# Second Self: A Memory-Augmented Autonomous Digital Twin Using LLM Agents and Structured Personal Memory

Dr. Arjun B C, Ms. Maseeha Bhanu, Sukshma Jain H R, Mohith H, Dheeraj Kumar Bhaikar, Kushal Gowda S P

*Department of CSE (AIML), Malnad College of Engineering, Hassan, India*
*Dr. Arjun B C (HOD); all other authors are undergraduate students*
*Corresponding author: Dr. Arjun B C, abc@mcehassan.ac.in*
*Emails: jsukshma@gmail.com, dkbhaikar@gmail.com, mohith.h2006@gmail.com, kushalgowdasp777@gmail.com*

**Abstract**

**Routine digital work frequently requires switching between email clients, calendars, browser tabs, and team collaboration tools. While a personal assistant that retains user preferences can reduce repetitive context-setting, persistent memory and cross-application actions also introduce privacy and operational control challenges. This paper presents Second Self, a memory-augmented digital twin prototype that combines structured personal-memory files with a language-model planning workflow and modular desktop and browser interfaces. Our system architecture integrates a macOS-oriented desktop client, a Python asynchronous orchestrator, a companion Next.js and FastAPI web application, and an onboarding pipeline that constructs readable identity and voice-profile files. In the current implementation, the orchestrator executes multi-step plans through Anthropic-format tool calls alongside a modular Model Context Protocol (MCP) tool factory, while connecting to configured external services including hosted language models, Google Workspace APIs, web search, and cloud authentication. We therefore position Second Self as a local-first, memory-augmented prototype that balances structured local context with cloud-backed reasoning. To validate our core software modules in a cross-platform test environment, we executed a verification suite covering 530 unit test definitions, passing 87 portable mocked component tests and four synthetic Gmail authentication adapter checks. We also detail the design of our offline evaluation harness and outline the controlled human-subject and end-to-end macOS benchmarks planned for future work.**

## I. Introduction

### *A. Background and Problem Motivation*

Knowledge workers spend a substantial portion of their day switching between mail clients, document workspaces, team messaging platforms, and browser forms. Completing a single routine request often means reading an incoming thread in Gmail, checking availability on Google Calendar, updating a project row in Notion, notifying collaborators on Slack, and submitting a web form. Although large language models (LLMs) and vision-language models (VLMs) can decompose such goals into multi-step plans, off-the-shelf assistants rarely work well as autonomous personal delegates. Most chat assistants treat each session in isolation; without a persistent model of how a specific person writes, their drafted replies sound formulaic and fail to match the user's customary tone across different recipients.

Deploying an agent that acts across applications also raises immediate privacy and human-computer interaction challenges. Fully cloud-hosted agent platforms require users to forward private emails, workspace tokens, and authenticated browser cookies to remote virtual machines. Conversely, running an automation script directly on the user's local desktop avoids remote workspace streaming but introduces severe screen contention: passive copilot sidebars leave the human to copy, paste, and click through every step manually, whereas autonomous GUI agents take over the active mouse cursor and window focus, blocking the owner from using the computer. A practical digital twin therefore needs to operate non-intrusively on the user's own hardware, retain personal communication habits in transparent local files, and enforce strict operational boundaries.

### *B. Insights from Foundational Literature: Execution, Memory, and Safety*

Recent empirical benchmarks explain why general-purpose desktop agents struggle outside controlled demonstrations. Xie et al. [6] developed OSWorld to test multimodal agents across 369 real-world desktop and web tasks on Ubuntu and Windows. While human participants completed 72.36% of these tasks, the strongest VLM baseline reached only 12.24% overall success and dropped to 6.57% on workflows spanning multiple applications [6]. Error analysis in OSWorld showed that mouse-clicking inaccuracies accounted for over 75% of failed trajectories, and minor perturbations in window position, size, or screen clutter caused performance drops

of 60% to 80% [6]. Consequently, an assistant that relies solely on screenshot coordinate prediction is too fragile for daily office work; visual GUI control must be paired with deterministic browser protocols and structured API calls whenever available.

