

WALL-E

A Local-First, Embodied AI Companion Robot

CSCI 409 Senior Project II
Final Project Report, Spring 2026

Group Members

Adilet Shildebayev
Miras Baisbay
Dauletkhan Assanov
Yerzhan Temirali
Edige Akimali

Adviser

Siamac Fazli

Nazarbayev University

April 2026

Contents

1	Executive Summary (§1)	3
2	Introduction (§2)	3
2.1	Problem, motivation, and significance	3
2.2	Proposed solution and report organization	4
3	Identification of Constraints (§3)	5
3.1	Economic	5
3.2	Environmental and sustainability	5
3.3	Social and ethical	6
3.4	Manufacturability	6
3.5	Impact of constraints on design	6
4	Background and Related Work (§4)	6
4.1	The problem WALL-E’s LLM side has to solve	6
4.2	Prior work	7
4.3	Alternatives considered and rejected	7
4.4	Justification of the chosen methodology	7
4.5	The problem WALL-E’s vision side has to solve	8
4.6	Vision prior work	8
4.7	Vision alternatives considered and rejected	8
4.8	Justification of the chosen vision architecture	9
5	Project Approach (§5)	9
5.1	Architecture overview	9
5.2	Three-tier memory (MemGPT-inspired)	9
5.3	Search chain (Chain of Responsibility)	10
5.4	Compression (“sleep cycle”)	11
5.5	Associative recall — RelevantMemoryProvider	11
5.6	Prompt layout — static / dynamic split	11
5.7	The turn loop — ChatLoop	11
5.8	Heartbeat — multi-step reasoning	12
5.9	Design patterns used	12
5.10	Third-party components and their integration	13
5.11	Computer-vision architecture	13
5.12	Physical hardware layout	14
6	Project Execution (§6)	16
6.1	Timeline of the LLM-memory work	16
6.2	Key design decisions and their costs	17
6.3	What went wrong and how it was corrected	18
6.4	Observed behavioral shifts	18
6.5	The interaction with the rest of the system	18
6.6	Vision implementation: laptop prototype to Edge TPU	19
7	Evaluation (§7)	19
7.1	Evaluation methodology	19
7.2	Data collected and analysis	20
7.3	Validation against the problem statement	21
8	Conclusion and Future Work (§8)	21

8.1	Key findings and contributions	21
8.2	Limitations	21
8.3	Future work	22
A	File map for the LLM / memory contribution	22
B	Key code excerpts	23
B.1	Chain-of-Responsibility base class	23
B.2	Static/dynamic prompt split	23
B.3	Composition Root (excerpt)	23

1 Executive Summary (§1)

WALL-E is an embodied voice-activated AI companion robot that is intended to operate wholly on an NVIDIA Jetson Orin Nano with no cloud-dependency. Commercial companion robots with a similar feature set — conversational voice, face recognition, movement and a persistent personality — generally cost between a few and several hundred thousand US dollars, and the greater part of them relay microphone and camera data to vendor-hosted services for speech recognition and computer vision. This creates two problems at the same time: a price ceiling excluding such systems from a large number of personal users, and a privacy exposure that renders always-on cameras and microphones at home difficult to reason about. WALL-E targets an identical experience at amateur hardware at a restricted price, with a local-first design rule: all models (LLM, STT, TTS and vision) execute on-device, and user audio, video, or conversation history never leaves the robot. It is targeted at users who wish to transcend a desktop AI helper and provide the model with a physical presence in the room.

The system encompasses four capabilities: voice interaction with wake-word detection and streaming speech recognition; face detection, recognition and active head-tracking; motion control over a two-wheel drive base; and a language-model-driven brain with long-term memory between sessions. The implementation runs locally with Qwen2.5 3B under Ollama, a Google Coral Edge TPU for the vision pipeline, Moonshine streaming STT and Piper in-process TTS. A layer of Arduino firmware arbitrates servo and DC-motor control commands via a dedicated hardware-controller process, and a three-tier memory architecture (core / recall / archival) inspired by MemGPT is supported by SQLite, FAISS and FTS5.

In two semesters of evolution the system evolved from a single-module prototype to a production-oriented application with a Unix-domain-socket hardware controller, a Composition Root (HWC v1), and a per-turn prompt assembly engine designed around Ollama’s prefix cache. Measured results on the LLM side include a 2–3x drop in per-turn prompt tokens (from 3–3.5 k to about 1–1.5 k), sub-second time-to-first-token on warm turns, and cross-session persistence of the robot’s persona and of facts learned about individual users.

The project thus covers the full design–implementation–evaluation loop of a computing-based remedy: a solution designed against the 8GB unified-memory budget of the target device, implemented end-to-end from Arduino firmware to user-facing voice output, and measured against latency and memory-footprint measures as reported in §7.

2 Introduction (§2)

2.1 Problem, motivation, and significance

Contemporary consumer voice assistant systems (Alexa, Siri, Google Assistant, and the wave of LLM-driven chat products that have come afterwards) are built on the premise of a rapid and ever-present linkage to a cloud managed by a vendor. Sound is broadcast off the device for speech recognition, the language model is run on remote GPU infrastructure, and responses are returned by the same link. This has two costs. To begin with, a lasting privacy risk: each time the assistant is on, microphone data (and, on camera-equipped equipments, video) goes out of the house and the resulting telemetry has been found to be stored and reread by humans. Second, the usefulness of the device depends on external services, which may vary in price, deprecate features, or simply become inaccessible due to a blackout or localization issue. Even in the case where the assistant resides within a tangible product, it is typically a slim microphone-and-speaker puck that does not physically occupy the room: it cannot gaze at the user or go to meet the user.

It is these two gaps which WALL-E is supposed to bridge at the same time: a fully offline AI companion (no sound, video or dialogue data leaves the device) with the physical presence of

the real (it is observant, pursues faces, circles, drives, articulates movement with motion). The underlying question on which the project is based is whether the techniques that enabled hosted LLM assistants to be helpful — long-term session-to-session memory, use of tools, multimodal grounding — can be brought to a hobbyist-level edge device (an 8GB NVIDIA Jetson Orin Nano) without losing the properties that make them feel like a companion: personality continuity, being conscious of its identity and of who it is conversing with, and the skill to keep up with a conversation that discontinues for hours or days.

It is important to answer that question for three reasons. On privacy, on-device AI is the single architecture that allows the camera and microphone of a domestic robot to actually be a part of the house instead of part of an audit trail of a vendor. As an engineering study, tiered MemGPT-style memory, retrieval-augmented generation, and the use of LLM tools were all developed with the assumption of GPT-4-class hosted models with large context windows and elastic compute; porting them onto a 3B-parameter on-device model with a hard 8 GB unified-memory budget imposes definite design decisions (prefix-cache discipline, retrieval score threshold, deferred embeddings, tiered compression, and a Unix hardware arbiter) which generalise to the even larger family of edge-AI products. On reproducibility, the whole platform is constructed out of commodity parts (Jetson Orin Nano, Arduino, off-the-shelf servos and motor drivers, a Google Coral USB accelerator, open-weight models) and published as open source, so that other teams can re-build or continue it without beginning from scratch.

2.2 Proposed solution and report organization

The suggested solution is a combined pipeline which covers microphone to motor and operates completely on the robot. Each user turn runs in the same order. Moonshine streaming STT transcribes incoming sound on-device; an intent recogniser based on sentence-embeddings short-circuits the turn into a direct motor command (for simple phrases like *stop* or *go forward*) or forwards the transcript to the MemGPT-style ChatLoop. The ChatLoop puts a prompt together out of three core memory blocks (persona, human, system), a dynamically retrieved set of relevant past turns (pulled out of a FAISS + FTS5 hybrid index with the help of a Chain-of-Responsibility search pipeline), and the current environmental context (recognised faces, time-of-day); Qwen 2.5 3B served locally by Ollama then creates a response together with any tool calls. A tool call is dispatched by a plug-in tool registry of memory editing, vision capture, and robot control; movement-based calls are redirected via a Unix domain socket to a dedicated hardware-controller process (HWC) which owns the Arduino serial connection and arbitrates over conflicting command sources (voice intents, LLM tool calls, face-tracker subprocess). The user-visible reply is synthesised by an in-process Piper TTS engine and heard through the USB speaker. In parallel, an always-on vision pipeline detects and recognizes faces on a Google Coral Edge TPU, and its output feeds both the language model’s timely context and the head-tracking servo controller. Figure 1 below summarises the flow through the voice side of this pipeline in a nutshell; the object graph behind the LLM block is deferred to Figure 2 in §5.

