AI-Powered Desktop Voice Assistants: A Review of Speech, Language-Model, And Autonomous-Agent Technologies with The MAYA (Multimodal Autonomous, Yielding, Assistant) Reference Architecture

Kshitij Surykant Kale[1], Prof. Dr. Taware Ganesh G.[2], Prof. Dr.Bere Sachin S.[3]
[1]*Dept. of Computer Engineering, Dattakala Shikshan Sanstha, Dattakala Group of Institutions, Faculty of Engineering, A/p Swami Chincholi, Tal. Daund, Dist. Pune 413130, India*
[2,3]*Associate Professor Dept. of Computer Engineering, Dattakala Shikshan Sanstha, Dattakala Group of Institutions, Faculty of Engineering, A/p Swami Chincholi, Tal. Daund, Dist. Pune 413130, India*

***Abstract*—Voice and natural language are among the most intuitive ways of instructing a computer, yet most desktop assistants remain limited to short question-and-answer exchanges and leave research, file processing, reminders, and desktop automation to separate tools. Advances in automatic speech recognition, large language models, speech synthesis, and tool-using agents now make it practical to combine these functions in a single application. This review examines how foundational work on speech recognition, language models, multimodal models, text-to-speech, and autonomous agents supports the design of a modular AI-powered desktop assistant. We synthesize nine foundational studies and tools and use MAYA, a Python-based desktop assistant, as a worked reference architecture. MAYA accepts typed, push-to-talk, or wake-word input; selects among Gemini Live, local model servers (Ollama, LM Studio, llama.cpp, vLLM), and compatible hosted APIs; and runs long tasks in a background engine with checkpoint-and-resume and a deep-research workflow that checks quotations against saved pages. Actions are reported as verified, unverified, or failed, and sensitive operations require user confirmation. We also propose a six-experiment evaluation protocol covering input recognition, intent accuracy, engine selection, task recovery, research output, and verified execution, and we identify five research gaps, including quantitative evaluation under noise, recovery of interrupted agents, and honest reporting of action results. The review rests on reported objectives and findings rather than reproduced experiments, and no performance figures are claimed for MAYA; actual performance depends on the configured providers, network availability, operating-system permissions, and computer hardware.**

***Index Terms*—AI voice assistant, speech recognition, large language models, text-to-speech, autonomous agents, deep research, desktop automation, persistent memory, MAYA**

I. INTRODUCTION

Artificial intelligence is changing the way people interact with computers. Conventional desktop use asks the user to type commands, search manually, and operate several applications one after another. An intelligent voice assistant can shorten these steps by letting the user speak or type in natural language and by acting on the request. Three families of technology have made this practical: neural speech recognition that converts audio to text [1], [2], large language models that interpret open-ended instructions and generate natural responses [3]–[5], and lightweight speech synthesis that can run on a local computer [8]. Most assistants nonetheless concentrate on short question-and-answer exchanges. Voice interaction, online research, file processing, reminders, and desktop automation are usually provided by separate tools, so the user still switches between applications and repeats steps by hand. Long tasks such as research and report preparation can block the conversation and are lost if the application is closed. Dependence on a single hosted AI service adds questions of privacy, cost, and internet availability. A further concern is honesty of reporting: an assistant that announces success for an action it has not checked can mislead the user, and sensitive operations such as shutdown or restart must remain under user control.

Agent-oriented research addresses part of this gap. ReAct interleaves reasoning with actions [6], and Toolformer shows that language models can learn to call external tools [7]. Long-running work, however, additionally requires progress tracking, checkpointing, error handling, and recovery so that interrupted tasks need not restart from the beginning. Safe automation requires controlled functions, permission checks, user confirmation for sensitive operations, and verification of the observed result.

This paper reviews the foundational literature behind such a system and examines it through a concrete reference design, MAYA, a Python-based AI-powered personal assistant for desktop computers. MAYA receives spoken or typed commands, interprets the request, selects an AI engine and a tool, performs a supported operation, and returns a response by text and synthesized speech. The scope is deliberately specific: a single-user Windows desktop assistant that can use Gemini Live or a separately configured combination of speech recognition, language model, and speech synthesis, with local model platforms (Ollama, LM Studio, llama.cpp, vLLM) or compatible hosted APIs.

Four research questions guide the review.