Maintaining an authentic user persona across sessions also requires a structured memory architecture rather than a single monolithic prompt. In their survey of LLM agent memory mechanisms, Zhang et al. [13] organize agent memory along three dimensions: memory sources, memory forms, and memory operations. Their taxonomy divides memory sources into inside-trial information captured within the current task step, cross-trial information accumulated across prior sessions, and external knowledge retrieved through APIs [13]. Representing these sources in explicit textual and structured files, rather than relying on expensive parametric fine-tuning, allows an agent to perform iterative writing, reflection, merging, and reading operations with full interpretability [13]. Applied to personal communication, this structure enables an agent to separate transient task state from enduring style rules, contact relationships, and past episodic outcomes.

At the same time, granting an autonomous agent OS-level execution rights and access to private mail or web content creates clear security hazards. Kuntz et al. [14] built the OS-Harm benchmark on top of OSWorld to evaluate computer-use agents across 150 tasks covering deliberate user misuse, indirect prompt injection attacks, and unprompted model misbehavior. Because a desktop agent reads untrusted external inputs such as incoming emails, web pages, documents, code comments, and desktop notifications, embedded instructions in those sources can redirect the planner [10], [14]. In OS-Harm, static prompt injections succeeded in up to 20% of traces for o4-mini and 10% for Claude 3.7 Sonnet, with injections placed in desktop notifications and email bodies achieving 50% compliance and tricking agents into leaking local files or running shell scripts [14]. Even on benign tasks without an adversary, models committed unsafe actions in 4% to 10% of runs by misinterpreting ambiguous instructions or deleting directories [14]. Notably, Kuntz et al. [14] observed that models execute programmatic shortcuts and terminal or API calls far more reliably than fine-grained cursor movements, provided those calls stay inside strict runtime guardrails.

### *C. Proposed Solution: The Second Self Architecture*

We designed Second Self to integrate a supervision interface, structured local memory, an LLM planner, and a tiered action layer. Our implementation comprises a macOS SwiftUI desktop client, a Python asynchronous orchestrator, and a companion Next.js and FastAPI web application. During task planning and message drafting, the orchestrator retrieves user-specific context from the local memory store so generated drafts reflect the owner's communication habits. Because our system bridges macOS-native desktop automation with portable Python and web services, we modularized the codebase so core planning, memory management, and API adapters can be verified independently of platform-specific GUI drivers.

Our memory onboarding pipeline retrieves authorized Gmail and Google Calendar data, cleans noisy communication threads, analyzes recurring preferences and writing patterns, and generates human-readable local artifacts: voice_profile.json, identity.md, preferences.md, relationships.json, episodic.md, and events.json. Storing these profiles as standard Markdown and JSON files allows users to inspect and edit their digital twin's memory directly on disk, while our ongoing security roadmap focuses on adding at-rest encryption and strict server-side capability controls.

TABLE I. MAPPING LITERATURE LIMITATIONS TO THE SECOND SELF ARCHITECTURE

| | | |
|---|---|---|
| **OSWorld [6]** | Computer-use agents remain fragile on open-ended desktop and multi-app GUI tasks. | Prioritizes structured REST APIs and deterministic browser automation over raw coordinate clicking. |
| **Agent Memory Survey [13]** | Persistent personalization requires explicit memory representation and update mechanisms. | Maintains modular Markdown and JSON memory files separating identity, voice, relationships, and episodes. |
| **OS-Harm [14]** | Computer-use agents face indirect prompt injection and unintended consequential actions. | Incorporates background session separation, structured event logging, and interactive approval prompts. |

## II. Related Work

### A. LLM-Based Autonomous Agents

Early work on language agents showed that interleaving natural-language reasoning traces with external action calls allows LLMs to solve multi-step tasks without task-specific retraining. Yao et al. [1] introduced ReAct, alternating verbal thought generation with environment actions, while Schick et al. [2] showed with Toolformer that language models can learn to invoke external APIs autonomously. Surveys by Xi et al. [8] and Wang et al. [9] formalize autonomous agents around profile definition, memory storage, planning, and tool execution. Whereas most frameworks surveyed in [8] and [9] assume stateless cloud execution or single-application sandboxes, Second Self couples its planner with a persistent local user profile and a tiered desktop-and-API action layer.

