Architectural Blueprint for "Alex": An Autonomous, Self-Reflective AI Assistant

1. Introduction: The Paradigm of Recursive Agentic Systems

The evolution of artificial intelligence from passive response engines to persistent, state-aware agents marks a fundamental shift in software architecture. The objective of this report is to define the architectural specifications for "Alex," a personal AI assistant designed not merely as a conversational interface, but as a self-modifying software entity. Unlike traditional chatbots that reset their state with every session, Alex is conceived as a continuous process—a digital organism that retains a detailed autobiography of its interactions, understands the distinct topology of its own source code, and possesses the agency to upgrade its own infrastructure through a human-in-the-loop deployment pipeline.

This report addresses the specific requirements for Alex's construction: a dual-tiered intelligence engine utilizing Google's Gemini 3 models, a sophisticated memory system capable of temporal reasoning, and an autonomous engineering capability driven by Anthropic's Claude Code. Central to this architecture is the rejection of flat, vector-based retrieval in favor of a Temporal Knowledge Graph, enabling the recursive summarization necessary for long-term coherence. Furthermore, the infrastructure choices prioritize Google Cloud Platform (GCP) for its robust serverless ecosystem, while pragmatically integrating external tools where they offer superior agentic capabilities.

The following analysis provides a comprehensive technical roadmap, moving from high-level cognitive hierarchies to specific Python implementation patterns, ensuring Alex operates as a resilient, cost-effective, and highly intelligent personal aide.

2. Cognitive Architecture: The Dual-Cortex Model

The intelligence of an autonomous agent is defined by the orchestration of its underlying models. For Alex, a monolithic approach—using a single model for all tasks—is economically inefficient and architecturally brittle. Instead, we propose a "Dual-Cortex" architecture that segregates duties based on cognitive load, latency requirements, and cost efficiency. This design leverages the distinct strengths of the Gemini 3 family: the speed and context capacity of the Flash variant, and the reasoning depth of the Pro variant.

2.1 The Basal Cortex: Gemini 3 Flash Preview

The primary driver for Alex's routine operations is **Gemini 3 Flash Preview**. This model serves as the "always-on" consciousness of the agent, handling the vast majority of interactions, initial intent classification, and real-time data processing.

2.1.1 Architectural Justification

The selection of Gemini 3 Flash is driven by three critical architectural constraints: economic viability for high-frequency operations, massive context retention, and multimodal readiness.

Economic Efficiency for Continuous Operation An AI assistant designed to "live" alongside a user generates a significant volume of tokens. Every log line analyzed, every chat interaction, and every background summarization task consumes computational resources. Gemini 3 Flash offers a compelling pricing structure at $0.50 per 1 million input tokens and $3.00 per 1 million output tokens.[1] This is markedly lower than the Pro tier, which commands prices upwards of $2.00 to $4.00 per million input tokens depending on context length.[2] For an agent that may process hundreds of megabytes of logs or long conversation histories daily, the Flash model enables a "thinking budget" that would be prohibitive with frontier-class reasoning models. This efficiency allows Alex to perform aggressive background housekeeping—constantly re-reading and summarizing its own memory—without incurring unsustainable operational costs.

The Context Window as Working Memory Gemini 3 Flash supports a context window of up to 1 million tokens.[4] This capacity fundamentally alters the retrieval architecture. In traditional RAG (Retrieval-Augmented Generation) systems, the agent is limited to retrieving small, fragmented chunks of text. With a 1M token window, Alex can ingest entire documents, long conversation threads, or significant portions of its own codebase in a single pass. This reduces the system's reliance on complex chunking strategies for immediate, short-term tasks. For instance, when the user asks, "What did we discuss regarding the server migration last week?", Alex can load the entire week's transcript into the Flash context window, allowing the model to perform a comprehensive "in-context" search rather than relying solely on database retrieval.

