that harnesses the collective expertise of multiple agents, the multi-agent pub/sub system enables the formation of cooperative coalitions across hierarchy levels. This structure not only ensures efficient resource allocation during inference but also reduces unnecessary inter-layer communication and mitigates response delays. However, further exploration is required to fully understand how the number of LLM-based agents affects overall output quality and to develop methods that enhance communication efficiency and distribute workload effectively among agents. Additionally, there is a need to know how enhancing the reasoning capabilities of LLM agents impacts output quality and the structural dynamics of the system.

Given these considerations, `UserCentrix` framework integrates intelligence into its architecture specifically designed for user-centric services within the context of smart buildings. Our main contributions are summarized as follows:

- We propose a hybrid self-organizing LLM agent framework with proactive scaling and LLM memory management that dynamically adapts its decision-making strategy and allocates inference time budgets based on user task context.
- We demonstrate the adaptation of decision-making strategy can be driven by the Value of Information (VoI), which serves as the guiding factor in the framework's reasoning and prioritization process to identify what is most relevant to the user.
- We examine the significance of user context in guiding decisions to balance factors such as speed and accuracy, as well as allocating resources to enhance output quality.
- We develop a personal LLM agent designed to function as a knowledge-driven AI with advanced memory management and self-evaluation capabilities to make reliable decisions and efficient responses.
- We implement a memory-augmented agents with meta-reasoning capability and in-context learning allow for dynamic adjustments in roles, relationships, and the number of LLM calls in response to changes in VoI.
- We develop cooperative reasoning networks where the low-level agents can negotiate over the terms of their collaboration to avoid tasks conflicts, ensuring alignment with diverse user requirements.
- We develop an environment agent to track ongoing tasks dispatched through a message queue which is equipped with time-to-launch (TTL) settings, ensuring that control commands are sent to control system and the environment agent at the specified time.

- We consider environmental changes during ongoing tasks, enabling the environment agent to dynamically track and adjust the environment to support user needs.

The remainder of this paper is structured as follows: Section 2 reviews recent studies, highlighting their objectives and limitations. Section 3 outlines a general framework for smart spaces. Section 4 details the implemented scenario along with its requirements. Section 5 analyzes the experimental outcomes using various LLMs and SLMs. Finally, Section 6 summarizes the findings and limitations, and discusses directions for future research.

2 Related Works

2.1 LLMs for edge-cloud continuum

Integration LLM agents within the computing continuum represent a promising research direction [4], paving the way for more effective applications. While traditional LLMs typically depend on cloud computing, this reliance often results in increased latency, limiting their responsiveness. By contrast, edge computing offers a practical solution to these challenges by enabling the deployment of LLMs directly on edge devices, closer to the data sources. Several studies have explored deploying LLMs in edge-cloud computing environments and highlighted the potential of LLM agents across the edge-cloud continuum by addressing computational, latency, and resource management challenges through innovative edge-cloud collaboration and optimization strategies.

Shen et al. [17] proposed a cloud-edge-client hierarchical framework that enables edge AI systems to automatically organize, adapt, and optimize themselves to meet users' diverse requirements. By leveraging LLMs, the framework efficiently coordinates edge AI models to interpret user intentions and cater to personalized demands. A collaborative edge computing framework for LLM inference was proposed in [18]. This framework employs dynamic programming to partition models into shards and deploy them on distributed devices spanning edge devices and cloud servers. This hybrid approach facilitates collaboration between edge and cloud resources.

Hao et al. [19] proposed a hybrid inference framework featuring dynamic token-level edge-cloud collaboration. This framework balances both edge and cloud resources utilization to enhance inference performance. Yu et al. [20] developed Edge-LLM, a decentralized framework focused on optimizing LLM adaptation on edge devices through integrating layer-wise compression , adaptive layer tuning, and a hardware scheduling strategy for computational efficiency. Ding et al. [21] propose a method for optimizing service placement strategies by considering both model requests and the associated computational resource requirements. Their approach employs a routing mechanism that dynamically assigns queries to either a small or large model, based on the predicted difficulty of the query. A cloud-edge collaborative inference framework for edge intelligence to efficiently deploy LLM agents is proposed in [22]. This work focuses on optimizing service placement and inference task offloading strategies, leveraging cached LLMs on both cloud and edge servers. DLoRA [23] presents a distributed PEFT framework. Where the LLM is executed in the cloud servers while the PEFT modules are trained entirely within the user devices.

2.2 Structures design of multi-agent systems

Several recent approaches provide valuable insights into designing system structures that effectively harness collective capabilities in multi-agent systems. Some approaches focus on hierarchical architectures. For instance, a hierarchical structure with dynamic organization based on task requirements is introduced in [24]. "Mixture-of-Agents" (MoA) architecture [25] uses multiple LLMs organized across layers, allowing iterative refinement of outputs through collective agent input. Similarly, MegaAgent [26] employs a hierarchical structure that dynamically creates and manages agents based on task requirements.

GraphAgent-Reasoner [27] leverages centralized management to delegates reasoning tasks to multiple agents with scaling effectively for complex tasks with increasing the number of agents. In constant, MORPHAGENT [28] emphasize decentralized coordination with the efficiency of structured role optimization based on task requirements and real-time feedback, promoting adaptability and response to dynamic requirements. These recent approaches underscore how multi-agent systems that employ hierarchical coordination and dynamic role adaptation can enhance response quality by leveraging collective capabilities.

2.3 Reasoning LLM agent

Recent advancements in large language model (LLM) research have focused on enhancing reasoning and response quality, aiming to leverage the thinking capabilities of LLM agents more effectively. These include scaling inference-time computing by increasing the number of generated samples per problem [29] and adapting based on prompt

Table 1: Summary of Related Works.

Name	Task	Algorithm	Goal	Limitation
Cloud-Edge-Client Framework [17]	Develop a hierarchical framework for autonomous edge AI systems.	LLMs to organize, adapt, and optimize edge AI systems automatically.	Meet diverse user requirements with minimal latency.	-Limited focus on energy efficiency. -Resource allocation challenges in highly dynamic environments.
EdgeShard [18]	Design a collaborative edge computing framework for efficient LLM inference.	Dynamic programming to partition LLMs between edge and cloud resources.	Minimize inference latency and maximize throughput.	Resource allocation challenge.
Hybrid Inference Framework [19]	Propose a hybrid inference framework for LLM inference.	Dynamic token-level interactions during decoding time, combining edge and cloud resources for inference.	Enhance inference performance	Limited latency improvements.
Edge-LLM [20]	Develop a framework for optimizing LLM adaptation on edge devices.	Layer-wise compression (pruning and quantization), adaptive layer tuning (voting mechanism), and hardware scheduling	-Mitigate high computational and memory demands of LLMs. -Optimize performance on resource-constrained edge devices.	Lack focus on power consumption.
Hybrid LLM [21]	Optimizing service placement strategies for LLMs.	Considering model requests and computational resource requirements, and dynamically assign queries to either a small or large model	Improve memory and storage efficiency.	Insufficient to address the real-world need for a diverse array of LLMs.
Cached model-as-a-resource [22]	Propose service placement and inference task offloading strategies in a cloud-edge collaborative inference framework.	Auction mechanism.	Minimize the total inference cost of edge servers and the cloud.	Scalability challenge to serve a higher volume of user requests simultaneously.
Dlora[23]	Propose framework for collaborative training of LLMs across cloud and edge devices.	Distributed PEFT approach where the LLM parameters are stored in cloud servers, while PEFT modules are fine-tuned on edge devices.	-Reduce the computational workload on edge devices. -Minimize communication overhead.	-Challenges in resource-constrained networks. -Limitations of computational resources on edge devices.
Criticize-Reflect [24]	Explore how hierarchical structures and leadership roles impact multi-agent coordination.	Dynamically organizes LLMs based on task requirements.	Achieve continuous improvement in communication efficiency and cooperation.	Lack scalability with more agents in large environments.
Mixture-of-Agents [25]	Utilize multiple LLMs organized across layers for iterative output refinement.	Agents share and refine information across layers through cycles.	Enhance response quality by leveraging the unique strengths of diverse LLMs, demonstrating "collaborativeness".	Dependence on multiple MoA layers, increasing computational overhead.
MegaAgent [26]	Employ hierarchical structures for dynamic agent creation and management based on task requirements.	System-level parallelism with autonomous task splitting in centralized coordination.	Overcome challenges like limited cooperation and scalability issues.	Scalability issues and single points of failure in large-scale MAS due to centralized approach.
GraphAgent-Reasoner [27]	A collaborative Architecture of multi-agent with centralized management.	Distributed, node-centric task processing managed by a "Master LLM."	Scale effectively for complex tasks by delegating reasoning tasks to multiple agents.	Lack scalability forin larger and more complex real-world reasoning scenarios.
MORPHAGENT [28]	Enable decentralized multi-agent systems.	Agents dynamically adjust profiles—roles, skills, and strategies—based on task requirements and feedback.	Promote adaptability and responsiveness and address complex tasks.	Computational overhead.
Large Language Monkeys [29]	Increase the number of samples generated per problem for a model's coverage (the percentage of problems solved).	Generate multiple attempts instead of relying on single-attempt solutions.	Boost the accuracy and reliability of LLM outputs.	The strength of the verifier can constrain this process.
Compute-Optimal Scaling [30]	Adapt compute allocation during testing based on prompt difficulty.	Smarter use of computational resources during the test phase.	Improve efficiency and performance during inference.	Limiting LLMs' reasoning robustness and generalizability due to difficulties for verifiers in identifying errors.
Talker-Reasoner Agent Model [31]	Utilize a dual-agent framework inspired by Kahneman's "Thinking Fast and Slow".	Combines real-time interaction (Talker) with multi-step problem-solving (Reasoner).	Enable dynamic adaptation and improve efficiency.	-No strategy for minimizing Reasoner use when Talker suffices. -Lack of automatic Talker-Reasoner switching based on query complexity.
Collaborative Verification [32]	Explore multiple solution paths during inference.	Integrate "Chain of Thought" (CoT) and "Program of Thought" (PoT) approaches.	Enhance reasoning accuracy of LLMs.	-Resource-intensive when deployed in real-time applications. -The verifier can constrain this process.
Meta-Reasoning Prompting [34]	Dynamically select the most suitable reasoning method for each task.	Choose from Chain-of-Thoughts, Tree-of-Thoughts, Self-Refine, Step-Back Prompting, etc., based on task requirements.	Improve reasoning efficiency and accuracy.	Lack of an ensemble relevant methods approach to address complex problems.
OpenR [35]	Focus on intermediate reasoning steps using open-source frameworks.	Reinforcement learning for decision-making with process reward models (PRMs) for detailed feedback.	Enhance reasoning during text generation.	Limitation in the scale and diversity training datasets, and testing using mid-sized models.
Thought Preference Optimization [36]	Equip LLMs with the ability to think before responding.	Train LLMs to generate and optimize internal thoughts iteratively.	Produce thoughtful and accurate outputs for complex instructions.	Lack of exploration the thinking with larger-scale models and a more diverse set of thought prompts.
Quiet-STaR [37]	Simulate internal thought processes during text generation.	Generate useful internal rationales for each token to guide predictions using REINFORCE learning.	Predict future text more accurately. Improve capabilities for reasoning tasks.	Computationally intensive as it is applied at every token.
rStar [33]	Propose collaborative problem-solving approach.	SLM with Monte Carlo Tree Search to generate reasoning pathways, while another SLM evaluates them.	Enhance reasoning capabilities of SLMs.	Dependency on the ability of SLM to verify reasoning quality.

difficulty [30], illustrating the need for efficient use of computational resources during inference. Other strategies focus on real-time adaptation of reasoning techniques to meet specific task requirements. For example, a dual-system approach [31] enables dynamic adaptation through the two modes of thinking based on task requirement, while Liang et al. [32] and Qi et al. [33] explore generating multiple reasoning paths to enhance LLM reasoning capabilities. Meta-Reasoning Prompting (MRP) [34] further improves reasoning by dynamically selecting the most suitable approach for each task. Additional methods focus on enhancing reasoning during the text generation process itself, such as optimizing intermediate reasoning steps [35] and refining internal thoughts [36, 37]. Table 1 provides a summary of these related works, outlining their goals, algorithms used, main tasks, and limitations.

3 `UserCentrix` Framework

In this paper, we present `UserCentrix`, an agentic AI framework designed for smart spaces. This framework is dynamically adjusting to user context and autonomously scaling computational resources in response to task demands. `UserCentrix` is structured around a dual-layer architecture, comprising a user-side and a building-side, each with distinct functionalities to facilitate intelligent and efficient interactions within smart spaces as shown in Fig. 1.

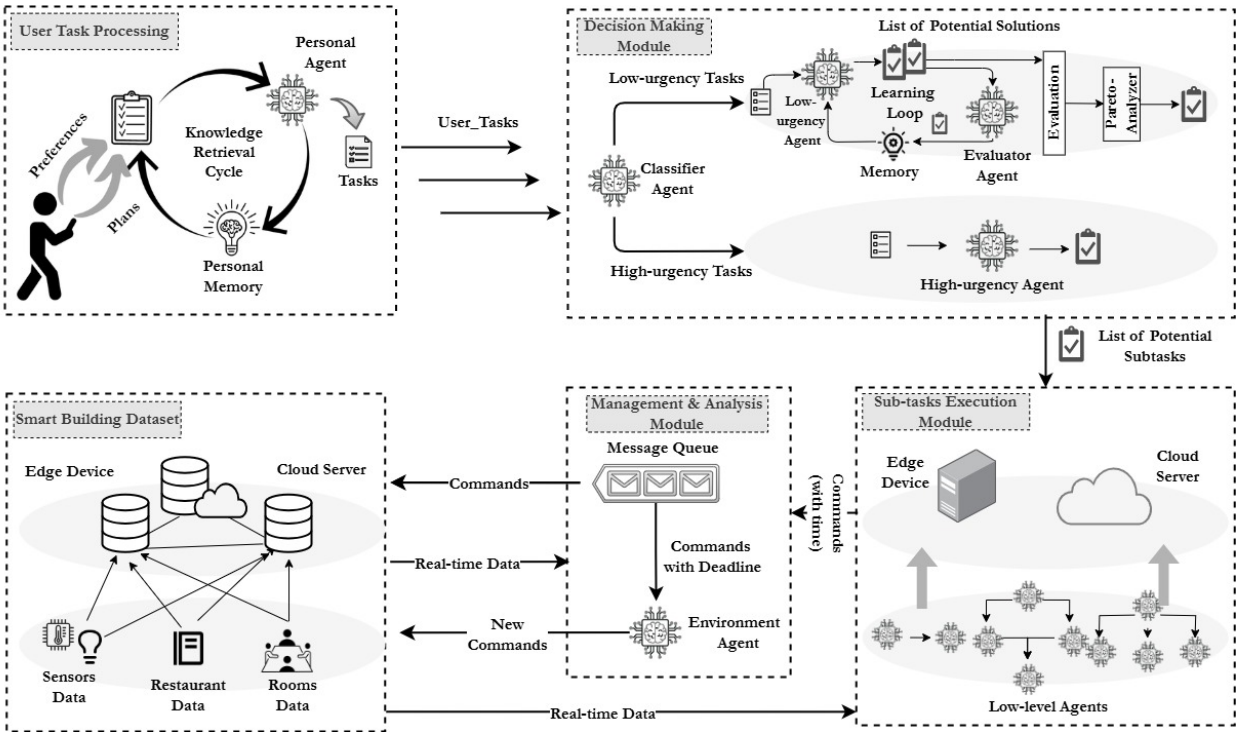


Figure 1: UserCentrix Framework.

