API Agents vs. GUI Agents: Divergence and Convergence

Chaoyun Zhang [1] **Shilin He** [1] **Liqun Li** [1] **Si Qin** [1] **Yu Kang** [1] **Qingwei Lin** [1] **Saravan Rajmohan** [1] **Dongmei Zhang** [1]

Abstract

Large language models (LLMs) have evolved beyond simple text generation to power software agents that directly translate natural language commands into tangible actions. While API-based LLM agents initially rose to prominence for their robust automation capabilities and seamless integration with programmatic endpoints, recent progress in multimodal LLM research has enabled GUI-based LLM agents that interact with graphical user interfaces in a human-like manner. Although these two paradigms share the goal of enabling LLM-driven task automation, they diverge significantly in architectural complexity, development workflows, and user interaction models.

This paper presents the first comprehensive comparative study of API-based and GUI-based LLM agents, systematically analyzing their divergence and potential convergence. We examine key dimensions and highlight scenarios in which hybrid approaches can harness their complementary strengths. By proposing clear decision criteria and illustrating practical use cases, we aim to guide practitioners and researchers in selecting, combining, or transitioning between these paradigms. Ultimately, we indicate that continuing innovations in LLM-based automation are poised to blur the lines between API- and GUI-driven agents, paving the way for more flexible, adaptive solutions in a wide range of real-world applications.

1. Introduction

The advent of large language models (LLMs) (Zhao et al., 2023) has ushered in a new era of artificial intelligence, enabling advanced natural language understanding and generation across a wide range of domains. While LLMs have long been recognized for their ability to produce coherent text, recent developments have led to LLM-based agents capable of mapping language inputs to real actions in digital environments (Wang et al., 2024a). By grounding LLMs in tangible operations, these agents can interact with various software systems, execute commands, and have a practical impact on the software ecosystems they inhabit.

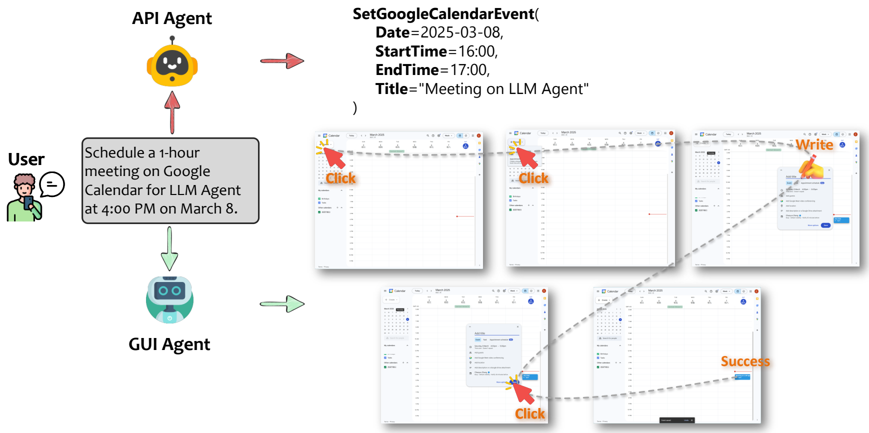
Initially, software LLM agents were predominantly Application Programming Interface (API)-centric, interacting with external tools, functions, or services through well-defined programmatic interfaces (Du et al., 2024). This approach allowed agents to orchestrate microservices, query search engines, or even control third-party applications via documented APIs, with a automated and efficient manner. Products such as Microsoft's Copilot exemplify how API-based LLM agents have become mainstream (Stratton, 2024), rapidly transitioning from research prototypes to widely adopted industrial solutions. The traction of these agents, both in academia and industry, underscores their capacity to streamline tasks through automation while maintaining robust scalability and interoperability.

Concurrently, with multimodal capabilities gaining prominence in LLM research, a new class of agents has emerged: Graphical User interfaces (GUIs)-based LLM agents (Zhang et al., 2024a). These agents interact not only through APIs but also by "observing" and manipulating graphical user interfaces of software, be they on desktop, mobile, or web applications. Projects such as UFO (Zhang et al., 2024b), CogAgent (Hong et al., 2024), and OpenAI Operator (OpenAI, 2025) illustrate how GUI-based agents can bring richer user experiences, improved accessibility, and provide more general automated control of software. By integrating vision or screen-based understanding with text-based reasoning, these agents push the boundaries of human-computer interaction, showcasing how AI can seamlessly blend with intuitive, visual workflows.

Despite the promise of both paradigms, API-based and GUI-based LLM agents exhibit significant differences in their architectures, development methodologies, and user interaction models (Zhang et al., 2024a). For example, as illustrated in Figure 1, an API-based agent scheduling a Google Calendar meeting could make a single call provided the proper endpoint and authentication, to instantly create the

[1] Microsoft. Correspondence to: Chaoyun Zhang <chaoyun.zhang@microsoft.com>.