- RQ1: How do advances in speech recognition, language models, and speech synthesis support a modular voice-and-text assistant pipeline, and what limits remain?
- RQ2: How can interchangeable cloud and local AI engines be combined, and what trade-offs arise in privacy, cost, availability, and hardware demand?
- RQ3: How can long-running tasks such as research and report preparation run in the background and be recovered after an application restart?
- RQ4: How should tool execution be made safe and reported honestly, so that an action is announced as complete only when its result has been verified?

Sources were selected from foundational work on automatic speech recognition, natural language processing and large language models, multimodal models, text-to-speech, and tool-using agents, together with the open-source speech-synthesis engine used by MAYA and a standard reference text. Nine sources were examined, and their contributions were assessed from their stated objectives, methods, and reported findings. The contributions of this review are fourfold. First, it organizes the supporting literature around the functions of a desktop assistant. Second, it contrasts cloud and local processing paradigms. Third, it describes a reference architecture with a verification-aware execution layer and a checkpointed background task engine, and proposes an evaluation protocol. Fourth, it identifies five research gaps that can guide future work.

The remainder of the paper is organized as follows. Section 2 presents the technical foundations. Section 3 reviews the literature. Section 4 describes the MAYA reference architecture. Section 5 sets out the evaluation protocol, research gaps, and future directions, and Section 6 concludes.

II. FOUNDATIONS OF AN AI-POWERED DESKTOP ASSISTANT

2.1. Speech Input and Recognition

Automatic speech recognition (ASR) converts spoken language into text. Whisper uses neural models trained on a large amount of multilingual audio to recognize speech across the supported languages [1], and earlier work showed that deep neural networks greatly improved acoustic modelling [2]. Recognition quality nevertheless depends on the microphone, the language, the speaking style, and background noise. A desktop assistant must therefore combine a recognizer with a sensible activation method. In MAYA, typed text is always accepted, push-to-talk is activated with Ctrl + Space, and an optional local wake-word detector responds to the phrase “Hey Jarvis”. After a period without commands the assistant goes to sleep, and quiet hours can suppress activity altogether.

2.2. Language Understanding and Tool Selection

Natural language processing identifies the meaning and intent of a command. The Transformer architecture [3] enabled language models that interpret open-ended instructions, keep conversational context, and produce natural responses [4]; multimodal families such as Gemini extend this to images and audio [5]. A practical assistant combines such reasoning with defined tools and permission checks. A simplified formulation follows. Let x be the user input (text, or text obtained from speech recognition), c the context retrieved from memory, and T the set of available tools and plugins. A language model M with parameters θ gives the probability of choosing a tool

or a direct response t ∈ T ∪ {respond}, as in (1), and the selected action is the most probable one, as in (2).

$$P(t \mid x, c) = M(x, c; \theta) \quad (1)$$

$$t^* = \underset{t \in T \cup \{respond\}}{\arg\max} P(t \mid x, c) \quad (2)$$

After execution, the result r = t*(x) is checked by a verification function V, whose outcome is reported to the user, as in (3). The model parameters θ belong to pre-trained language, speech-recognition, and speech-synthesis models and are not trained in the system under review; the engineering contribution lies in integration, tool execution, verification, memory, background tasks, and the interface.

$$V(r) \in \{\text{verified}, \text{unverified}, \text{failed}\} \quad (3)$$

2.3. Speech Output

Text-to-speech (TTS) converts generated text into spoken output. Local engines such as Piper support offline speech generation for the available voices [8], whereas hosted services such as ElevenLabs provide streamed voice synthesis.

The selected engine affects voice quality, latency, and language availability, so the speech output stage should be configurable independently of the language model.

2.4. End-to-End Processing Pipeline

Figure 1 shows the methodology flow that follows from these components. Voice or text input is converted to text, if necessary, the intent and context are analysed, an AI engine and then a tool or agent are selected, the action or research task is executed, the result is verified, and the response is generated as text and, when enabled, as speech. Because the engine configuration is separate from the core logic, the background task model can be configured independently of the live voice engine, and a different AI provider can be selected without changing the rest of the application.

