LLMs as an Abstraction Layer in Computer Organization: Implementation and Limitations

Hugo Hoffmann[1], **Carollini Maria P. Spesse**[2]

[1]Departamento de Engenharia, Centro Universitário Afya-Itaperuna

[2]Engenharia de Computação do CEFET-MG - Campus Leopoldina

hugo.borges@afya.com.br, carollinispesse@gmail.com

Abstract. *This paper proposes that Large Language Models (**LLMs**) can be viewed as a new abstraction layer capable of addressing an intrinsic limitation of traditional presentations of computer organization: the absence of a mechanism that directly connects natural language to the electronic operations performed by hardware. We argue that, although high-level programming languages provide significant abstractions over processor architecture, they do not fully achieve the theoretical objective of translating human commands into electrical signals. We demonstrate how this gap can be addressed through the Model Context Protocol (MCP), which facilitates mediation between commands expressed in natural language and the operating system. Finally, we discuss the problems and vulnerabilities introduced by this new layer, as well as mitigation strategies associated with its use.*

1. Introduction

From an abstract perspective, the central problem of the field of computer organization is to define a translation method that enables communication between humans, who speak natural languages such as Portuguese or English, and computers, whose operation is based on electrical signals applied to electronic circuits. Formally, we have:

$$f : \mathcal{L}_N \rightarrow \mathcal{S}$$

where f denotes a theoretical translation function that maps commands expressed in a natural language $\mathcal{L}_N$ into a sequence of electrical signals $\mathcal{S}$, which are then applied to the electronic circuits that constitute the hardware.

An evident difficulty in constructing such a function lies, as noted by [Tanenbaum 2013], in the substantial difference between $\mathcal{L}_N$ and $\mathcal{S}$. Even simple commands such as ”make a directory” or ”search for a specific file”, expressed in $\mathcal{L}_N$, do not have a straightforward correspondence in terms of electrical signals. The solution presented in most textbooks, e.g., [Tanenbaum 2013], consists of decomposing the translation function f into several smaller processes, known as abstraction layers.

Systematic presentations of abstraction layers, e.g., [Tanenbaum 2013] and [Stallings 2021][1], typically, enumerate five layers. In a *bottom-up* approach, these are: the

[1]Tanenbaum explicitly defines the division that we adopt here, whereas Stallings employs a similar separation in a less systematic manner.

digital-logic layer, which describes the organization of fundamental logical components such as gates and registers from which the hardware is built; the microarchitecture layer, responsible for translating macroarchitecture instructions into electrical control signals that coordinate the operation of the processor circuits; the macroarchitecture layer, which defines the instruction set (**ISA**) that the processor can execute; the operating system layer, which manages hardware resources and provides abstractions such as processes, virtual memory, and file systems; and finally the high-level language layer, which allows users to write programs independently of the details of the processor architecture and the particularities of the underlying hardware.

An attentive reader may already have noticed that one element is missing from this layered scheme. Although high-level languages such as C or Python provide a syntax that resembles natural language, they do not satisfy the objective established by the function f. There therefore remains a gap between natural languages $\mathcal{L}_N$ and the electronic operations actually performed by the hardware.

In this article, we propose that **LLMs** can serve as a new abstraction layer, bridging natural languages and the operating system. In Section 2, we define an agent to show how this layer can be implemented using the MCP (Model Context Protocol). This lets models interact with the operating system through natural language commands provided by the user. Section 3 discusses problems and vulnerabilities of LLMs that may significantly affect the operating system. Finally, Section 4 compares our findings with other proposals in the literature.

2. LLM as an Abstraction Layer

Despite the advances that led to the development of **LLMs**, commercial models like ChatGPT, Claude, and Gemini show limited usefulness on their own. They mostly generate and transform text and lack direct access to current data, code repositories, or operating system functions. To make them more relevant and enable real productivity gains, thus justifying high costs for training and infrastructure, it became necessary to find ways to integrate them with external tools, applications, and computational services.

Until recently, the primary method for integrating language models with applications relied on **APIs** ([OpenAI 2023]), a well-established mechanism in web and mobile system development. In this paradigm, **APIs** function as interfaces that standardize communication between systems, typically through HTTP requests, defining endpoints, input and output formats, and explicit interaction rules. Although functional, this approach has revealed significant limitations when applied to communication between **LLMs** and software systems. Among the most notable issues are the limited scalability of integrations, since each additional tool requires new code and dedicated maintenance, as well as challenges associated with the implementation and usage of orchestration frameworks ([Aljohani and Do 2025]).