Workshop on Computer-use Agents @ ICML 2025, Vancouver, Canada. Copyright 2025 by the author(s).



Schedule a 1-hour meeting on Google Calendar for LLM Agent at 4:00 PM on March 8.

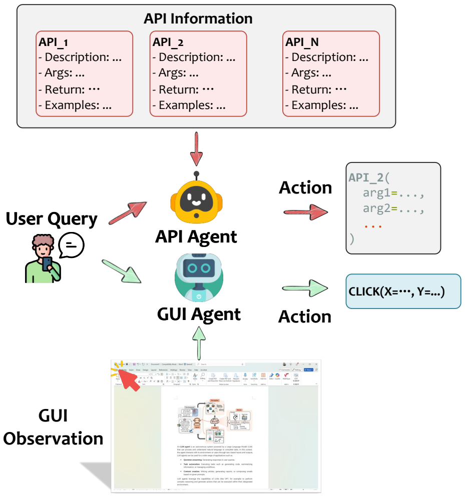
```
SetGoogleCalendarEvent(
    Date=2025-03-08,
    StartTime=16:00,
    EndTime=17:00,
    Title="Meeting on LLM Agent"
)
```

Figure 1. The difference between an API agent and a GUI agent in completing the task “Schedule a 1-hour meeting on Google Calendar for LLM Agent at 4:00 PM on March 8”.

event. In contrast, a GUI-based agent would open the web interface, navigate through the calendar visually, fill in relevant fields, and click buttons to finalize the meeting details. While the API-based approach tends to be faster and more resource-efficient, it depends on well-defined endpoints and reliable infrastructure. Conversely, GUI-based agents can interface with application’s front end but typically require multiple user-like steps, which can be slower and more error-prone (Song et al., 2024).

These trade-offs underscore the fundamental divergence between API- and GUI-based agents and have sparked vigorous discussions regarding their relative merits, overlap, and even necessity. Although they may appear to compete, many believe that a thorough examination of both differences and shared attributes is crucial. Indeed, no unified framework or comparative analysis has yet offered clear guidelines on when one approach might outperform the other—or if a hybrid strategy could combine their strengths.

To address this gap, this paper systematically examines both the divergence and potential convergence of API-based and GUI-based LLM agents in software automation. We begin by clarifying the conceptual foundations of each paradigm, followed by an in-depth comparative analysis across key dimensions. We then explore how ongoing innovations in LLM-based interactions are increasingly blurring the lines between API and GUI agents, and discuss a hybrid approach that leverages their respective strengths while mitigating their shortcomings. By outlining clear decision criteria, we aim to guide practitioners and researchers in selecting the most suitable agent type based on project requirements, resource constraints, and user experience objectives. Ultimately, this work serves as a comprehensive resource for academics and industry professionals, offering insights into how these agent types differ, where they converge, and how they may evolve to address emerging challenges.



```
API_2(
  arg1=...,
  arg2=...,
  ...
)
```

```
CLICK(X=···, Y=...)
```

Figure 2. The difference between an API agent and a GUI agent.

2. Background

LLM agents are designed to accept natural language requests from users and execute tasks in real or virtual environments. While they share the common goal of translating language into actionable commands, two distinct paradigms have emerged based on how these agents interact with their underlying systems: API-first and GUI-first LLM agents. Below, we provide an overview of each paradigm, highlighting their core principles and operational differences.

2.1. API-Based LLM Agents

An API-first LLM agent can be defined as

> *Intelligent agents that leverage LLMs as their cognitive engine to invoke one or more predefined APIs to fulfill user requests automatically.*

In this paradigm, the agent’s capabilities are bounded by a predefined set of tools, plugins, or function calls—collectively referred to as "APIs" (Shen, 2024), as shown in Figure 2. This constrained set of functions not only ensures reliability and safety but also simplifies the agent's decision space. Rather than generating full program code, the agent identifies which API to call for each step and populates the required parameters based on user intent.

Relevant API information, such as function names, descriptions, parameters, and schemas, is included in the LLMs prompt. When a user issues a natural language request, the LLM agent interprets the intent and chooses the most suitable API to perform the task. This approach has been widely adopted in many state-of-the-art models and frameworks, including function-call modes in GPT-4 (Hurst et al., 2024) and the plugin-only mode in TaskWeaver (Qiao et al., 2023).

2.2. GUI-Based LLM Agents

In contrast to the API-first paradigm, GUI-based LLM agents operate by interacting directly with GUIs rather than invoking predefined functions. As defined in (Zhang et al., 2024a), these agents can be described as:

> *Intelligent agents that operate within GUI environments, leveraging LLMs as their core inference and cognitive engine to generate, plan, and execute actions in a flexible and adaptive manner.*

GUI-based agents primarily rely on visual or multimodal inputs, such as application screenshots and textual representations (*e.g.*, accessibility trees or metadata). Instead of calling API endpoints, these agents navigate and manipulate on-screen elements using actions akin to human interactions, such as mouse clicks and keyboard inputs (Wang et al., 2024b), as shown in Figure 2. This "human-like" operational flow means that the agent must interpret visual layouts, locate relevant controls, and execute sequences of clicks, drags, or keypresses to accomplish tasks.

