Development of Cooperative AI Agents

for Improving the Efficiency of Household Tasks

Kota Kawagoe

Kanagawa Institute of Technology

Abstract

The spread of smart homes and web services has improved daily convenience, but services remain fragmented and users still perform detailed operations across multiple applications. Although generative-AI-based natural-language control is promising, existing approaches often lack mechanisms for retaining long-term preferences and context, making it difficult to handle ambiguous instructions and deliver consistent automation across both web and physical environments. To address this problem, this study developed a collaborative multi-agent platform centered on an orchestrator with long-term and short-term memory, integrating four specialized agents for web operation, IoT control, knowledge retrieval, and schedule management. The system adopts Model Context Protocol (MCP) to standardize capabilities and improve extensibility. The orchestrator resolves ambiguous utterances using dialogue history and user attributes (e.g., address, health status), decomposes them into executable tasks, and allocates tasks to appropriate agents. For IoT control, we implemented hierarchical inference that combines high-capability cloud LLMs (via API) with lightweight edge LLMs (e.g., on Jetson), balancing privacy and responsiveness. In evaluations across ten scenarios, enabling memory improved task-achievement score by about 1.7x over baseline and reduced clarification questions. The system showed strong personalization, including health-aware recipe suggestions and location-aware information retrieval. We also confirmed that edge LLMs in the billion-parameter range can accurately convert abstract instructions into physical control commands under appropriate prompting. These results demonstrate the practical value of autonomous agent systems that infer intent from minimal instructions and provide integrated support across digital and physical domains.

1. Introduction

1.1 Background of this study

Smart-home devices and web services have become increasingly diverse, but user workflows remain fragmented across service boundaries. For example, using e-commerce and mobile-ordering services from different providers often requires repeated registration of email addresses, passwords, and shipping information, along with repeated UI operations. In IoT environments, control is still dominated by voice commands and simple rule-based triggers (e.g., IFTTT). As a result, support for ambiguous requests ("If it looks sunny tomorrow morning...") and composite conditions ("If someone is in the room and humidity is high, dehumidify; if no one is home, turn off the lights") remains limited. Additional issues include the burden and error risk of re-entering personal data, inconsistent operability across device types, and difficulty ensuring reliable cross-service automation.

At the same time, advances in generative AI have made tool-using agents increasingly practical [1]. These agents can interpret natural-language intent and coordinate multiple tools in sequence. If such capabilities are combined with personalized memory (user preferences, history, and constraints), a shared execution layer spanning both web and IoT domains, and stronger decision support, they can potentially overcome key limitations of traditional automation, especially ambiguity handling and cross-domain integration. For example, Home Assistant has already begun integrating LLM-based workflows [2]. From this perspective, this study investigates an integrated system that unifies web-based AI agents and physical devices under a consistent orchestration policy.

1.2 Purpose of this study

The objective of this study is to show that reliable task automation can be achieved from minimal natural-language instructions by integrating four specialized agents (including Browser and IoT agents) under an orchestrator equipped with personalized memory. Specifically, the study pursues the following goals:

(1) Integrated architecture design and prototyping: Define a unified architecture composed of a memory-enabled orchestrator, a RAG-based Life-Style agent, a Browser agent, an IoT agent, and a Scheduler agent, and clarify APIs, permissions, and data flow.

(2) Personalization and disambiguation: Store user attributes, history, preferences, and constraints (e.g., address, hobbies, occupation) in memory, and map underspecified instructions to optimal executable procedures.

(3) Evaluation using representative use cases: Execute tasks such as weather- and schedule-aware lighting control and automatic web-form filling using stored personal memory. Prepare multiple fictional personas and run multiple sample tasks. Evaluate performance using a score based on scenario subgoal achievement and the number of agent clarification questions (human intervention), with browser step count as a supplementary metric.

Through these efforts, we provide implementation guidelines and evaluation findings for cross-

domain integration of distributed web services and IoT to realize personalized automation.

1.3 Related Work

With the rapid development of LLMs, research on collaborative multi-agent task execution has expanded significantly. This section summarizes representative projects related to the theme of this paper, "task-efficiency improvement through collaborative AI agents," from the perspectives of objective, architecture, and similarities/differences from our proposal.

(1) **AutoGen :** AutoGen is a multi-agent conversational framework proposed by Microsoft Research [3]. It treats LLMs, humans, and tools as agents and supports diverse applications through configurable inter-agent dialogue patterns. While both AutoGen and this study center on collaboration through dialogue, AutoGen is a general-purpose framework and does not primarily target personal living spaces (cross-web/IoT execution) or persistent memory design for individual optimization.

(2) **Magentic-One :** Magentic-One is a general-purpose multi-agent problem-solving system [4]. A central orchestrator handles planning, progress tracking, and replanning while coordinating specialized agents for web operation, file operation, and code generation/execution. Its modularity supports extension, but adapting it to personal living spaces with lightweight deployment, privacy, and responsiveness requires additional design.

(3) **Magentic-UI :** Built on Magentic-One, Magentic-UI introduces a human-in-the-loop architecture [5]. It models humans as agents and supports intervention mechanisms such as co-planning, handoff, approval, result verification, and long-term memory. It also supports external-tool integration via MCP [5]. Our work similarly emphasizes human intervention and tool integration, but differs by focusing on personal daily-life tasks and jointly optimizing personal memory with a web+IoT execution layer. In addition, whereas Magentic-UI verification primarily supports human judgment via logs and links, our orchestrator closes the full loop of plan, execution, verification, completion judgment, replanning, re-execution, and memory update using automatic re-checking (e.g., web re-query and IoT state re-acquisition).

(4) **LLMind :** LLMind explores LLM-based orchestration of complex tasks including IoT [6]. It converts natural-language instructions into external module/device operations and supports multi-step execution through state management and experience accumulation. Our work extends this direction by integrating web search, web-UI operation, and scheduling in addition to IoT control, with explicit disambiguation based on memory of user attributes and preferences.

(5) **HuggingGPT :** HuggingGPT uses an LLM orchestrator to select and coordinate specialized models (e.g., from Hugging Face) for multimodal tasks [7]. Although it demonstrates effective model collaboration, its objective differs from our integration of heterogeneous tools such as web operation, IoT control, scheduling, and knowledge

retrieval.

(6) **Implementation platforms and tool-use learning :** AgentOS has been proposed as a platform that provides common capabilities for agent operations, including memory, permission management, and tool connectivity [8]. LangChain facilitates LLM application development through modular components such as chains, agents, and memory. Toolformer further showed that LLMs can self-learn API use for capability extension [1]. Building on these studies, our work focuses on implementing and evaluating personalization (long-term/short-term memory) and cross-domain execution (web + IoT) under one orchestrator in personal living environments.

(7) **Reasoning and action generation in smart homes :** Prior work has also studied goal-oriented reasoning and action generation for smart homes using LLMs [9]. In real living spaces, however, web information/service operations and IoT control are frequently coupled in a single task sequence, and ambiguous instructions must be resolved using personal context (preferences, schedules, environmental constraints), making integrated design and practical evaluation essential.

Summary : Prior work has advanced general multi-agent task execution (AutoGen, Magentic-One, Magentic-UI), automated complex task execution including IoT (LLMind), model-collaborative decomposition and execution (HuggingGPT), and infrastructure/tool-learning foundations (AgentOS, LangChain, Toolformer). However, end-to-end implementations that jointly target personal living spaces, integrate web information/UI operations with physical IoT control on a unified execution layer, incorporate memory-based disambiguation/personalization, and explicitly divide roles between edge and cloud inference remain limited. This paper contributes three core points: (1) abstraction of web and IoT capabilities as equivalent MCP tools under a single orchestrator, (2) a hierarchical inference architecture that combines API-based large models with edge-side lightweight models for privacy and responsiveness, and (3) memory-driven completion of ambiguous instructions into executable plans, validated through integrated evaluation.

1.4 Structure of this paper

This paper is organized as follows. Chapter 2 describes the overall system. Chapter 3 details the architecture and behavior of each agent. Chapter 4 presents the evaluation results. Chapter 5 discusses implications and limitations. Chapter 6 concludes the paper.

2. Overview of the Entire System

The proposed system is a collaborative multi-agent platform that interprets ambiguous user instructions and executes tasks across both web and physical (IoT) domains. At the center is an orchestrator that stores user context and long-term preferences. Around it, four specialized agents are arranged in a hub-and-spoke architecture: Browser, IoT, Life-Style, and Scheduler.

2.1 Overall system configuration

The system is implemented as a set of independent microservices running in Docker containers. Each agent exposes an HTTP-based REST API, which the orchestrator uses to dispatch tasks and collect results. When a user sends an instruction via chat, the orchestrator parses the request and decomposes it into executable subtasks. For example, for a request such as "adjust things based on tomorrow's weather," the orchestrator first assigns a weather-search task to the Browser agent and then issues an IoT control task based on the retrieved forecast. This enables multi-step tasks that require both external information acquisition and physical control beyond what a single LLM can complete directly.

2.1.1 Adoption of Model Context Protocol (MCP)

To standardize agent capabilities and improve interoperability, we adopt Anthropic's Model Context Protocol (MCP) [10]. MCP is an open protocol that connects AI models with external data and tools. In addition to REST APIs, each agent also functions as an MCP server. As a result, MCP-compatible external clients can directly invoke platform functions such as browser operation, schedule management, knowledge retrieval, and IoT control through a unified interface, without additional integration code. This design supports orchestrator-agnostic reuse of agents and improves extensibility of the overall agent ecosystem.

2.2 Orchestrator and Memory Functions

The orchestrator is implemented as a LangGraph-based state machine and runs an autonomous Plan, Execute, Review loop [11].

(1) Plan : Generate a JSON execution plan indicating which agent should perform which command, based on user input, dialogue history, and memory.

(2) Execute : Execute: Call each specialized agent API according to the plan.

(3) Review : Evaluate outputs and determine whether the original intent was satisfied; if not, request re-execution. The key personalization mechanism is dual memory.

(1) Long-term Memory : Stored in a JSON "long-term memory database" containing persistent attributes such as address, family structure, hobbies, and chronic conditions.

(2) Short-term Memory : Stored in a JSON "short-term memory database" containing recent interests and temporary context.

This enables intent inference from shorthand instructions such as "as usual," without repeatedly asking the user for full background information.

2.3 Collaborating Specialized Agents

The orchestrator coordinates four specialized agents. Each provides both REST APIs and MCP tool definitions.

(1) **Life-Style agent :** A knowledge-base agent that uses RAG (Retrieval-Augmented Generation) to retrieve and answer household/lifestyle information. FAISS-based vector retrieval [12] is used to search large document collections.

(2) **Browseragent :** An agent built on browser-use [13] that autonomously controls a real Chrome browser for web search, navigation, and form interactions.

(3) **IoT agent :** An agent that manages heterogeneous IoT devices (appliances, sensors, actuators), converts natural-language instructions into device-specific commands, and executes them asynchronously via a job queue.

(4) **Scheduler agent :** An agent for routines, tasks, and notes, using LLM function calling to execute schedule checks, additions, updates, and completion operations reliably against a database.

Each specialized agent can also switch LLM models dynamically based on task complexity and evaluation purpose. For example, lightweight models (e.g., Gemini 2.5 Flash Lite) can be used for simple schedule registration, while higher-capability models (e.g., GPT-5.1) are assigned to tasks requiring deeper reasoning. This supports cost-performance optimization and enables analysis of model effects on overall system behavior.

3. Details and Operation of Each Agent

3.1 Orchestrator Agent

The orchestrator agent is the core component of the system. It decomposes ambiguous user instructions into concrete task plans and supervises the execution of specialized agents. It is implemented with LangGraph [14], which provides state-machine and graph-based control of multi-step reasoning. Through repeated Plan, Execute, and Review cycles, the orchestrator supports robust task completion with self-correction. Figure 3.1 shows an example screen.

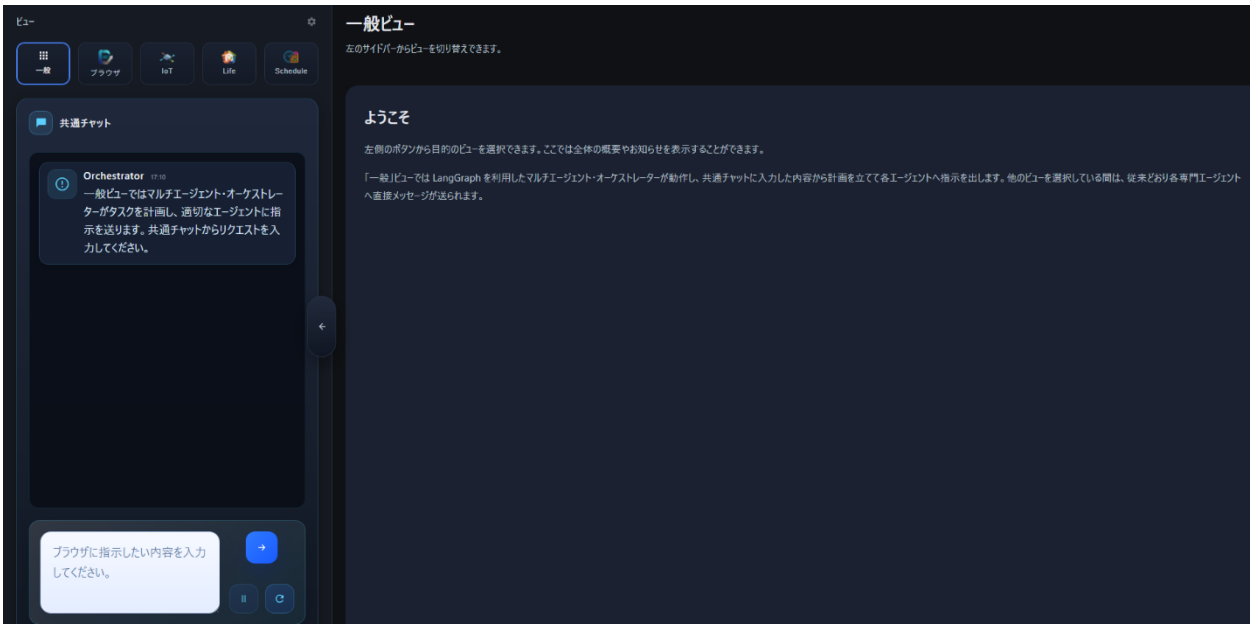


Figure3.1 Orchestrator screen example

3.1.1 Architecture and autonomous control loop

The orchestrator behavior is defined as a state-transition loop

Plan, Execute, Review (Figure 3.2).

(1) Plan : An LLM (Gemini, Claude, GPT, or Groq) generates a task plan from user input, dialogue history, and long-/short-term memory. The plan is represented in JSON and specifies which agent capability ("knowledge base," "browser operation," "IoT control," or "schedule management") should execute which command.

