HALO - Holistic Agent Living Orchestration

Sofiya Flenova, Tim Frankel, Vinethmi Kulasinghe, Lakshaya Navaneetham

May 2026

Abstract

HALO is an open-source, privacy-first multi-agent system where autonomous software agents represent each occupant and each smart device, negotiating resources peer-to-peer with no cloud dependency and no central orchestrator. The system learns routines implicitly through passive observation, adapts in real time to ecological data (weather forecasts, grid carbon intensity, drought alerts), and recovers autonomously from device failures, all while keeping all sensitive data local and user-controlled.

1 Introduction

1.1 Background

The Smart Home is defined as "a residence equipped with computing and information technology which anticipates and responds to the needs of the occupants, working to promote their comfort, convenience, security and entertainment through the management of technology within the home and connection to the world beyond" [**Aldrich2003**] and extended to "add healthcare, education and communication" [**SamSolaimani2015**]

Smart homes currently are managed by either proprietary, vendor locked interfaces (Google Nest, Amazon Echo) or less locked down but still proprietary, local-supported systems such as Aeotech. Similarly, software based hubs like Samsung SmartThings exist on the lower end of the technical difficulty scale. These proprietary hubs are all easy-to-use but currently are not autonomous. They support features such as routines that can do pre-programmed tasks.

Open-source systems are much more difficult to set-up, although they have more extensibility and customisation options. Currently the smart-home industry has a double-edged sword problem: ease of setup vs customisability and autonomy.

The majority of these systems are also reactive, so they respond to voice or touchscreen commands, or specifically timed events. Systems like Amazon Echo have basic triggers, but it is up to the user to explicitly set these things up. There is no simple or easy interface to make the smart home "smart".

While there is no official measure of "home autonomy" currently, we can draw parallels from the US National Highway Traffic Safety Authority's levels of self-driving autonomy [**NationalHighway**], where different levels correspond to different amounts of human intervention. The current state of common home automation is akin to level 1 and in some cases level 2. Smart homes generally require manual input for doing tasks, with scheduling of preset tasks and basic hooks to trigger other tasks.

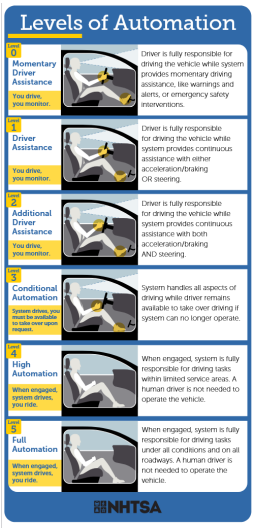

Figure 1: NHTSA Autonomy Levels

1.2 Problem statement

We propose to make an open-source, agentic standard for smart home governance. We have a mixture of multipurpose and specialised agents that govern individual devices, advocate for individuals and accumulate information. These agents negotiate resources in a peer-to-peer local network, with no core internet dependency and no central orchestrator. Agents learn routines through observations, negotiate with other agents and apply constraints to device governance based on externally collected data. The system is flexible and extensible, and most importantly is autonomous to a much greater degree.

This approach should achieve a level 3 or conditional automation approach.

1.3 Problem breakdown

This project has quite broad aims, so we break down the problem into the following key areas:

- Anticipation

- Negotiation
- Actuation
- Initialisation
- Extension

And the agents:

- Governor agents (device control)
- Advocator agents (user advocates)
- Aggregation agents (information collection and processing)

1.3.1 Anticipation

Anticipation can be defined as the "state of preparation" [**oxfordanticipation**] for an event. We believe that for a smart home to break across autonomy levels and progress, there must be a standard of preparedness. A smart home system should be able to know when events happen and schedule actuations accordingly. There also should be a functionality for adjusting anticipation in light of external factors and random variation. For instance, turning the heating on when a user is expected to return home from work/daily activities, turning off lights when everyone is out. But for instance, when there is a security system attached, if suspicious circumstances are detected, the system should be able to activate lights internally to simulated home occupancy and ward off the potential intruder. Similarly, given a virtual "shopping" device, the smart home system could anticipate the need to buy more milk, eggs or any arbitrary household goods based on learned patterns for usage.

1.3.2 Negotiation

Negotiation is the process of discussing to reach an agreement [**Pruitt1981**]. An agentic system needs an efficient, accountable system for making decisions across the network. The negotiation system should be aligned with the home users' concerns, so the sharing of concerns, rather than separation should be a key feature of the network. Advocating agents are strongly weighted to the needs of the users, while device agents have more weight to their own longevity, the environment and costs. We can easily model these negotiation problems as tractable optimal control or reinforcement learning problems.

1.3.3 Actuation

Actuation is the process of delivering physical effects, which is the embodied aspect of our AI system. Actuation/governing agents need to be aware of device states, failure modes and be able to attempt to operate nondestructive troubleshooting, or even call for maintenance support.

1.3.4 Initialisation

Initialisation is the process of setting up the system. For smart homes where there are many parts from different vendors, this is typically a time-intensive, technical process that involves many different setup processes and tools. There are however certain standards, such as Matter, that make devices easier to interoperate on an application layer, making setup easier. Thus we propose to embed Matter into our project to facilitate the development of the rest of the system.

1.3.5 Extension

Extension is important because we want a future-proof and up or downgradeable system. Users needs' change and hardware updates quickly. Giving plug and play support makes the system easier to adopt. Furthermore, with our proposed architecture, devices don't have to be embodied, and developers can make custom virtual plugins that can interface with websites. This can support more autonomous functionality, such as meal planning, shopping and even health monitoring*.

2 Current limitations and our approach

Current smart-home automation relies heavily on cloud-dependent central hubs. And while that means that most systems like Alexa Home, Google Home and others can be controlled from anywhere, kept up-to-date and operated without any understanding of networking, it also creates structural weaknesses that are rarely acknowledged.

By centralising the intelligence in the cloud, these systems introduce a single point of failure, they work with third party servers, sharing sensitive behavioural information about the user, as well as locking the user to proprietary ecosystems, where hardware functionality depends heavily on the vendor and its continued operation.

The real future of home automation is hybrid, combining local control for reliability and the cloud for heavy AI. HALO approaches this problem from another direction: it removes the assumption that intelligence should be centralised at all. Instead of splitting the responsibility between a local hub and remote server, HALO distributes intelligence across the environment itself. Every occupant and their devices in the household operate as an autonomous agent with local reasoning capability. The coordination is managed through peer-to-peer communication instead of a top-down orchestration. There is no cloud brain and no master controller. Heavy computation is not offloaded to a distant data centre but is shared across agents that each handle only the slice of reasoning relevant

to them. This shifts the design to a self-organising network of embodied decision making agents.

Moreover, existing solutions require manual rule configuration, which can be time consuming for the users, who expect the system to make their life easier, not take up more time. It operates through rigid logic that cannot adapt to unexpected situations, recover from device failures or optimise for real-world environmental constraints. Most systems do not meaningfully account for drought restrictions, heatwaves or sudden cold snaps. But, more importantly, the existing solutions cannot adapt to a household timetable, where multiple occupants with conflicting preferences must coordinate resources like temperature, shower timing, dishwasher queues and guest access, all without a central authority mediating every decision.

The system consists of three agent types:

- Person Agents represent each occupant, learning their preferences and routines through passive observation. These agents run locally on the occupant's phone or a shared edge device in the home.
- Device Agent represent smart devices such as thermostats, lights, dishwashers, showers, locks. Each agent understands its device's capabilities and exposes them to the network.
- Specialist Agents monitor external weather forecasts, grid carbon intensity, water authority alert, then broadcast observations that other agents can act upon.

The learning is implicit, no user ever sets their preferences or defines their schedule. Agents build models of normal behaviour using lightweight methods—moving averages, Bayesian inference and decision trees through statistical learning or preferences. The system adapts as routines change.

