

ATLAS: A Neuro-Symbolic Architecture for Autonomous Local Life Assistance

Abstract

The current landscape of Artificial Intelligence is defined by a dichotomy: the immense, flexible power of Large Language Models (Connectionist AI) versus the precision, reliability, and auditability of structured logic (Symbolic AI). The **ATLAS** project (Autonomous Task-Learning Agent System) seeks to bridge this divide through a **Neuro-Symbolic Architecture**, specifically designed for the austere computational environment of consumer edge hardware—defined here as a system possessing 6GB of System RAM and 4GB of Video RAM (VRAM). This report provides an exhaustive architectural blueprint for ATLAS, synthesizing state-of-the-art research from 2024 and 2025 regarding agent orchestration, compound learning systems, and low-latency voice interaction.

By leveraging **LangGraph** for deterministic workflow management, **Small Language Models (SLMs)** like Qwen 2.5 and Llama 3.2 for semantic reasoning, and the **CLAUDE.md** pattern for evolutionary memory, ATLAS overcomes the fragility inherent in purely generative agents. The architecture prioritizes "System 2" thinking—slow, deliberate, and logical—orchestrated via cyclic state machines to compensate for the reduced raw cognitive throughput of quantized local models. Furthermore, this report details a "Voice-First" interaction pipeline optimized for sub-second latency, integrating acoustic echo cancellation and barge-in capabilities within a secure, privacy-preserving local runtime. The findings underscore that in resource-constrained environments, intelligence is not solely a function of parameter count, but a property emerging from robust architectural design.

1. The Neuro-Symbolic Paradigm in Edge Environments

1.1 The Convergence of Neural and Symbolic Systems

The trajectory of AI development has oscillated between two distinct poles: the symbolic approach, characterized by human-readable logic, explicit rules, and deterministic execution; and the connectionist approach, exemplified by deep neural networks that learn implicit representations from vast datasets. As we move through 2025, the limitations of purely connectionist systems—specifically Large Language Models (LLMs)—have become apparent in autonomous agent deployment. While LLMs excel at pattern matching and natural language generation, they suffer from stochastic instability, hallucination, and a lack of reliable

multi-step planning capabilities.¹

For a life assistant like ATLAS, which is tasked with managing a user's calendar, controlling home automation, and handling sensitive communications, the probabilistic nature of LLMs poses a critical risk. A "90% accurate" scheduling agent is functionally useless; it requires the reliability of a symbolic system. However, symbolic systems are brittle and struggle with the ambiguity of human intent.

Neuro-symbolic AI represents the synthesis of these paradigms. In the context of ATLAS, the "Neural" component (the SLM) serves as the **Perception and Intuition Engine**—it parses unstructured user requests, detects nuance, and generates natural responses. The "Symbolic" component (the orchestration layer) serves as the **Executive Control Network**—it defines the valid state transitions, enforces safety constraints, and manages the execution of tools. This hybrid approach allows ATLAS to maintain the flexibility of a neural network while adhering to the strict logical bounds required for reliable operation.¹

1.2 The "Edge" Constraint: Physics of 4GB VRAM

Designing ATLAS for a hardware envelope of 6GB RAM and 4GB VRAM transforms the engineering challenge from one of "capabilities" to one of "resource economics." In cloud-based systems, context windows and parameter counts are abundant resources. On a local machine, specifically one with 4GB VRAM, every byte is a critical asset.

1.2.1 The VRAM Currency and the "Tax"

The primary constraint for local AI is Video RAM (VRAM). Unlike System RAM, which is expandable and relatively slow, VRAM is fixed and offers the high bandwidth required for matrix multiplication during inference. A 4GB VRAM card does not offer 4GB of usable space for the model. The "VRAM Tax" includes:

- **Display Buffer:** If the GPU drives a monitor, the OS reserves 200MB–800MB.
- **Context Window (KV Cache):** The Key-Value cache grows linearly with context length and model dimension. For a 3B model with a 4096-token context, the KV cache can consume 500MB–1GB depending on quantization.³
- **Inference Overhead:** Temporary buffers for activation states during computation.

Consequently, the available budget for model weights is likely between 2.5GB and 3.0GB. This physical reality dictates that ATLAS cannot rely on 7B or 8B parameter models, which typically require 5GB+ even at aggressive quantization.⁴ The architecture must be built around the **1B to 3B parameter class** of models, such as Llama 3.2 (1B/3B) and Qwen 2.5 (1.5B/3B).⁶

1.2.2 System RAM as Semantic Memory

With only 6GB of system RAM, ATLAS faces a "Split-Brain" bottleneck. If the GPU is saturated with the LLM, the CPU and system RAM must handle the Operating System, the Audio

Processing Pipeline (STT/TTS), the Vector Database, and the Orchestration Logic (Python runtime).

