Beyond the Interface: AI Integration Through Input Stream Mediation and Intelligent Output Simulation

Divij H. Patel
Independent Researcher, Ontario, Canada

***Abstract*—As Natural Language Processing (NLP) technologies continue to advance, their integration within everyday software tools remains fragmented and often constrained by environment-specific plugins or proprietary AI interfaces. This study introduces a lightweight, platform-independent framework that transforms any text-input surface, ranging from simple editors to browsers and full-featured integrated development environments (IDEs), into an intelligent, AI-assisted interface. Leveraging keystroke dynamics and key-chord recognition, the system operates unobtrusively in the background to interpret user intent and enable real-time interaction within active applications. It provides context-aware suggestions, completions, and insights directly within the active window, effectively turning ordinary typing environments into responsive, intelligent companions. Beyond unifying AI support across diverse applications, the framework enables users to seamlessly switch among multiple open-source AI and NLP models via simple key combination triggers, supported through flexible API integration, thereby providing dynamic access to different linguistic capabilities on demand. The architecture also incorporates a protective intelligence layer that detects suspicious behavior patterns and safeguards sensitive information, offering an additional shield against unauthorized data exposure. By employing a universal mediation layer for input and output, supported by controlled keystroke simulation, the approach eliminates the need for custom extensions or application-specific integrations, presenting a scalable model for embedding real-time artificial intelligence into everyday computing environments.**

Keywords*—Keystroke dynamics; keystroke simulation; AI-Assisted Interfaces; open-source AI; key-chord recognition; API integration; real-time interaction systems

I. Introduction

Over the past decade, the artificial intelligence (AI) ecosystem has experienced significant growth in the development of open-source AI and Natural Language Processing (NLP) models made available through well-documented APIs. These APIs facilitate integration into software ecosystems using widely adopted programming languages, thereby enabling AI-driven functionalities across a broad spectrum of applications [1]. Despite these advancements, interaction with such models has largely remained confined to proprietary interfaces, including standalone web portals, dedicated desktop applications, or platform-specific plug-ins. This architectural dependency often forces users to switch between environments to access AI capabilities, resulting in fragmented workflows and diminished productivity.

Recent AI-assisted interaction systems have primarily integrated NLP-based functionalities through application-specific graphical interfaces, browser extensions, or embedded plug-in architectures. While these solutions enable task-oriented assistance such as grammar correction or development support, they remain dependent on platform-level integration and typically require manual configuration for each environment. Consequently, such implementations offer limited interoperability across heterogeneous computing platforms and do not support unified AI-assisted interaction across multiple text-input surfaces.

Additionally, existing research has largely emphasized improvements in model performance or domain-specific deployment of NLP systems, with limited focus on system-level interaction frameworks that enable application-independent AI mediation. Most current solutions rely on proprietary interfaces or tightly coupled API integrations confined to specific software environments. In contrast, the proposed framework introduces a platform-independent architecture that enables real-time AI-assisted interaction across conventional text-based interfaces without reliance on model-specific plug-ins or environment-dependent integrations.

To address this limitation, this study proposes a hybrid architecture comprising three primary components: real-time keystroke monitoring, virtual keystroke simulation, and asynchronous NLP inference over open-access APIs. The proposed framework introduces a lightweight background service that operates independently of application-level interfaces and continuously monitors user input within text-based environments. Upon detecting predefined trigger conditions, such as a hotkey sequence, the captured input is transmitted to an open-source NLP model via an API endpoint, and the generated response is re-inserted into the active application using simulated keystrokes. This mechanism enables interface-independent AI interaction within commonly used applications, including standard text editors such as Notepad and WordPad, as well as embedded terminal environments [2].

Recent developments have attempted to mitigate this limitation through AI-powered browser extensions and application-level integrations tailored for specific use cases. Tools such as grammar correction engines and development assistance environments exemplify this trend toward embedded intelligence. However, these solutions remain tightly coupled to individual platforms or applications and typically require manual installation and configuration for each environment. As a result, users must rely on multiple independent tools for tasks such as grammar correction, paraphrasing, and code analysis, leading to inconsistencies in user experience and limited

interoperability between applications [3]. Existing literature primarily focuses on improving model performance or interface-level integration, with comparatively limited attention given to system-wide interaction frameworks capable of providing application-independent AI assistance.

From a systems perspective, modern operating environments interpret keyboard input through standardized scan-code interpretation and virtual key mapping mechanisms based on established character encoding standards [4]. When a user initiates a keypress event, the keyboard hardware generates a scan code that is interpreted by the operating system and mapped to a virtual keycode corresponding to the active keyboard layout [5]. In addition to capturing real-time keyboard input, contemporary operating systems provide low-level APIs capable of generating synthetic input events through functions such as keybd_event() and SendInput(), thereby enabling the simulation of keystrokes programmatically [6], [7]. These virtual input events are processed by the system kernel in the same manner as physical keystrokes, allowing applications to receive them as standard user input without requiring interface-specific modifications.

Concurrently, the emergence of transformer-based NLP models such as GPT-2, BERT, and spaCy has enabled access to powerful text-generation capabilities trained on large-scale linguistic corpora [8], [9]. These models are typically accessed through graphical interfaces or dedicated applications, necessitating environment switching for interaction. Although RESTful API endpoints allow developers to embed AI functionalities within software applications using languages such as Python or JavaScript, such integrations remain application-dependent and do not provide a unified mechanism for system-wide interaction [10], [11], [12]. This highlights a critical research gap in the development of interface-agnostic AI mediation frameworks capable of operating seamlessly across heterogeneous text-input environments.

The generalized framework eliminates the need for model-specific plug-ins by introducing a system-level layer of AI augmentation that decouples the input source from the NLP interface. This enables context-aware text generation across diverse computing environments while minimizing friction between users and AI-assisted tools. Furthermore, the proposed architecture is scalable to non-traditional text-editing platforms, including online content editors, live-stream overlays, and integrated development environments such as Visual Studio. In such scenarios, user input is intercepted by the background service, processed by an NLP engine, and the generated output is seamlessly appended within the same interface without reliance on third-party extensions [13]. This approach provides real-time assistance while reducing cognitive overhead associated with environment switching.

ANSI escape sequences and virtual terminal protocols form the foundational communication layer of the proposed system, given their widespread use in character encoding, network messaging, industrial control systems, and embedded device scripting. By incorporating AI-driven inference, secure API interactions, and automated keystroke simulation within ANSI-based workflows, the proposed framework advances existing interaction paradigms and introduces new opportunities for intelligent and secure human–computer interaction.

The primary contributions of this work are summarized as follows:

- The design of an interface-agnostic AI mediation framework for real-time interaction across heterogeneous text-input environments.
- A hybrid system architecture integrating keystroke monitoring, keystroke simulation, and asynchronous NLP inference via open-access APIs.
- A system-level AI augmentation layer that eliminates dependency on application-specific plug-ins or proprietary interfaces.
- A scalable implementation capable of enabling context-aware AI assistance within conventional text editors and development environments.

The subsequent Methodology section details the implementation of this framework and its integration within standard computing environments.

II. Methodology

This proposed functionality can be implemented in any modern programming language, such as Python, JavaScript, or C++. Once the main thread starts, which is responsible for continuously monitoring keystrokes and checking for matching key combinations to start and stop the child loops and threads [2]. These child threads can be programmed for different operations, which will be covered in the rest of the study. For example, they might send input to NLP and AI agents, trigger security events, and train existing models for behavior biometrics and typing patterns. These threads can be started/stopped through special key combinations matched via the main thread. Many of the programming flows of the child threads are described in the upcoming sections.

A. Legacy System Foundation and Transition to Intelligent NLP Framework

As previously discussed, the proposed framework aims to offer system-level natural language augmentation by combining open-source NLP models with traditional keystroke-capture techniques. In its initial prototype phase, the system used a multidimensional-array-based string-matching algorithm that provided a deterministic, hardcoded mapping between user input and predefined responses [2]. This is again not a GUI version; it provides a terminal-independent interface that simulates a virtual input/output environment using ANSI escape sequences and raw device file access.