On the user side, the framework employs personalized knowledge AI agents [38] powered by LLM. These agents serve as intelligent assistants tailored to individual users, leveraging personalized knowledge bases to improve user experiences. These user-centric AI agents continuously learn and adapt based on evolving user preferences. On the building side, the framework integrates memory-augmented meta-reasoning agents with learning capabilities. These agents are responsible for managing the broader smart space by processing data from various datasets and optimizing resource allocation.

3.1 User Side

The effectiveness of personal LLM-powered agents in UserCentrix framework relies significantly on their ability to effectively understand user need, keep track of the user, adapt dynamically to new needs. This adaptability involves not only understanding the user's immediate needs but also developing "Rethink" technique to apply existing knowledge to new situations as well as memorizing capability, resulting in efficient agent responses. In UserCentrix framework, we employ knowledge-based LLM agents with memorizing capability capable of understanding and analyzing user preferences and plans.

These agents maintain an internal representation of knowledge, enabling them to use Case-Based Reasoning (CBR) [39] to assess and respond to changing user demands. The knowledge base, acting as a repository, holds background data on previous user interactions, including plans, plan timestamps, plan types, and detailed preferences. Each plan is stored both as textual descriptions to reuse in future contexts. This repository becomes a decision-making guide in new scenarios by leveraging past knowledge to inform present actions, enabling more accurate tasks as shown in Figure 1.

The personal agent is designed to initiate with an empty and dynamic repository. It populates continuously over time as tasks are submitted by the user. This repository stores information about past executions, enabling the automatic incorporation of historical data into prompts for future agent executions. This ensures coherent responses to new user inputs. User tasks within the system include configuring smart building settings, reserving meeting rooms, ordering meals, and handling a wide range of personalized requests based on individual preferences and needs. This adaptive functionality enhances efficiency and user experience within the smart environment.

When a user submits a plan without specifying preferences, the agent utilizes its personal memory to assess semantic similarity by comparing the embeddings of the current plan type with those of past plans types. As well as, It also

compares the plan's timestamp with previous task timestamps. If a high degree of similarity is detected and less than one hour difference, the agent retrieves the most recent matching plan and automatically applies the associated preferences. This process enables the agent to update its knowledge base and adapt its responses based on knowledge and user history to make updated tasks, ensuring a context-aware user experience. For more details, we provide the personal agent's prompt in the Appendix. 6.

Furthermore, we have integrated self-evaluation capabilities into the agent to enhance its decisions' reliability. This feature allows the agent to assess its own responses, ensuring they align with the user's needs and preferences. By continuously validating its outputs, the agent can provide more accurate and trustworthy results. These evaluation capabilities rely on comparing the personal agent's output with the user's task. In cases where there is a discrepancy, and the agent retrieves results from past tasks, the agent must provide a reason for the retrieval.

3.2 Smart Building Side

`UserCentrix` framework for smart building operates under a hybrid hierarchical structure that integrates centralized and distributed control to optimize task management as shown in Fig.1. At the core of this framework, **Decision-making Module** which consists of high-level agents that operate as rational utility-based entities. These agents select action that maximize overall utility, ensuring that the framework maintains responsiveness while optimizing precision based on the situation's urgency. This action include sub-tasks that will be executed by **Sub-tasks Execution Module**, a critical component consisting of low-level agents. These agents are responsible for generating commands to implement the sub-tasks, ensuring environmental adjustments based on real-time data from the smart building system. These commands are managed by the final module, **Management and Analysis Module**, which stores the generated commands in a message queue and dispatches them according to a predefined schedule. Additionally, it continuously detects changes in adjusted settings during the assigned time slot, generating new commands to correct any unexpected alterations. We provide a more detailed explanation of these modules in the following sections.

3.2.1 Decision-making Module:

This module consists of high-level agents. It initiates with **Classifier Agent** that creates dedicated time slices for user tasks, tailored specifically to the task's urgency and VoI according to time sensitivity. These time slices enable the design of workflows that are optimized for each task's specific requirements, allowing for efficient communication setups between low-level agents and determine appropriate levels of reasoning depth. This agent uses a tool to extract the current time and compares it with the plan time retrieved from the repository. tasks are classified into two urgency levels: *High-urgency Level* ($\mathcal{U} \geq \vartheta_1$) if the time difference between the plan time and the current time is less than two hours, and *Low-urgency Level* ($\mathcal{U} \leq \vartheta_1$) if it is two hours or more. The classifier agent's prompt is provided in the Appendix. 6.

In critical situations, the decision-making module prioritizes speed over precision through a **High-urgency Agent** ($\mathcal{A}_{High}$), enabling faster but less detailed decisions to meet real-time demands. Allocating additional time to such tasks could enhance precision, but the urgency necessitates quick responses. This agent generates a streamlined reasoning path tailored to the user's context with a primary focus on time-sensitive depth. This path represents a solution aimed at accelerating task completion by reducing the number of steps and minimizing the required LLM calls, while still achieving an acceptable level of task fulfillment. To achieve this, the agent simplifies tasks by reducing sub-tasks, prioritizes actions that yield the most impactful outcomes in the shortest time for faster decision-making, and reduces inter-dependencies to further accelerate task execution. When two or more LLM calls can be processed independently, they are assigned the same rank for parallel execution, grouping them to be processed simultaneously within the same task rank. This approach enhances responsiveness in time-sensitive scenarios, ensuring that critical tasks are addressed promptly and effectively. In the Appendix. 6, we provide the high-urgency agent's prompt.

In contrast, during less critical periods, the decision-making module can allocate more time to refine decisions through **Low-urgency Agent** ($\mathcal{A}_{Low}$), allowing for more accurate and resource-intensive computations that enhance overall decision quality. These agents function as meta-reasoning entities, employing a **Thinking about Thinking** approach to decision-making. They critically assess the decision-making process itself, aiming to identify the optimal balance between accuracy, speed, and efficiency. It begins by generating a diverse initial set of reasoning paths for the task, each varying in complexity, depth, and focus on different decision criteria. These paths represent multiple potential solutions, breaking the task into detailed sub-tasks that correspond to logical steps, including LLM calls.

These sub-tasks reflect a unique perspectives or focuses, each incorporating specific criteria or priorities. These priorities may involve environmental analysis and exploring the real-time status of resources relevant to the request, such as room availability, meal options, and environmental conditions (e.g., temperature suitability and lighting levels). Furthermore, another criteria could prioritize resources whose conditions match or are closest to the user's specified preferences.

Another criteria could leverage natural and smart adjustments, such as suggesting opening curtains or blinds to enhance natural light if the user prefers natural lighting.

This variety enables a broad assessment of potential reasoning strategies, decision pathways, and resource allocation choices. The low-urgency agent's prompt is provided in the Appendix. 6. Low-urgency agent functions as a memory-augmented entity, operating through an iterative learning process. After generating potential solutions, both the solutions and the original task are stored in external memory. Simultaneously, the solutions are sent to an **Evaluator Agent** ($\mathcal{A}_{Evalu}$), which assesses them and selects the *most optimal solution* with logical *reasons*. If the evaluator agent determines that no path meets efficiency and success criteria, it provides actionable insights as *comments* to refine future solutions. For a new task, before the low-urgency agent generates new potential solutions, it first measures the semantic similarity between the current task embeddings and previously stored tasks embeddings in memory. If a highly similar task is found, its best solution and reasons and comments are retrieved and dynamically incorporated into the low-urgency agent's prompt as a hint to guide the new solution generation.

Within the learning loop, by recalling and dynamically injecting these past solutions based on context similarity, the agent refines its potential solutions-generating process. This iterative feedback loop allows the low-urgency agent to continuously improve by using in-context learning to incorporate its best previous solutions. Through this process, it also learns the key factors that led to better outcomes, allowing it to generate increasingly effective solutions over time.

The final potential solutions will be evaluated by **Pareto Analyzer**, which employs multiple fitness functions to optimize decision-making. These functions include evaluating semantic similarity, measuring the precision of the LLM's output, and analyzing the cost associated with LLM calls. These functions assess whether the additional computational resources and extended inference time required for higher precision provide significant utility and value in decision-making quality.

We chose the semantic similarity as it reflects how effectively a solution fulfills the user's task. Additionally, we selected precision as it measures the overlap between the generated response and the original task, indicating the degree of match between them. Meanwhile, LLM call usage cost measures the efficiency of resource consumption, quantified by the number of LLM calls made. Each additional call contributes to the overall time and computational cost, with higher call counts typically indicating greater resource usage (e.g., time and computation cost). We calculates a cost metric based on the number of LLM calls (N_{calls}) and maximum calls (N_{max}) of task solutions.

$$\text{LLM Call Usage Cost} = 1 - \exp\left(-\frac{Ncalls}{Nmax}\right) \tag{1}$$

Eq.(1) creates a decay effect where the cost scales non-linearly as the LLM call count increases. For small call counts, the cost increases slowly, but it rises more steeply as the call count approaches maximum. This encourages minimizing resource consumption relative to available limits. To identify optimal solution, we applied *Pareto dominance*, aiming to minimize resource costs while maximizing semantic similarity ($\mathcal{S}$) and precision score.

$$Pareto = \min(LLMCallUsageCost), \max(\mathcal{S}), \max(Precision) \tag{2}$$

This approach allows us to achieve the most efficient trade-offs between accuracy and resource efficiency. By assessing trade-offs between accuracy and resource usage, the agent dynamically allocates computational resources based on expected utility. By ranking solutions based on *Pareto* efficiency, the agent identifies the strongest solution that dominate other solutions in terms of the selected criteria. The final decision will include a solution that must execute by low-level agents. The number of sub-tasks determines whether a single low-level agent can manage the task independently or whether multiple low-level agents should collaborate to tackle more complex tasks that require deeper reasoning.

Time complexity of Algorithm 1

Estimating the time complexity for AI-agents can be challenging, but we approach it using a step-count method based on the algorithm's structure. The time complexity of this algorithm is primarily driven by the number of user tasks (m) and the operations performed for each task. For each task, embedding computation is a key operation, with a complexity of $O(d)$, where d represents the embedding dimension. Determining whether a task is high or low urgency requires constant time $O(1)$ and does not significantly impact the overall complexity. High urgency tasks have constant complexity since they involve only quick solution generation without additional operations. However, low urgency tasks generate n sub-task solutions, and each sub-task solution performs semantic similarity calculation (using llama-index) with complexity $O(s)$, where s depends on the embedding dimension of sub-task solutions. Additional operations for precision calculation, LLM Call Usage Cost computation, and Pareto optimization each require constant time $O(1)$. Thus, the time complexity for n sub-task solutions can be expressed as $O(n \times s)$. Memory recall and injection

Algorithm 1 `UserCentrix` Decision-Making Module

Require: User_tasks, $T = \{1 : m\} \forall T_i = \mathcal{D}_i \exists \mathcal{D}_i \in \mathcal{D}$ and $i \in \{1 : m\}$
Ensure: $\mathcal{T} = \{\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_n\}$ where each $\mathcal{T}_i$ is a sub-task for $i \in \{1, 2, \ldots, n\}$.