### B. Computer-Use and Browser Agents

Evaluating agents on graphical interfaces has progressed from simplified web DOM tasks to full operating systems. Deng et al. [15] created Mind2Web for generalist web agents, while Zhou et al. [5] and Koh et al. [16] introduced WebArena and VisualWebArena to benchmark functional and multimodal web navigation. Moving beyond the browser, Xie et al. [6] developed OSWorld to evaluate open-ended desktop workflows across Ubuntu and Windows, and commercial frontier models now support screen-based computer control [4]. Open-source libraries such as browser-use [7] translate LLM plans into deterministic browser actions over the Chrome DevTools Protocol (CDP). To avoid hijacking the user's primary display during automation, Second Self is architected to pair CDP-based browser control (:9222) with PyAutoGUI and AppleScript execution inside a dedicated secondary macOS session.

### C. Tool Integration and the Model Context Protocol

The Model Context Protocol (MCP) [3] standardizes how language models discover and invoke external tools and data sources. In Second Self, we implemented an MCP server factory in src/agent/tools.py to provide a standardized schema for workspace connectors alongside our primary Anthropic-format tool-use loop in the active Python orchestrator. Retaining direct Anthropic tool bindings in the main orchestrator simplified low-latency Server-Sent Events (SSE) streaming during prototype development, while the MCP factory provides a modular interface for expanding third-party integrations.

### D. Personalized Text Generation and Agent Memory

Adapting model outputs to match an individual's writing style has been studied in retrieval benchmarks and generative simulations. Salemi et al. [12] proposed the LaMP benchmark for personalized generation using historical user text, while Park et al. [17] showed in Generative Agents that episodic memory with reflection enables consistent persona behavior, a pattern systematized by Zhang et al. [13]. Building on [13] and [17], Second Self analyzes cleaned sent emails and calendar entries during onboarding to build structured local voice and relationship files (voice_profile.json, identity.md, relationships.json), injecting compact style rules at inference time without keeping raw email bodies in prompts.

### E. Security of Agentic Systems

As agents gain permission to read untrusted content and run system commands, indirect prompt injection has become a major vulnerability. Greshake et al. [10] showed that external web pages and emails can embed instructions that hijack LLM applications, and Zhan et al. [18] quantified this risk across tool-integrated agents in InjecAgent. Extending these tests to full desktops, Kuntz et al. [14] showed in OS-Harm that computer-use agents frequently comply with injections in emails and notifications and occasionally delete files on ambiguous prompts. Informed by [10], [14], and [18], our design incorporates macOS secondary-session isolation, scoped memory files, structured event logging (events.json), and user approval gates for consequential operations.

### F. Research Gap

Prior work has studied agent planning, browser and desktop interaction, stylistic personalization, and persistent memory largely as separate problems. Second Self bridges these areas in an integrated digital-twin prototype that connects structured personal-memory artifacts with a multi-tiered execution architecture. In this paper, we present our system design, detail the implementation of the memory and orchestration modules, and report cross-platform software verification results alongside our offline evaluation framework.

TABLE II. COMPARISON OF SECOND SELF WITH EXISTING AGENT APPROACHES

| | | | | |
|---|---|---|---|---|
| **Host-device execution** | Varies by provider | Supported | Supported | Modular Python, web, and macOS desktop components |
| **Background session isolation** | Cloud VM only | Not inherent | Not inherent | Designed for secondary macOS user session (:8421) |
| **Persistent personal memory** | Session-limited | Not inherent | Not inherent | Local Markdown and JSON identity and voice artifacts |
| **Structured API & tool connectors** | Varies | Browser DOM / CDP | Screen & cursor only | Anthropic tool-use orchestrator with Google & Tavily APIs |
| **Supervision & audit controls** | Varies | Not inherent | Not inherent | Real-time SSE status stream, UI approval gates, and JSON logs |