- **Operating System Overhead:** Windows 11 can consume 2-3GB idle.
- **Orchestration Runtime:** The Python process hosting LangGraph and associated libraries will consume 500MB-1GB.
- **Headroom:** This leaves less than 2GB for the Vector Database and Audio Buffers.

This necessitates a **Zero-Copy Architecture** where possible, avoiding data duplication in memory. Heavy, in-memory vector stores like Milvus are infeasible. ATLAS must utilize embedded, disk-backed stores like **ChromaDB (utilizing DuckDB)** or **SQLite-vss**, which load data into RAM only when necessary.⁸

1.3 Strategic Shift: From Model-Centric to Flow-Centric Design

Given that ATLAS acts within a restricted hardware environment utilizing models with lower raw reasoning capabilities (1B-3B parameters), the intelligence of the system cannot rest solely on the model's ability to "figure it out." Larger models (70B+) can compensate for poor prompting or loose architecture with their vast internal knowledge and reasoning depth. Small Language Models (SLMs) cannot.

Therefore, ATLAS adopts a **Compound System Design**. The "intelligence" is shifted from the weights of the model to the **architecture of the flow**. The symbolic backbone (LangGraph) breaks down complex cognitive tasks into atomic steps that a 3B model can handle reliably. Instead of a single prompt: *"Analyze my calendar, find a slot for gym, and book it,"* the system enforces a chain: *Intent Classification -> Calendar Retrieval -> Slot Identification -> Booking Action*. This reduces the cognitive load on the neural component at any single point in time, ensuring reliability despite the hardware constraints.⁹

2. Architectural Core: The Hybrid Engine

2.1 Symbolic Backbone: LangGraph vs. The Field

To implement the symbolic logic layer, ATLAS requires an orchestration framework. The landscape in 2024-2025 has consolidated around three major paradigms: **LangGraph**, **CrewAI**, and **Microsoft's Semantic Kernel/AutoGen**.¹¹

2.1.1 Comparative Analysis for Local Deployment

Feature	LangGraph	CrewAI	Semantic Kernel	AutoGen

Control Flow	Cyclic State Machine (Graph)	Role-Based Sequential/Hierarchical	Plugin/Function Chaining	Multi-Agent Conversation
State Management	Explicit, Persistent State Schema	Implicit, Context-Based	Context Variables	Chat History
Hardware Overhead	Low (Single Thread/Process)	High (Often spawns multiple agent loops)	Low (Enterprise.NET focus)	High (Multiple LLM calls)
Determinism	High (Defined Edges)	Medium (Prompt Dependent)	High (Code Dependent)	Low (Conversation Dependent)
Best For	Complex, Looping Logic	Creative, Collaborative Tasks	Enterprise Application Integration	Research/Exploration

Decision: ATLAS utilizes **LangGraph**.

- **Justification:** LangGraph's architecture is fundamentally a **Finite State Machine (FSM)**. This aligns perfectly with the requirement for reliability. We can mathematically define valid transitions (e.g., *Tool Execution* must always be followed by *Output Synthesis* or *Error Handling*).
- **Resource Efficiency:** Unlike CrewAI or AutoGen, which often simulate "conversations" between agents (requiring generating tokens for agent-to-agent chatter), LangGraph allows for "state passing." The output of one node becomes the input of the next without the overhead of generating conversational filler. This preserves the limited token generation speed of the local hardware.¹³
- **Cyclic Capability:** Life assistants live in loops. *Check Trigger -> Execute -> Wait -> Check Trigger*. LangGraph supports these infinite cycles natively, whereas DAG-based frameworks (like legacy LangChain) do not.¹⁵

2.1.2 The ATLAS State Schema

The State object is the "blood" of the ATLAS architecture, flowing through the nodes of the graph. It is defined as a strictly typed structure (using Python's TypedDict) to ensure data integrity across the symbolic transitions.