1: **for** each $T_i \in T$ **do**
2: Classify = $\mathcal{D}_i - t$ ▷ t is current time.
3: **if** (Classify ≤ 0) **then**
4: $T = T - \{T_i\}$ ▷ T_i is removed from task list.
5: **else if** (Classify $\geq \vartheta_1$) **then** ▷ ϑ_1 is Threshold.
6: $\mathcal{U}_i = 0$; ▷ Low urgency level
7: $\mathcal{A}_{\text{Low}} \longleftarrow T_i$ ▷ Send task to low-urgency agent
8: $M_{Low} \longleftarrow$RecallMemory($\mathcal{A}_{\text{Low}}$)
9: $\mathbf{E}_{\text{new}} \longleftarrow$ Embed_all-MiniLM-L6-v2($\mathcal{T}_i$) ▷ Calculate embedding of new solution
10: $\mathbf{E}_{\text{past}} \longleftarrow$ Embed_all-MiniLM-L6-v2($\mathcal{M}_{\text{Low}}$) ▷ Calculate embeddings of past solutions
11: Similarity($\mathbf{E}_{\text{new}}, \mathbf{E}_{\text{past}}$) $\longleftarrow \frac{\mathbf{E}_{\text{new}} \cdot \mathbf{E}_{\text{past}}}{\|\mathbf{E}_{\text{new}}\| \|\mathbf{E}_{\text{past}}\|}$ ▷ Calculate similarity between new and past solution embeddings
12: **if** Similarity($\mathbf{E}_{\text{new}}, \mathbf{E}_{\text{past}}$) $\leq \vartheta_2$ **then** ▷ $\vartheta_2 = 0.7$ in our work
13: Generate n potential solutions for $\mathcal{T}_i$ ▷ Generate solutions from scratch
14: **else**
15: Retrieve solution $\mathcal{E}_i$ with the reason $\mathcal{R}_i$ and factors $\mathcal{F}_i$ to $\mathcal{T}_i$ by $\mathcal{A}_{\text{Low}}$ ▷ Inject corresponding solution with the reason and factors provided by evaluator agent of this high similarity task into the agent's prompt
16: Generate n potential solutions for $\mathcal{T}_i$ using $\mathcal{E}_i \& \mathcal{R}_i \& \mathcal{F}_i$ ▷ Generate solutions by leveraging previous responses from the evaluator agent
17: **end if**
18: **for** each $\mathcal{T}_i$ **do**
19: Calculate semantic similarity ($\mathcal{S}(\mathcal{T}_i)$): using `llama-index`
20: Precision ($\mathcal{T}_i$) = $\frac{TP}{TP+FP}$
21: $LLMCallUsageCost(\mathcal{T}_i) = 1 - \exp\left(-\frac{N_{calls}}{N_{max}}\right)$
22: Pareto($\mathcal{T}_i$) = $\min(LLMCallUsageCost(\mathcal{T}_i)), \max(\mathcal{S}(\mathcal{T}_i)), \max(Precision(\mathcal{T}_i))$
23: $\mathcal{E}_i, \mathcal{R}_i, \mathcal{F}_i \longleftarrow \mathcal{A}_{Evalu}(\mathcal{T}_i)$ ▷ Evaluator agent selects the most optimal solution with giving reason and factors
24: $\mathcal{M} \longleftarrow \mathcal{M} \cup \mathcal{E}_i, \mathcal{R}_i, \mathcal{F}_i$ ▷ Inject evaluator agent' response into memory
25: $\mathcal{M} \longleftarrow \mathcal{M} \cup \mathcal{T}_i$ ▷ Inject task into memory
26: **end for**
27: **else**
28: $\mathcal{U}_i = 1$; ▷ High urgency level
29: $\mathcal{A}_{\text{High}} \longleftarrow T_i$ ▷ Send task to high-urgency agent
30: $\mathcal{A}_{\text{High}}$ generates a quick solution $\mathcal{T}_i$ for task T_i
31: **end if**
32: **end for**
33: **return** $\mathcal{T}$

operations depend on the size of memory i.e., k, which express $O(k)$ to the complexity. Overall, the time complexity of Algorithm 1 can be expressed as $O(m \times (k + d + n \times s))$.

To clarify the workflow of the *`UserCentrix` Decision-Making Module*, we present Algorithm 1. For each task T provided by the user, the classifier agent determines its urgency level by computing the difference between the task's deadline $\mathcal{D}_i$ and the current time t (Lines 1 & 2). If the resulting classification score is below zero, indicating that the task is no longer relevant, it is removed from the task list (Lines 3 & 4). Otherwise, tasks are further categorized into low or high urgency based on a predefined threshold ϑ_1 (Lines 5-6 & 27-28). Low-urgency tasks are assigned to a low-urgency agent $\mathcal{A}_{\text{Low}}$ (Line 7), which retrieves relevant past memory and computes semantic embeddings of both current and previous solutions using MiniLM (Lines 8–11). A similarity score is then calculated between the new and past embeddings. If the similarity is below a specified threshold ϑ_2, the system proceeds to generate n potential new solutions (Lines 12 & 13); otherwise, it retrieves suitable existing solution with the reason and factors provided by the evaluator agent (Line 15) and use them as hints to generate potential solutions (Line 16). Each solution is subsequently evaluated in terms of semantic similarity (using LlamaIndex), precision, and usage cost (Lines 18–21). A Pareto optimization method is employed to select the most optimal solution $\mathcal{E}_i$ (Line 22). The evaluator agent is responsible for assessing the generated solutions and selecting the most appropriate one with its logical reasoning (Line

Algorithm 2 Sub-tasks Execution Module

Require: k, Sub-tasks Solutions $\mathcal{T} = \{1 : k\}$ and Dataset $D = \{1 : \Delta\}$
Ensure: Command $\mathcal{C}$
1: Generate k low-level agents i.e., $A = \{A_i \mid A_i \text{ executes } \mathcal{T}_i, \forall i \in \{1, 2, \ldots, k\}\}, \forall A_i$ handling a $\mathcal{T}_i$
2: **for** $\forall \mathcal{T}_i \in A_i$ **do**
3: Retrieve the dataset $D_i = \{\delta \in D \mid \mathbf{1}_{\text{compatible}}(\delta, \mathcal{T}_i) = 1\}$ ▷ $\mathbf{1}_{\text{compatible}}$ give appropriate dataset
4: $\mathcal{C}_i \longleftarrow$ Execute$(\mathcal{T}_i, D_i)$.
5: $\mathcal{C} \longleftarrow \mathcal{C} \cup \{\mathcal{C}_i\}, \forall i \in \{1, 2, \ldots, k\}$.
6: **end for**
7: Return commands $\mathcal{C}$

23), which will be injected along with the corresponding task into the memory module $\mathcal{M}$ (Lines 24 & 25). In contrast, high-urgency tasks $\mathcal{A}_{\text{High}}$ are directed to the high-urgency agent, which quickly generates suitable solutions to meet tight deadlines (Lines 29 & 30). The module concludes by returning the final set of updated sub-tasks $\mathcal{T}_i$ (Line 33).

3.2.2 Sub-tasks Execution Module

After selecting the solution for each task, Sub-tasks Execution Module form groups of low-level agents to execute the sub-tasks and generate commands to adjust smart building settings. These groups operate in parallel to improve decision-making speed, while considering the execution time of each task. In the case of no execution conflict between tasks, the low-level agents of each group generate commands based on the hierarchical structure of the order of execution. The agents within each group share their responses from one level to another. If parallel sub-tasks are performed, tasks have access to responses from prior tasks in the order at same time.

In cases where execution conflicts arise, we establish a meta-cooperative reasoning network with a distributed setup. In this network, agents from different groups negotiate and work in parallel, sharing intermediate reasoning results to avoid conflicts, such as selecting the same room at the same time. This collaborative approach enhances both the speed and accuracy of the responses by leveraging cooperative problem-solving among agents.

Time complexity for Algorithm 2

In Algorithm 2, we perform agent generation, dataset selection, task execution, and solution aggregation. Our algorithm uses k agents based on sub-task solutions, which has a linear complexity of $O(k)$ for initialization. For each agent, we need to select the most appropriate dataset from a collection of datasets (our case we assume, Δ datasets). This selection process takes $O(\Delta)$ time for each agent, as we need to evaluate the compatibility of each dataset. Once a dataset is selected, the execution step (Execute$(\mathcal{T}i, D_i)$) processes the chosen dataset. Let's denote the size of the largest dataset as η. In the worst case, the execution time for each agent would be $O(\eta)$. These operations are performed for each of the k agents. The final step of aggregating solutions into $\mathcal{C}$ takes constant time $O(1)$ per agent. Therefore, the overall time complexity of Algorithm 2 can be expressed as $O(k \times (\Delta + \eta))$.

Algorithm 2 presents the *Sub-tasks Execution Module*, which is designed to execute a set of sub-task solutions $\mathcal{T} = \{1 : k\}$ using a corresponding dataset D. The execution process begins with the creation of k low-level agents, where each agent is assigned a specific sub-task $\mathcal{T}_i$ (Line 1). For each sub-task $\mathcal{T}_i$, the respective agent retrieves a compatible dataset D that satisfies the task's requirements (Line 2 & 3) to perform the execution of the sub-task (Line 4). The resulting commands are collected and aggregated into a final command set $\mathcal{C}$ (Line 5), which is returned upon completion of the execution phase (Line 7).

3.2.3 Management and Analysis Module

Management and Analysis Module include a message queue to store generated commands, including support for time-to-launch (TTL) settings. It aggregates all commands generated from low-level agents. Based on the scheduled task time, the message queue dispatches these commands to the control system, managing both the actuators and the environment agent. The environment agent is designed to track ongoing tasks and ensure they align with user preferences. It compares the user's specified requirements for the task with the current status of the booked resources. If any changes are detected, the agent generates an alert command and sends it to the control system. This approach ensures user satisfaction and QoE remain high by proactively addressing potential issues during task execution. The environment agent's prompt is provided in the Appendix. 6.

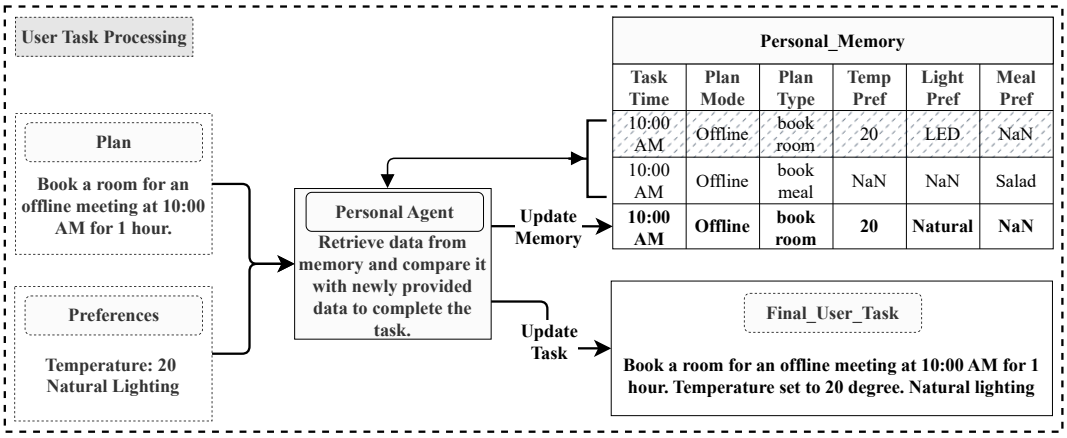


Figure 2: User Task Processing within `UserCentrix` Framework.

3.3 Use Case

3.3.1 User Task Processing Scenario:

As shown in Fig. 2 the personal agent is equipped with an external personal memory that serves as a repository. It initially starts empty and gradually fills up as the user submits tasks. This repository retains information about past tasks, including user preferences. In this scenario, the user has previously submitted two tasks, each with its own preferences stored in the repository. When the user submits a new plan without specifying preferences, the agent evaluates the semantic similarity between the new plan type embeddings and those stored in personal memory. It also compares the plan's timestamp with previous task timestamps. If the time difference is less than one hour and the similarity score exceeds 0.5, the agent retrieves preferences from the most recent matching entry in personal memory. The agent then updates the new task to incorporate these preferences, allowing it to adapt its responses based on user history. This process ensures a context-aware experience, enabling more personalized and relevant task updates.

3.3.2 High-urgency Scenario:

Fig. 3 illustrates the workflow of the framework's building modules, beginning with the classifier agent, which determines the urgency level of a submitted user task. In this scenario, the agent classified the task as high urgency because the time difference between the current time and the task time was less than two hours. It then forwarded the task to the high-urgency agent, which is responsible for generating a time-sensitive solution with minimal sub-tasks. This agent identifies the necessity for two separate LLM calls, each corresponding to a distinct sub-task. These sub-tasks are organized into a hierarchical structure with two levels, enabling efficient task decomposition and execution.

Next, the Sub-task Execution Module is activated, generating two agents—one for each sub-task. Each agent executes its assigned sub-task, issuing commands to book a specific room and adjust environmental settings based on user needs. The agents use the Smart Campus dataset to ensure accurate configurations.These generated commands are placed into a message queue within the Management and Analysis Module, which then forwards them to actuators for execution. Additionally, environment agents continuously monitor any changes between user preferences and the Smart Campus dataset during the booking period. If changes are detected, they generate new commands to adjust the environment accordingly, ensuring real-time adaptation to user needs.

3.3.3 Low-urgency Scenario:

Fig. 4 illustrates the workflow of the framework's building modules, beginning with the classifier agent, which determines the urgency level of a submitted user task. In this scenario, the agent classified the task as low urgency because the time difference between the current time and the task time was more than two hours. It then forwarded the task to the low-urgency agent, which is responsible for generating all possible reasoning solutions for each task in a smart building, such as booking rooms, scheduling meals, or adjusting environmental settings with leveraging from the best solutions stored in memory with the reason and factors which impact of the selecting the solution as best as well as insights to refine future solutions. Before generating solutions, the low-urgency agent retrieves memory to check for previously stored tasks. If any are found, it calculates the similarity between the current plan embeddings and the

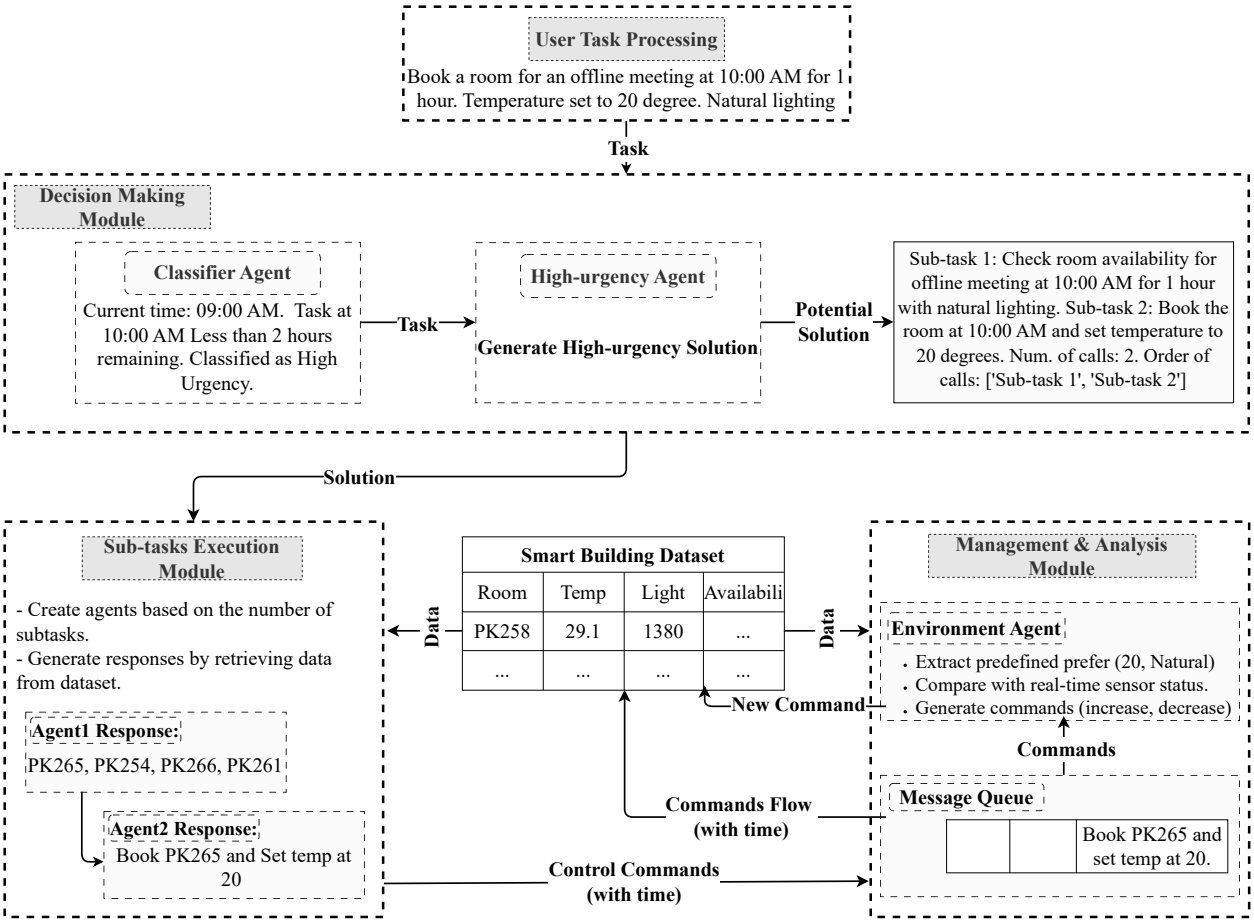


Figure 3: High-urgency Workflow within `UserCentrix` Framework.

embeddings of past plans. If a highly similar task is identified, the corresponding solutions, along with the reasons and comments provided by the evaluator agent, are injected into the prompt as hints for generating new solutions.

