Architectural Frameworks for Autonomous AI Assistants: Reasoning, Planning, and Execution for ATLAS

1. Executive Context: The Evolution of Agentic Cognition

The development of ATLAS, an autonomous AI assistant designed to navigate the complex intersection of physiological data constraints and user-centric workflow execution, represents a paradigmatic shift from simple conversational interfaces to truly agentic systems. As of early 2026, the landscape of Large Language Model (LLM) agents has bifurcated into two distinct operational modalities: the rapid, heuristic-driven "System 1" thinking necessary for real-time interaction, and the deliberate, search-intensive "System 2" reasoning required for complex problem-solving.

The specific use case—generating a morning workout regimen based on Garmin recovery metrics and a "sore shoulder" constraint—serves as a perfect microcosm for the broader architectural challenges facing modern AI. It requires the synthesis of hard, numerical data (recovery statistics) with semantic, unstructured constraints (injury reports), all delivered within a latency budget of approximately three seconds to maintain a natural voice interface.[1] This requirement for speed, juxtaposed with the need for high-fidelity reasoning to ensure physical safety, necessitates a move beyond monolithic prompting strategies toward hybrid, neuro-symbolic architectures.

Current research from ICLR 2024, NeurIPS 2024, and ACL 2025 highlights a transition away from linear reasoning chains toward hierarchical and graph-based execution models. The era of simple "prompt engineering" has ceded ground to "flow engineering," where the cognitive control structure—how an agent plans, executes, reflects, and remembers—is as critical as the underlying model itself. This report provides an exhaustive analysis of these architectural patterns, evaluating their theoretical underpinnings, implementation complexities, and practical viability for the ATLAS system.

---

2. Comparative Analysis of Reasoning Frameworks

The core engine of any autonomous agent is its reasoning pattern—the algorithmic structure that governs how the model processes information, formulates plans, and interacts with the external world. For ATLAS, the choice of reasoning pattern dictates not only the quality of the

generated workout plan but also the system's responsiveness and reliability.

2.1. Chain-of-Thought (CoT): The Linear Baseline

Chain-of-Thought (CoT) represents the foundational baseline for LLM reasoning. Introduced as a method to elicit step-by-step reasoning capabilities, CoT prompts the model to generate intermediate reasoning traces before arriving at a final answer. This mimics human cognitive processes where complex problems are decomposed into a sequence of simpler steps.[3]

In the context of ATLAS, a CoT approach would involve a single prompt asking the model to "think step-by-step" about the user's recovery data and injury before outputting a workout. While effective for static logic puzzles or arithmetic, CoT exhibits significant limitations in dynamic environments. It operates as an open-loop system; once the chain of thought begins, it cannot pause to verify assumptions against external reality. If the model hallucinates that a specific Garmin metric indicates "high readiness" when it actually indicates "strain," the entire subsequent reasoning chain is poisoned, leading to potentially harmful advice.[3] Furthermore, standard CoT lacks the inherent ability to utilize tools, making it insufficient for tasks requiring real-time data retrieval from APIs like Garmin Connect.

2.2. ReAct: Synergizing Reasoning and Acting

ReAct (Reasoning + Acting) advances the CoT paradigm by interleaving reasoning traces with explicit action execution. This framework, which has become the industry standard for production agents as of 2024-2025, fundamentally changes the model's role from a passive generator to an active operator.[4]

The ReAct loop operates on a cyclical mechanism:

1. **Thought:** The agent analyzes the current state and determines the immediate informational need (e.g., "I need to check the user's recovery score").
2. **Action:** The agent emits a structured command to a tool (e.g., garmin_tool.get_body_battery()).
3. **Observation:** The tool executes and returns the raw output to the agent's context window (e.g., "Body Battery: 45").
4. **Thought:** The agent synthesizes this new information (e.g., "The score is low, indicating fatigue") and determines the next step.

For ATLAS, ReAct is highly effective for sequential information gathering. It allows the system to ground its decisions in real-time data, correcting potential hallucinations. If the agent initially plans a high-intensity interval session but the *Observation* from the Garmin API reveals a low Heart Rate Variability (HRV), the subsequent *Thought* can pivot the strategy toward active recovery.

However, ReAct suffers from inherent limitations in long-horizon planning. It is essentially a

greedy algorithm, optimizing for the immediate next step without necessarily maintaining a cohesive global strategy. Research indicates that ReAct agents can get "stuck" in loops or lose the thread of the original goal when faced with complex, multi-dependency tasks.[7] Additionally, the serial nature of the Reasoning-Action-Observation loop implies that latency scales linearly with the number of steps, posing a significant challenge for the three-second response budget required by voice interfaces.