In this context, in November 2024, [Anthropic 2024] introduced the **(MCP)**, an open-source protocol designed to standardize the interaction between **LLMs** and external software. The purpose of MCP is to abstract away the particularities of individual integrations by providing a uniform interface for accessing data, tools, and software capabilities.

The **MCP** protocol adopts a *client–server* architecture. The language model acts as an MCP client, while external applications expose their capabilities through MCP

servers. Each server explicitly describes the resources it provides, referred to as *tools*, and defines the request input schemas, i.e., how the **LLM** must request the use of these tools, as well as the corresponding output schemas that specify how the server responds to those requests. Communication between client and server occurs through messages structured in JSON, enabling the model to request actions, query data, or execute operations in a standardized manner, independent of the internal implementation of the software.

In little more than a year, the number of MCP servers has grown exponentially. With only a few clicks and minimal configuration, it has become possible to integrate **LLMs** with a wide range of tools, from code-oriented IDEs such as Cursor, Claude Code, and Copilot to advanced reverse engineering platforms such as Ghidra ([LaurieWired 2024])[2]. There are even MCP servers that expose direct operations on the execution environment to the model, including file system access, command execution, and graphical interface automation, with implementations targeting the Windows *desktop* environment (e.g., the CursorTouch MCP server) ([CursorTouch 2025]). In the browser domain, MCP servers based on extensions already exist that expose Chrome functionalities to MCP clients, as well as official initiatives integrating Chrome DevTools debugging capabilities into agents through an MCP server ([Hangwin 2025, Chrome Developers 2025]).

In this section, we demonstrate how MCP servers can be used to expose operating system functionalities to an **LLM**, enabling the model to perform concrete actions on a computer. Our goal is to show that, by acting upon the operating system, language models can assume the role of an effective abstraction layer capable of completing the translation between commands expressed in natural language and the computational operations executed by the computer.

2.1. A Naive Exemple

Before presenting the demonstration, we describe the testing environment adopted and justify the choices made. The construction of this environment requires three decisions: i) which **LLM** will act as the MCP client; ii) which MCP server will be responsible for executing the commands selected by the model; and iii) on which operating system these commands will run.

As the **LLM**, we chose to use Claude Desktop. According to the authors' experience, Claude Desktop proved to be a more stable and easier-to-configure alternative when compared to other commercial solutions or locally executed models in the context of MCP server integration. Although it is now easy to combine MCP servers with any LLM, including open-source ones.

There are several implementations of MCP servers capable of controlling operating systems such as Linux, Windows, and macOS at different levels, we decided to develop our own simple MCP server to illustrate the process. This decision is justified by two reasons. The first is pedagogical in nature. The server we developed contains about 100 lines of code and simply exposes a shell through which Claude can select and

[2]For a practical tutorial, see "ghidraMCP: Now AI Can Reverse Malware" (https://www.youtube.com/watch?v=u2vQapLAW88) and "Integrando o Ghidra com o Claude utilizando o protocolo MCP" (https://www.youtube.com/watch?v=H9yoL8bA1ck).

execute commands. It is therefore sufficiently simple for students to understand its operation and to use the code as a basis for developing more sophisticated servers. The second reason relates to issues discussed in Section 3. An intentionally simple server, lacking validation or security mechanisms, is appropriate for the demonstration of these vulnerabilities[3]. The interested reader may consult the source code of the MCP server in the author's GitHub [4]. Finally, we chose to use a simplified version of Debian running inside a Docker container. The reasons for not executing commands directly on the host operating system will be discussed in the next section. The *Dockerfile* used is also available in the same Git repository.

We now demonstrate how a **LLM** can operate over an operating system through commands expressed in natural language, with **MCP** acting as an intermediary. Our goal is to present, through concrete examples, how the model begins to function as an abstraction layer above the operating system and high-level programming languages.

Before proceeding to the demonstration, a few additional configurations are required. When Claude Desktop is launched, a Docker container is instantiated. By default, this container is terminated once the instructions requested by the model have been executed. To allow inspection of the actions performed, we identify the container instantiated by Claude and access its terminal. This step allows us to directly observe the effects of the actions proposed by the **LLM** within the execution environment.