Multimodal Foundations for Future Voice Integration The requirement for a future voice interface necessitates a model capable of native multimodal processing. Gemini 3 Flash supports audio inputs natively, priced at $1.00 per 1 million tokens.[1] This capability allows the architecture to be "voice-ready" without requiring a separate transcription pipeline (STT/TTS). By designing the input schema to accept audio buffers now, the transition to voice interaction will essentially be a frontend update rather than a backend re-architecture.

2.2 The Executive Cortex: Gemini 3 Pro Preview

While Flash handles the routine, **Gemini 3 Pro Preview** acts as the "Executive Cortex," invoked only when high-order reasoning, complex planning, or ambiguity resolution is required.

2.2.1 Escalation Protocols

The system utilizes a router-based design pattern. An incoming user request is first evaluated

by a lightweight classifier (running on Flash) that assigns a "complexity score." If the score exceeds a predefined threshold, the state is transferred to the Pro model.

Trigger Conditions for Pro Escalation:

1. **Ambiguity and Nuance:** When user instructions are vague or contradictory (e.g., "Refactor the memory module but keep it compatible with the old schema"), the Pro model's superior reasoning capabilities are required to infer the unstated constraints and propose a safe execution plan.[6]
2. **Complex Tool Chains:** Tasks that require a multi-step sequence of tool invocations—such as diagnosing a bug, reading logs, hypothesizing a fix, and writing a test case—demand the "agentic workflow" mastery attributed to the Gemini 3 Pro series.[6]
3. **Code Architecture Design:** Before any code is written, the architectural implications must be analyzed. Pro is tasked with reviewing the proposed changes against the system's "self-knowledge" to ensure new features do not violate existing patterns defined in CLAUDE.md.

2.3 The Engineering Sub-Agent: Claude Code with Opus 4.5

A unique requirement for Alex is the ability to understand and modify its own software architecture using "Claude Code." This is not merely an API call but the integration of a specialized CLI tool designed for autonomous software engineering.

2.3.1 Role of Claude Code

Claude Code, powered by the **Opus 4.5** model, represents the "Hands" of the system. While Gemini 3 acts as the brain, Claude Code is the instrument that interacts with the file system and the terminal. The research highlights Opus 4.5 as a state-of-the-art model for coding tasks, capable of handling ambiguity and reasoning about trade-offs without hand-holding.[7]

2.3.2 Operationalizing a CLI Tool

Integrating a CLI tool meant for humans into an autonomous pipeline requires a "headless" wrapper. The Python architecture must invoke Claude Code as a subprocess, managing the stdin/stdout streams programmatically.

- **Command Construction:** Alex (Gemini Pro) formulates a natural language instruction for Claude Code (e.g., "Run the unit tests in tests/test_memory.py and fix any failures").
- **Execution Mode:** The wrapper must utilize flags such as --print or --non-interactive to ensure the tool does not hang waiting for user confirmation.[8]
- **Output Parsing:** The standard output of Claude Code is captured and fed back into Alex's context. This closes the feedback loop: Alex plans -> Claude Code executes -> Alex reviews output -> Alex confirms or iterates.

This tripartite division of labor—Flash for routine cognition, Pro for executive reasoning, and Claude Code for engineering execution—creates a system that is both cost-effective and capable of deep, autonomous action.

3. Memory Architecture: The Temporal Knowledge Graph

The most significant architectural decision for Alex is the design of its memory system. The requirement is for a "comprehensive memory system" that allows for "fast retrieval via periodic summaries (weekly, monthly, annually)." This specific requirement for hierarchical, temporal summarization strongly contraindicates the use of standard vector databases or simple tagging systems.