(2) Execute : The orchestrator calls the HTTP APIs of specialized agents in sequence according to the task list.

(3) Review : The LLM evaluates execution outcomes and determines whether the original user objective has been satisfied. If not, it can request retries.

This loop avoids premature termination after a single failure and improves robustness through iterative correction. Figures 3.2 and 3.3 illustrate the architecture and flow.

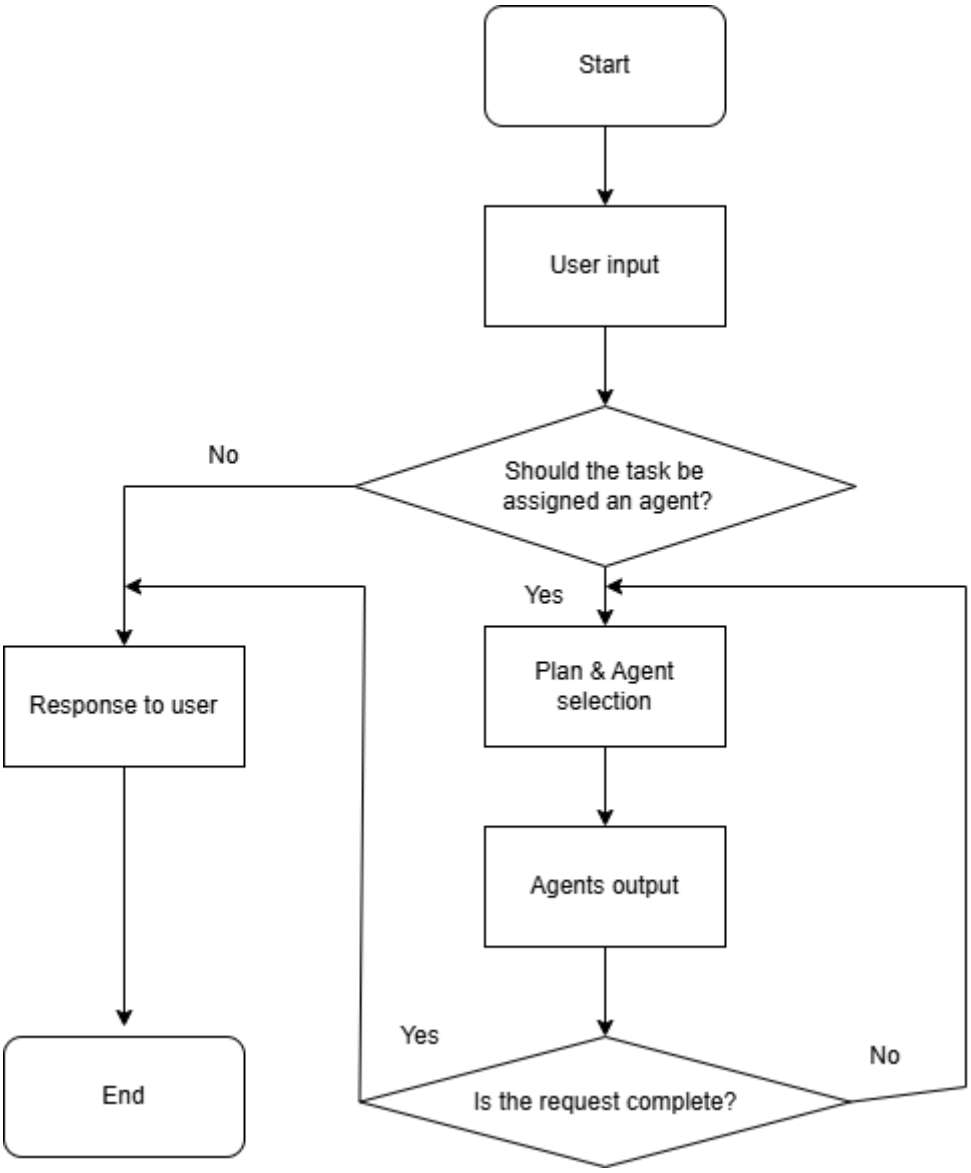


Figure 3.2 Orchestrator flow

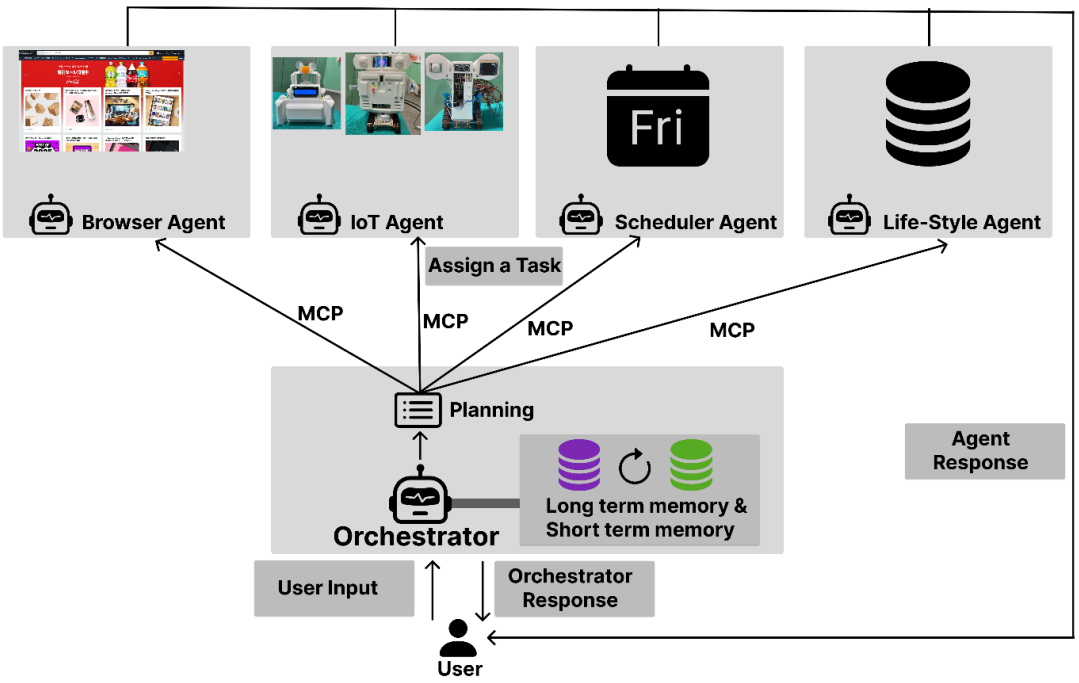


Figure 3.3 Entire system centered on the orchestrator

3.1.2 Memory Update Algorithm

In this system, memory is centrally managed by a "MemoryManager", and LLM inferences are applied as structured diffs to existing memory (informed by general long-term-memory/tool design patterns) [15]. Updates are not simple overwrites; they prioritize consistency and persistence through the procedure below. The mechanism is designed to follow a forgetting-curve pattern (Figure 3.4).

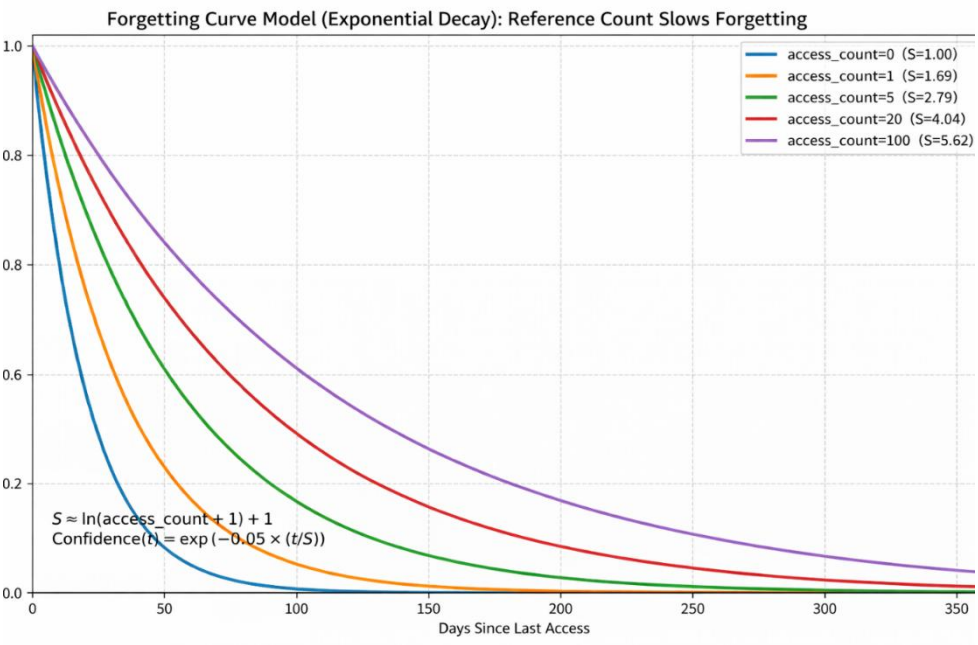


Figure 3.4 Example of the memory update algorithm

(1) Update Diff

When conversation length reaches a threshold, the conversation log and memory snapshot are passed to the LLM, which produces update content as a JSON diff.

(a) Short-term memory : Treated as a working buffer where overwrite and discard are allowed.

(b) Long-term memory : Uses an append-first policy. List updates are forced to diff operations such as {"add": [...], "remove": [...]} rather than full replacement.

(2) Diff Application

(a) Update data : Dictionaries are merged recursively. Lists are updated by add/remove with deduplication to prevent accidental loss of existing information.

(b) Operation sequence :

(i) Slot update : Variant expressions are merged via ID normalization and fuzzy matching, then update history, confidence, and priority are refreshed.

(ii) Usage record : Reference counts and scores are updated to increase the weight of frequently used information.

(iii) Episode/project management : Episodic memory and project state are maintained.

(3) Promotion

At short-term-memory TTL expiration, slots/episodes whose reference count or importance exceeds thresholds are promoted to long-term memory.

(4) Decay (Forgetting Curve)

Confidence for unreferenced information decays exponentially. This reinforcement/decay design was inspired by MemoryBank, which proposed long-dialogue memory updates [16].

(a) Stability S （harder to forget with more references）:

$$S \approx \ln(\text{access\_count} + 1) + 1$$

(b) Confidence(t) （Decays according to the number of days since the last access, t）:

$$\text{Confidence}(t) = \exp\left(-0.\ \ 05 \times \frac{t}{S}\right)$$

The coefficient `0.05` was chosen empirically. Under this setting, `Confidence(t)` decreases to about `0.5` when `t/S ≈ 14` days, balancing adaptation to short-term context shifts against over-erasure of long-term attributes. Coefficient optimization is left for future work.

Long-unreferenced information is downgraded to low priority, but traces are retained rather than hard-deleted.

This design separates working context (short-term) from persistent user profile information (long-term) while enabling autonomous update and consolidation over ongoing dialogue.

3.1.3 Planning phase: Role of the planner and safety checks

In the planning phase, an LLM uses plan-generation prompts to analyze user requests and determine action strategy. If no tool operation is needed (e.g., greeting or simple knowledge

confirmation), the system selects direct-response mode and replies immediately without generating tasks, improving latency and reducing resource usage.

If specialized agents are required, the request is decomposed into executable units under a predefined maximum task count. The system does not freeze a full plan upfront; instead, it uses incremental replanning. After each task, the result (success/failure/obtained information) is fed back into context, and planning is rerun. This enables dynamic adjustment of subsequent tasks (e.g., IoT lighting control) based on prior outcomes (e.g., weather retrieved by browser search).

After plan generation, a safety mechanism called executable-feasibility checking is applied. A separate LLM call verifies whether each task command is concretely executable. Additional clarification questions are asked only when essential information is missing for irreversible actions (e.g., payment or delivery), reducing uncertainty while limiting user burden.

3.1.4 Execution phase: Calling specialized agents

In the execution phase, specialized-agent APIs are called according to the agent identifier in each planned task:

(1) **"Knowledge base function" (**Life-Style **agent):** Sends an HTTP POST request to the answer-generation endpoint via the Life-Style-agent call function. Figure 3.5 shows this operation.

(2) **"Browser operation function" (Browser agent):** Calls the Browser agent's chat API via the Browser-agent call function (default service address). Streaming mode is supported for real-time progress updates. Figure 3.6 shows this operation.

(3) **"IoT control function"** (IoT agent): Sends HTTP requests through the IoT-command dispatch function to issue device-control commands. Figure 3.7 shows this operation.

(4) **"Schedule management function"** (Scheduler agent): Sends chat messages via the Scheduler-agent call function to perform schedule checks and task add/update operations. Figure 3.8 shows this operation.

3.1.5 Review phase: self-correction mechanism

After each task completes, the review phase starts. A dedicated review prompt asks the LLM to evaluate execution results. If the result is classified as retry status, the orchestrator re-executes the same task up to the retry limit. This self-correction mechanism enables recovery from transient errors and insufficient outcomes, improving overall task success.