After generating several solutions based on different criteria, the evaluation process begins. This process involves two key components: the pareto analyzer and the evaluator agent. Pareto analyzer applies fitness functions to each solution and calculates corresponding values. The goal is to identify the solution that achieves maximum semantic similarity, maximum precision score, and minimum cost. Meanwhile, evaluator agent selects the best solution and stores it in memory along with the reason and factors influencing the decision. Once the pareto analyzer completes its evaluation, it forward the solution to the Sub-task Execution Module, which generates three agents based on the number of sub-tasks. Each agent executes its assigned sub-task, issuing commands to book a specific room and adjust environmental settings based on user needs using the Smart Campus dataset. These generated commands are placed into a message queue within the Management and Analysis Module, which then forwards them to actuators for execution. Additionally, environment agents continuously monitor any changes between user preferences and the Smart Campus dataset during the booking period. If changes are detected, they generate new commands to adjust the environment accordingly, ensuring real-time adaptation to user needs.

4 Implementation

In practice, we perform all experiments on a desktop equipped with an Intel(R) Core(TM) i5-1135G7 CPU is assumed as edge, and simultaneously on Google Colab using Intel(R) Xeon(R) CPU treated as cloud. We use the *University of Oulu* as our experimental setting, leveraging data from the *University of Oulu Smart Campus* Dataset [40]. Specifically,

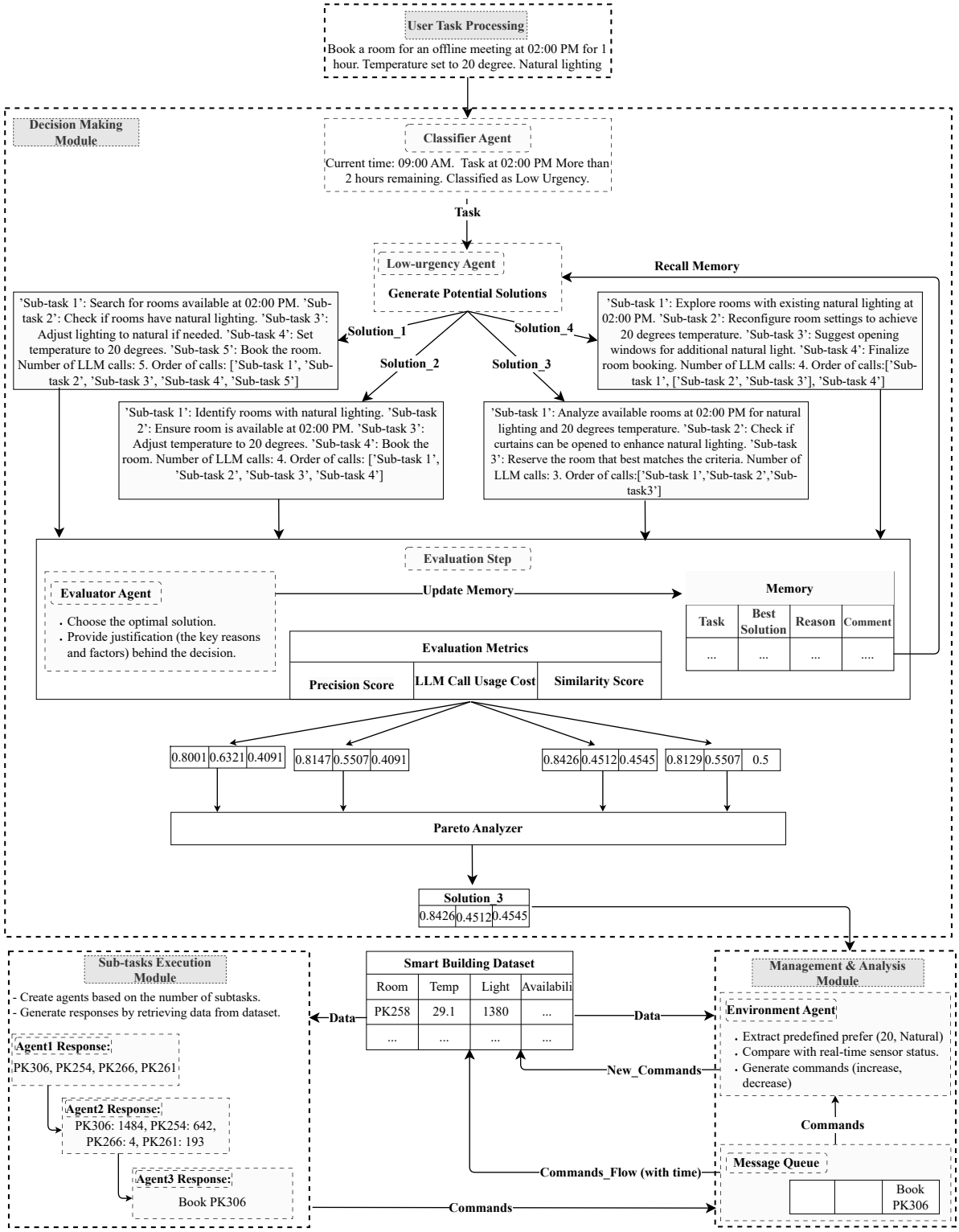


Figure 4: Low-urgency Workflow within `UserCentrix` Framework.

we focus on meeting rooms equipped with *Elsys ERS CO2* sensors, which provide comprehensive indoor environmental measurements, including motion, temperature and light intensity. These sensors are calibrated before deployment, and their data quality is extensively validated to ensure reliability and accuracy [41]. The selected rooms include: *TS501, PK258, PK265, PK306, PK254, PK266, PK261, PK253, PK267, PK262, PK309, PK268, PK263, PK308, and PK264*. In addition to the selected meeting rooms, since our goal is to enable low-level agents to identify rooms that match user preferences defined by room temperature, light status, and availability, we generated a synthetic dataset. This synthetic dataset includes room availability based on typical working hours, along with temperature and light intensity, and will be used by low-level agents for room selection and by the environment agent for ongoing environmental changes tracking.

All implemented agents in the experiments are built using *LangChain*. We developed the memory as a custom repository type using *LangChain*[1].

For embeddings and similarity, we use *all-MiniLM-L6-v2* model[2] which is designed for semantic textual similarity (STS) tasks. It creates embeddings (vector representations) for sentences to capture their semantic meaning. These embeddings allow the model to compute similarity scores between texts.

To enable the evaluator agent to assess the potential solutions generated by the low-urgency agent and select the most optimal one, we employ the *o1 model*[3], due to its advanced reasoning capabilities, ensuring that the chosen solution aligns best with the original task objectives. We utilized *Pareto dominance* with the *paretoset 1.2.4* library [4]. For fitness functions, semantic similarity was evaluated using the *LlamaIndex* framework[5], while precision was assessed using the *ragas* framework[6].

We selected models that allow for a comprehensive evaluation of reasoning capabilities across a diverse range of both large and small language models. LLMs can play a crucial role in facilitating user-centric interactions and dynamically scaling computational resources. However, the increasing focus on SLMs or on-device LLMs highlights their potential in enhancing latency and delivering personalized user experiences. These models, typically contain fewer parameters, are optimized for deployment on edge devices, enabling responsive technologies such as smart environments and real-time applications. In our experiment, we incorporate the following language models:

- *Gemini 1.5 Flash* (8B), a lightweight model, developed by *Google DeepMind*[7]. This model is selected due to its advanced capabilities in long-context reasoning, and optimized for low-latency performance and enhanced efficiency in agentic interactions.
- *GPT-4o*[8], a large model developed by *OpenAI*. It is incorporated due to its advanced reasoning capabilities, particularly in real-time analysis, making it well-suited for real-world applications.
- *Claude 3.5 Sonnet*(8.03B)[9], developed by *Anthropic* and known for its strong agentic capabilities.
- *Command-r7b* (8.03B)[10], the smallest model in *Cohere*'s R series with powerful agentic capabilities, is optimized for diverse use cases, including deployment on edge devices.
- *Mistral* (7.25B)[11], an open-source model developed by *Mistral AI* with advanced reasoning capabilities and rapid inference speed.
- *IBM*'s Granite models[12], with *granite3.1-MoE* (3B) which employs a mixture-of-experts (MoE) architecture, making it particularly suitable for low-latency applications.

As there are no prior studies in the literature that address the same problem while considering the specific requirements and objectives of `UserCentrix` framework, our primary objective is to evaluate agents' responses by analyzing elapsed time, CPU usage, and memory utilization, along with various metrics associated with each module in our framework. As well as, we compare the accuracy of responses generated by different agents across all modules against a baseline model to measure improvements or deviations. For more details:

[1] `https://www.langchain.com/`
[2] `https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2`
[3] `https://openai.com/o1/`
[4] `https://pypi.org/project/paretoset/`
[5] `https://docs.llamaindex.ai/en/stable/`
[6] `https://docs.ragas.io/en/latest/concepts/metrics/`
[7] `https://deepmind.google/technologies/gemini/`
[8] `https://platform.openai.com/docs/models`
[9] `https://www.anthropic.com/news/claude-3-5-sonnet`
[10] `https://cohere.com/blog/command-r7b`
[11] `https://mistral.ai/`
[12] `https://www.ibm.com/granite/`

- **User Task Processing Module:** We evaluate the personal agent's responses in analyzing the user's task when executed on a cloud server and an edge device, measuring elapsed time, CPU usage, and memory utilization across various models. Additionally, we assess accuracy in two scenarios, when memory is empty and when it is full, while determining whether the agent retrieves the relevant information from memory or operates without memory access. This assessment is based on criteria that we incorporated into the agent prompt 6 from the primary evaluation template available at[13].
- **Decision-Making Module:**
  1. **Classifier Agent Performance:** We evaluate the classifier agent's responses in determining the urgency level of different tasks as either <High>or <Low>when executed on a cloud server and an edge device, measuring elapsed time, CPU usage, and memory utilization across various models. Additionally, we measure factual correctness in precision mode by comparing the agent's responses across various models to its response with the o1 model.
  2. **Performance of Low-urgency and High-urgency Agents:** We evaluate the performance of high-urgency and low-urgency agents in solution generation when executed on a cloud server and an edge device by measuring elapsed time, CPU usage, and memory utilization across various models.
  3. **Low-urgency Agent Performance:** We verify whether the optimal solution selected by Pareto aligns with the preferred requirement chosen by the o1 model. Additionally, we assess the agent's response improvement through an in-context learning loop using the o1 model's response.
- **Sub-tasks Execution Module:** We evaluate the performance of low-level agents in executing sub-tasks when deployed on a cloud server and an edge device, measuring elapsed time, CPU usage, and memory utilization across various models.
- **Management and Analysis Module:** We evaluate the environment agent's responses in detecting changes and generating the commands when executed on a cloud server and an edge device, measuring elapsed time, CPU usage, and memory utilization across various models. Additionally, we measure Factual Correctness in recall mode to assess how well the environment agent's response aligns with the O1 model's response in generating the correct commands.

This evaluation framework ensures a comprehensive assessment of model performance in terms of efficiency, accuracy, and decision-making quality.

We utilize the factual correctness metric from RAGAS[14], leveraging GPT-4o. This metric evaluates the factual accuracy and alignment of generated responses with a reference, ranging from 0 to 1, where higher values indicate superior performance. The precision is calculated using the Eq.(3):

$$\text{Precision} = \frac{TP}{TP + FP} \tag{3}$$

Meanwhile, the recall is calculated using the Eq.(4):

$$\text{Recall} = \frac{TP}{TP + FN} \tag{4}$$

Where True Positive (TP) is number of claims in the response that are present in the reference, False Positive (FP) is number of claims in the response that are not present in the reference, and False Negative (FN) is number of claims in the reference that are not present in the response.

We select precision and recall as evaluation metrics because they align with the nature of references and responses in our framework.

- Classifier Agent: Precision is used to determine the proportion of responses that differ from the reference, distinguishing between *High* and *Low* variations in the output.
- Environment Agent: Recall is chosen to evaluate the accuracy of commands (e.g., increase or decrease), the degree of change, and other key features. Since complete coverage of the response relative to the reference is essential, recall ensures a thorough assessment of correctness and consistency.

