AI-Native Operating Environment for Intelligent Task Automation

Mr. Jagdish Hadaman Jangid[1], Dr. Anita Sachin Mahajan[2], Dr. Pankaj Agarkar[3]
[1]*Pursuing Masters of Engineering in Computer Engineering from Ajeenkya, DY Patil School of Engineering, Lohgaon Pune*
[2]*Assistant Professor and M.E. Coordinator in Computer Engineering from Ajeenkya, DY Patil School of Engineering, Lohgaon Pune*
[3]*Head of Computer Engineering from Ajeenkya DY Patil School of Engineering, Lohgaon Pune*

***Abstract*—As modern digital workflows become increasingly intricate, there is a growing demand for computing systems that can interpret natural language instructions, reason through complex decisions, and execute multi-step operations autonomously. Traditional operating systems and scripts excel at routine, predictable tasks; however, they quickly break down when faced with dynamic conditions, unstructured data, or complex cross-application workflows.**

To bridge this gap, AI-native operating environments embed artificial intelligence directly into the core execution engine combining large language models (LLMs), autonomous agents, application interfaces, memory modules, and real-time feedback loops. Rather than relying on manual clicks and rigid scripts, these systems translate high-level user intent into coordinated, cross-platform actions. For instance, a simple command to "generate a summary report using scattered project files" enables the system to locate the underlying data, synthesize relevant insights, and draft the final document autonomously.

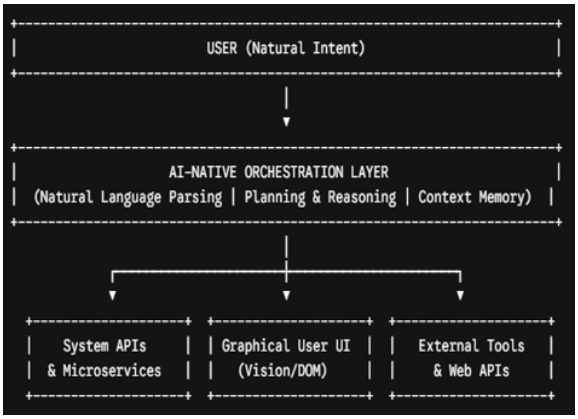
This paper explores the architecture, enabling technologies, and real-world applications of AI-native computing environments for intelligent automation. It evaluates critical challenges including execution reliability, security risks, high computational overhead, latency, and the ongoing need for human oversight. Drawing from recent developments in GUI-centric agents and computer-use benchmarks, we show that multi-step interaction across native graphical interfaces remains a major hurdle. Finally, we propose a multi-layered architectural framework designed to enhance planning safety and outline future directions toward more robust, trustworthy AI-driven computing platforms.

***Index Terms*—AI-native operating environment, intelligent task automation, large language models, AI agents, computer-use agents, GUI automation, human–AI interaction, task planning.**

I. INTRODUCTION

Digital computing has undergone a fundamental paradigm shift—moving from rigid systems requiring manual, instruction-by-instruction input toward dynamic platforms capable of assisting with high-level digital workflows. In conventional operating setups, users interface with software through structured menus, terminal commands, or pre-scripted macros. While these traditional workflows deliver high precision and predictability under fixed conditions, they quickly deteriorate when faced with unstructured inputs, natural language commands, dynamic software states, or complex cross-application dependencies.

Recent advancements in artificial intelligence offer a promising path forward. Large language models (LLMs) excel at parsing nuanced human instructions and synthesizing structured plans, while autonomous agents leverage reasoning, short/long-term memory, and tool integration to carry out multi-step operations. Rather than treating AI as a superficial add-on or isolated feature, an AI-native operating environment embeds intelligence directly into the core execution pipeline. Rather than replacing low-level kernels or underlying operating systems outright, this approach introduces an intelligent orchestration layer—an intermediary capable of mediating between user intent, application APIs, GUI elements, and system services.