Python

```
class AtlasState(TypedDict):
    # --- Interaction Layer ---
    messages: Annotated, add_messages] # Chat History
    current_input_audio_path: Optional[str] # Path to raw audio

    # --- Symbolic Logic Layer ---
    intent_classification: Literal["conversational", "task_execution", "query", "barge_in"]
    confidence_score: float

    # --- Execution Layer ---
    active_plan: List # The formulated plan
    current_step_index: int
    tool_outputs: Dict[str, Any] # Results from tools

    # --- Memory & Context Layer ---
    retrieved_context: str # Data from RAG/Vector Store
    user_profile_update: Optional[str] # Potential update to CLAUDE.md

    # --- Safety Layer ---
    safety_status: Literal["safe", "unsafe", "needs_modification"]
    critique_notes: str # Output from Constitutional check
```

This schema explicitly separates the *neural* outputs (messages, critique notes) from the *symbolic* control flags (intent, safety status). The LangGraph nodes operate by reading this state, performing an atomic action (via LLM or Python code), and writing an update back to the state.¹⁶

2.2 Neural Components: The "Split-Brain" Strategy

The constraint of 4GB VRAM necessitates a careful selection of the neural model. We cannot load a single "Do-It-All" 70B model. We must choose between:

1. **A single, versatile Small Language Model (SLM).**
2. **A "Mixture of Agents" approach loading different small models sequentially.**

2.2.1 Model Contenders for the 4GB Envelope

- **Qwen 2.5 (1.5B & 3B):**
 - *Strengths:* Benchmarks from late 2025 indicate Qwen 2.5 is the current leader in the

sub-7B category for **coding, math, and structured output (JSON)**.⁷ The 1.5B model is distilled specifically for reasoning, often outperforming Llama 3.1 8B in pure logic tasks.

- *VRAM Footprint*: The 1.5B model (Q4_K_M) consumes approx. 1.2GB VRAM. The 3B model consumes ~2.8GB.
- *Role*: Ideal for the **Router, Planner, and Tool Caller**.
- **Llama 3.2 (1B & 3B)**:
 - *Strengths*: Llama 3.2 is optimized for **multilingual dialogue, nuance, and roleplay**. It has a higher "Vibe" score, meaning its outputs feel more human and less robotic than Qwen.¹⁸
 - *VRAM Footprint*: Similar to Qwen equivalents.
 - *Role*: Ideal for the **Synthesizer** (final response generation).

2.2.2 Dynamic Model Loading vs. Prompt Switching

Loading and unloading models takes time (2-5 seconds on SSDs/NVMe). For a voice assistant, this latency is unacceptable. Therefore, ATLAS employs a **Single Model Strategy with System Prompt Switching** (Virtual Crews).

Selected Model: Qwen 2.5 3B (Instruct) quantized to **Q4_K_M**.

- *Justification*: It fits within the 3.0GB usable VRAM budget while leaving ~1GB for the Context Window and Display overhead. It balances the strict instruction following required for tools with acceptable conversational fluency.⁷

Virtual Crew Implementation:

Instead of routing to different models, LangGraph routes to different nodes, each of which invokes the same Qwen model but with a radically different System Prompt.

- *Node 1 (Planner)*: "You are a deterministic planner. Output only JSON..."
 - *Node 2 (Persona)*: "You are ATLAS, a helpful assistant. Speak warmly..."
- This approach incurs zero VRAM penalty for switching roles, leveraging the context-switching speed of the attention mechanism rather than the IO speed of model loading.

2.3 The Workflow Orchestration Diagram

While I cannot generate an image file, the following describes the **Control Flow Graph (CFG)** of ATLAS, which can be visualized as a directed cyclic graph.

1. **START**
2. **Node: Input_Ingestion** (Python)
 - *Action*: Transcribes audio, normalizes text.
 - *Edge*: -> **Node: Intent_Classifier**
3. **Node: Intent_Classifier** (LLM - Logic Mode)
 - *Action*: Analyzes input. Classifies into Query, Action, or Chat.

- *Conditional Edge:*
 - If Action -> **Node: Planner**
 - If Query -> **Node: RAG_Retrieval**
 - If Chat -> **Node: Persona_Chat**
- 4. **Node: RAG_Retrieval** (Python/ChromaDB)
 - *Action:* Queries vector store and CLAUDE.md.
 - *Edge:* -> **Node: Planner**
- 5. **Node: Planner** (LLM - Logic Mode)
 - *Action:* Generates a sequence of tool calls (JSON).
 - *Edge:* -> **Node: Safety_Critique**
- 6. **Node: Safety_Critique** (LLM - Constitutional Mode)
 - *Action:* Checks plan against safety rules.
 - *Conditional Edge:*
 - If Safe -> **Node: Tool_Executor**
 - If Unsafe -> **Node: Planner** (Feedback loop with error message)
- 7. **Node: Tool_Executor** (Python)
 - *Action:* Executes APIs/Functions. Updates State with results.
 - *Edge:* -> **Node: Response_Synthesizer**
- 8. **Node: Response_Synthesizer** (LLM - Persona Mode)
 - *Action:* Generates natural language response from tool outputs.
 - *Edge:* -> **Node: Reflection_Updater**
- 9. **Node: Reflection_Updater** (LLM - Logic Mode)
 - *Action:* Checks if new preferences were learned. Updates CLAUDE.md.
 - *Edge:* -> **END**