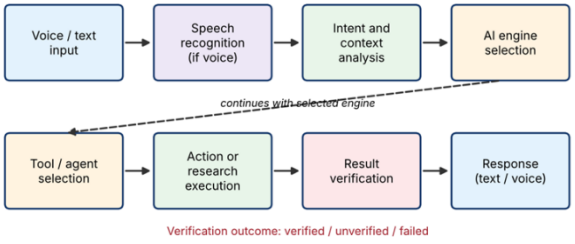

Figure 1. Methodology flow of a modular voice-and-text assistant: from input to verified response.

III. LITERATURE REVIEW

3.1. Speech Recognition and Synthesis

Table 1 summarizes the sources examined. Radford et al. train a recognizer on large-scale weakly supervised multilingual audio [1], building on the earlier finding of Hinton et al. that deep neural networks substantially improve acoustic modelling [2]. On the output side, Piper provides a fast local neural TTS system suitable for offline use [8].

A common limitation is that these works characterize recognition and synthesis in general; they do not specify how a desktop assistant should behave when the microphone, language, or noise conditions are poor, or when the hardware cannot support a local model.

Table 1. Foundational Studies and Tools Reviewed

Ref.	Focus	Reported contribution and scope note
[1]	Speech recognition	Whisper: large-scale weakly supervised multilingual recognition; quality still depends on microphone, language, style, and noise
[2]	Acoustic modelling	Deep neural networks greatly improved acoustic modelling for recognition; earlier foundation for neural ASR
[3]	Transformer	Attention-based architecture that enabled large language models
[4]	Large language models	Few-shot learners that interpret open-ended instructions, keep context, and generate natural responses
[5]	Multimodal models	Gemini model family extending language models to images and audio
[6]	Agents	ReAct: interleaves reasoning with actions; persistence of long tasks is outside its scope
[7]	Tool use	Toolformer: language models can learn to call external tools
[8]	Speech synthesis	Piper: fast local neural TTS enabling offline speech generation for available voices
[9]	Reference text	Textbook treatment of speech and language processing fundamentals

3.2. Language Models and Multimodal Models

The Transformer [3] is the architectural basis of today's large language models, and Brown et al. show that sufficiently large models can follow instructions from a few examples [4]. The Gemini family extends this capability to images and audio [5], which is relevant for assistants that accept live voice or camera input. Jurafsky and Martin provide the background on speech and language processing against which these developments can be placed [9]. For an assistant, the practical consequence is that one model can both understand a free-form command and decide whether a tool should be invoked, as formalized in (1) and (2). The reviewed works, however, do not address how the decision should be constrained by permissions or checked afterwards.

3.3. Autonomous Agents and Background Processing

Agent-based systems divide a complex goal into smaller steps and use tools to complete them. ReAct interleaves reasoning with actions [6], and Toolformer shows that language models can learn to call external tools [7]. These works establish how a model can plan and act. Long-running tasks, for example research, comparison, and report preparation, additionally require progress tracking, checkpointing, error handling, and recovery so that interrupted work does not restart from the beginning. Within the sources examined here, such persistence is an engineering requirement rather than a studied outcome, which motivates the background task engine described in Section 4.2.

3.4. Desktop Automation, Safety, Memory, and Routines

Operating-system interfaces and application APIs allow software to open applications, manage windows, and perform supported actions. Safe automation uses controlled functions, permission checks, user confirmation for sensitive operations, and verification of the observed result. Persistent memory stores useful context for later retrieval, and scheduling systems support reminders and repeated tasks; a unified routine engine can manage scheduled briefings, check-ins, monitoring, and quiet periods through one configurable mechanism. These areas are mostly covered by software platforms and engineering practice rather than by the foundational papers above, so their treatment in this review draws on the design of MAYA.

3.5. Processing Paradigms for Engine Selection

Table 2 contrasts the two processing paradigms that MAYA supports. Hosted engines such as Gemini Live and compatible APIs give access to capable models without local hardware demands but depend on a network connection.

Local model platforms (Ollama, LM Studio, llama.cpp, vLLM) allow processing on the user's computer but are limited by processor, memory, and storage. The comparison is qualitative and reflects architectural properties rather than measured figures; it explains why the user should be able to choose between them according to privacy, cost, and internet availability.