Despite its potential, transitioning from theoretical agentic capabilities to real-world execution presents steep technical hurdles. Empirical studies on computer-use agents underscore this reality: for instance, evaluations on the OSWorld benchmark demonstrate that operating across native desktop environments, dynamic user interfaces, and multi-app workflows remains highly error-prone [1]. Furthermore, recent analyses emphasize that real-world viability hinges not only on baseline accuracy, but also on operational efficiency—specifically minimizing task completion latency and redundant execution steps [2].

To address these core bottlenecks, this review investigates the architecture, enabling mechanisms, and operational design principles essential for scalable AI-native environments. We focus on how these platforms parse unstructured goals, construct adaptable execution graphs, monitor runtime state, and recover from real-time errors. Finally, we establish that security, user privacy, auditability, and meaningful human supervision must serve as foundational architectural constraints rather than secondary considerations.

II. LITERATURE REVIEW

Evolution of intelligent task automation

Task automation has traditionally relied on predefined rules, scripts, workflow engines, and application programming interfaces (APIs). These methods are effective when the task structure, inputs, and expected outputs are clearly defined. For example, a script can transfer data from one spreadsheet to another or generate a report using a fixed template. However, such automation becomes less flexible when instructions are expressed in natural language, input formats change, or the task requires decisions that were not anticipated during development.

Robotic process automation (RPA) expanded the scope of automation by enabling software robots to interact with user interfaces and execute repetitive business processes. Nevertheless, conventional RPA workflows often depend on predefined steps and stable interface elements. When an application changes its layout or a process encounters an unexpected condition, the workflow may require manual correction or redevelopment.

The development of large language models (LLMs) introduced a different approach. Rather than requiring every possible instruction to be represented as a separate rule, an LLM can interpret natural-language requests and generate possible actions or execution plans. This capability has encouraged research into systems that combine language understanding with external tools, memory, feedback, and autonomous decision-making.

Wang et al. (2024), in A Survey on Large Language Model Based Autonomous Agents, examine how LLMs can serve as the central reasoning component of agents that perceive information, plan activities, use tools, and maintain memory. Their survey provides a broad foundation for understanding the shift from isolated language-model applications toward systems that interact with their environments.

However, an AI agent and an AI-native operating environment are not identical concepts. An agent may perform one bounded task inside a single application, whereas an AI-native operating environment aims to coordinate tasks, tools, application state, permissions, and execution feedback across a broader computing environment. This distinction is important for the present review because research on individual agent capabilities does not automatically demonstrate that a complete AI-native operating environment has been achieved.

The literature therefore suggests a progression from rule-based automation toward language-model-assisted and agent-based automation. The central engineering challenge is no longer simply generating a suitable response; it is ensuring that the system can translate an instruction into correct, secure, and verifiable actions.

LLM-based autonomous agents: architecture and capabilities

An LLM-based autonomous agent can be understood as a system that uses a language model to interpret goals and select actions while interacting with an external environment. Depending on its design, it may include perception, planning, tool selection, memory, execution, and feedback components.

Wang et al. (2024) organize the field of LLM-based autonomous agents around their construction, underlying components, and applications. Their work helps establish a conceptual foundation for examining agents as complete systems rather than treating the language model as the only relevant component.

A typical agent workflow includes the following stages:

1. Goal interpretation: The system identifies the intended outcome from a user instruction.
2. Context acquisition: Relevant documents, application states, and available tools are identified.
3. Task planning: The overall goal is divided into smaller, manageable operations.
4. Action selection: The agent selects an appropriate API, application, or interface operation.
5. Execution: The selected action is performed through the available tools.
6. Outcome verification: The resulting state is checked against the intended result.
7. Recovery or completion: If an action fails, the agent may revise the plan, retry within limits, or request human intervention.

These stages provide a useful framework for comparing different agent architectures. In practice, they do not always operate as a strictly linear pipeline. An agent may return to an earlier stage when new information appears or an action fails.

From an operating-environment perspective, the main implication is that agent intelligence must be supported by reliable execution infrastructure. A model may correctly understand a request but still fail because it selected the wrong file, used an invalid API parameter, misunderstood a graphical interface, or did not verify whether an operation succeeded.

