

SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE

Fakulta elektrotechniky a informatiky

Evidenčné číslo: FEI-xxxx-xxxx

**ShAI — Hodnotenie efektívnosti využitia
lokálnych LLM modelov ako asistentov v
príkazovom riadku**

Bakalárska práca

SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE

Fakulta elektrotechniky a informatiky

Evidenčné číslo: FEI-xxxx-xxxx

**ShAI — Hodnotenie efektívnosti využitia
lokálnych LLM modelov ako asistentov v
príkazovom riadku**

Bakalárska práca

Študijný program:	Aplikovaná informatika
Študijný odbor:	Informatika
Školiace pracovisko:	Názov školiaceho pracoviska
Školiteľ:	Ing. Roderik Ploszek, PhD.



ZADANIE BAKALÁRSKEJ PRÁCE

Študent: **Meno Priezvisko**
ID študenta: 012345
Študijný program: jadrové a fyzikálne inžinierstvo
Študijný odbor: elektrotechnika
Vedúci práce: tituly, Meno Priezvisko, tituly
Vedúci pracoviska: tituly Meno Priezvisko, tituly
Miesto vypracovania: Ústav jadrového a fyzikálneho inžinierstva

Názov práce: **Rozšírená šablóna záverečnej práce na FEI STU v Bratislave
v systéme LaTeX**

Jazyk, v ktorom sa práca vypracuje: slovenský jazyk

Špecifikácia zadania:

- Vytvorte šablónu záverečnej práce pre študentov bakalárskeho a inžinierskeho štúdia na FEI STU v Bratislave.
- Napište návod na písanie záverečnej práce s použitím vytvorenej šablóny.
- Vytvorte prehľad najpoužívanejších typov citovaných zdrojov a zdokumentujte ich tak, aby boli v súlade s normou ISO 690.

Zoznam odbornej literatúry:

- Zákon č. 131/2002 Z. z. z 21. februára 2002 o vysokých školách a o zmene a doplnení niektorých zákonov [online]. [cit. 2024-09-01]. Dostupné na https://www.slov-lex.sk/static/pdf/2002/131/ZZ_2002_131_20240901.pdf
- Vyhláška č. 233/2011 Z. z. Ministerstva školstva, vedy, výskumu a športu Slovenskej republiky z 1. júla 2011, ktorou sa vykonávajú niektoré ustanovenia zákona č. 131/2002 Z. z. o vysokých školách a o zmene a doplnení niektorých zákonov v znení neskorších predpisov [online]. [cit. 2024-09-01]. Dostupné na https://www.slov-lex.sk/static/pdf/2011/233/ZZ_2011_233_20201015.pdf

3. Metodické usmernenie Ministerstva školstva, vedy, výskumu a športu SR č. 56/2011 o náležitostiach záverečných prác, ich bibliografickej registrácii, kontrole originality, uchovávaní a sprístupňovaní č. 2011-11513/31015:4-071 [online]. [cit. 2024-09-24]. Dostupné na <http://www.minedu.sk/metodicke-usmernenie-c-562011-o-nalezitostiach-zaverecnych-prac-ich-bibliografickej-registracii-uchovavani-a-spristupnovani/>.
4. STN ISO 214: 1998. Dokumentácia. Abstrakty (referáty) pre publikácie a dokumentáciu. Bratislava: Slovenský ústav technickej normalizácie. Bratislava. Úrad pre metrológiu a skúšobníctvo Slovenskej republiky.
5. STN ISO 690: 2022. Dokumentácia – Bibliografické odkazy – Obsah, forma a štruktúra. Bratislava. Úrad pre metrológiu a skúšobníctvo Slovenskej republiky.
6. STN 01 6910: 2023. Pravidlá písania a úpravy písomností. Bratislava. Úrad pre metrológiu a skúšobníctvo Slovenskej republiky.
7. STN ISO 2145: 1997. Dokumentácia. Číslovanie oddielov a pododdielov písaných dokumentov. Bratislava. Úrad pre metrológiu a skúšobníctvo Slovenskej republiky.
8. STN ISO EN 80 000-1: 2022. Veličiny a jednotky. Bratislava. Úrad pre metrológiu a skúšobníctvo Slovenskej republiky.
9. LICHNEROVÁ, L. a B. BELLÉROVÁ. Nové pravidlá citovania podľa ISO 690 z roku 2021. *ITlib. Informačné technológie a knižnice*, 3–4/2023, 10 – 22. [cit. 2024-09-06]. Dostupné na <http://doi.org/10.52036/1335793X.2023.3-4.10-22>.
10. HOFTICH, M. ISO 690 biblatex style [online]. [cit. 2024-09-30]. Dostupné na <https://mirrors.ibiblio.org/CTAN/macros/latex/contrib/biblatex-contrib/biblatex-iso690/biblatex-iso690.pdf>.