Figure 3.5 Example of screen where orchestrator is operating Life-Style agent

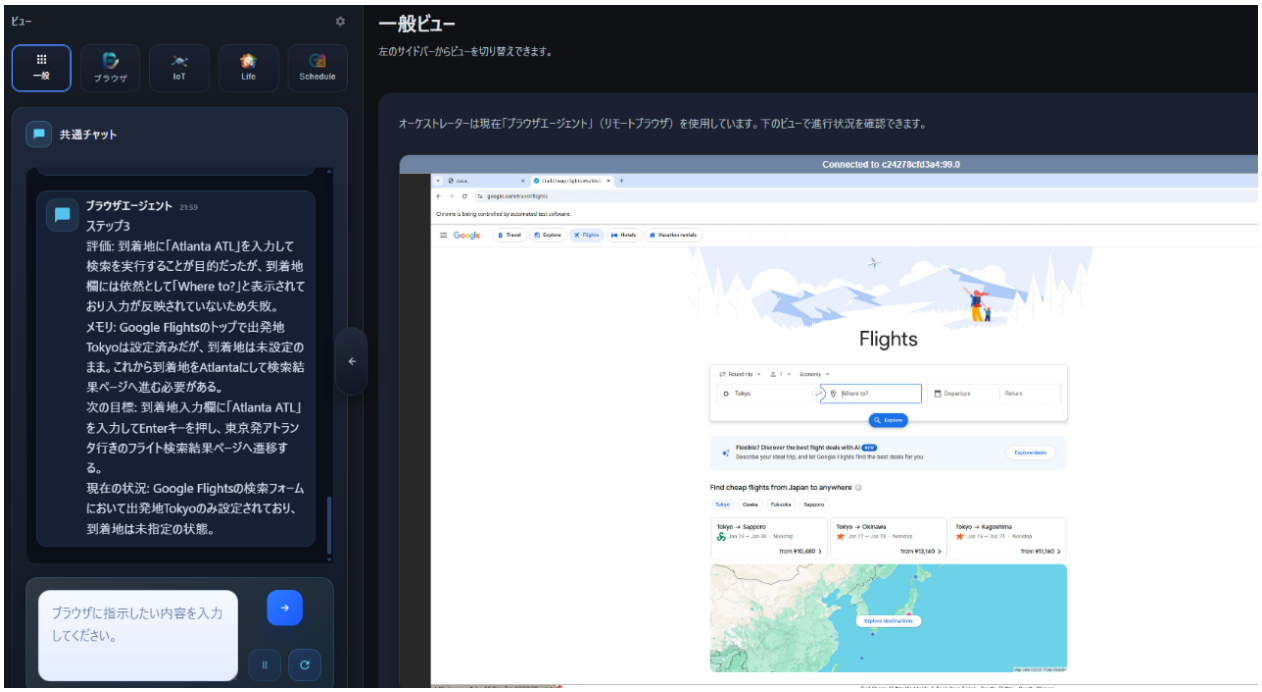


Figure 3.6 Screen example where the orchestrator is operating the Browser agent

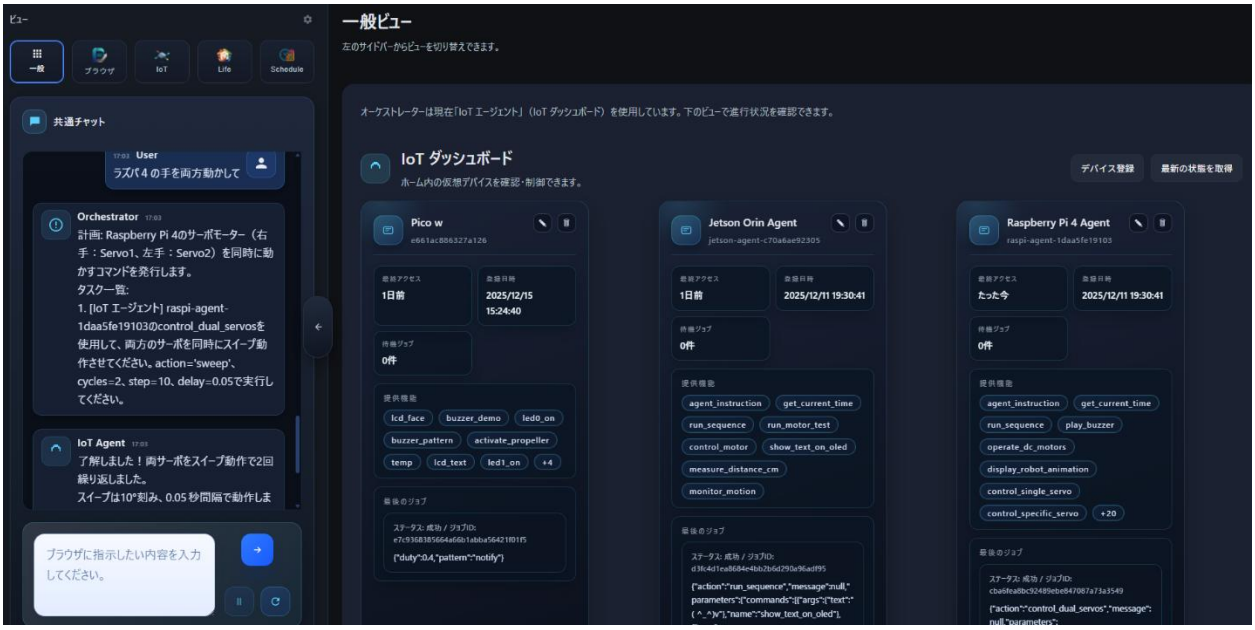


Figure 3.7 Example of screen where orchestrator is operating IoT agent

Figure 3.8 Example of a screen where the orchestrator is operating the Scheduler agent

3.2 Browser Agent

The Browser agent is a specialized component that controls a Chrome browser in a human-like manner and autonomously performs web tasks such as information search, navigation, and service interaction. It is implemented as a Flask-based API server and operates on requests from other components (e.g., the orchestrator). Figure 3.9 shows an example screen.

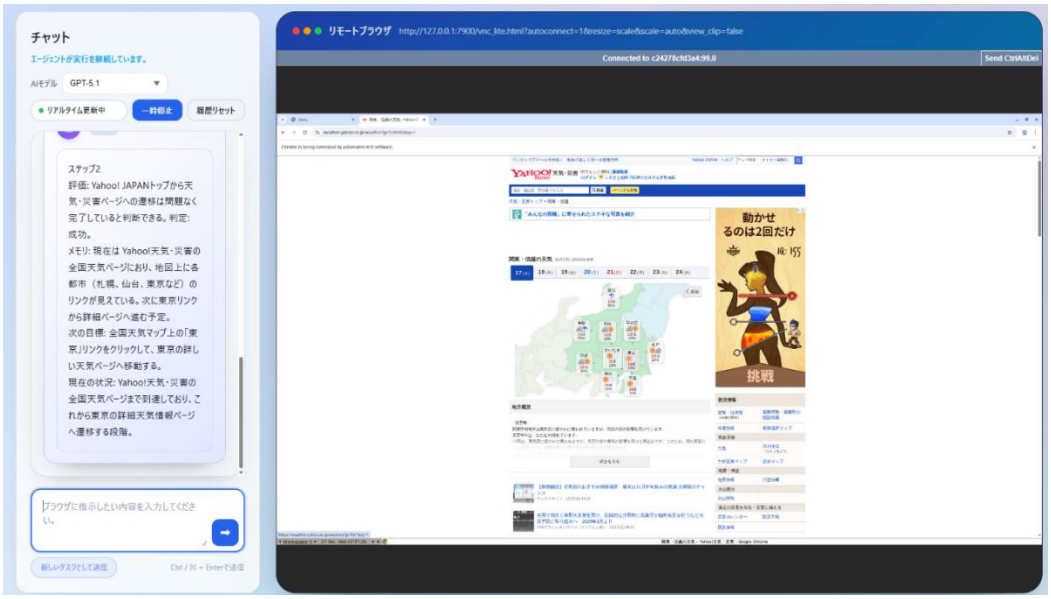


Figure 3.9 Browser agent screen example

3.2.1 Overview and technology stack

The core technology is browser-use [13], driven by LLMs (Gemini, Claude, GPT, and Groq). This enables operation of dynamic websites and complex UIs that are difficult for conventional scraping approaches based only on DOM parsing and element matching. Communication with the browser uses Chrome DevTools Protocol (CDP), which provides stable remote control even for headless browsers in isolated environments such as Docker containers.

The configuration of the Browser agent is shown in Figure 3.10.

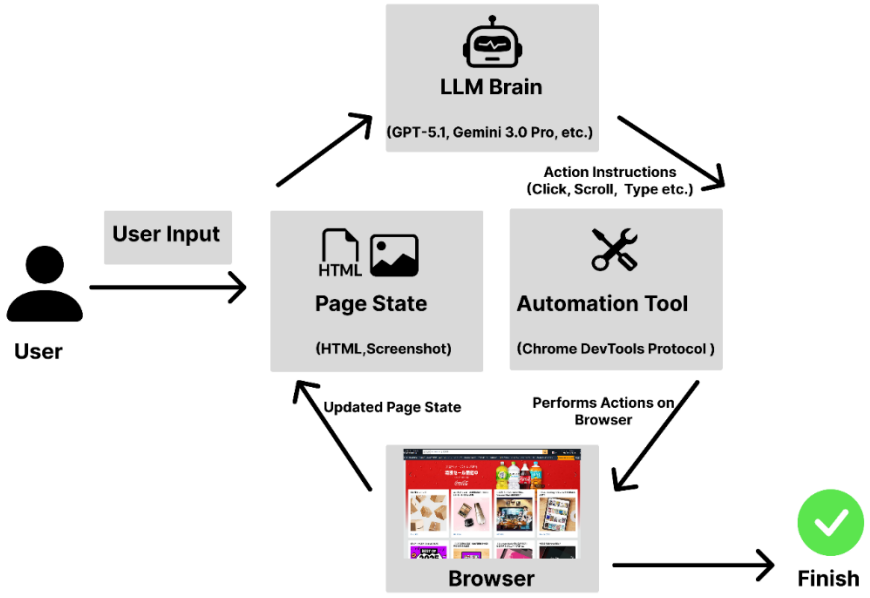


Figure 3.10 Browser agent configuration

3.2.2 Architecture and main features

The internal architecture is centered on a controller that manages the browser-operation lifecycle.

(1) **Agent control and session management:** The controller centralizes browser startup, task execution, and teardown. The latest implementation strengthens session persistence (Keep-Alive) and automatic recovery so the browser can retain state after task completion and remain ready for follow-up instructions. EventBus and Watchdog processes monitor browser crashes, unexpected pop-ups, download completion, and related runtime events to improve long-running stability.

(2) **Real-time progress streaming:** Execution status is streamed via Server-Sent Events (SSE), allowing users to inspect step-level progress such as current page and input actions.

(3) **Model flexibility:** LLM models can be switched dynamically (e.g., Gemini 2.5 Flash Lite, GPT-5.1) according to task complexity and cost targets.

(4) **Text-response mode:** If browser operation is unnecessary (e.g., questions answerable from general knowledge), the system replies in text without launching a browser, minimizing latency and resource usage.

3.2.3 Integration with External Systems (API)

This agent provides a RESTful API for flexible integration with external systems.

Main endpoints are:

(1) **Main Chat API:** Primary interface for natural-language instructions from users and orchestrators. If a task is already running, follow-up instructions are queued and processed incrementally.

(2) **Agent Relay API ("Agent Cooperation API"):** Dedicated endpoint that allows other agents to invoke browser operations as isolated subtasks. These interactions are excluded from the main conversation history to avoid history contamination.

3.3 IoT Agent

The IoT agent is an integrated gateway that connects physical IoT devices (e.g., Raspberry Pi Pico W, Raspberry Pi 4, Jetson Orin Nano) to the AI-agent system. It is implemented as a Flask-based web server and converts natural-language instructions into device-specific control commands for asynchronous execution. Figure 3.11 shows an example screen.

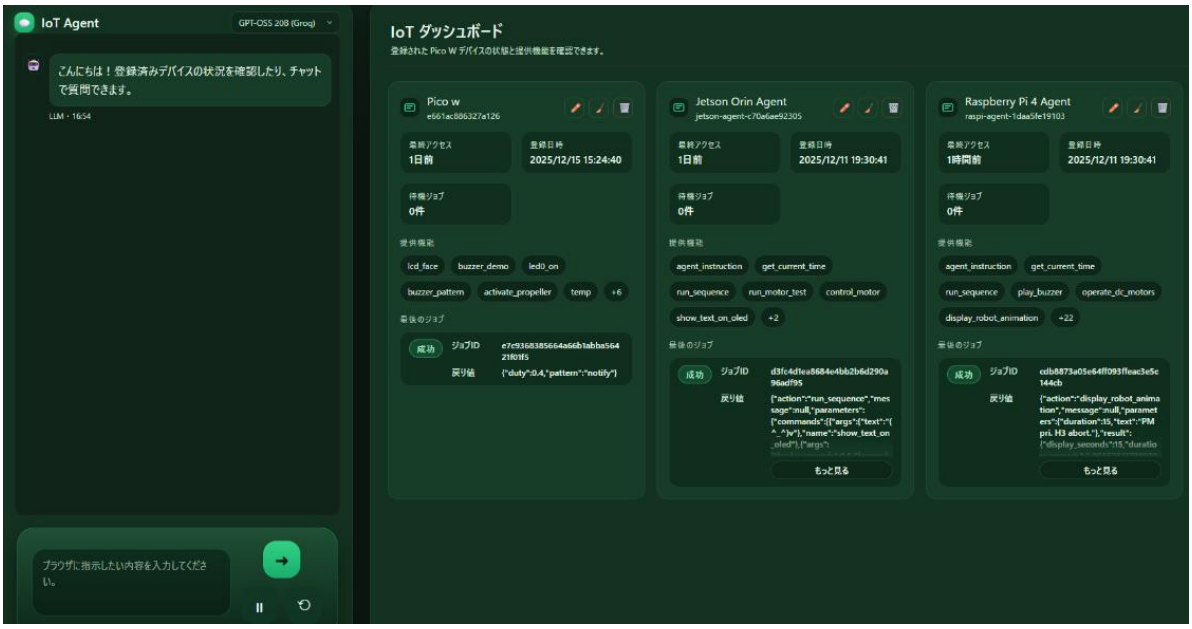


Figure 3.11 IoT agent screen example

3.3.1 Overview and role

The IoT agent abstracts heterogeneous devices with different protocols and control interfaces behind a unified API. It receives HTTP requests from the orchestrator or direct input from the web UI and routes commands to appropriate registered devices. It also runs as an MCP server and exposes standardized tools such as device control and device-list retrieval. This allows LLMs to select and execute these tools autonomously in natural-language context. Figure 3.12 shows the architecture.

3.3.2 System architecture

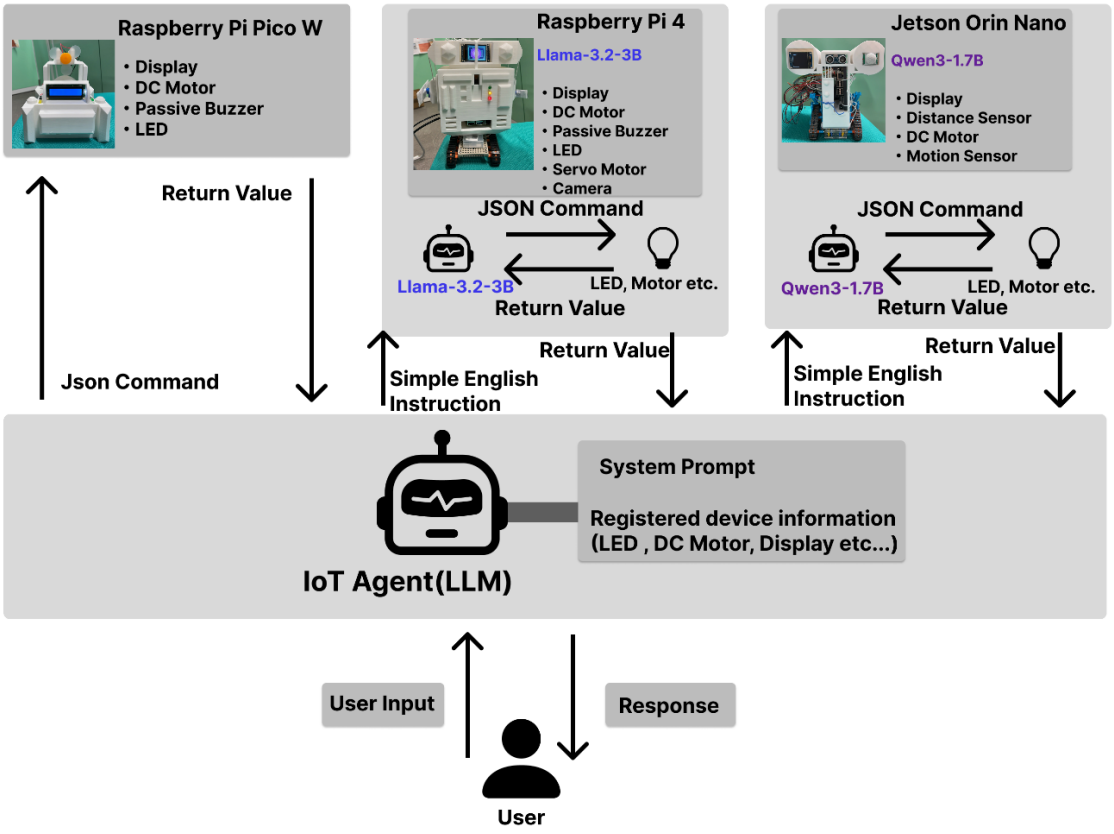
The IoT agent consists of the following components.