Table 2. Cloud and Local AI Engines for a Desktop Assistant (Qualitative)

Property	Hosted or cloud engine	Local model engine
Connectivity	Internet connection needed for cloud and API-based features	Local-model features can run without a network
Hardware demand	Light on the local computer	Depends on processor, RAM, and storage; insufficient hardware is a failure case
Privacy and cost	Depend on the chosen provider and its pricing	Processing stays on the computer; no per-request fee for the local model
Failure behavior	Network or API failure affects cloud-dependent features	Poor hardware headroom limits the selectable model
Role in MAYA	Gemini Live, OpenAI-compatible and other hosted APIs	Ollama, LM Studio, llama.cpp, vLLM

3.6. Synthesis

Three observations follow from the sources examined. First, the foundations for speech input, language understanding, and speech output are mature, but their quality still depends on the microphone, language, noise, and the selected engine [1], [8]. Second, agent research concentrates on reasoning and tool calling [6], [7], whereas durable progress, recovery after interruption, and checking of results are described as requirements rather than evaluated outcomes. Third, safety measures such as permission checks and user confirmation are well understood in principle but are seldom coupled with an explicit report of whether an action was verified. These observations motivate the reference architecture proposed next.

IV. MAYA REFERENCE ARCHITECTURE

4.1. Functional Blocks

MAYA is a modular desktop assistant that follows a sequence of input collection, speech or text processing, intent interpretation, model and tool selection, execution, and response generation. Background tasks and scheduled routines operate alongside the main conversation. Figure 2 shows the eight stages of the central column: (1) user input by voice, text, hotkey, or wake word; (2) input processing by speech-to-text, NLP, and intent detection; (3) a task planner that chooses an action or agent; (4) selection of an engine and tools; (5) execution as background tasks or actions; (6) response generation as text, speech, and actions; (7) presentation through voice, the heads-up display (HUD), and notifications; and (8) a continue decision, in which a new command repeats the cycle and otherwise the assistant goes to sleep through auto-sleep or quiet hours.