2.3. Tree of Thoughts (ToT): Strategic Exploration

Tree of Thoughts (ToT) generalizes the linear CoT approach by enabling the model to explore multiple reasoning paths simultaneously. It frames the problem-solving process as a search over a tree structure, where each node represents a partial solution or "thought".[3]

In a ToT implementation for ATLAS, the agent might generate three distinct workout strategies simultaneously:

1. *Path A:* A cardio-focused session avoiding the shoulder.
2. *Path B:* A lower-body strength session.
3. *Path C:* A full rest day with mobility work.

The system then evaluates these paths—either through self-reflection or an external scorer—to determine which best satisfies the constraints of "recovery data" and "sore shoulder." This approach allows for lookahead and backtracking; if Path B requires holding a barbell (which irritates the shoulder), the agent can discard it before presenting it to the user.

While ToT excels in strategic planning where the "best" solution depends on conflicting variables, its computational cost is prohibitively high for real-time interaction. Generating and evaluating multiple branches often requires multiple LLM inference calls per step, exploding the latency. It is generally unsuitable for synchronous voice interaction unless used asynchronously—for instance, planning the week ahead while the user sleeps, rather than the immediate morning routine.

2.4. ReAcTree: Hierarchical Decomposition

ReAcTree (2024/2025) represents a significant architectural leap tailored for embodied agents and complex planners. It addresses the limitations of ReAct's flatness and ToT's computational cost by combining tool use with a hierarchical tree structure.[7]

ReAcTree dynamically constructs an agent tree to manage complexity. A root agent decomposes the high-level goal into subgoals and delegates them to specialized child agent nodes. Crucially, it integrates explicit **Control Flow Nodes** (such as Sequence and Selector nodes inspired by Behavior Trees) to manage the execution logic.

- **Mechanism:** The root node "Workout Planner" might spawn child nodes: Recovery_Agent, Injury_Agent, and Schedule_Agent.

- **Action Space:** The action space is expanded to $A \cup L \cup E$, where A are atomic actions (tools), L is language (reasoning), and E represents "Expanding" the tree by creating new nodes.[12]
- **Memory Integration:** ReAcTree utilizes a dual memory system: episodic memory for retrieving agent-level experiences and working memory for sharing observations between nodes.[12]

Research on benchmarks like WAH-NL demonstrates that ReAcTree achieves a 61% goal success rate compared to ReAct's 31%, particularly in long-horizon tasks.[11] For ATLAS, this hierarchical approach ensures that the "Sore Shoulder" constraint is handled by a dedicated sub-agent whose sole responsibility is to validate safety, preventing the "push through pain" errors common in flatter architectures. The trade-off is a higher number of decision steps (75 vs. 58 for ReAct), but the hierarchical structure allows for parallel execution of sub-branches, potentially mitigating the latency impact.[7]

2.5. LATS: Language Agent Tree Search

Language Agent Tree Search (LATS) stands as the current state-of-the-art, unifying reasoning, acting, and planning by integrating Monte Carlo Tree Search (MCTS) with LLMs.[14]

LATS treats the LLM as an agent, a value function, and an optimizer simultaneously. It builds a search tree where each node is a state (interaction history).

1. **Selection & Expansion:** It uses MCTS to select the most promising node and expands it by sampling possible actions.
2. **Simulation:** It simulates the outcome of actions, potentially interacting with the environment (e.g., querying the Garmin API).
3. **Evaluation:** It uses the LLM to reflect on the state and assign a reward or value score.
4. **Backpropagation:** This value is propagated up the tree to update the Q-values of the parent nodes.

This framework allows the agent to learn from trial-and-error within the session, exploring different tool parameters or strategies before committing to a final answer. For ATLAS, LATS would provide the highest quality solutions for novel, high-risk scenarios, such as designing a rehabilitation plan where safety is paramount. However, the "Extreme" implementation complexity and latency make it viable only as an offline or background process.[15]

2.6. Comparative Summary Table

Feature	CoT	ReAct	ToT	ReAcTree	LATS
Core Mechanis	Linear Reasoning	Interleaved Action/Reas	Multi-path Exploration	Hierarchical Delegation	MCTS + LLM

m		oning			
Best Use Case	Simple Logic, No Tools	Sequential Tool Use	Strategic Brainstorming	Long-Horizon Planning	Novel/High-Stakes Solving
Environment Interaction	None	High	Low/Simulated	High	High
Latency Profile	Low	Medium (Linear)	High	High (Parallelizable)	Extreme
Implementation Complexity	Low	Medium	High	Very High	Extreme
Fault Tolerance	Low (Hallucination)	Medium (Error Propagation)	High (Backtracking)	Very High (Isolation)	Maximum
Recommendation for ATLAS	Leaf Nodes Only	Execution Layer	Offline Planning	**Core Architecture**	Training/Distillation

---

3. Strategic Task Decomposition

The query "prepare my morning workout based on Garmin recovery data and my sore shoulder" is deceptively simple. It requires accurate weighing of conflicting inputs: "Garmin says high energy" vs. "Shoulder says stop." A monolithic prompt asking for the final output will likely fail to prioritize the medical constraint adequately.