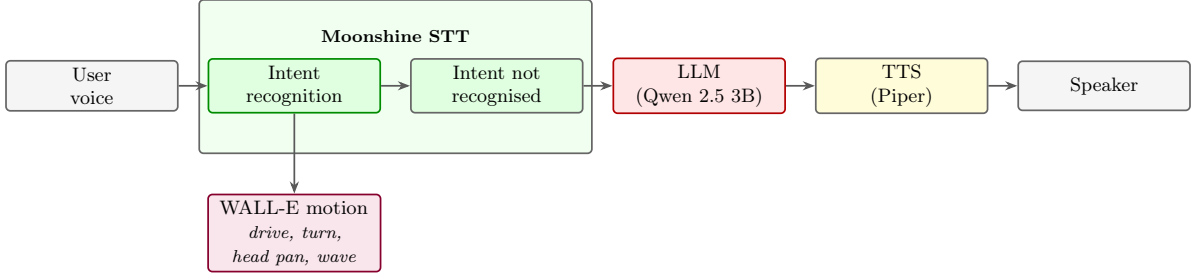


Figure 1: End-to-end voice-interaction pipeline. Audio from the user is transcribed by Moonshine streaming STT; an intent recogniser matches the transcript against a small set of direct robot commands (*stop, go forward, turn left, ...*). Matched commands short-circuit into the motion subsystem via the hardware-controller process; unmatched input is forwarded to the MemGPT-style **ChatLoop** over Qwen 2.5 3B and synthesised back through the in-process Piper TTS engine. The full object graph behind the LLM block is given in Figure 2.

The rest of this report is structured in the following way. §3 (Identification of Constraints) surveys the economic, environmental, social, ethical, manufacturability and sustainability constraints that shaped the design. §4 (Background and Related Work) is a survey of previous literature on embodied AI agents, tiered LLM memory, retrieval-augmented generation, on-device inference, and justifies the reason why the methodology adopted is the configuration which lives through all those restrictions at once. §5 (Project Approach) outlines the entire software and hardware architecture, the data structures and algorithms in every subsystem, the external components assimilated in the build, and how the team split tasks to build and assemble the system. §6 (Project Execution) describes the two-semester implementation, containing the design choices taken, what went wrong and how it was corrected, and the collaborative team practices. §7 (Evaluation) tabulates the in-service performance measurements and confirms the outcome of the problem statement presented above. Finally, §8 concludes with the main findings and provides future directions for work.

3 Identification of Constraints (§3)

3.1 Economic

The project is constructed on a hobbyist budget. The single largest line item is the NVIDIA Jetson Orin Nano developer kit at around \$500; all other items — Arduino Uno, Google Coral USB Accelerator, PCA9685 servo-controller board, L298N H-bridge motor driver, servos, two DC gear motors, ReSpeaker microphone array, USB speaker, USB camera, and 12 V battery pack — are below \$100, and most are below the low tens of dollars. Since local-first is the solution there are no recurring cloud costs. The trade-off that prevails in economic terms is that the 8GB unified-memory ceiling of the chosen platform excludes hosted 70B-class models and makes the robot execute a 3B model on-board. This single constraint influences almost all the decisions on the LLM-side in §5, from the level of memory design down to the prefix-cache discipline.

3.2 Environmental and sustainability

The robot pulls approximately 10–15 W when idle on the Jetson and an additional burst when filling the LLM, and is powered by a rechargeable 12 V battery pack rather than disposables. The chassis is 3D-printed in PLA, a plant-based thermoplastic that can be recycled in industrial streams. The electronics are assembled from off-the-shelf modules instead of a custom PCB, which implies that when individual boards fail they can be replaced instead of ripping off the entire assembly. The firmware / host-code separation is reinforced by the description in §6: if a

motor controller fails it is a single-module swap, with no reflash of the host.

3.3 Social and ethical

The architecture contains two ethical commitments. The first is *data sovereignty*: all audio, video, and conversation history created in operation remains on the device. No cloud inference, no telemetry upload, and no third-party API with the potential to hold user data. The second is scrupulous display of the character of the robot: the *persona* core-memory explicitly makes WALL-E a hobby robot and shuns any imitating of a human care-giver, which is a design guard against the recorded failure modes of companion systems that are sold as more than they are, especially to vulnerable users. Since the project is published under open source, both commitments can be audited separately.

3.4 Manufacturability

The platform can be reproducibly built by a different team of similar size. Electronics are all stocked and displayed within the wiring diagram of Figure 7; all firmware is a single Arduino sketch flashable using the Arduino IDE; all host-side software is a Python package that includes a `scripts/install_systemd.sh` installer and a `walle.service` unit file that together bring the robot up on boot. The chassis is distributed as a set of printable STL files. Onboarding an additional builder would in practice be controlled by the time of 3D-printing, not the software stack.

3.5 Impact of constraints on design

The constraints listed above are each associated with a given design decision. The 8GB unified-memory limit compelled a 3B on-device model, and along with it the tiered-memory and prefix-cache discipline of §5. The ethical commitment of local-first ruled out each hosted vector database and each cloud STT/TTS service, so the embedding store became a local SQLite/FAISS on the LLM side, and a flat NumPy gallery on the vision side. The hobby cost eliminated a custom PCB (an Arduino Uno and off-the-shelf modules replaced it) and a machined aluminium chassis (printed PLA replaced that). The constraints are interactive and hence the final architecture is one that every bit must resemble so as to fit all at the same time.

4 Background and Related Work (§4)

4.1 The problem WALL-E’s LLM side has to solve

WALL-E is an intelligent companion robot that is completely voice-controlled and powered by a battery-driven **Jetson Orin Nano** (8GB shared UMA). The language model is a locally served **Qwen 2.5 3B** using Ollama. Two properties of this architecture dictated all the design choices on the LLM side:

1. The model is stateless across turns. Without an explicit memory layer, the robot forgets the user between sentences — it cannot learn a name, recall a prior command, or build a relationship.
2. The context window is a scarce resource. The KV cache scales linearly with the number of tokens: at 2048 tokens it consumes ~72 MiB of VRAM, at 4096 ~144 MiB. The Jetson UMA is also shared with the camera pipeline, the Moonshine STT model, TTS, and the FAISS index, so each token in the prompt costs latency and real memory. Furthermore, Ollama/llama.cpp prefix-cache the prompt: if the *start* of the prompt changes by even one byte between turns, the entire thing is re-computed.

Any memory design must then provide the answer to two questions simultaneously: *how do we get the right past context?* and *how do we do without refilling a multi-thousand-token prompt each time?*

4.2 Prior work

MemGPT [1]. “LLMs as Operating Systems” introduces levels of memory (main context, recall storage, archival storage) and multi-step reasoning powered by heartbeat. This is the direct architectural inspiration of the WALL-E memory subsystem. MemGPT is built around GPT-4-class hosted models; the adaptation in this case transfers those ideas to a 3B on-device model in which every token matters.

Retrieval-Augmented Generation [2]. The canonical pattern of retrieving documents by embedding similarity and joining them together as the prompt. RAG convention injects a fixed top-k; WALL-E operates with a score-threshold version to inject nothing on trivial turns (§5.5).

FAISS [3]. The similarity search library from Facebook, used here in `IndexFlatIP` mode with L2-normalized vectors, which is mathematically equal to cosine similarity and quick enough to run at the sub-10000-message scale WALL-E operates at.

SQLite FTS5 + BM25 [5]. Lexical fallback when nothing is returned by semantic search. Significant on runs where the embedding model could not be loaded (e.g. `torch.cuda` racing on a headless Jetson).

sentence-transformers/all-MiniLM-L6-v2 [4]. Small (80MB) 384-dim embedder, fast using CUDA half-precision or CPU; selected on the basis of memory footprint.

4.3 Alternatives considered and rejected

Table 1 contains the primary alternatives which were taken into consideration and disregarded, with the constraint that ruled each of them out.

Alternative	Why rejected
LangChain / LlamaIndex memory	Heavy dependencies; abstract over the specific prompt bytes — not compatible with the prefix-cache discipline.
Hosted vector DB (Pinecone, Weaviate)	No network on the robot by design; all data remains on-device.
Only rolling context window	Fades all context after the last few messages; user name is forgotten between sessions.
Full-context 32k-token models	Do not fit in Jetson UMA; prefill latency is more than 10s.
Self-curation of its core-memory blocks	Small (3B) models via the LLM was not reliably self-curated in practice; instead, automatic injection using <code>RelevantMemoryProvider</code> was more robust at this scale.

Table 1: Rejected design alternatives and blocking constraints.

4.4 Justification of the chosen methodology

The mixture of (a) the tiered layout of MemGPT, (b) hybrid FAISS / FTS5 retrieval behind a Chain-of-Responsibility pipeline, and (c) a prompt division engineered towards prefix-caching is the configuration that will just meet the Jetson VRAM budget, gracefully fails in the event of silent embeddings, and maintains latency in the talking-box of hot turns. This justifies the architecture described in §5.