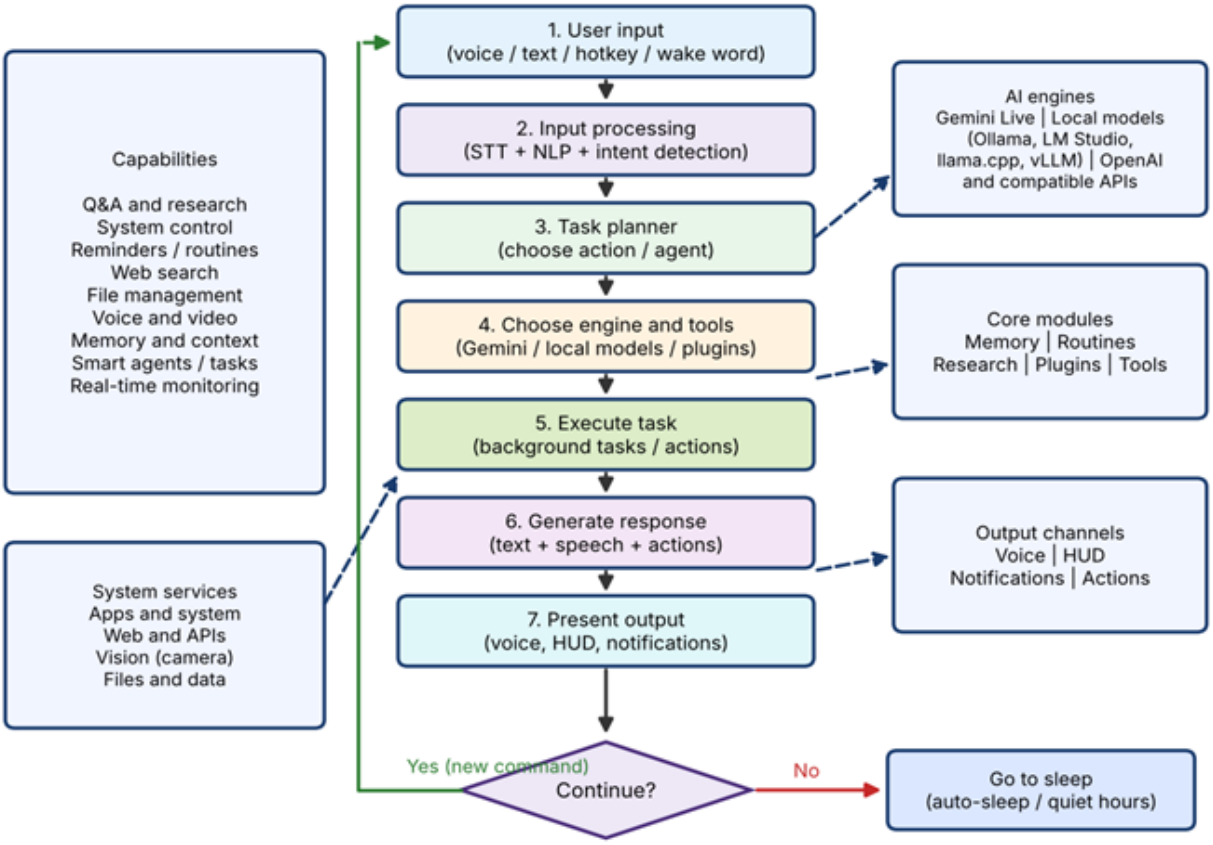

Figure 2. MAYA system architecture and flow diagram. Side panels show capabilities, AI engines, core modules, system services, and output channels.

The side panels show the supporting components. The capabilities include answering questions and research, system control and automation, reminders and routines, web search, file and data management, voice and video interaction, multi-language support, memory and context, smart agents and tasks, and real-

time monitoring. The AI engines are Gemini Live, local models, OpenAI-compatible APIs, and others such as Groq. The core modules are memory, routines, research, plugins, and tools; the system services are applications and the operating system, web and APIs, vision through a camera, and files and data. The assistant core is a Python application with a modular plugin system, and each plugin performs a defined operation and returns an outcome. The tools and platforms are listed in Table 3.

Table 3. Software Stack and Hardware of MAYA with Design Notes

Component	Role	Design note
Python application	Assistant core and plugin system	Engine configuration is kept separate from core logic
Gemini Live, hosted APIs	Live voice engine and cloud models	Needs an internet connection
Ollama, LM Studio, llama.cpp, vLLM	Local language models	Limited by processor, RAM, and storage
Whisper	Local speech recognition	Quality depends on microphone and noise
Piper, ElevenLabs	Local and streamed speech synthesis	Affects voice quality, latency, and languages
OS scheduling and automation interfaces	Application control, routines	Runs within permissions granted to the application
Computer (Windows)	Host platform	Multi-core CPU; 8 GB RAM or more recommended for local models
Microphone, speakers, optional webcam	Voice input, output, visual input	Required for voice interaction

4.2. Background Task Engine and Deep Research

A user can assign a long-running task while continuing the conversation. The task engine splits the work into steps and stores the result of each step as a checkpoint, as sketched in Figure 3, so that completed steps are retained and unfinished work resumes from the last saved step after an application restart. The deep-research workflow in Figure 4 plans questions, gathers and reads source material, saves the pages, checks quotations against the saved pages, and prepares a cited report. Separating the background-task model from the live voice engine allows a slower but more capable model to be used for research without delaying the conversation.

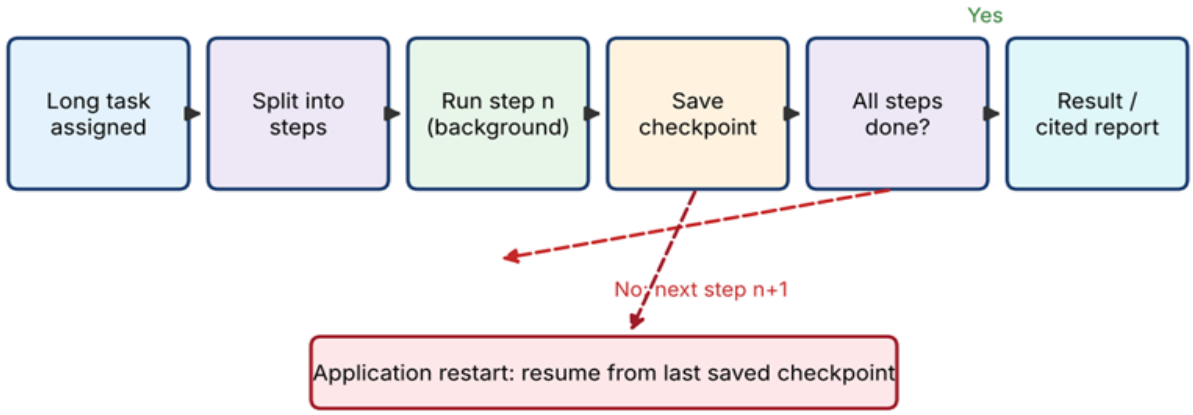

Figure 3. Background task engine with checkpointing and recovery after an application restart.