Consequently, a practical AI-native environment should separate reasoning from execution and provide explicit mechanisms for tool validation, state tracking, permission enforcement, and error handling.

III. REASONING, PLANNING, AND ACTION EXECUTION

One of the most important research areas for intelligent task automation is the ability of an AI agent to convert a high-level instruction into a sequence of executable actions. A user may request an outcome without specifying every intermediate step. The agent must determine what needs to be done, which operations depend on one another, what information is missing, and whether the final result satisfies the original request.

Yao et al. (2023) introduced ReAct (Reasoning and Acting), a framework that interleaves reasoning with actions and observations. Instead of producing a complete answer without interacting with the environment, the agent can take an action, observe its result, and use the new information to decide what to do next. The authors investigated this approach on question-answering, fact-verification, and interactive decision-making tasks. Their work demonstrates the value of connecting reasoning with environmental feedback.

For example, consider a task in which an agent must locate a document, extract particular information, and prepare a summary. A reasoning-only system might generate a plausible summary based on incomplete context. A ReAct-style agent could instead search for the document, inspect the retrieved content, extract the required information, and revise its next action based on what it observes.

Huang et al. (2024), in Understanding the Planning of LLM Agents: A Survey, examine planning techniques used by language-model-based agents. Their taxonomy includes task decomposition, plan selection, external modules, reflection, and memory. This classification is particularly relevant to operating environments because these capabilities address different parts of the task-execution process.

Task decomposition allows a complex objective to be divided into smaller operations. Plan selection helps the system choose among alternative sequences of actions. External modules can provide capabilities that the language model does not possess directly, such as database access, code execution, or structured calculations. Reflection and memory can help the agent reconsider earlier decisions and retain relevant information from previous interactions.

Despite these developments, generating a plausible plan does not guarantee successful execution. Plans may contain invalid steps, overlook dependencies, or assume that an application supports an operation that is unavailable. An agent can also lose track of the current application state during a long workflow.

Therefore, the literature supports an important design principle for AI-native operating environments: planning should be connected to execution monitoring and state validation. Rather than assuming that every planned operation will succeed, the environment should verify important intermediate outcomes and permit the plan to be revised when necessary.

IV. TOOL INTEGRATION AND APPLICATION ORCHESTRATION

An AI agent becomes more useful for task automation when it can interact with external tools instead of relying exclusively on generated text. Tool integration allows the system to retrieve information, perform calculations, access databases, modify files, and invoke application functions.

Schick et al. (2023), in Toolformer: Language Models Can Teach Themselves to Use Tools, investigate how language models can learn when and how to call external tools. Their work addresses the limitations of relying on a model's internally generated knowledge for tasks that require external information or specialized operations.

Tool use can be implemented through several mechanisms:

- API-based integration: The agent calls structured application functions with defined inputs and outputs.
- Command-based integration: The agent invokes approved commands or scripts through a controlled execution layer.
- Graphical interface interaction: The agent observes and operates application interfaces using screenshots, visual grounding, mouse actions, and keyboard input.
- Workflow orchestration: The system coordinates multiple tools in a particular sequence, passing validated outputs from one operation to the next.

API-based interaction is often easier to validate because requests and responses follow defined structures. However, APIs may not expose every function available to a user. GUI interaction can support applications without suitable APIs, but it introduces additional difficulties, including visual ambiguity, layout changes, and uncertainty about whether an action has taken effect.

Li's review, A Review of Prominent Paradigms for LLM-Based Agents: Tool Use (Including RAG), Planning, and Feedback Learning, published in the proceedings of COLING 2025, provides a unified way to examine tool use, planning, and feedback-oriented approaches. It highlights the need to understand these techniques as interacting parts of an agent workflow rather than as entirely independent capabilities.

For an AI-native operating environment, the orchestration layer should act as a controlled intermediary between the agent and the underlying applications. It should maintain a registry of available tools, validate parameters, enforce access permissions, manage execution order, and capture results.