(1) **Flask server ("main server program"):** Core process that provides REST endpoints and handles authentication, LLM communication, and job management.

(2) **MCP server module:** Built with the MCP SDK to process LLM tool calls and translate them into concrete device commands.

(3) **In-memory data store:** Maintains device connection state, registered capabilities, pending job queues, and execution logs in memory. The lightweight non-persistent design supports low-latency response.

(4) **Edge-device client:** Python scripts running on Raspberry Pi, Jetson, and related devices. Each client polls the server, executes jobs addressed to that device, and returns results (success/failure, sensor readings, image data, etc.).



3.12 IoT agent configuration

3.3.3 Command conversion and execution flow by LLM

Instructions from users or the orchestrator are processed through the following MCP-based

(1) Message analysis: Incoming natural-language messages are sent to the LLM with current dialogue history, together with available MCP tool definitions.

(2) Function calling: The LLM interprets intent (e.g., "turn on the LED"), selects an appropriate tool and arguments (e.g., `control_device(device_id="raspi-agent-123", command="led1_on", args={"duration": 5})`), and emits a function call.

(3) Execution and validation: The MCP server module receives the call and validates arguments (required fields, device capability support, etc.).

(4) Queuing: Valid requests are enqueued as jobs and executed when the target device polls.

Compared with regex-based parsing or naive JSON generation, this mechanism enables more accurate and flexible device control.

To reduce user burden, prompt-level autonomous decision logic is also introduced:.

(1) Automatic default completion: If detailed parameters are omitted (e.g., motor duration/angle, buzzer tone), the system applies predefined defaults (e.g., 5.0 s movement, 90-degree servo) rather than asking follow-up questions.

(2) Context-based target inference: If only one device is registered, or context clearly identifies the target, the command is issued without additional confirmation ("read the room"

behavior).

(3) Device-difference absorption: Command-system differences across similar device capabilities (e.g., text display functions on Jetson vs. Raspberry Pi) are absorbed via prompt instructions so that the LLM selects appropriate internal commands by device type.

3.3.4 Device management and job queue

The system supports dynamic device registration and management. At startup, each device reports its ID and capabilities (e.g., power control, color change, image capture) to the server. Commands are executed through an asynchronous job queue, enabling stable server-side dispatch even for devices on unstable networks or with slow operations (e.g., camera upload workflows). Each job receives a unique ID, and execution status/results can be retrieved later by polling or wait-based synchronization.

3.3.5 Visual Extension Functionality

The IoT agent also supports camera-based visual extensions (e.g., Raspberry Pi + camera module). When chat messages contain keywords such as "Tell me about the surroundings" or "Show me a photo," the system automatically selects a vision-capable model (e.g., GPT-5.1 or Gemini 3 Pro) and sends a camera-capture command to the device. Captured images are uploaded from edge devices to the server and passed to the LLM. This enables context-aware judgments such as "Is the room messy?" or "Are the lights on?" based on visual evidence rather than sensor values alone. Figures 3.13 and 3.14 illustrate this workflow.

3.3.6 Implemented edge devices

To validate the system, three types of edge devices were physically built and integrated. The latest implementation supports parallel action execution across devices, enabling combinations such as motion with simultaneous display output or buzzer playback.

Figure 3.13 IoT agent explaining the image sent from the device side

(1) Jetson Orin Nano 8GB

(a) **Role**

Using its high edge-AI compute capability, this device performs on-device command interpretation and control with a local LLM (`Qwen3-1.7B-Q4_K_S.gguf`) [17].

(b) Functions (Input/Output and Devices)

(i) Movement control: Omnidirectional motion via motor driver (L293D)

(ii) Display: Text output on display (SH1107)

(iii) Measurement: Ultrasonic ranging (HC-SR04)

(iv) Detection: Motion detection via PIR sensor (SR501)

(c) **Execution form (control/orchestration)**

Supports both parallel and sequential execution by combining the above functions

(d) Features (architecture)

Implements a distributed-intelligence model in which natural-language instructions from the server are interpreted and executed directly on the device.

(e) **Figures/Tables (References)**

Figures 3.15 and 3.16 show the physical device and configuration. Table 3.1 compares token-generation speed for the model on Jetson Orin Nano CPU vs GPU.

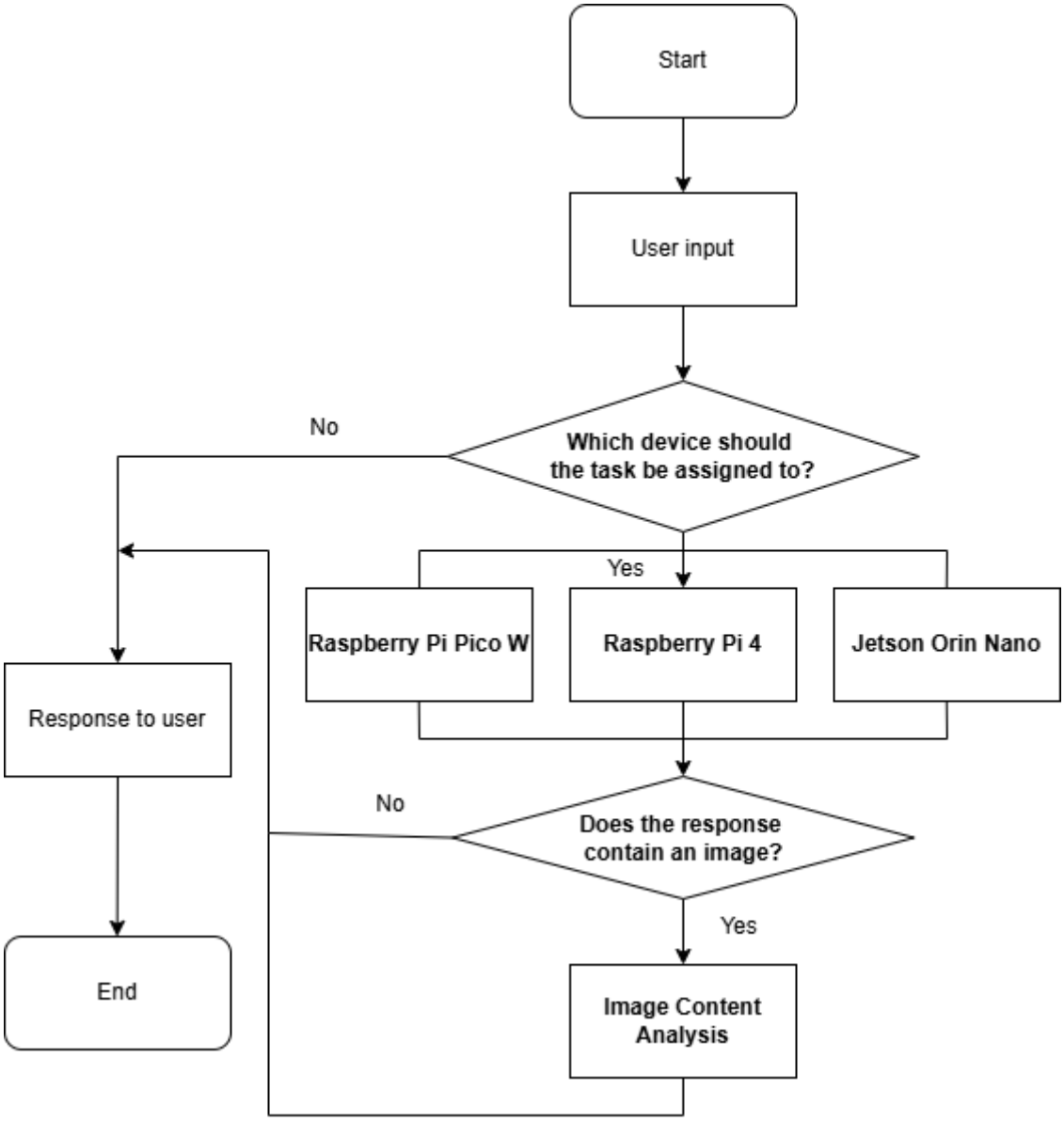


Figure 3.14 Flow of image analysis by IoT agent

Table 3.1 CPU inference and Qwen3 1.7B model in Jetson Orin Nano

GPU inference token generation speed comparison

Number of trials	CPU Speed (tokens/sec)	GPU Speed(tokens/sec)	Speedup (x)
1	8.53	21.95	2.57
2	9.75	24.85	2.55
3	9.78	25.43	2.60
4	9.75	25.13	2.58
5	9.81	24.69	2.52
6	9.87	25.07	2.54
7	9.70	24.97	2.54
8	9.75	25.17	2.58
9	9.73	25.37	2.61
10	9.79	24.92	2.55
Average	**9.64**	**24.76**	**2.57**

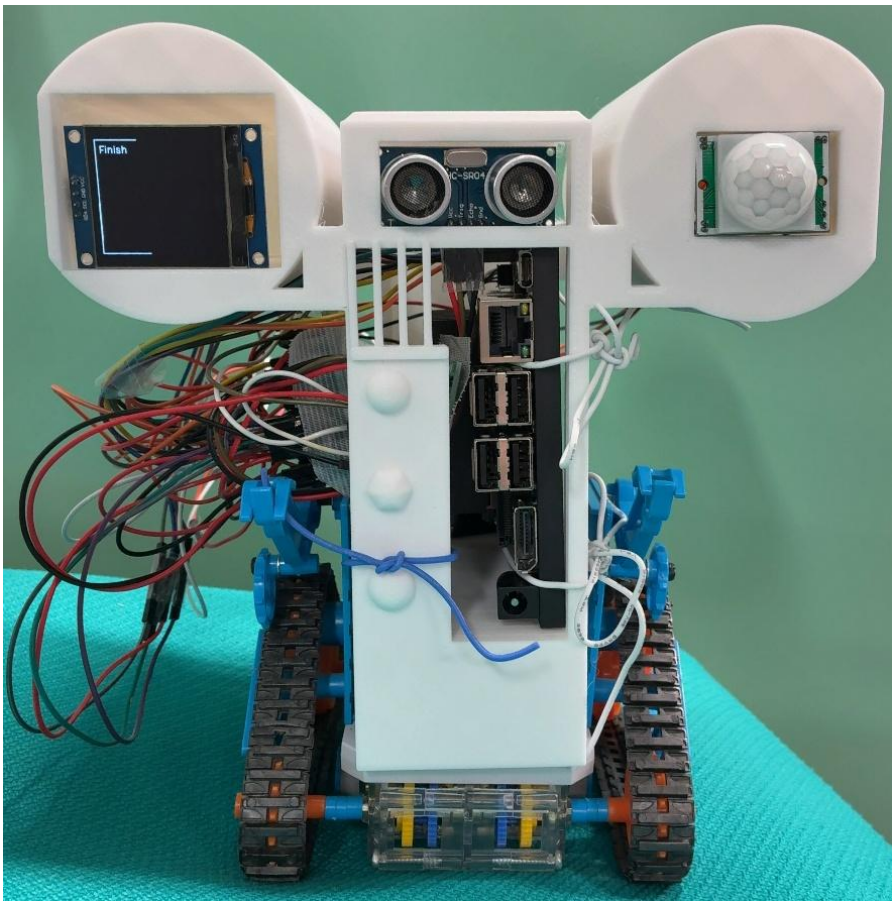


Figure 3.15 IoT device equipped with Jetson Orin Nano

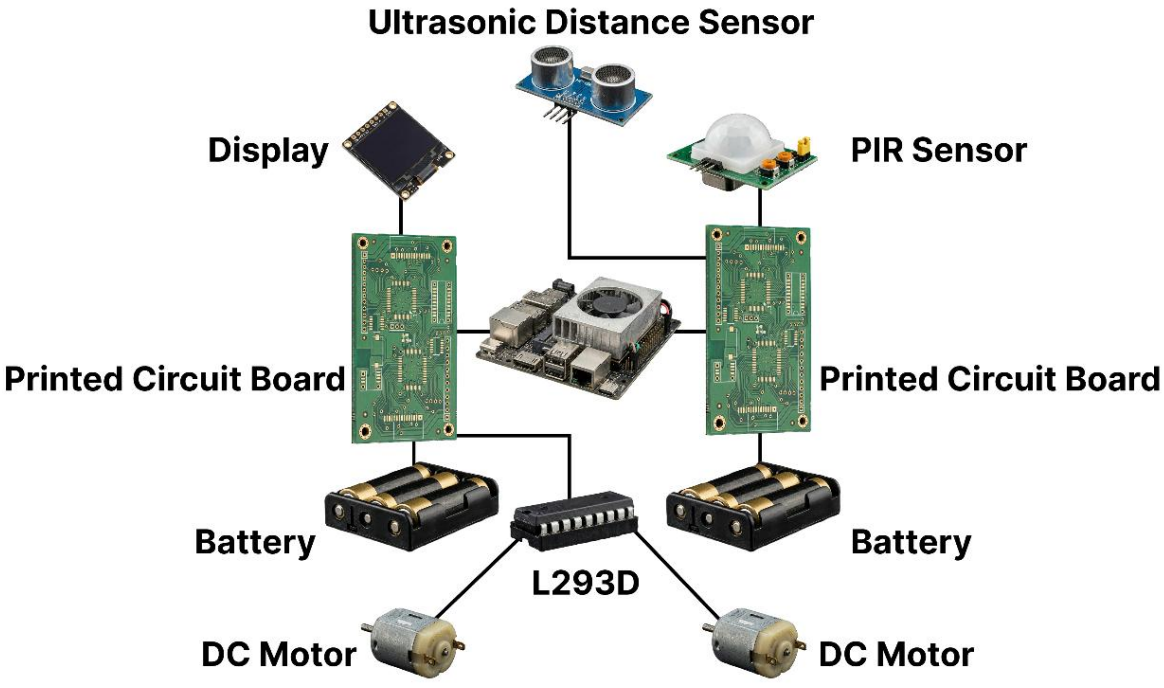


Figure 3.16 Configuration of IoT device equipped with Jetson Orin Nano

Table 3.1 shows that GPU inference was 2.57x faster on average than CPU inference (CPU: 9.64 tokens/s, GPU: 24.76 tokens/s). Based on this result, edge-side LLM inference in our system prioritizes GPU execution to reduce end-to-end delay from user interaction to device control. Offloading inference to GPU also frees CPU resources for sensing, communication, and control, improving overall responsiveness.