If we see this program in the terminal, the captured keystrokes' ASCII values and converted string look like Fig. 1.

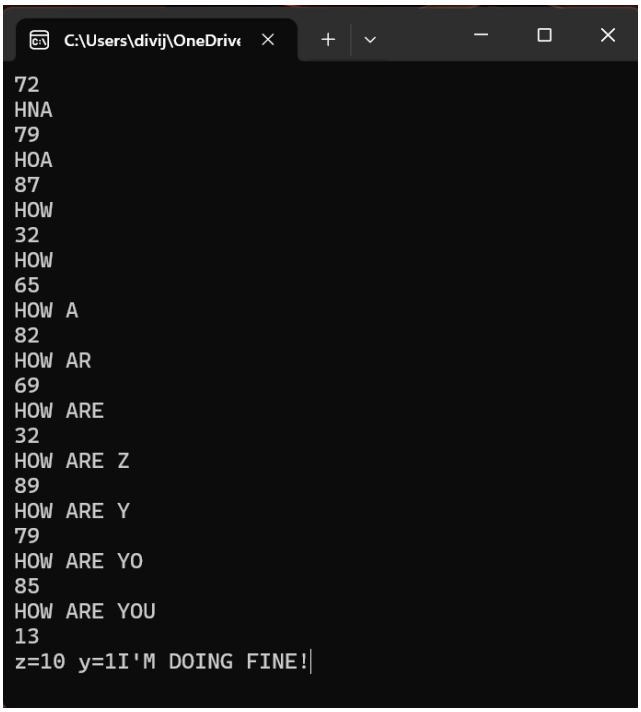

Fig. 1. Converting captured ANSI to text.

When a user enters a key event on the keyboard, such as "How are you doing?", the operating system's built-in module captures it and reads it using a built-in function of a programming language mentioned in section I. Once the keystroke is captured, like the ASCII number "72", the program converts it to the character 'H' (hex 48). It continues capturing and forming sentences until it reads the code "13", which is for the "enter key", indicating the sentence is complete. It then sends this data to either an NLP system or a conventional array-matching loop, which generates the virtual keystroke for the response, such as "I'M DOING FINE!" or whatever reply is received from the NLP server, by triggering the corresponding ASCII numbers in a specific sequence.

This low-level interaction model kept the system non-intrusive by running it in the background without disrupting everyday keyboard-based user actions. Behind-the-scenes control was implemented by custom hotkey combinations that trigger pause, resume, and terminate actions, as mentioned in the study [2].

By monitoring mode, the author refers to maintaining continuous availability of the proposed AI-assistance framework; the system must remain active in the background, independent of user interaction. This persistent operation can be achieved directly in Python and C++ through long-running program structures, as well as through operating system-level tools that support background execution. Modern operating systems provide native service managers, such as systemd on Linux and Windows Services on Microsoft platforms, which are specifically designed to oversee long-running programs, automatically restart them when needed, and ensure they stay active throughout system operation. These service frameworks enable the system to function reliably as a background component, even when no user interface is active [21], [22].

Within the program itself, Python programming language supports persistent background operations via an infinite event loop, asynchronous execution via the asyncio library, and thread-based concurrency via the threading module. These mechanisms enable continuous monitoring, periodic polling, and non-blocking communication with external NLP models. Likewise, C++ provides a rich background-execution model through its standard threading library (std::thread), enabling developers to create long-lived detached threads or asynchronous tasks (std::async) that persist independently of the main application flow. When combined with OS-level service management, these programmatic approaches ensure that the framework behaves as a resilient, always-active process capable of delivering real-time AI support across heterogeneous environments [23], [24].

The current iteration marks a significant architectural shift toward context-aware behavior by incorporating transformer-based NLP models such as BERT [8], GPT-2 [9], and spaCy [12]. These integrations allow for probabilistic, semantically coherent outputs rather than rigid string retrieval, dramatically improving the framework's versatility and scalability.

B. Proposed System Architecture and Operational Workflow

The config file, created during installation, sets up the framework's runtime behavior by defining how NLP endpoints, credential-based access, and linked hotkey connections are enabled. It indicates what actions to take when the key combination is triggered. Similar to file configuration formats, its style is consistent with .env or YAML, with each entry specifying an NLP model endpoint.

This configuration file is also encrypted (using the OpenSSL AES-256 symmetric encryption method) and can be decrypted only by the process in question, which does not compromise the integrity of the sensitive API tokens or service credentials [14]. This encryption strategy ensures confidentiality and integrity in multi-user or shared environments.

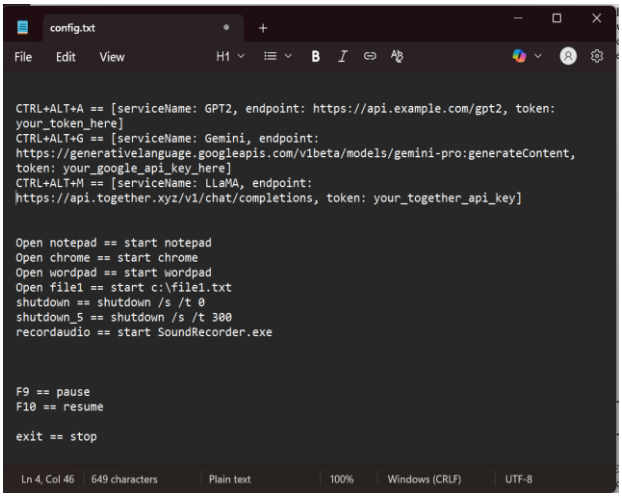

Fig. 2. Config file example.

Fig. 2 shows what the config file looks like when opened. It contains many predefined key combinations, and its system commands are already set for use. The developer can add more or modify the existing ones as needed.

As we discussed, the main thread will continuously check for the hot key combination and, if none are found. It will then generate keystroke-based output from a predefined, updateable array of sentences hardcoded into the program. If the user presses a combination to activate an NLP model, such as CTRL+ALT+A, the main thread spawns a child thread that sends registered keystrokes to the selected NLP model and returns the response by generating virtual keystrokes. Similarly, a user or developer can change the NLP model's preference by pressing a key combination to switch reactions in real time.