This arrangement also supports modularity. A new application connector can be introduced without redesigning the entire reasoning component, provided that the connector follows the environment's interface and security requirements.

Nevertheless, tool integration creates a new class of risks. An incorrectly selected function or malformed request may modify the wrong record, send an unintended message, or overwrite data. The system must therefore distinguish between proposing an action and authorizing its execution. Sensitive or irreversible operations should be subject to explicit validation and, where appropriate, human approval.

V. COMPUTER-USE AGENTS AND OPERATING-SYSTEM INTERACTION

A major development in intelligent task automation is the study of agents that interact directly with desktop and web applications. Such systems go beyond text-based assistance by performing actions in a real computing environment.

Xie et al. (2024) introduced OSWorld: Benchmarking Multimodal Agents for Open-Ended Tasks in Real Computer Environments at NeurIPS 2024. The benchmark includes 369 tasks involving real applications, file operations, and workflows spanning multiple applications. It supports execution-based evaluation across desktop environments rather than

relying only on the quality of an agent's textual response.

A particularly important finding reported in the OSWorld paper is the gap between human performance and the performance of the evaluated computer-use agents. The authors report that humans achieved more than 72.36% task success, whereas the best evaluated model achieved 12.24% under their experimental conditions. These figures are specific to the benchmark and model evaluations in that study; they should not be interpreted as universal performance estimates for all current AI agents.

This result highlights the difficulty of translating language understanding into reliable operations in real applications. Computer-use agents must correctly identify interface elements, understand the current state, choose appropriate actions, and respond to unexpected changes. Even a small error early in a workflow may cause later steps to fail.

For example, an agent asked to update a spreadsheet and prepare a report may need to locate the correct workbook, identify the relevant worksheet, distinguish between headers and data, edit the intended cells, save the changes, and verify the resulting file. Completing only some of these steps does not necessarily satisfy the user's original goal.

The OSWorld study therefore provides evidence that benchmark evaluation must examine actual task outcomes. A generated explanation that claims success is insufficient when the required file was not saved or the requested

application state was not achieved.

For the proposed AI-native operating environment, this research motivates three design requirements:

1. State awareness: The environment must maintain an accurate understanding of the current application and task state.
2. Execution verification: Important operations must be checked against observable results.
3. Recovery mechanisms: Failed actions should trigger bounded retries, alternative plans, or requests for human intervention.

These requirements also suggest that an AI-native operating environment should not depend exclusively on GUI interaction. Where possible, it should select reliable APIs or structured interfaces and use GUI automation when those alternatives are unavailable or insufficient.

Comparative Analysis of Existing Approaches: The literature reviewed above addresses different components of intelligent task automation. The following table synthesizes their contributions and limitations for the proposed research topic.

Research approach	Main contribution	Remaining limitation for an AI-native environment
Wang et al. (2024) — autonomous-agent survey	Organizes agent architecture, applications, and evaluation	Broad agent coverage does not by itself establish a complete operating-environment design
Yao et al. (2023) — ReACt	Connects reasoning, actions, and observations	Does not independently solve all state-management and execution-safety problems
Huang et al. (2024) — planning survey	Classifies task decomposition, plan selection, external modules, reflection, and memory	Planning quality does not guarantee that actions will succeed in real applications
Li (2025) — tool-use and planning review	Unifies several major paradigms in LLM-agent workflows	Integration still requires reliable interfaces, permissions, and state validation
Zhang et al. (2024) — memory survey	Examines memory mechanisms for LLM-based agents	Memory must be kept relevant, secure, and consistent with current application state

Research Gap: Recent advancements in large language models (LLMs), autonomous agents, tool-use frameworks, and computer-use automation have created new opportunities for intelligent task execution. Existing research has investigated several essential capabilities, including natural-language understanding, task planning, external tool integration, memory management, and interaction with graphical user interfaces. Studies such as ReAct by Yao et al. (2023) and OSWorld by Xie et al. (2024) demonstrate the importance of combining reasoning with environmental interaction and evaluating agents through the successful completion of real computer tasks. However, the availability of these individual capabilities does not necessarily ensure reliable end-to-end automation across a complete computing environment.