[13] `https://github.com/langchain-ai/langchain/blob/master/libs/langchain/langchain/evaluation/criteria/prompt.py`

[14] `https://docs.ragas.io/en/stable/`

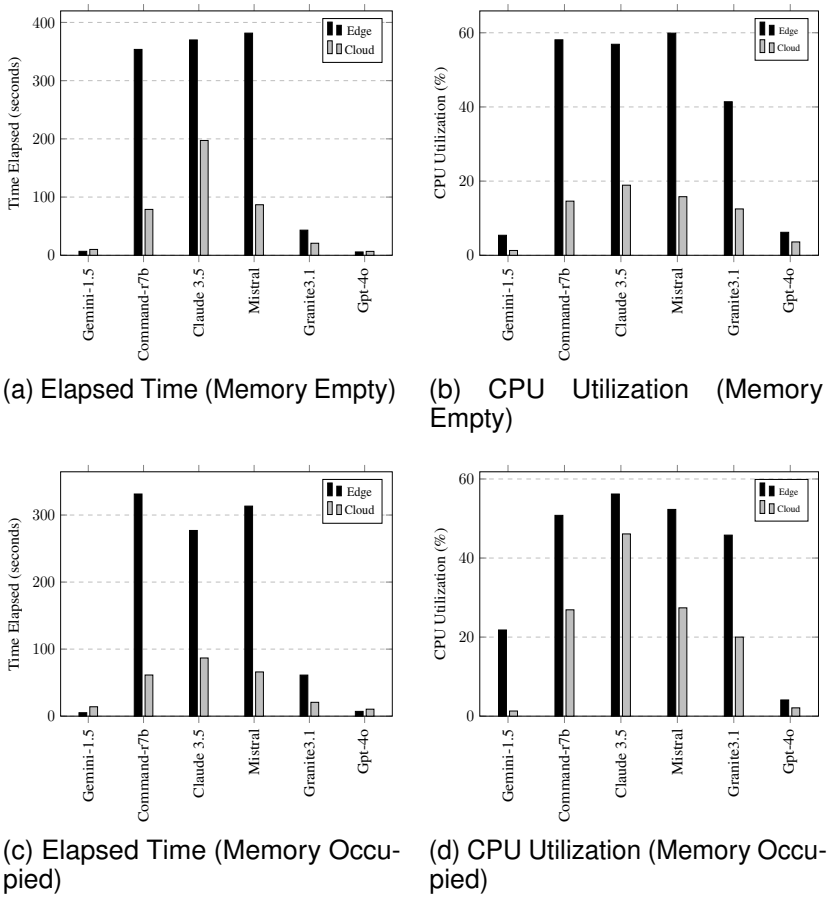
Table 2: Personal Agent Performance Evaluation.

Model Name	Memory State												
	Empty						Occupied						
	Cloud Server			Edge Device			Cloud Server			Edge Device			
	Elapsed Time	CPU Utilization	Memory Utilization	Elapsed Time	CPU Utilization	Memory Utilization	Elapsed Time	CPU Utilization	Memory Utilization	Elapsed Time	CPU Utilization	Memory Utilization	Accuracy
Gemini-1.5 flash	10.034	1.30%	6.50%	6.801	5.40%	42.20%	14.018	1.30%	6.60%	5.1216	21.80%	43.90%	✓
command-r7b	78.8173	14.60%	20.80%	353.6496	58.10%	67.70%	61.3768	26.90%	20.70%	331.3826	50.80%	67.30%	✗
Claude 3.5 Sonnet	97.228	18.90%	32.60%	369.9368	56.90%	65.90%	86.7674	46.10%	19.80%	276.9919	56.20%	65.50%	✗
Mistral	86.8284	15.80%	18.00%	381.648	59.90%	63.20%	65.9482	27.40%	17.90%	313.3207	52.30%	63.30%	✗
granite3.1-MoE	20.6631	12.50%	8.90%	43.1752	41.40%	43.50%	20.6424	20.00%	9.00%	61.2677	45.80%	43.40%	✓
Gpt-4o	6.708	3.60%	9.10%	5.738	6.20%	41.80%	10.4106	2.10%	9.10%	7.11	4.10%	41.70%	✓

5 Results Analysis

In this section, we present a comprehensive evaluation of the framework's performance across various modules. The analysis aims to assess the effectiveness, efficiency, and robustness of the proposed approach through multiple evaluation criteria.

5.1 Personal Agent Performance Evaluation



(a) Elapsed Time (Memory Empty)

(b) CPU Utilization (Memory Empty)

(c) Elapsed Time (Memory Occupied)

(d) CPU Utilization (Memory Occupied)

Figure 5: Personal Agent Performance Evaluation.

The performance evaluation of the personal agent across different models highlights significant variations in execution efficiency, resource utilization, and accuracy in retrieving relevant information when memory is occupied, as summarized in Table 2. Fig. 5 shows that GPT-4o consistently outperforms other models, showing the shortest elapsed time and the lowest CPU utilization across both cloud and edge devices, demonstrating its superior efficiency in processing user tasks. Additionally, GPT-4o and Gemini-1.5 flash maintain lower memory utilization, making them highly suitable for deployment in low-memory environments. In contrast, models such as Claude 3.5 Sonnet and Mistral exhibit

Table 3: Classifier Agent Performance Evaluation.

Model Name	Cloud Server			Edge Device			Accuracy
	Elapsed Time	CPU Utilization	Memory Utilization	Elapsed Time	CPU Utilization	Memory Utilization	
Gemini-1.5 flash	5.3925	23.60%	14.40%	2.4466	7.90%	31.30%	1
command-r7b	119.7979	6.80%	16.00%	127.8999	62.40%	71.90%	0.75
Claude 3.5 Sonnet	119.2964	33.30%	15.20%	118.6518	58.50%	70.50%	0.75
Mistral	88.08312	5.00%	13.50%	143.5072	61.20%	66.30%	1
granite3.1-MoE	61.7082	23.30%	32.40%	16.4528	59.30%	55.80%	0.5
Gpt-4o	6.33347	9.80%	12.60%	8.671	8.70%	42.20%	1

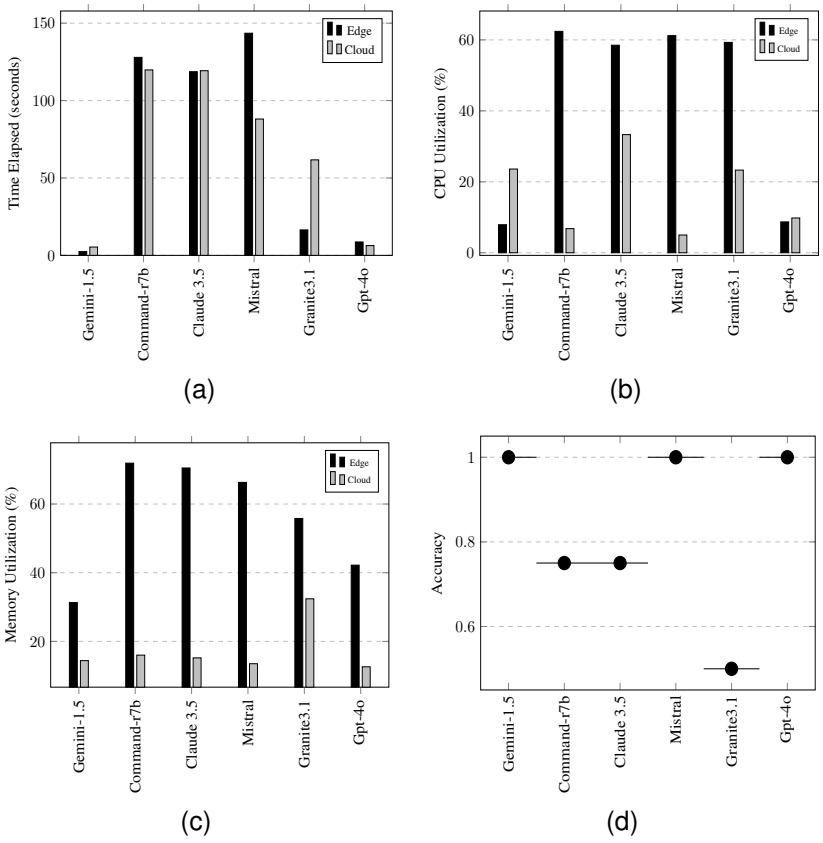


Figure 6: Classifier Agent Performance Evaluation.

significantly higher CPU and memory consumption, which poses challenges for deployment on resource-constrained edge devices, potentially hindering real-time processing.

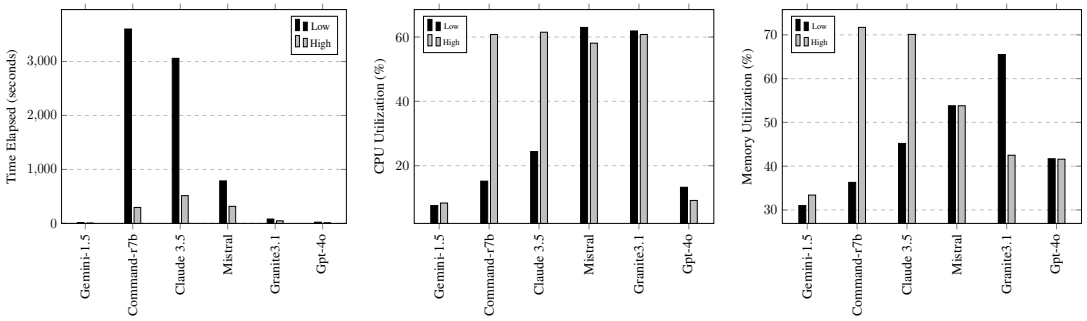
When memory is occupied, the agent should retrieve relevant stored information instead of reprocessing the task from scratch, a critical factor in enhancing user experience. GPT-4o and Gemini-1.5 flash effectively retrieve stored information, ensuring optimal performance and minimizing redundant computation. Moreover, Claude 3.5 Sonnet, Mistral, and command-r7b fail to retrieve memory-stored information, leading to inefficiencies, increased computational overhead, and degraded performance as shown in Table 2.

5.2 Classifier Agent Performance Evaluation

The evaluation of the classifier agent's performance across different models demonstrates substantial variations when categorizing tasks into high and low urgency levels. The results, summarized in Table 3, highlight key differences in elapsed time, CPU usage, memory consumption, and factual correctness when compared against the o1 model baseline.

Table 4: Evaluation of Low-urgency and High-urgency Agent Performance.

Model Name	Cloud Server						Edge Device					
	Elapsed Time		CPU Utilization		Memory Utilization		Elapsed Time		CPU Utilization		Memory Utilization	
	Urgency Level											
	Low	High	Low	High	Low	High	Low	High	Low	High	Low	High
Gemini-1.5 flash	10.756	4.339	1.40%	1.50%	6.60%	6.70%	14.1069	4.8304	7.60%	8.40%	31.00%	33.40%
command-r7b	94.521	53.97	40.30%	31.80%	19.90%	19.90%	3601.6761	295.8871	15.20%	60.80%	36.30%	71.70%
Claude 3.5 Sonnet	149.222	74.278	8.30%	24%	18.30%	18.30%	3058.6028	514.0639	24.40%	61.50%	45.20%	70.10%
Mistral	151.503	47.847	41.60%	17.20%	17.40%	18.10%	787.05659	316.2107	63.00%	58.10%	53.80%	53.80%
granite3.1-MoE	49.2456	17.9563	36.70%	17.60%	9.10%	9.10%	80.8196	47.7772	61.90%	60.80%	65.50%	42.50%
Gpt-4o	25.938	6.171	1.60%	1.40%	17.80%	6.80%	22.227	10.1647	13.30%	9.20%	41.70%	41.60%



(a) Elapsed Time (b) CPU Utilization (c) Memory Utilization

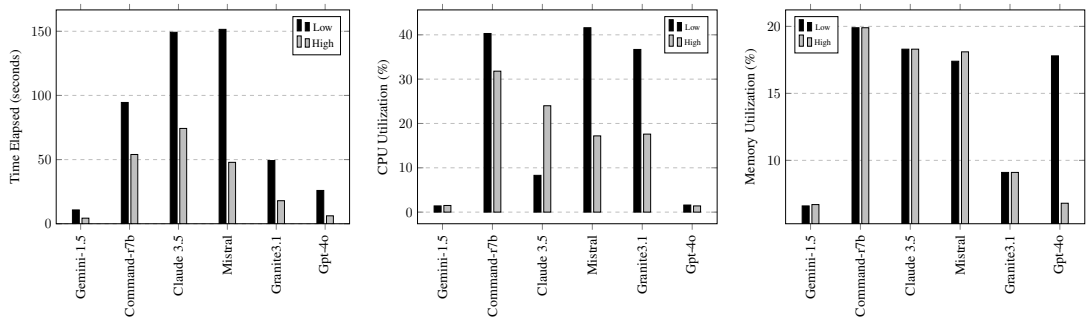
Figure 7: Evaluation of Low-urgency and High-urgency Agent Performance at the Edge.

Fig. 6 shows that GPT-4o and Gemini-1.5 flash emerge as the most efficient models, achieving an optimal balance between execution speed, low resource utilization, and high classification accuracy, making them well-suited for real-time deployment. These models demonstrate the shortest elapsed time and the lowest memory consumption across both cloud and edge devices, highlighting their effectiveness in performing classification tasks with minimal computational overhead.

In contrast, Claude 3.5 Sonnet, Mistral, and command-r7b demonstrate significantly higher elapsed times and increased memory utilization, particularly on edge devices. This elevated resource consumption may introduce delays in real-time classification, making them less ideal for latency-sensitive applications. Furthermore, granite3.1-MoE's low accuracy underscores its limitations in high-stakes classification tasks where precision is paramount.

5.3 Evaluation of Low-urgency and High-urgency Agent Performance

The evaluation of low-urgency and high-urgency agents across different models highlights significant variations in elapsed time, CPU utilization, and memory consumption when generating solutions. As shown in Table 4, high-urgency



(a) Elapsed Time (b) CPU Utilization (c) Memory Utilization

Figure 8: Evaluation of Low-urgency High-urgency Agent Performance at the Cloud.

Table 5: Evaluation of Low-Level Agent Performance and Pareto Analysis.