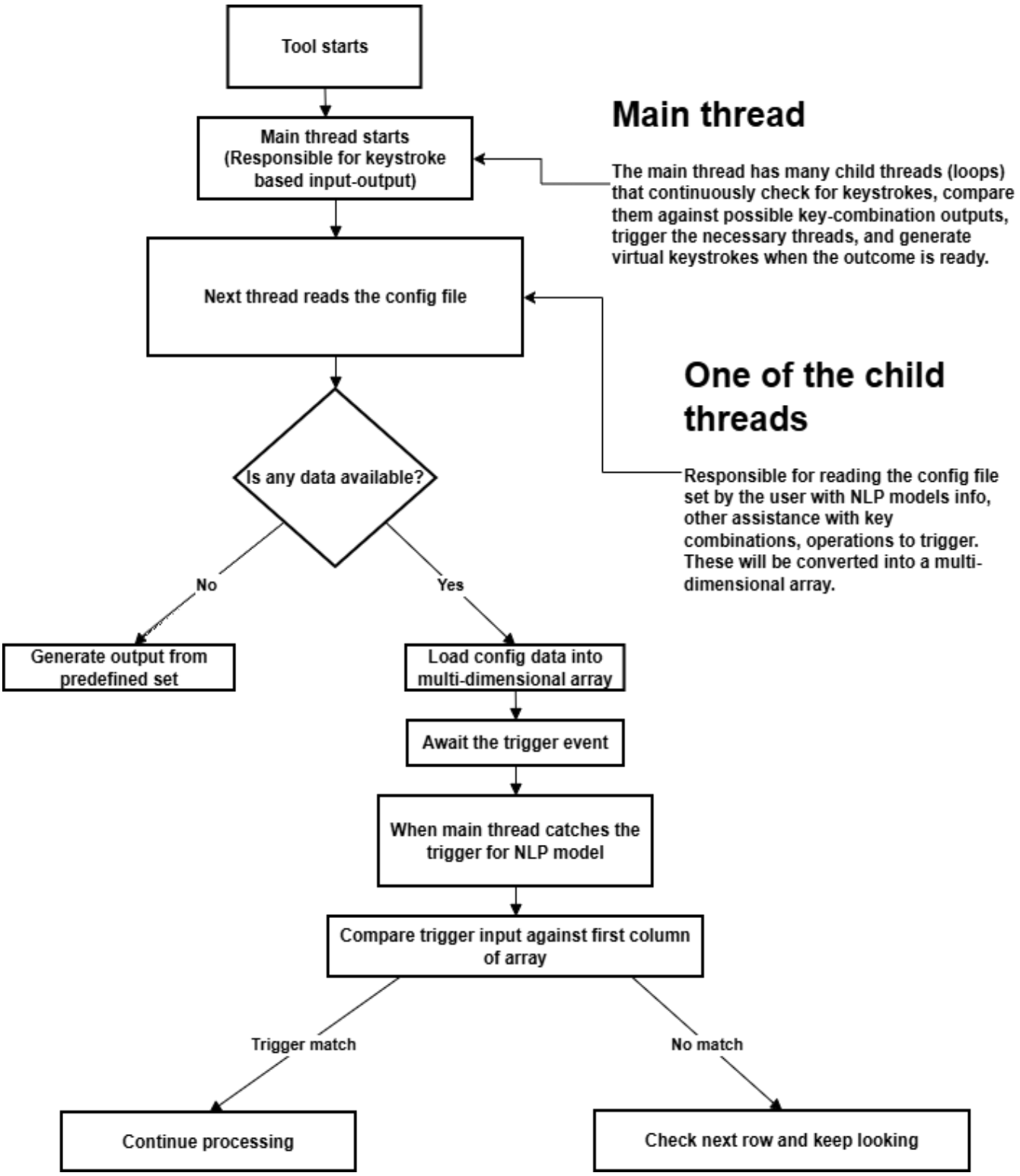

Fig. 3. Switching between NLP models.

Upon detecting a predefined key sequence, the framework executes the following logic pipeline:

1) Event handling and dispatch: A global key listener registers the triggering hotkey and asynchronously parses the associated configuration entry.

2) Secure API invocation: Using Axios (JavaScript) or libcurl (C++), depending on the platform context, a structured POST request is sent to the specified model endpoint, embedding the API token within the Authorization header [15], [16].

3) Response handling: The received NLP model output is parsed and sanitized, then passed to the UI injection engine, which uses virtual keystroke APIs (e.g., Windows SendInput, Linux uinput) to simulate direct text input into the active application window.

4) State management: The framework maintains a lightweight in-memory buffer to cache the last N responses, enabling rollback, regeneration, or comparison functionality within the interaction loop.

The above sequence enables real-time language augmentation across any text-editable interface, whether it be an IDE, browser, email client, or command shell. Unlike monolithic desktop NLP tools, this system features a modular, microservice-based design that supports plug-and-play integration with additional models or downstream services, such as spellcheckers or summarizers.

C. Technical Contributions and Deployment Considerations

1) Hotkeys binding logic: Hotkey binding is a crucial feature of modern operating systems. It enables users to activate specific actions through key combinations, boosting efficiency and workflow. In this work, the researcher has expanded on this traditional concept by employing keystroke-level event interception as an interaction layer for real-time AI assistance. This results in a more flexible and context-aware experience within operating systems. This system has been implemented using Xlib on Linux and SetWindowsHookEx on Windows. It supports dependable key-chord recognition and includes mechanisms to prevent race conditions, ensuring stable and predictable shortcut-event handling across platforms [18].

2) Cross-platform API wrappers: Native bindings to libcurl, along with Axios wrappers for Electron-based environments, enable seamless cloud communication.

3) Error handling: Includes retry logic with exponential backoff and token refresh middleware to maintain long-session stability with external NLP providers. If the retry fails, it automatically generates the appropriate error response and types it via virtual keystroke at the cursor position. The current tool does not have its own GUI (Graphical User Interface). It is crucial to generate proper error codes and display them at the cursor point to ensure clarity for users and developers.

4) Platform abstraction: Layer enables portability across Windows, Linux, and macOS by abstracting system-specific keystroke injection mechanisms. In this study, the author has defined a concept that can be implemented in any preferred programming language and on any operating system.

This design ensures high extensibility, allowing for future integration with on-device LLMs or fine-tuned domain-specific models using frameworks such as Hugging Face Transformers [19]. The following flowchart Fig. 3, illustrates the overall process and operational flow of the proposed framework.

Upon initialization, the Main thread starts and continues to look for the special key combinations and other keystrokes. For example, if the user presses the enter key after some keystrokes, it will be captured and processed as a text-based prompt.

Another child thread accesses a local configuration file through standard read/write operations, enabled by the host operating system's POSIX-compliant filesystem layer. The initialization routine first performs a structural integrity check on the configuration file. If the file is found to be empty or malformed, the system defaults to legacy mode, in which outputs are selected from a hardcoded static array using deterministic logic, as discussed in Section II. Conversely, when valid user-defined entries are detected, the framework parses the data upon receiving the NLP model's matching key combination. It maps it into a three-dimensional array, designated as a[x][y][z], where x indicates the number of configurations, y represents individual data fields (such as hotkey, endpoint, API token), and z specifies the character buffer allocated for each field.

For example, a configuration entry such as:

CTRL+ALT+G == [serviceName: GPT2, endpoint: https://api.example.com/gpt2, token: your_token_here]

Is stored in the array as follows:

- a[0][0]: Hotkey identifier (CTRL+ALT+G)
- a[0][1]: Model name (GPT2)
- a[0][2]: API endpoint URL
- a[0][3]: Authentication token

This data is subsequently referenced during runtime to dispatch model-specific API requests in response to user-defined hotkey events.

In addition to array-based data management, the system supports a modular configuration model that leverages JavaScript Object Notation (JSON) for serialization and deserialization [20]. JSON offers interoperability across multiple programming languages and ensures backward compatibility with web services. This dual-format support, with array-based low-level access and JSON for high-level abstraction, enhances the system's portability and extensibility across diverse environments.

Designed to operate in headless mode, the framework's main thread continuously intercepts and processes keystroke data from the hardware abstraction layer (HAL) without relying on graphical elements. Memorable hotkey sequences, such as CTRL+ALT combined with an alphanumeric trigger key, are intercepted via a low-level keyboard hook. If no special key is

pressed, input is evaluated against a static input-response map (Section II). When a trigger combination is detected, the framework performs a sequential scan through the array index a[x][1] to locate the appropriate service descriptor. Upon a successful match, it retrieves the associated serviceName, endpoint, and token, thereby enabling dynamic model switching.

This design enables users to switch effortlessly between multiple NLP backends (such as GPT-2, BERT, spaCy, and Google Gemini) by simply editing the hotkey configuration file and activating the relevant key combination. The modular and extensible nature of this configuration-based approach not only simplifies deployment but also facilitates real-time inference management across various AI engines.

Once an appropriate Natural Language Processing (NLP) model is selected and configured, the system enters a passive monitoring mode, continuously running in the background to intercept and log the user's keystrokes. The framework uses a low-level keystroke listener, often implemented via OS-level hooks or input stream interception (e.g., SetWindowsHookEx in Windows or XRecord in Unix-based systems) [17]. This enables the system to monitor input events without interfering with the user interface or application foreground processes.