Figure 4. Deep-research workflow producing a cited report with quotation checks.

4.3. Verified Action Reporting and Safety

After interpreting a request, MAYA selects a capability such as web research, file processing, application control, reminder management, or a direct conversational response. The function executes within the permissions available to the application, and the tool result is checked where possible. The assistant distinguishes a verified completion, an unverified attempt, and a failure, as in (3), and reports an action as complete only when verification confirms it. Sensitive operations such as shutdown and restart require user confirmation, and API keys are kept out of source code and shared files. If we write A(t) for the availability of the required permission, S(t) for a sensitive action, and C(t) for user confirmation, the execution rule can be stated as (4). This is a formalization of the behavior described above rather than a measured property.

$$\text{execute}(t) \Leftrightarrow A(t) \wedge \left(\neg S(t) \vee C(t)\right) \quad (4)$$

Failures are expected and must be handled rather than hidden: a network or API failure affects cloud-dependent features, recognition degrades in noisy environments, local models may exceed available hardware, and missing operating-system permissions block automation. In each case the assistant should inform the user instead of reporting false success.

4.4. Memory, Routines, and Command-Centre Interface

Relevant stored information is searched and added to the active context to support continuity between sessions. The routine engine manages one-time and repeating schedules, startup activities, quiet-period check-ins, and background monitoring, and users can modify supported routines through natural-language instructions. The command-centre HUD presents glass-style panels, live gauges, a transcript log, an audio waveform, a task and routine overview, and a software-rendered reactor core, while an animated avatar provides facial expressions and mouth movement synchronized with speech.

V. EVALUATION PROTOCOL, RESEARCH GAPS, AND DIRECTIONS

5.1. Evaluation Protocol

Accuracy figures are to be reported only after measurement on the final build, and none are claimed here. Table 4 lists six experiments, each tied to a research question. Voice and text input are tested under normal and noisy conditions, and four quantities are defined. The input recognition rate (5) is the share of correctly recognized inputs among all tested inputs; the intent accuracy (6) is the share of correct intent predictions among all test commands; the execution success rate (7) is the share of successfully verified tasks among all execution attempts, with unverified attempts and failures recorded separately; and the average response time (8) is the mean interval between submitting a command and receiving a response, which varies with network speed, model latency, speech processing, and computer performance.

$$\text{Input recognition rate} = \frac{\text{Correctly recognized inputs}}{\text{Total tested inputs}} \times 100 \quad (5)$$

$$\text{Intent accuracy} = \frac{\text{Correct intent predictions}}{\text{Total test commands}} \times 100 \quad (6)$$

$$\text{Execution success rate} = \frac{\text{Successfully verified tasks}}{\text{Total execution attempts}} \times 100 \quad (7)$$

$$\text{Average response time} = \frac{\sum_{i=1}^{N} T_i}{N} \quad (8)$$

Table 4. Proposed Experimental Protocol

Exp.	Measurements	Answers
E1	Voice and text input: input recognition rate under normal and noisy conditions	RQ1
E2	Intent and tool selection: intent accuracy over a set of test commands	RQ1
E3	Cloud versus local engine on the same tasks: response time and success, on target hardware	RQ2
E4	Background task recovery: start a multi-step task, restart the application, record lost or repeated steps	RQ3
E5	Research output: structured report, citations, and quotation checks against saved pages	RQ3
E6	Task execution: verified, unverified, and failed outcomes; confirmation of sensitive actions	RQ4

Overall, implemented functions are tested individually. Voice input and speech output are checked under normal operating conditions, API-dependent features are tested with a working network

connection, local-model features are tested on the target hardware, and experimental or unavailable functions are clearly identified. This plan allows the assistant to be evaluated without assuming unmeasured accuracy or performance.

5.2. Research Gaps

Five gaps follow from the review; they are the authors' synthesis of the sources examined and of the design of MAYA. G1, quantitative evaluation: integrated assistants are rarely reported with measured recognition, intent, and execution rates under noisy and quiet conditions and with more users and environments. G2, recovery of interrupted agents: lost or repeated steps after a restart are not commonly measured. G3, honest result reporting: separating verified, unverified, and failed outcomes is seldom made an explicit evaluation criterion. G4, engine trade-offs: cloud and local engines are not usually compared on the same task set with response time and hardware demand. G5, language and activation coverage: support for additional Indian languages and the accuracy of wake-word detection need study. Table 5 maps each gap to a measurable response in Table 4.