(2) Raspberry Pi 4

(a) **Role**

Functions as a standard gateway with general-purpose processing and rich I/O interfaces. It can host a local LLM (`Llama-3.2-3B-Instruct-Q4_K_M.gguf`) [18].

(b) Functions (Input/Output and Devices)

(i) Vision: Image capture and transfer using camera module (Picamera2)

(ii) Control (movement): Independent control of servo motor (hand) and DC motor (foot)

(iii) Expression:

- LED (red/yellow/green) lighting pattern control
- Melody playback by passive buzzer
- Animation display of robot eyes on display (ST7735)

(c) Control unit (granularity of operation interface)

Supports part-level control such as "right hand (servo)" and "left hand (servo)."

(d) **Features (positioning in the system)**

Acts as the core node for visual extensions and supports multimodal situational reasoning by sending captured images to the server.

(e) Figures/Tables (References)

Images of the actual device and its configuration are shown in Figures 3.17 and 3.18.

Future scalability

At first glance, routing tasks from the IoT agent to an SLM and then running inference on edge devices may appear inefficient because it adds an extra processing layer. However, this design is reasonable under future scale assumptions. If appliance-class hardware increasingly supports practical AI models, this architecture enables advanced in-home assistance while keeping household data (text, audio, image, video) local rather than uploading it externally. It also supports minimum viable inference/control even under poor connectivity, which can improve latency and operating cost.

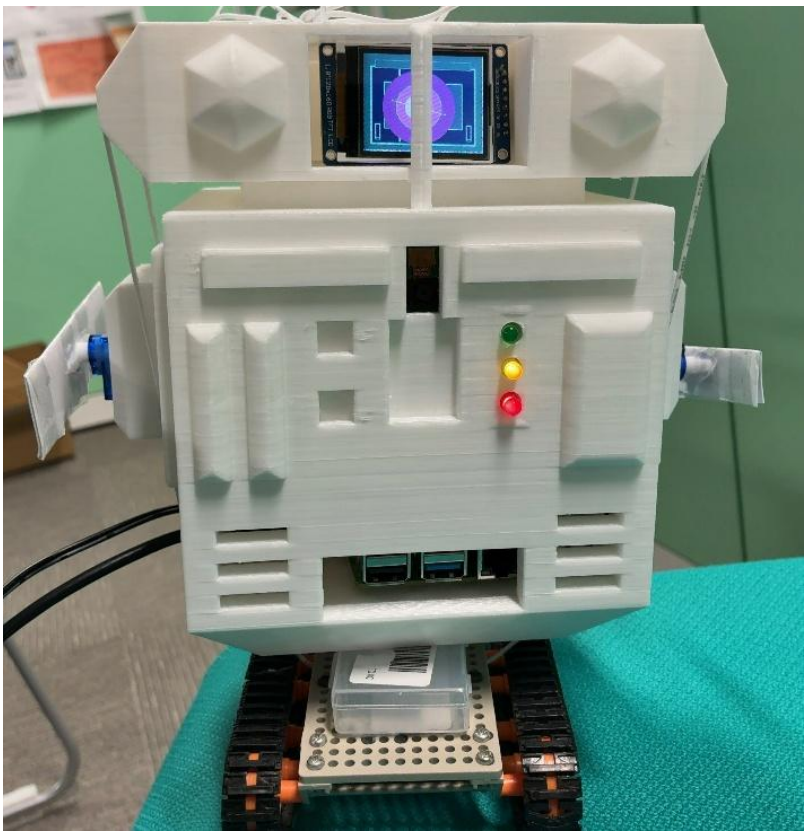


Figure 3.17 IoT device equipped with RaspberryPi4

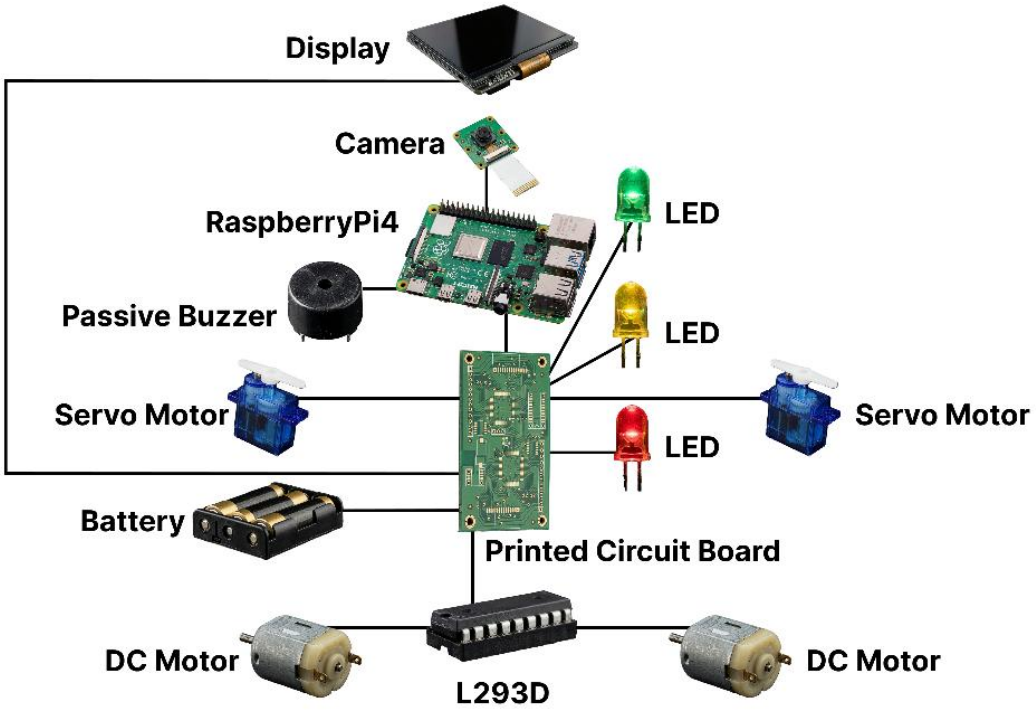


Figure 3.18 Configuration of IoT device equipped with RaspberryPi4

(3) Raspberry Pi Pico W

(a) **Role**

A low-power, low-cost microcontroller endpoint implemented in MicroPython.

(b) Functions (Input/Output and Devices)

Communicates with the web server and executes lightweight JSON commands for:

(i) LED flashing

(ii) buzzer playing

(iii) DC motor drive

(iv) Facial animation and text output on LCD (HD44780)

(c) Execution form (control/orchestration)

Also supports parallel execution.

(d) Characteristics (division of responsibilities/constraints)

Because of limited resources, it does not perform complex inference and is specialized for executing concrete function calls generated server-side. It models common household appliances (e.g., air conditioners, washing machines) that run on microcontrollers without full operating systems.

(e) Figures/Tables (References)

Images of the actual device and its configuration are shown in Figures 3.19 and 3.20.

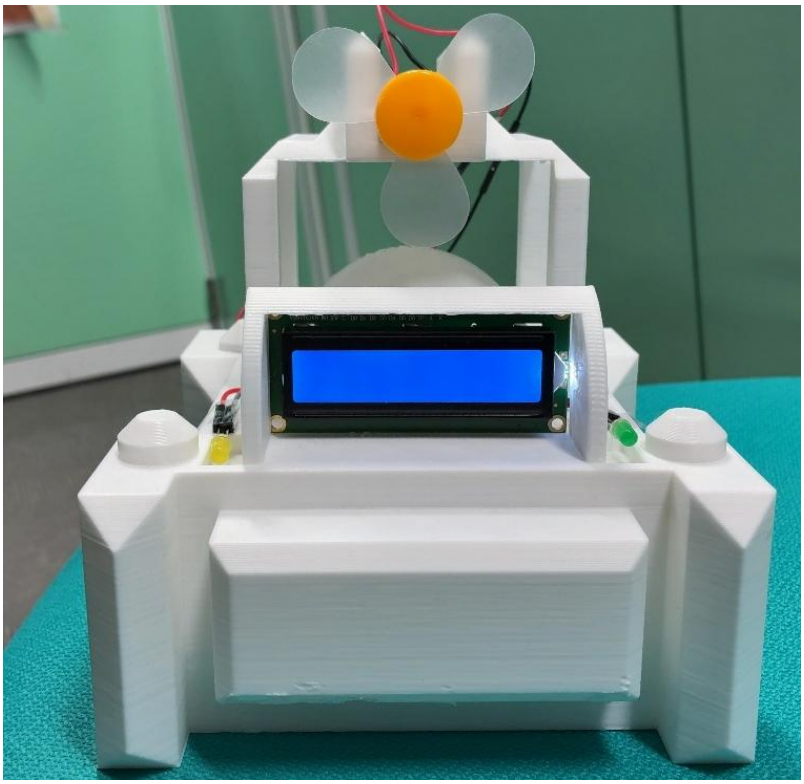


Figure 3.19 IoT device equipped with RaspberryPi Pico W

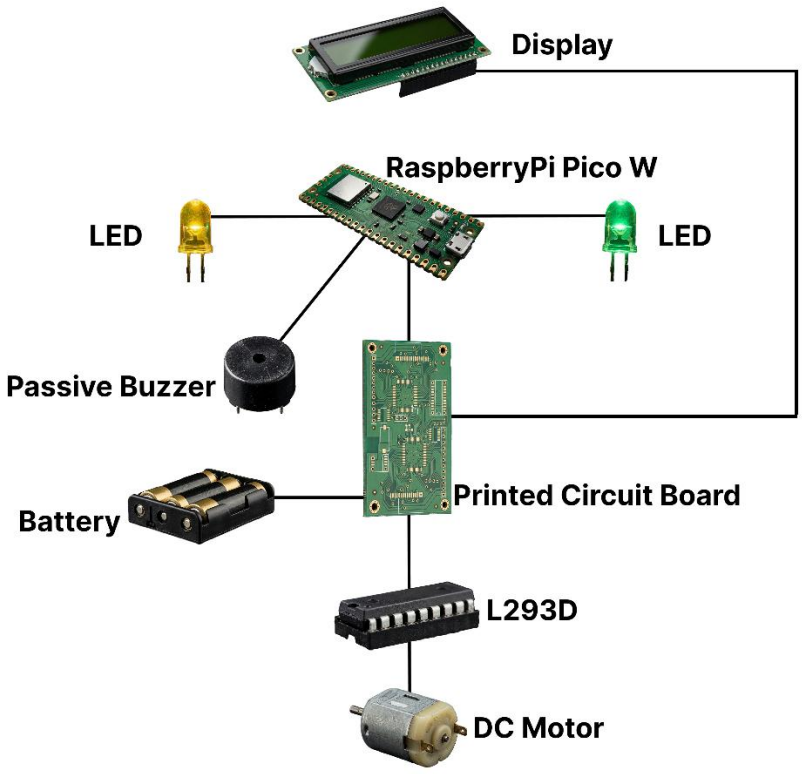


Figure 3.20 Configuration of IoT device equipped with RaspberryPi Pico W

3.4 Life-Style Agent

The Life-Style agent ("life-support agent") serves as the system's knowledge base. It specializes in document-grounded information such as household rules, appliance manuals, and practical daily-life knowledge (cooking, cleaning, laundry, etc.). The orchestrator references it through the alias "knowledge base function." Its core technology is RAG: relevant evidence is retrieved from a large pre-vectorized document store, and the LLM then generates natural responses grounded in that evidence. This enables individualized support within the scope of the curated knowledge base, beyond generic model priors. Figure 3.21 shows an example screen.

3.4.1 Technology stack and data processing

This agent is implemented with LangChain [19] and FAISS (Facebook AI Similarity Search) [12]. Figure 3.22 illustrates the RAG-centered architecture.

(1) **Data ingestion:** A script set processes structured JSONL Q&A data (`{"question": "...", "answer": "..."}`), splits documents into suitable chunks, embeds them with models such as `intfloat/multilingual-e5-large` [20], and persists them as FAISS indices.

(2) **Multi-index structure:** The vector database is partitioned by topic into subdirectories, enabling selective or cross-index retrieval as needed. This supports efficient management of heterogeneous domains (e.g., daily-life guidance and appliance operation).

(3) **Dynamic model selection:** As in other agents, LLM backends (Gemini, Claude, GPT, Groq) can be switched dynamically according to cost and response-speed requirements.

Figure 3.21 Life-Style agent screen example

3.4.2 RAG-Based Answer Generation Process

User or orchestrator queries are processed as follows:

(1) Retrieval: Embed the query and extract top-K semantically similar chunks from FAISS.

(2) Generation: Provide retrieved chunks plus the query to an LLM (Gemini, Claude, GPT, Groq). System prompts enforce evidence-grounded answers and explicit citation of source files to suppress hallucinations.

(3) Response: Return generated answer text together with source metadata (file name and page), enabling user-side traceability.

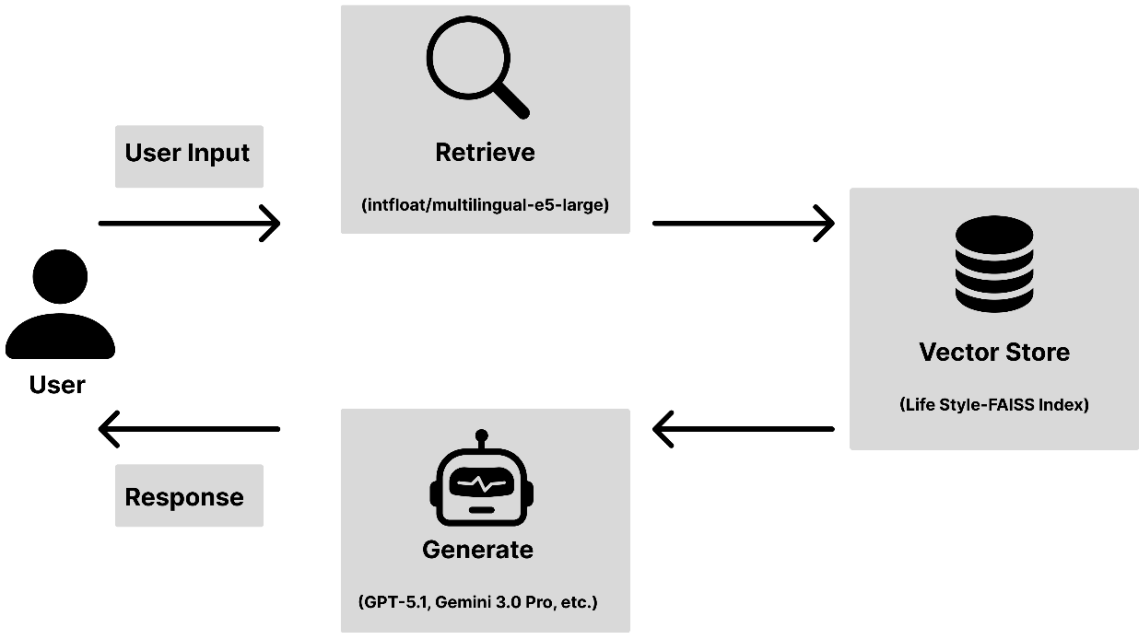


Figure 3.22 Life-Style agent configuration

3.4.3 External Interfaces and MCP

In addition to its RESTful API, this agent also runs as an MCP server and exposes knowledge-base tools externally. The RAG answer-generation tool accepts daily-life questions and returns evidence-grounded responses generated through the RAG pipeline. Whether conversation history is stored can also be controlled.