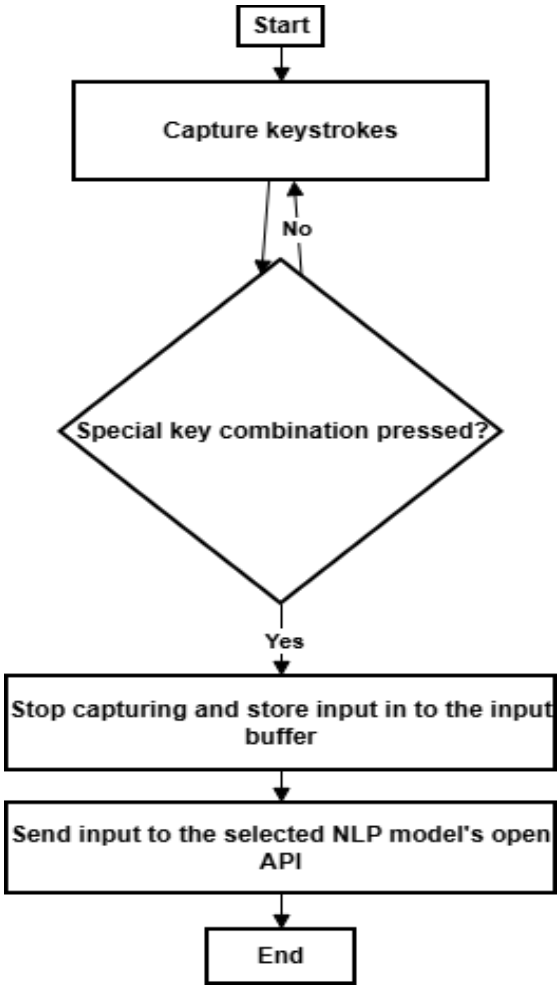

Fig. 4. Send keystrokes to NLP model.

As shown in Fig. 4, the captured string input is transmitted to the selected NLP model endpoint via an OpenAPI-compliant HTTPS request that includes an authentication token to validate the session. These requests are typically handled using lightweight client libraries, such as Axios for JavaScript or libcurl for C++ environments, which support asynchronous communication and robust error handling [15], [16].

The raw answer from the NLP model's inference is presented to the user, and the text is normalized and post-processed. During this step, control characters, special Unicode encodings, and formatting artifacts (such as escape sequences or HTML tags) are discarded to ensure the output conforms to human-readable formatting principles. These changes may be performed using natural language preprocessing libraries or custom sanitization routines to improve the clarity and readability of the output text [25].

After receiving the model's response, the framework dynamically injects the generated output into the user's active application window, using programmatic virtual keystroke generation at the cursor to render the results without modifying the application itself, as shown in Fig. 5.

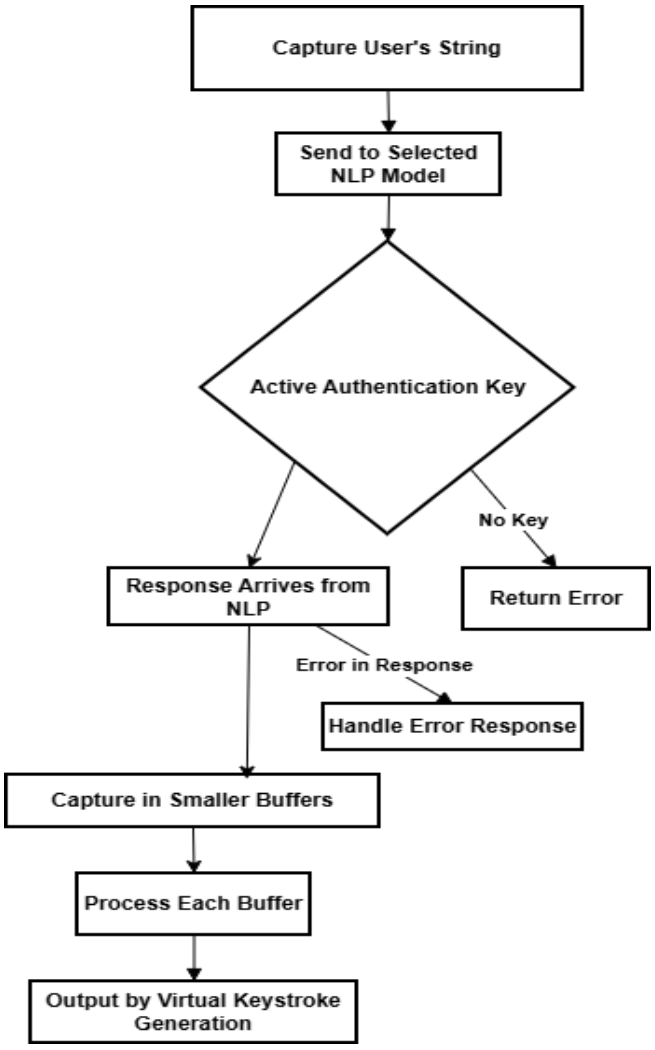

Fig. 5. NLP response to virtual keystroke.

To demonstrate in a real-world scenario, the researcher has simulated an interaction between a user and the GPT-4 model in a standard text editor such as Notepad. Once the program is initialized, users can start typing as usual. The framework remains locally held in memory, capturing all provided written prompts via capturing keystrokes, converting them into a string, and feeding them into the chosen NLP engine. The output is then fetched and formatted via the system and reinserted into the editor as it happens. The generated text is shown below in Notepad, providing both a context-aware interaction experience and an intuitive experience between the traditional input method and modern AI-based language models.

To illustrate the practical implementation of this feature, a scenario is presented in which a user interacts with OpenAI's GPT-4 model via a conventional text editor such as Windows Notepad. After the framework is initialized, it enters a persistent resident mode and operates as a background process. As the user types, the system captures keystroke sequences, processes them as input prompts, and dispatches them asynchronously to the selected NLP endpoint.

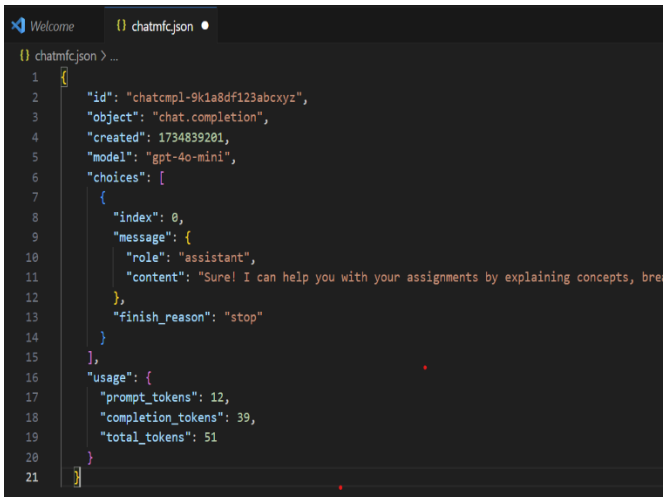

Fig. 6. NLP JSON response to Open-AI.

Upon receiving the model's inference, the framework formats and injects the generated text back into the editor in real time via synthetic keystroke generation. The Fig. 6 shows the JSON response from the NLP server.

Subsequently, the framework triggers a keystroke injection routine using platform-specific input simulation libraries (e.g., Win32 API SendInput) [3]. The output is programmatically written back into the user's active application window, preceded by the ">>" symbol to denote system-generated text. Upon completion of the full response rendering, the system appends a final status marker:

This marker serves as a visual delimiter to indicate the end of model-generated output, enhancing usability and separating AI responses from subsequent user input. In Fig. 7, the NLP response was directly key-stroked in Microsoft Windows Notepad.

A similar result can be achieved using any of the available platforms, such as Microsoft Word, sticky notes (as shown in Fig. 8), in-browser live chat, the Visual Studio Code editor, and many more.

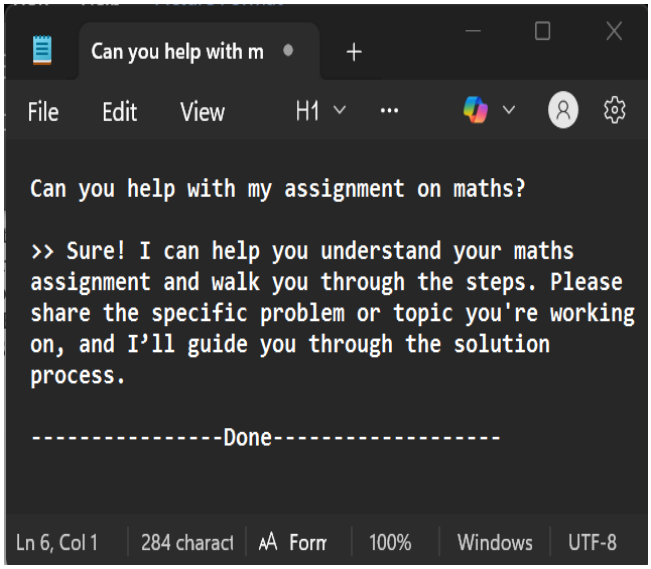

Fig. 7. NLP response in Notepad.