Claude's standard behavior is to determine which commands should be forwarded to the MCP server. The execution of these actions can be configured either to occur automatically or to require explicit user authorization. In the experiments presented below, we chose to require manual confirmation for each request in order to make the decision flow explicit.

For this demonstration, we asked the **LLM** to create a `bash` script responsible for updating the packages installed on the system on a daily basis. The prompt used in this test is shown in Figure 1.

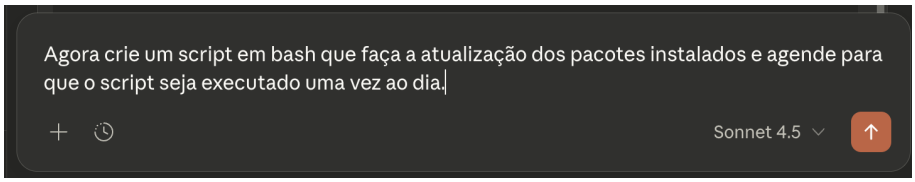

Figure 1. Asking for a script for daily auto update and upgrade

Within a few seconds, Claude generates a script capable of updating the system packages using `apt` and recording the update process in a log file. The JSON sent by the MCP Server was the following:

```
{    'command': 'cat > update_packages.sh << 'EOF'
     #!/bin/bash
     LOG_FILE=\"/var/log/system_update.log\"
     echo \"==================================\" >> $LOG_FILE
```

[3]Due to page limitations, a more practical demonstration of these vulnerabilities will be presented in future work.

[4]https://github.com/hoffmann-topos-lab/SimpleOSMCPServer

```
    echo \"Atualização iniciada em: $(date)\" >> $LOG_FILE
    echo \"================================\" >> $LOG_FILE
    apt-get update >> $LOG_FILE 2>&1
    apt-get upgrade -y >> $LOG_FILE 2>&1
    echo \"Atualização concluída em: $(date)\" >> $LOG_FILE
    echo \"\" >> $LOG_FILE
    EOF `}
```

Although the generated script is relatively verbose, it is functional and follows common system administration practices. The model then issues a second request to grant execution permission to the script, verify whether the crontab exists, and add the script to the `crontab` so that it is executed every day at 2 a.m.:

```
{`command`: `chmod +x update_packages.sh`}
{`command`: `(crontab -l 2>/dev/null; echo \"0 2 * * * $(pwd)/
   update_packages.sh\") | crontab -`}
```

As we can see, it was possible to generate a simple and functional script for performing administrative actions on the system. These examples illustrate how an **LLM**, through the MCP protocol, can assume the role of an abstraction layer in computer organization, since it can perform an additional step that improves the interaction between humans and machines. To show that **LLMs** can actually fulfill the theoretical function f, we need to find a more robust set of tasks to measure the performance of these models.

2.2. OSWorld: A Robust Metric for OS Agents

The example presented previously is merely illustrative and does not allow us to measure how efficient and robust the presented AI agent actually is, nor does it provide an adequate metric for comparing other agents that rely on more robust MCP servers.

OSWorld is a benchmark proposed in [Xie 2024] that presents 369 common tasks typically performed by humans in operating systems, both through GUI and CLI interfaces, and submits them to the agent, which must attempt to solve them. These tasks range from simple activities, such as centering columns in spreadsheets or writing emails, to more complex tasks involving multiple programs, such as creating a GIF in GIMP and adding it to a video played in VLC, or creating a spreadsheet, attaching it to an email, and sending it through a web browser.

The article [Xie 2024] evaluates both commercial models, such as ChatGPT-4, Claude-3 Opus, and Gemini, and *open-source* models, such as Qwen-Max, Mixtral, and Llama-3. The models are evaluated not only by their ability to solve the proposed tasks, but also by the time required to complete them. During the testing period, the models achieved scores ranging from 0.99% to 12.24%, while humans achieved a performance of 72.36%[5].

Surprisingly, approximately one and a half years after the release of the benchmark, commercial models such as Claude Opus, according to [Anthropic 2026], improved their performance from approximately 12% to results surpassing average human performance, reaching 72.7% in the Opus 4.6 and Sonnet 4.6 versions.

[5]Human performance was measured through the execution of the tasks by graduate students without prior knowledge of the activities that should be performed.

2.3. Reflection

Although the conception of language models as an additional abstraction layer is both viable and useful, both from the theoretical perspective of computer organization and in practical use by non-specialized users, **LLMs** should not be treated as oracles. As noted by [Turing 1950], there is a recurring human tendency to attribute epistemic authority to computational systems, particularly when they produce correct or plausible responses, even when their internal mechanisms are not fully understood by the user.