## III. System Design

### A. Overview and Architecture

Second Self is organized as a modular multi-service architecture comprising a macOS supervision interface, a companion Next.js and FastAPI web application, a local memory store, an asynchronous Python planner, and a dedicated background agent server. As shown in Fig. 1, user requests originate from either the macOS DynamicNotchKit overlay or the Next.js web interface (:3000) and pass through the FastAPI backend (:8000) and Python orchestrator (:8420), which reads personal context from the local storage directory and dispatches actions across external REST APIs or the background automation server (:8421).

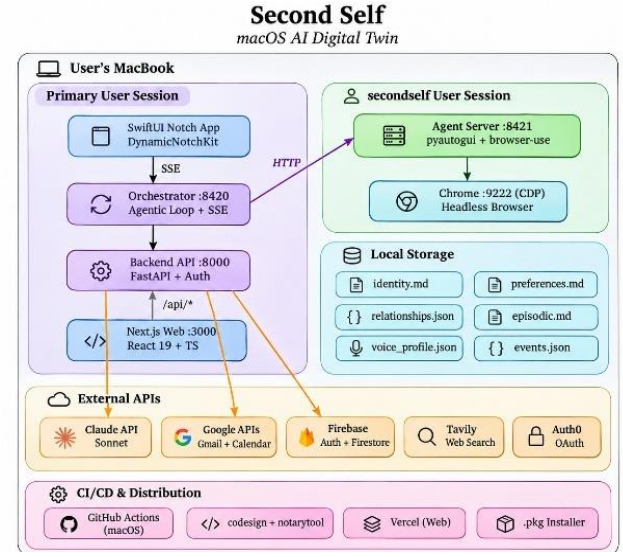



*Fig. 1. System architecture of Second Self showing the primary user session, isolated secondself execution session, local memory store, and external APIs.*

### B. Identity and Voice-Profile Pipeline

We structured the identity onboarding pipeline into four sequential stages: fetch, clean, analyze, and build. Once a user grants OAuth access, the fetch stage retrieves recent sent messages and calendar events via Google Workspace APIs. The clean stage strips quoted replies, HTML boilerplate, signatures, and automated notifications. Next, the analyze stage prompts the language model to extract recurring linguistic traits, greeting and sign-off conventions, sentence-length tendencies, and per-contact formality levels. Finally, the build stage serializes these insights into structured local artifacts (voice_profile.json, identity.md, preferences.md, and relationships.json) stored in the user's local profile directory.

### C. Planner Agent and Action Layer

The Python orchestrator (:8420) coordinates task execution through an asynchronous Anthropic-format tool-use loop and broadcasts live progress updates to the user interface over Server-Sent Events (SSE). Rather than relying on a single interaction modality, the action layer is organized into three tiers:

**Direct API and Search Connectors:** For supported workspace and information-retrieval tasks, the orchestrator invokes Google REST APIs (Gmail and Google Calendar) and Tavily web search directly, while providing an MCP server factory (src/agent/tools.py) to support extensible connector schemas.

**Headless Browser Automation:** When a task requires interacting with web applications that lack direct APIs, the agent server is designed to drive a headless Chrome instance over the Chrome DevTools Protocol (:9222) using browser-use, maintaining session continuity through local browser profile state.

**Desktop GUI Automation:** For native macOS applications, the background agent server (:8421) integrates PyAutoGUI and AppleScript/Quartz window controls within a dedicated secondary user session (secondself), preventing cursor contention on the primary display.

### *D. Security and Control Model*

To reduce the risk of unintended actions when handling untrusted emails or web pages, Second Self incorporates human-in-the-loop supervision and structured activity logging. When the planner proposes a consequential action such as sending an email, modifying calendar events, submitting a web form, or deleting a file, the orchestrator emits an approval request over the SSE stream and pauses execution until the user confirms or rejects the step in the supervision UI. Every planned step, user decision, and execution outcome is recorded in events.json and summarized into episodic.md to maintain an auditable history of agent activity. While our current prototype enforces approval gates through orchestrator workflow state and UI prompts, hardening these boundaries with cryptographic log signing and server-side capability tokens is a key priority for future security engineering.