This allows MCP-enabled clients and peer agents to access knowledge-base functionality through a unified protocol without handling HTTP-level implementation details.

3.5 SchedulerAgent

The Scheduler agent ("schedule-management agent") is a web-application-based component for managing user routines, tasks, and diary entries. It provides a natural-language chat interface for schedule lookup, task creation/update, and completion reporting. Figure 3.23 shows an example screen.

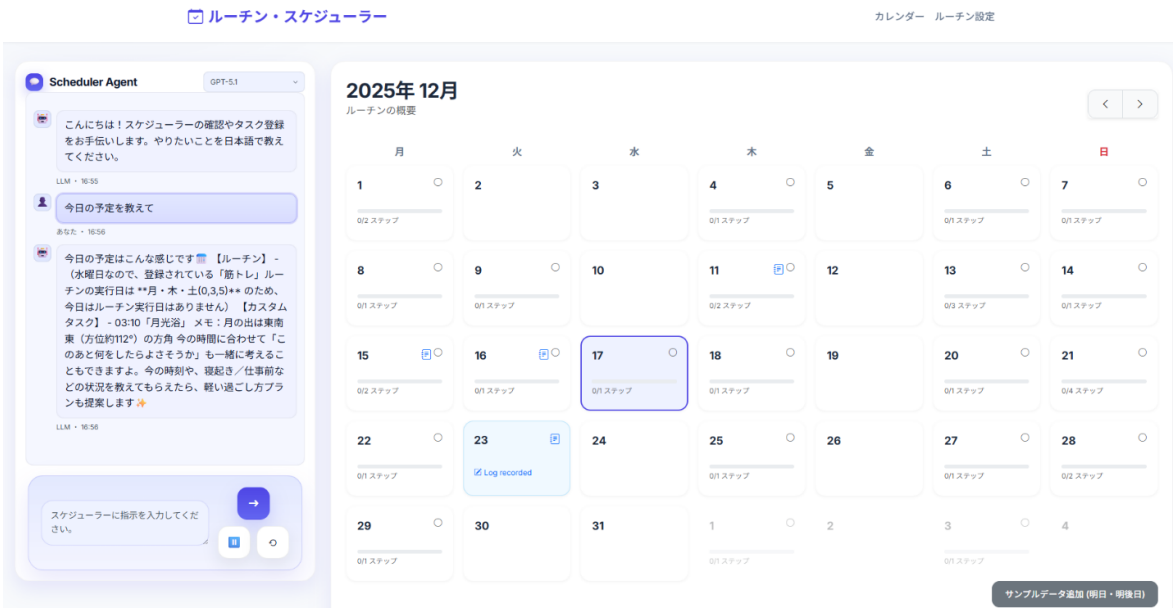


Figure 3.23 Scheduler agent screen example

3.5.1 Overview and role

Its main role is efficient management of recurring routines and ad hoc tasks in daily life. Unlike conventional to-do applications, it integrates an LLM, allowing natural utterances such as "Schedule a meeting tomorrow at 8 a.m." or "I finished today's workout." It also includes a diary function, allowing users to retain life logs together with task-completion status. Figure 3.24 shows the architecture.

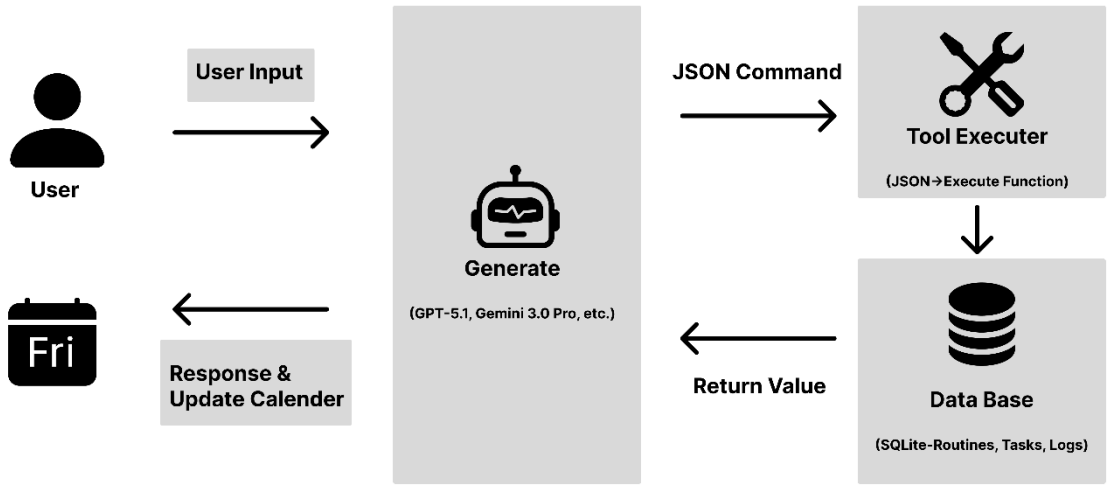


Figure 3.24 Scheduler agent configuration

3.5.2 System architecture

This agent is implemented as a Flask web application with SQLite (via SQLAlchemy) for persistence. Main components are:

(1) **Web server:** Entry point that renders HTML templates and exposes API endpoints. It also

defines database models (`Routine`, `Step`, `DailyLog`, `CustomTask`, `DayLog`), plus an evaluation-seed function and an evaluation module for recording model-response accuracy.

(2) **LLM client:** Provider-agnostic module for OpenAI, Gemini, Anthropic, and others. It calls models through a common interface and parses function-calling outputs.

(3) **Front end:** SPA-like interface built with HTML/CSS/vanilla JavaScript. Chat and task-list views are integrated, with asynchronous server communication. It also provides an embeddable calendar view for integration with external pages or dashboards.

3.5.3 Natural Language Interaction and Tool Invocation by LLM

The core feature of the Scheduler agent is reliable task manipulation via LLM function calling, combined with response generation grounded in actual execution outcomes. When a user sends a message, the system assembles context (current date/time, unfinished tasks, routines, recent daily logs) and sends it to the LLM.

When the LLM emits tool calls, the system does not respond immediately; it executes an explicit execution/summarization pipeline:

(1) Action decision**:** The LLM interprets user intent and generates JSON tool calls for database operations.

(2) Database execution**:** The server executes the calls (create/delete/update), determines success/failure, and produces an execution-result log.

(3) Response generation**:** The execution-result log is fed to another LLM pass (or separate context) to generate the final user-facing report.

This prevents inconsistencies such as "LLM says done but DB update failed," while preserving both operational reliability and natural-language usability.

The main tools available have been significantly expanded as follows.

(1) **Task management:**
   (a) create_custom_task: Create a new task.
   (b) delete_custom_task: Delete task.
   (c) toggle_custom_task: Toggle completed/uncompleted.
   (d) rename_custom_task, update_custom_task_time, update_custom_task_memo: Edit detailed task information.

(2) **Routine management:**
   (a) add_routine, delete_routine: Define and delete habits (routines).
   (b) add_step, delete_step: Edit each step in the routine.
   (c) toggle_step: Toggle step completion/incomplete.

(3) **Log/information acquisition:**
   (a) append_day_log: Addition to daily report.
   (b) update_log: Overwrite correction of daily report.
   (c) get_day_log: Get daily report for specified day.

(d) list_tasks_in_period: Get a list of tasks in the specified period.

(e) get_daily_summary: Get a summary of the day's activities (tasks/daily reports).

This multifunctional set of tools allows for advanced schedule management interactions, such as "checking what you did last Friday," "listing next week's plans," and "correcting daily habits," in addition to simple schedule entry.

3.5.4 Dynamic model selection

Like other agents, the Scheduler agent supports dynamic model switching. Models from Gemini, Claude, GPT, and Groq can be selected according to task complexity, enabling cost-performance optimization.

4. Evaluation

4.1 Standalone Performance Evaluation

In this section, each agent is evaluated independently under multiple LLM settings. The results are used to select configurations for the integrated evaluation.

4.1.1 Model performance evaluation

To analyze how model capability affects agent-level task performance, we classify models into three classes using parameter scale and MMLU (Massive Multitask Language Understanding), a common reasoning benchmark. For proprietary models with undisclosed parameter counts, we use published benchmark scores and provider-level performance categories as references.

(1) **Strong:** Frontier-class models with high public benchmark scores (e.g., MMLU), generally capable of stable multi-step reasoning, long-instruction following, and context retention. ※MMLU and parameter count are not universal indicators. They do not directly guarantee tool-operation accuracy or UI-operation success rate. Therefore, "strong" is treated as a relative category in this study.

(2) **Normal:** Mid-range models with sufficiently strong reasoning but typically less stability on difficult problems than frontier-class models. They often provide a favorable cost-performance tradeoff.

(3) **Weak:** Lightweight models (including edge-oriented models) optimized for low cost and low latency, suitable for lighter workloads such as routing, simple classification/extraction, and short instruction handling.

Using this classification, Table 4.1 defines the model set used to analyze system robustness and cost-performance tradeoffs.

Table 4.1 Model configuration of LLM used for evaluation

Strength	Provider	Model ID
strong		
	OpenAI	GPT-5.1
	Google	Gemini 3 Pro
	Anthropic	Claude Opus 4.5
normal		
	Anthropic	Claude Haiku 4.5
	Groq	Llama 3.3 70B
	Groq	Qwen3 32B
weak		
	Google	Gemini 2.5 Flash Lite
	Groq	Llama 3.1 8B
	Groq	GPT-OSS 20B

Through this evaluation, we aim to derive practical criteria for selecting LLMs according to agent-task complexity and to improve both system robustness and cost efficiency.

4.1.2 Browser Agent Evaluation

To evaluate Browser-agent capability, we used WebArena, a standard benchmark for autonomous web-agent evaluation [21]. WebArena provides realistic, reproducible web environments and evaluates agents on functional correctness in live web applications (e.g., e-commerce, forums, GitLab), not just static HTML parsing.

In this study, we evaluated only the Shopping environment, which reflects frequent user workflows and requires multi-step operations such as product search, cart insertion, and attribute filtering. The environment is built on Adobe Magento and includes realistic product data.

Evaluation procedure: Shopping tasks were loaded using the integrated WebArena task loader, and success/failure was judged automatically by:

(1) URL Match: Whether the final browser state reached a required URL (e.g., order completion page, target product page).

(2) String Match: Whether page text or agent output contained required values (exact or partial match).

(3) DOM state verification (Program HTML Check)**:** Whether JavaScript checks on designated DOM elements (e.g., cart total, filter state) matched expected values.

To compare model effects, experiments used the three representative models from Section 4.1.1: Strong (GPT-5.1), Normal (Qwen 32B), and Weak (GPT-OSS 20B). We measured task success rate and completion steps to analyze the relationship between LLM reasoning and web-tool operation accuracy.

The evaluation results are shown in Table 4.2.

Table 4.2 Browser agent evaluation in WebArena

Model	Number of Successes/Total Number	Score	Required Time
GPT-5.1	61 / 187	32.6%	267minutes 48.0seconds
Qwen 32B	43 / 187	23%	155minutes 5.1seconds
GPT-OSS 20B	49 / 187	26%	125minutes 3.9seconds

Although model size and performance often correlate, under this setup GPT-OSS 20B (26%) exceeded Qwen 32B (23%). Because WebArena outcomes are affected by runtime variability (search branching, pop-ups, etc.), this gap alone is insufficient to conclude superiority. Future work will run each model multiple times with different random seeds and report mean/variance (or 95% confidence intervals) for reproducibility. If the difference persists, web-UI success may depend not only on general reasoning quality but also on command-format compatibility and

instruction-following stability.

4.1.3 IoT Agent Evaluation

For standalone IoT-agent evaluation, we designed the following 10 tasks to verify two capabilities: (1) conversion of natural-language instructions into device-control commands, and (2) correct situational reasoning using visual information. Tasks range from single-device operations to coordinated multi-device operations (Jetson, Raspberry Pi 4, Raspberry Pi Pico W), including abstract prompts such as "darkness" and "entertain children."

Evaluation criteria:

○: All required device operations for the task were completed.

△: The task required multiple devices and at least one device operation succeeded.

×: All required device operations failed.

(1) Check for 3 seconds to see if there is any movement.

(2) Check the surrounding situation, and when the shooting is finished, sound the success sound with the buzzer.

(3) Move Jetson forward for 3 seconds, and when it stops, display "Arrived" on the display screen.

(4) Display "funny" on the Pi4 robot and wave your hand (servo) vertically in affirmation.

(5) Investigate the room environment. While measuring the temperature with Pico, blow air with the propeller.

(6) Give a welcome performance. Make Pico's screen smile (happy) and play a welcome melody (startup) from Pi4.

(7) Confirm the existence of all devices. Jetson displays "Ready", Pi4 displays "I'm here", and Pico lights up yellow.

(8) I feel like there is something in the dark. Measure the distance to the wall and turn on all the LEDs. Do everything at the same time.

(9) Start intruder warning mode. Display "alert" on the Jetson's display, make Pi4's LED flash like a police car, and Pico sound the buzzer.

(10) Perform some action on all devices to entertain a child.

The evaluation results are shown in Table 4.3.

Table 4.3 Outcomes of each model on IoT-agent evaluation tasks

Task ID	GPT-5.1	Gemini 3 Pro	Anthropic Claude Opus 4.5	Anthropic Claude Haiku 4.5	Llama 3.3 70B	Qwen 3 32B	Gemini 2.5 Flash-Lite	Llama 3.1 8B	GPT-OSS 20B
1	○	○	○	○	○	○	○	×	×
2	○	○	○	○	○	○	○	○	○
3	○	○	○	○	×	○	○	○	○
4	△	○	○	○	△	○	○	○	○
5	○	△	○	○	△	△	△	○	○
6	○	○	○	○	○	○	○	△	○
7	△	○	○	△	○	○	○	△	○
8	△	△	○	○	○	○	○	△	△
9	○	○	○	○	△	○	○	○	○
10	○	○	○	○	○	△	△	○	△

Results indicate that the Strong model group (larger models) generally showed high instruction comprehension, but still did not achieve perfect performance across all control tasks. For example, GPT-5.1 and Claude Haiku 4.5 occasionally failed on tasks such as No.7 and No.8, including LED color mismatch and timeout cases, suggesting difficulty with feedback waits and complex physical-device branches.