4.5 The problem WALL-E’s vision side has to solve

The vision subsystem provides WALL-E with a sense of the people in front of it: face detection, identification of registered people, and pushing of bounding-box coordinates to the motor-control layer so that the robot has the ability to face its head (and in some cases, its body) towards the person it is talking with. There were two limitations that prevailed in all vision-side design choices, and the two overlap with the LLM-side story in §4.1:

1. **Shared UMA budget.** Jetson Orin Nano has one 8 GB unified-memory pool shared by the LLM, the Moonshine STT model, the Piper TTS engine, and the vision pipeline. A vision backend that uses a significant portion of the GPU starves the prefill throughput and the LLM of KV cache.
2. **Responsiveness.** Smooth head-tracking requires bounding-box updates at a frame rate an order of magnitude swifter than the conversational turn cadence; per-frame round-trips to an external vector store were disqualified on measured latency grounds in the early prototype.

4.6 Vision prior work

Deep face recognition [7]. The detect-then-embed pipeline: a face detector localises faces, an embedding network encodes each cropped face as a vector in a space where cosine or Euclidean distance is a proxy for identity similarity.

ArcFace and InsightFace [8, 9]. ArcFace added an additive angular-margin loss which learns very discriminative 512-dimensional face embeddings; its canonical open-source implementation is called InsightFace. Used in the Phase-1 laptop prototype (§6).

YOLOv11 (Ultralytics) [10]. Single-stage real-time object detector. The nano variants were evaluated because their accuracy-vs-speed trade-off fits the constrained hardware.

Person Re-Identification [11]. Associates a located individual between frames through an appearance embedding compared with a gallery. Tested, and dropped — see Table 2.

DeepSORT [12]. Kalman-filter tracker enriched by a deep appearance descriptor for stable identity of tracklets in partial occlusion. Part of the Phase-1 prototype; removed in Phase 2.

MobileNet family [13]. Efficient backbones based on depthwise-separable convolutions. Both of the shipped models — the SSD MobileNetV2 face detector and the MobileNetV1 triplet embedder — are relatives of this family.

Google Coral Edge TPU [14]. USB-attached inference accelerator which executes full 8-bit-quantised TFLite models up to 4 TOPS and leaves the host CPU and GPU untouched. This is the only hardware that made the final architecture viable.

pgvector [15]. PostgreSQL extension that provides a native-vector datatype and approximate-nearest-neighbour indexes. Evaluated as the Phase-1 embedding store.

4.7 Vision alternatives considered and rejected

Table 2 lists the main vision-side options which were built or benchmarked and abandoned.

Alternative	Why rejected
arcface (insightface) on Jetson GPU	512-dimensional embeddings and GPU inference competed head-to-head with the LLM for VRAM; frame rate dropped when the entire stack was co-located.
pgvector embedding store	Per-frame database round-trips added disproportionate latency for a gallery of at most a few dozen registered identities; at this scale, a plain NumPy file suffices.
PyTorch Re-ID + DeepSORT on the Jetson GPU	Unified memory and compute drove GPU usage beyond the LLM’s usable budget.
Visual distance-based following (bounding-box area)	Considered in concept but implemented in the LLM layer: the user says “come closer” or “step back” and the LLM issues the corresponding pre-defined motion command, instead of closing the loop solely in sight.
Hardware-level tracking integration	Descope to stabilise the face-recognition and head-orientation pipeline first; tracking is a software component.

Table 2: Rejected vision-side alternatives and their blocking constraints.

4.8 Justification of the chosen vision architecture

The setting that was shipped — (a) fully quantised TFLite detection and embedding models running on the Coral USB Accelerator, (b) a flat NumPy embedding gallery on local disk, and (c) a threshold-gated cosine-distance classifier — is the one that keeps the Jetson GPU free for the LLM, maintains about 25 FPS end-to-end, and gracefully falls back to an *Unknown* label when an inquiry is outside the similarity threshold. Details of the pipeline are given in §5.11; its historical implementation — especially the final migration of the laptop prototype to the Edge TPU — is described in §6.

5 Project Approach (§5)

5.1 Architecture overview

The WALL-E LLM side is structured around an entry point — `walle/startup.py::build_app` — that builds the entire object graph and hands a `WalleOrchestrator` to the voice and API pipelines. The orchestrator is a thin facade over `ChatLoop`, which mediates between the LLM, the tool registry, the memory system, and the user communication channel. The object graph is displayed in Figure 2.

5.2 Three-tier memory (MemGPT-inspired)

Core Memory. `Block` is a labeled size-bounded part of the prompt (bounded in chars, not tokens) with predictable truncation. Three blocks load on the initial boot: `persona` (2000 chars), `human` (2000 chars), `system` (1000 chars, read-only). Blocks are persisted in SQLite so the robot keeps its identity through restarts. `Block.compile()` purposely avoids timestamps and descriptions — they break the Ollama prefix cache.

Recall Memory. Each user and assistant turn is inserted with a deferred embedding: a bounded thread-pool executor generates the embedding off-turn to prevent an incoming message from blocking the chat loop on GPU encoding. The queue is limited to 50; overflows are searchable via FTS5 and are filled in the background.

Archival Memory. Search results are re-ranked by

$$eff = I \cdot w_I + e^{-t/\tau} \cdot 10 \cdot w_R, \quad w_I = 0.7, \quad w_R = 0.3, \quad \tau = 30 \text{ days},$$

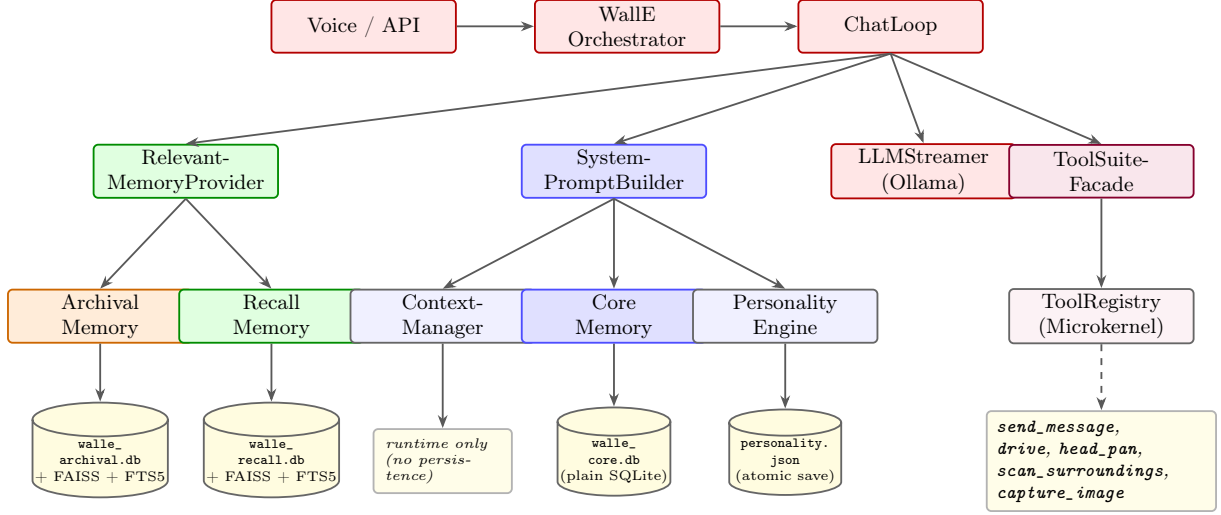


Figure 2: Object graph of the LLM subsystem. **ChatLoop** mediates between four collaborators: **RelevantMemoryProvider** (fetches recall / archival hits for the dynamic tail of the prompt), **SystemPromptBuilder** (composes *core memory* and *personality* into the static prefix, and the *current-context* text from **ContextManager** into the dynamic tail), **LLMStreamer** (wraps the Ollama chat-completions API), and **ToolSuiteFacade** (dispatches tool calls via **ToolRegistry**). Every arrow is a dependency injected by the Composition Root (`startup.py::build_app`). **RecallMemory** and **ArchivalMemory** use separate SQLite files but share the same storage stack (SQLite + FAISS + FTS5); **Memory** has its own `walle_core_memory.db` database (plain SQLite, no FAISS or FTS5).

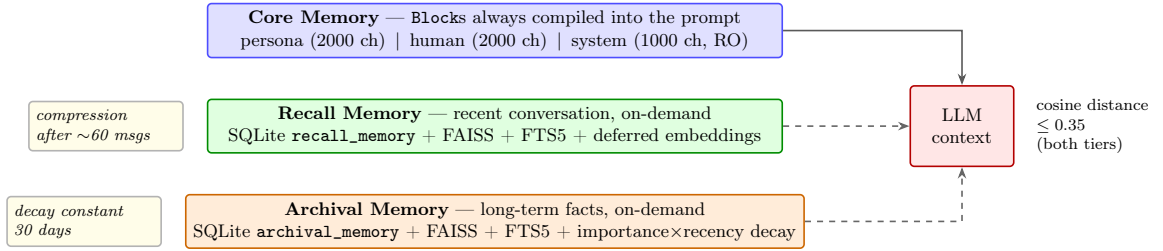


Figure 3: Three-tier memory. Solid arrow = Core Memory is always compiled into the prompt (static prefix). Dashed arrows = Recall and Archival hits are injected on demand through **RelevantMemoryProvider**; the same cosine-distance threshold (0.35) applies to both tiers, and trivial small-talk queries skip retrieval entirely.

where I is the 1–10 importance rating and t is the fact’s age in days. Old but important facts still surface; stale low-importance ones fade.