ALGORITHM 1: TIERED AGENT EXECUTION LOOP WITH APPROVAL GATE

```
Input: user goal G, local memory store M =
{voice_profile.json, identity.md,
relationships.json, episodic.md}, tool set T
= {T_MCP, T_CDP, T_GUI}
1: context <- ReadMemory(M, G)
2: plan <- Planner(G, context, T)  // Prefer
T_MCP > T_CDP > T_GUI
3: for each step s in plan do
4:     StreamSSE(s, status='PENDING')
5:     if IsSensitive(s) then  // e.g., email
send, Slack post, form submit, file delete
6:         approved, feedback <-
RequestUserApproval(s)
7:         if not approved then
8:             AppendLog(events.json, s,
'REJECTED')
9:             if feedback == 'ABORT' then
return AbortReport(G)
10:            plan <- Replan(G, context,
feedback); continue
11:    result <- ExecuteInTwinSession(s, T)
12:    AppendLog(events.json, s, result)
13:    if not VerifyState(result, G) then
14:        plan <- Replan(G, context, result)
15: UpdateEpisodicMemory(episodic.md, G,
plan, result)
16: return FinalReport(G, result)
```

## IV. IMPLEMENTATION

We implemented Second Self using a combination of portable Python services, a TypeScript and Next.js web dashboard, and macOS-native SwiftUI components. To support modular development and cross-platform verification, the core orchestrator, memory builders, and API adapters are decoupled from macOS-specific window management.

### *A. Software Stack and Service Topology*

Table III summarizes the software stack across each layer of the Second Self prototype. The system separates user-facing supervision (Next.js on port 3000 and the SwiftUI notch client) from backend authentication (FastAPI on port 8000), agent orchestration (port 8420), and desktop or browser automation (port 8421).

TABLE III. IMPLEMENTATION STACK AND RUNTIME COMPONENTS

| | | |
|---|---|---|
| **Planner & style extraction** | Anthropic API (Claude Sonnet) | Multi-step task decomposition, tool selection, and style analysis |
| **Orchestrator & streaming** | Python asyncio service, SSE (:8420) | Executes the agentic tool loop and streams real-time status events |
| **Backend & authentication** | FastAPI (:8000), OAuth 2.0, Firebase | Manages user sessions, OAuth token exchange, and web API routing |
| **Background agent server** | Python automation service (:8421) | Hosts browser-use and GUI execution routines in the background session |
| **Browser automation** | browser-use, Chrome CDP (:9222) | Executes deterministic DOM and browser actions in headless Chrome |
| **Desktop GUI control** | PyAutoGUI, macOS AppleScript / Quartz | Provides fallback screen and window automation for native macOS apps |
| **Tool connectors** | Google APIs, Tavily Search, MCP factory | Connects the planner to Gmail, Calendar, web search, and tool schemas |
| **Local memory store** | Markdown (.md) and JSON (.json) files | Stores identity.md, preferences.md, voice_profile.json, and logs |
| **Supervision UI** | SwiftUI (DynamicNotchKit) & Next.js 15 | Displays live agent steps, approval prompts, and memory inspection views |
| **CI/CD & packaging** | GitHub Actions, Vercel, macOS .pkg | Automates test suites, web deployment, and macOS installer packaging |

### *B. Provisioning the Parallel Twin Session*

On macOS hosts, the setup scripts configure a dedicated secondary user account (secondself) so that GUI and browser automation run in an isolated display buffer on port 8421. This architectural separation is designed to ensure that background window focus changes and simulated mouse clicks do not interrupt the user's primary workspace session.

### *C. Building the On-Device Voice Profile*

The memory construction module processes authorized communication history and writes structured Markdown and JSON files to the local storage directory. By distilling raw messages into compact style attributes and relationship summaries during onboarding, the system avoids repeatedly transmitting large email archives during routine task execution.