By contrast, Claude Opus 4.5 and Gemini 3 Pro produced context-aware creative behavior for abstract prompts (No.10: "entertain children"), such as child-oriented message and emoticon output.

Interestingly, some weaker/smaller models, especially Llama 3.1 8B, executed commands faster and more stably than higher-tier models on tasks that required less complex reasoning (e.g., No.3 and No.10). This suggests that large models are not always optimal for real-time edge-control workloads.

Failures in specific tool calls (e.g., temperature retrieval, specific display output in No.4 and No.5) were observed for Gemini 2.5 Flash Lite and Llama 3.3 70B. These failures may reflect prompt/tool-definition compatibility or training-data bias in control-code patterns, not only intrinsic model capability.

Overall, IoT control requires not only high-capability models but also model-size selection by task characteristics (low-latency response vs deep context reasoning), together with prompt engineering to absorb model-specific quirks.

4.1.4 Life-Style Agent Evaluation

For standalone Life-Style-agent evaluation, we prepared 10 tasks. Each task consists of a category-specific question (home appliances, cooking, finance, or mental health) and an expected answer.

Evaluation criteria:

○: Contains all core keywords from the expected answer.

△: Contains one or more core keywords from the expected answer.

×: Contains none of the core keywords from the expected answer.

(1) [Home appliances] What should be done immediately after ironing to prevent rebound wrinkles?
Correct answer: Cool it on a hanger.
(2) [Cooking] When determining when an avocado is ready to eat, what should I check other than the color of the skin?
Correct answer: It's best if you can feel the elasticity when you press it lightly with your finger, or if you can feel the stem coming off (it's ripe).
(3) [Finance] What is the annual investment limit for the "Tsumitate Investment Allowance" under the new NISA?
Correct answer: 1.2 million yen per year (100,000 yen per month).
(4) [Mental] How many seconds does the peak of anger last?
Correct answer: 6 seconds.
(5) [Cooking] What specific preparation keeps grilled chicken breast moist instead of dry?
Correct answer: Cut into slices, rub with salt koji or mayonnaise and leave for 10 minutes, or sprinkle with potato starch and bake over low heat.
(6) [Mental] What is the specific method of the "4-7-8 breathing method" that is effective when your chest is pounding due to anxiety?
Correct answer: Inhale for 4 seconds, hold for 7 seconds, and exhale for 8 seconds.
(7) [Cooking] I'm very tired and don't want to use a knife. I have pork belly and Chinese cabbage; what easy dish can I make?
Correct answer: "Layered steamed pork belly and Chinese cabbage." All you have to do is spread it out in a pot, add sake and chicken stock, and leave it alone.
(8) [Cooking] I want to meal-prep on weekends. Please suggest side dishes suitable for lunch boxes and freezing.
Correct answer: Chicken breast pickled in nanban, simmered hijiki and soybeans, stir-fried infinite peppers with miso, etc.
(9) [Finance] I am choosing between iDeCo and NISA for retirement savings. What are iDeCo's unique advantages and key restrictions?
Correct answer: The advantage is that the premium is fully tax deductible, resulting in high tax savings. The restriction is that, in principle, you cannot withdraw money until you are 60 years old.
(10) [Mental] I have an important presentation tomorrow and cannot sleep because of anxiety. What should I do?
Correct answer: Darken the room and try the "US military-style sleep method" for muscle relaxation. If you still can't fall asleep, say, "Just lying down and closing your eyes will give your body about 80% rest."
The evaluation results are shown in Table 4.4.

Table 4.4 Outcomes of each model on Life-Style-agent evaluation tasks

Task ID	GPT-5.1	Gemini 3 Pro	Anthropic Claude Opus 4.5	Anthropic Claude Haiku 4.5	Llama 3.3 70B	Qwen 3 32B	Gemini 2.5 Flash-Lite	Llama 3.1 8B	GPT-OSS 20B
1	○	○	○	○	○	○	○	○	○
2	○	○	○	○	○	○	○	○	○
3	○	○	○	○	○	○	○	○	○
4	○	○	○	○	○	○	○	○	○
5	○	○	○	○	○	△	○	○	○
6	○	○	○	○	○	○	○	○	○
7	○	○	○	○	○	○	○	○	○
8	○	○	○	○	○	○	○	○	○
9	○	○	○	○	○	○	○	○	○
10	○	○	○	○	○	○	○	○	○

Because scoring is based on core-keyword inclusion, semantically equivalent paraphrases can still be penalized if keywords are absent. The Qwen3 32B difference may therefore reflect limits of surface-matching evaluation (low paraphrase tolerance), not necessarily model capability. Future evaluation should include synonym-aware and semantic-similarity-based scoring.

Also, all tasks used the same embedding model (`intfloat/multilingual-e5-large`) in retrieval, so LLM inputs from the RAG retrieval phase were largely consistent across models. This likely reduced inter-model variance and indicates that retrieval quality (embedding model performance) dominated generation differences in this benchmark.

4.1.5 Scheduler Agent Evaluation

For standalone Scheduler-agent evaluation, we prepared the following 10 tasks.

Evaluation criteria:

○: All 3 trials succeeded.

△: 1-2 trials succeeded.

×: All trials failed.

(1) Add the task "Buy detergent."

(2) Show my schedule from tomorrow through the day after tomorrow.

(3) Rename "buying detergent" to "buying fabric softener."

(4) Mark the "buy fabric softener" task as completed.

(5) Add a "dentist" appointment at 3:30 p.m. tomorrow.

(6) After all, "buy fabric softener" should be deleted. Then, add a note saying "Don't forget your insurance card" to the "Dentist" task.

(7) Create a new routine called "muscle training" for Mondays and Thursdays.

(8) Add a step (10 minutes) called "squats" to the "muscle training" routine you just created.

(9) Also set the "muscle training" routine for Saturdays.

(10) Show me the daily report from last Friday.

The evaluation results are shown in Table 4.5.

GPT-5.1: In Task 10, date arithmetic failed. With verification date Tuesday, December 9, 2025, "last Friday" was interpreted as November 28, 2025 (correct: December 5, 2025). Day-of-week recognition was correct, but cross-week calculation was incorrect.

(1) Claude Haiku 4.5: In Tasks 4, 6, 7, 8, and 9, tool calls were often not emitted (text-only responses, no function-calling JSON).
(2) Llama 3.3 70B: Task 9 used the wrong weekday (Sunday instead of Saturday), and Task 10 produced a one-day date shift (December 6).
(3) Gemini 2.5 Flash Lite: Many failures occurred because required JSON tool calls were not generated as instructed. A likely cause is over-application of the system rule "do not output JSON in user-facing responses," mistakenly suppressing tool-call JSON as well.
(4) Llama 3.1 8B: In Tasks 3 and 5, generated Japanese arguments contained corrupted strings (e.g., "Enkeikaku wo wo"), likely due to tokenizer limits or weak Japanese coverage in training data. Task 10 also miscalculated the date (December 4).
(5) GPT-OSS 20B: In compound instructions (e.g., Task 6: delete + add), one sub-instruction was often dropped. Task 10 also produced a large date error (December 2).
(6) Qwen3 32B: Solved all tasks despite being in the Normal class, indicating high suitability for Scheduler-agent workflows.

Table 4.5 Outcomes of each model on Scheduler-agent evaluation tasks

Task ID	GPT-5.1	Gemini 3 Pro	Claude Opus 4.5	Claude Haiku 4.5	Llama 3.3 70B	Qwen3 32B	Gemini 2.5 Flash Lite	Llama 3.1 8B	GPT-OSS 20B
1	○	○	○	○	○	○	×	○	○
2	○	○	○	○	○	○	×	○	○
3	○	○	○	○	○	○	×	△	○
4	○	○	○	×	○	○	×	○	○
5	○	○	○	○	○	○	×	△	○
6	○	○	○	×	○	○	×	○	×
7	○	○	○	×	○	○	△	○	○
8	○	○	○	×	○	○	×	○	○
9	○	○	○	×	×	○	×	△	○
10	×	○	○	○	×	○	△	×	×

Main failure factors for each model:

4.2 Scenario-Based Integrated Evaluation

In this section, we evaluate how effectively the full system executes ambiguous user instructions through agent collaboration. The primary focus is the effect of personalized memory on task

success and execution efficiency.

4.2.1 Experimental settings and evaluation indicators

LLMs for each agent were selected based on Section 4.1 standalone results, prioritizing either the best cost-performance balance for each task type or the fewest critical error patterns. The selected configuration is shown in Table 4.6.

Experimental setup:

(a) Baseline (no memory): No user-specific personal information retained.

(b) With memory: Agent behavior optimized using personal data (address, hobbies, health status, etc.).

(i) Subtasks: Evaluate how well ambiguous instructions are concretized.

(ii) All personal data used in this evaluation (address, hobbies, health status, etc.) are fictitious and contain no real personally identifiable information.

(2) Evaluation metrics:

(a) **Achievement level:** For each scenario subgoal, judge achieved (○) or not achieved (×).

(b) **Score calculation:**

(i) Add 1 point per achieved subgoal (○).

(ii) Subtract 1 point per clarification question asked by the agent.

※ This weighting treats user burden (cognitive/time cost of additional questions) as a negative factor equivalent to one achieved subgoal.

(iii) Per-scenario raw score: `G = number of ○`

(iv) `Q = number of questions`

(v) In this study, a "question" is one utterance requesting additional task-critical information (e.g., date/time, location, target identity).

(vi) Final score: `Score = max(0, G - Q)`

Supplementary metric: browser step count (reference only).

Table 4.6 Usage model configuration

Agent	Model
Orchestrator	Claude Opus 4.5
IoT agent	Claude Haiku 4.5
Browser agent	GPT-5.1
Scheduler agent	Qwen3 32B
Life-Style agent	Llama 3.3 70B

4.2.2 Evaluation scenario and criteria

We evaluate the system using the following 10 scenarios.

(1) Scenario 1

(a) User instruction: "Let's do our best today too!"

(b) Task: The orchestrator selects and generates an appropriate response to an abstract utterance.

(c) Criterion 1: Can the orchestrator respond appropriately without invoking other agents?

(2) Scenario 2

(a) User instruction: "Check the weather forecast for next week and record that day's weather."

(b) Task: Search weekly weather using residential-location memory and record the result.

(c) Criterion 1: Can weather be searched for the user's residential area using memory?

(d) Criterion 2: Is the result recorded correctly?

(3) Scenario 3

(a) User instruction: "Find a one-way flight from Tokyo to Atlanta departing on a weekday in January 2026, and add the cheapest one to my calendar."

(b) Task: Search flight-booking sites for the cheapest route option and register it in the calendar.

(c) Criterion 1: Was flight information retrieved successfully?

(d) **基準2:** Criterion 2: Was it registered in the calendar?

(4) Scenario 4

(a) User instruction: "Find the direction and time the moon will be visible tonight, schedule a 'moonbath' then, and turn off the lights."

(b) Task: Determine local moonrise direction/time, register the event, and switch off lights via IoT.

(c) Criterion 1: Was moon direction for the user's residential area identified?

(d) Criterion 2: Was the schedule registered?

(e) Criterion 3: Were device lights turned off?

(5) Scenario 5

(a) User instruction: "After checking an online earthquake-evacuation manual and understanding interpersonal precautions at shelters, flash the warning light (red LED) as a drill."

(b) Task: Sequentially perform web manual lookup, Life-Style-agent advisory retrieval, and IoT warning-light control.

(c) Criterion 1: Was the evacuation manual confirmed?

(d) Criterion 2: Was interpersonal-relations guidance obtained appropriately from the Life-Style agent?

(e) Criterion 3: Was IoT control executed?

(6) Scenario 6

(a) User instruction: "Create a dinner recipe, add it to today's tasks, and display 'complete' on the device."

(b) Task: Propose a recipe reflecting user health status/preferences, add it to task list, and display completion text.

(c) Criterion 1: Was the recipe aligned with health constraints/allergies?

(d) Criterion 2: Was it added to today's tasks?

(e) Criterion 3: Was "complete" displayed on at least one device?

(7) Scenario 7

(a) User instruction: "Find an event related to my hobby, tell me about it, and add it to my schedule on that date."

(b) Task: Search for hobby-related events near the user's location and register a relevant event.

(c) Criterion 1: Was a hobby-related event found that was still upcoming at evaluation time?

(d) Criterion 2: Was local-area search performed around the address (Kamakura, Nakameguro, Omiya)?

(e) Criterion 3: Was the event registered in the schedule?

(8) Scenario 8

(a) User instruction: "Is there a nearby store for that food I love? If so, write it in today's memo and display the store name in English."

(b) Task: Search for nearby stores serving user-preferred food, write memo entry, and display store name.

(c) Criterion 1: Was search performed based on user preference (favorite food)?

(d) Criterion 2: Was a memo entry created?

(e) Criterion 3: Was the store name displayed?

(9) Scenario 9

(a) User instruction: "Find a nearby place for a picnic. Then tell me how to discuss and plan it with my family. Schedule it for next Sunday and notify me by buzzer when done."

(b) Task: Combined workflow covering nearby spot search, Life-Style advisory response, schedule registration, and buzzer notification.

(c) Criterion 1: Were nearby picnic spots identified?

(d) Criterion 2: Did the Life-Style agent provide advice?

(e) Criterion 3: Was the schedule registered?

(f) Criterion 4: Did the buzzer sound?

(10) Scenario 10

(a) User instruction: "Run your usual routine."

(b) Task: Recognize a user-defined routine and execute multiple contained actions sequentially.

(c) Criterion 1: Was the routine recognized?

(d) Criterion 2: Was the first routine action executed?

(e) Criterion 3: Was the second routine action executed?

The evaluation results with and without memory are shown in Table 4.7, Table 4.8, Table 4.9, and Table 4.10.

4.2.3 Evaluation results

Table 4.7 Baseline (no memory) evaluation results

Scenario	Criterion 1	Criterion 2	Criterion 3	Criterion 4	Number of questions	Number of steps (browser)	Score
1	○	—	—	—	0	—	1
2	×	○	—	—	0	19	1
3	×	×	—	—	0	19	0
4	×	×	○	—	0	16	1
5	○	○	○	—	0	17	3
6	×	○	○	—	0	—	2
7	○	×	○	—	1	15	1
8	×	○	○	—	0	3	2
9	×	○	○	○	0	7	3
10	×	○	○	—	1	—	1
Total						96	15

(1) Scenario 2: The system retrieved weather for December 24, 2025, but searched Tokyo rather than the user's residential area.