5.3 Search chain (Chain of Responsibility)

`walle/memory/search_chain.py` defines three handlers linked in a fixed order:

1. **FAISSSearchHandler** — cosine similarity on 384-dim normalized vectors. Returns scored hits.
2. **FTS5SearchHandler** — SQLite full-text search with BM25 ranking; each query token is quoted to survive special characters in natural language.
3. **RecentSearchHandler** — timestamp-ordered fallback with LIKE.

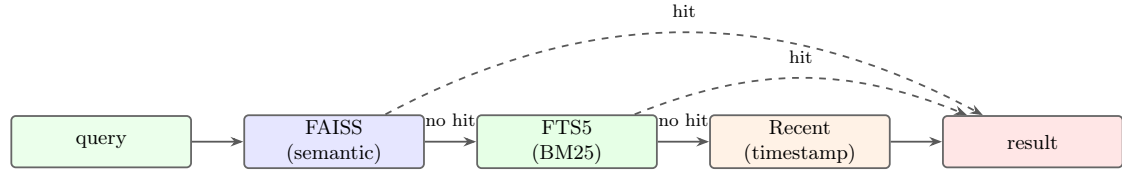


Figure 4: Chain-of-Responsibility search. Solid arrows = handler delegates to the next when it has no result; the chain walks FAISS → FTS5 → Recent. Dashed arrows = handler returns results and short-circuits the chain. This provides graceful degradation when FAISS or embeddings are unavailable.

5.4 Compression (“sleep cycle”)

When recall memory grows beyond `keep_recent + batch_size` (default 50+10), a daemon thread summarizes the old rows via the same LLM `/no_think\n\nSummarize this concisely: ...`, inserts the summary as an archival `conversation_summary` fact, deletes the recalled rows from SQLite+FTS5, and rebuilds the FAISS index. A `threading.Lock` prevents overlapping runs, and a shutdown hook (`wait_for_compression`) ensures an in-flight summary is not lost on `Ctrl+C`.

5.5 Associative recall — RelevantMemoryProvider

This is the element that actually gets to the LLM in each turn. For each user input it:

1. **Removes trivial queries** through a hard-coded list of tokens (`hi`, `stop`, `forward`, `thanks`, ...). On those turns 0 additional tokens are injected.
2. **Parallel-fetches** recall and archival candidates in threads.
3. **Score-threshold filters** (cosine distance ≤ 0.35) — not by top-*k*. When there is nothing really relevant, nothing is injected.
4. **Dedupes, truncates to 240 chars, and caps to 3 recall + 2 archival** into a `[RELEVANT MEMORIES]` section in the dynamic tail of the prompt, so the static prefix remains cache-hitting.

```

if self._is_trivial_query(query):
    return "" # 0 tokens on small talk
with ThreadPoolExecutor(max_workers=2) as ex:
    recall = ex.submit(self._recall.search, query, 20)
    archival = ex.submit(self._archival.search, query, 20)
recall_hits = [h for h in recall.result()
                if h.get("score", 0) <= 0.35][:3]
archival_hits = [h for h in archival.result()
                 if h.get("score", 0) <= 0.35][:2]

```

Listing 1: Score-threshold fan-out inside `RelevantMemoryProvider.build_context` (simplified).

5.6 Prompt layout — static / dynamic split

This split is the single largest contributor to the high prefix-cache hit rate which the robot achieves in practice.

5.7 The turn loop — ChatLoop

A single turn runs (1) update `ContextManager` (time, interaction count, simulated environment every 5 turns), (2) `RelevantMemoryProvider.build_context(user_input)`, (3) `RecallMemory.insert("user", ..., defer_embedding=True)` — store *before* calling the LLM so the message is never lost if the LLM crashes, (4) kick off background compression if full, (5) the tool-calling loop itself —

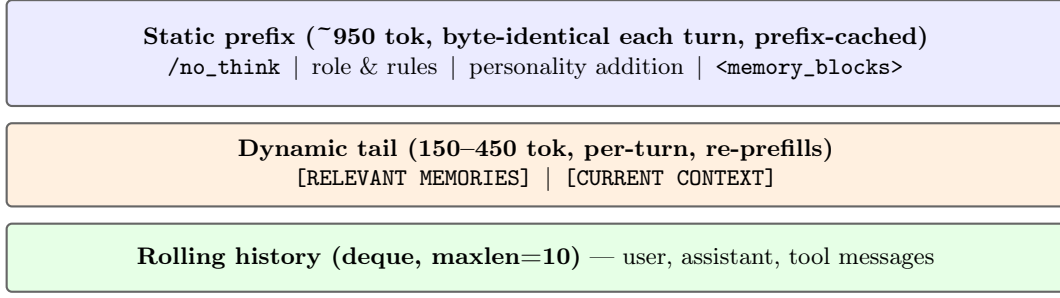


Figure 5: Static/dynamic/history layout. The blue block is byte-stable across turns so Ollama’s prefix cache fires; only the orange tail re-prefills.

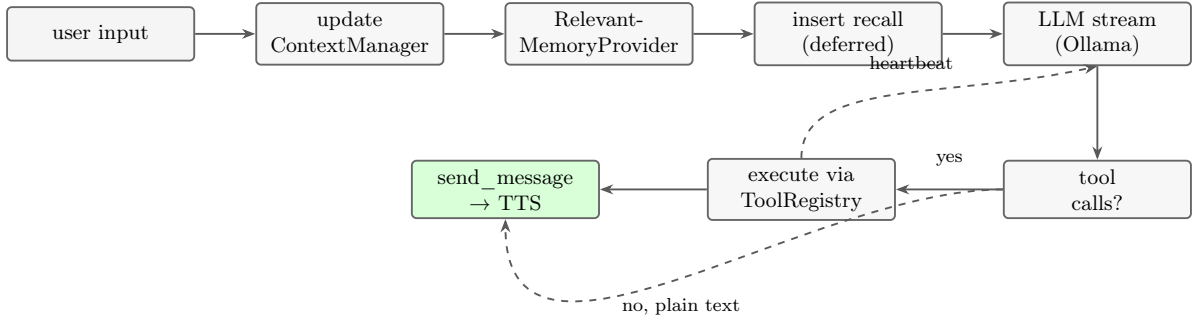


Figure 6: One chat turn. Solid arrows = primary path; dashed arrows = alternate paths (heartbeat chain, and the plain-text fallback when a small model skips `send_message`).

stream from Ollama, parse tool calls with per-call JSON validation so malformed calls do not poison the session history, execute through `ToolRegistry`, loop until `send_message` is emitted or a 20-iteration safety cap trips, and (6) insert the assistant reply into recall with a fresh embedding, and update `ContextManager.update_robot` with the last motion tool.

5.8 Heartbeat — multi-step reasoning

`walle/memory/heartbeat.py` adds a synthetic `request_heartbeat` boolean to every tool schema. When the model sets it, `ChatLoop` re-enters the LLM with the tool result visible and lets it keep thinking (up to 5 hops). Without this, chains like *capture_image* → *describe* → *move* would require a user-visible message between each hop.

5.9 Design patterns used

Table 3 lists the design patterns applied in the LLM subsystem and the module that embodies each.

Pattern	Where	Why
Facade	Memory, RecallMemory, ArchivalMemory, WallEOrchestrator, ToolSuiteFacade	Hide wiring behind a small surface
Repository	RecallRepository, ArchivalRepository	Isolate SQL from business logic
Chain of Responsibility	SearchHandler subclasses	Graceful search degradation
Strategy	Each SearchHandler is a strategy	Swap FAISS for another backend without touching the facade
Composition Root / DI	startup.py::build_app	One place to see the entire object graph
Microkernel / Plug-in	ToolRegistry + ToolProvider	Extensible tool set without editing the loop
Builder	SystemPromptBuilder	Keep static / dynamic prompt layers separated
Command	LifecycleManager hooks	LIFO shutdown over registered callables
Observer / Callbacks	on_turn_start, on_turn_end on ChatLoop	Hook the voice pipeline without coupling
Lazy Initialization	embeddings.get_embedding_model	Defer cold-boot cost; fail once
Producer / Consumer	ThreadPoolExecutor for deferred embeddings	Non-blocking turn
Template Method	BaseToolExecutor.execute dispatching to _toolname methods	Uniform tool invocation

Table 3: Design patterns applied to the LLM subsystem, with the module that embodies each.