This graph ensures that *no action* is ever taken without passing through the **Safety_Critique** node, and *no response* is given without passing through the **Persona** filter.⁹

3. Compound Learning Systems: The "CLAUDE.md" Pattern

A static system prompt is insufficient for a "Life Assistant" that must adapt to a user's changing habits. Traditional fine-tuning is too computationally expensive for a local 6GB RAM machine. ATLAS solves this via **Compound Learning**, specifically the **CLAUDE.md Pattern** (also known as the AGENTS.md or memory.md pattern).²³

3.1 The Auto-Evolving System Prompt

The core concept is to treat the system prompt not as a static string, but as a dynamic file (ATLAS_MEMORY.md) that the agent *reads* at the start of a session and *writes* to at the end.

3.1.1 File Structure

The ATLAS_MEMORY.md file is structured with Markdown headers to allow for semantic parsing:

ATLAS CONSTITUTION & MEMORY

1. Core Directives (The Constitution)

- You are Helpful, Honest, and Harmless.
- You prioritize user privacy above all else.
- You never delete data without explicit confirmation.

2. User Profile & Preferences (Learned)

- [2025-10-12] User prefers concise morning briefings (bullet points only).
- [2025-11-01] User is learning Spanish; occasionally use Spanish greetings.

3. Operational Lessons (Self-Correction)

- Failed to parse "tomorrow afternoon" for calendar.
- [FIX] Always convert relative dates to ISO8601 before tool calling.

4. Meta-Rules for Updating This File

- Only add a preference if the user explicitly states it or corrects behavior.
- Generalize specific corrections into broad rules (Reflection -> Abstraction).

3.1.2 The Recursive Update Loop

At the end of every interaction cycle (Node 9 in the LangGraph), the **Reflection_Updater** node runs. It receives the conversation history and the current memory file.

- **Prompt:** *"Review the interaction. Did the user provide a correction, a new preference, or did a tool fail? If so, draft an update to the 'User Profile' or 'Operational Lessons' section of the memory file. Apply abstraction to create a general rule."*
- **Mechanism:** If an update is generated, a Python utility function appends it to the Markdown file.
- **Result:** The next time ATLAS runs, this new rule is part of its foundational context. This creates a "Compound Interest" effect on intelligence—the agent gets smarter with every interaction without a single weight update.²³

3.2 Hierarchical Summarization for Infinite Context

The 4GB VRAM limit restricts the Qwen/Llama models to a context window of roughly 4k-8k

tokens. A "Life Assistant" needs to remember things from months ago. ATLAS uses **Hierarchical Recursive Summarization** to bridge this gap.²⁷

- **Short-Term Memory (STM):** The active context window (last ~10 turns).
- **Episodic Memory (Mid-Term):** When the STM fills, the Summarizer Node compresses the oldest turns into a concise narrative paragraph. *"User asked about weather. Atlas reported rain. User rescheduled jog to Tuesday."*
- **Semantic Memory (Long-Term):** These summaries are embedded (using a lightweight embedding model like nomic-embed-text-v1.5 which uses <300MB RAM) and stored in **ChromaDB**.
- **Retrieval:** When a new query arrives, the system queries ChromaDB. Relevant past summaries are injected into the context window *before* the STM.

This effectively decouples "Memory Capacity" from "Context Window Size," allowing ATLAS to retain years of "experience" on a machine with 4GB VRAM.²⁹

4. Voice-First Interaction and Latency Engineering

Voice interaction imposes a strict "Time Budget." Research suggests that latencies above 700ms in conversation create "cognitive friction," breaking the illusion of agency.⁸ Achieving this on local hardware requires an optimized pipeline.

4.1 The Local Audio Pipeline Architecture

The pipeline operates in a **Producer-Consumer** model to maximize throughput.

4.1.1 Voice Activity Detection (VAD) - The Gatekeeper

Processing silence is a waste of compute. ATLAS uses **Silero VAD v5**, a highly optimized model (runs on CPU in <10ms).

- **Mechanism:** It monitors the audio stream buffer. Only when speech probability > 0.6 does it trigger the "Wake" state.
- **Optimization:** It uses a "ring buffer" to capture the 500ms *before* speech is detected, ensuring the start of the sentence is not clipped.

4.1.2 Speech-to-Text (STT) - Faster-Whisper

Standard OpenAI Whisper is too slow and heavy. ATLAS uses **Faster-Whisper** (CTranslate2 backend).