3 Proposed architecture

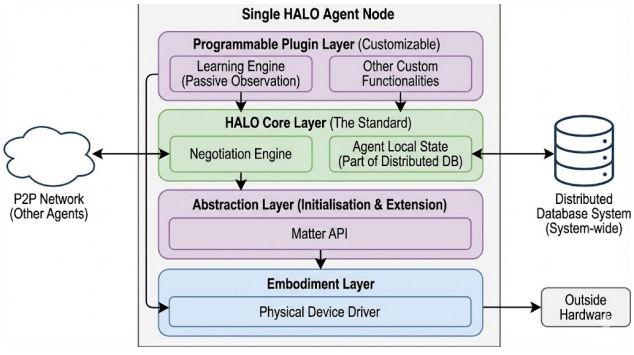

Figure 2: Single node stack

4 Simulation

4.1 Technical Specification

The current HALO proof-of-concept combines a discrete-event household simulation with a lightweight local web dashboard. The environment is designed to evaluate agent negotiation, device governance, specialist broadcasts and recovery behaviour under controlled and repeatable conditions without requiring physical hardware or a fully deployed peer-to-peer network. All processes operate in simulated time rather than wall-clock time, with one simulated minute as the base tick and `MINUTES_PER_DAY = 1440` defining a day.

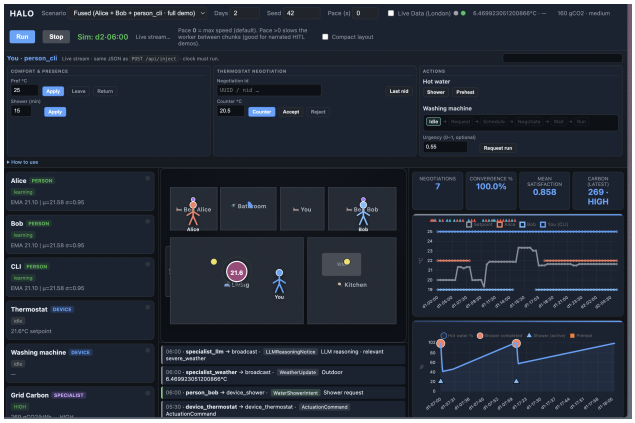

Figure 3: HALO simulation dashboard

Simulation Engine

The simulation is built on SimPy, where agents register asynchronous processes (generator coroutines that yield `env.timeout` and `Store.get/put` events) on a shared `simpy.Environment` driven by a global simulated clock. Each run is parameterised by a predefined scenario, a random seed used to construct an isolated random stream per scenario. A simulation horizon expressed in days, which ensures deterministic and reproducible execution. If run through the dashboard, the SimPy run executes inside a daemon worker thread separate from the HTTP server, and `env.run` is sliced into short two-simulated-minute chunks so that a client disconnect can stop the worker promptly and events can be flushed to the browser without buffering.

Agents and Messaging

The system models person agents, device agents (thermostat, dishwasher, shower, lights) and specialist agents providing contextual information such as weather and grid carbon intensity. Agents communicate through a local in-memory `MessageBus` using typed `Message` dataclasses, carrying a UUID, sender and recipient identifiers, message type, timestamped payload and priority, rather than shared global state, and each agent owns a simpy.Store inbox into which the bus enqueues deliveries with a small delay to

enforce deterministic ordering at the minute boundary. The protocol defines roughly two dozen message types, covering preference declarations, presence and sleep notices, the full negotiation lifecycle, carbon and weather updates, device failure and recovery, actuation commands, telemetry and the LLM-derived events described below. Device agents expose observable state transitions through periodic telemetry. Negotiation between persons and devices uses a weighted preference convergence rule of the form

$$p = \frac{\sum_i v_i w_i + v_d w_d^{\text{eff}}}{\sum_i w_i + w_d^{\text{eff}}},$$

where each occupant's stake w_i is scaled by presence (away contributors are masked out) and the device-side weight w_d^{eff} is boosted whenever grid carbon intensity exceeds 250 gCO_2/kWh. A round is considered converged when the variance of the latest proposals falls below a fixed threshold, and is capped at ten iterations before falling back to the unweighted mean. Device failures and recovery are simulated as a per-minute Bernoulli draw with a configurable failure probability, a 30-minute recovery timeout and up to three retries before a hard fault is recorded.

Implemented Scenario

The current demonstration scenario includes multiple occupants (Alice and Bob on scripted daily schedules and a human-controlled `person_cli`), thermostat control bounded by `THERMOSTAT_MIN/MAX` of 14–28 °C, dishwasher and shower subsystems. The shower is backed by a normalised hot-water-tank model with automatic preheat triggers with specialist agents broadcasting carbon every 30 simulated minutes and weather every 60, and scripted carbon-intensity spikes (a forced 17:00–20:00 peak at `CARBON_SPIKE_INTENSITY = 280` gCO_2/kWh) layered on top of a 24-hour UK baseline carbon profile, to demonstrate adaptive and carbon-aware behaviour.

Ecological Data Integration

Specialist agents can integrate live external data sources, including Open-Meteo weather feeds (`api.open-meteo.com/v1/forecast`, London coordinates 51.5074, −0.1278) and the UK National Grid carbon intensity APIs (`api.carbonintensity.org.uk/intensity`, with date-scoped and 24-hour forecast variants). Network access is gated by a `live_data` flag and mediated by an `ExternalDataClient`; external failures, non-200 responses and request timeouts are handled through cached or synthetic fallback values driven by the deterministic seasonal model in configuration, ensuring simulation continuity and resilience against unreliable third-party services.

Specialist Language Layer and External Data Integration

Simulation configurations include an LLM-based specialist agent (`LLMSpecialistAgent`) operating on the same SimPy time base as the rest of the household, scheduled by an internal `reasoning_interval` of 60 simulated minutes. At fixed intervals, the agent analyses a rolling window of the ten most recent non-self bus messages, each of which is summarised into a one-line natural-language entry annotated with simulated wall-clock time. It queries a configured language model endpoint to determine whether any external information source is relevant to the current household state. The LLM call is dispatched by `LLMClient` as a synchronous `httpx` POST to either the Anthropic `/v1/messages` or an OpenAI-shaped `/v1/chat/completions` endpoint suitable for LiteLLM-style proxies, with the default model `claude-haiku-4-5`. The model is instructed to return a strict JSON object {`relevant, api_ids, reason`}, naming up to four entries drawn from an explicit `ApiRegistry` of permitted sources, each carrying its own trigger conditions, keyword filters, cooldown window and target HALO message type. If an external source is selected, the specialist performs a bounded synchronous HTTP request with an eight-second timeout to retrieve the required data. Supported integrations include UK gov.uk weekly road-fuel-price publications, Open Food Facts product search, NewsAPI disruption monitoring and additional Open-Meteo severe-weather forecasting. Failures and timeouts are logged directly into the simulation event stream as structured records to preserve experiment traceability, and per-API cooldowns (60-240 simulated minutes) prevent repeated calls within a single household horizon. Retrieved responses are subsequently passed through a second language-model interpretation stage that converts the raw external data into structured HALO-compatible observations whose fields conform to a per-API schema. These observations are then broadcast onto the message bus like any other specialist event, allowing person and device agents to react independently of the originating data source. For example, cost-pressure messages transiently raise the thermostat's device-side negotiation weight, weather-forecast alerts temporarily clamp the comfort negotiation range, and disruption events extend simulated return-home delays or slightly reduce occupant comfort weights.