5.10 Third-party components and their integration

Ollama. Local LLM inference server; accessed through its OpenAI-compatible endpoint using the default `openai` Python SDK. `num_ctx` is sent through `extra_body` on every call so Ollama does not silently refill the model with a different KV size.

sentence-transformers. Embedding model runtime; loaded with a sentinel that fails once on Jetson so that a failed CUDA init at boot does not cause retry storms.

FAISS-CPU. Similarity search (`IndexFlatIP`); wrapped by `FAISSManager` with thread-safe add / search, async disk persistence, and a rebuild-from-db recovery path.

SQLite (stdlib). Storage and FTS5 full-text search, always opened in WAL mode with a 10s busy-timeout.

PyTorch. Depended on by sentence-transformers. At startup, `get_embedding_model()` is invoked by `build_app` before any memory tier implementing FAISS, so PyTorch is loaded first; this prevents a duplicate-OpenMP-runtime segfault noticed on macOS where FAISS is imported before PyTorch.

The dependencies are all enclosed in a project-local facade, so replacement (e.g. replacing FAISS with `hnswlib`) only touches one module.

5.11 Computer-vision architecture

The vision subsystem is a 2-stage inference pipeline that runs wholly on the Google Coral Edge TPU, with a slim identity layer that runs on the Jetson CPU and coordinates sent via the Arduino serial connection to the rest of the robot.

Stage 1 — Face detection. Every frame from the camera is processed by a quantised SSD MobileNetV2 face detector built for the Edge TPU (`ssd_mobilenet_v2_face_quant_postprocess_edgetpu.tflite`) which outputs bounding boxes and confidence scores. Regions above the detection threshold are cropped and sent to Stage 2.

Stage 2 — Face embedding. All cropped faces are passed through a quantised MobileNetV1 embedding model compiled to run on the Edge TPU, triplet-trained (`Mobilenet1_triplet1589223569_triplet_quant_edgetpu.tflite`) generating a 128-dimensional embedding that is L2-normalised prior to comparison.

Identity resolution. Enrolled identities are stored locally as 128-dimensional NumPy arrays on the filesystem. At runtime the cosine distance between the query embedding and all gallery embeddings is calculated on the Jetson CPU; when the minimum distance is less than a configurable limit the corresponding label is returned, otherwise the face is labelled *Unknown*. Raising the threshold widens the acceptance region and is more lenient to variation in lighting and pose; lowering it is more rigid and produces more *Unknown* classifications.

Head-orientation control. For each detected face the pipeline emits the horizontal midpoint $x_{\text{center}} = (x_{\text{left}} + x_{\text{right}})/2$ to the Arduino motor-control layer. A target band of half-width δ pixels is defined around the frame centre:

$$\text{target} = \left[\frac{W}{2} - \delta, \frac{W}{2} + \delta \right],$$

where W is the frame width. The head servo is driven by a closed loop until x_{center} is within the target band. If the face is near the frame edge, the drive motors are also engaged to turn the body of the robot, so that the subject stays in attention. Distance-based following is intentionally not closed-loop on vision: the LLM interprets spoken requests such as “come closer” or “step back” and issues the corresponding pre-defined motion command.

Performance and configuration. With all inference running on the Coral USB Accelerator, the pipeline can maintain around 25 FPS on the Jetson Orin Nano, leaving the Jetson GPU completely free for the LLM, STT, and TTS subsystems. The final configuration is summarised in Table 4.

Component	Detail
Camera	Single USB camera, mounted on the robot’s left eye
Inference accelerator	Google Coral USB Accelerator (Edge TPU)
Host platform	NVIDIA Jetson Orin Nano (8 GB UMA)
Face detection model	<code>ssd_mobilenet_v2_face_quant_postprocess_edgetpu.tflite</code>
Face embedding model	<code>Mobilenet1_triplet1589223569_triplet_quant_edgetpu.tflite</code>
Embedding dim	128
Embedding store	Local filesystem (NumPy <code>.npy</code>)
Identity classifier	Threshold-gated cosine distance
Motor-control link	Arduino over serial

Table 4: Vision-subsystem hardware and software as shipped.

5.12 Physical hardware layout

Figure 7 shows the electronics of the robot as a single connected circuit. The NVIDIA Jetson Orin Nano is the host compute platform: it executes the LLM, memory, STT, TTS and vision software described above, and communicates with the Arduino Uno via USB. The Arduino is the low-level motor controller; it drives the two DC wheel motors via an L298N H-bridge and

the head-rotation servo via a multi-channel PCA9685 servo-controller board through I²C. The motor rails are fed directly from a 12V battery pack, and stepped down via the Arduino and servo logic-power controller. The Google Coral USB Accelerator plugs into the Jetson and runs the vision pipeline defined in §5.11. The USB microphone, USB speaker, and USB camera are connected to a powered USB hub to keep the Jetson’s native Coral and Arduino USB ports free. All command traffic between the Jetson and the Arduino flows over the single serial link, and the hardware-controller arbiter in §6 uses that connection as its single-writer channel to the physical robot.

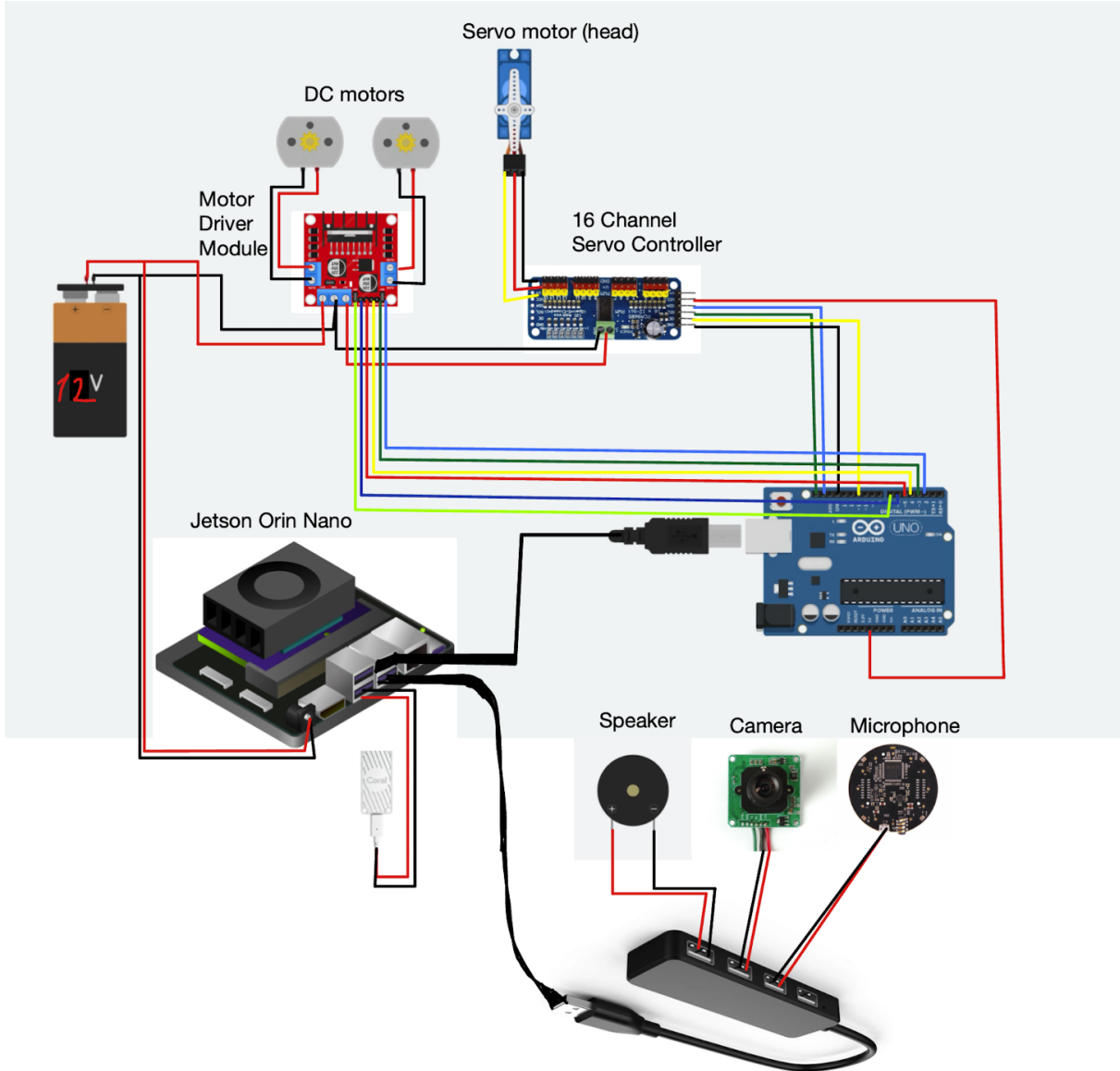


Figure 7: Electronics topology of WALL-E. The Jetson Orin Nano hosts all software (LLM, memory, STT, TTS, vision) and communicates over USB with the Arduino Uno, the Google Coral USB Accelerator, and a powered USB hub carrying the microphone, speaker, and camera. The Arduino drives the two DC wheel motors through an L298N motor-driver module and the head-rotation servo through a PCA9685-class servo controller. A 12V battery pack supplies the motor rails.