Experiment												
				Metrics			Cloud Server			Edge Device		
Model Name	**Urgency Level**	**LLM Call Count**	**Hierarchy Depth Count**	**Similarity Score**	**LLM Call Usage Cost**	**Precision Score**	**Elapsed Time**	**CPU Utilization**	**Memory Utilization**	**Elapsed Time**	**CPU Utilization**	**Memory Utilization**
Gemini-1.5 flash	**Low**	3	3	0.8704	0.5276	0.3636	23.87156	6.10%	13.90%	3.67961	10.40%	51.60%
		4	**3**	**0.8659**	**0.6321**	**0.4091**	**15.7927**	**4.80%**	**14.00%**	**3.8502**	**7.00%**	**36.70%**
	High	3	3	0.8602	0.5276	0.4091	17.1799	4.60%	14.00%	3.5883	8.80%	36.40%
		2	2	0.9039	-	0.3636	3.7146	5.60%	14.00%	3.439471	15.50%	36.30%
command-r7b	**Low**	**2**	**2**	**0.8913**	**0.6321**	**0.4545**	**498.1659**	**30.00%**	**15.60%**	**625.2602**	**64.00%**	**73.10%**
		2	2	0.8860	0.6321	0.2727	492.8408	49.10%	15.70%	843.0399	60.80%	72.00%
	High	2	2	0.9183	-	0.5455	676.5058	49.80%	15.60%	616.2926	60.50%	67.60%
Claude 3.5 Sonnet	**Low**	5	5	0.8702	0.6321	0.3636	1689.107	42.70%	15.30%	2265.092	59.10%	67.70%
		4	**4**	**0.8488**	**0.5507**	**0.2727**	**2034.623**	**51.10%**	**14.90%**	**1621.683**	**57.50%**	**68.00%**
	High	4	4	0.8421	0.5507	0.2727	1569.989	52.60%	14.90%	1869.202	57.80%	69.10%
		2	2	0.8809	-	0.3636	643.4445	50.00%	14.40%	666.3432	57.50%	68.90%
Mistral	**Low**	3	3	0.8733	0.6321	0.4091	802.7946	41.70%	14.20%	949.0744	57.60%	66.70%
		3	3	0.8624	0.6321	0.3636	786.7635	51.60%	14.00%	946.0731	57.50%	66.80%
		2	**2**	**0.8752**	**0.4866**	**0.3636**	**491.5169**	**40.30%**	**14.10%**	**627.3577**	**57.90%**	**63.00%**
	High	2	2	0.8993	-	0.4091	514.6962	49.80%	14.00%	801.5293	58.70%	64.60%
granite3.1-MoE	**Low**	**3**	**3**	**0.8282**	**0.6321**	**0.3636**	**202.6593**	**37.20%**	**9.20%**	**285.3483**	**56.00%**	**51.60%**
		3	3	0.8270	0.6321	0.3182	179.3239	43.60%	9.20%	277.4193	55.70%	51.80%
		3	3	0.8144	0.6321	0.2727	124.8736	41.90%	9.20%	111.0891	56.50%	52.10%
		3	3	0.8191	0.6321	0.3636	196.0647	47%	9.30%	202.7976	56.40%	51.90%
	High	4	4	0.8858	-	0.4545	199.4555	46.50%	9.30%	297.2756	56.70%	52.00%
Gpt-4o	**Low**	3	3	0.7741	0.6321	0.4737	3.5381	2.70%	3.60%	7.6907	3.10%	52.90%
		3	3	0.7872	0.6321	0.4737	2.1682	3.30%	3.60%	6.5495	4.40%	53.40%
		2	2	0.8288	0.4866	0.4211	11.3214	2.40%	3.60%	3.6525	4.70%	52.90%
		3	**3**	**0.7937**	**0.6321**	**0.5263**	**13.8917**	**2.00%**	**3.60%**	**9.4319**	**2.30%**	**53.20%**
		3	3	0.7938	0.6321	0.4211	11.6523	3.30%	3.60%	3.4147	3.10%	52.60%
	High	2	2	0.8364	-	0.6315	1.9784	3.40%	3.60%	1.3812	5.60%	49.00%

agents generally execute tasks faster than their low-urgency agents, which is expected given their prioritization in processing. However, this efficiency often comes at the cost of increased CPU and memory utilization, particularly on resource-constrained edge devices. Fig. 7 and Fig. 8 show that GPT-4o and Gemini-1.5 flash models achieve the best trade-off between execution speed and resource utilization, making them ideal for real-time solution generation, especially in high-urgency scenarios. Conversely, Claude 3.5 Sonnet and Mistral struggle with high resource consumption, particularly on edge devices, limiting their feasibility for real-time applications.

5.4 Evaluation of Low-level Agent Performance and Pareto Analyzer

The evaluation of low-urgency and high-urgency agent performance highlights critical differences in reasoning-based solution generation, LLM call count, execution hierarchy depth, and resource efficiency across different models. Table 5 highlights the execution evaluation of low-level agents in terms of elapsed time, CPU utilization, and memory consumption.

Low-urgency agent generates multiple reasoning solutions, each associated with a varying LLM call count (different number of sub-tasks) and hierarchy depth count (parallel execution levels) as illustrated in Table 5. High-urgency agent prioritize immediate decision-making and reduce LLM call counts to minimize latency. Across all models except granite3.1-MoE, high-urgency level tasks consistently show lower LLM call counts and shallower hierarchy depths compared to low-urgency solutions, reinforcing the importance of minimizing computational overhead for real-time responsiveness.

While evaluating the trade-offs between semantic similarity, precision, and LLM call usage cost. Among the models, Claude 3.5 Sonnet demonstrates relatively low precision, making it less suitable for tasks requiring structured reasoning or high accuracy as shown Fig. 9c. In contrast, Gpt-4o deliver the highest precision scores, indicating their strength in producing accurate and reliable responses as shown Fig. 9f. Mistral, command-r7b, and Gemini-1.5 Flash consistently demonstrate their ability to generate contextually relevant and coherent solutions as shown Fig. 9d, Fig. 9b, and Fig. 9a. The gray-highlighted rows in the table represent Pareto-optimal solutions. The bold values indicate the final selections made by the o1 model after assessing each solution. These Pareto-optimal configurations consistently align with the o1 selections, confirming the system's ability to identify high-quality, resource-efficient outputs, particularly under low urgency conditions.

Meanwhile, after executing the sub-tasks using low-level agents, we evaluate the elapsed time, CPU utilization, and memory usage across different models to assess their performance under varying urgency levels. Fig. 10a, Fig. 10g

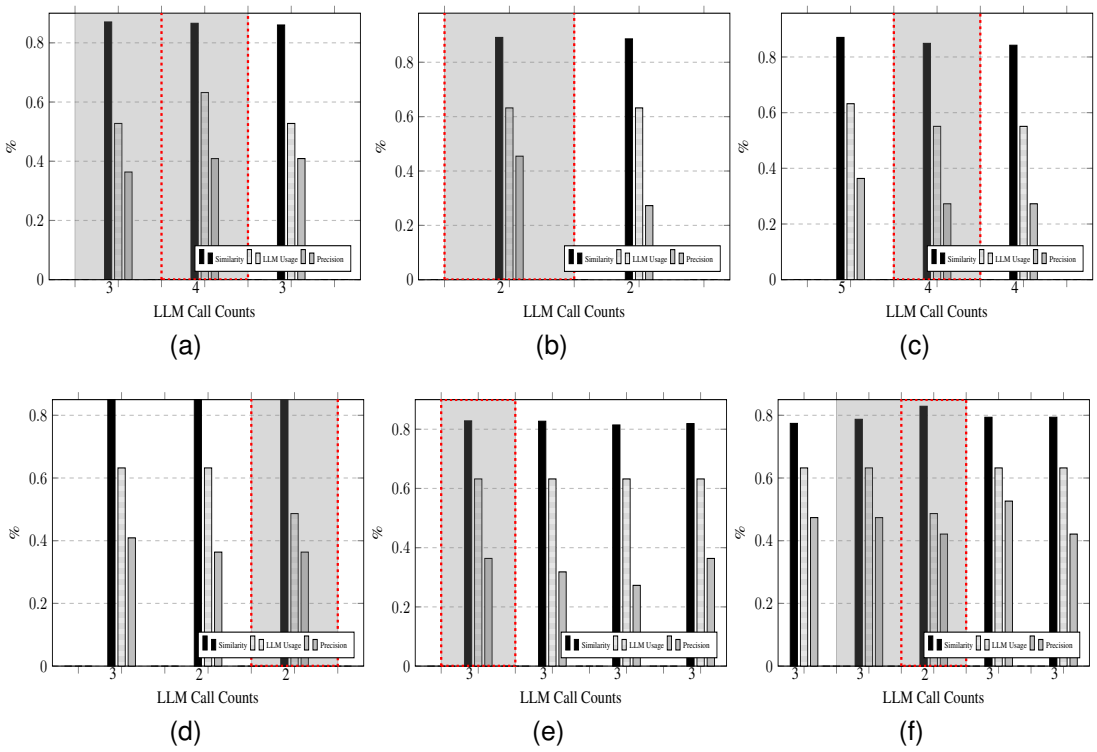


Figure 9: Evaluation of Low-Level Agent Performance Metrics and Pareto Analysis. (a) Gemini-1.5 flash (b) Command-r7b (c) Claude 3.5 Sonnet (d) Mistral (e) Granite3.1-MoE (f) Gpt-4o

show that Gemini-1.5 Flash is the most lightweight and efficient, showing consistently low CPU usage and short elapsed times across urgency levels, making it ideal for quick responses in low-resource environments. Fig. 10f, and Fig. 10l show that Gpt-4o is even faster in execution time with minimal CPU consumption, demonstrating exceptional efficiency and scalability, particularly for edge computing. On the other hand, Claude 3.5 Sonnet and Mistral are among the most resource-intensive models as shown in Fig. 10c, Fig. 10i, Fig. 10d, and Fig. 10j. Claude 3.5 Sonnet and Mistral suffer from excessive delays with high CPU usage, highlighting inefficiencies in reasoning and computationally expensive and less scalable for real-time applications. Command-R7b and granite 3.1-MoE provide a moderate performance with lower elapsed times than Mistral or Claude 3.5 Sonnet, making it a middle-ground option as shown in Fig. 10b, Fig. 10h, Fig. 10e, and Fig. 10k. Overall, Gpt-4o and Gemini-1.5 Flash emerge as the most efficient in both speed and CPU consumption. Overall, GPT-4o and Gemini-1.5 flash emerge as the most effective models, achieving high accuracy with minimal computational overhead, making them ideal for both structured reasoning and real-time decision-making.

5.5 Low-urgency Agent Learning Performance Evaluation

Table 6 provides a comparative analysis of the low-urgency agent's performance using the Gpt-4o model, evaluated before and after learning across multiple metrics in a low-urgency setting on a cloud server. Key evaluated metrics include LLM call count, hierarchy depth, similarity score, LLM call usage cost, precision score, as well as elapsed time, CPU utilization, and memory utilization.

Before learning, Fig. 11 shows two Pareto-optimal solutions for the agent, indicated by the gray-shaded rows. These solutions represent the best trade-offs among similarity score, LLM call usage cost, and precision score. The evaluator agent powered by o1 model selected one of these solutions, indicating that the system was already capable of identifying high-quality, balanced outputs. When the selected solution was used as input for the next round (after learning), Fig. 12 shows that both the Pareto-optimal selection and the evaluator agent aligned on the same response with maximum elapsed time, and minimal CPU and memory utilization. This demonstrate improved alignment, consistency, and confidence in the agent's decision-making process post-learning.

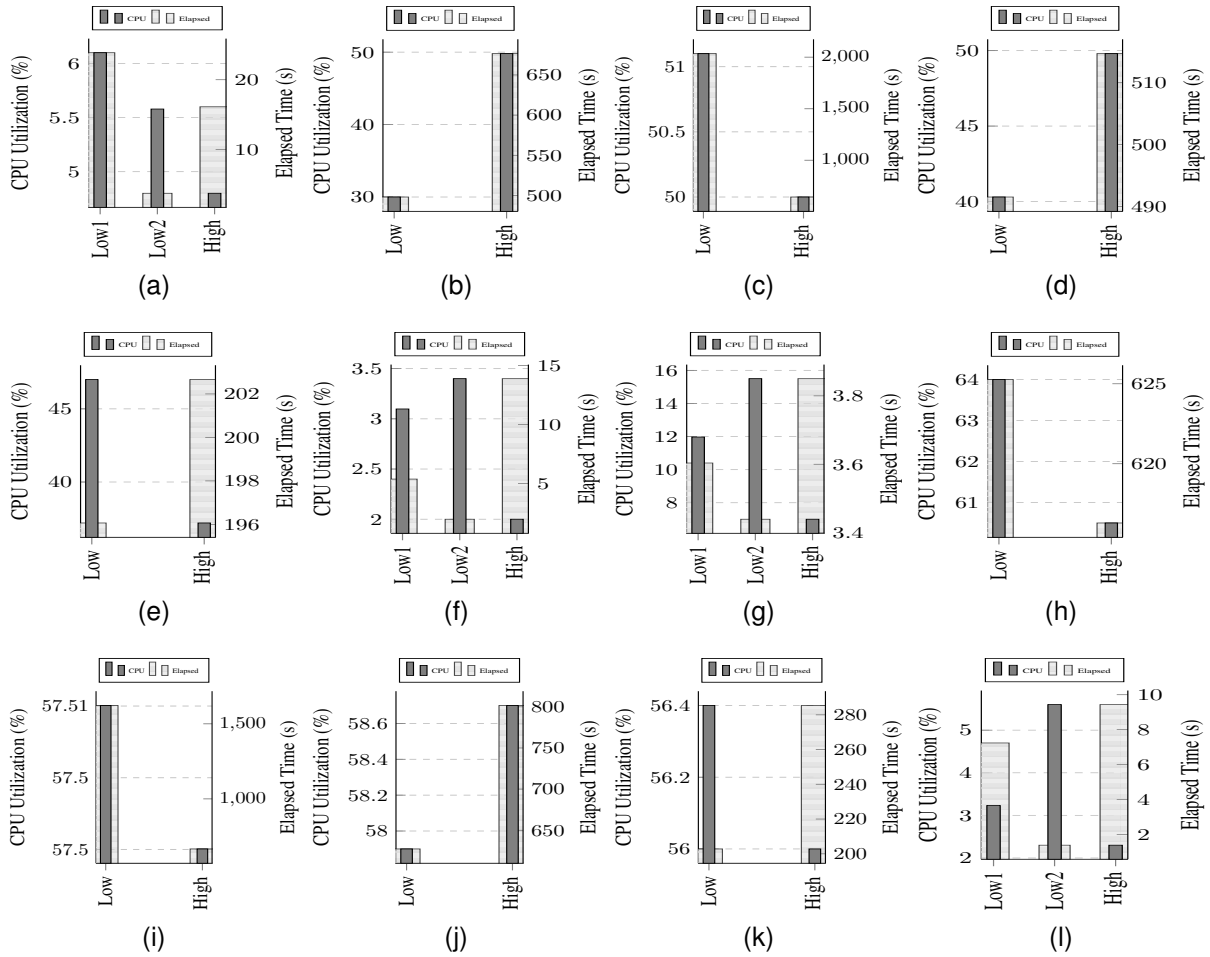


Figure 10: Evaluation of Low-Level Agent CPU Performance at Cloud: (a) Gemini-1.5 flash (b) Command-r7b (c) Claude 3.5 Sonnet (d) Mistral (e) Granite3.1-MoE (f) Gpt-4o; Evaluation of Low-Level Agent CPU Performance at the Edge: (g) Gemini-1.5 flash (h) Command-r7b (i) Claude 3.5 Sonnet (j) Mistral (k) Granite3.1-MoE (l) Gpt-4o;