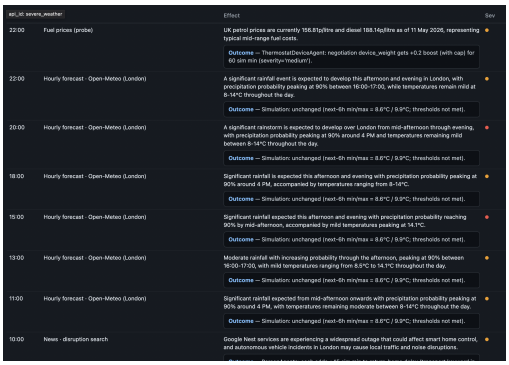

Figure 4: The LLM inspector view of the dashboard, showing reasoning cycles, external API calls and the resulting bus observations for a running simulation

Human-in-the-loop Operation

Validated JSON commands can be injected directly into the simulation message bus through a `POST /api/inject` endpoint, allowing an experimenter to modify occupant preferences, presence states, dishwasher and shower requests, and negotiation responses (counter, accept, reject) during runtime while the remainder of the environment continues autonomously. Each request is parsed by a strict `validate_queue_item` schema and enqueued on a `queue.Queue` that is the only thread-safe bridge between the HTTP request thread and the SimPy worker; a dedicated `BridgeInjector` SimPy process polls this queue every 0.05 simulated minutes and posts the corresponding messages to the bus as `person_cli`. Because the inject queue and stream pointer live on the FastAPI `app.state`, the server is run as a single Uvicorn worker, and the `CliPersonAgent` is configured with manual scheduling and manual negotiation so that no autonomous clock advances on its behalf between human actions.

Dashboard and Metrics

A FastAPI server provides a live dashboard that streams events from the simulation in real time using Server Sent Events (SSE). Through the dashboard, users can observe negotiations between agents, device activity, telemetry updates and changes in household state as they happen. The browser connects to the simulation through a single streaming endpoint, where the scenario configuration, simulation length, random seed and optional wall-clock slowdown can be adjusted. This slowdown feature is particularly useful for demonstrations, as it stretches a simulated day across a longer real-world time period so human observers can follow the interactions more easily. Different categories of events are streamed separately, including negotiation messages, device telemetry, agent state updates and the outputs of the LLM specialist. This allows the front-end to visualise each type of activity independently using live feeds, charts and a dedicated inspector for the LLM reasoning pipeline, external API calls and failure events. Alongside the live interface, the system also records internal metrics such as negotiation convergence time, final negotiated outcomes, occupant satisfaction estimates, device failures, recovery behaviour and LLM interactions. These results are exported as CSV files and visual summaries for later analysis of coordination behaviour and emergent system dynamics.

5 Network

For our smart home system, the network is the backbone of our system. A good network design minimizes downtime, reduces message passing latency and ultimately, allows for proper functioning of our smart home system.

5.1 Network Startup

In a HALO world, every home and workplace is on its own isolated HALO network. Agents on the network should know which network it is on, and not try and talk to a neighbouring network.
Every HALO network is started by a special Bootstrap agent. This agent is ephemeral and exists solely to create a new network, and set an advocator agent (home occupant) as an admin. It is then promptly destroyed.
The bootstrap agent is started on a device with a visible screen. It generates a network UUID, and a public-private encryption keypair. It then creates a QR code from the public key, which the mobile app can scan. Once scanned, the advocator agent is now sent the network UUID, and the bootstrap agent is destroyed. Any agent connecting to the admin is now added onto the network, and is given the UUID.
Agents on different networks have multiple UUIDs, but each keypair connection between agents is generated via a UUID, so one keypair from one network will be completely unreadable from another network.
This UUID will also be used for the fully local provenance chain.
This covers the network part of the initialisation problem mentioned in our 5 pillars of HALO.

5.2 Network Discovery

Network discovery is dealt with via the multicast Domain Name Server protocol [**mDNS**], which sends discovery messages to all devices listening on the network, with an identity and address. It is then at the discretion of the individual peers to connect or not connect.

This approach was chosen as it is fully dynamic, meaning there is no need for hardcoded IP addresses, which allows our system to have parts added and subtracted dynamically.

The major limitation is for running in containers on Windows, as the Windows Subsystem for Linux virtual networking driver actively blocks multicast packets for security reasons, and workarounds are complicated to

setup. This limits the network on Windows containers to only be able to discover other local agents, however this doesn't pose such a huge limitation as the benefit of separate agents is more of a logical desire than a physical need.

5.3 Message Schema

The HALO message schema is designed to be simple, with a data-driven design. This fits in very well with the actor/agent model being used.

Messages are dictionaries with the following fields:

- action
- source
- target
- params
- on-success
- on-failure

The on-success and on-failure fields contain nested maps of the same schema. This allows for nested and chained logic, and is easy for both humans and LLMs to understand the flow of data.

5.4 Agent Lifecycle

Agents are expected to register the functions they can do in their init function. The handler registry is a dictionary that contains LLM readable names that are mapped to their respective class methods.

Behind the scenes these are shared at a fixed period with the network, so other agents can be aware of the agent's capabilities.

Multiple threads are instantiated after this to do the following: network discovery, heartbeat, network pruning, message listening, state backup and function discovery. These functions have their own threads and execute regularly, sending messages to the network to see if there are new agents to connect to, tell the network the agent is alive, remove "dead" agents, and propagate their functions to agents that might have to call them.

5.5 Event Loops

HALO agents have less traditional event loops. Agents run multiple background processes by default, and the user can specify to write active foreground event loops for any non-event based logic. As for receiving messages from the network, a listening function is constantly waiting for messages. When a message is received, it is consumed and routed, either for direct execution, delayed execution (with a specified time to sleep the thread) or dispatched to an execution queue, which is saved for scheduled events. The "action" field is used to look up the specific method to be called. After an action occurs, its on-success or on-failure actions are then evaluated. The result of the previous action can be injected in, if a wildcard is specified in the parameters list of the on-success or on-failure action. These following actions are typically messages to other parts of the network, but can also be to the same agent.

With this execution pipeline, agent authors don't have to worry about interpreting custom messages or writing their own executors or chained logic. They can simply write functions that return values that can be used by the rest of the network. If the author wants to write a continuously running function, such as checking for price updates or observation of a security camera, they can write a function that runs on its own thread and is executed in the main function of a program. At any point where an event needs to be propogated to the network, their function simply needs to call a "send_msg" function to send events that can be consumed by other agents.

With this combination of execution pipelines, any functionality can be emulated in the form of a HALO agent.

5.6 State management

Agents keep their own relevant state and can share when the specific keys are requested. To discover the keys belonging to an agent, any other agent can send a "discover state" message. Furthermore, the language agent injects its own state to the prompt, such as the current time and location. This dynamic fetching of state depending on queries allows for a much slimmer networked memory, without the issues of prolonged LLM context windows. The LLM is essentially contextless compared to a traditional chat app. There are disadvantages, which may be addressed by a short term cache of chats based on sentiment and relevance. Furthermore this memory relies on the network, so if a node goes down, portions of memory are inaccessible, and it can be slower to retrieve than traditional direct-in-memory databases.

5.7 Specialist Agents

Some important specialist agents in HALO's core provide a seamless smart home experience:

5.7.1 Language/Telegram Agent

The language agent is the smart agent on the network. We chose to connect it to a Telegram frontend, as it has an accessible and easy-to-use bot API.

The Language agent forwards Telegram messages to the LLM, along with a general system prompt, the current network state and connected devices. The LLM decides how to act and orchestrate multi-staged tasks in order to assist the home occupants. It is able to prompt itself, as well as send messages to the user and it uses this ability to format messages nicely, and to

break up the context window to allow for more directed responses with less hallucinations.

5.7.2 Watchdog Agent

The watchdog agent is our networks' self healing system. It keeps track of the docker containers that run the agents on the network, and when it hasn't received heartbeats from an agent for a certain timeout time, it talks to the docker daemon directly and revives the container, starting a new version of the agent process.

5.7.3 Bootstrap Agent

The bootstrap agent is in charge of starting up a HALO network securely. It is not yet implemented fully but a necessary part of making the system production ready.

5.8 Emergent Behaviour

A major part of this project involves the exploration of emergent behaviour from such an agentic, distributed system. To differentiate itself from other, existing home automation projects, we want to offload much of the action chaining, thought and scheduling to the smart "brains", whether they be the RL learning policies or an LLM action orchestrator. HALO is a proactive, thinking system, thus we want to observe and guide its behaviour rather than hardcode them.