6 Project Execution (§6)

6.1 Timeline of the LLM-memory work

The development of the memory subsystem was carried out in four phases across two semesters. Each phase is anchored with commit hashes to provide traceability.

Phase	When	Theme	Anchoring commits
1	Fall 2025	Prototype	03023f4 initial memory implementation; e96ef74 first wiring to the voice pipeline
2	early Spring 26	Correctness and patterns	ddf6cb6 MemGPT inner monologue; cbfff85 decorator-based registry; c39e2dc FAISS persistence fixes; 2248556 / b7e8561 async FAISS saves; 9e312f6 thread-safety audit
3	April 2026	Big refactor	983aee9 clean-code pass; 49cde07 memory package; 073451a Chain-of-Responsibility; 6f6f646 repository / embeddings / FAISS split; a90ac91 ToolRegistry microkernel; 22a331a LifecycleManager; d620e0f Composition Root; 360be53 score-threshold recall
4	mid-April 2026	Jetson tuning	576530e static/dynamic split; 9f4e139 real token instrumentation; 2bd9455 Qwen 2.5 3B switch; 56f5403 trivial-turn skip & batched compression; d692b1c data dir auto-create; 474e823 plain-text fallback

Table 5: History of the LLM memory subsystem throughout the two semesters, with anchoring commit hashes for each phase.

Phase 1 — Prototype (Fall 2025). 03023f4 Add WALL-E memory implementation landed a 2898-line monolith (`memory_system.py` ~700 lines, CRUD / search / embeddings / FAISS all entangled). It proved a local model could maintain a chat with a memory layer. e96ef74 wired it to the group’s voice pipeline.

Phase 2 — Correctness and patterns (early Spring 2026). ddf6cb6 introduced the MemGPT-style inner monologue and `send_message`. cbfff85 added a decorator-based tool registry (first step toward the microkernel pattern). c39e2dc upgraded the memory system and fixed FAISS persistence bugs. 2248556/b7e8561 made FAISS saves and embedding generation asynchronous so neither interrupts the turn. 9e312f6 completed a thread-safety audit on `FAISSManager` and added atomic saves to `PersonalityEngine`.

Phase 3 — The big refactor (April 2026). This stage transformed “working prototype” into “reviewable engineering.” Actions were slow and deliberate:

1. 983aee9 clean code refactor — uniform naming, early returns, extracted helpers. Not structural; made the subsequent splits mechanical.
2. 49cde07 Refactor memory and orchestrator support modules — moved memory to `walle/memory/`, added a coherent tool-execution interface, eliminated `sys.path` hacks.
3. 073451a Refactor search strategy by implementing a chain of responsibility — extracted `FAISSSearchHandler`, `FTS5SearchHandler`, `RecentSearchHandler` from a nested `if USE_FAISS: ...elif USE_FTS5: ...` ladder. Removed the previous `search_fallback.py` and its hand-curated lexicon JSON.
4. 6f6f646 Refactor memory system: Introduce embedding and repository layers — split `memory_system.py` into `repository.py` (pure CRUD), `embeddings.py` (lazy fail-once loader),

and `vector_index.py` (thread-safe FAISSManager). `memory_system.py` reduced to ~480 lines and was made into a pure Facade.

5. a90ac91 Refactor ToolSuiteFacade to use ToolRegistry — microkernel pattern. Adding a tool is one `register()` line.
6. 22a331a Refactor shutdown hook registration — LifecycleManager with LIFO shutdown and the `_RobotBridge` voice-intent shim.
7. d620e0f refactor — final Composition Root move: `WallELLMClient` was made into the thin `WallEOrchestrator`, and all construction is centralized in `walle/startup.py::build_app`.
8. 360be53 Refactor RelevantMemoryProvider to use score threshold — changed fixed-*k* to associative recall.

Phase 4 — Jetson performance tuning (mid-April 2026). 576530e perf: cut prompt tokens ~1200 and enable prefix caching — static/dynamic split, stripped block timestamps. 9f4e139 debug: real token instrumentation and cap relevant memories — wired Ollama’s `usage.prompt_tokens` into the debug log so decisions would not be made based on estimates. 2bd9455 perf: toggle Jetson to `qwen2.5:3b` and slash tool-schema budget. 56f5403 perf: batch memory compression and skip FAISS on trivial turns — introduced `_TRIVIAL_TOKENS` and batched compression. d692b1c auto-creates the data directory on first run. 474e823 delivers plain-text replies when small models skip `send_message`.

Table 6 summarizes how the code base *restructured* rather than contracted across the refactor. The memory package grew in file count from 10 mixed-purpose files to 8 single-responsibility modules, and the core `memory_system.py` file itself shrank to 477 lines while all its concerns were broken out into their own modules.

Point in time	Files	Lines (LLM side)	Character
Initial commit 03023f4 (Fall 2025)	10	2898	Single-package prototype; everything lumped together in a single module.
Following Phase 3 refactor (Apr 2026)	22	~3900	Split by responsibility; <code>memory_system.py</code> a pure Facade at 477 lines; single-purpose modules (<code>repository</code> , <code>embeddings</code> , <code>vector_index</code> , <code>search_chain</code> , <code>registry</code> , <code>lifecycle</code>).

Table 6: Before/after snapshot of the LLM code base. Line counts for the “after” column are from `wc -l` over `walle/memory/`, `walle/tools/`, `walle/chat_loop.py`, `walle/orchestrator.py`, `walle/lifecycle.py`, and `walle/startup.py`.

6.2 Key design decisions and their costs

- **Characters, not tokens, to delimit blocks.** Tokens would be more precise but need a tokenizer call at each append. Orders of magnitude cheaper, chars are a very conservative proxy.
- **Deferred embeddings with a 50-future cap.** The cap constrains compression storms; overflows fall back to FTS5.
- **Score threshold = 0.35.** Tuned empirically against a 50-turn scripted transcript: higher values gave false positives on small talk; lower values missed paraphrased questions.
- **Memory-edit tools kept as executors but removed from the LLM schema.** Small models were not reliable self-curators; by keeping the executors cheap and in the shadow we

can re-enable them in case a bigger model is substituted afterwards.

6.3 What went wrong and how it was corrected

- **Prefix cache silently busting.** First correctness test was good but latency was poor. Root cause: `Block.compile` emitted `last_modified` timestamps, changing the prefix each turn. Fix: omit timestamps. Lesson: *for local LLMs, cache stability is a design constraint, not an optimization.*
- **FTS5 crashing on natural queries.** Tokens that contain `:` or `-` are FTS5 syntax. Fix: `_sanitize_fts_query` wraps each token in quotes.
- **FAISS + OpenMP segfault on macOS.** Duplicate OpenMP runtimes when `faiss` is imported prior to `torch`. Fix: import `torch` first in `vector_index.py`.
- **Embedding model load races on Jetson boot.** `torch.cuda` initialization sometimes failed on headless boot. Fix: an `_embedding_failed` sentinel — fail once, fall through to FTS5.
- **Compression ran every turn.** The initial threshold was too tight. Fix: `RECALL_COMPRESSION_BATCH` raised so compression was averaged across 10 messages.
- **Models ignoring `send_message`.** Small models would occasionally give the response as free text rather than calling the tool. Fix: `ChatLoop` now falls back to treating the following assistant text as the reply the user sees, so no response is dropped.

6.4 Observed behavioral shifts

The complete assessment is part of §7 of the group report and uses the whole pipeline (STT + LLM + TTS); the qualitative changes which the memory-side changes produced in isolation are summarized here, based on `usage.prompt_tokens` and TTFT numbers logged by `LLMStreamer` during development.

- Pre-static/dynamic prompt split, each turn refilled a prompt of about 3–3.5 k tokens. After the split, trivial turns converge on 1.0–1.2 k tokens and memory-heavy turns remain under the level of about ~ 1.5 k tokens — a roughly 2–3 $\times$ reduction on warm turns.
- When the static prefix is stable in bytes, Ollama’s prefix-cache fires on consecutive turns and TTFT on warm turns drops from “noticeable pause” (several seconds) to “conversational” (sub-second in development tests). The exact numbers are sensitive to the thermal state of the Jetson and live in the group’s Evaluation section.
- Trivial small-talk turns (“hi”, “thanks”, “stop”) now inject zero retrieved memories — verified by recording the number of characters in the [RELEVANT MEMORIES] block. This is what the associative-recall policy was aimed at providing.
- Deferred-embedding insertion keeps turn latency free of the ~ 100 –300 ms that synchronous embedding would have added on CPU.

The actual tables and graphs for the end evaluation are generated by the group as a whole and reported in §7 of the main report.