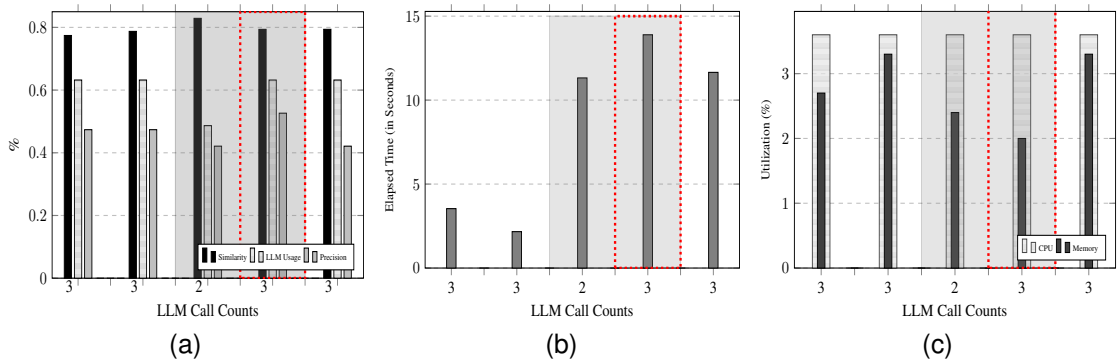


Figure 11: Low-urgency Learning Performance Evaluation before learning (a) Response of Metrics (b) Elapsed time before learning (c) CPU and Memory Utilization

Table 6: Low-urgency Learning Performance Evaluation.

Before Learning

Model Name	Urgency Level	LLM Call Count	Hierarchy Depth Count	Metrics: Similarity Score	Metrics: LLM Call Usage Cost	Metrics: Precision Score	Cloud Server: Elapsed Time	Cloud Server: CPU Utilization	Cloud Server: Memory Utilization
Gpt-4o	Low	3	3	0.7741	0.6321	0.4737	3.5381	2.70%	3.60%
		3	3	0.7872	0.6321	0.4737	2.1682	3.30%	3.60%
		2	2	0.8288	0.4866	0.4211	11.3214	2.40%	3.60%
		3	**3**	**0.7937**	**0.6321**	**0.5263**	**13.8917**	**2.00%**	**3.60%**
		3	3	0.7938	0.6321	0.4211	11.6523	3.30%	3.60%

After Learning

Model Name	Urgency Level	LLM Call Count	Hierarchy Depth Count	Metrics: Similarity Score	Metrics: LLM Call Usage Cost	Metrics: Precision Score	Cloud Server: Elapsed Time	Cloud Server: CPU Utilization	Cloud Server: Memory Utilization
Gpt-4o	Low	3	3	0.8027	0.6321	0.3673	16.5101	4.60%	3.40%
		3	**3**	**0.8421**	**0.6321**	**0.3636**	**23.8452**	**3.40%**	**3.40%**
		3	3	0.8048	0.6321	0.2909	30.2243	3.50%	3.60%
		3	2	0.8294	0.6321	0.3934	19.5182	3.50%	3.50%
		3	2	0.8109	0.6321	0.3157	21.6117	3.70%	3.40%

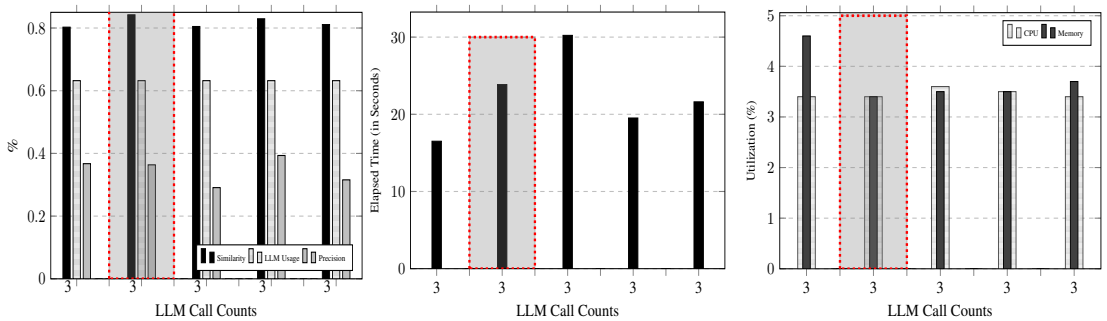


Figure 12: Low-urgency Learning Performance Evaluation after learning (a) Response of Metrics (b) Elapsed time (c) CPU and Memory utilization

Table 7: Environment Agent Performance Evaluation.

Model Name	Cloud Server: Elapsed Time	Cloud Server: CPU Utilization	Cloud Server: Memory Utilization	Edge Device: Elapsed Time	Edge Device: CPU Utilization	Edge Device: Memory Utilization	Accuracy
Gemini-1.5 flash	2.7076	5.60%	3.70%	3.5179	14.40%	35.70%	1
command-r7b	220.2311	35.00%	15.40%	192.1994	57.60%	64.90%	1
Claude 3.5 Sonnet	298.506	45.20%	15.10%	284.1143	57.30%	61.70%	0
Mistral	251.7816	44.20%	13.70%	242.4539	57.10%	57.80%	0.5
granite3.1-MoE	38.577	10.20%	9.00%	35.5997	52.90%	77.30%	0
Gpt-4o	8.3605	2.50%	3.80%	10.1599	10.80%	36.50%	1

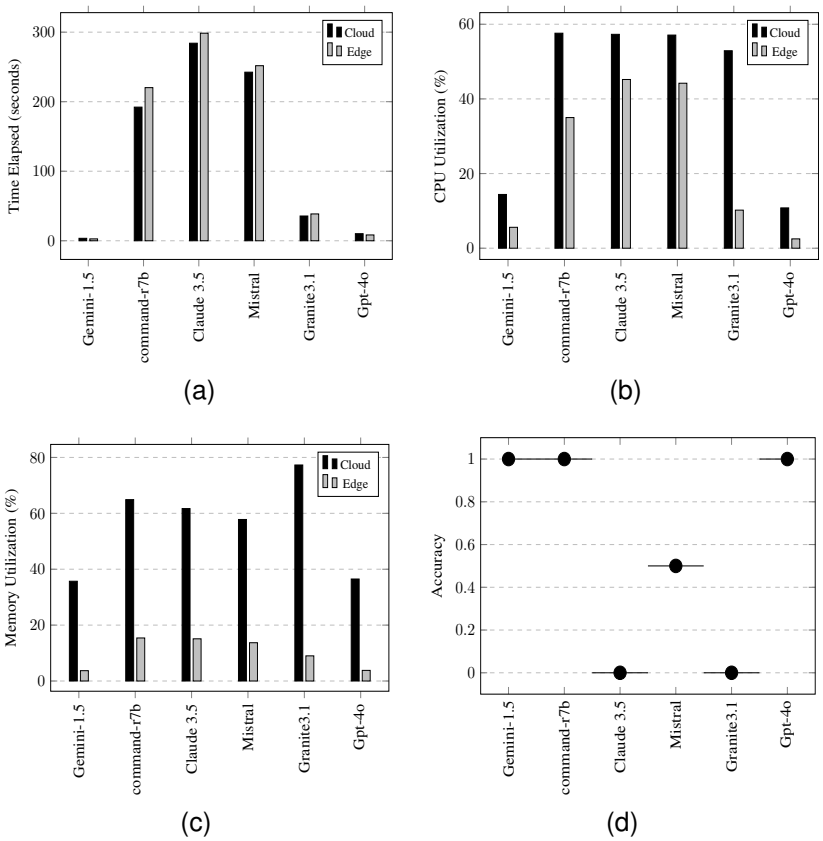


Figure 13: Environment Agent Performance Evaluation.

5.6 Environment Agent Performance Evaluation

The evaluation of the environment agent across different models reveals substantial differences in execution efficiency, resource utilization, and factual correctness when tracking real-time datasets and generating new commands. The results presented in Table 7 compare execution performance in terms of elapsed time, CPU utilization, memory consumption, and accuracy when benchmarked against the o1 model's commands.

Fig. 13 shows that GPT-4o and Gemini-1.5 flash emerge as the most efficient models, offering the best trade-off between execution speed, low resource consumption, and high accuracy, making them ideal for real-time environment monitoring and command generation. These models achieve the fastest elapsed times, minimal memory usage, and low CPU utilization, ensuring real-time responsiveness with minimal computational overhead. In contrast, Claude 3.5 Sonnet, Mistral, and command-r7b exhibit significantly longer elapsed times and high CPU consumption, leading to inefficiencies in processing real-time data and potential performance bottlenecks, particularly in resource-constrained environments. Moreover, models with low accuracy scores, such as granite3.1-MoE and Claude 3.5 Sonnet, demonstrate poor command reliability, posing risks in critical applications where incorrect actuator commands could degrade system performance and reduce overall QoE.

6 Conclusion and Future Work

This paper presented `UserCentrix`, an agentic memory-augmented AI framework designed to enhance the efficiency and adaptability of smart spaces. By integrating personal LLM agents with meta-reasoning capabilities, the framework dynamically adapts to user preferences and optimizes real-time decision-making. The hierarchical structure, combining centralized control with distributed autonomy, enables efficient resource allocation and multi-agent collaboration. Our experimental evaluation demonstrated significant improvements in computational efficiency, reasoning accuracy, and response latency, making the system highly effective for real-world deployment. Additionally, `UserCentrix`'s adaptive orchestration strategies provide a scalable and context-aware solution for dynamically evolving smart environments.

The findings underscore the importance of incorporating advanced memory augmentation and cooperative reasoning mechanisms to enhance AI-driven smart spaces.

One limitation of the current study is its reliance on a specific dataset, the University of Oulu Smart Campus Dataset, which was extended with a synthetic dataset to support the experiments. However, the framework's effectiveness in other types of smart environments has not been explicitly validated, primarily due to the lack of access to diverse real-world datasets. As a result, the ability of the system to handle a broader range of real-world tasks and operate effectively in larger, more heterogeneous deployment settings remains unexplored. Furthermore, the evaluation scope is relatively constrained, as it focuses on a limited set of large and small reasoning language models. This constraint is mainly due to computational limitations associated with using a desktop environment, which restricts the range and scale of models that could be tested.

To address these limitations, several promising directions for future work are proposed. First, the integration of the `UserCentrix` framework into other smart domains, such as healthcare, transportation, or industrial IoT could significantly broaden its applicability and reveal new optimization challenges and opportunities. Lastly, incorporating user-centric feedback loops into the agent design could enhance adaptability by enabling personal agents to refine their behavior through direct user input, ultimately improving trust, personalization, and long-term performance in dynamic environments.

Acknowledgments

This research is funded by the Research Council of Finland through the evoS3 (Grant Number 362594) and the 6G Flagship (Grant Number 369116) projects, and by Business Finland through the Neural Pub/Sub research project (Diary Number 8754/31/2022).

REFERENCES

[1] A. Varol, N. H. Motlagh, M. Leino, S. Tarkoma, and J. Virkki, "Creation of ai-driven smart spaces for enhanced indoor environments–a survey," *arXiv preprint arXiv:2412.14708*, 2024.

[2] Y. Chang, X. Wang, J. Wang, Y. Wu, L. Yang, K. Zhu, H. Chen, X. Yi, C. Wang, Y. Wang, W. Ye, Y. Zhang, Y. Chang, P. S. Yu, Q. Yang, and X. Xie, "A survey on evaluation of large language models," *ACM Trans. Intell. Syst. Technol.*, vol. 15, Mar. 2024.

[3] N. H. Motlagh, M. A. Zaidan, L. Lovén, P. L. Fung, T. Hänninen, R. Morabito, P. Nurmi, and S. Tarkoma, "Digital twins for smart spaces—beyond iot analytics," *IEEE internet of things journal*, vol. 11, no. 1, pp. 573–583, 2023.

[4] T. Meuser, L. Lovén, M. Bhuyan, S. G. Patil, S. Dustdar, A. Aral, S. Bayhan, C. Becker, E. de Lara, A. Y. Ding, *et al.*, "Revisiting Edge AI: Opportunities and Challenges," *IEEE Internet Computing*, vol. 28, no. 4, pp. 49–59, 2024.

[5] ETSI, "Experiential networked intelligence (eni); study on ai agents based next-generation network slicing." `https://www.etsi.org/deliver/etsi_gr/ENI/001_099/051/04.01.01_60/gr_ENI051v040101p.pdf`.

[6] L. Lovén, M. Bordallo López, R. Morabito, J. Sauvola, and S. Tarkoma, eds., *Large Language Models in the 6G-Enabled Computing Continuum: a White Paper [White paper]*. 6GFlagship, University of Oulu, Oulu, Finland: 6G Research Visions, No. 14, 2025.

[7] A. Lapkovskis, B. Sedlak, S. Magnússon, S. Dustdar, and P. K. Donta, "Benchmarking dynamic slo compliance in distributed computing continuum systems," *arXiv preprint arXiv:2503.03274*, 2025.

[8] A. Saleh, P. K. Donta, R. Morabito, N. Hossein Motlagh, S. Tarkoma, and L. Lovén, "Follow-Me AI: Energy-Efficient User Interaction with Smart Environments," *IEEE Pervasive Computing*, pp. 1–10, 02 2025.

[9] A. Saleh, S. Pirttikangas, and L. Lovén, "Pub/Sub Message Brokers for GenAI," *arXiv preprint arXiv:2312.14647*, 2023.