The rapid evolution of multimodal LLM (Hong et al., 2024; Zheng et al., 2025) capabilities has sparked considerable interest in GUI-based agents, leading to a range of prototypes, frameworks, and commercial products. Use cases include automated software testing, workflow automation, and accessibility enhancements. Ongoing research aims to address the inherent challenges of screen parsing, error handling, and robust action planning, areas where GUI-based LLM agents continue to evolve.

Together with API-first LLM agents, this GUI-based paradigm forms a complementary, but often contrasting approach to building intelligent systems that bridge natural language understanding and real-world task execution. As subsequent sections will illustrate, understanding both paradigms is crucial for selecting or designing an agent architecture suited to specific project goals and constraints.

3. Divergence Between API and GUI Agents

Although both API-based and GUI-based agents aim to automate tasks using natural language instructions, they diverge significantly in various perspectives. This section presents a comparative analysis across several key dimensions: Modality, Reliability, Efficiency, Availability, Flexibility, Security, Transparency, Human-Like Interaction, and Maintainability. By examining these factors, we can better understand the foundational distinctions between the two paradigms and their impact on real-world applications.

3.1. Modality

The most evident difference lies in the way each agent perceives and interacts with software. API agents rely on textual specifications for each available endpoint. They interpret the user's request, map it to the relevant function, and provide the necessary parameters for execution. By contrast, GUI agents process visual or multimodal inputs, such as screenshots or accessibility trees, and then navigate and manipulate user interface elements. Since GUI agents operate on actual interface controls, accurate visual grounding and interpretation become essential (Lu et al., 2024). While accessibility trees can offer some structured information, image-based comprehension remains central to the agent's interaction in a GUI environment.

3.2. Efficiency

Efficiency covers both the time and computational resources required to complete a task. API agents can generally handle complex tasks in a single function call, minimizing latency and reducing the inference costs. In contrast, GUI agents frequently must perform a series of user-like actions—opening menus, typing text, clicking buttons—to accomplish the same goal. Even routine operations, such as navigating through application panes, can require multiple steps. This user-level approach, while intuitive, can slow task execution and increase operational overhead compared to a well-designed API (Zhang et al., 2024a). If completing a task takes significantly longer than it would for a human, it can limit GUI agents' practical applicability.

3.3. Reliability

Reliability often stems from the complexity of the underlying interaction model. API agents typically exhibit robust performance when they can access stable, well-defined endpoints. Such endpoints are easily maintained, versioned, and tested, leading to predictable outcomes. GUI agents, however, encounter challenges whenever application layouts or screen elements change unexpectedly. This is because GUIs

Table 1. Comparison of API vs. GUI agents across key dimensions.

Dimension	API Agents	GUI Agents
Modality	Rely on text-based API calls	Depend on screenshots or accessibility trees
Reliability	Generally higher with well-defined endpoints	Lower due to visual parsing and layout changes
Efficiency	Achieve complex tasks in a single call	Require multiple user-like actions
Availability	Limited to published or pre-defined APIs	Can operate on any visible UI element
Flexibility	Constrained by existing APIs	Highly adaptable to new or unexposed features
Security	Manageable via granular endpoint controls	Riskier due to broad access to UI elements
Maintainability	Stable if APIs remain versioned	Prone to breakage on UI redesigns
Transparency	Often hidden, back-end driven	Step-by-step, visually traceable
Human-Like Interaction	Purely programmatic	Simulates user actions on a screen

are primarily designed for human interaction, often containing redundant elements and potential distractions that can disrupt automated workflows (McKay, 2013). These changes introduce uncertainties in the agent's visual parsing and planning, making GUI-based approaches more prone to errors. In addition, the multi-step decision-making process of GUI agents can compound errors at each step, ultimately reducing overall accuracy. As a result, GUI agents often require ongoing refinements and are still not as production-ready as their API-based counterparts in many scenarios.

3.4. Availability

Availability depends on how readily an agent can access the functionality necessary to fulfill a user's request. API agents are constrained by the endpoints or functions that developers have defined and exposed. If a desired feature is omitted, the agent cannot invoke it directly. This is particularly common in mobile applications, where developers often restrict external API access to maintain control over their private ecosystems.

Conversely, GUI agents can interact with virtually any application that presents a GUI, without requiring explicit API definitions. The universality of GUI-based interaction can be an advantage in environments where no formal APIs exposed, but it also demands more sophisticated interpretation and error handling to manage diverse or evolving UIs.