3.1. Static vs. Dynamic Decomposition

Static Decomposition (Predetermined) involves hard-coding the workflow steps. For a recognized intent like prepare_workout, the system would always execute a fixed sequence: 1.

Get Weather, 2. Get Biometrics, 3. Generate Plan.[18]

- *Pros:* Extremely fast and deterministic. Latency is predictable.
- *Cons:* Brittle. If the user adds a novel constraint ("...and I only have 20 minutes"), a static script might not have a slot for "Time Check," leading to a plan that is too long.

Dynamic Decomposition (Discovered) relies on the LLM to analyze the request and generate a plan on the fly.

- *Pros:* Highly flexible. The agent can insert a "Research shoulder rehab exercises" step only when the user mentions the injury.[19]
- *Cons:* Slower and non-deterministic. The planner might occasionally decide to check the stock market instead of the weather.

The "ADAPT" Hybrid Strategy:
For ATLAS, the optimal strategy is the ADAPT (As-Needed Decomposition and Planning with Language Models) methodology.20 The system should maintain a library of "Atomic Skills" (e.g., get_recovery_data, check_weather). A Planner LLM recursively decomposes the high-level goal.

- If a sub-task is recognized as a standard routine (e.g., "get recovery data"), it dispatches to a static, optimized script (System 1).
- If the sub-task is novel or complex ("account for shoulder pain"), it dynamically decomposes that further into a sub-plan (System 2).
  This hybrid approach balances the speed of static workflows with the flexibility of dynamic agents.

3.2. Hierarchical Depth and Dependency Management

Research suggests that decomposition should generally not exceed three levels of depth to prevent context fragmentation and timeouts.[19]

Proposed Decomposition for ATLAS:

1. **Level 0 (Root):** Generate Safe Workout Plan.
2. **Level 1 (Sub-Goals):**
   - *Context Gathering (Parallel):* Fetch Biometrics, Fetch Safety Constraints, Check Environment.
   - *Synthesis (Sequential):* Filter Exercises, Compose Routine.
   - *Presentation (Sequential):* Format for Voice.
3. **Level 2 (Atomic Actions):** GarminAPI.get_body_battery, VectorDB.query('shoulder injury').

Handling Dependencies:
Dependencies must be managed via a Directed Acyclic Graph (DAG). The "Plan-and-Execute" pattern excels here. The Planner generates the DAG, identifying that the Synthesis node cannot start until both Biometrics and Safety Constraints are complete. Frameworks like

LangGraph allow defining these edges explicitly.

- *Implementation:* Use asyncio.gather in Python to execute the independent "Context Gathering" nodes in parallel. This is critical for meeting the 3-second budget. The *Synthesis* node is then an "awaiting" node that triggers only when all inputs are received.[21]

---

4. Execution Architectures: Planning vs. Interleaving

The user notes that "Boris" starts in Plan Mode. This aligns with the **Plan-then-Execute** pattern, which contrasts with the **Interleaved** (ReAct) approach.

4.1. Plan-then-Execute: The Strategic Separator

In this paradigm, a "Planner" agent generates a complete schedule of actions first, which is then passed to an "Executor" agent.

- **Advantages:** Separation of concerns. The Planner focuses on the *strategy* (e.g., "Ensure the workout is low-impact due to injury"), while the Executor focuses on the *tactics* (e.g., "Call Garmin API with the correct auth token"). This reduces the cognitive load on the LLM during execution, reducing the likelihood of getting "distracted" by tool outputs.[21]
- **Disadvantages:** It is open-loop. If the execution of Step 1 reveals new information (e.g., Garmin API returns "Sensor Error"), the subsequent plan might become invalid, requiring a full re-plan.

4.2. Interleaved (ReAct): The Adaptive Explorer

The agent plans one step, executes it, observes the result, and plans the next.

- **Advantages:** Highly adaptive. It can immediately react to the Garmin sensor error by asking the user for manual input or switching to a backup data source.[4]
- **Disadvantages:** It can lose sight of the original goal. In long workflows, ReAct agents often suffer from "context drift," getting bogged down in sub-tasks (e.g., debugging the API) and forgetting to generate the workout.[8]

4.3. The "Replanning" Hybrid: Best of Both Worlds

For ATLAS, a **Replanning** architecture is the superior choice.[21]

1. **Initial Plan:** The system generates a high-level DAG (Plan-then-Execute style).
2. **Execution:** Agents begin executing the nodes.
3. **Dynamic Refinement:** After each node execution, the state is passed back to a lightweight "Evaluator." If the output matches expectations, execution proceeds. If an anomaly occurs (e.g., "Garmin data missing"), the Evaluator triggers a **Re-plan** event.