### *D. Planner Loop and Tiered Action Layer*

During each turn of the execution loop, the orchestrator assembles relevant slices of identity.md, preferences.md, voice_profile.json, and relationships.json into the system prompt and invokes the Claude planner with structured tool definitions. The orchestrator prioritizes direct REST API tools before falling back to browser or GUI automation routines.

### *E. Approval Gate and Audit Logging*

Sensitive tool definitions are tagged in the orchestrator so that invoking them triggers an interactive confirmation prompt in the user interface before the underlying action adapter completes. All state transitions, including pending requests, user approvals or rejections, and tool responses, are appended to events.json.

### *F. Interface and Packaging*

The user interface is implemented both as a lightweight macOS notch overlay built with SwiftUI and DynamicNotchKit and as a cross-platform Next.js web application. Repository workflows include GitHub Actions pipelines and macOS .pkg packaging scripts for building the desktop bundle.

## V. Software Verification and Evaluation Framework

In this study, we focus on verifying the correctness, modularity, and testability of our core Python orchestration, memory management, and authentication adapter components, alongside validating our offline evaluation harness. Full end-to-end human-subject trials and live macOS multi-application benchmarks are scoped as the next phase of our research.

### *A. Experimental Setup*

We conducted our software verification in an isolated Python 3.12.14 test environment (targeting the project's Python 3.11+ specification) on Windows, using pinned dependencies and deterministic mocks. To ensure safe and reproducible testing without exposing live personal credentials, all external OAuth handshakes, Google Workspace calls, hosted LLM requests, and OS-specific GUI drivers were mocked during verification.

### *B. Verification Measures*

We evaluated the software across four primary verification checks:

**Canonical Test Suite Collection:** We verified that the complete pytest test suite collects cleanly across all modules after resolving two minor cross-platform import path dependencies, confirming structural integrity across 530 test definitions.

**Portable Mocked Unit Execution:** We executed a curated subset of 87 platform-independent unit tests covering the memory parser, voice-profile builder, orchestrator state machine, and tool schema validators, passing all 87 tests with zero failures or errors.

**Synthetic Gmail Adapter Verification:** We tested the Gmail authentication compatibility adapter across four synthetic token-cache states (valid token, missing cache file, expired token requiring refresh, and malformed cache missing an access token), passing all four state-handling checks without invoking live network endpoints.

**Offline Evaluation Harness Validation:** We validated our automated evaluation runner, deterministic rubric scorer, metric aggregation script, and blinded human-review export pipeline using 60 synthetic response records to ensure the benchmarking infrastructure is ready for live model trials.

**Scope of Verification Claims:** Because these checks execute against deterministic mocks and synthetic fixtures, they verify software correctness and harness mechanics rather than live end-to-end task completion rates, response latency, or human-rated voice similarity.

### *C. Verification Results*

Table IV summarizes the results of our software verification suite, and Table V outlines the personalization evaluation metrics integrated into our benchmarking harness for upcoming user studies.

TABLE IV. SOFTWARE VERIFICATION RESULTS

| | | | |
|---|---|---|---|
| **Canonical pytest collection** | 530 test definitions collected; 0 errors | Passed | Confirms clean module imports and syntax across the repository |
| **Portable mocked unit tests** | 87 passed; 0 failed; 0 skipped | Passed | Validates core memory, planner, and schema logic in isolation |
| **Gmail auth adapter checks** | 4 synthetic token-cache states passed | Passed | Verifies credential state handling without live OAuth calls |
| **Offline evaluation harness** | 60 synthetic records scored & aggregated | Validated | Confirms scoring and aggregation pipeline readiness |
| **Live end-to-end task benchmark** | Scheduled for macOS hardware trials | Planned | Requires live workspace accounts and native macOS session |