A review of the existing literature highlights several areas that require further investigation.

Limited integration of agent capabilities into a unified operating environment

Existing studies frequently focus on individual aspects of intelligent agents, such as planning, memory, tool selection, or computer interaction. Although these capabilities contribute to automation, their coordination within a unified environment remains an

important engineering challenge. An effective AI-native operating environment must connect user-intent interpretation, task decomposition, application orchestration, controlled execution, and outcome verification. Further investigation is required to determine how these components can be integrated while maintaining modularity, scalability, and consistent system behaviour.

VI. RELIABILITY OF MULTI-STEP TASK EXECUTION

Many real-world tasks require a sequence of dependent operations across multiple applications. An error in an early step may affect subsequent actions and produce an incorrect final result. Existing computer-use benchmarks, including OSWorld, demonstrate the difficulty of completing tasks reliably in real application environments. This creates a need for stronger mechanisms for intermediate-state validation, failure detection, bounded retries, and recovery planning. In particular, further research is needed into how an operating environment can verify that the intended outcome has been achieved rather than relying solely on an agent's generated response.

Coordination between application interfaces and intelligent agents

AI agents may interact with applications through APIs, command-line interfaces, or graphical user interfaces. Each mechanism has different advantages and limitations. APIs offer structured interactions when suitable endpoints are available, whereas graphical interfaces can provide access to functions that are not exposed through APIs. However, GUI-based automation is sensitive to interface changes, visual ambiguity, and unexpected application states. A research opportunity exists in developing an orchestration mechanism that can select appropriate interaction methods, validate their results, and switch to alternative methods when execution fails.

Insufficient coordination between automation and security controls

Autonomous agents may require access to files, application data, communication tools, and external services. This creates risks involving excessive permissions, malicious instructions embedded in external content, unintended modifications, and disclosure of sensitive information. Although security research has examined prompt injection and agent-level defences, security must also be enforced at the operating-environment level. Further investigation is required into context-sensitive permissions, risk-based human approval, execution isolation, auditability, and safe recovery mechanisms that remain effective throughout a multi-step workflow.

Lack of a unified, multi-dimensional evaluation approach

Agent performance is commonly assessed through task success, benchmark scores, or individual capability tests. However, an AI-native operating environment must satisfy several requirements simultaneously, including accuracy, execution time, computational cost, recovery capability, and security. A system that completes a task quickly but performs an unauthorized operation cannot be considered effective. This motivates the development of an evaluation methodology that considers functional correctness and operational efficiency alongside safety and reliability under consistent testing conditions.

Managing contextual information and execution state

Memory mechanisms can help agents retain task history, relevant knowledge, and user preferences. However, information retained in a language model's context is not necessarily an accurate representation of the current application state. For example, an agent may remember that it attempted to save a file without confirming that the file was actually saved. Further research is therefore needed into the coordination of contextual memory, structured workflow state, and direct verification against application data. Such coordination could improve continuity across long-running tasks while reducing errors caused by outdated or incomplete information.

VII.RESEARCH MOTIVATION

The above gaps indicate that intelligent task automation requires more than improvements to the reasoning capability of individual language models. It requires a coordinated environment that connects planning, tool orchestration, execution control, memory, verification, and security. The present review therefore focuses on examining a layered AI-native operating environment in which these functions are treated as interconnected components of a complete task-execution lifecycle.

The proposed research direction is to investigate how a modular, security-aware architecture with explicit state validation and feedback mechanisms could support more reliable and efficient task automation across multiple applications. Its effectiveness would need to be established through a systematic comparison with suitable baseline approaches using predefined tasks and measurable evaluation criteria.

AI-Native Operating Environment Architecture: An AI-native operating environment reorganizes classical computing layers around an artificial intelligence core. Rather than treating AI as a superficial application running on top of a conventional operating system, an AI-native system positions large language models (LLMs), autonomous agents, and dynamic execution graphs as central middleware orchestrators. This design bridges the gap between high-level user intent and low-level system execution.