This aligns with **ReAcTree** concepts: the "Planner" is the high-level node, and the "Executors"

are the leaf nodes. If a leaf node fails, the error propagates up the tree, and the parent node decides whether to retry, ignore, or re-plan.[7] This structure provides the strategic coherence of planning with the tactical flexibility of ReAct.

Switching Strategies:
Recent research suggests using "Switching Strategies" based on uncertainty or failure. If a simple linear execution fails (e.g., API timeout), the system switches to a more complex "branching" mode to find an alternative. Conversely, if the task is standard, it remains in a fast, linear execution mode.24

---

5. Cognitive Control: Observation, Reflection, and Correction

Robustness in autonomous agents is derived not from avoiding errors, but from the ability to perceive and correct them.

5.1. The Art of "Grounded" Observation

In a ReAct loop, the quality of the *Observation* determines the quality of the next *Thought*. A common failure mode is "Context Overflow," where a tool returns a massive JSON object (e.g., 500 lines of raw Garmin data), flushing the agent's instructions out of the context window.

- **Recommendation:** Implement **Observation Wrappers**. These are middleware layers—either regex parsers or small, fast LLMs (e.g., GPT-3.5-Turbo, Llama-3-8B)—that parse raw API data into natural language summaries *before* injecting them into the agent's context.
- *Bad Observation:* {"HRV": 32, "RHR": 55, "Sleep": [... 200 lines...]}
- Good Observation: Observation: Garmin API returned valid data. Body Battery is 45 (Low). HRV is 32ms (Stressed). User slept 5 hours.
  This "grounding" ensures the agent attends only to the signals relevant to the decision.6

5.2. Reflection and the "Reflexion" Pattern

Reflection involves the agent critiquing its own plan or output. The **Reflexion** pattern maintains a memory of past failures and "heurisitcs" derived from them.[26]

- **Mechanism:** If the agent previously recommended "Overhead Press" for a sore shoulder and was corrected by the user or a safety filter, it generates a reflection: "I should avoid overhead loading when the user reports shoulder pain." This reflection is stored in episodic memory and retrieved for future planning.
- **Frequency:** Reflection is computationally expensive. It should not happen at every step. It should be triggered by:
  1. **Tool Error:** API failure or timeout.

2. **Sanity Check Failure:** A separate "Critic" agent reviews the final workout plan against constraints (e.g., "Does this plan contain shoulder loading?") before presenting it to the user.
3. **User Feedback:** If the user says "I can't do that," the agent reflects on *why* and re-plans.

5.3. Abandonment and Course Correction

To detect failing plans, use **Entropy-based metrics** or **Circular Detection**. If the agent repeats the same tool call with the same inputs twice, it is likely stuck in a loop.

- **Circuit Breaker Strategy:** If a tool fails N times, the "Circuit Breaker" opens, preventing further calls to that tool. The agent is then forced to re-plan without that resource (e.g., "I cannot access Garmin data, so I will plan based on your subjective feel").[28]
- **Course Correction:** When failure is detected, the system should escalate to a "Help" state or a simplified fallback mode.

---

6. Memory Architectures for Long-Horizon Continuity

Handling "Garmin data" (episodic/temporal) and "sore shoulder" (semantic/fact) requires distinct memory systems. A single context window is insufficient for long-term user adaptation.

6.1. Episodic vs. Semantic Memory

- **Semantic Memory (Facts & Knowledge):** Stores the user's stable profile and general knowledge.
  - *Content:* "User has a left rotator cuff injury," "User prefers morning workouts," "User has a kettlebell."
  - *Implementation:* A Vector Database (e.g., Pinecone, Chroma) storing "profile documents." These are retrieved based on query relevance. When the user says "sore shoulder," the system retrieves the "Shoulder Injury Protocol" document.[30]
- **Episodic Memory (Events & Experience):** Stores the history of past interactions and specific execution traces.
  - *Content:* "On Jan 4th, we skipped push-ups," "Garmin data from yesterday showed high stress," "The user struggled with 10kg weights last week."
  - *Implementation:* A rolling buffer of the last N turns, plus a vector store for retrieving relevant past sessions ("What did we do last time my shoulder hurt?"). This allows the agent to learn from experience.[32]

6.2. Context Maintenance & Summarization

For long workflows, the context window fills up with reasoning traces ("I need to check X... X is 5... I need to check Y...").