Behaviours noted: the use of self prompting for prettifying displayed data, retrieval of state to inform commands in 40% of cases, and filtering long search results for ease of use. None of these behaviours were explicitly programmed in, although some will be mentioned in the system prompt. See appendix: HALO messages

6 Local LLMs

A range of local LLMs were tested briefly for initial selection as the main HALO model. Models were qualitatively compared based on how relevantly they responded to messages and commands and the rate of halluctination. It was found that all 2-4bn parameter models (Phi3.5, Phi4 Mini, Gemma 4B, Gemma 2B, Llama 3.2) were unable to even complete single tasks without hallucinating. The best balance of performance was Qwen 2.5 and Llama 3.1, both 7 and 8 billion parameter models that could, with significant hand holding, chain multiple commands together for execution. However at the current stage, the local models are not ready for HALO, we propose to write an automated multi-shot prompt finetuner using an actor-critic model, in order to improve the performance of these local models to that of a larger, hosted one.

7 HALO Ecosystem

In the adoption of any software engineering project, the ecosystem can make or break the viability of our project. Furthermore, it is important we have a range of available agents to test the network with, incorporating real physical devices as well as external, 3rd party APIs to make useful integrations. The following integrations were successfully developed: Miele smart washer-dryer, Hive smart thermostat, Android smart TV, Tavily search engine, OpenWeather API, Telegram messaging, Netflix, Disney+, Luna games and Spotify API, generic HTTP doorbell security camera with suspicious person detection. The following devices we fabricated in a miniature, mock hardware house and created HALO agents for: "smart" LED light, washing machine, TV, automatic blinds.

The combination of these agents allowed us to test a network with 10+ concurrently connected agents and observe successful utilisation of many in one "context" session.

8 Habit Learning

The habit learning subsystem implements the anticipation pillar of HALO introduced in Section 1.3.1. Its purpose is to transform passive behavioural observations into a probabilistic representation of occupant routines and use this representation to proactively recommend actions before they are explicitly requested. Rather than relying on manually programmed schedules, the subsystem continuously learns temporal patterns from observed activity and adapts its behaviour over time.

Reinforcement learning was selected instead of static rule-based automation because household behaviour is uncertain and context dependent. The policy must balance multiple competing contextual signals such as weather, occupancy probability and activity timing while avoiding repetitive or unnecessary actions.

The full pipeline consists of four stages: synthetic behavioural data generation, probabilistic profile extraction, reinforcement learning policy training, and incremental online adaptation. The trained policy is then integrated into the wider HALO multi-agent architecture through the shared message bus.

8.1 Synthetic Behavioural Dataset and Behavioural Profiling

A major challenge during development was the absence of long-term real-world behavioural data and the lack of deployed smart-home hardware capable of collecting it. To address this limitation, a synthetic data generator was implemented to simulate six months of realistic household activity for a single occupant.

Each generated user profile contains:

- Timestamp and day of week

- Season and weather conditions
- Occupancy status
- Event type
- Heating preference
- Contextual recurring events such as: dog walks, grocery trips, dance classes and family movie nights

The generator was designed to model human behaviour with random noise added to event timings to prevent models from overfitting to exact schedules.
Cross-day dependencies are also modelled through a latent fatigue carry-over variable. For example, a late Friday movie night probabilistically delays wake-up times on Saturday morning. Similarly, weather-dependent behaviour is incorporated by increasing the probability of cancelling outdoor activities (dog walks) during heavy rainfall and delaying grocery trips by several hours.

The raw event log is then aggregated into a deterministic behavioural profile containing summary statistics such as:

- Typical heating temperatures by season
- Mean leave and return times
- Frequency counts for recurring activities

To preserve temporal uncertainty, each recurring activity is modelled using a one-dimensional Kernel Density Estimate (KDE) over a 24-hour cycle. This allows the system to model uncertainty in occupant routines rather than relying on fixed schedule estimates.

8.2 RL Environment

The probabilistic profile is embedded inside a custom Gymnasium environment, which formulates anticipation as a sequential decision-making problem.
The environment has discrete actions:

- Reduce heating when out
- Preheat before return home
- Recommend movie selection
- Set movie settings (dim lights)
- Suggest alternative walking time (rain)
- Check fridge and suggest shopping list prompt
- No action needed

The RL model is trained on the occupant behaviour profile json file and outputs the action, which is best suited for a certain scenario.

8.3 Reward Engineering

Each action is associated with a hand-designed reward function that scales according to the relevant profile-mass probabilities. Early experiments revealed that the agent repeatedly carried out certain actions such as preheat house action throughout the morning because the reward remained positive during winter regardless of recent usage.

Two mechanisms were introduced to eliminated exploitative repetitive behaviour:
Stateful cooldown periods:
Actions such as heating activation were assigned cooldown windows of six to eight hours. Reissuing an action during the cooldown incurs a substantial negative reward.
Contextual eligibility constraints:
Positive heating rewards are only granted when the predicted return-home time falls within approximately 1.5 hours.

8.4 Integration with HALO Multi-Agent System

The trained RL policy is integrated into the wider HALO ecosystem through a dedicated RL agent operating on the same peer-to-peer message bus as the other system agents. Every fifteen minutes, the RL agent collects the current environmental and behavioural state, constructs the observation vector, queries the PPO policy and then sends a typed recommendation message onto the network. The message is first routed to the language agent, which converts the machine-readable intent into a contextualised human-readable suggestion. The relevant person agent then delivers the recommendation to the occupant through Telegram.
Real-world routines inevitably drift over time. However, retraining from scratch on low-power hardware would be computationally expensive. HALO therefore performs lightweight monthly incremental fine-tuning, allowing the anticipation policy to gradually adapt to long-term behavioural changes.

8.5 Security

One of the key security features implemented in the HALO network system is the suspicious person detection system, which utilises external surveillance devices such as CCTV cameras and Ring doorbells to monitor activity outside the house in real time. The system continuously analyses video footage using a simple detection algorithm to identify suspicious people. When a suspicious person is detected, HALO immediately sends a warning notification to the homeowner via telegram. The alert would include a short video clip or captured footage of the detected individual.
In addition, HALO incorporates an automated deterrence mechanism. If the system determines that the homeowner is not currently at home and the house lights are switched off, it automatically activates the indoor lights. This creates the illusion of an active household, which can scare away potential burglars before any intrusion occurs. This feature enhances home

security by combining real-time surveillance, instant user alerts, and smart home automation to provide proactive protection against possible threats.

8.5.1 Suspicious Person Detection Algorithm

Burglars often attempt to conceal their identity by covering parts of their faces with items such as sunglasses, hats, or masks, making it more difficult for surveillance cameras to accurately recognise them. Taking this into consideration, the HALO detection algorithm was designed to identify suspicious individuals by analysing facial visibility.

The algorithm monitors four key facial features: the right eye, left eye, nose, and mouth. If two or more of these features are obscured for longer than three seconds, the individual is classified as suspicious and the system triggers a security alert. To determine whether these facial features are covered, the system first maps the approximate centre positions of each feature based on standard facial proportions. After locating these centre points, bounding boxes are drawn around the expected feature locations. The algorithm then checks whether the corresponding facial feature is detected within each box. If a feature is successfully detected, the box is highlighted in green; otherwise, it is highlighted in red to indicate that the feature may be obscured or missing.

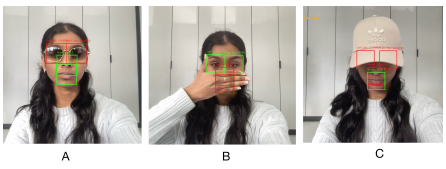

Figure 5: Suspicion detection results

Figure 5 demonstrates the operation of the suspicious person detection algorithm. Image A shows an individual wearing sunglasses, resulting in both eyes being obscured. Image B represents a person wearing a mask, which covers the nose and mouth. Image C shows an individual wearing a hat that obscures the eyes and partially covers the nose. In each of these cases, two or more facial features are hidden for an extended period of time, causing the system to classify the individual as suspicious. As a result, HALO would automatically notify the homeowner and provide the relevant surveillance footage.