- Paradigm Shift: Moves away from rigid, pre-scripted workflows toward goal-oriented, adaptive computing.
- Orchestration Core: Uses AI as an intelligent intermediary that bridges user prompts with underlying system tools, APIs, and graphical interfaces.
- Holistic Integration: Combines reasoning, memory, tool use, and safety protocols into a unified computing framework.

High-Level Architectural Layers

The architecture is structured as a multi-layered stack, where each layer handles specific responsibilities ranging from intent capture to hardware execution:

- User Intent & Multimodal Interface Layer:
- Accepts high-level natural language requests (e.g., "Compile research paper summaries, generate a comparison chart, and attach them to an email draft").
- Parses inputs across multiple modalities, including text, voice, visual screenshot capture, and spatial context.

Cognitive Orchestration Engine:

- Planner & Task Decomposition: Breaks down unstructured user goals into Directed Acyclic Graphs (DAGs) of executable sub-tasks.
- Context & Memory Architecture: Utilizes short-term working memory for multi-step execution loops (e.g., ReAct, Plan-and-Solve) and long-term memory (vector/graph stores) for user preferences and historical workflows.
- Reflection & Fault Recovery: Evaluates tool outputs or GUI screenshots to verify success and automatically re-routes execution if errors occur.

Agentic Execution Middleware:

- Tool & API Integration Engine: Invokes structured system calls, web services, microservices, and command-line interfaces.
- Computer-Use / GUI Agent Engine: Interacts directly with native graphical applications via computer vision, Accessibility Tree (a11y) parsing, and simulated mouse/keyboard events when formal APIs are absent.

Environment State Monitoring: Tracks process completion, file system mutations, and active window states in real time.

Governance, Safety & Security Boundary:

- Enforces least-privilege permissions based on user security scopes and sandboxed execution zones.
- Implements Human-in-the-Loop (HITL) gatekeepers to pause execution and request user approval for high-risk actions (e.g., financial transactions, file deletions, or external communications).
- Maintains rigorous audit logs tracking decision rationale and execution chains.

Underlying Operating System & Hardware:

- Traditional OS kernels (Linux, Windows, macOS) managing hardware resources, memory allocation, low-level drivers, and process scheduling.

Comparison: Traditional vs. AI-Native Environments

The shift from traditional to AI-native computing fundamentally changes how software components interact:

- Interaction Model:
  - Traditional: Relies on command-line scripts, manual UI clicks, and strict menu navigation.
  - AI-Native: Driven by natural language expressions and goal-oriented directives.

Task Execution:

- Traditional: Rigid, pre-scripted, and deterministic workflows.
- AI-Native: Adaptive execution graphs featuring real-time planning and dynamic pathfinding.

Cross-Application Integration:

- Traditional: Dependent on custom software integrations and static API connectors.
- AI-Native: Unified multimodal agent orchestration capable of bridging disparate APIs and graphical user interfaces natively.

Error Handling & Recovery:

- Traditional: Handled via hardcoded exception catches and rigid fallback scripts.
- AI-Native: Employs agentic reflection, visual verification, self-correction, and iterative retry policies.

Primary Technical Challenges

Developing robust AI-native operating environments requires overcoming several persistent technical bottlenecks:

- Execution Reliability & Latency: Ensuring that multi-step agent actions execute accurately without compounding errors or introducing excessive processing delays.
- GUI Interaction Robustness: Maintaining stable execution when dealing with dynamic UI layouts, unexpected pop-ups, and non-standard visual elements across desktop applications.
- State Management & Consistency: Synchronizing the agent's internal cognitive memory with the actual, fast-changing state of the real-world computing environment.

Challenges and Limitations: The development of an AI-native operating environment introduces several technical and operational challenges. Although large language models (LLMs) and autonomous agents can interpret natural-language instructions, plan actions, and interact with external tools, their integration into a reliable computing environment remains complex. Unlike conventional automation, which follows predefined rules, AI-driven automation must handle uncertainty, changing application states, incomplete information, and unexpected outcomes.