- **Scratchpad Summarization:** Use a "scratchpad" approach. Periodically (every 5 steps), an auxiliary LLM summarizes the state: "Completed: Checked Weather (Rain), Checked Garmin (High Energy). Pending: Select Indoor Cardio." This summary replaces the raw conversation history in the prompt, keeping the context clean and focused.[30]
- **Shared Working Memory:** In ReAcTree, working memory is shared between nodes. If Node A (Medical Agent) learns "Shoulder is sore," it writes this to a shared State object. Node B (Workout Generator) reads this state directly, without needing to see the full transcript of Node A's discovery process. This decoupling is essential for scalability.[12]

---

7. Engineering for Real-Time Voice Interaction (The 3-Second Budget)

The requirement for a ~3-second response time via voice interface is the primary engineering constraint. Standard ReAct loops with large models (GPT-4) can easily take 10-15 seconds to execute multiple steps.

7.1. The Physics of Voice Latency

Latency in voice agents is measured as **Time-to-First-Audio (TTFA)**. This is the delay between the user stopping speech and the agent producing the first sound.[35]

- **VAD (Voice Activity Detection):** The system must detect "End of Speech" rapidly. Modern VAD models (like Silero) can detect silence in ~200ms.
- **Latency Masking:** The "Thinking Time" of the LLM must be masked.
  - *Technique:* **Speech-First Response**. The agent should be architected to immediately generate acknowledgment tokens ("Let me check your recovery stats...") which are sent to the TTS engine *while* the heavier reasoning/tool calls happen in the background.
  - *Streaming:* Text must be streamed from the LLM to the TTS engine. As soon as the first sentence is complete, audio generation begins.

7.2. Asynchronous Architecture

To achieve the 3-second target, the voice interface must be decoupled from the reasoning engine.[1]

- **Synchronous Layer:** Handles Speech-to-Text (STT), filler generation ("Hmm, let me see..."), and Text-to-Speech (TTS). This layer acts immediately.
- **Asynchronous Layer:** Handles the ReAct/LATS loop. It performs the "deep thinking" and API calls.
- **Pattern:**

1. **User:** "Prepare my workout..."
2. **Sync Layer:** "Sure, checking your Garmin data now..." (TTS starts immediately, buying ~2-3 seconds).
3. **Async Layer:** Calls Garmin API (1s), analyzes shoulder constraint (1s).
4. **Sync Layer:** Receives signal from Async Layer.
5. **Async Layer:** "Okay, I see you're well-rested but have shoulder pain. Here is a lower-body focused routine..."

7.3. Distilled LATS for Runtime

Since LATS (Tree Search) is too slow for runtime execution, the recommendation is to use **Offline LATS with Distillation**.[36]

- *Method:* Run LATS simulations overnight on common user scenarios (e.g., "High Recovery + Injury") to find the optimal reasoning paths.
- *Distillation:* Fine-tune a smaller, faster model (e.g., Llama-3-8B) on these optimal trajectories.
- *Result:* The runtime agent possesses the "intuition" of the tree search (knowing which path to take) without incurring the computational cost of expanding the tree in real-time. This allows a small, fast model to approximate the performance of a large reasoning model.

---

8. Practical Implementation and Reliability Patterns

The following implementation utilizes **LangGraph**, a library for building stateful, multi-actor applications with LLMs, to implement the **ReAcTree** and **Plan-and-Execute** patterns discussed.

8.1. Architecture Diagram (Textual)

Code snippet

```
graph TD
    UserInput --> Router
    Router --> PlannerNode
    PlannerNode --> |Generates DAG| ExecutionEngine

    subgraph ExecutionEngine [Parallel Execution Layer]
        BioAgent --> |ReAct Loop| GarminTool
        SafetyAgent --> |RAG Lookup| MedicalDB
```

```
    end

    ExecutionEngine --> SynthesizerNode
    SynthesizerNode --> CriticNode

    CriticNode --> |Pass| FinalResponse
    CriticNode --> |Fail| PlannerNode
```

8.2. Python Implementation Pattern (LangGraph)

This code demonstrates a **Plan-and-Execute** pattern with a **Circuit Breaker** for tool reliability, designed for the ATLAS requirements.