Table 5. Research Gaps and Proposed Responses

Gap	Observation	Proposed response
G1	Few measured results for integrated assistants under noise and across users	E1, E2 on a larger and more varied test set
G2	Recovery of interrupted background tasks seldom measured	E4: checkpoint and restart test
G3	Verified, unverified, and failed outcomes seldom reported separately	E6: outcome-level reporting
G4	Cloud and local engines rarely compared on one task set	E3: same tasks, both engine types
G5	Indian-language support and wake-word accuracy need study	Extend E1 to additional languages and wake-word tests

5.3. Future Directions

Several enhancements are possible. Smart-home integration could connect compatible IoT devices to control lights, appliances, and sensors through authenticated commands, and a mobile application could give remote access to the assistant, task status, reminders, and selected desktop functions. Advanced multi-agent collaboration could divide research, analysis, coding, and report writing among specialized agents. Improved offline intelligence would support more local language and speech models, subject to processor, memory, and storage, and enhanced visual understanding could add scene understanding and object detection while respecting camera permissions and privacy.

Stronger security and privacy are also needed, including user authentication, permission profiles, encrypted memory storage, and improved API-key protection. Further directions include cross-device synchronization of selected tasks, routines, and settings; an expanded plugin ecosystem for productivity tools, calendars, and messaging platforms with user consent and error handling; finer-grained permission profiles for automation; and a packaged installer so that the assistant can be deployed easily on other computers. A larger quantitative evaluation with more users and environments, together with feedback from users, should guide iterative improvement.

5.4. Limitations of the Review

Two limitations apply. The corpus is small, consisting of nine foundational sources, and was assembled by targeted selection rather than a registered protocol, so relevant studies may be missing; contributions were assessed from reported objectives and findings, which means implementation details were not independently verified. In addition, the reference architecture is described from its design, no measured performance of MAYA is reported, and actual performance depends on the configured providers, network availability, operating-system permissions, and computer hardware.

VI. CONCLUSION

This review examined foundational work on speech recognition, language models, multimodal models, speech synthesis, and tool-using agents to assess how well it supports a modular AI-powered desktop

assistant. The foundations for speech input, language understanding, and speech output are mature, and interchangeable cloud and local engines give users flexibility in privacy, cost, and availability. Durable background execution, recovery after interruption, and honest verification of action results are less well documented. We therefore described the MAYA reference architecture, in which a checkpointed background task engine, a deep-research workflow with quotation checks, user confirmation for sensitive actions, and verified, unverified, or failed reporting are combined in one application, and we proposed a six-experiment protocol and identified five research gaps. Because the sources were assessed from their reported objectives and findings and no performance figures are claimed for MAYA, the conclusions should be read as a map of the field; experimental validation on the final build is the natural next step.

CONFLICTS OF INTEREST:
The authors declare no conflict of interest.

REFERENCES

[1] A. Radford, J. W. Kim, T. Xu, G. Brockman, C. McLeavey, and I. Sutskever, "Robust speech recognition via large-scale weak supervision," in *Proc. Int. Conf. Machine Learning (ICML)*, 2023.
[2] G. Hinton *et al*., "Deep neural networks for acoustic modeling in speech recognition," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 82–97, 2012.
[3] A. Vaswani *et al*., "Attention is all you need," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.
[4] T. Brown *et al*., "Language models are few-shot learners," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.
[5] Gemini Team, Google, "Gemini: A family of highly capable multimodal models," *arXiv preprint* arXiv:2312.11805, 2023.
[6] S. Yao *et al*., "ReAct: Synergizing reasoning and acting in language models," in *Proc. Int. Conf. Learning Representations (ICLR)*, 2023.
[7] T. Schick *et al*., "Toolformer: Language models can teach themselves to use tools," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.
[8] Rhasspy Project, "Piper: A fast, local neural text-to-speech system," open-source software, GitHub repository.
[9] D. Jurafsky and J. H. Martin, *Speech and Language Processing*, 3rd ed., draft, Stanford University.