(2) Scenario 4: Moon information was also searched for Tokyo without clarification.

(3) Scenario 6: The system did not ask about health constraints or preferences and proposed only one recipe already registered in the Life-Style agent.

(4) Scenario 7: The system asked about hobbies but not address; however, it did retrieve events in 2025. This suggests variance in year/date interpretation.

(5) Scenario 8: The system searched for a generally popular ramen restaurant and completed quickly.

Table 4.8 Persona 1 (with memory) evaluation results

Scenario	Criterion 1	Criterion 2	Criterion 3	Criterion 4	Number of questions	Number of steps (browser)	Score
1	○	—	—	—	0	—	1
2	○	○	—	—	0	20	2
3	×	×	—	—	0	20	0
4	○	○	○	—	0	4	3
5	○	○	○	—	0	14	3
6	○	○	○	—	0	—	3
7	×	○	○	—	0	20	2
8	○	○	○	—	0	18	3
9	○	○	○	○	0	19	4
10	○	○	○	—	0	—	3
Total						115	24

(1) Scenario 3: The run failed during flight date-widget selection.

(2) Scenario 5: Agent selection and handoff were appropriate.

(3) Scenario 6: The system proposed a low-salt dinner recipe reflecting high blood pressure and buckwheat allergy constraints.

(4) Scenario 7: Two Kamakura-area events were registered (12/7 and 12/9), but both had already occurred at evaluation time; therefore Criterion 1 was marked ×.

(5) Scenario 10: Routine recognition and execution were correct.

Table 4.9 Persona 2 (with memory) evaluation results

Scenario	Criterion 1	Criterion 2	Criterion 3	Criterion 4	Number of questions	Number of steps (browser)	Score
1	○	—	—	—	0	—	1
2	○	○	—	—	0	11	2
3	○	○	—	—	0	17	2
4	○	○	○	—	0	9	3
5	○	○	○	—	0	19	3
6	○	○	○	—	0	—	3
7	×	○	○	—	0	19	2
8	○	○	○	—	0	16	3
9	○	○	○	○	0	12	4
10	○	○	○	—	0	—	3
Total						103	26

(1) Scenario 7: Although the system recognized the current month as December 2025, it searched past events (December 2024-January 2025) and scheduled one in January 2025.

(2) Scenario 9: A suitable park near Nakameguro was found.

(3) Scenario 10: Memory retrieval and execution were accurate.

Table 4.10 Persona 3 (with memory) evaluation results

Scenario	Criterion 1	Criterion 2	Criterion 3	Criterion 4	Number of questions	Number of steps (browser)	Score
1	○	—	—	—	0	—	1
2	○	○	—	—	0	7	2
3	○	○	—	—	0	18	2
4	○	○	○	—	0	4	3
5	○	×	○	—	0	10	2
6	○	○	○	—	0	—	3
7	×	○	○	—	0	17	2
8	○	○	○	—	0	19	3
9	○	○	○	○	0	3	4
10	○	○	○	—	0	—	3
Total						78	25

(1) Scenario 2: The system correctly retrieved weather information for the user's residential area.

(2) Scenario 4: A visually clear, simple-structure site was accessed first, reducing browser steps.

(3) Scenario 5: Criterion 2 should ideally be handled by the Life-Style agent, but a web search was used instead; although the answer content was acceptable, this was treated as failure due to incorrect tool allocation.

(4) Scenario 6: Responses reflected allergies and preferences.

(5) Scenario 7: Events from January 2024 were searched despite the expected current-year context, indicating a year-handling error.

(6) Scenario 8: Nearby hamburger stores were searched appropriately.

(7) Scenario 9: Step count decreased because the system returned a quick response after nearby-park search without detailed follow-up.

The Browser agent did not always maintain accurate current-time grounding and occasionally retrieved 2024 information. A likely cause is prompt dilution: when temporal constraints are embedded in a very large system prompt, date information may lose relative salience. When the orchestrator passed an explicit "December 2025" date, retrieval was typically correct; when only "a day in December" was passed, 2024 results sometimes appeared. This behavior is consistent with knowledge-cutoff-related bias.

We also analyzed browser efficiency (step count). Before experiments, we hypothesized that memory would reduce steps by clarifying user preferences and constraints. In practice, no significant step-count difference was observed between memory and no-memory conditions. However, behavior-log analysis suggests a qualitative shift in steps:

Without memory, the agent tended to perform broad and shallow search.

With memory, the agent performed more verification-oriented steps (detail-page checks, cross-checking against user constraints such as preference and location).

This implies that memory does not simply reduce operational cost; instead, it encourages behavior that prioritizes output suitability, even with additional verification overhead. For real-world support tasks where failure cost is high, this is desirable.

Table 4.11 summarizes overall scores across evaluation settings.

Table 4.11 Evaluation results comparing presence and absence of memory

Evaluation target	Overall score (points)
1. No memory (Baseline)	15
2. Persona 1 (with memory)	24
3. Persona 2 (with memory)	26
4. Persona 3 (with memory)	25

These results confirm that enabling memory (user profile + contextual retention) significantly improves performance relative to baseline. Gains were most visible in interpretation of ambiguous

instructions and context-appropriate tool selection.

Evaluation target	Overall score (points)
1. No memory (Baseline)	15
2. Persona 1 (with memory)	24
3. Persona 2 (with memory)	26
4. Persona 3 (with memory)	25

5. Discussion

This chapter discusses the effectiveness of the proposed system, current limitations, and future directions based on the evaluation results in Chapter 4.

5.1 Personalization and Efficiency with the Memory Function

In the scenario-based integrated evaluation (Section 4.2), enabling memory produced a substantially higher score (average 25.0) than disabling memory (15). This indicates that the orchestrator-memory pair is highly effective at converting underspecified user utterances into concrete, executable tasks.

Another important result is the reduction in clarification questions. Without memory, the system could not reliably infer location and preference constraints, often resulting in generic outputs (e.g., nationwide weather, generic recipes) or additional user prompts. With memory enabled, the system autonomously referenced address and historical profile data (e.g., allergies, hobbies) to resolve expressions such as "as usual" and "nearby." In Scenario 6, for example, the system proposed recipes that reflected buckwheat allergy and high blood pressure, indicating value beyond simple automation.

This effect comes from cooperation between long-term memory (persistent profile attributes such as address, health, hobbies) and short-term memory (recent interests and temporary context). The most influential memory types were:

(1) **Address data:** In Scenarios 2 and 4, referencing stored residential area data (e.g., Kamakura, Nakameguro, Omiya) allowed the Browser agent to retrieve location-relevant results instead of defaulting to generic Tokyo queries.
(2) **Health status:** In Scenario 6, the Life-Style agent used stored conditions (e.g., buckwheat allergy, hypertension) to generate low-sodium and allergen-aware recipes.
(3) **Preferences/hobbies:** In Scenarios 7 and 8, long-term memory of hobbies (e.g., shogi, haiku) and preferred foods (e.g., ramen, hamburgers) enabled targeted event/store search and smoother downstream schedule/memo operations.

These observations show that performance gains depend not only on whether memory exists, but also on how personal-information types are designed and mapped to agent behaviors.

5.2 LLM Knowledge Cutoff and Time-Perception Challenges

One major issue identified in evaluation is mismatch between LLM knowledge cutoff and current-time grounding. Even when the system prompt explicitly specified the current period (December 2025), the agent repeatedly prioritized 2024 event/calendar information (Scenarios 2 and 7).

This suggests that temporal priors from training data sometimes outweigh short-term prompt context. In web-query generation, unless "2025" was explicitly included, the model tended to search older periods treated as "current" in pretraining distribution.

This is a common challenge for real-time deployment of LLM agents. Practical countermeasures include stricter prompt constraints (always inject explicit year in query generation) and RAG pipelines that prioritize recent data.

A concrete implementation is to attach explicit timestamps whenever the orchestrator dispatches tasks and whenever user prompts are processed. This can keep agents anchored to the current time axis and reduce drift toward stale data.

5.3 Limits and Potential of Complex Web UI Operations

Browser-agent evaluation (Section 4.1.2 and Scenario 3) shows that complex web-UI operation remains difficult for current LLM agents. Under WebArena conditions, success remained around 25% even with GPT-5.1. Failures on airline date-picker interactions indicate limits of purely DOM-centric approaches and of text-only situational reasoning.

For dynamic DOMs and visually anchored operations (e.g., calendar clicks), text-based HTML context alone is often insufficient. In contrast, simple search and information extraction tasks (Scenarios 2 and 4) showed high practicality.

A promising direction is multimodal UI grounding: combine DOM reasoning with visual screenshots via models such as GPT-5.1 and Gemini 3 Pro [22].

5.4 Role Allocation Between Edge AI and Cloud AI

In IoT-agent evaluation (Section 4.1.3), lightweight edge LLMs running on Jetson and Raspberry Pi (`Qwen3 1.7B` [17], `Llama-3.2 3B` [18]) successfully controlled devices without JSON syntax failures in command generation.

This indicates that with well-designed tool definitions (function calling) and prompts, small edge-side models can act as practical local reasoners. The results support the validity of the system's layered architecture: privacy-sensitive data handling, sensing, and low-latency control stay at the edge, while higher-level planning is delegated to cloud models.

6. Conclusion

This study designed and implemented a collaborative multi-agent system that integrates fragmented web services and physical IoT devices and autonomously executes tasks from ambiguous natural-language instructions. The architecture combines an MCP-standardized set of specialized agents (browser operation, IoT control, lifestyle knowledge retrieval, scheduling) under a memory-enabled orchestrator that maintains user context and preferences.

Evaluation yielded three key findings. First, long-term and short-term memory substantially improved context handling: total score increased by about 1.7x versus baseline, while clarification questions decreased from 2 (baseline) to 0 across the three memory-enabled personas (Tables 4.7-4.10). This enabled robust interpretation of underspecified expressions such as "as usual."

Second, edge-cloud cooperation was practical. Lightweight edge LLMs (1.7B-3B class) on Jetson/Raspberry Pi could reliably convert abstract instructions into device-control commands under appropriate prompt/tool design. This validates hierarchical intelligence allocation: low-latency and privacy-sensitive control at the edge, higher-level planning in the cloud.

Third, two limitations remain clear: temporal grounding drift related to knowledge cutoff, and weak robustness on complex GUI interactions under DOM-centric web control. These are common barriers for real-world LLM deployment. Future work should combine multimodal visual UI understanding with stronger real-time grounding in retrieval pipelines.

Overall, the results indicate that AI can evolve from tool automation into a context-aware partner that supports daily life across both digital and physical domains. Future work will validate long-term memory evolution in real deployment and expand integration with broader services and devices toward practical human-AI co-living environments.

References

[1] Schick, T. et al.: *Toolformer: Language Models Can Teach Themselves to Use Tools*, arXiv:2302.04761, https://arxiv.org/abs/2302.04761
(accessed 2025/12/18).

[2] Home Assistant: *2024.6: Dipping Our Toes in the World of AI Using LLMs*, Home Assistant Blog, https://www.home-assistant.io/blog/2024/06/05/release-20246/
(accessed 2025/12/18).

[3] Wu, Q. et al.: *AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation*, arXiv:2308.08155, https://arxiv.org/abs/2308.08155
(accessed 2025/12/18).

[4] Fourney, A. et al.: *Magentic-One: A Generalist Multi-Agent System for Solving Complex Tasks*, arXiv:2411.04468, https://arxiv.org/abs/2411.04468
(accessed 2025/12/18).

[5] Mozannar, H. et al.: *Magentic-UI: Towards Human-in-the-loop Agentic Systems*, arXiv:2507.22358, https://arxiv.org/abs/2507.22358
(accessed 2025/12/18).

[6] Cui, H. et al.: *LLMind: Orchestrating AI and IoT with LLM for Complex Task Execution*, arXiv:2312.09007, https://arxiv.org/abs/2312.09007
(accessed 2025/12/18).

[7] Shen, Y. et al.: *HuggingGPT: Solving AI Tasks with ChatGPT and Its Friends in Hugging Face*, arXiv:2303.17580, https://arxiv.org/abs/2303.17580
(accessed 2025/12/18).

[8] agno-agi: *Agno (AgentOS)*, GitHub, https://github.com/agno-agi/agno
(accessed 2025/12/18).

[9] King, E. et al.: *Sasha: Creative Goal-Oriented Reasoning in Smart Homes with Large Language Models*, arXiv:2305.09802, https://arxiv.org/abs/2305.09802
(accessed 2025/12/18).

[10] Anthropic: *Model Context Protocol*, https://modelcontextprotocol.io/
(accessed 2025/12/18).

[11] Rawat, M. et al.: *Pre-Act: Multi-Step Planning and Reasoning Improves Acting in LLM Agents*, arXiv:2505.09970, https://arxiv.org/abs/2505.09970
(accessed 2025/12/18).

[12] Facebook Research: *FAISS*, GitHub, https://github.com/facebookresearch/faiss (accessed 2025/12/18).

[13] browser-use: *browser-use*, GitHub, https://github.com/browser-use/browser-use (accessed 2025/12/08).

[14] LangChain: *LangGraph*, https://python.langchain.com/docs/langgraph/
(accessed 2025/12/18).

[15] Adimi, Alaa Dania: *Augmenting LLMs with Retrieval, Tools, and Long-term Memory*, Medium, https://medium.com/infinitgraph/augmenting-llms-with-retrieval-tools-and-long-term-memory-b9e1e6b2fc28
(accessed 2025/12/19).

[16] Zhong, W. et al.: *MemoryBank: Enhancing Large Language Models with Long-Term Memory*, arXiv:2305.10250, https://arxiv.org/abs/2305.10250
(accessed 2025/12/19).

[17] Unsloth: *Qwen3-1.7B-Q4_K_S.gguf*, Hugging Face, https://huggingface.co/unsloth/Qwen3-1.7B-GGUF/blob/main/Qwen3-1.7B-Q4_K_S.gguf
(accessed 2025/12/08).

[18] Unsloth: *Llama-3.2-3B-Instruct-Q4_K_M.gguf*, Hugging Face, https://huggingface.co/unsloth/Llama-3.2-3B-Instruct-GGUF/blob/main/Llama-3.2-3B-Instruct-Q4_K_M.gguf
(accessed 2025/12/08).

[19] LangChain: *LangChain*, https://python.langchain.com/ (accessed 2025/12/18).

[20] intfloat: *multilingual-e5-large*, Hugging Face, https://huggingface.co/intfloat/multilingual-e5-large
(accessed 2025/12/10).

[21] WebArena: *WebArena: A Realistic Web Environment for Building Autonomous Agents*, https://webarena.dev/ (accessed 2025/12/18).

[22] Chae, H. et al.: *Web Agents with World Models: Learning and Leveraging Environment*

Dynamics in Web Navigation, arXiv:2410.13232,
https://arxiv.org/abs/2410.13232 (accessed 2025/12/18).