Python

```python
import operator
import asyncio
from typing import Annotated, List, TypedDict, Union
from langgraph.graph import StateGraph, END
from langchain_core.messages import BaseMessage, HumanMessage, AIMessage
from langchain_openai import ChatOpenAI

# --- 1. State Definition ---
# Using ReAcTree concept: shared state for sub-agents
class PlanState(TypedDict):
    input: str
    plan: List[str]  # The list of steps
    biometrics: dict # Shared working memory
    constraints: List[str] # Shared working memory
    response: str
    errors: List[str]

# --- 2. Tool Circuit Breaker Pattern  ---
class CircuitBreaker:
    def __init__(self, failure_threshold=3, recovery_timeout=60):
        self.failures = 0
        self.threshold = failure_threshold
        self.state = "CLOSED" # OPEN, CLOSED, HALF-OPEN

    def call(self, tool_func, *args):
```

```
        if self.state == "OPEN":
            return {"error": "Tool unavailable due to repeated failures."}
        try:
            result = tool_func(*args)
            self.failures = 0 # Reset on success
            return result
        except Exception as e:
            self.failures += 1
            if self.failures >= self.threshold:
                self.state = "OPEN"
            return {"error": f"Error: {str(e)}"}

# --- 3. Nodes ---

def planner_agent(state: PlanState):
    """Decomposes the high-level query into parallelizable steps."""
    print("--- PLANNING ---")
    # In production, use an LLM to generate this DAG
    # For ATLAS, we detect 'workout' and activate the ADAPT hybrid plan
    return {
        "plan": ["fetch_biometrics", "check_safety", "synthesize"],
        "errors":
    }

async def biometrics_node(state: PlanState):
    """Leaf Agent: Fetches Garmin Data."""
    print("--- FETCHING BIOMETRICS ---")
    # Simulate Async API call with Circuit Breaker
    cb = CircuitBreaker()
    # Mock Garmin API
    data = cb.call(lambda: {"body_battery": 85, "hrv_status": "Balanced"})
    return {"biometrics": data}

async def safety_node(state: PlanState):
    """Leaf Agent: Checks Medical Constraints."""
    print("--- CHECKING SAFETY ---")
    # Simulate RAG lookup for "sore shoulder"
    constraints =
    return {"constraints": constraints}

def synthesizer_node(state: PlanState):
    """Synthesizes data into a final plan."""
    print("--- SYNTHESIZING ---")
```

```
    bio = state.get("biometrics")
    safe = state.get("constraints")

    if "error" in bio:
        return {"response": "I couldn't access your Garmin data, but based on your shoulder..."}

    # Logic to combine inputs
    workout = f"Based on Body Battery {bio['body_battery']} and your shoulder constraints ({',
'.join(safe)}), here is your plan: 1. Warmup with band pull-aparts..."
    return {"response": workout}

def reflector_node(state: PlanState):
    """Critic: Checks if constraints were respected."""
    response = state["response"]
    constraints = state.get("constraints",)

    # Simple keyword check (Production would use LLM)
    for c in constraints:
        if "Overhead Press" in response: # Violation!
            return {"errors":}

    return {} # Pass

# --- 4. Graph Construction ---
workflow = StateGraph(PlanState)

workflow.add_node("planner", planner_agent)
workflow.add_node("biometrics", biometrics_node)
workflow.add_node("safety", safety_node)
workflow.add_node("synthesizer", synthesizer_node)
workflow.add_node("reflector", reflector_node)

workflow.set_entry_point("planner")

# Parallel Execution Edge
workflow.add_edge("planner", "biometrics")
workflow.add_edge("planner", "safety")

# Synchronization Point
workflow.add_edge("biometrics", "synthesizer")
workflow.add_edge("safety", "synthesizer")

workflow.add_edge("synthesizer", "reflector")
```

```
def should_continue(state: PlanState):
    if state.get("errors"):
        return "planner" # Re-plan on error
    return END

workflow.add_conditional_edges("reflector", should_continue)

app = workflow.compile()
```

8.3. Best Practices for Production

1. **Logging Agent Reasoning:** Do not just log the final output. You must capture the *thought traces*. Use structured logging (JSON) to capture {"thought": "...", "tool": "...", "args": "...", "latency": 1.2s}. Tools like **LangSmith** or **Arize Phoenix** are essential for visualizing these traces and debugging why a specific "thought" led to a tool failure.[38]
2. **Timeout Strategies:** Set aggressive timeouts on tool calls (e.g., 2 seconds for Garmin API). If the tool times out, the agent must not hang; it should receive a TimeoutError observation and decide to proceed with default data or ask the user.[29]
3. **Semantic Caching:** Cache the "Planner" output. If the user asks "What's my workout?" at 8:00 AM and again at 8:05 AM, do not re-run the reasoning chain. Serve the cached plan unless the underlying state (biometrics) has changed significantly. Use **Redis** for semantic caching of vector embeddings.[30]

---

9. Conclusion

Building ATLAS requires balancing the depth of **Tree Search** with the speed of **ReAct**. The optimal architecture for 2026 is a **Hierarchical ReAcTree** system implemented via **LangGraph**.