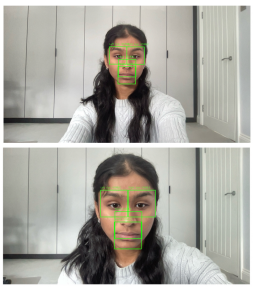

Figure 6: Dynamic box sizes

The algorithm is also designed to dynamically adjust the size of the bounding boxes around each facial feature based on the person's distance from the camera. As a person moves closer to the camera and their face appears larger within the frame, the bounding boxes automatically increase in size accordingly. Figure 6 illustrates this behaviour, where the bounding boxes in the bottom image are noticeably larger because the individual is positioned closer to the camera.

This dynamic resizing mechanism is essential for maintaining accurate facial feature detection across varying distances. By adapting the box dimensions in real time, the system remains flexible and robust under different distances from the camera.

9 Hardware Integration

A central claim of HALO is that intelligence is distributed across embodied device agents rather than centralised in a cloud hub. To showcase this, a hardware subsystem was developed to validate the HALO architecture on real, networked physical devices at small scale.

The objective of the hardware work was to demonstrate that the HALO protocol, peer-to-peer discovery, and the agent lifecycle operate correctly when the actuating endpoint is a real microcontroller on a real network rather than a simulated process. Each physical device is a low-cost mock of a real appliance, some of which are shown below in the demo pictures.

Figure 7: Mock living room with TV

Figure 8: Mock Smart Blinds

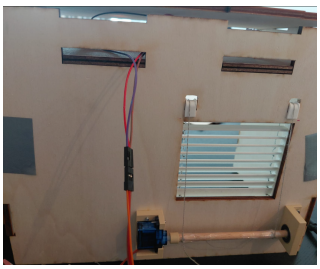

Figure 9: Smart Blind Mechanism

Figure 10: Kitchen including smart washing machine

This section describes the devices implemented, the bridging architecture that connects constrained microcontrollers to the HALO mesh, the security model used on the physical link, and the practical constraints encountered during deployment.

9.1 Devices

The following mock devices were implemented, most on its own ESP32-S3, while the 'smart lights' were spread over all of them due to one being in every room.

- Mock Television - an ESP32-S3 driving an ILI9341 TFT display, reading pre-rendered frame data from an SD card to show a television home screen, a streaming-service intro animation and a streaming home screen.
- Washing Machine - an ESP32-S3 driving a continuous-rotation servo to represent a rotating drum, with a time-scaled wash cycle and a completion notification pushed back to the user.
- Window Blinds - an ESP32-S3 driving a servo to tilt blind slats between an open and a closed position.
- Smart Lights - an ESP32-S3 driving a 5 mm LED through a PWM-controlled GPIO, supporting on, off and proportional brightness. The same firmware and agent are reused for multiple room lights, distinguished only by configuration.

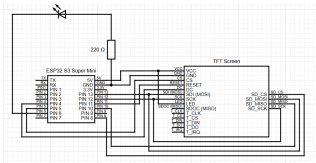

Figure 11: Mock TV Circuit Diagram

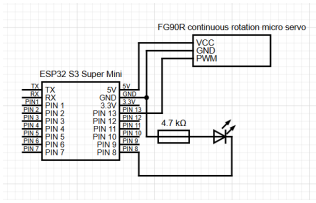

Figure 12: Washing Machine and Blinds Circuit Diagram

A combined firmware variant was also produced in which a single ESP32-S3 acts simultaneously as the television and the living-room light, used where physical microcontroller stock was limited. This is discussed in Section 9.6.

9.2 Bridging Architecture

HALO agents communicate over an encrypted ZeroMQ transport, discover one another through multicast DNS, and exchange the dictionary-based message schema. A constrained microcontroller such as the ESP32 cannot reasonably participate in this transport directly: the CurveZMQ, the Python agent runtime and the discovery machinery are not practical to reimplement on the device, and doing so would couple the firmware tightly to internal protocol changes.

The solution that was adopted preserves the HALO architecture unmodified by introducing, for every physical device, a dedicated lightweight Python agent that runs on the host alongside all other agents. This proxy agent is a full HALO agent: inherits from the standard agent base class, registers handlers, and can be discovered over the mDNS.

The proxy agent's sole additional responsibility is translation. When it receives a HALO action message, it converts that action into a small HTTP request and issues it to the corresponding ESP32 over the local network.
The microcontroller therefore never participates in HALO discovery, encryption or message schema. From the perspective of the rest of the network, the proxy agent is the device; the HTTP hop behind it is

an implementation detail.

This design has some properties that were important to the project, such as no protocol modification, meaning neither the discovery layer nor the agent base class required changes to support hardware. Physical devices are indistinguishable from software agents at the protocol level. Additionally, because each device has it's own proxy agent in its own process, the watchdog and restart supervision apply uniformly to hardware devices.

Each proxy agent registers semantically named handlers together with a natural-language description. These descriptions are propagated to the Language Agent through the standard capability-sharing mechanism, allowing the LLM to map free-form occupant requests onto the correct device action without any device-specific prompt engineering. This shows the previously discussed emergent chaining behaviour exercised on physical hardware.

9.3 Security

Although the ESP32 does not participate in the encrypted HALO transport, the HTTP link between a proxy agent and its device is still authenticated, so that an arbitrary host on the same local network cannot actuate a device by issuing unsigned requests.

Authentication uses HMAC-SHA256 request signing with a single shared 32-byte key provisioned to every proxy agent and flashed into every device. Each POST /command carries two headers: a Unix timestamp and a hex-encoded HMAC computed over the concatenation of the timestamp and the request body. On receipt the device recomputes the HMAC and rejects the request if the signature does not match, if the device's clock is not yet synchronised, or if the timestamp lies outside a fixed freshness window. The freshness window provides replay resistance: a captured request is only valid for a short interval. The timestamp check requires the device to hold accurate wall-clock time. Because the ESP32 has no battery-backed real-time clock, each device synchronises its clock over NTP at boot before the HTTP server is started. This introduces an external dependency that has practical consequences for network selection, discussed below.

9.4 Deployment Constraints

Several practical constraints were encountered when moving from simulation to physical deployment. They are documented here because they materially affect reproducibility.

- Network discovery on Windows - the Windows Subsystem for Linux virtual network driver blocks multicast traffic, which constrains mDNS-based discovery when the stack is run in containers on Windows. Agents remain able to discover other local agents, which is sufficient for the logical separation HALO requires, but this restricts cross-host discovery in that environment.
- Address volatility - The proxy agent reaches its device using the device's network address. Because the microcontrollers obtain addresses by DHCP, an address assigned on one network is not valid on another, and a device may receive a different address after a network change or a reboot. Each network transition therefore requires the device addresses to be reconfirmed. A static address configured in firmware removes this volatility and is the recommended configuration for a fixed deployment.
- Network suitability - The clock-synchronisation requirement means each device requires genuine network access at boot. Captive-portal guest networks, on which a browser-based acceptance step is mandatory before traffic is permitted, are unusable by the microcontrollers because the device cannot satisfy the portal. Networks that isolate clients from one another likewise prevent the proxy agent from reaching the device. A conventional WPA2 network, or a mobile hotspot configured on the 2.4 GHz band, was found to be the reliable choice; the ESP32-S3 does not operate on 5 GHz.
- Power -Servo-driven devices draw current well in excess of what the microcontroller's regulated output can supply, particularly under mechanical load. Servos were therefore powered from a dedicated 5 V supply with a common ground shared with the microcontroller, rather than from the device's own logic supply, to avoid brown-out and consequent loss of network connectivity mid-operation.
- Continuous versus positional servos - Continuous-rotation and positional servos respond differently to the same control signal: a positional servo holds a commanded angle, whereas a continuous-rotation servo interprets the same command as a rotational speed. The washing-machine drum is naturally suited to a continuous-rotation servo driven for a scaled duration, whereas the blinds require a positional servo so that the open and closed states correspond to stable, repeatable physical positions. Device firmware is therefore matched to the servo class rather than being interchangeable.