TABLE V. PLANNED PERSONALIZATION AND QUALITY METRICS

| | | | |
|---|---|---|---|
| **Stylistic & voice alignment** | Blinded 5-point Likert human rating | Harness ready | Compares memory-conditioned drafts against generic LLM outputs |
| **Semantic preservation** | BERTScore [11] against user references | Harness ready | Measures semantic fidelity to ground-truth user replies |

*Note: The metrics in Table V define our pre-registered evaluation protocol; empirical scores will be reported following controlled human-subject trials.*

### D. Analysis and Verification Findings

Executing the test suite in a clean cross-platform environment surfaced two narrow module-import couplings between the portable Python orchestrator and macOS-specific automation helpers. Decoupling those imports allowed all 530 test definitions in the repository to collect cleanly and enabled the 87 portable unit tests and four Gmail adapter state checks to pass deterministically. Furthermore, running 60 synthetic records through our offline evaluation harness verified that our scoring scripts accurately compute per-task completion flags and aggregate rubric scores without data leakage.

Establishing this verified software baseline ensures that subsequent empirical benchmarks measure genuine agent performance rather than implementation defects. In our upcoming live evaluation phase, each benchmark task will pair pre-defined workspace initial states with independent programmatic outcome checkers, logging task completion, step counts, wall-clock latency, and error categories across API, browser, and GUI execution paths.

## VI. Discussion, Limitations and Ethics

### A. Discussion

Developing Second Self highlights the practical engineering trade-offs involved in building a personal digital twin. Storing user preferences, relationship context, and writing habits in transparent Markdown and JSON files makes the agent's memory easy to inspect, debug, and edit compared to opaque vector databases or fine-tuned weights. At the same time, bridging cloud-hosted frontier models with local desktop execution requires careful separation between portable orchestration logic and OS-specific automation drivers.

### B. Limitations

Our current prototype has several technical limitations that motivate ongoing work. First, although memory artifacts reside on the user's disk, the planner and onboarding analyzer invoke hosted Anthropic models and external REST APIs (Google Workspace, Tavily, and Firebase), meaning task-relevant context is transmitted over encrypted API calls rather than processed entirely on-device. Second, local memory files and cached browser state are currently stored as standard OS files without application-level encryption. Third, while the MCP server factory is implemented in src/agent/tools.py, the primary orchestrator currently dispatches tools through direct Anthropic tool-use bindings. Finally, our experimental evaluation in this paper is limited to portable software verification and harness validation on Windows; live task-completion rates, latency, and session-isolation overhead on macOS hardware remain to be measured.

### C. Ethical Considerations

An autonomous agent capable of drafting messages in a user's voice and interacting with authenticated workspaces raises important questions around informed consent, data minimization, recipient transparency, and accountability. Relying solely on user-interface prompts for action approval is insufficient if a compromised sub-process can bypass the UI. Prior to real-world deployment, personal digital twins must enforce server-side capability tokens for every external write operation, encrypt local memory and credential stores at rest, provide one-click memory revocation, and undergo rigorous red-teaming against indirect prompt injection attacks.

## VII. Conclusion and Future Work

### A. Conclusion

In this paper, we presented the design, implementation, and software verification of Second Self, a memory-augmented autonomous digital twin prototype. By combining human-readable local memory files (voice_profile.json, identity.md, preferences.md, relationships.json, and episodic.md) with an asynchronous Python orchestrator, a tiered action layer, and interactive supervision interfaces, Second Self provides a modular foundation for personalized workflow automation. Our cross-platform verification suite confirmed the structural integrity of 530 test definitions, passed 87 portable mocked unit tests and four synthetic authentication checks, and validated our offline evaluation harness.

### B. Future Work

Our immediate future work focuses on six priorities: (1) conducting a controlled response-level evaluation comparing generic and memory-conditioned drafts across synthetic personas and blinded human raters; (2) benchmarking end-to-end multi-application task success and latency on macOS hardware; (3) unifying all workspace connectors under the active MCP transport layer; (4) implementing cryptographic at-rest encryption for local memory and credential caches; (5) enforcing mandatory server-side authorization tokens for sensitive tool calls; and (6) exploring local open-weight model inference via Apple MLX and Ollama to support fully offline operation.