- **Reasoning:** Use **ReAcTree** concepts to break the "Workout" goal into independent "Biometric" and "Medical" sub-agents. This ensures the "Sore Shoulder" constraint is handled by a specialist node.
- **Execution:** Adopt a **Plan-then-Execute** flow with **Interleaved Replanning** to handle dynamic exceptions (like API failures) without losing the strategic thread.
- **Memory:** Isolate **Episodic** (Interaction History) from **Semantic** (Medical Constraints) using vector stores to prevent context window saturation.
- **Latency:** Mask reasoning time using **Speech-First** filler tokens and parallelize the "Data Fetching" and "Constraint Checking" branches of the tree using asynchronous execution.

By moving from a single monolithic LLM call to a structured cognitive architecture, ATLAS can achieve the reliability of a rules-based system with the flexibility of a generative agent, delivering safe, personalized coaching within the strict constraints of a real-time voice

interface.

Works cited

1. AsyncVoice Agent: Real-Time Explanation for LLM Planning and Reasoning - arXiv, accessed on January 5, 2026, https://arxiv.org/html/2510.16156v1
2. Engineering for Real-Time Voice Agent Latency - Cresta, accessed on January 5, 2026, https://cresta.com/blog/engineering-for-real-time-voice-agent-latency
3. Comparing Reasoning Frameworks: ReAct, Chain-of-Thought, and Tree-of-Thoughts | by allglenn | Stackademic, accessed on January 5, 2026, https://blog.stackademic.com/comparing-reasoning-frameworks-react-chain-of-thought-and-tree-of-thoughts-b4eb9cdde54f
4. ReAct - Prompt Engineering Guide, accessed on January 5, 2026, https://www.promptingguide.ai/techniques/react
5. Recap of all types of LLM Agents - Towards Data Science, accessed on January 5, 2026, https://towardsdatascience.com/recap-of-all-types-of-llm-agents/
6. ReAct Agent: Guide to understand its functionalities and create it from scratch, accessed on January 5, 2026, https://www.plainconcepts.com/react-agent-ai/
7. REACTREE: HIERARCHICAL TASK PLANNING WITH DYNAMIC TREE EXPANSION USING LLM AGENT NODES - OpenReview, accessed on January 5, 2026, https://openreview.net/pdf/41275fd872ac0d51f3c880ae615f76ef09a4fd34.pdf
8. RestGPT: Connecting Large Language Models with Real-World RESTful APIs - arXiv, accessed on January 5, 2026, https://arxiv.org/pdf/2306.06624
9. ReAct, Tree-of-Thought, and Beyond: The Reasoning Frameworks Behind Autonomous AI Agents - Coforge, accessed on January 5, 2026, https://www.coforge.com/what-we-know/blog/react-tree-of-thought-and-beyond-the-reasoning-frameworks-behind-autonomous-ai-agents
10. ReAct, Chain of Thoughts and Trees of Thoughts explained with example | by Mehul Gupta | Data Science in Your Pocket | Medium, accessed on January 5, 2026, https://medium.com/data-science-in-your-pocket/react-chain-of-thoughts-and-trees-of-thoughts-explained-with-example-b9ac88621f2c
11. ReAcTree: Hierarchical LLM Agent Trees with Control Flow for Long-Horizon Task Planning, accessed on January 5, 2026, https://chatpaper.com/paper/206486
12. ReAcTree: Hierarchical LLM Agent Trees with Control Flow for Long-Horizon Task Planning - arXiv, accessed on January 5, 2026, https://arxiv.org/html/2511.02424v1
13. ReacTreeV2 - Visual Studio Marketplace, accessed on January 5, 2026, https://marketplace.visualstudio.com/items?itemName=abhi11verma.reactreev2
14. wll199566/Awesome-LLM-Planning-Capability: [ACL 2025] A curated list of papers and resources based on "PlanGenLLMs - GitHub, accessed on January 5, 2026, https://github.com/wll199566/Awesome-LLM-Planning-Capability
15. Language Agent Tree Search - GitHub Pages, accessed on January 5, 2026, https://langchain-ai.github.io/langgraph/tutorials/lats/lats/
16. Language Agent Tree Search Unifies Reasoning, Acting, and Planning in Language Models, accessed on January 5, 2026, https://arxiv.org/html/2310.04406v3