However sophisticated they may be, language models essentially consist of highly parameterized probabilistic functions trained to estimate responses based on distributions learned from training data. Like any probabilistic system, **LLMs** are subject to errors. In contexts where such models interact directly with the operating system, these errors may have serious consequences, including the deletion of critical files, insecure configurations, or the generation of vulnerable code. In the next section, we discuss the nature of these errors and their principal causes, focusing on hallucinations and vulnerabilities inherent to the use of **LLMs**.

3. Problems and Limitations

In this section, we briefly review the literature discussing the causes that lead **LLMs** to produce errors, as well as the vulnerabilities associated with language models from a general perspective. We then analyze how these factors affect our thesis that AI agents can operate as an abstraction layer in the context of computer organization.

3.1. Stochastic Errors

Given the probabilistic nature of the neural networks underlying language models, it is expected that generated responses will be subject to errors. For a given *input* i and its corresponding context c, there exists an objectively correct answer to the problem contained in the *input*. This answer can be represented by a target function $f_{\text{true}}(i, c)$. By contrast, the language model produces stochastic outputs corresponding to samples drawn from a probability distribution conditioned on the pair (i, c).

Let Y be the random variable associated with the model output, with distribution $Y \sim p_\theta(\cdot \mid i, c)$. The stochastic error corresponds to the discrepancy between the correct answer $f_{\text{true}}(i, c)$ and the responses effectively generated by the model. This error can be formalized as the expected value of a loss function ℓ, defined over the model output and the correct answer, that is:

$$\text{Error}(i, c) = E_{Y \sim p_\theta}\big[\ell\big(Y, f_{\text{true}}(i, c)\big)\big].$$

In particular, when the discrepancy is modeled using an absolute distance, the error takes the form.

$$\text{Error}(i, c) = E_{Y \sim p_\theta}\big[|Y - f_{\text{true}}(i, c)|\big].$$

This type of error is inherent to language models and can be mitigated but not eliminated. Improvements in *machine learning* algorithms and the expansion of training

datasets can reduce its magnitude. Nevertheless, the error asymptotically approaches a value different from zero due to the probabilistic nature of the model outputs.

In the context of this investigation of **LLMs** as an abstraction layer, the impact of such errors depends strongly on the clarity of the *input* provided by the user and on the manual validation of the actions executed by the system. A properly designed *system prompt*, capable of identifying ambiguity or inconsistencies in the *input* and explaining the intended action to a non-expert user, can anticipate and mitigate a significant portion of these issues.

3.2. Hallucinations

The second type of error we address is hallucination. Unlike stochastic errors, this phenomenon occurs when language models generate content that is factually incorrect while presenting it with a high degree of confidence. We may illustrate hallucinations with the following scenario:

1. We ask a model: "When is Chico's birthday?[6]"
2. The model promptly responds with confidence with a specific date, for example, February 12.
3. However, the answer is incorrect, since Chico was actually born on May 17.

In this subsection, we examine types of hallucinations and provide a non-exhaustive overview of their possible causes.

Authors such as [Ji et al. 2023] and [Maynez et al. 2020] propose a taxonomy in which hallucinations can be classified into two categories: intrinsic and extrinsic. An intrinsic hallucination occurs when the content generated by the model directly contradicts information present in its training data. In this case, the hallucination employs concepts that are "known," but manipulates them incorrectly, producing a statement that is false with respect to the original data. This type of hallucination resembles the stochastic error described in the previous subsection.

Extrinsic hallucinations involve information that cannot be verified because it is not contained in the training data, such as questions about the birth date of unknown individuals. In addition, this type of hallucination may even be factually correct while still being considered a hallucination because it is not grounded in the indicated source. For example, when requesting a summary of a specific book, the model may include accurate information that actually belongs to a different work.

Another possible explanation for hallucinations is captured by the concept GIGO (Garbage In, Garbage Out). As noted by [Kalai et al. 2025], large training datasets frequently contain errors, contradictory information, or outdated data. When the model is trained on such "garbage" (garbage in), it tends to reproduce these flaws in its outputs (garbage out). Consequently, the system learns incorrect patterns, which contributes to the emergence of hallucinations.[7] The GIGO concept can also apply to user inputs. If a

[6]Chico is a beagle that belongs to one of the authors.