3.5. Flexibility

In addition to availability, flexibility denotes how easily the agent can adapt to new or modified use cases. API agents can call only the APIs that have been developed, documented, and integrated in advance. Expanding their functionality depends on creating and deploying additional endpoints. GUI agents, on the other hand, can theoretically operate on any visible element within an interface, thereby offering a higher degree of freedom. This freedom, however, requires advanced multimodal reasoning capabilities to locate and interact with UI objects consistently.

3.6. Security

Security plays a crucial role in deciding which paradigm to adopt. API agents typically offer more granular protection, as each endpoint can be individually secured with authentication, access control, or rate limiting. GUI agents may inadvertently access parts of the interface that perform privileged or destructive operations, raising the risk of unintended consequences. Since graphical interfaces are designed primarily for human users, enforcing comprehensive security policies on automated, mouse-and-keyboard-like interactions can be challenging. As a result, GUI-based agents may require additional safeguards to avoid unauthorized operations or misuse (Zhang et al., 2024b).

3.7. Maintainability

An other dimension relates to how easily the agent's functionality can be maintained and updated over time. API agents benefit from versioned, standardized interfaces. As long as the underlying endpoints remain stable, the agent logic remains mostly intact. New APIs can be seamlessly integrated into the agent by simply adding their descriptions to the prompt, ensuring easy maintenance.

By contrast, GUI agents are highly susceptible to interface redesigns, pop-up windows, layout shifts, and element renaming or relocation (Zhang et al., 2024c), all of which can break the automation if the GUI agent is unfamiliar with the change. This fragility can substantially increase the cost and frequency of maintenance.

3.8. Transparency

From a user's perspective, transparency refers to the clarity of observing how the agent fulfills a task. API agents often execute behind the scenes, providing limited visibility into the step-by-step process. Users typically see the final outcome without knowing which endpoints were invoked.

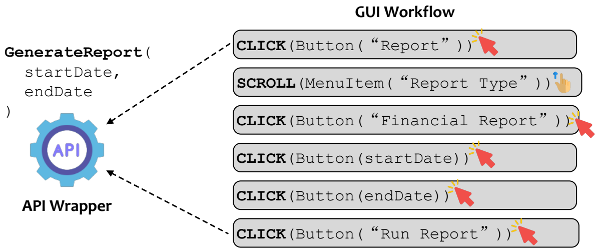
In contrast, GUI agents replicate user-level interactions, making each click and text entry visible as it unfolds. Rather than operating invisibly in the background like API calls, GUI agents offer a visible and interactive execution process, allowing users to observe, intervene, or adjust the workflow as needed. This design can be particularly beneficial in scenarios requiring step-by-step verification, training simulations, or automation of tasks where visual confirmation is necessary. This improves the interpretability of the workflow, allowing human observers to track and validate the agent's progress more intuitively.

3.9. Human-Like Interaction

Closely linked to transparency is the notion of simulating human behavior. API agents employ a purely programmatic approach, executing function calls directly without mimicking user interactions. They are optimized for efficiency, reliability, and scalability, but they lack any visual or interactive representation of the task execution. In contrast, GUI agents replicate the exact steps a human user would take—navigating through menus, filling in forms, and interacting with interface elements in a natural, sequential manner. This human-like execution enhances interpretability, making it easier for users to follow and understand the agent's actions, thereby fostering trust and a more intuitive user experience. This introduces a novel paradigm for human-computer interaction by bridging AI automation with user-centric workflows (Lin et al., 2025).

3.10. Summary

In summary, these dimensions illustrate the fundamental ways in which API-based and GUI-based agents diverge in practice. API agents provide efficiency, security, and reliability when backed by robust endpoints, but they are bounded by the limited set of exposed functions. GUI agents offer broad applicability and user-like workflows, yet they must overcome challenges in visual parsing, interface changes, and slower task execution. As the complexity of software ecosystems grows, understanding these divergent properties is essential for selecting the most suitable approach—or for designing hybrid solutions that combine the best attributes of both paradigms.



```
GenerateReport(
  startDate,
  endDate
)
```

```
CLICK(Button(“Report”))
SCROLL(MenuItem(“Report Type”))
CLICK(Button(“Financial Report”))
CLICK(Button(startDate))
CLICK(Button(endDate))
CLICK(Button(“Run Report”))
```

Figure 3. An example of a API wapper over a GUI workflow.

4. Convergence and the "Hybrid" Approach

Although API-based and GUI-based agents have traditionally been studied as separate paradigms, their conceptual foundations are not mutually exclusive. In practice, there are numerous scenarios where these approaches intersect or complement one another, leading to an emerging "hybrid" model. By drawing on the strengths of both API and GUI paradigms, this hybrid approach can achieve broader coverage of use cases, higher efficiency, and a more human-like interaction style. While still in its early stages, the potential for these convergent solutions to reshape how agents operate is becoming increasingly evident.