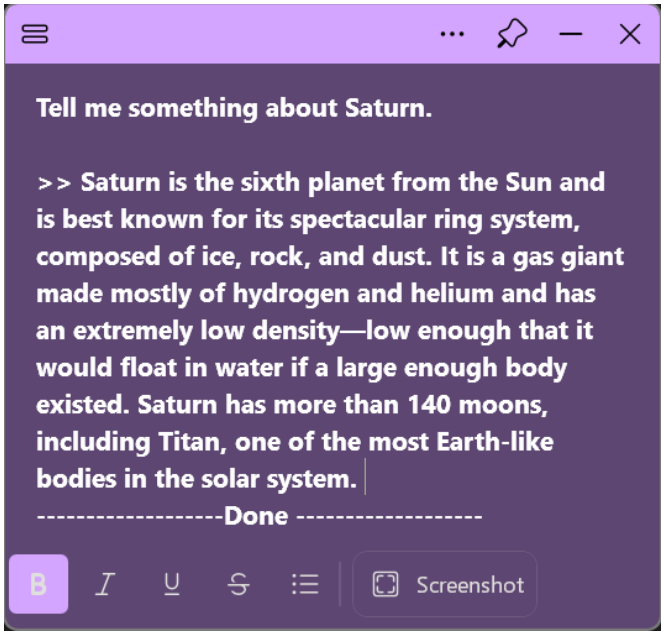

Fig. 8. NLP response in sticky notes.

Similarly, it works with any available text editor, even a browser. The user does not necessarily need to open a text editor; pressing keys on the keyboard can be registered and produce a response at the cursor point.

III. Multi-AI Orchestration via An Interface-Less Model

Within the interface-independent integration framework, you can run these heterogeneous AI assistants in parallel, combining the strengths of various natural language processing (NLP) models and task-specific intelligent agents into a single operational pipeline. Unlike existing NLP engines that focus on context-aware, text-based answers to users' queries, other AI

assistants enable procedural, action-oriented automation. These particular agents can undertake deterministic computations like intelligent scheduling, habit and routine orchestration, multi-calendar synchronization, and workflow optimization, using rule-based reasoning and machine learning-backed event prioritization [26].

By coordinating these capabilities without the constraints of a graphical interface, the system facilitates seamless interoperability, enabling composite AI-driven solutions that extend beyond conversational interaction into fully automated operational ecosystems. Several intelligent personal assistants have been developed in recent years, including OpenAI's ChatGPT [27], Google Gemini [28], Microsoft Copilot [29] and Amazon Alexa [30], Each is leveraging advanced artificial intelligence and natural language processing to interact with users and perform tasks efficiently.

A. Conventional Method to Assist the User

Earlier implementations have incorporated predefined, task-oriented commands to facilitate user interaction in general-purpose computing settings [31]. Available literature presents batch scripting techniques and Windows Command Prompt (CMD) utilities for performing context-specific operations, including opening applications, accessing files, and controlling the system [32]. This capability was implemented using a keyword-to-command mapping schema, in which user-entered input tokens were programmatically matched to shell commands stored in a persistent configuration file (as described in Section 1.2).

For example, the command mapping file may contain entries such as:

Open notepad == start notepad

Open chrome == start chrome

Open wordpad == start wordpad

Open file1 == start c:\file1.txt

shutdown == shutdown /s /t 0

shutdown_5 == shutdown /s /t 300

recordaudio == start SoundRecorder.exe

A user types a command string like "open chrome" into some interface, which the system captures the keystroke stream, parses the first keyword ("chrome"), and searches the mapping file for a corresponding match. The shell command retrieved is then run through the "system()" function of C++ and passed to the operating system's native command processor [33]. Such a methodology enables both immediate (in real time) and delayed (e.g., initiating a shutdown after 300 seconds upon detecting shutdown_5) operations.

In addition to launching applications, it supports auxiliary batch script utilities for hardware-level interfaces, like generating audio/video recording subsystems within Windows operating systems. For example, when RecordAudio is run, SoundRecorder.exe is launched to capture sound for a fixed period. This utility can be used differently depending on the operating system and programming language.

This traditional assistance framework expands its scope to include security-critical automation, enabling it to trigger keystrokes for specific protective actions. As shown in the authors' prototype, threat-detection keywords are stored in the command mapping file. When typed, either intentionally by the owner or accidentally by an unauthorized user, the system activates a security protocol that captures audio and video streams, encrypts or locks private directories, and executes a coordinated shutdown [34], [35]. Such mechanisms provide a low-latency, interface-independent safeguard against potential data breaches by leveraging event-driven command-execution pipelines without requiring GUI intervention.

B. AI Model-based Assistance in One Place

As outlined in Section I, state-of-the-art Natural Language Processing (NLP) models have been successfully implemented to provide domain-specific services across various application sectors [1], [2]. Building on the traditional keystroke-capture-and-trigger method discussed earlier, the framework combines multiple AI-driven service modules into a single, interface-independent environment. This design allows users to access various AI Assistance-based features simultaneously without being limited by application-specific GUIs or platform constraints. [2].

While NLP models primarily focus on generating semantically coherent responses to text-based queries, AI assistance agents are typically trained and configured to execute context-aware actions within browser environments, operating systems, or hybrid computing ecosystems [36]. For example:

1) Motion: An AI-driven scheduling engine for dynamic task reprioritization and meeting rescheduling based on real-time availability constraints [37].

2) Clockwise: A temporal resource optimization system designed to protect focus time and reduce meeting fragmentation [38].

3) Reclaim.AI: A buffer-creation and automated rescheduling assistant for time-block management [39].

4) Google: Workspace-based automation tools for integrated email handling, calendar synchronization, and cross-application task execution [40].

Whilst these tools will each provide a specific functionality in a proprietary or web-based environment, Open APIs (Application Programming Interfaces) offer a standardized path for integration. In the proposed system, API credentials and endpoint configurations are kept in encrypted configuration files (described in Section II). An identification keystroke triggers each AI assistant. The system captures an event when the user enters a pre-defined keystroke sequence, resolves the mapping to the corresponding API call, and transmits the request payload to the assistant's open API endpoint over HTTPS using RESTful API protocols [15], [16].

Direct task orchestration doesn't require an intermediary GUI. For example, typing in the command:

"Send a Thank You email to xyz at 13:00"

would push the corresponding AI assistant API to schedule execution, achieving prompt execution with asynchronous

processing guarantees and secure API key or OAuth 2.0 token authentication [41].

This centralized AI model orchestration framework combines multiple automation pipelines, such as scheduling, communication, and system-level operations, using a unified, low-latency, event-driven command-processing architecture. This utility can be set up in the same way as other commands. For example, the system utilizes AI-assisted scheduling via an open API to automate email scheduling. All it needs is to set up the user's email and other prerequisites. Then, this keystroke-based command can be sent to the open API of the AI assistance, and it will schedule the email [32].

The system enables the currently running AI assistant to schedule email transmission at a specified time using keystroke input. Another example is ordering dinner or telling the assistant to order a pizza from a particular restaurant in a specific size at a specific time. Now, using the method described in Section I, the User can switch the AI assistance model with key strokes and use different assistance services in real time, just like NLP models.

There are several AI assistants available that offer open APIs to help developers utilize them programmatically. Each assistant is specialized and already trained to perform different tasks efficiently. Details for these open APIs can be added to the settings file and easily triggered and used with a single keystroke combination.

To demonstrate the practical use of the proposed framework beyond productivity tasks, consider automating a food delivery request, like ordering a pizza. Usually, this process involves users navigating a vendor's website or mobile app, selecting items, entering delivery information, and completing the payment, or, if using AI assistance, opening their application or website, writing the prompt and details, waiting for the order response, and then switching back to their working window. With an AI assistant integrated via Open API endpoints, these steps can be performed programmatically with minimal user effort. After selecting the appropriate model via a combination of keystrokes, the user can type the prompt anywhere, and it will be executed.