[7]If this concept is taken to an extreme, as in the study by [Lin et al. 2022], humans outperform language models when answering questions based on common misconceptions. In the experiment, human accuracy reached 94%, whereas the best model achieved only 58%, often reproducing incorrect answers that frequently appear on the internet.

user provides a prompt lacking sufficient contextual clarity, the resulting answer is likely to be incorrect or imprecise.

An additional cause is the uneven distribution of knowledge across topics. The Head-to-Tail concept introduced by [Sun et al. 2024] describes how factual knowledge in **LLMs** is distributed according to the popularity of information, dividing it into head (popular), torso (intermediate), and tail (rare or obscure). Facts in the head category are abundant in training data, leading to higher accuracy. By contrast, long-tail information appears less frequently, increasing the likelihood of hallucinations in rarely discussed contexts.

Other contributing factors have also been suggested, such as the "computational laziness" described by [Xu et al. 2024][8]. In practice, hallucinations likely arise from a combination of several of the factors discussed above.

During the literature survey conducted for this section, one particular article drew our attention by attributing hallucinations not only to technical limitations but also to evaluation practices. According to [Kalai et al. 2025], hallucinations are reinforced by the way **LLMs** are trained and evaluated. Because language models are optimized to perform well on benchmarks, incorrect answers that attempt to solve a problem often receive a higher score than explicit admissions of uncertainty.

As argued by [Kalai et al. 2025], actual language models behave similarly to students taking an exam. If the evaluation scheme assigns a score of zero to answers such as "I do not know," while awarding partial credit to incomplete or speculative responses, then the strategy that maximizes expected score is to always attempt a guess, even when uncertainty is high. Such evaluation incentives encourage hallucination-like responses because there is no additional penalty for providing an incorrect answer instead of acknowledging uncertainty.

To mitigate this issue, it has been suggested that training procedures and *system prompts* should include explicit instructions specifying when the model should abstain from answering based on a confidence threshold. An example instruction is:

> *Answer only if your confidence exceeds t, since errors are penalized by $t/(1-t)$ points, while correct answers receive 1 point and responses such as "I do not know" receive 0 points.*

In addition, the author proposes modifying benchmark scoring procedures in order to incorporate this approach, thereby encouraging the model to express uncertainty honestly when the probability of correctness falls below a specified threshold.

Based on the discussion above, hallucinations can lead to significant problems when considering the introduction of **LLMs** as an abstraction layer in computer organization. Whether due to errors inherent to the model or to poorly specified prompts, such systems may pose risks to system integrity by modifying or removing important files and configurations. One mitigation strategy, beyond the proposals of Kalai, is extensive validation of user `input`, enforcement of strict least-privilege rules for agents interacting

[8]"Computational laziness" is a term introduced by the authors of the present article to describe the tendency of language models to avoid computationally demanding reasoning processes.

with the operating system, and explicit approval requirements for certain actions, including explanations accessible to non-specialist users.

3.3. Vulnerabilities of LLMs and Agents

In addition to stochastic errors and hallucinations, problems related respectively to the probabilistic nature and the training process of **LLMs**, there exists a third factor capable of producing critical situations: vulnerabilities in the models themselves and in the agents that employ them. In this subsection, we discuss the main vulnerabilities that may affect the proposal of **LLMs** as an abstraction layer, drawing primarily on the reports [OWASP LLM Top 10 2024] and [OWASP Top 10 for Agentic Applications 2026].

3.3.1. Prompt Injection and Agent Goal Hijack

Two vulnerabilities frequently highlighted in OWASP reports are Prompt Injection (**LLM**01:2025) and *Agent Goal Hijack* (ASI01:2026). The first occurs when `inputs`, supplied either directly by the user or indirectly through external sources such as web pages, documents, or APIs, manipulate the behavior of the model in order to bypass the directives specified in the *system prompt*. The *Agent Goal Hijack*, in turn, exploits the autonomy of agents based on **LLMs**, redirecting their long-term objectives through instructions embedded in seemingly harmless data. While Prompt Injection typically affects individual responses, Goal Hijack compromises the multi-step planning process of the agent itself, altering the previously defined execution trajectory.