9.5 Single-Microcontroller Composition

To accommodate limited microcontroller stock, a combined firmware was produced in which one ESP32-S3

simultaneously drives the television display and the living-room LED. A single proxy agent registers the union of both devices' handlers and dispatches `TV_*` commands to the display and `LIGHT_*` commands to the LED on the same device.

Because the screen and the light are then the same agent, the "dim the light when streaming starts" automation reduces to an internal state change rather than an inter-agent message, which is both simpler and more robust as it removes a network round trip and a failure mode.

This composition is offered as a deployment option rather than the default; the one-agent-per-device model remains the architecturally preferred arrangement, as it keeps device identities, capability schemas and failure boundaries distinct, consistent with the principles in Section 5.

9.6 Summary

The hardware subsystem demonstrates that the HALO architecture extends to physical devices without modification to the protocol, discovery layer or agent model. By representing each microcontroller with a dedicated proxy agent that bridges the HALO mesh to an authenticated HTTP link, constrained devices participate in discovery, capability exposure, chained actuation and inter-device automation as first-class network citizens. The implementation validated, on real hardware, the same emergent chaining and automation behaviour previously demonstrated only in simulation, while surfacing the practical networking, power and actuator constraints that a physical HALO deployment must account for.

9.7 Simulation

The simulation demonstrated stable negotiation behaviour between occupant and device agents across multiple scenarios. Conflicting occupant temperature preferences consistently converged toward compromise setpoints within the bounded negotiation rounds defined by the protocol. During simulated high-carbon periods, device agents exerted greater influence on negotiations, resulting in lower final heating setpoints and delayed dishwasher cycles. These behaviours demonstrated that environmental constraints and device-side priorities could meaningfully influence household decisions without requiring a central controller. Human-in-the-loop negotiation through the dashboard and CLI interface also remained compatible with the negotiation protocol, allowing live user intervention without destabilising the wider system. The specialist-agent subsystem successfully incorporated ecological context into household decision-making. Weather updates, carbon-intensity spikes and external cost-pressure events altered negotiation weighting and device behaviour dynamically throughout the simulation. In particular, simulated evening carbon spikes caused thermostat and dishwasher agents to favour energy-saving outcomes, demonstrating adaptive environmentally-aware coordination behaviour rather than static rule execution. The simulation also demonstrated successful human participation within the multi-agent environment. Users were able to inject negotiation actions, preference changes and occupancy events into the running simulation through both the dashboard and CLI interfaces. The remaining autonomous agents continued operating normally during these interventions, allowing the human participant to function as another peer agent rather than as an external controller.

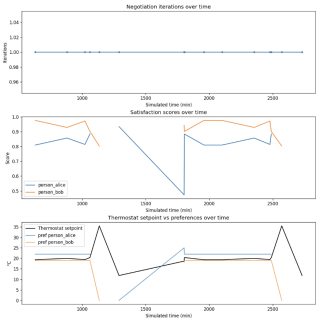

Figure 13: Negotiation outcome over a multi-day simulated run

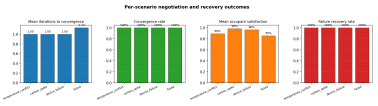

Figure 14: Per-scenario aggregates over the four implemented scenarios. Every thermostat negotiation converged within the protocol's iteration cap, with a mean of a single iteration in three of the four scenarios and 1.13 iterations. Mean occupant satisfaction ranged from 0.85 to 0.98, and every induced device failure recovered within the configured retry cap.

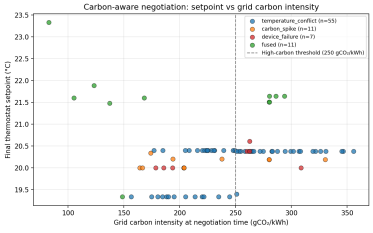

Figure 15: Final thermostat setpoint against grid carbon intensity at the moment of negotiation, across all four scenarios. The dashed line marks the 250 gCO_2/kWh threshold above which the device-side weight is boosted in the convergence formula; rounds on the right of this threshold cluster more tightly around the device's preferred operating point, while low-carbon rounds spread more widely across occupant preferences.

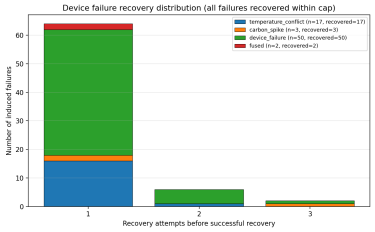

Figure 16: Distribution of recovery attempts per induced device failure across the four scenarios. Of the 72 random failures injected, most recovered on the first attempt and none escalated to a hard fault, confirming a resilient recovery profile.

This demonstrates that HALO can resolve conflicting occupant and device priorities without centralised control.

9.8 Habit Learning

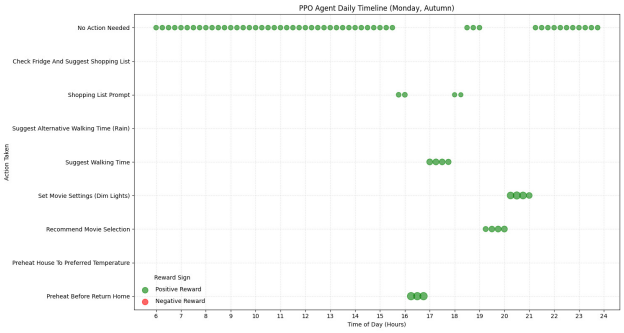

Figure 17: Timeline of PPO policy

Figure 17 illustrates the behaviour of the trained PPO anticipation agent over a simulated weekday during autumn conditions. The horizontal axis represents the simulated time of day, while the vertical axis shows the action selected by the policy at each decision interval. Green markers indicate positive reward outcomes, meaning that the selected action aligned well with the occupant's learned behavioural profile and environmental context.

During most low-context periods, the agent selects the "No action needed" state, demonstrating that the policy avoids unnecessary interventions rather than continuously triggering automation events. This behaviour became significantly more stable after introducing contextual eligibility constraints and action cooldown periods during reward engineering.

As the simulated occupant approaches their expected return-home window in the late afternoon, the policy outputs action "Preheat before return home". This demonstrates that the reinforcement learning agent successfully learned temporal occupancy patterns from the probabilistic behavioural profile. Around the early evening period, the policy begins recommending contextual lifestyle actions such as "Suggest walking time", followed later by entertainment-related actions including "Recommend movie selection" and "Set movie settings (dim lights)". These recommendations correspond to recurring behaviours embedded in the synthetic training data, such as evening dog walks and family movie nights.

Overall, the timeline illustrates that the habit-learning subsystem can generate proactive, context-aware recommendations while avoiding excessive or repetitive automation behaviour, supporting HALO's anticipation-oriented autonomous smart-home architecture.

9.9 Demonstration

We demonstrated the final project to the Cisco team, where we showed them both our hardware and simulation models and explained reasoning behind the methodologies used. The demonstration focused on showcasing HALO's core principles of distributed negotiation, anticipation and autonomous coordination between agents. During the session, we demonstrated real-time negotiations between occupants and devices within the simulation dashboard, the operation of specialist agents using ecological data, reinforcement-learning based anticipation behaviour and the interaction between the language agent and external integrations such as Telegram. The simulation component performed reliably throughout the demonstration, so did the anticipation and initialisation pillars. However, we did experience several difficulties with connectivity and integration between physical devices due to a new location.

10 Discussion and Evaluation

The HALO project successfully demonstrated the feasibility of a distributed multi-agent smart-home system capable of negotiation, anticipation and context-aware adaptation. However, the project also revealed several limitations. Due to the broad scope of the system, significant development time was spent refining individual subsystems in depth rather than integrating components. The reinforcement learning subsystem was trained entirely on synthetic behavioural data, meaning the learned policies cannot yet fully represent real-world household behaviour. Additionally, while the use of LLM-based specialist agents enabled flexible reasoning and orchestration, model outputs occasionally remained unpredictable and required bounded controls to prevent unreliable behaviour. Finally, emergent behaviours observed within HALO were evaluated mainly qualitatively rather than through formal quantitative metrics.