3.1 The Failure of Vector Databases for Long-Term Memory

Vector databases (such as Pinecone, Milvus, or Weaviate) have become the default solution for AI memory, but they suffer from a critical limitation: they flatten complex, structured relationships into high-dimensional similarity scores. As noted in the research, vector databases function effectively as search engines but fail as "brains".[9]

Why Vectors Fail for Alex:

1. **Loss of Temporal Context:** A vector embedding of a chat log captures the *semantic meaning* of the text but loses the *temporal metadata*. If the user asks, "How has my coding style changed over the last year?", a vector database might return snippets of code from various times based on similarity, but it cannot structurally order them to show a progression.
2. **Inability to perform Recursive Roll-ups:** Vector databases do not support the concept of hierarchical summarization natively. One cannot easily ask a vector database to "summarize the vectors from last week" without retrieving all of them and processing them externally.
3. **The "Bag of Facts" Problem:** Vector RAG retrieves isolated chunks of information. It struggles with multi-hop reasoning (e.g., "Who was the client I emailed after the server crash?"). The link between the "server crash" and the "email" is a structural relationship, not necessarily a semantic one.

3.2 The Superiority of Knowledge Graphs (GraphRAG)

To meet the requirement for "nuanced understanding" and "periodic summaries," Alex will utilize a **Knowledge Graph**, specifically **Neo4j**. This approach, known as GraphRAG, combines the structural precision of graph databases with the semantic power of LLMs.[10]

3.2.1 Explicit Modeling of Time and Event

In a graph database, time is not just a timestamp property; it is a structural entity. We will implement a **Time Tree** schema.

- **Nodes:** Year, Month, Week, Day.
- **Relationships:** (:Year)-->(:Month)-->(:Week)-->(:Day).

- **Event Attachment:** Every interaction, system log, or code commit is an Event node linked to a specific Day node via a `` relationship.

This structure makes the "periodic retrieval" requirement trivial. To generate a weekly summary, Alex does not need to search the entire database. It simply queries for the specific Week node and retrieves all Event nodes connected to its child Day nodes. This query is deterministic, fast (O(1) complexity relative to total history), and comprehensive.[12]

3.2.2 The Recursive Summarization Algorithm

The "comprehensive memory" is achieved through a background metabolic process we term **Recursive Summarization**. This process transforms high-volume, low-level data (raw logs) into low-volume, high-level wisdom (summaries).

The Pipeline:

1. **Level 0 (Raw Data):** Every user interaction is stored as an Interaction node linked to the current Day.
2. **Level 1 (Daily Consolidation):** A scheduled task (Cloud Scheduler) triggers Gemini 3 Flash every 24 hours. The model retrieves all Interaction nodes for the past day. It synthesizes a DailySummary node, which is then linked to the Day node.
3. **Level 2 (Weekly Roll-up):** At the end of the week, the system retrieves the 7 DailySummary nodes (crucially, *not* the raw interactions). It synthesizes a WeeklySummary node.
4. **Level 3 (Strategic Review):** Monthly and Annual nodes are generated similarly, aggregating the summaries of the level below.

Insight: This architecture solves the context window problem permanently. Even after 10 years of operation, to summarize the "Year," Alex only needs to read 12 "Monthly Summary" nodes—a trivial token load—rather than millions of raw log lines.[14] This mimics human memory consolidation, where specific episodic details fade, but semantic, narrative structures persist.

3.3 Hybrid Retrieval: The Best of Both Worlds

While the graph provides structure, vector search is still useful for vague, unstructured queries. Modern graph databases like Neo4j support **Vector Indexing** on nodes.[16] Alex will employ a **Hybrid RAG** strategy:

1. **Semantic Entry:** The user asks, "What was that project about climate data?" The system uses a vector index to find Project or Conversation nodes with high semantic similarity.
2. **Graph Traversal:** Once the entry node is found, the system traverses the edges to find connected context: "This project was worked on in *November 2025* (Time Tree), involved *Python* (Concept), and resulted in *Commit 4a7f2* (Action)."

This hybrid approach satisfies the requirement for "fast retrieval" while maintaining the "detailed understanding" that only a graph can provide.