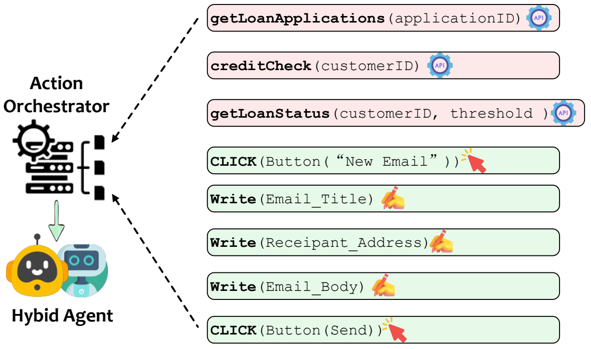
4.1. API Wrappers Over GUI Workflow

Some vendors transform GUI-based applications into quasi-API services by introducing a "headless mode" or a scripting interface that accepts function calls. This approach effectively abstracts GUI interactions into structured commands, allowing applications originally designed for human navigation to be automated in a more programmatic and scalable manner. For example, a specialized accounting application may traditionally require users to navigate through multiple dialog boxes and menus to generate a financial report. However, in a headless or scripted version, the same application could expose a function such as `GenerateReport(startDate, endDate)`, enabling direct execution without requiring manual UI navigation. This is conceptually similar to Robotic Process Automation (RPA) bots (Wornow et al., 2024), which mimic user actions but can also be optimized for backend workflows.

Although these wrappers still rely on GUI workflows under the hood, they present an API-like interface to developers, simplifying integration into broader automation pipelines. This transformation represents a subtle yet impactful form of convergence: an application originally designed for direct GUI interactions is reinterpreted as an API service, reducing the need for a dedicated GUI agent while retaining compatibility with existing software ecosystems. By bridging GUI automation with structured API-like interfaces, this approach enhances efficiency, scalability, and

Table 2. Examples of convergence paths in hybrid agent systems.

Approach	Key Benefit	Primary Challenge
API Wrappers Over GUI Tools	Provides a quasi-API experience for GUI-only software	Still relies on underlying GUI elements that may change
Unified Orchestration Tools	Hides agent-type details from the user	Complex logic to choose between API and GUI in real time
Low-Code / No-Code Solutions	Simplifies design of advanced workflows	May introduce hidden dependencies and abstractions



```
getLoanApplications(applicationID)
creditCheck(customerID)
getLoanStatus(customerID, threshold )
CLICK(Button( “New Email” ))
Write(Email_Title)
Write(Receipant_Address)
Write(Email_Body)
CLICK(Button(Send))
```

Figure 4. An example orchestrator to manage API and GUI actions.

ease of integration, particularly in enterprise environments where legacy applications must be incorporated into modern automation frameworks.

4.2. Unified Orchestration Tools

Enterprise-grade automation frameworks and process orchestration tools increasingly offer a single, unified environment where developers or operators can build high-level workflows without delving into the underlying agent mechanisms. Consider, for example, a large financial institution automating its loan approval process. Within an orchestration tool, a user could design a flowchart that checks a customer's credit score (using a secure API endpoint), then updates a Customer relationship management (CRM) system if the credit score meets a certain threshold. If no relevant API exists for updating the CRM, the platform can seamlessly switch to a GUI-based agent that navigates through the CRM's web interface in a user-like fashion. Thus, the tool automatically determines whether an API call or GUI interaction is most suitable for each task. We show such an example in Figure 4. By shielding users from these low-level decisions, orchestration platforms reduce complexity and streamline the automation pipelines.

Prototyping Experiments. We extend the UFO framework (Zhang et al., 2024b) to prototype a hybrid GUI and API approach that prioritizes API usage whenever available, using GPT-4o and o1 as the foundation model. To evaluate its effectiveness, we focus on *27* office-related tasks in OSWorld (Xie et al., 2024) and manually implement *12* APIs for Word, Excel, and PowerPoint. These applications expose COM interfaces that support the creation of custom functions, making them ideal candidates for deeper integration with the operating system and application layers. Table 5 summarizes the developed APIs in the Appendix A.

Table 3. Comparison of Success Rate (SR) and Average Completion Steps (ACS) between GUI-only and GUI with API actions.

Model	SR	ACS
GUI-only (GPT-4o)	16.3%	13.8
GUI + API (GPT-4o)	**22.4%**	**12.9**
GUI-only (o1)	16.3%	16.0
GUI + API (o1)	**24.5%**	**6.6**

Table 3 compares two configurations—GUI only and GUI with API—on two key metrics: *(i)* overall Success Rate (SR) and *(ii)* Average Completion Steps (ACS). To ensure fairness, ACS is computed only on the subset of tasks that both configurations complete successfully.

