SLODON: An AI Agent Revolutionizing User Space Program Interactions with Large Language Models

Tamás Hadházy

Abstract

The Slodon project is still a work in progress, kind of like its early baby steps, but I think it's really important to point out the idea and spread more information about it. This CLI application aims to control user space programs and interactions using extensive language models and automated processes, drawing inspiration from Auto GPT. The project's primary stack could potentially be Python, capable of running within the terminal. CLI applications can be packaged as Linux AppImages, making them a flexible choice in terms of deployment and compatibility. Users can simply chat with Slodon through the terminal and easily close the whole thing whenever they want. One of the main ideas behind this application is that it handles and interacts with the programs on the user computer. The automation part can be influenced from pyautogui library, which itself uses the xlib. The entire project revolves around a language model that empowers the entire system. Slodon, in particular, may require a custom model trained on an extensive corpus of Linux source code and applications. Once all the initial components are in place, Slodon may require a substantial amount of configuration data specific to the user's computer, as well as various real-time data sources.

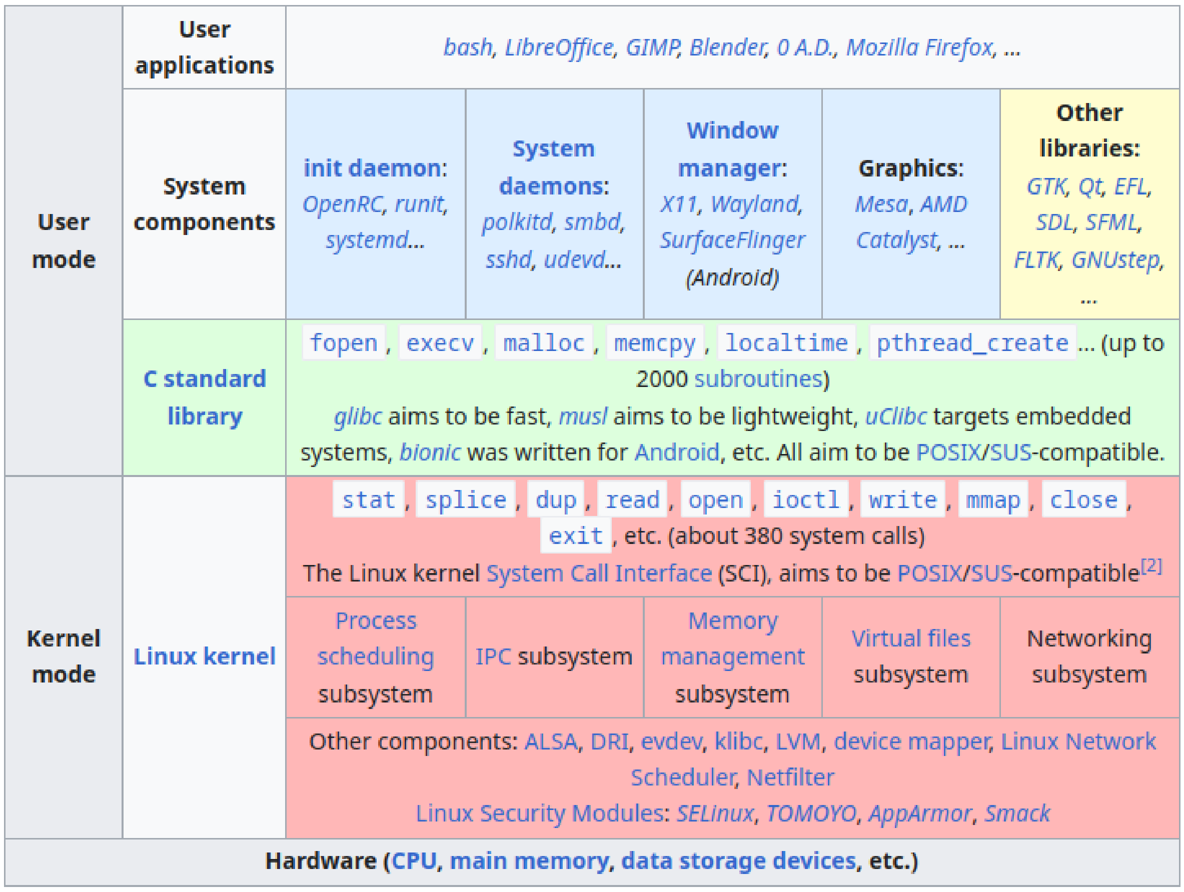
1 Method

Slodon is made up of four main components: **model**, **API**, **CLI**, and the **Automate library.** These pieces work together to create an awesome experience.

1.1 User Applications

The user programs or user appliciations are basically the programs in the user_land or user mode that the operating system uses to talk to the kernel. Think of these applications as the ones you use every day, or to put it simply, they're your way of talking to the kernel. Why does Slodon need to know about these programs and how does it help? Well, these programs are the key software components that Slodon needs to handle and control to carry out its designated functions. Slodon itself is counted among those programs. it **doesn't require** direct communication with the **kernel or the "ring-0" level**.