4. Infrastructure and Deployment: Google Cloud Platform

The user has specified a preference for **Google Cloud Services (GCP)**. The infrastructure must support the continuous, autonomous nature of Alex while remaining cost-effective and manageable by a "human-in-the-loop."

4.1 Compute Strategy: Cloud Run vs. Vertex AI

The choice of compute is between the managed **Vertex AI Agent Builder** and the more flexible **Cloud Run**.

Analysis:

- **Vertex AI Agent Builder** offers a low-code/no-code environment for building agents.[17] However, it is primarily designed for standard customer service bots and lacks the flexibility to run arbitrary CLI tools like Claude Code or manage a complex, custom Neo4j connection natively.
- **Cloud Run** is a serverless container platform. It allows the deployment of any Docker container, providing the full flexibility of a custom Python environment.[18]

Decision: Cloud Run is the superior choice for Alex. It allows the installation of the specific dependencies required (Neo4j drivers, Anthropic CLI, LangGraph runtime) and provides a standard HTTP interface for the chat frontend.

4.1.1 Handling "Scale to Zero" and Background Tasks

Cloud Run scales to zero when not in use, which is excellent for cost savings but problematic for a "living" agent that needs to perform background tasks (like memory summarization).

- **Mitigation Strategy:** We decouple the interactive component from the maintenance component.
  - **Service A (Alex Interface):** A Cloud Run service that handles user chat requests. It scales to zero when the user is away.
  - **Cloud Scheduler:** A managed cron service in GCP that triggers specific endpoints on Service A (e.g., /tasks/summarize_day) at defined intervals.[20] This "wakes up" the agent to perform its introspection and memory maintenance, ensuring the periodic summaries are generated reliably without requiring an always-on server.

4.2 Repository and Deployment Pipeline

The user requires a "standard code repository" and a "server deployment pipeline."

Repository: While Google Cloud Source Repositories exists, **GitHub** is the industry standard and offers superior integration with **Claude Code**. Claude Code is optimized for the gh (GitHub CLI) tool set, allowing it to create Pull Requests, view diffs, and manage issues natively.[22]

Therefore, hosting the code on GitHub while using GCP for compute is the optimal configuration.

Deployment Pipeline (CI/CD):

We will utilize **Google Cloud Build** connected to the GitHub repository.

1. **Trigger:** A push to the main branch (or a specific deploy tag) triggers the build.
2. **Build:** Cloud Build constructs the Docker image containing the updated Alex code.
3. **Deploy:** The image is pushed to Google Container Registry (GCR) and deployed to Cloud Run.

The "Self-Update" Loop:

When Alex (via Claude Code) determines a code change is needed:

1. Claude Code modifies the local files in the container's ephemeral workspace.
2. Claude Code runs tests.
3. If successful, Claude Code commits the changes and pushes to a feature-branch on GitHub.
4. **Human-in-the-Loop:** The user reviews the PR. Upon approval and merge, Cloud Build automatically redeploys Alex. This satisfies the requirement for the user to keep the system updated while leveraging Alex's autonomy.

4.3 Cost Optimization and Safety

A critical risk with autonomous agents on serverless infrastructure is "runaway costs." As highlighted in the research, a misconfigured "min-instance" setting or an infinite loop of self-requests can lead to massive bills.[23]

Safety Measures:

- **Concurrency Limits:** Cloud Run concurrency will be strictly limited to prevent parallel processing spikes.
- **Budget Alerts:** GCP Budget Alerts will be configured to trigger email notifications at low thresholds (e.g., $10/month).
- **Circuit Breakers:** The LangGraph router will implement a "max-steps" counter for any autonomous tool execution loop, preventing Alex from getting stuck in an infinite retry loop with Claude Code.

5. The Agentic Engineer: Operationalizing Claude Code

A defining feature of Alex is the ability to use **Claude Code**—a CLI tool intended for human developers—as a sub-agent. This requires a sophisticated wrapper to bridge the gap between the Python application and the terminal interface.