The results show that incorporating APIs improves SR for both GPT-4o (an increase of 6.1%) and o1 (an increase of 8.2%), demonstrating the benefit of combining GUI and API actions. GPT-4o benefits primarily by avoiding control detection failures, which often arise from unannotated interface elements. In contrast, o1 more frequently addresses planning errors by using direct API calls as shortcuts, reflecting its stronger reasoning capabilities and preference for concise execution plans.

In addition, the hybrid configuration reduces the effort required for task completion. Compared to GUI-only execution, the GUI with API setup achieves a 6.5% reduction in steps for GPT-4o and a substantial 58.5% reduction for o1. This improvement for o1 is due to its ability to strategically use APIs to bypass multiple GUI interactions. Overall, these confirm that combining GUI automation with API calls enhances both robustness and efficiency.

4.3. Low-Code / No-Code Solutions

Low-code and no-code platforms abstract many technical details behind visual interfaces, enabling non-experts (often

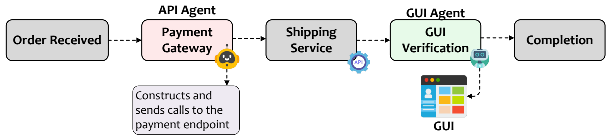


Figure 5. One example of a no-code platform to create workflows integrating both API calls and GUI agents.

referred to as "citizen developers") to construct applications or automations through drag-and-drop components (Tang et al., 2025). As illustrated in Figure 5, each block in this order-processing workflow represents a distinct task: the user drags a "Payment Gateway" block into the designer to handle transactions, configures it visually,while behind the scenes an API agent automatically constructs and sends calls to the payment endpoint. It then connects to a "Shipping Service" block for fulfillment. Behind the scenes, the platform typically issues API calls to the payment and shipping services, allowing the user to focus on the workflow's logical sequence rather than on lower-level protocols. Conversely, if a given step calls for GUI-based verification—for instance, checking a specific user interface element on a legacy system—the platform can seamlessly insert a GUI agent, simulating human interactions with the software. This combination of API-based and GUI-driven actions makes it straightforward to build end-to-end automations, blending the speed and scalability of APIs with the accessible, visual nature of GUI-centric operations.

4.4. Summary

These strategies illustrate how the once-distinct boundaries between API-based and GUI-based agents are gradually merging. The hybrid approach harnesses the flexibility and universality of GUI-driven interfaces alongside the reliability and performance of direct API calls. This convergence allows for more comprehensive automation, catering to diverse scenarios that range from rapid data processing to intricate user-interface validation, meeting varied requirements, whether efficiency, user-centric validation, or rapid development. Although further research and refinement are needed, these converging paradigms foreshadow a future in which agent-based automation is both wide-ranging and intelligent, adapting seamlessly to the evolving complexities of modern software ecosystems.

5. API vs. GUI Agents: Strategic Considerations

While the preceding sections have contrasted API- and GUI-based LLM agents in terms of architecture, reliability, efficiency, and potential convergence, many practical deployments hinge on a more fundamental question: *which paradigm should be employed under various real-world conditions?* This section provides guidance on selecting the most suitable strategy.

5.1. When to Favor API Agents

API-based agents tend to be the most compelling choice when well-defined programmatic interfaces exist. Official, stable APIs typically come with rigorous documentation and versioning, enabling strong error handling and consistent performance. In such an environment, developers can harness the inherent speed and reliability of API calls to execute tasks efficiently, thereby minimizing system overhead and latency. This approach is especially advisable when applications are designed for backend integrations, or when enterprise-level reliability is paramount for mission-critical workflows. By leveraging stable endpoints, API-based agents can also reduce long-term maintenance burdens, as changes in the system often entail versioned updates rather than complete interface overhauls.

In addition, API agents provide controlled access to applications, restricting functionality to a predefined and manageable scope. This is a crucial consideration in agent-based systems, where safety and security are paramount. In such cases, API agents are ideal, as their actions are confined to a constrained set of operations, ensuring predictable and secure interactions.

5.2. When to Favor GUI Agents

GUI-based agents become particularly relevant in scenarios where no direct API exists, or the available APIs provide only partial coverage of the required automation tasks. This is especially evident in mobile applications, where each app operates as an isolated environment, restricting external API access. Furthermore, system-level operations on mobile devices often require root access, further limiting API usability and necessitating GUI-based automation as an alternative.

Another key advantage of GUI agents is their ability to perform visual validation, which is essential in workflows that require confirming on-screen text, UI element positioning, or interface consistency before taking action. In such cases, a GUI agent, which interacts with software much like a human user, offers clear benefits over an API-based approach. Similarly, legacy or proprietary systems that lack extensible backend services can leverage GUI-driven automation, allowing agents to navigate existing interfaces without requiring modifications to the underlying codebase or the development of new APIs. This makes GUI agents particularly valuable in enterprise environments where implementing and maintaining new API integrations would be impractical or cost-prohibitive.