11 Conclusion and Future Work

Overall, the project successfully demonstrated the feasibility of a distributed multi-agent smart-home architecture capable of negotiation, anticipation and con-

textual adaptation. While several components remain experimental and require tighter integration into a single cohesive system, the project successfully validated the core architectural principles behind HALO and provided a strong foundation for future research into autonomous domestic systems.

12 Bibliography

A Source Code

`https://github.com/vision-05/HALO`

B HALO Messages

Figure 19: Response to food recommendation - failed before sending result

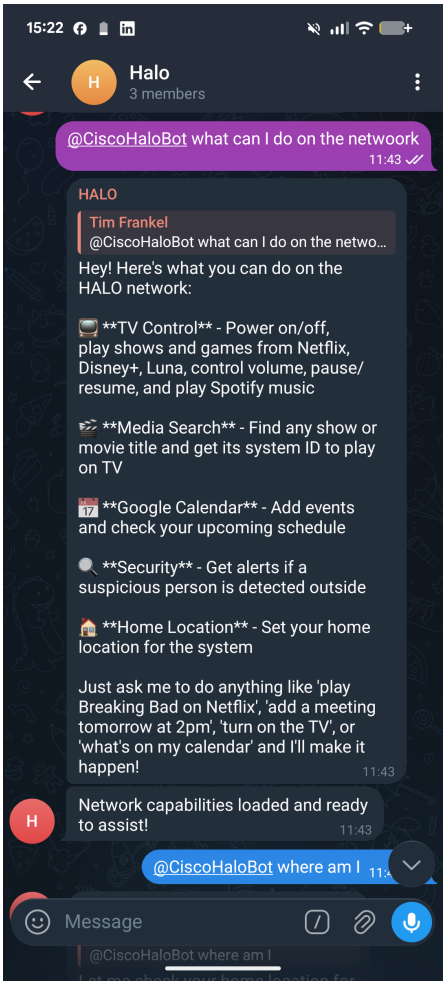

Figure 18: Requesting HALO its functionality

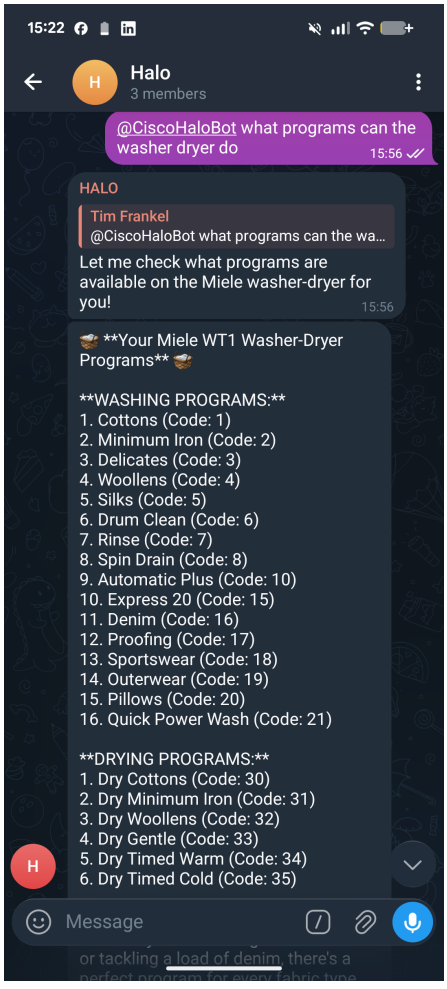

Figure 20: Requesting washing machine capabilities

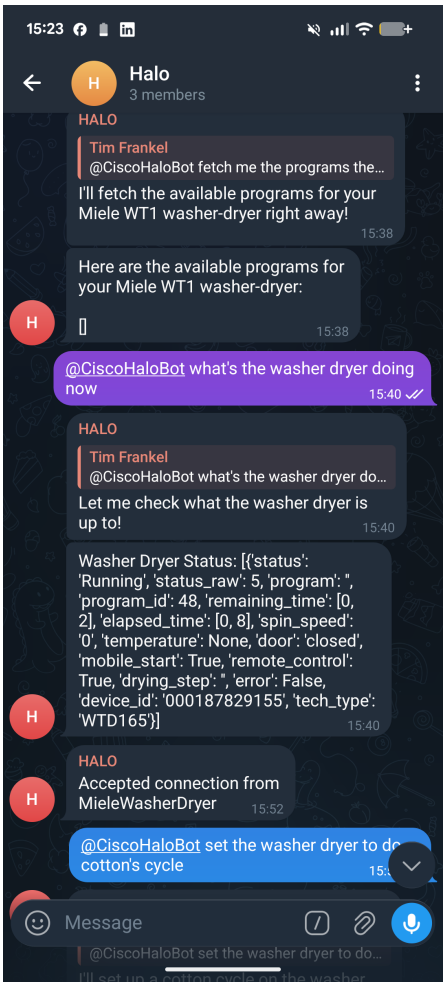