In practice, the system leverages natural language parsing and intent recognition models to capture the user's request. For instance, a keystroke trigger such as:

"Order a large pepperoni pizza with extra cheese from Domino's at 7:30 PM."

Would be intercepted by the framework's command mapping module and mapped to the corresponding Domino's API (or an intermediary food delivery platform API such as Uber Eats or DoorDash. An AI assistance model will do that by itself in the background.

The AI agent translates this natural language request into a structured JSON payload containing attributes such as:

- restaurant_id (e.g., Domino's)
- item_id (e.g., large pepperoni pizza)
- customization (e.g., extra cheese)
- delivery_time (e.g., 19:30 hours)
- payment_token (securely stored OAuth 2.0 or PCI-DSS-compliant token).

The payload is then sent via a RESTful POST request to the vendor's ordering API. Once authenticated and validated, the system sends a confirmation response that the AI assistant may display to the user as a text notification. Then it can be generated at the user's cursor point via keystroke generation. (e.g., "Your pizza order has been scheduled for 7:30 PM").

From a systems perspective, this workflow illustrates that task-oriented AI assistance not only automates systems but also extends to delivering real-world, consumer-facing services. The same event-driven orchestration pipeline, keystroke capture, intent parsing, API mapping, response handling, and so on, simplifies automation of productivity tasks (such as calendar scheduling) and lifestyle tasks (like food delivery), highlighting the versatility and extensibility of the proposed system architecture [42], [43].

IV. Leveraging NLP in the Present Interface-Free System

A. Evolution from Static Mapping to AI-Driven Interaction

The ability of this model to adopt a specific pattern has been a consideration in earlier models, which explored rule-based mappings in which a predefined set of user inputs produces deterministic outputs, and this was the starting point of the proof-of-concept work [2]. Early systems in the field, while limited by scalability, were a good way to demonstrate keyboard-controlled automation of user assistance. Based on the framework mentioned above, this research encompasses next-generation transformer-based Natural Language Processing (NLP) architectures (GPT-2 and BERT, in particular) with open-access APIs and open-source libraries. This transition from hardcoded mappings to context-aware, generative AI architectures massively enhances the adaptability and, hence, robustness of human-computer interactions [8] [9].

Furthermore, the system now supports language inference via tokenized input processing, dynamic prompt engineering, and response conditioning, allowing it to handle open-ended tasks beyond fixed templates. These advancements facilitate the transition from static to semantic-level understanding, aligning the system's capabilities with modern NLP standards [44].

B. Augmented User-Centric Functionalities of MLP-based Frameworks

1) Offline response caching and persistent knowledge storage: One key enhancement in this research is the local caching of AI-generated responses for offline use. This capability addresses a critical limitation of cloud-based NLP models, which typically require uninterrupted internet access. The system enables selective storage of high-utility content, including commonly queried definitions, domain-specific formulas, unit conversions, standardized datasets, and language-dictionary-based questions, in JSON or CSV formats. If the User generated the prompt without internet access, it will look in the cache for similar keywords and create a response

that matches the best at the cursor position. (This procedure for the keyword matching and generating a response is already explained in reference [2].) These cached assets can be accessed via background threads or scheduled jobs, ensuring that offline inference remains feasible even without invoking remote APIs [45].

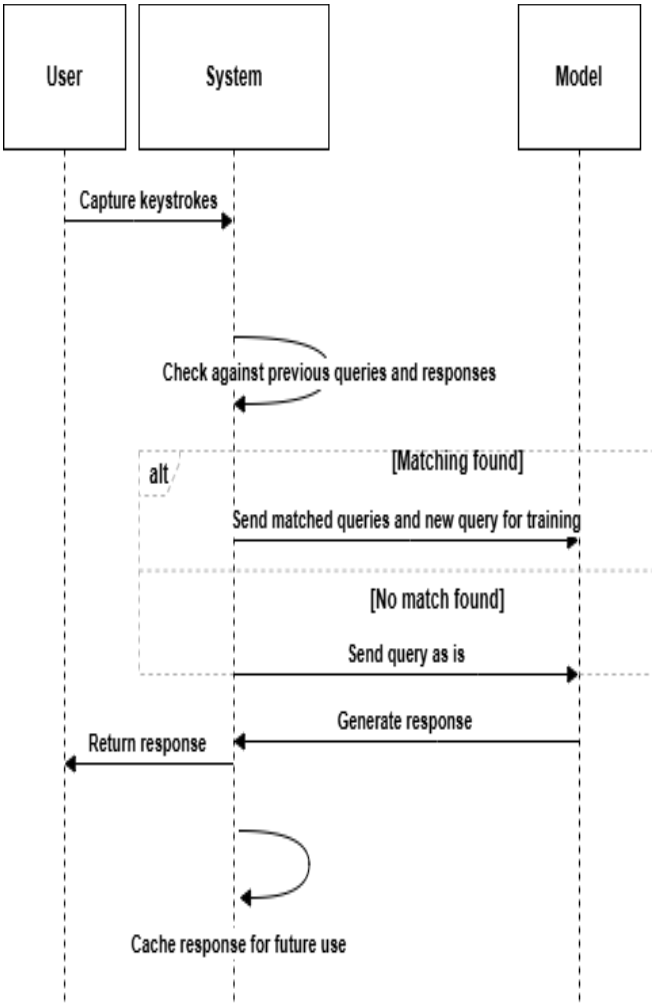

Fig. 9. Cached data to train the model.

This approach aligns with principles of edge AI, where inference and computation are offloaded to local devices to reduce latency and dependency on external networks, thereby enhancing both resilience and real-time performance [46].

2) Using the cached data to train the existing model: As discussed in Section II, when a special keystroke combination is met, the child thread starts the cached base operation. All caching will be stored on the User/Developer's local device and used only with authorized permission to generate a more accurate prompt via connected NLP models. It can be stopped at any time by pressing the special key combinations again.

It can be stored in the same format as a multi-dimensional array mentioned in Section II above, or in JSON format, depending on the programming language selected to develop the tool. Still, the flow of the operation will be the same.

Once the user starts using the tool, it begins capturing their keystrokes. When the hotkey is pressed, it registers the sequence of keystrokes as query sentences. It can submit queries to different open-source AI models and receive responses. As illustrated in Fig. 9, every time a query is made by pressing the Enter key, it is checked against previously cached questions (Only after the user activates the cache-based system via a special key combination, which is set at "start cached" and "stop cached", later users can change it via config file) and responses; if the sentence structure or keywords match, it is noted that this may be a similar request. (sentence matching algorithm is provided in [2]). Then the previous response will be sent along with the query to train the specific model to generate a response that matches the last successful pattern. If no match is found, the query will be sent to the open-source API as-is.

Once the framework's thread finds a matching cache activation combination, it will activate the program's other child thread(loop). This thread is responsible for matching strings from the previous response, sending the matched cache along with the new request, and storing the input and output. If the user presses the same combination again, the thread will stop this loop, and no cache will be sent or stored from that point.

3) Cross-application language support and system-wide paraphrasing: The framework also proposes application-agnostic language support, including real-time spelling correction, semantic synonym generation, language translation, and context-sensitive paraphrasing. Conventional implementations are for word processors (e.g., Microsoft Word or Google Docs, typically via plugins or macros). In comparison, this system works in the background at the OS level, where text is intercepted from any active application window and routed through NLP APIs [2], [47]. For example, users who write emails in Thunderbird and take notes in Sublime Text will receive AI-generated paraphrases or vocabulary recommendations without installing software-specific extensions. Like OS-level accessibility services, this architecture ensures uniform NLP augmentation across a broad spectrum of user applications and workflows.

4) Behavioral biometrics and typing pattern-based security: Beyond usability, the proposed system incorporates a lightweight, AI-driven keystroke dynamics monitoring module that passively analyzes a user's typing biometrics in real time. This component captures parameters such as keypress duration, inter-key latency, and frequency patterns to establish a unique behavioral fingerprint [48]. These learned patterns can be frequently used sentences and stored keywords from the cached memory and the config file. The child thread will continue matching the pattern once started via a special key combination. If anomalous input sequences deviate from the learned pattern, a security submodule is triggered, which may execute actions such as user alerts, session locking, intrusion logging, or system shutdown, as per Section III.