17. Policy Guided Tree Search for Enhanced LLM Reasoning - arXiv, accessed on January 5, 2026, https://arxiv.org/html/2502.06813v1
18. Dynamic Planning vs Static Workflows: What Truly Defines an AI Agent | by Tao An - Medium, accessed on January 5, 2026, https://tao-hpu.medium.com/dynamic-planning-vs-static-workflows-what-truly-defines-an-ai-agent-b13ca5a2d110
19. Task Decomposition for Coding Agents: Architectures, Advancements, and Future Directions, accessed on January 5, 2026, https://mgx.dev/insights/task-decomposition-for-coding-agents-architectures-advancements-and-future-directions/a95f933f2c6541fc9e1fb352b429da15
20. ADAPT - Dynamic Decomposition and Planning for LLMs in Complex Decision-Making, accessed on January 5, 2026, https://promptengineering.org/adapt-dynamic-decomposition-and-planning-for-llms-in-complex-decision-making/
21. Plan-and-Execute - GitHub Pages, accessed on January 5, 2026, https://langchain-ai.github.io/langgraph/tutorials/plan-and-execute/plan-and-execute/
22. Plan-and-Execute Agents - LangChain Blog, accessed on January 5, 2026, https://blog.langchain.com/planning-agents/
23. AVIS: Autonomous Visual Information Seeking with Large Language Model Agent - NeurIPS, accessed on January 5, 2026, https://proceedings.neurips.cc/paper_files/paper/2023/file/029df12a9363313c3e41047844ecad94-Paper-Conference.pdf
24. Task Decomposition, Dynamic Role Assignment, and Low-Bandwidth Communication for Real-Time Strategic Teamwork *, accessed on January 5, 2026, https://web.media.mit.edu/~cynthiab/Readings/Stone-veloso-AIJteamwork.pdf
25. Asynchronous Reasoning: Training-Free Interactive Thinking LLMs - arXiv, accessed on January 5, 2026, https://arxiv.org/pdf/2512.10931
26. Building a Self-Correcting AI: A Deep Dive into the Reflexion Agent with LangChain and LangGraph | by Vi Q. Ha | Medium, accessed on January 5, 2026, https://medium.com/@vi.ha.engr/building-a-self-correcting-ai-a-deep-dive-into-the-reflexion-agent-with-langchain-and-langgraph-ae2b1ddb8c3b
27. Reflection Agents - LangChain Blog, accessed on January 5, 2026, https://blog.langchain.com/reflection-agents/
28. Multi-Agent System Reliability: Failure Patterns, Root Causes, and Production Validation Strategies - Maxim AI, accessed on January 5, 2026, https://www.getmaxim.ai/articles/multi-agent-system-reliability-failure-patterns-root-causes-and-production-validation-strategies/
29. danielfm/pybreaker: Python implementation of the Circuit Breaker pattern. - GitHub, accessed on January 5, 2026, https://github.com/danielfm/pybreaker
30. Build smarter AI agents: Manage short-term and long-term memory with Redis | Redis, accessed on January 5, 2026, https://redis.io/blog/build-smarter-ai-agents-manage-short-term-and-long-term-memory-with-redis/

31. Long-term Memory in LLM Applications, accessed on January 5, 2026, https://langchain-ai.github.io/langmem/concepts/conceptual_guide/
32. Memory overview - Docs by LangChain, accessed on January 5, 2026, https://docs.langchain.com/oss/python/concepts/memory
33. Understanding Episodic Memory in Artificial Intelligence | DigitalOcean, accessed on January 5, 2026, https://www.digitalocean.com/community/tutorials/episodic-memory-in-ai
34. ReAcTree: Hierarchical LLM Agent Trees with Control Flow for Long-Horizon Task Planning | Request PDF - ResearchGate, accessed on January 5, 2026, https://www.researchgate.net/publication/397279095_ReAcTree_Hierarchical_LLM_Agent_Trees_with_Control_Flow_for_Long-Horizon_Task_Planning
35. Engineering low-latency voice agents - Sierra AI, accessed on January 5, 2026, https://sierra.ai/blog/voice-latency
36. Learn-by-interact: A Data-Centric Framework for Self-Adaptive Agents in Realistic Environments - arXiv, accessed on January 5, 2026, https://arxiv.org/html/2501.10893v1
37. LEARN-BY-INTERACT: A DATA-CENTRIC FRAME- WORK FOR SELF-ADAPTIVE AGENTS IN REALISTIC ENVIRONMENTS - ICLR Proceedings, accessed on January 5, 2026, https://proceedings.iclr.cc/paper_files/paper/2025/file/dd745a0c4f91fe91866fe6788be9cc28-Paper-Conference.pdf
38. Strands Agents SDK: A technical deep dive into agent architectures and observability - AWS, accessed on January 5, 2026, https://aws.amazon.com/blogs/machine-learning/strands-agents-sdk-a-technical-deep-dive-into-agent-architectures-and-observability/
39. Understanding your agent's behavior in production - Vellum AI, accessed on January 5, 2026, https://www.vellum.ai/blog/understanding-your-agents-behavior-in-production
40. How to optimise latency for voice agents - Nikhil R, accessed on January 5, 2026, https://rnikhil.com/2025/05/18/how-to-reduce-latency-voice-agents