5.1 The "Headless" Interface

Claude Code is designed to be interactive, prompting the user for confirmation on dangerous actions (like file deletion or API calls). To automate this, Alex must run Claude Code in a non-interactive mode.

- **Flags:** The --print flag (or equivalent non-interactive configuration) is essential to force the tool to output text to stdout rather than rendering a TUI (Text User Interface).[8]
- **Input Injection:** For commands that strictly require confirmation, the Python wrapper must use subprocess.Popen to inject "y\n" into the stdin stream, effectively "signing off" on the agent's behalf—subject, of course, to the strict guardrails defined in the system prompts.

5.2 The Tool Definition

In the LangGraph architecture, Claude Code is defined as a **Tool Node**.

- **Input Schema:** The tool accepts a goal (string) and a context_files (list).
- **Execution Logic:**
  1. The Python wrapper constructs a shell command: claude --print "GOAL: {goal} CONTEXT: {context_files}".
  2. The command is executed in a sandboxed directory (a clone of the repo).
  3. The stdout and stderr are captured.
  4. **Output Parsing:** The wrapper parses the output to determine success or failure. If Claude Code indicates a file was modified, the wrapper verifies the file hash.
- **Feedback Loop:** The output is returned to the Gemini 3 Pro model. If the tool reports a syntax error, Gemini 3 Pro analyzes the error message and formulates a *new* prompt for Claude Code to fix the bug.

6. Self-Knowledge: "Alex, Know Thyself"

For Alex to have a "detailed understanding of its own software architecture," static training data is insufficient. The codebase changes; the training data does not. Alex needs a dynamic, real-time understanding of its own source code.

6.1 The CLAUDE.md Constitution

Research into Anthropic's best practices reveals the importance of a CLAUDE.md file in the repository root.[22] This file acts as the "Constitution" or "ReadMe for the AI."

- **Content:** It contains high-level architectural decisions, code style guidelines (PEP 8), explanation of the folder structure, and specific "Do's and Don'ts" (e.g., "Never modify the cloudbuild.yaml without human approval").
- **Context Injection:** Whenever the Claude Code tool is invoked, the content of CLAUDE.md is automatically pre-pended to the context window. This ensures that the agent always respects the architectural boundaries defined by the user and its own previous iterations.

6.2 Dynamic Repository Mapping

To supplement CLAUDE.md, Alex will employ a **Repository Map** generator. This is a Python script that runs periodically (e.g., alongside the daily memory summary).

- **AST Analysis:** The script uses Python's built-in ast module to parse every .py file in the repository.
- **Extraction:** It extracts class names, method signatures, docstrings, and import relationships.
- **Graph Ingestion:** These extracted entities are ingested into the Neo4j Knowledge Graph. A Class becomes a node; a Method becomes a node; a CALLS relationship is created between them.
- **Querying Self:** When the user asks, "How does your memory module work?", Alex does not guess. It queries the Knowledge Graph for the MemoryModule node, retrieves the current docstrings and method signatures, and generates an answer based on the *actual* state of the code.

7. Implementation Plan

The following section translates the architectural theory into a concrete implementation roadmap, utilizing Python and the specified stack.

7.1 Tech Stack Summary

- **Language:** Python 3.11+
- **Orchestration:** LangGraph (for stateful, cyclic agent flows)
- **Intelligence:** Google GenAI SDK (gemini-3-flash-preview, gemini-3-pro-preview)
- **Engineering:** Claude Code CLI (Opus 4.5)
- **Memory:** Neo4j (Graph Database) + LangChain Neo4j Integration
- **Infrastructure:** GCP Cloud Run, Cloud Build, Secret Manager

7.2 Phase 1: The Core Graph Architecture (LangGraph)

The application is structured as a graph where nodes represent operational states.