Figure 21: Requesting washing machine status - bad formatting

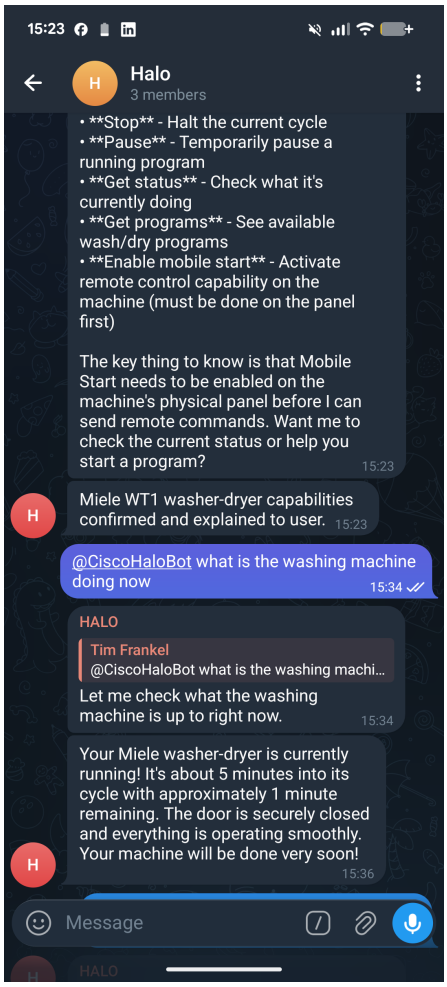

Figure 22: Requesting washing machine status - good formatting

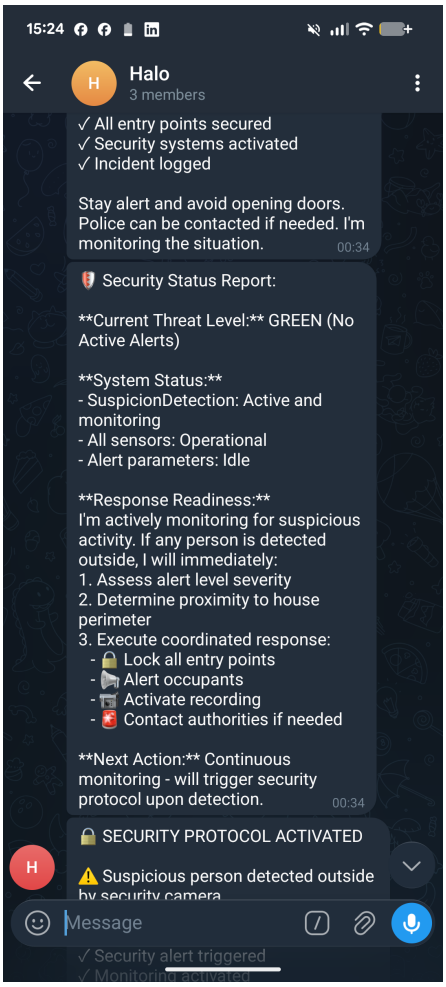

Figure 23: Security notification - no suspicious people outside

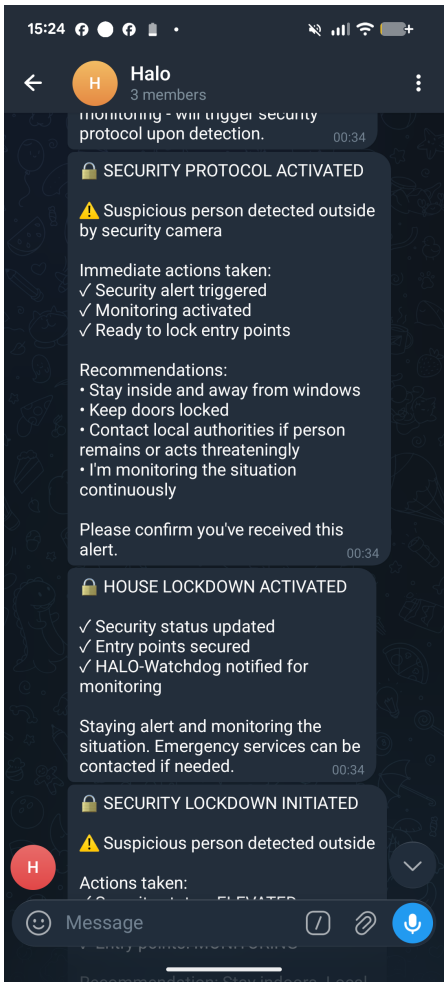

Figure 24: Security notification - suspicious person detected by camera

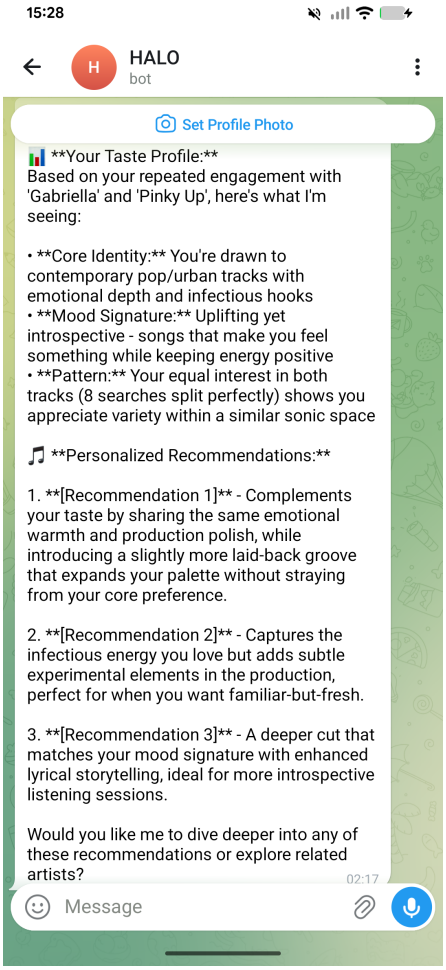

Figure 25: Music profile based on listen history

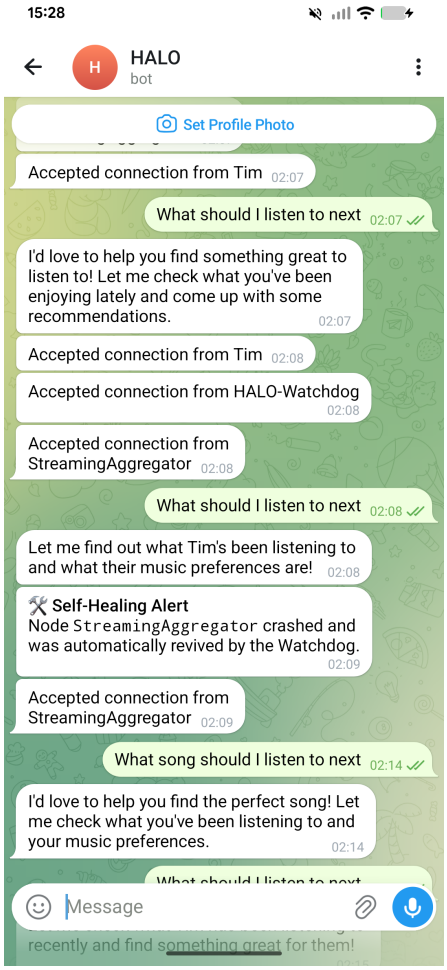

Figure 26: HALO watchdog auto-recovery

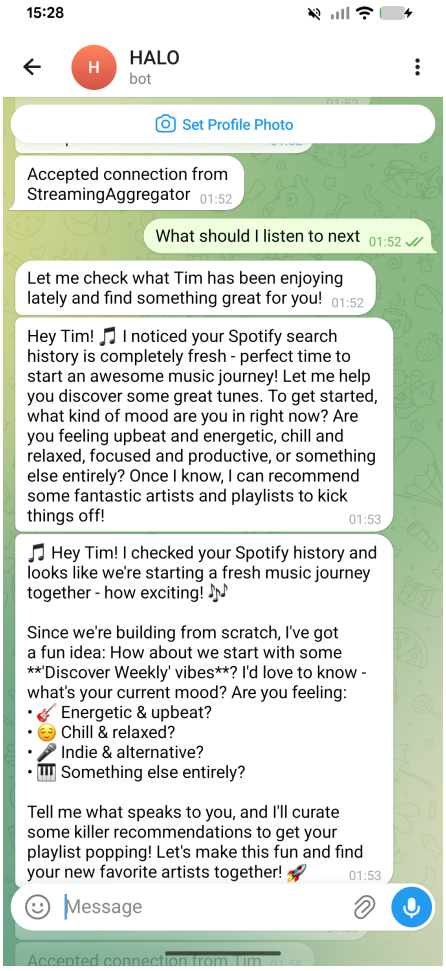

Figure 27: Music recommendation by listening profile

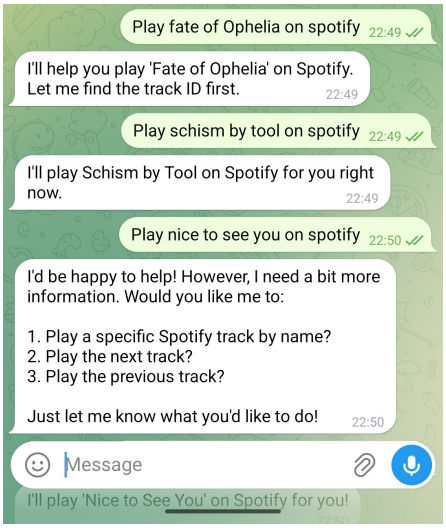

Figure 28: Spotify media request - successful

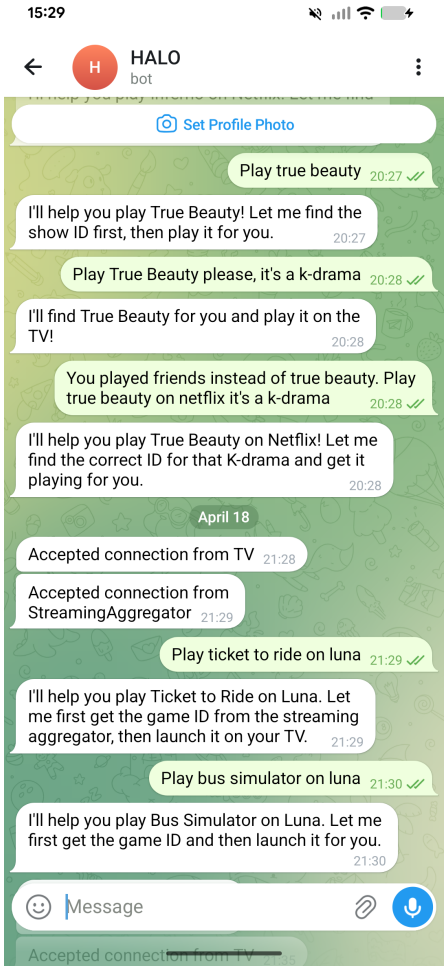

Figure 29: General media requests - successful