The following section examines the major challenges and limitations relevant to the design of AI-native operating environments for intelligent task automation.

Reliability and Accuracy of Task Execution

One of the primary challenges in AI-native operating environments is ensuring that automated tasks are executed correctly and consistently. LLM-based agents may generate incorrect assumptions, select inappropriate tools, or produce execution plans that do not match the actual application state. Although an agent may generate a response indicating successful completion, the intended operation may not have been performed.

This problem becomes more significant in multi-step workflows, where the output of one operation serves as the input to the next. An error during data extraction, file selection, or application interaction may propagate through subsequent steps and affect the final result.

Existing research on computer-use agents, including the OSWorld benchmark, highlights the difficulty of completing tasks reliably in real application environments. Therefore, AI-native operating environments require execution monitoring, intermediate-state validation, result verification, and controlled recovery mechanisms. However, verification itself can introduce additional processing time and computational overhead.

Complexity of Task Planning and Decision-Making

Natural-language instructions often describe the desired outcome without specifying every intermediate operation. The system must interpret the request, identify dependencies, divide the task into manageable subtasks, and select an appropriate execution sequence.

LLM-based planning can be affected by ambiguous instructions, incomplete context, limited reasoning reliability, and incorrect assumptions about available tools. Furthermore, an initially suitable plan may become invalid when an application returns unexpected results or when external conditions change during execution.

Dynamic replanning can help address these situations, but repeated planning may increase latency and resource consumption. An additional limitation is that a generated plan may appear logically consistent while

still containing actions that are unavailable or inappropriate for the target environment. Consequently, planning mechanisms must be connected to tool validation and feedback from actual execution.

Integration with Existing Applications and Systems

An AI-native operating environment may need to interact with office applications, databases, web browsers, enterprise software, file systems, and external services. These systems often differ in their interfaces, authentication mechanisms, data formats, and operational constraints.

Application programming interfaces (APIs) provide structured access to supported functions, but not every application exposes all required operations. Graphical user interface (GUI) automation can extend the range of accessible applications, although it may be sensitive to changes in screen layout, element positions, loading delays, and interface behaviour.

Integrating heterogeneous applications also creates challenges involving data consistency, session management, permissions, and error handling. A common orchestration layer can simplify integration, but developing and maintaining reliable connectors for different applications requires considerable engineering effort.

Security and Privacy Risks

AI-native operating environments may receive access to sensitive documents, emails, user accounts, business records, and other digital resources. Such access increases the potential consequences of incorrect decisions or malicious inputs.

A significant concern is indirect prompt injection, in which an agent encounters malicious instructions embedded in external content such as documents, emails, or web pages. If the agent treats these instructions as trusted commands, it may perform actions that conflict with the user's original intent.

Other risks include excessive permissions, unauthorized data disclosure, insecure tool execution, and unintended modification or deletion of files. Privacy concerns also arise when information is transmitted to external model providers or retained in memory beyond its intended use.

These risks cannot be addressed through model reasoning alone. Effective safeguards require least-privilege access, input validation, execution isolation, secure credential management, audit logs, and explicit approval for sensitive or irreversible actions. Nevertheless, designing security controls that remain effective across diverse applications and long-running workflows remains a significant challenge.

Computational Cost and Execution Latency

Intelligent task automation may require several models calls for instruction interpretation, planning, tool selection, error analysis, and result verification. Complex workflows may also involve repeated retrieval operations, large contextual inputs, and multiple application interactions.

These activities can increase computational cost, energy consumption, and execution time. For interactive tasks, excessive latency may reduce usability. In enterprise environments, high inference costs may also limit the feasibility of deploying agents across large numbers of users or workflows.