Moreover, GUI agents are inherently well-suited for applications that rely on interactive or graphical manipulation. Tasks such as creating animations, drawing in Photoshop,

Table 4. Strategic criteria for selecting agent paradigms.

Scenario	Recommended Approach	Rationale
Stable, well-documented APIs	API Agents	Exploit robust endpoints for speed and reliability
Performance-critical operations	API Agents	Reduce latency and overhead via direct function calls
Controlled access to applications	API Agents	Ensure safety and security
Legacy or proprietary software	GUI Agents	Automate tasks without requiring new backend integration
Visual validation or UI testing	GUI Agents	Verify on-screen text or elements directly
Interactive or graphical manipulation	GUI Agents	Seamlessly replicate human-like interactions with visual elements
Partial API coverage	Hybrid	Combine UI-based steps where APIs are unavailable with direct calls for data-heavy tasks
Future-proofing	Hybrid	Facilitate switching from GUI to API as endpoints evolve

or interacting with complex design tools are best executed through direct visual interactions rather than API commands. In these cases, GUI-based automation closely mirrors the natural way humans interact with such applications, making it the preferred choice over an API-driven approach.

5.3. When to Consider a Hybrid Approach

A hybrid strategy combines the strengths of both paradigms, providing a unified workflow that can accommodate a wide range of requirements. This approach is particularly advantageous when some aspects of the task map neatly onto existing APIs, while other components remain exclusively accessible through a graphical interface. In such cases, using an API-based agent for data-intensive or programmatically streamlined operations preserves performance, while a GUI agent handles specialized front-end interactions or visual validations. Moreover, adopting a hybrid solution offers flexibility for future system evolution; as new APIs become available, tasks initially managed via the GUI can be seamlessly transitioned to API calls.

5.4. Summary

In summary, the strategic considerations for deploying API-based versus GUI-based agents depend on the nature of the target software, the level of integration or validation required, and long-term sustainability concerns. API agents excel when stable, documented endpoints exist, offering a reliable and performant mode of automation. GUI agents are advantageous in contexts where interfaces are the only means of access or where visual confirmation is essential. Finally, hybrid approaches strike a balance between these strengths, allowing organizations to adapt as their software ecosystems evolve. By taking these factors into account, decision-makers can ensure they select the agent paradigm that best aligns with their specific requirements.

6. Conclusion & Looking Forward

The advent of LLM-based agents represents a significant leap forward in automation. These agents embody two core paradigms: one centered on well-defined programmatic interfaces (API agents) and one rooted in human-like interactions with graphical interfaces (GUI agents). By design, these paradigms differ in their operational principles, leading to a divergence in architectural choices, performance profiles, and real-world applicability. However, they also exhibit complementary strengths—API agents excel at speed, security, and reliability, while GUI agents offer flexibility, broad applicability, and transparency—making them poised for a future of hybridization and convergence.

Looking Ahead. The ongoing maturation of LLM technologies will likely reinforce both strands of agent development. On one hand, increasingly capable coding assistants promise to simplify the creation and maintenance of APIs, thereby enhancing the scalability of API agents. On the other hand, the rise of powerful multimodal models will expand the scope of GUI-based agents, enabling more robust visual understanding and sophisticated manipulation of GUIs. These point to an evolving ecosystem in which API-centric and GUI-centric approaches become increasingly interwoven.

Looking forward, the seamless integration of these agent types may give rise to entirely new forms of software—tools that automatically generate or refine APIs for efficient back-end operations and also dynamically orchestrate user-interface elements for transparent front-end interactions. This confluence of paradigms has the potential to transform human-computer interaction, blurring the boundaries between what is generated by code and what is experienced through a visual interface. In the long term, it could reshape how we conceive of software development, user experience, and the broader workflows that underlie digital ecosystems.

References

Yu Du, Fangyun Wei, and Hongyang Zhang. Anytool: Self-reflective, hierarchical agents for large-scale api calls. *arXiv preprint arXiv:2402.04253*, 2024.

Wenyi Hong, Weihan Wang, Qingsong Lv, Jiazheng Xu, Wenmeng Yu, Junhui Ji, Yan Wang, Zihan Wang, Yuxiao Dong, Ming Ding, et al. Cogagent: A visual language model for gui agents. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 14281–14290, 2024.

Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, et al. Gpt-4o system card. *arXiv preprint arXiv:2410.21276*, 2024.

Ziyue Lin, Siqi Shen, Zichen Cheng, Cheok Lam Lai, and Siming Chen. Carbon and silicon, coexist or compete? a survey on human-ai interactions in agent-based modeling and simulation. *arXiv preprint arXiv:2502.18145*, 2025.