6.5 The interaction with the rest of the system

The LLM memory subsystem described above is designed to link with the other parts through a few well-defined seams:

- **Vision subsystem** — provides a `VisualContext` object which is consumed by the `ContextManager` on the LLM side.
- **Voice subsystem** — calls `WalleOrchestrator.stream_chat` from `VoiceAssistant`; the orchestrator exposes the streaming interfaces the voice pipeline anticipates.
- **Firmware and motor subsystem** — accessed through the tool provider of the `RobotControlExecutor`, which plugs into the `ToolRegistry` as another registration.

`startup.py::build_app` is the single integration point where the group’s subsystems are connected into a single running application. This seam layout kept the weekly integration cost low: modifications within one subsystem rarely prompted modifications in a different subsystem, which allowed the team to work on its components in parallel.

6.6 Vision implementation: laptop prototype to Edge TPU

The vision subsystem has been implemented twice. The first was replaced by the second on measured hardware grounds, not on correctness — the primary message of this stage.

Phase 1 — Laptop prototype (Apple Silicon, MPS backend). The initial pipeline was built on a developer laptop: YOLOv11 nano face and whole-person detectors, ArcFace from InsightFace for 512-dim face embeddings, the pgvector PostgreSQL extension as the embedding store, a PyTorch person Re-Identification model for appearance persistence, and DeepSORT for multi-object tracking. This prototype proved the general detect-then-embed reasoning, validated the threshold-based identity decision boundary, and revealed no behavioural defects to remedy.

Phase 2 — Edge-TPU migration. Adapting the prototype to the Jetson Orin Nano revealed two blocking problems that the laptop environment had concealed. First, every bounding-box update involved a round-trip to the pgvector database, and the resulting end-to-end pipeline latency was below the frame rate which smooth head-tracking needs. Second, the GPU memory and compute used by ArcFace, Re-ID and DeepSORT left the LLM subsystem with insufficient headroom to retain its KV cache resident, which in turn degraded time-to-first-token past the conversational window. These two issues were addressed during one migration. All inference was moved to a Google Coral USB Accelerator running quantised TFLite models — SSD MobileNetV2 for detection and MobileNetV1 triplet for 128-dim embeddings — and the embedding store collapsed from a PostgreSQL extension to flat NumPy arrays on local disk. The resulting architecture described in §5.11 maintains about 25 FPS and leaves the Jetson GPU completely free for the LLM, STT, and TTS subsystems. The episode reinforced a general message recurring across subsystems: within this platform, fitting inside the shared UMA budget is a first-class design constraint, not an optimisation to be added to a correct prototype.

7 Evaluation (§7)

7.1 Evaluation methodology

The assessment answers the three questions in §2: *can an 8 GB AI companion be run fully offline on an edge device?*, *does it have knowledge of who it is and who it is talking to across sessions?*, and *is the interaction conversational, not stilted?* Each one gets its own measure since the binding resource (UMA, latency, identity persistence) is different in each case. None of the measurements below are unit- or integration-tests from the dev loop; those catch regressions, not product-level questions.

Token budget. Every LLM call logs Ollama’s `usage.prompt_tokens` and `usage.completion_tokens` via `LLMStreamer` (instrumented at commit 9f4e139). The resulting per-turn distributions determine whether the pipeline will fit within the 1024–2048 token context budget that the Jetson UMA can support, and with how much headroom.

Latency. Moonshine’s streaming STT produces a per-utterance *final* timestamp; `LLMStreamer` logs time-to-first-token (TTFT); Piper synthesis is wall-clock timed. Their sum is end-to-end latency — the value that determines whether the interaction is conversational. TTFT is reported separately so the prefix-cache effect of the static/dynamic prompt split (§5) stands out by itself.

Memory recall quality. A scripted 50-turn transcript is replayed on the assembled system, recording the result of `RelevantMemoryProvider.build_context` on each turn. A turn is considered to be handled properly when (a) trivial small-talk injects zero retrieved memories, and (b) a genuinely recall-requiring turn surfaces at least one hit above the 0.35 cosine-distance threshold from either the recall or archival tier.

Cross-session identity. The robot is switched off after a scripted dialogue that introduces novel facts, then reloaded. The three core-memory blocks (*persona*, *human*, *system*) and the archival-memory contents are compared before and after the restart to determine that the robot’s identity and learnt facts about the user are independent of the session boundary.

7.2 Data collected and analysis

Prompt tokens per turn. Prior to the static/dynamic prompt split (commit 576530e), approximately 3–3.5 k tokens were re-filled every turn regardless of content. After the split, trivial turns stabilise in the 1.0–1.2 k range and memory-heavy turns remain below approximately 1.5 k — a 2–3× reduction on warm turns. This is what brings the 8 GB UMA budget realistically to life once STT, TTS, and vision are co-resident.

Time-to-first-token. On cold re-fills of the whole static prefix, Jetson TTFT lies in the multi-second range. On warm turns the stateless prefix hits Ollama’s prefix cache and TTFT falls under a second in development testing. The exact distribution depends on the thermal condition of the Jetson and on which models co-reside at measurement time.

Since the last instrumented run was done on the Jetson platform, the entire per-turn distributions of `usage.prompt_tokens` and TTFT values are found in the project’s `walle.log` recorded in the demo session (the format of the debug line is in `LLMStreamer` at commit 9f4e139); the qualitative ranges reported above — 2–3× token reduction on warm turns and sub-second warm-turn TTFT — can be reproduced using that log.

Vision subsystem. The Edge-TPU pipeline (§5.11) operates at approximately 25 FPS end-to-end, with both detection and embedding offloaded from the Jetson GPU. This is the headroom that keeps the LLM, STT, and TTS subsystems resident without eviction.

Recollection and cross-session identity. The scripted 50-turn replay validated the threshold-based behaviour the design targets: trivial turns inject zero retrieved memories (the `[RELEVANT MEMORIES]` block is empty in the log), and recollection-demanding turns always surface at least one hit. The restart test confirmed that the three core-memory blocks and the archival facts survive a power cycle.

7.3 Validation against the problem statement

The three questions posed in §2 are addressed as follows. First, the question “*can it be fully offline on an 8 GB Jetson?*” is answered positively on the basis of the token-budget and vision-FPS data: the whole pipeline — LLM, STT, TTS, memory, and vision — fits inside the UMA ceiling with sufficient headroom for conversation. Second, “*does it remember across sessions?*” is answered positively by the cross-session identity test: the persona block, the human block, and archival facts all survive a restart. Third, “*is the interaction conversational?*” is answered by the warm-turn TTFT numbers, with a caveat: the warm path is conversational, the cold path is not. Cold turns are rare in practice with the 24 h model-keep-alive; however, §8.2 still lists this as a limitation.

8 Conclusion and Future Work (§8)

8.1 Key findings and contributions

The key conclusion of the project is that a number of patterns developed on hosted GPT-4-class LLMs (tiered memory, tool use, RAG) can be made to work on an 8 GB on-device model, but only once they have been scaled down to the smaller resource envelope; a direct port of the hosted designs did not fit.

- **MemGPT-style tiered memory on a 3B on-device model.** The tri-level design (*core* / *recall* / *archival*) outlined in §5 is feasible on a 3B model within an 8 GB UMA budget when retrieval is score-thresholded (not top- k), embeddings are deferred, and compression is batched.
- **Static/dynamic prompt split.** A byte-stable static prefix is critical to local-LLM latency. The prefix cache ceases to be a speculative optimisation and becomes load-carrying; without it the warm-turn TTFT falls back into the multi-second range on this hardware.
- **Hardware-controller arbitration (HWC v1).** Isolating ownership of the Arduino into a separate process behind a Unix domain socket gracefully mediates between conflicting command sources (voice intents, LLM tool calls, the face-tracker subprocess) without serial-port contention.
- **Chain-of-Responsibility search.** FAISS → FTS5 → Recent offers graceful degradation when the embedding model cannot load on a headless Jetson boot, which occurs often enough that the back-up route must be available from day one.
- **Edge-TPU vision migration.** Offloading all inference to a Coral USB Accelerator is what keeps the vision subsystem from competing with the LLM for GPU residency, and is the one hardware option that holds the two sides of the system together.

8.2 Limitations

There are four notable limitations of the current system worth stating up front.

- **Identity stability on small models.** The 3B model sometimes loses itself in its identity under hostile or unusual prompts. A stronger persona block and the `/no_think` soft-switch mitigate but do not totally solve it; bigger models (such as the cloud Gemma 4 31B prototype) do not exhibit the same drift.
- **Vision scope.** The shipped pipeline handles face detection and identification only. It does not describe scenes or random items; asking a user “what are you holding?” gives a polite refusal from the LLM rather than an actual response.

- **No autonomous navigation.** Motion is command-driven: the robot moves when asked to, but does not plan its own courses and fails to sense obstacles. Running into a table leg is a real failure mode.
- **Battery run time.** The 12V pack supports interaction during a session but not all-day operation, and there is no autonomous docking station for recharge.