Smaller models, caching, task decomposition, and selective use of more capable models may reduce overhead. However, optimization must not compromise the correctness or security of the resulting actions. A practical AI-native environment therefore needs to balance task success, resource consumption, and response time according to the requirements of each ta

Conclusion and Future Scope: AI-native operating environments represent a paradigm shift in computing, moving away from rigid, pre-scripted workflows toward intelligent, goal-oriented automation that interprets human intent and coordinates complex digital tasks. By embedding large language models, autonomous agents, memory mechanisms, and feedback loops directly into the computing stack, these systems bridge the gap between high-level user directives and execution across diverse software interfaces. However, as highlighted by recent benchmarking and research, realizing this vision requires overcoming substantial technical hurdles including compounding error rates in multi-step execution, the brittleness of GUI and computer-use interactions, high computational latency, and critical security vulnerabilities like indirect prompt injection.

Looking forward, the future scope of AI-native operating environments lies in the development of robust, multi-layered architectural frameworks that prioritize execution reliability, deterministic verification, and transparent decision-making. Future

research must focus on advancing hybrid reasoning models that couple neural flexibility with symbolic verification, improving real-time visual-state synchronization, and establishing fine-grained, least-privilege security sandboxes. Furthermore, striking an optimal balance between full agentic autonomy and seamless human-in-the-loop oversight will be essential to ensure that next-generation computing environments are not only exceptionally efficient, but also inherently safe, auditable, and trustworthy for everyday deployment.

REFERENCES:

Citations

Research Topic / Focus Area in Paper	Suggested In-Text Citation Mapping
LLM Agent Architecture	[1], [7]
Reasoning and Action Planning	[2], [8]
Application and Operating-System Interaction	[3], [4]
Tool Integration	[5], [7]
Task Reliability and Evaluation	[3], [4]
Prompt Injection and Excessive Autonomy	[6]
Overall Limitations and Future Research	[1], [3], [7]

"Although LLM-based agents have demonstrated the ability to combine reasoning, planning, and external tool use, reliable execution across real computer environments remains challenging [1]–[4]. Furthermore, granting agents access to external tools introduces security risks, including prompt injection and excessive agency, which require system-level safeguards [6]."

REFERENCES:

[1] Wang, L., et al., "A survey on large language model based autonomous agents," Frontiers of Computer Science, vol. 18, Art. no. 186345, 2024. [Online]. Available: https://doi.org/10.1007/s11704-024-40231-1

[2] Yao, S., et al., "ReAct: Synergizing reasoning and acting in language models," in Proceedings of the International Conference on Learning Representations (ICLR), 2023. [Online]. Available: https://arxiv.org/abs/2210.03629

[3] Hu, X., et al., "OS Agents: A Survey on MLLM-based Agents for Computer, Phone and Browser Use," in Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (ACL), 2025, pp. 7436–7465, doi: 10.18653/v1/2025.acl-long.369. [Online]. Available: https://aclanthology.org/2025.acl-long.369/

[4] Xie, T., et al., "OSWorld: Benchmarking multimodal agents for open-ended tasks in real computer environments," in Advances in Neural Information Processing Systems (NeurIPS), vol. 37, 2024. [Online]. Available: https://arxiv.org/abs/2404.07972

[5] Schick, T., et al., "Toolformer: Language models can teach themselves to use tools," in Advances in Neural Information Processing Systems (NeurIPS), vol. 36, 2023. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2023/hash/d842425e4bf79ba039352da0f658a906-Abstract-Conference.html

[6] OWASP Foundation, "OWASP Top 10 for Large Language Model Applications," 2026 release. [Online]. Available: https://owasp.org/www-project-top-10-for-large-language-model-applications/

[7] Chowa, S. S., et al., "From language to action: A review of large language models as autonomous agents and tool users," Artificial Intelligence Review, vol. 59, Art. no. 71, 2026, doi: 10.1007/s10462-025-11471-9. [Online]. Available: https://doi.org/10.1007/s10462-025-11471-9

[8] Huang, X., et al., "Understanding the planning of LLM agents: A survey," arXiv:2402.02716, 2024. [Online]. Available: https://arxiv.org/abs/2402.02716