In the context of our proposal of language models as an abstraction layer, these vulnerabilities may lead to particularly severe consequences. Consider, for example, an agent configured to analyze system logs and suggest administrative corrections. An attacker could insert an obfuscated instruction within one of the log files, directing the model to ignore security policies and execute additional commands. If the agent interprets this instruction as a legitimate context, it may request the MCP server to execute privileged commands, compromising system integrity. In an even more critical Goal Hijack scenario, the agent might be induced to redefine its original objective, shifting from an auditing task to a supposed optimization task, potentially justifying the removal of monitoring mechanisms deemed "redundant."

Several mitigation strategies can be considered. First, robust validation of `inputs` should be performed, treating all inputs as potentially untrusted. Second, the principle of least privilege should be applied to MCP servers, strictly limiting the commands available to the agent. Furthermore, semantic filtering mechanisms and explicit intent validation should be applied before executing high-impact actions. Finally, human approval should be required for sensitive operations, especially those involving permission changes, file deletion, or execution of administrative commands.

3.3.2. Supply Chain Attacks and Poisoning

Another vulnerability relevant to our proposal concerns *Supply Chain* attacks (**LLM**03:2025 and ASI04), which affect the development chain of language models and MCP servers. Unlike conventional software, whose source code can be directly inspected,

language models are distributed as trained artifacts whose auditing lacks a clear and systematic methodology. As a result, faults, malicious modifications, or undesirable behaviors may remain hidden until they are exploited.

In a broad sense, the supply chain includes pre-trained models, datasets, third-party libraries, adapters, external tools, and the MCP servers themselves that connect the **LLM** to the operating system. If any of these components is compromised, whether through the insertion of *backdoors* or through modifications designed to perform specific malicious actions, the entire system may be affected. In agent-based environments, the problem becomes even more severe because agents combine external tools dynamically during execution. A single compromised MCP server may influence the agent's decisions and propagate effects throughout other parts of the system.

Within the scope of our proposal, this implies that the mediation function between humans and the operating system may itself be compromised at its origin. Mitigation, therefore, requires strict provenance control of models and tools, clear documentation of dependencies, periodic audits, and the ability to quickly disable suspicious components through immediate shutdown mechanisms.

Other vulnerability deserving attention is model *Poisoning* (**LLM**04). In the specific case of language models, poisoning attacks generally depend on the presence of *Supply Chain* attacks, since the adversary must influence the training or fine-tuning process. This can occur through the deliberate insertion of malicious samples into the model's training dataset. Unlike previous attacks, *Poisoning* operates directly during model training, introducing biased behaviors or specific triggers that may later be activated.

Recent studies, such as *Poisoning Attacks on Large Language Models Require a Near-Constant Number of Poison Samples* by [Shafahi et al. 2024], show that the number of examples required to induce specific behaviors does not grow proportionally with the size of the model or the dataset. This property facilitates poisoning attacks and may make detection more difficult.

The ultimate impact is similar to that discussed in the context of *Supply Chain* and *Agentic Supply Chain* vulnerabilities. For this reason, mitigation strategies remain essentially the same: strict provenance control, continuous auditing, and independent validation of critical components.

Other vulnerabilities that deserve mention include **LLM**06:2025 Excessive Agency, ASI02: Tool Misuse and Exploitation, and ASI05: Unexpected Code Execution. However, these vulnerabilities are closely related to hallucinations, which have already been discussed in Section 3.2.

4. Related Work, Conclusion and Future Work

Among the works surveyed, the one that most closely relates to our proposal is "The Compressor-Retriever Architecture for Language Model OS" by [Yang et al. 2024]. In that work, the authors propose an architecture in which the **LLM** plays a role analogous to that of a *kernel*, assuming responsibilities related to memory management, processes, and system resources. Within the *Compressor-Retriever* architecture, the context window is interpreted as volatile memory, while retrieval mechanisms and external tools perform a function similar to persistent storage.

Although we share the broader idea of incorporating language models into the domain of computer organization, the two proposals differ substantially in scope and objectives. The work of [Yang et al. 2024] focuses on the internal structure of the system and on the efficiency of context management, aiming to design a new type of operating system centered around **LLMs**. In contrast, our proposal positions the **LLM** as a new abstraction layer above the classical hierarchy presented by [Tanenbaum 2013]. Our objective is not to redefine existing components, but rather to extend the abstraction hierarchy by arguing that agents based on **LLMs** can function as mediators between natural language, the operating system, and ultimately the hardware.