Various layers within Linux, also showing separation between the userland and kernel space



User mode	**User applications**	*bash*, *LibreOffice*, *GIMP*, *Blender*, *0 A.D.*, *Mozilla Firefox*, ...				
	System components	**init daemon:** *OpenRC*, *runit*, *systemd*...	**System daemons:** *polkitd*, *smbd*, *sshd*, *udevd*...	**Window manager:** *X11*, *Wayland*, *SurfaceFlinger* *(Android)*	**Graphics:** *Mesa*, *AMD Catalyst*, ...	**Other libraries:** *GTK*, *Qt*, *EFL*, *SDL*, *SFML*, *FLTK*, *GNUstep*, ...
	C standard library	`fopen`, `execv`, `malloc`, `memcpy`, `localtime`, `pthread_create`... (up to 2000 subroutines) *glibc* aims to be fast, *musl* aims to be lightweight, *uClibc* targets embedded systems, *bionic* was written for Android, etc. All aim to be POSIX/SUS-compatible.				
Kernel mode	**Linux kernel**	`stat`, `splice`, `dup`, `read`, `open`, `ioctl`, `write`, `mmap`, `close`, `exit`, etc. (about 380 system calls) The Linux kernel System Call Interface (SCI), aims to be POSIX/SUS-compatible[2]				
		Process scheduling subsystem	IPC subsystem	Memory management subsystem	Virtual files subsystem	Networking subsystem
		Other components: ALSA, DRI, evdev, klibc, LVM, device mapper, Linux Network Scheduler, Netfilter Linux Security Modules: *SELinux*, *TOMOYO*, *AppArmor*, *Smack*				
Hardware (CPU, main memory, data storage devices, etc.)						

1.2 Automate process

This part is actually one of the key highlights in the project. The library **handles all the interactions,** and it's something we can continuously enhance over time. It's a highly tangible feature that users can directly experience and benefit from. This can be a **sub-library within the Slodon repository,** written in Python. It plays a vital role in the overall functionality, **requiring abundant user prompts** as well as prompts **from the language model** to generate the desired outputs. As I mentioned earlier, this agent is something you can actually watch in action as it performs the interactions. **It's not some hidden magic happening behind the scenes**. When you give it a task, this library **takes charge and**

carries out the necessary interactions, but it definitely needs a lot of improvement over time to make it even better. One of the main benefits is that you can actually **keep an eye on the agent** and make adjustments if needed. It's like **watching the whole process unfold in front of you**, instead of just receiving a file and having to start over from scratch if something goes wrong. It's much mor**e interactive and gives you more control over the outcome**. I'm really exited about designing our own sub-library for this.

1.3 API

The API has two main functions in this context. Firstly, it is allowing for **account synchronization**. Secondly, it functions as a **data provider for the automated processes,** supplying data from the model. We can use t**he synchronization process** to keep track of the user and handle **various types of metadata**. Another function is the model feature, where the language model sends back the response when we send the prompt. This interaction **occurs through the API connection.**

1.4 Language Model

1.4.1

So, we've got two main things to think about. First, we need to find a language model that's available to everyone and knows a lot of stuff already. That's what we'll use as the basis for generating responses and providing information. Second, we should consider what else we can add to make the model even better at predicting and giving us the results we want. When it comes to choosing a language model, we want something that's **open source**, **well-tested, and highly respected**. It's important to find a model that's widely used and trusted by the community.

As a starting point, **the LLaMA model** seems to be a suitable choice for our needs. It can serve as the foundation for our project, offering a reliable and versatile framework to build upon.

1.4.2

The **Llama** model is a great starting point for our project. But to make it even better, we **need to add some fresh information about Linux source codes and application details**. This will help the model give more accurate and detailed responses when it comes to questions and discussions related to Linux and its applications.

I get that there are some constraints, but hey, let me give you an example to help you understand better.

[Example:]

"

-The user opens the CLI and set up the basic informations(prompts) that slodon has to know first.
- Then the user create the tasks that we should execute. Like: -open the libre office application, write a 100 pages essay, then save the file in the documents folder.

"

As you can see, the whole "writing a 100-page essay" thing could be handled by the **basic LLaMA model itself**. But here's the deal: when it comes to opening applications and doing different things inside them, we **need to give the model some new info**. That way, it knows all about different applications and their features, like where to find buttons and how they work, by incorporating this complete information, our automate library **gains the ability to perform tasks with a deeper understanding** and **execute them effectivel**y. The idea is to **beef up a well-known model** with **some new information** about Linux and its applications. This way, the model can become even better at dealing with **Linux-related queries and tasks.**

Our model component **is all about handling this new information** and e**xtending the well-known base model.**

<u>**So, to put it simply, we're going to fine-tune the LLaMA model using our brand new custom dataset.**</u>

1.5 CLI

The CLI part is pretty straightforward. It'll kick off the process and wrap it up nicely using a Python library like **textual**.

2 How to contribute to the paper?

Well, you can always add something new and then don't forget to add your name.