This implementation is inspired by research in behavioral user authentication and continuous identity verification, which have been applied in domains such as secure terminals, financial systems, and high-stakes editorial workflows [49]. Integrating

this with an NLP interface enables intelligent, non-intrusive surveillance, adding a security layer that is particularly beneficial to journalists, academic writers, and researchers handling sensitive or original content.

To mitigate the risk of capturing sensitive user inputs such as passwords or confidential credentials, the proposed framework incorporates application-aware filtering mechanisms that identify protected input fields based on standard UI attributes, including masked or secure text-entry indicators. Input interception is selectively suspended when interactions occur within recognized password or secure-entry contexts, ensuring that sensitive data is neither processed nor transmitted to external inference services. In addition to automated filtering, the framework provides a user-controlled pause and resume mechanism that allows monitoring to be temporarily disabled through predefined commands [2] (e.g., typing a pause trigger), enabling users to manually suspend input capture during interactions involving sensitive information or private credentials.

With respect to the behavioral protection layer, the biometric keystroke analysis module operates on anonymized keystroke dynamics features such as typing rhythm and latency patterns without retaining raw textual content. The performance of this layer is influenced by variability in individual typing behavior and environmental conditions, which may result in occasional false-positive or false-negative classifications during anomaly detection. In the current implementation, the biometric layer is intended to function as a supplementary risk-assessment mechanism rather than a primary authentication system. Further evaluation of classification thresholds and adaptive behavioral modeling is identified as an area for future work to improve detection accuracy and minimize unintended behavioral misclassification.

5) Additional applications and future-scalable extensions: To broaden the scope, several novel use cases are proposed or in development:

- AI-augmented Integrated Development Environments (IDEs): Automatic code snippet insertion, real-time explanation of programming constructs, and citation generation using NLP models like Codex or CodeBERT.
- Education Platforms: Adaptive learning assistants that dynamically rephrase explanations or suggest learning resources in response to student queries.
- Legal and Medical Transcription: Real-time conversion of typed input into domain-specific legalese or medical jargon using fine-tuned Large Language Models (LLMs).
- Multilingual Operating Systems: Automatic translation of input text into target languages on-the-fly for bilingual users, with support for right-to-left (RTL) and logographic scripts.

These future extensions illustrate the potential for a modular AI service layer across personal computing environments, where diverse domains can benefit from real-time, context-aware, NLP-powered interactions.

V. Current Limitations and Room for Improvement

Although the proposed method demonstrates how traditional user-selected IDEs can be enhanced with AI-driven intelligence through keystroke-level interactions, some technical issues remain. First, when generating synthetic keystrokes, words with repeated characters (like “small,” which has a double "l") can sometimes be misread by the keystroke emission module. In some cases, the system registers only one keypress event, resulting in outputs like “smal” instead of the correct “small.” This issue occurs due to event coalescing behavior when rapidly dispatching key events, a known challenge in low-level input simulation frameworks.

Second, if the NLP model produces a lengthy or complex response, the generated ASCII sequence might exceed the active processing buffer's capacity. This can cause the buffer to fill up, which may slow performance or, in severe cases, crash the main thread responsible for capturing system events and generating keystrokes. Similar buffer overflow and thread-blocking issues have been observed in asynchronous I/O systems and event-driven input pipelines.

Both of these limitations can be addressed in future versions of the system by implementing a segmented buffering strategy combined with asynchronous queuing algorithms for keystroke storage and controlled output. Using dynamic buffer allocation, non-blocking queues, and per-character acknowledgment techniques can help manage repeated-character sequences effectively and prevent overflow during significant responses.

Although operating system-level input hooks are supported across major platforms such as Windows, macOS, and Linux, their practical implementation differs due to variations in security policies, event-handling frameworks, and permission models. For instance, Windows environments allow programmatic access to input simulation through system calls such as SendInput(), whereas macOS requires the use of Quartz Event Services with additional accessibility permissions for event interception and synthesis. Similarly, Linux-based systems rely on mechanisms such as X11 or Wayland input subsystems, which may impose sandboxing or privilege restrictions. These platform-specific differences can influence the deployment and operational behavior of keystroke monitoring and simulation modules within the proposed framework. A comprehensive cross-platform implementation strategy, accounting for OS-level constraints and permission requirements, remains an area for future investigation to enhance the portability and robustness of the system.

VI. Future Scope

In the proposed framework, users can dynamically switch among multiple AI and NLP models to leverage their distinct functional advantages. Each model offers specialized capabilities optimized for specific linguistic or contextual tasks. Future enhancements aim to incorporate adaptive analytics modules that autonomously select the most appropriate model based on the user’s active application context and ongoing workflow. This adaptive orchestration will enable real-time utilization of heterogeneous AI agents, thereby enhancing overall system efficiency and contextual responsiveness.

Beyond its current capabilities with open-source NLP models, the proposed framework presents significant potential for the development of personalized AI assistants. Future extensions of this work will focus on enabling the assistant to learn individual user behavior, task preferences, and interaction patterns over time. By leveraging adaptive learning techniques, the system can evolve into a user-specific intelligent agent capable of autonomously handling routine daily activities, providing contextual recommendations, and optimizing task execution based on personal workflows.

Moreover, future research will explore the integration of behavior-aware security mechanisms within personalized assistants. By modeling normal user interaction patterns, the system can detect anomalous activities and initiate preventive actions, thereby strengthening protection against unauthorized access and data misuse. Such advancements will transform the framework into a comprehensive, user-centric AI ecosystem that combines intelligent automation, personalization, and proactive security.

VII. Conclusion

The proposed interface-agnostic AI integration framework allows diverse NLP models and AI-based assistants to work smoothly without any traditional graphical user interfaces. It demonstrates how intelligent automation can be integrated into user workflows through low-level input monitoring, standardized communication protocols, and secure API orchestration. This minimizes reliance on platform-specific plug-ins or standalone tools to optimize task execution and user productivity. Moreover, AI-assisted decision-making with context-aware keystroke interpretation introduces a new approach to adaptive computing that learns user intent and automatically summons the appropriate AI model for a given task. This architecture is modular, secure, and scalable enough to support ever-evolving AI ecosystems and open APIs. It's possible to further expand this framework with improved cross-device synchronization, advanced behavioral analytics, and self-optimizing task delegation to make it accessible across enterprise and personal computing environments in future iterations. This work adds a valuable insight to the ongoing evolution of intelligent human–computer interaction by bridging traditional input processing with contemporary AI-driven automation.

References

[1] J. Z. K. L. P. L. Pranav Rajpurkar, "SQuAD: 100,000+ Questions for Machine Comprehension of Text," Association for Computational Linguistics, p. 2383–2392, November 2016.

[2] D. H. Patel, "Real-time Interactive and Artificial Intelligence," International Journal of Engineering and Advanced Technology (IJEAT), vol. 8, no. 3, pp. 179-183, 2019.

[3] A. U. H. H. S. M. B. R. A. M. A. S. H. J. H. B. O. B. S. L. Jamil Hussain, "Model-based adaptive user interface based on context and user experience evaluation," Journal on Multimodal User Interfaces, pp. 1-16, 2018.

[4] E. a. T. H. Wiki, "Milestones: American Standard Code for Information Interchange (ASCII), 1963," ETHW, 2016.

[5] I. Corporation, "IBM Personal Computer Technical Reference," IBM, August 1981. [Online]. Available: https://bitsavers.org/pdf/ibm/pc/pc/6025008_PC_Technical_Reference_Aug81.pdf?utm_source=chatgpt.com. [Accessed 14 November 2025].

[6] Y. Yang and L. Li, "Turn Smartphones into Computer Remote Controllers," International Journal of Computer Theory and Engineerin, vol. 4, no. 4, pp. 561-564, 2012.

[7] M. Corporation, "Keyboard Input Overview," Microsoft, 2025.