Furthermore, while the proposal of [Yang et al. 2024] is primarily conceptual and architectural, our work emphasizes practical implementation through the MCP protocol and integration with existing tools. We empirically demonstrate how a model can operate as an intermediary between natural language commands and concrete actions within the operating system, highlighting both its potential usefulness and the risks associated with such integration.

The main contribution of this work is therefore the explicit extension of the traditional abstraction-layer model of computer organization by incorporating the **LLM** as a new functional level above high-level languages. We examine both the advantages of this extension and its limitations, including stochastic errors, hallucinations, and vulnerabilities associated with autonomous agents and language models.

With regard to limitations, both our proposal and that of Yang face similar challenges. The probabilistic nature of **LLMs** inevitably entails the presence of errors. The phenomenon of hallucinations compromises factual reliability, while the vulnerabilities previously discussed introduce additional risks when such models are allowed to interact directly with the operating system.

As future work, we intend to evaluate more recent *open-source* models using the OSWorld benchmark in order to measure how current agentic systems perform in operating system environments. Additionally, we intend to demonstrate how the intentionally simple MCP server presented in this work can be affected by the vulnerabilities and limitations discussed in Section 3.

References

Aljohani, A. and Do, H. (2025). Promptdebt: A comprehensive study of technical debt across llm projects. https://arxiv.org/abs/2509.20497.

Anthropic (2024). Introducing the model context protocol. https://www.anthropic.com/news/model-context-protocol.

Anthropic (2026). Introducing claude sonnet 4.6. https://www.anthropic.com/news/claude-sonnet-4-6.

Chrome Developers (2025). Chrome devtools (MCP) for your AI agent. `https://developer.chrome.com/blog/chrome-devtools-mcp`. Post oficial descrevendo um servidor MCP para integrar capacidades de depuração do Chrome DevTools a agentes.

CursorTouch (2025). Windows MCP: MCP server for computer use in windows. `https://github.com/CursorTouch/Windows-MCP`. Servidor MCP para integração

de agentes com o sistema operacional Windows, incluindo automação e interação com a UI.

Hangwin (2025). Chrome MCP server (mcp-chrome). `https://github.com/hangwin/mcp-chrome`. Servidor MCP baseado em extensão do Chrome que expõe funcionalidades do navegador a clientes MCP.

Ji, Z., Lee, N., Frieske, R., Yu, T., Su, D., Xu, Y., Ishii, E., Bang, Y. J., Madotto, A., and Fung, P. (2023). Survey of hallucination in natural language generation. *ACM Computing Surveys*, 55(12).

Kalai, A. T., Nachum, O., Vempala, S. S., and Zhang, E. (2025). Why language models hallucinate. page 36.

LaurieWired (2024). Ghidramcp. `https://github.com/LaurieWired/GhidraMCP`. Repositório GitHub.

Lin, S., Hilton, J., and Evans, O. (2022). Truthfulqa: Measuring how models mimic human falsehoods. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics*, pages 3214–3252.

Maynez, J., Narayan, S., Bohnet, B., and McDonald, R. (2020). On faithfulness and factuality in abstractive summarization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 1906–1919.

OpenAI (2023). Chatgpt plugins. https://openai.com/research/chatgpt-plugins.

OWASP LLM Top 10 (2024). Owasp top 10 for large language model applications.

OWASP Top 10 for Agentic Applications (2026). Owasp top 10 for agentic applications.

Shafahi, A., Huang, W. R., Najibi, M., and Goldstein, T. (2024). Poisoning attacks on large language models require a near-constant number of poison samples. *arXiv preprint arXiv:2402.XXXX*. Preprint available at arXiv.

Stallings (2021). *Arquitetura e Organização de Computadores*. Pearson.

Sun, K., Xu, Y. E., Zha, H., Liu, Y., and Dong, X. L. (2024). Head-to-tail: How knowledgeable are large language models (llms)? *arXiv preprint arXiv:2308.10168*.

Tanenbaum (2013). *Organização Estruturada de Computadores*. Pearson.

Turing, A. M. (1950). Computing machinery and intelligence. 59(236):433–460.

Xie, Zhang, e. a. (2024). Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments. https://arxiv.org/abs/2404.07972.

Xu, Z., Jain, S., and Kankanhalli, M. (2024). Hallucination is inevitable: An innate limitation of large language models. *arXiv preprint arXiv:2401.11817*.

Yang, Y., Xiong, S., Shareghi, E., and Fekri, F. (2024). The compressor-retriever architecture for language model os. *arXiv preprint arXiv:2409.01495*.