Yadong Lu, Jianwei Yang, Yelong Shen, and Ahmed Awadallah. Omniparser for pure vision based gui agent. *arXiv preprint arXiv:2408.00203*, 2024.

Everett N McKay. *UI is communication: How to design intuitive, user centered interfaces by focusing on effective communication*. Newnes, 2013.

OpenAI. Operator system card, January 2025. Released on January 23, 2025.

Bo Qiao, Liqun Li, Xu Zhang, Shilin He, Yu Kang, Chaoyun Zhang, Fangkai Yang, Hang Dong, Jue Zhang, Lu Wang, et al. Taskweaver: A code-first agent framework. *arXiv preprint arXiv:2311.17541*, 2023.

Zhuocheng Shen. Llm with tools: A survey. *arXiv preprint arXiv:2409.18807*, 2024.

Yueqi Song, Frank F Xu, Shuyan Zhou, and Graham Neubig. Beyond browsing: Api-based web agents. 2024.

Jess Stratton. An introduction to microsoft copilot. In *Copilot for Microsoft 365: Harness the Power of Generative AI in the Microsoft Apps You Use Every Day*, pages 19–35. Springer, 2024.

Jiabin Tang, Tianyu Fan, and Chao Huang. Metachain: A fully-automated and zero-code framework for llm agents. *arXiv preprint arXiv:2502.05957*, 2025.

Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, et al. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18 (6):186345, 2024a.

Lu Wang, Fangkai Yang, Chaoyun Zhang, Junting Lu, Jiaxu Qian, Shilin He, Pu Zhao, Bo Qiao, Ray Huang, Si Qin, et al. Large action models: From inception to implementation. *arXiv preprint arXiv:2412.10047*, 2024b.

Michael Wornow, Avanika Narayan, Krista Opsahl-Ong, Quinn McIntyre, Nigam Shah, and Christopher Re. Automating the enterprise with foundation models. *Proceedings of the VLDB Endowment*, 17(11):2805–2812, 2024.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Jing Hua Toh, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, et al. Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments. *Advances in Neural Information Processing Systems*, 37:52040–52094, 2024.

Chaoyun Zhang, Shilin He, Jiaxu Qian, Bowen Li, Liqun Li, Si Qin, Yu Kang, Minghua Ma, Qingwei Lin, Saravan Rajmohan, et al. Large language model-brained gui agents: A survey. *arXiv preprint arXiv:2411.18279*, 2024a.

Chaoyun Zhang, Liqun Li, Shilin He, Xu Zhang, Bo Qiao, Si Qin, Minghua Ma, Yu Kang, Qingwei Lin, Saravan Rajmohan, et al. Ufo: A ui-focused agent for windows os interaction. *arXiv preprint arXiv:2402.07939*, 2024b.

Yanzhe Zhang, Tao Yu, and Diyi Yang. Attacking vision-language computer agents via pop-ups. *arXiv preprint arXiv:2411.02391*, 2024c.

Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. A survey of large language models. *arXiv preprint arXiv:2303.18223*, 1(2), 2023.

Jiani Zheng, Lu Wang, Fangkai Yang, Chaoyun Zhang, Lingrui Mei, Wenjie Yin, Qingwei Lin, Dongmei Zhang, Saravan Rajmohan, and Qi Zhang. Vem: Environment-free exploration for training gui agent with value environment model. *arXiv preprint arXiv:2502.18906*, 2025.

A. APIs for Office Applications

To demonstrate the practicality of integrating API-level control into desktop automation, we developed a set of *12* task-specific APIs across three popular Microsoft Office applications: Word, Excel, and PowerPoint. These APIs are designed to complement GUI-based interaction by enabling direct access to underlying application functions through COM interfaces.

For **Word**, the supported APIs include text and paragraph selection, font formatting, and file export. For **Excel**, the APIs provide capabilities such as table insertion, range selection, column reordering, and sheet export. For **PowerPoint**,

Table 5. APIs supported across Office applications.

API	**Application**	**Description**
`select_text`	Word	Select matched text in the document.
`select_paragraph`	Word	Select a paragraph in the document.
`set_font`	Word	Set the font size and style of selected text.
`save_as`	Word	Save the current document to a desired format.
`insert_excel_table`	Excel	Insert a table at the desired position.
`select_table_range`	Excel	Select a range within a table.
`reorder_column`	Excel	Reorder columns of a table.
`save_as`	Excel	Save the current sheet to a desired format.
`set_background_color`	PowerPoint	Set the background color of slide(s).
`save_as`	PowerPoint	Save the current presentation to a desired format.

we implement functions to change slide background colors and save presentations. Notably, a common `save_as` API is reused across all three applications, illustrating the potential for cross-application generalization.

Table 5 summarizes the available APIs and their functionalities. These APIs serve as examples of how direct function invocation can improve reliability and reduce the number of GUI actions needed, particularly for tasks involving complex formatting or structured content manipulation.