[8] J. Devlin, M.-W. Chang, K. Lee and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," in NAACL-HLT 2019, 2019.

[9] OpenAI, "GPT-2: Language Models are Unsupervised Multitask Learners," OpenAI, San Francisco, CA, 2019.

[10] M. Neumann, D. King, I. Beltagy and W. Ammar, "ScispaCy: Fast and Robust Models for Biomedical Natural Language Processing," in Association for Computational Linguistics (ACL), Florence, 2019.

[11] M. N. Group, "MITIE: MIT Information Extraction," 03 February 2015. [Online]. Available: https://github.com/mit-nlp/MITIE. [Accessed 22 September 2025].

[12] S. Contributors, "spacy-nlp (Node.js wrapper for spaCy)," NPM, 2018.

[13] J. D. Weisz, S. Kumar, M. Muller, K.-E. Browne, A. Goldberg, E. Heintze and S. Bajpai, "Examining the Use and Impact of an AI Code Assistant on Developer Productivity and Experience in the Enterprise," arXiv, 2024.

[14] O. S. Foundation, "OpenSSL Cryptography and SSL/TLS Toolkit," OpenSSL Software Foundation, 2025. [Online]. Available: https://www.openssl.org/.

[15] A. Contributors, "Axios – Promise based HTTP client for the browser and node.js," Axios, 2024.

[16] T. C. Project, "libcurl – client-side URL transfer library," The curl Project, 2024. [Online]. Available: https://curl.se/libcurl.

[17] M. Corporation, "SetWindowsHookExA function (winuser.h)," Microsoft, 2023.

[18] X. Foundation, "Xlib – C Language X Interface," X.Org Foundation, 1985.

[19] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, J. Davison, S. Shleifer, P. von Platen, C. Ma, Y. Jernite, J. Plu and C. Xu, "Transformers: State-of-the-Art Natural Language Processing," in Association for Computational Linguistics (ACL), Online, 2020.

[20] E. International, "The JSON Data Interchange Syntax, ECMA-404, 2nd ed.," ECMA International, 2017.

[21] T. L. Foundation, "systemd.service — Service unit configuration," 2015–present. [Online]. Available: https://www.freedesktop.org/software/systemd/man/systemd.service.html.

[22] Microsoft, "Service Applications," 2025. [Online]. Available: https://learn.microsoft.com/en-us/windows/win32/services/service-applications.

[23] P. S. Foundation, "asyncio — Asynchronous I/O," Python 3 Documentation, [Online]. Available: https://docs.python.org/3/library/asyncio.html.

[24] I. C. Foundation, "std::thread — C++ standard threading library," [Online]. Available: https://en.cppreference.com/w/cpp/thread/thread.

[25] S. Bird, E. Klein and E. Loper, Natural Language Processing with Python: Analyzing Text with the Natural Language Toolkit, Sebastopol, CA: O'Reilly Media, 2009.

[26] S. M. H. Zaidi, R. Iqbal, A. Alharbi, H. H. Zuberi, A. Ahmed and M. U. Sheikh, "Intelligent Process Automation Using Artificial Intelligence to Create Human Assistant," International Journal of Information Technology and Decision Making, vol. 24, no. 5, pp. 1234-1256, 2025.

[27] OpenAI, "ChatGPT: Optimizing Language Models for Dialogue," OpenAI, San Francisco, CA, 2023.

[28] G. Developers, "Gemini is now accessible from the OpenAI Library," Google, Mountain View, CA, 2024.

[29] Z. Khan, "Microsoft 365 Copilot APIs: Unlocking enterprise knowledge for AI with the Retrieval API," Microsoft 365 Developer Blog, 2025.

[30] M. Yoshitake, "Introducing AI native SDKs for Alexa+," Amazon.com, Inc., 2025.

[31] B. W. Kernighan and R. Pike, "The UNIX Programming Environment," Englewood Cliffs, NJ, Prentice-Hall, 1984, pp. 71-99.

[32] D. Hennig, "Windows PowerShell: Batch Files on Steroids," Saskatoon, Stonefield Software Inc., 2019.

[33] Cppreference.com, "std::system," C++ Reference Documentation, 2024. [Online]. Available: https://en.cppreference.com/w/cpp/utility/program/system.

[34] D. H. Patel, "Spyware Triggering System by Particular String Value," International Journal of Engineering Research and Development, vol. 11, no. 09, pp. 32-36, 2015.

[35] A. Somayaji and S. Forrest, "Automated Response Using System-Call Delays," in IEEE Symposium on Security & Privacy, Berkeley, CA, 2000.

[36] H. Xueyu, X. Tao, y. Biao, W. Zishu , X. Ruixuan, C. yurun, y. Jiasheng, T. Meiling , Z. Xiangxin, Z. Ziyu, L. Yuhuai, X. Shengze, W. Shenzhi, X. Xinchen , Q. Shuofei, W. Zhaokai, k. Kun, Z. tieyong, W. Liang, l. Jiwei, j. Yuchen Eleanor, Z. Wanchunshu, W. Guoyin, Y. Keting, Z. Zhou, Y. hongxia, w. Fan, Z. Shengyu and w. Fei, "OS Agents: A Survey on MLLM based Agents for General Computing Devices Use," arXiv, 2025.

[37] Motion, "AI-Powered Calendar and Task Management Platform," Motion, Inc., 2025.

[38] Clockwise, "AI-Powered Time Management Calendar," Clockwise, Inc., 2025.

[39] Reclaim.ai, "AI Calendar for Work and Life," Reclaim.ai, 2025.

[40] Google Workspace, "Automation and Integration Tools for Google Workspace," Google Workspace Marketplace, 2025.

[41] D. Fett, R. Küsters and G. Schmitz, "A Comprehensive Formal Security Analysis of OAuth 2.0," arXiv, 2016.

[42] O. Kalkan, "Event Driven AI Workflows: Powering Real Time Automation," Gibion.ai Blog, 2025.

[43] IBM, "What is AI Orchestration?," IBM Think, 2025.

[44] J. Devlin, M.-W. Chang, K. Lee and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," in Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics (NAACL), 2019.

[45] T. Wolf, V. Sanh, L. Debut, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz and J. Brew, "Transformers: State-of-the-Art Natural Language Processing," in Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP): System Demonstrations, 2020.

[46] H. Lin and S. Zhu, "Edge Intelligence: Paving the Last Mile of Artificial Intelligence With Edge Computing," Proceedings of the IEEE, vol. 107, no. 8, pp. 1738-1762, 2019.

[47] Y. Hu, X. Jing, K. Youlim and R. T. Julia, "Misspelling Correction with Pre-trained Contextual Language Model," arXiv, 2021.

[48] P. S. Teh, A. B. J. Teoh and S. Yue, "A Survey of Keystroke Dynamics Biometrics," The Scientific World Journal, vol. 2013, 2013.

[49] F. Monrose and A. D. Rubin, "Keystroke Dynamics as a Biometric for Authentication," Future Generation Computer Systems, vol. 16, no. 4, pp. 351-359, 2000.

AUTHOR'S PROFILE

DIVIJ H PATEL was born in Nadiad, Gujarat, India, in 1995. He received the B.E. degree in Computer Engineering from Gujarat Technological University, Gujarat, India, in 2017. He has pursued the postgraduate certification in Information and Communication Technology Solutions for Small Businesses at Confederation College, Thunder Bay, ON, Canada.

During his undergraduate studies, he conducted research in several areas of computer science and engineering. He has published three research papers in international journals, including "Spyware Triggering System with the Particular String Value" and "Multipurpose Reminder System" in the International Journal of Engineering Research and Development, and "Copy-Paste Command with Additional Facilities" in the International Journal of Computer Applications, New York, USA.

He has filed three patents with the Indian Patent Office covering innovations in wearable devices and human–computer interaction. His patented works include the Smart Wearing Cap Device, a method for Assigning Shortcut Keys, and the Folding Calculator. His research and innovations have also been featured in newspapers and television media such as Gujarat Samachar, Sandesh News, Vichar Kranti News, DNN Link News Channel, and VTV News.