State Definition:

Python

```
from typing import TypedDict, List, Annotated
from langgraph.graph.message import add_messages
```

```python
class AlexState(TypedDict):
    messages: Annotated[List[dict], add_messages]
    intent: str
    complexity_score: float
    user_id: str
    memory_context: dict
    tool_outputs: dict
```

The Router:

The entry point is a conditional edge that evaluates the user's request.

```python
def route_request(state: AlexState):
    # Use Flash to classify intent
    classification = flash_model.invoke(f"Classify intent: {state['messages'][-1].content}")

    if classification.is_coding_task:
        return "architect_agent" # Escalates to Pro
    elif classification.requires_deep_memory:
        return "memory_retrieval_node"
    else:
        return "chat_agent" # Stays on Flash
```

7.3 Phase 2: The Memory Implementation (GraphRAG)

This phase involves setting up the Neo4j instance and the summarization pipelines.

Neo4j Schema Setup:

```cypher
// Create constraints
CREATE CONSTRAINT FOR (u:User) REQUIRE u.id IS UNIQUE;
CREATE CONSTRAINT FOR (d:Day) REQUIRE d.date IS UNIQUE;
// Time Tree structure
MATCH (y:Year)-->(m:Month)-->(d:Day)
```

The Summarization Worker:

A separate Cloud Run endpoint is created for the scheduler to hit.

Python

```python
@app.post("/tasks/summarize_daily")
def summarize_daily():
    # 1. Fetch yesterday's interactions from Neo4j
    interactions = graph.query("MATCH (d:Day {date: $date})<--(i:Interaction) RETURN i.text")

    # 2. Summarize with Flash
    summary = flash_model.invoke(f"Summarize these logs: {interactions}")

    # 3. Write Summary Node back to Graph
    graph.query("""
        MATCH (d:Day {date: $date})
        CREATE (s:DailySummary {text: $text})
        CREATE (s)-->(d)
    """, params={"text": summary.content})
```

7.4 Phase 3: The Engineering Interface

Implementing the safe wrapper for Claude Code.

Python

```python
import subprocess

def invoke_claude_code(prompt: str, working_dir: str):
    """
    Executes Claude Code in a subprocess, capturing output.
    """
    cmd = ["claude", "--print", prompt]

    try:
        # Run with timeout to prevent hanging
```

```
        result = subprocess.run(
            cmd,
            cwd=working_dir,
            capture_output=True,
            text=True,
            timeout=600 # 10 minutes max for coding tasks
        )
        if result.returncode == 0:
            return {"status": "success", "output": result.stdout}
        else:
            return {"status": "error", "error": result.stderr}
    except subprocess.TimeoutExpired:
        return {"status": "error", "error": "Operation timed out"}
```

7.5 Phase 4: Deployment Configuration

Cloud Build (cloudbuild.yaml):

This configuration defines the continuous deployment pipeline.

YAML

```
steps:
  # 1. Build Docker Image
  - name: 'gcr.io/cloud-builders/docker'
    args:

  # 2. Deploy to Cloud Run
  - name: 'gcr.io/google.com/cloudsdktool/cloud-sdk'
    entrypoint: gcloud
    args:
```

8. Future-Proofing: Voice and Multimodal Expansion

The user indicated "voice later." The current architecture is designed to accommodate this with minimal friction.

8.1 Audio Ingestion

Since Gemini 3 Flash accepts audio tokens directly, the "chat" interface on the frontend can be upgraded to record audio blobs. Instead of transcribing these to text (STT) before sending them

to the agent, the raw audio data can be encoded and passed directly to the model's API. This preserves paralinguistic cues (tone, emotion, urgency) that are lost in transcription, allowing Alex to respond with greater empathy and context.

8.2 Latency Considerations

Voice interfaces require low latency. While Cloud Run's "scale to zero" saves money, the "cold start" time (spinning up a container from zero) can introduce a 2-3 second delay, which is unacceptable for voice conversation.