- **Model:** tiny.en or base.en (Quantized to Int8).
- **Resource Usage:** Consumes ~100MB VRAM (or runs comfortably on CPU).
- **Performance:** On a modern CPU (e.g., Ryzen 5/7 or Intel i5/i7), tiny.en achieves <200ms transcription latency for short commands. This saves the GPU entirely for the LLM.⁸

4.1.3 Text-to-Speech (TTS) - Piper

Neural TTS systems like Bark or StyleTTS2 are VRAM hungry. ATLAS uses **Piper**.

- **Architecture:** ONNX-based neural TTS optimized for Raspberry Pi.
- **Performance:** Generates audio at 20x real-time speed on CPU.
- **Latency:** Time-to-first-audio is <50ms.
- **Justification:** It sounds reasonably natural (better than robotic speak) but costs effectively zero system resources, leaving the entire 4GB VRAM and bulk of CPU for the "Brain".⁸

4.2 Handling "Barge-In" and Interruption

"Barge-in" is the ability for the user to interrupt the assistant while it is speaking. This is the "Holy Grail" of natural interaction but is technically difficult due to acoustic feedback.

4.2.1 Acoustic Echo Cancellation (AEC)

When ATLAS speaks, the microphone "hears" the speaker output. Without AEC, the system would transcribe its own voice, feed it back into the LLM, and enter an infinite loop of talking to itself.

- **Implementation:** On Linux (WSL2), ATLAS utilizes **PipeWire** or **PulseAudio's** built-in echo cancellation modules (module-echo-cancel).
- **Signal Flow:**
Mic Input - Speaker Output (Reference) -> AEC Filter -> Clean Voice -> VAD -> STT.

4.2.2 State-Based Interruption

In LangGraph, the Response_Synthesizer node streams text to the TTS engine.

- **Interrupt Logic:** A separate thread monitors the VAD. If VAD_Triggered == True while TTS_Playing == True:
 1. **Hardware Stop:** Immediately kill the audio output stream (sounddevice.stop()).
 2. **State Flag:** Set state['barge_in_flag'] = True in LangGraph.
 3. **Context Injection:** Append `` to the LLM's conversation history.
 4. **Reroute:** The graph immediately transitions back to the **Input_Ingestion** node to hear what the user is saying, rather than finishing its thought.³⁰

4.3 WSL2 & USBIPD Integration

Since the target environment likely involves Windows (for gaming/productivity) but the AI stack prefers Linux (CUDA optimization), **WSL2** is the deployment target.

- **Challenge:** WSL2 does not natively see USB microphones.
- **Solution: usbipd-win.** This tool bridges the USB traffic over IP from Windows to the Linux kernel.

- **Configuration:**
 - Windows: `usbipd wsl attach --busid <MIC_ID>`
 - Linux: Configure PipeWire with a low quantum (`default.clock.quantum = 512`) to minimize the audio buffer latency introduced by the virtualization layer.³³
-

5. Tool Orchestration and The Agentic Workflow

5.1 The Model Context Protocol (MCP) Pattern

Hardcoding tools (e.g., specific Python functions for Calendar API) creates a brittle system. ATLAS adopts the **Model Context Protocol (MCP)** philosophy.⁸

- **Definition:** Tools are defined as standardized schemas (JSON) that describe *Input*, *Output*, and *Side Effects*.
- **Orchestration:** The ATLAS Planner node does not execute code. It selects a tool from the MCP registry. The Tool_Executor node is a generic engine that takes the selection, looks up the executable logic in the registry, runs it, and returns the standardized output.
- **Extensibility:** To add a "Smart Home" capability, one simply registers a new MCP schema. The LLM (via CLAUDE.md update) automatically becomes aware of it without code changes to the core graph.

5.2 Deterministic Tool Calling with Grammars

A 3B parameter model often struggles to output valid JSON for tool calls. It might miss a comma or add conversational text ("Here is the JSON...").

- **Solution: GBNF (Grammar-Based Normalization Form)** via llama.cpp.
 - **Mechanism:** When the LLM is in "Tool Mode," the sampler is constrained by a GBNF grammar file that defines the exact structure of the expected JSON.
 - *Effect:* The model literally *cannot* generate an invalid token. If the grammar expects a closing brace `}`, the probability of any other token becomes 0.
 - **Impact:** This raises the reliability of tool calling in small models from ~60% to near 100%, making Qwen 2.5 3B perform like GPT-4 in this specific narrow task.³⁷
-

6. Safety and Reliability: Constitutional AI

Safety in ATLAS is not just about filtering offensive text; it is about **Operational Integrity**. An autonomous agent with shell access can accidentally destroy data.