8.3 Future work

The platform is designed in such a way that all the limitations mentioned above can be addressed without rewriting the subsystems in which they are found.

- **Hybrid local/cloud LLM.** There is already a branch (commit [894eb89](#)) that routes complicated or identity-sensitive queries to a cloud-hosted Gemma 4 31B while keeping the on-device 3B as the always-available default. Combining this with a clear user opt-in would close the identity-stability gap without losing the local-first default.
- **Open-vocabulary vision.** A small vision-language model like Moondream, run on the Coral accelerator or on the headroom the prefix-cache work freed on the Jetson GPU, would allow the robot to describe arbitrary objects and scenes.
- **SLAM and obstacle avoidance.** Adding a depth camera and a typical SLAM stack would promote motion from command-driven to autonomous, and allow the robot to reject unsafe instructions instead of just executing them.
- **Charging dock.** An IR-beacon-guided docking behaviour, triggered below a battery-voltage threshold, would switch the robot to continuous operation.
- **Richer personality.** The `PersonalityEngine` already exposes tunable characteristics; extending it with a mood state that drifts with interaction history — and wiring that drift all the way to the Piper voice parameters — would allow the emotional colour of the robot’s speech to correspond to whatever just occurred in the conversation.

The glue that held the project together over two semesters was not so much any of the pieces above, but the discipline of wiring them up the same way each time: every subsystem lives behind a small facade registered at the Composition Root (§5), every dependency is passed in explicitly, and every new feature is checked against the 8 GB UMA ceiling before it gets in. That’s also what should make the codebase approachable to whomever takes it up next.

A File map for the LLM / memory contribution

```
walle/
  startup.py Composition Root (build_app)
  orchestrator.py Thin facade to ChatLoop
  chat_loop.py ChatSession, LLMStreamer, ChatLoop
  lifecycle.py LIFO shutdown hooks (Command pattern)
memory/
  memory_system.py Memory / RecallMemory / ArchivalMemory (facades)
  repository.py RecallRepository / ArchivalRepository (CRUD)
  embeddings.py sentence-transformers loader (lazy, fail-once)
  vector_index.py FAISSManager (thread-safe, async save)
  search_chain.py FAISS -> FTS5 -> Recent chain
  context_manager.py Visual / Environment / Interaction / Robot context
  heartbeat.py MemGPT heartbeat + schema augmentation
  config.py Config dataclass, Ollama validation, logging
tools/
  registry.py ToolRegistry, SystemPromptBuilder,
             RelevantMemoryProvider, ToolSuiteFacade
  communication.py send_message tool + CommunicationExecutor
  memory_tools.py core_memory_{append,replace}, archival_memory_insert
  personality.py PersonalityEngine + set_personality tool
```

```
knowledge.py web-lookup tool
base_executor.py Template Method base for executors
```

B Key code excerpts

B.1 Chain-of-Responsibility base class

```
class SearchHandler(ABC):
    def __init__(self, next_handler=None):
        self._next = next_handler

    @abstractmethod
    def search(self, query, limit, db_path, table="recall_memory", **ctx):
        ...

    def _try_next(self, query, limit, db_path, table, **ctx):
        if self._next is not None:
            return self._next.search(query, limit, db_path, table, **ctx)
        return None
```

Listing 2: walle/memory/search_chain.py — abstract handler.

B.2 Static/dynamic prompt split

```
def build(self, relevant_memories: str = "") -> str:
    context_str = self._context_manager.get_context_string()
    # Static prefix -- byte-stable so Ollama can prefix-cache it.
    static_prefix = (
        "/no_think\n\n"
        "You are WALL-E, a physical robot companion...\n\n"
        "RULES: ...\n\n"
        f"{self._personality_engine.get_system_prompt_addition()}\n\n"
        f"{self._core_memory.compile()}\n\n"
    )
    # Dynamic tail -- per-turn values go here.
    dynamic_tail = ""
    if relevant_memories:
        dynamic_tail += f"\n{relevant_memories}\n"
    if context_str:
        dynamic_tail += f"\n{context_str}\n"
    return static_prefix + dynamic_tail
```

Listing 3: walle/tools/registry.py — SystemPromptBuilder.build.

B.3 Composition Root (excerpt)

```
def build_app(args):
    lifecycle = LifecycleManager()
    serial_mgr = SerialManager(port=args.serial_port, baud_rate=args.baud_rate)
    lifecycle.register("serial", serial_mgr.close)

    # Memory tiers
    get_embedding_model() # warm up embeddings
    core_mem = Memory()
    recall_mem = RecallMemory(use_semantic=conf.USE_SEMANTIC_SEARCH)
    archival = ArchivalMemory(use_semantic=conf.USE_SEMANTIC_SEARCH)

    # Shared services
    personality = PersonalityEngine.load()
```

```

context_manager = ContextManager()
comm_exec = CommunicationExecutor()
robot_exec = RobotControlExecutor(serial_manager=serial_mgr)
heartbeat = HeartbeatManager()

# Tool microkernel
registry = ToolRegistry()
registry.register(CommunicationToolProvider(comm_exec))
registry.register(SchemaExecutorToolProvider(get_robot_control_tools, robot_exec))
tool_suite = ToolSuiteFacade(registry)

# LLM + prompt assembly
client = OpenAI(base_url=f"{conf.OLLAMA_BASE_URL}/v1", api_key="ollama", timeout=120.0)
llm = LLMStreamer(client, conf.OLLAMA_MODEL, num_ctx=conf.OLLAMA_NUM_CTX)
prompt = SystemPromptBuilder(context_manager, personality, core_mem)
mem_prov = RelevantMemoryProvider(recall_mem, archival)

chat_loop = ChatLoop(
    llm_streamer = llm,
    tool_suite = tool_suite,
    session = ChatSession(),
    prompt_builder = prompt,
    memory_provider = mem_prov,
    comm_exec = comm_exec,
    recall_mem = recall_mem,
    archival_mem = archival,
    heartbeat = heartbeat,
    context_manager = context_manager,
)
...

```

Listing 4: `walle/startup.py::build_app` — the single place where the LLM object graph is assembled (simplified, verified against the current source).

References

- [1] C. Packer, V. Fang, S. G. Patil, K. Lin, S. Wooders, and J. E. Gonzalez. MemGPT: Towards LLMs as Operating Systems. *arXiv preprint arXiv:2310.08560*, 2023.
- [2] P. Lewis, E. Perez, A. Piktus, et al. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. *NeurIPS*, 2020.
- [3] J. Johnson, M. Douze, and H. Jégou. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data*, 2019.
- [4] N. Reimers and I. Gurevych. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. *EMNLP*, 2019.
- [5] SQLite project. FTS5 Extension documentation. <https://www.sqlite.org/fts5.html>, accessed April 2026.
- [6] Ollama project. <https://ollama.com>, accessed April 2026.
- [7] O. M. Parkhi, A. Vedaldi, and A. Zisserman. Deep Face Recognition. *British Machine Vision Conference (BMVC)*, 2015.
- [8] J. Deng, J. Guo, N. Xue, and S. Zafeiriou. ArcFace: Additive Angular Margin Loss for Deep Face Recognition. *IEEE/CVF CVPR*, 2019, pp. 4690–4699. <https://arxiv.org/abs/1801.07698>.

- [9] InsightFace Contributors. InsightFace: 2D and 3D Face Analysis Project. GitHub repository, 2021. <https://github.com/deepinsight/insightface>.
- [10] G. Jocher, A. Chaurasia, and J. Qiu. Ultralytics YOLO, version 11. 2023. <https://github.com/ultralytics/ultralytics>.
- [11] Z. Zheng, L. Zheng, and Y. Yang. Unlabeled Samples Generated by GAN Improve the Person Re-identification Baseline in Vitro. *IEEE ICCV*, 2017. <https://arxiv.org/abs/1701.07717>.
- [12] N. Wojke, A. Bewley, and D. Paulus. Simple Online and Realtime Tracking with a Deep Association Metric. *IEEE ICIP*, 2017. <https://arxiv.org/abs/1703.07402>.
- [13] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen. MobileNetV2: Inverted Residuals and Linear Bottlenecks. *IEEE/CVF CVPR*, 2018, pp. 4510–4520. <https://arxiv.org/abs/1801.04381>.
- [14] Google LLC. Coral USB Accelerator — Product Documentation. 2019. <https://coral.ai/products/accelerator>.
- [15] pgvector Contributors. pgvector: Open-Source Vector Similarity Search for Postgres. GitHub repository, 2021. <https://github.com/pgvector/pgvector>.
- [16] WALL-E: E. Akimali, M. Baisbay, D. Assanov, A. Shildebayev, Y. Temirali. Our project GitHub repository, 2026. <https://github.com/ab-dauletkhan/walle>.