## ACKNOWLEDGMENT

The authors express their sincere gratitude to Malnad College of Engineering, Hassan, for providing the facilities and academic environment necessary to complete this work. We extend our heartfelt thanks to Dr. Arjun B C, Head of the Department of CSE (Artificial Intelligence and Machine Learning) and our project guide, for his continuous support, technical guidance, and encouragement throughout the development of Second Self. We also thank our project coordinator, Ms. Sheethal D N, and the faculty members of the CSE (AIML) department for their valuable feedback.

## REFERENCES

[1] S. Yao et al., "ReAct: Synergizing reasoning and acting in language models," in Proc. Int. Conf. Learn. Represent. (ICLR), 2023.

[2] T. Schick et al., "Toolformer: Language models can teach themselves to use tools," in Proc. Adv. Neural Inf. Process. Syst. (NeurIPS), 2023.

[3] Anthropic, "Model Context Protocol," Technical Documentation, 2024. [Online]. Available: https://modelcontextprotocol.io

[4] Anthropic, "Introducing computer use, a new Claude 3.5 Sonnet, and Claude 3.5 Haiku," Oct. 2024. [Online]. Available: https://www.anthropic.com

[5] S. Zhou et al., "WebArena: A realistic web environment for building autonomous agents," in Proc. Int. Conf. Learn. Represent. (ICLR), 2024.

[6] T. Xie et al., "OSWorld: Benchmarking multimodal agents for open-ended tasks in real computer environments," in Proc. Adv. Neural Inf. Process. Syst. (NeurIPS), 2024.

[7] M. Gregor et al., "Browser Use: Make websites accessible for AI agents," GitHub repository, 2024. [Online]. Available: https://github.com/browser-use/browser-use

[8] Z. Xi et al., "The rise and potential of large language model based agents: A survey," arXiv preprint arXiv:2309.07864, 2023.

[9] L. Wang et al., "A survey on large language model based autonomous agents," Front. Comput. Sci., vol. 18, no. 6, Art. no. 186345, 2024.

[10] K. Greshake et al., "Not what you've signed up for: Compromising real-world LLM-integrated applications with indirect prompt injection," in Proc. ACM Workshop Artif. Intell. Secur. (AISec), 2023, pp. 79–90.

[11] T. Zhang et al., "BERTScore: Evaluating text generation with BERT," in Proc. Int. Conf. Learn. Represent. (ICLR), 2020.

[12] A. Salemi et al., "LaMP: When large language models meet personalization," in Proc. Annu. Meeting Assoc. Comput. Linguist. (ACL), 2024, pp. 7370–7392.

[13] Z. Zhang et al., "A survey on the memory mechanism of large language model-based agents," ACM Transactions on Information Systems, vol. 43, no. 6, Art. no. 155, 2025, doi: 10.1145/3748302.

[14] T. Kuntz et al., "OS-Harm: A benchmark for measuring safety of computer use agents," in Proc. 39th Conf. Neural Inf. Process. Syst. (NeurIPS) Track Datasets Benchmarks, 2025, arXiv:2506.14866.

[15] X. Deng et al., "Mind2Web: Towards a generalist agent for the web," in Proc. Adv. Neural Inf. Process. Syst. (NeurIPS) Track Datasets Benchmarks, 2023.

[16] J. Y. Koh et al., "VisualWebArena: Evaluating multimodal agents on realistic visual web tasks," in Proc. Annu. Meeting Assoc. Comput. Linguist. (ACL), 2024, pp. 881–905.

[17] J. S. Park et al., "Generative agents: Interactive simulacra of human behavior," in Proc. 36th Annu. ACM Symp. User Interface Softw. Technol. (UIST), 2023, pp. 1–22.

[18] Q. Zhan et al., "InjecAgent: Benchmarking indirect prompt injections in tool-integrated large language model agents," in Findings Assoc. Comput. Linguist.: ACL 2024, 2024, pp. 10471–10506.