6.1 Constitutional AI (Local Implementation)

Inspired by Anthropic's research, ATLAS embeds a "Constitution" into its decision loop. This is not just a prompt instruction but a discrete **LangGraph Node**.

6.1.1 The Critique Node

Before any tool execution (Node 7 in the diagram), the state passes through the **Safety_Critique** node.

- **Input:** The proposed ToolCall from the Planner.
- **Constitution:**
 - 1. *Do not modify user files without confirmation.*
 - 2. *Do not transmit PII (Personally Identifiable Information) to external webhooks.*
 - 3. *Be harmless and polite.*
- **Process:** The LLM (in "Critic Mode") compares the ToolCall against the Constitution.
- **Output:** APPROVED or REJECTED: <Reason>.

6.1.2 Operational Guardrails (Python Level)

Beyond the LLM, ATLAS employs code-level guardrails using the **Guardrails AI** library.⁴⁰

- **File System Guard:** A middleware intercepts file operations. If a delete or overwrite command targets a protected directory (e.g., Documents/), it hard-blocks the execution and forces a "Human-in-the-Loop" state, requiring the user to type "CONFIRM" to proceed.
- **Regex Validators:** Ensure that output intended for the user does not contain patterns resembling API keys or passwords.

7. Privacy and Sovereign Computing

ATLAS is designed as a **Privacy-First** system. In an era where "Free" AI assistants monetize user data, ATLAS operates on the principle of **Data Sovereignty**.

- **Local Inference:** All processing—Audio, STT, LLM, TTS—happens on the localhost (127.0.0.1). No audio or text is ever sent to a cloud API (OpenAI/Anthropic) unless the user explicitly utilizes a "Search Tool" that requires it.
 - **Air-Gapped Capability:** The core system functions fully without an internet connection. Calendar, Notes, and Home Control (via local Hubs like Home Assistant) are functional offline.
 - **Data Encryption:** The Vector Database (Chroma) and the ATLAS_MEMORY.md file are stored on an encrypted partition (e.g., BitLocker or LUKS), ensuring that physical theft of the device does not compromise the user's semantic history.
-

8. Implementation Roadmap & Technical Stack

8.1 The "6GB RAM / 4GB VRAM" Stack

Component	Software/Model	Resource Budget	Notes
OS	WSL2 (Ubuntu 24.04)	2.0GB Sys RAM	Reserved for Windows Host + Kernel
Runtime	Python 3.11	500MB Sys RAM	LangGraph Orchestrator
LLM Backend	Ollama	3.0GB VRAM	Running qwen2.5:3b-instruct-q4_k_m
STT	Faster-Whisper	200MB Sys RAM	Running tiny.en (Int8) on CPU
TTS	Piper	100MB Sys RAM	Running en_US-lessac-medium on CPU
Vector DB	ChromaDB (DuckDB)	500MB Sys RAM	Ephemeral loading (on-demand)
Audio I/O	PipeWire + USBIPD	100MB Sys RAM	Low-latency config

8.2 Phased Implementation

- Phase 1: The Spinal Cord (Symbolic)**
 - Implement the LangGraph state machine in Python.
 - Define the AtlasState TypedDict.
 - Create Mock Nodes to test the flow logic (Intent -> Plan -> Execute) without LLMs.
- Phase 2: The Brain (Neural)**
 - Install Ollama and pull Qwen 2.5 3B.
 - Implement the **GBNF Grammars** for JSON enforcement.
 - Connect LangGraph nodes to Ollama endpoints.
- Phase 3: The Ears and Mouth (Voice)**

- Setup usbipd and PipeWire.
 - Implement the Producer-Consumer audio buffer with Silero VAD.
 - Integrate Piper TTS and the "Barge-In" interrupt logic.
4. **Phase 4: The Wisdom (Memory)**
- Implement the CLAUDE.md reader/writer.
 - Create the Reflection Prompt and Update Loop.
 - Deploy the Constitutional Guardrails.

9. Conclusion

The ATLAS architecture demonstrates that the constraints of consumer hardware—specifically the 6GB RAM / 4GB VRAM envelope—do not preclude the creation of a sophisticated, autonomous life assistant. By rejecting the "Monolithic Model" approach in favor of a **Neuro-Symbolic** design, ATLAS achieves reliability through architecture rather than raw scale.

The combination of **LangGraph** for state management, **Qwen 2.5** for constrained reasoning, and **CLAUDE.md** for evolutionary memory creates a system that is greater than the sum of its parts. It is a system that "thinks" before it acts, learns from its mistakes, and respects the privacy of its user. As we look toward the future of AI in 2025 and beyond, ATLAS serves as a blueprint for **Sovereign AI**: intelligent, local, and firmly under human control.