- **Strategy:** When voice is enabled, the Cloud Run service configuration should be updated to set --min-instances 1. This keeps at least one container warm, ensuring instant response times, albeit at a slightly higher cost (approx. $30-$50/month depending on resource allocation).[19]

9. Conclusion

The architecture proposed for Alex represents a synthesis of modern agentic theory and pragmatic cloud engineering. By leveraging the **Gemini 3** family's tiered capabilities, we achieve a balance between high-frequency responsiveness and deep, complex reasoning. The **Neo4j-based Temporal Knowledge Graph** solves the critical problem of long-term, structured memory that vector databases cannot address, enabling the recursive summarization required for a coherent digital biography.

The infrastructure, built on **Google Cloud Run** and **GitHub**, provides a robust, scalable foundation that supports the unique requirements of the **Claude Code** engineering sub-agent. This design ensures that Alex is not just a passive tool, but an active participant in its own lifecycle—capable of introspection, self-correction, and autonomous evolution under human supervision.

This report serves as the definitive blueprint for construction. The next immediate step is the initialization of the GCP project and the provisioning of the Neo4j AuraDB instance to begin the "Day 1" ingestion of the CLAUDE.md constitution.

---

10. Comparative Analysis Tables

Table 1: Intelligence Model Tiering & Cost Analysis

Model Variant	Role	Capability Focus	Cost (Input / Output per 1M)	Trigger Condition

Gemini 3 Flash	**Basal Cortex**	Routine Chat, Summarization, Audio	**$0.50 / $3.00**	Default for all interactions.
Gemini 3 Pro	**Executive Cortex**	Complex Planning, Ambiguity Resolution	$2.00 / $12.00	High complexity score; Coding tasks.
Claude Code (Opus 4.5)	**Engineer Agent**	File System Ops, Testing, Refactoring	Varies (API driven)	Explicit "Code Change" intent.

Table 2: Memory Storage Solution Comparison

Feature	**Vector Database (e.g., Pinecone)**	**Tagging System**	**Temporal Knowledge Graph (Neo4j)**
Retrieval Mechanism	Semantic Similarity (Fuzzy)	Exact Keyword Match	**Structured Traversal & Semantic**
Temporal Reasoning	Weak (Metadata filtering only)	Weak (Linear lists)	**Strong (Explicit Time Tree)**
Recursive Summarization	Difficult (No hierarchy)	Impossible	**Native (Graph aggregation)**
Self-Knowledge	"Bag of Code Snippets"	N/A	**Full AST & Dependency Map**
Verdict	Secondary (Hybrid support)	Deprecated	**Primary Memory Engine**

Table 3: Infrastructure Selection Matrix

Component	Option A (Selected)	Option B (Discarded)	Rationale
Compute	**GCP Cloud Run**	Vertex AI Agent Builder	Cloud Run allows custom runtimes for Claude Code & Neo4j drivers.
Code Repo	**GitHub**	GCP Source Repos	GitHub has native integration with Claude Code CLI (gh).
Deployment	**GCP Cloud Build**	GitHub Actions	Tighter security integration with GCP Secret Manager & IAM.
Scheduling	**Cloud Scheduler**	Internal Python Loop	Decouples background tasks; allows "scale to zero" cost savings.