[10] T. Guo, X. Chen, Y. Wang, R. Chang, S. Pei, N. V. Chawla, O. Wiest, and X. Zhang, "Large language model based multi-agents: A survey of progress and challenges," *arXiv preprint arXiv:2402.01680*, 2024.

[11] Y. Cheng, C. Zhang, Z. Zhang, X. Meng, S. Hong, W. Li, Z. Wang, Z. Wang, F. Yin, J. Zhao, *et al.*, "Exploring large language model based intelligent agents: Definitions, methods, and prospects," *arXiv preprint arXiv:2401.03428*, 2024.

[12] J. Zheng, S. Qiu, C. Shi, and Q. Ma, "Towards lifelong learning of large language models: A survey," *ACM Comput. Surv.*, vol. 57, Mar. 2025.

[13] M. Xu, D. Cai, W. Yin, S. Wang, X. Jin, and X. Liu, "Resource-efficient algorithms and systems of foundation models: A survey," *ACM Comput. Surv.*, vol. 57, Jan. 2025.

[14] S. Lee, W. Sim, D. Shin, W. Seo, J. Park, S. Lee, S. Hwang, S. Kim, and S. Kim, "Reasoning abilities of large language models: In-depth analysis on the abstraction and reasoning corpus," *ACM Trans. Intell. Syst. Technol.*, Jan. 2025. Just Accepted.

[15] A. Mämmelä, J. Riekki, and M. Kiviranta, "Loose coupling: An invisible thread in the history of technology," *IEEE Access*, vol. 11, pp. 59456–59482, 2023.

[16] S. Han, Q. Zhang, Y. Yao, W. Jin, Z. Xu, and C. He, "Llm multi-agent systems: Challenges and open problems," *arXiv preprint arXiv:2402.03578*, 2024.

[17] Y. Shen, J. Shao, X. Zhang, Z. Lin, H. Pan, D. Li, J. Zhang, and K. B. Letaief, "Large language models empowered autonomous edge ai for connected intelligence," *IEEE Communications Magazine*, 2024.

[18] M. Zhang, J. Cao, X. Shen, and Z. Cui, "Edgeshard: Efficient llm inference via collaborative edge computing," *arXiv preprint arXiv:2405.14371*, 2024.

[19] Z. Hao, H. Jiang, S. Jiang, J. Ren, and T. Cao, "Hybrid slm and llm for edge-cloud collaborative inference," in *Proceedings of the Workshop on Edge and Mobile Foundation Models*, pp. 36–41, 2024.

[20] Z. Yu, Z. Wang, Y. Li, R. Gao, X. Zhou, S. R. Bommu, Y. Zhao, and Y. Lin, "Edge-llm: Enabling efficient large language model adaptation on edge devices via unified compression and adaptive layer voting," in *Proceedings of the 61st ACM/IEEE Design Automation Conference*, pp. 1–6, 2024.

[21] D. Ding, A. Mallick, C. Wang, R. Sim, S. Mukherjee, V. Ruhle, L. V. Lakshmanan, and A. H. Awadallah, "Hybrid llm: Cost-efficient and quality-aware query routing," *arXiv preprint arXiv:2404.14618*, 2024.

[22] M. Xu, D. Niyato, H. Zhang, J. Kang, Z. Xiong, S. Mao, and Z. Han, "Cached model-as-a-resource: Provisioning large language model agents for edge intelligence in space-air-ground integrated networks," *arXiv preprint arXiv:2403.05826*, 2024.

[23] C. Gao and S. Q. Zhang, "Dlora: Distributed parameter-efficient fine-tuning solution for large language model," *arXiv preprint arXiv:2404.05182*, 2024.

[24] X. Guo, K. Huang, J. Liu, W. Fan, N. Vélez, Q. Wu, H. Wang, T. L. Griffiths, and M. Wang, "Embodied llm agents learn to cooperate in organized teams," *arXiv preprint arXiv:2403.12482*, 2024.

[25] J. Wang, J. Wang, B. Athiwaratkun, C. Zhang, and J. Zou, "Mixture-of-agents enhances large language model capabilities," *arXiv preprint arXiv:2406.04692*, 2024.

[26] Q. Wang, T. Wang, Q. Li, J. Liang, and B. He, "Megaagent: A practical framework for autonomous cooperation in large-scale llm agent systems," *arXiv preprint arXiv:2408.09955*, 2024.

[27] Y. Hu, R. Lei, X. Huang, Z. Wei, and Y. Liu, "Scalable and accurate graph reasoning with llm-based multi-agents," *arXiv preprint arXiv:2410.05130*, 2024.

[28] S. Lu, J. Shao, B. Luo, and T. Lin, "Morphagent: Empowering agents through self-evolving profiles and decentralized collaboration," *arXiv preprint arXiv:2410.15048*, 2024.

[29] B. Brown, J. Juravsky, R. Ehrlich, R. Clark, Q. V. Le, C. Ré, and A. Mirhoseini, "Large language monkeys: Scaling inference compute with repeated sampling," *arXiv preprint arXiv:2407.21787*, 2024.

[30] C. Snell, J. Lee, K. Xu, and A. Kumar, "Scaling llm test-time compute optimally can be more effective than scaling model parameters," *arXiv preprint arXiv:2408.03314*, 2024.

[31] K. Christakopoulou, S. Mourad, and M. Matarić, "Agents thinking fast and slow: A talker-reasoner architecture," *arXiv preprint arXiv:2410.08328*, 2024.

[32] Z. Liang, Y. Liu, T. Niu, X. Zhang, Y. Zhou, and S. Yavuz, "Improving llm reasoning through scaling inference computation with collaborative verification," *arXiv preprint arXiv:2410.05318*, 2024.

[33] Z. Qi, M. Ma, J. Xu, L. L. Zhang, F. Yang, and M. Yang, "Mutual reasoning makes smaller llms stronger problem-solvers," *arXiv preprint arXiv:2408.06195*, 2024.

[34] P. Gao, A. Xie, S. Mao, W. Wu, Y. Xia, H. Mi, and F. Wei, "Meta reasoning for large language models," *arXiv preprint arXiv:2406.11698*, 2024.

[35] J. Wang, M. Fang, Z. Wan, M. Wen, J. Zhu, A. Liu, Z. Gong, Y. Song, L. Chen, L. M. Ni, *et al.*, "Openr: An open source framework for advanced reasoning with large language models," *arXiv preprint arXiv:2410.09671*, 2024.

[36] T. Wu, J. Lan, W. Yuan, J. Jiao, J. Weston, and S. Sukhbaatar, "Thinking llms: General instruction following with thought generation," *arXiv preprint arXiv:2410.10630*, 2024.

[37] E. Zelikman, G. Harik, Y. Shao, V. Jayasiri, N. Haber, and N. D. Goodman, "Quiet-star: Language models can teach themselves to think before speaking," *arXiv preprint arXiv:2403.09629*, 2024.

[38] S. J. Russell and P. Norvig, *Artificial intelligence: a Modern Approach, Fourth Edition*. Pearson, 2020.

[39] I. Watson and F. Marir, "Case-based reasoning: A review," *The knowledge engineering review*, vol. 9, no. 4, pp. 327–354, 1994.

[40] U. of Oulu, "Smart campus oulu indoor climate, air-quality and motion." `https://doi.org/10.23729/b9adb0a2-7381-45db-b32f-7e78ae1bc9e3`, 6 2021. University of Oulu, CWC - Verkot ja järjestelmät.

[41] N. H. Motlagh, P. Toivonen, M. A. Zaidan, E. Lagerspetz, E. Peltonen, E. Gilman, P. Nurmi, and S. Tarkoma, "Monitoring social distancing in smart spaces using infrastructure-based sensors," in *2021 IEEE 7th World Forum on Internet of Things (WF-IoT)*, pp. 124–129, IEEE, 2021.

APPENDIX SECTION

APPENDIX A

Personal Knowledge LLM Agent Prompt

You are an AI assistant specializing in understanding user tasks and evaluating the response. Your responsibilities include breaking down {user_task} into the following components, each representing a distinct aspect: Task_Time, Plans(Schedules or activities), Plan_Type, Plan_Mode, Preferences (Include personalized settings such as temperature, lighting, humidity, lunch type, or any other preference required by the user in their daily tasks). Instructions as follows:
1- Split the user's task into the following components, each representing a distinct aspect: Task_Time, Plans, Plan_Type, Plan_Mode, Preferences.
2- Find the plan type and decide if it is book room, or book meal or control sensors settings or any other.
3- Find the plan mode and decide if it is online or offline.
4- Extract the task time from {get_time}.
5- If the user didn't mention time (such as mentioning "now" or "after one hour"), then determine the time based on the context using the current time {get_time}.
6- If the Preferences value of the new plan is an empty list ([]), follow these steps to process the data:
- For the new plan, extract its Task_Time, and Plan_Type.
- For each entry in {personal_memory}:
1. Compare the stored Task_Time with the new Task_Time.
3. Generate embeddings:
a- {calculate_embeddings} of (new Plan_Type)
b- {calculate_embeddings} of (stored Plan_Type)
c- {calculate_similarity} between the embeddings
- If an entry has:
• time difference less than one hour, and
• similarity score more than 0.5,
then retrieve the preferences from the most recent matching entry in {personal_memory}.
- If no such entry exists (i.e., if the time gap is more than 1 hour, or the similarity is less than 0.5), return an empty list []. 7- Assess the response on a given task based on a criteria. Here is the data:
BEGIN DATA
Input: {user_task}
Submission: response
Criteria: Does the prediction exist within the query?
If the prediction of one column is empty and not mentioned in the query details, return true. Is the prediction referring to a real quote from the query?
END DATA Does the submission meet the Criteria? First, write out in a step by step manner your reasoning about each criterion to be sure that your conclusion is correct. Avoid simply stating the correct answers at the outset. Then print only the single character "Y" or "N" (without quotes or punctuation) on its own line corresponding to the correct answer of whether the submission meets the criteria. At the end, repeat just the letter again by itself on a new line. Parse the output text.

APPENDIX B

Classifier Agent Prompt

You are an intelligent assistant responsible for determining the urgency of tasks based on their time sensitivity. A user will provide a task description, and your job is to analyze the time constraints mentioned in the task and decide how urgent the task is. Please for each task, based on the user tasks provided in {personal_memory}:
- Extract current time and calculate the time remaining until the task's deadline based on task time {task_time}.
- Classify urgency level based on time sensitivity: <High or Low>
If the task is less than or equal to 2 hours, classify it as High Urgency. If the task is due in 2 hours to more than 1 day, classify it as Low Urgency.

APPENDIX C

High-urgency Agent Prompt.

You are a reasoning agent responsible for generating one time sensitive solution of each task should focus on completing the task quickly. For each task, create solution as following:
- Simplify each task by reducing or combining sub-tasks.
- Prioritize actions that achieve the most important outcomes in the shortest time for quick decision-making.
- If two or more LLM calls can be executed without waiting for each other , group them in the same rank for parallel execution. Ensure that all sub-tasks include complete details.

APPENDIX D

High-urgency Agent Prompt.

You are an intelligent reasoning agent responsible for generating all possible reasoning solutions for various tasks in a smart building, such as booking rooms, scheduling meals, adjusting environmental settings, or other. Your objective is to optimize resource usage efficiently while meeting user needs by applying causal reasoning and leveraging stored decision-making history when applicable. For each incoming task, follow below:
1- Recall <Task>in {solutions_memory}.
2- If no stored tasks is found, generate multiple diverse reasoning solutions based on criteria.
3- If stored tasks exist, {calculate_embeddings}for both the current task and stored tasks, then {calculate_similarity}. If similarity score more than 0.7, retrieve corresponding solutions and reasoning solutions from memory <Best_Solution><Reason><Comment>
Leverage these past solutions to generate optimized reasoning solutions that enhance efficiency while aligning with predefined criteria.
4 - If no high-similarity solutions exist, create multiple reasoning solutions from scratch, focusing on the following criteria.
5- Criteria for Solution Creation:
1. Exploring Available Resources:
- Analyze the current status of resources relevant to the request (e.g., rooms, meals, environmental settings).
2. Adapting or Reconfiguring Spaces and Services:
- Examine how existing resources (e.g., lighting, temperature, or meal customization) can be adapted to meet the user's preferences.
3. Searching for Availability:
- Identify available options (rooms, meals, or other requested facilities) during the required time window. - Ensure compatibility with any constraints, such as occupancy limits, reservation slots, or specific service times.
4. Matching Conditions to Preferences:
- Locate resources whose conditions (e.g., temperature, food preferences, or other) match or are closest to the user's specified preferences.
5. Suggesting Natural and Smart Adjustments:
- Recommend actionable enhancements, such as increasing natural light by opening curtains, adjusting heating or cooling, or suggesting meal modifications.
- Leverage smart building capabilities (e.g., automated climate control, dynamic lighting, or smart kitchen recommendations) to optimize the user experience.
Ensure each solution composed of sub-tasks tailored to address environmental adjustments that align with a user's preferences.
Each solution should outline a unique of sub-tasks, leveraging the above criteria as building blocks.
Organize these sub-tasks according to their dependencies:
- Sub-tasks without dependencies must execute in parallel.
- Sub-tasks with dependencies must execute in sequence.
- If two or more LLM calls can be executed without waiting for each other , group them in the same rank for parallel execution.

APPENDIX E

Environment Agent Prompt related to Smart Campus Dataset.

You are an intelligent assistant responsible for generating commands to control the temperature or lighting based on comparison between {preferred_value} and current values in dataset {Smart_Campus_dataset}. To generate the control commands, follow the instructions:
1- Consider the {Smart_Campus_dataset['Light']}. if it was LED lighting then the corresponding range between 500–800. if it was Dim lighting then the range between 900–1200. if it was Bright lighting or Natural Light, then the range between 1000–1500

2. Check if the room name {preferred_value} matches with any room in {Smart_Campus_dataset['Room_Name']}. If a match is found, compare the attributes of {preferred_value} with the corresponding values in {Smart_Campus_dataset}.

3. If the {preferred_value} of an attribute is higher than the current value in {Smart_Campus_dataset}, send the command: "increase [attribute name] in the [room name]".
- If the {preferred_value} of an attribute is lower than the current value in {Smart_Campus_dataset}, send the command: "decrease [attribute name] in the [room name]".

4. Only generate a response if a change is detected. If there is no change in value, do not output any command.