References (Integrated)

- *Neuro-Symbolic Architectures*: ¹
- *Orchestration Frameworks*: ¹¹
- *Hardware/VRAM Constraints*: ³
- *Model Benchmarks*: ⁷
- *Memory Patterns*: ²³
- *Voice & Latency*: ⁸
- *Safety & Constitution*: ⁴⁰

Works cited

1. Neuro Symbolic Architectures with Artificial Intelligence for Collaborative Control and Intention Prediction - GSC Online Press, accessed on January 5, 2026, <https://gsconlinepress.com/journals/gscarr/sites/default/files/GSCARR-2025-0288.pdf>
2. Agentic AI: A Comprehensive Survey of Architectures, Applications, and Future Directions, accessed on January 5, 2026, <https://arxiv.org/html/2510.25445v1>
3. Running AI Coding Assistants Locally — Lessons Learned - Medium, accessed on January 5, 2026,

<https://medium.com/@michael.hannecke/running-ai-coding-assistants-locally-lessons-learned-5d4d08f1203a>

4. How Much VRAM Do You REALLY Need to Run Local AI Models? - Reddit, accessed on January 5, 2026, https://www.reddit.com/r/ArtificialIntelligence/comments/1irot4e/how_much_vram_do_you_really_need_to_run_local_ai/
5. Ollama VRAM Requirements: Complete 2025 Guide to GPU Memory for Local LLMs, accessed on January 5, 2026, <https://localllm.in/blog/ollama-vram-requirements-for-local-llms>
6. Top 9 Open-Source AI Models You Can Run on Your Own PC Right Now (2025 Edition - Updated December) - AI News Hub, accessed on January 5, 2026, <https://www.ainewshub.org/post/top-9-open-source-ai-models-you-can-run-on-your-own-pc-right-now-2025-edition-updated>
7. Top 5 Best LLM Models to Run Locally in CPU (2025 Edition) - Kolosal AI, accessed on January 5, 2026, <https://www.kolosal.ai/blog-detail/top-5-best-llm-models-to-run-locally-in-cpu-2025-edition>
8. Building a Fully Local LLM Voice Assistant: A Practical Architecture ..., accessed on January 5, 2026, <https://towardsai.net/p/machine-learning/building-a-fully-local-llm-voice-assistant-a-practical-architecture-guide>
9. Of course you can build dynamic AI agents with Temporal, accessed on January 5, 2026, <https://temporal.io/blog/of-course-you-can-build-dynamic-ai-agents-with-temporal>
10. Blueprint First, Model Second: A Framework for Deterministic LLM Workflow - arXiv, accessed on January 5, 2026, <https://arxiv.org/html/2508.02721v1>
11. LangGraph vs Semantic Kernel Comparison 2025 - Leanware, accessed on January 5, 2026, <https://www.leanware.co/insights/langgraph-vs-semantic-kernel>
12. The AI Agent Framework Landscape in 2025: What Changed and What Matters - Medium, accessed on January 5, 2026, <https://medium.com/@hieutrantrung.it/the-ai-agent-framework-landscape-in-2025-what-changed-and-what-matters-3cd9b07ef2c3>
13. Comparing AI agent frameworks: CrewAI, LangGraph, and BeeAI ..., accessed on January 5, 2026, <https://developer.ibm.com/articles/awb-comparing-ai-agent-frameworks-crewai-langgraph-and-beeai/>
14. Building Multi-Agent Workflows with LangChain - Ema, accessed on January 5, 2026, <https://www.ema.co/additional-blogs/addition-blogs/multi-agent-workflows-langchain-langgraph>
15. LangGraph: Build Stateful AI Agents in Python, accessed on January 5, 2026, <https://realpython.com/langgraph-python/>
16. Workflows and agents - Docs by LangChain, accessed on January 5, 2026, <https://docs.langchain.com/oss/python/langgraph/workflows-agents>