Works cited

1. How Much Does the Gemini 3 Flash Cost? Full Pricing Breakdown - GlobalGPT, accessed January 23, 2026, https://www.glbgpt.com/hub/how-much-does-the-gemini-3-flash-cost/
2. Google Gemini API Pricing 2026: Complete Cost Guide per 1M Tokens - MetaCTO, accessed January 23, 2026, https://www.metacto.com/blogs/the-true-cost-of-google-gemini-a-guide-to-api-pricing-and-integration
3. Gemini Developer API pricing, accessed January 23, 2026, https://ai.google.dev/gemini-api/docs/pricing
4. Gemini 3 Flash Preview – Vertex AI - Google Cloud Console, accessed January 23, 2026, https://console.cloud.google.com/vertex-ai/publishers/google/model-garden/gemini-3-flash-preview
5. Gemini 3 Flash | Generative AI on Vertex AI - Google Cloud Documentation, accessed January 23, 2026, https://docs.cloud.google.com/vertex-ai/generative-ai/docs/models/gemini/3-flash
6. Gemini 3 Developer Guide | Gemini API - Google AI for Developers, accessed January 23, 2026, https://ai.google.dev/gemini-api/docs/gemini-3
7. Introducing Claude Opus 4.5 - Anthropic, accessed January 23, 2026, https://www.anthropic.com/news/claude-opus-4-5
8. Claude Code CLI Cheatsheet: config, commands, prompts, + best practices - Shipyard.build, accessed January 23, 2026, https://shipyard.build/blog/claude-

code-cheat-sheet/
9. Vector Databases Are A Trap. Why “GraphRAG” Is The Only Way Forward. | by Ari Vance | Dec, 2025 | Medium, accessed January 23, 2026, https://medium.com/@AgenticAri/vector-databases-are-a-trap-why-graphrag-is-the-only-way-forward-a242b89a9bed
10. Comparison: RAG with Vector Databases vs. ArangoDB GraphRAG with Knowledge Graphs, accessed January 23, 2026, https://arango.ai/resources/comparison-rag-with-vector-databases-vs-arangodb-graphrag-with-knowledge-graphs/
11. Graph RAG vs vector RAG: 3 differences, pros and cons, and how to choose, accessed January 23, 2026, https://www.instaclustr.com/education/retrieval-augmented-generation/graph-rag-vs-vector-rag-3-differences-pros-and-cons-and-how-to-choose/
12. Modeling Agent Memory - Graph Database & Analytics - Neo4j, accessed January 23, 2026, https://neo4j.com/blog/developer/modeling-agent-memory/
13. Graphing Space and Time - Neo4j, accessed January 23, 2026, https://neo4j.com/blog/healthcare/graphing-space-time-michael-zelenetz-graphconnect/
14. Recursively Summarizing Enables Long-Term Dialogue Memory in Large Language Models, accessed January 23, 2026, https://arxiv.org/html/2308.15022v3
15. [2308.15022] Recursively Summarizing Enables Long-Term Dialogue Memory in Large Language Models - arXiv, accessed January 23, 2026, https://arxiv.org/abs/2308.15022
16. Create a Neo4j GraphRAG Workflow Using LangChain and LangGraph, accessed January 23, 2026, https://neo4j.com/blog/developer/neo4j-graphrag-workflow-langchain-langgraph/
17. Vertex AI Agent Builder | Google Cloud, accessed January 23, 2026, https://cloud.google.com/products/agent-builder
18. Building AI Agents with Vertex AI Agent Builder - Google Codelabs, accessed January 23, 2026, https://codelabs.developers.google.com/devsite/codelabs/building-ai-agents-vertexai
19. Cloud Run pricing | Google Cloud, accessed January 23, 2026, https://cloud.google.com/run/pricing
20. Running services on a schedule | Google Cloud Documentation, accessed January 23, 2026, https://docs.cloud.google.com/run/docs/triggering/using-scheduler
21. Schedule automated tasks to keep your pipelines happy | Google Cloud Blog, accessed January 23, 2026, https://cloud.google.com/blog/products/serverless/what-is-cloud-scheduler/
22. Claude Code: Best practices for agentic coding - Anthropic, accessed January 23, 2026, https://www.anthropic.com/engineering/claude-code-best-practices
23. Google Cloud Run cost me $4,676 in 6 weeks with zero traff | Hacker News, accessed January 23, 2026, https://news.ycombinator.com/item?id=46378065