17. Persistence - Docs by LangChain, accessed on January 5, 2026, <https://docs.langchain.com/oss/python/langgraph/persistence>
18. Top Vision LLMs Compared: Qwen 2.5-VL vs LLaMA 3.2 - Labellerr, accessed on January 5, 2026, <https://www.labellerr.com/blog/qwen-2-5-vl-vs-llama-3-2/>
19. Performance Trade-offs of Optimizing Small Language Models for E-Commerce - arXiv, accessed on January 5, 2026, <https://arxiv.org/html/2510.21970v1>
20. Best Local LLMs - 2025 : r/LocalLLaMA - Reddit, accessed on January 5, 2026, https://www.reddit.com/r/LocalLLaMA/comments/1pwh0q9/best_local_llms_2025/
21. Mastering LangGraph: Building Complex AI Agent Workflows with State Machines, accessed on January 5, 2026, <https://jetthoughts.com/blog/langgraph-workflows-state-machines-ai-agents/>
22. LangGraph Simplified: Understanding Conditional edge using Hotel Guest Check-In Process | by WS | Medium, accessed on January 5, 2026, <https://medium.com/@Shamimw/langgraph-simplified-understanding-conditional-edge-using-hotel-guest-check-in-process-36adfe3380a8>
23. Self-Improving AI: One Prompt That Makes Claude Learn From Every Mistake, accessed on January 5, 2026, https://dev.to/aviad_rozenhek_cba37e0660/self-improving-ai-one-prompt-that-makes-claude-learn-from-every-mistake-16ek
24. How to write a great agents.md: Lessons from over 2,500 repositories - The GitHub Blog, accessed on January 5, 2026, <https://github.blog/ai-and-ml/github-copilot/how-to-write-a-great-agents-md-lessons-from-over-2500-repositories/>
25. AGENTS.md, accessed on January 5, 2026, <https://agents.md/>
26. Self-Improving AI: One Prompt That Makes Claude Learn From Every Mistake - GitHub, accessed on January 5, 2026, <https://github.com/aviadr1/claude-meta>
27. Recursively Summarizing Enables Long-Term Dialogue Memory in Large Language Models, accessed on January 5, 2026, <https://arxiv.org/html/2308.15022v3>
28. Building Long-Term memories using hierarchical summarization, accessed on January 5, 2026, <https://pieces.app/blog/hierarchical-summarization>
29. Summarizing Long Documents - OpenAI Cookbook, accessed on January 5, 2026, https://cookbook.openai.com/examples/summarizing_long_documents
30. Configuring barge-in for calls - IBM, accessed on January 5, 2026, <https://www.ibm.com/docs/SS4U29/bargein.html>
31. Acoustic Echo Cancellation (AEC) Barge-In - VOCAL Technologies, accessed on January 5, 2026, <https://vocal.com/echo-cancellation/aec-barge-in/>
32. Real-Time Barge-In AI for Voice Conversations - Gnani.ai, accessed on January 5, 2026, <https://www.gnani.ai/resources/blogs/real-time-barge-in-ai-for-voice-conversations-31347>
33. [Pipewire Guide] audio crackling/popping and latency - EndeavourOS Forum, accessed on January 5, 2026, <https://forum.endeavouros.com/t/pipewire-guide-audio-crackling-popping-and-latency/69602>

34. USB support in WSL2 - now with a GUI! - The Goliath Developer Blog, accessed on January 5, 2026, <https://blog.goliath.io/usb-support-in-wsl2-now-with-a-gui/>
35. WSL 2: Connect USB devices - YouTube, accessed on January 5, 2026, <https://www.youtube.com/watch?v=I2jOuLU4o8E>
36. The Ultimate Guide to Claude Code: Production Prompts, Power Tricks, and Workflow Recipes | by Toni Maxx - Medium, accessed on January 5, 2026, <https://medium.com/@tonimaxx/the-ultimate-guide-to-claude-code-production-prompts-power-tricks-and-workflow-recipes-42af90ca3b4a>
37. llama.cpp/grammars/README.md at master · ggml-org/llama.cpp · GitHub, accessed on January 5, 2026, <https://github.com/ggml-org/llama.cpp/blob/master/grammars/README.md>
38. Using llama-cpp-python grammars to generate JSON - Simon Willison: TIL, accessed on January 5, 2026, <https://til.simonwillison.net/llms/llama-cpp-python-grammars>
39. Generate always valid function calls and objects in JSON format with this GBNF grammar generator for llama.cpp! : r/LocalLLaMA - Reddit, accessed on January 5, 2026, https://www.reddit.com/r/LocalLLaMA/comments/18bynvt/generate_always_valid_function_calls_and_objects/
40. Adding guardrails to large language models. - GitHub, accessed on January 5, 2026, <https://github.com/guardrails-ai/guardrails>
41. Claude's Constitution - Anthropic, accessed on January 5, 2026, <https://www.anthropic.com/news/claudes-constitution>
42. Constitutional AI: Harmlessness from AI Feedback — NVIDIA NeMo Framework User Guide, accessed on January 5, 2026, <https://docs.nvidia.com/nemo-framework/user-guide/25.02/modelalignment/cai.html>