Termín odovzdania bakalárskej práce: 31. 05. 2025

Dátum schválenia zadania bakalárskej práce: 28. 02. 2025

Zadanie bakalárskej práce schválil: tituly Meno Priezvisko, tituly – garant študijného programu

Podakovanie

Na tomto mieste by som sa rád poďakoval ... za

Abstrakt

Tento dokument popisuje proces analýzy, návrhu, vývoja a implementácie Linux CLI asistenta s využitím lokálnych LLM modelov, aj hodnotenie jeho efektívnosti v rozličných scenároch použitia pre rozné modely a architektúry LLM.

Kľúčové slová

záverečná práca, LLM, lokálne veľké jazykové modely, asistenti príkazového riadku, integrácia do shellu, hodnotenie

Abstract

This document describes the process of analyzing, designing, developing, and implementing a Linux CLI assistant using local LLM models, as well as evaluating its effectiveness in various usage scenarios for different LLM models and architectures.

Keywords

Final thesis, LLM, local large language models, command line assistants, shell integration, evaluation

Obsah

Úvod	13
1 Theoretical Background	15
1.1 Terminal Interfaces and Applications. CLI, TUI	15
1.1.1 Terminal, Terminal emulator. Terminal Application	15
1.1.2 CLI. Shells and CLI Utilities	16
1.1.3 TUI. TUI Applications	17
1.2 Large Language Models (LLMs)	19
1.2.1 Architecture Overview. Functionality	19
1.2.2 Transformer Architecture. Encoder and Decoder	19
1.2.3 Mixture of Experts (MoE) Extension	20
1.2.4 AI Agent Systems	21
1.2.5 Retrieval-Augmented Generation (RAG)	21
1.2.6 Communication protocols. OpenAI API. MCP	22
2 Review and Analysis of Existing Solutions	23
2.1 General Solutions Overview	23
2.1.1 Warp Terminal	23
2.1.2 gptme	23
2.1.3 gemini-cli	24
2.1.4 Yai (Your AI Assistant)	24
2.1.5 Aider	25
2.1.6 ai-terminal-assistent (github)	25
2.1.7 cmd-ai (github)	25
2.1.8 termai	25
2.1.9 AiTerm	25
2.2 Detailed Analysis of Selected Solutions	26
2.2.1 gptme. AI architecture	26
2.3 Other Related Applications	26
2.3.1 Zsh. Fish, Bash	26
2.3.2 HSTR	26
2.3.3 GNU Readline	26
3 Design and Implementation	27
3.1 Design Considerations	27
3.2 System Architecture	27

3.3	Development Process	27
3.3.1	Component 1. Natural Language vs. Shell Command Classifier .	27
3.3.2	Component 2. Information retriever	28
3.3.3	Component 3. Task splitter	28
4	Evaluation	29
	Záver	30
	Literatúra	31
	Použitie nástrojov umelej inteligencie	32

Zoznam obrázkov a tabuliek

Tabuľka 1	Examples of shells and CLI utilities	17
Tabuľka 2	Examples of shells and CLI utilities	18
Tabuľka 3	Second half of comparison table for Local LLM Assistant Development	26

Zoznam algoritmov

Zoznam výpisov kódov

Úvod

Fast development of technology in recent years, especially in computing power and data storage, has enabled the machine learning field to evolve rapidly. This led us to the development of versatile ML models, which are used in wide range of applications. LLMs are one of the most popular types of ML models nowadays, which has shown great advancements and now has a lot of practical applications.

The nature of the interaction with LLM's makes us consider them as helpful assistants in various tasks. The most appreciated feature of LLM's is their ability to transfer from natural language instructions to executable commands.

One of common fields, where such feature can be utilized and increase the QoL of users (from developers to regular OS users, as now Linux distributions are becoming more user-friendly and popular), is the command line interface. It is to remain an essential part of modern development (and related stuff) processes as it enables us to perform tasks quickly and efficiently.

This is what motivates us to create and evaluate a command line assistant, which leverages LLM to break the barrier between NL instructions and computer commands, yielding the need to learn specific syntax for each command from the user.

The limitations of remote LLMs often present significant challenges in real-world scenarios. Relying on a network connection introduces latency, potential downtime, and dependency on external services. Furthermore, data privacy and security concerns frequently restrict the use of external LLMs, especially when dealing with sensitive information or proprietary tools. Consider the scenario of a developer needing to quickly execute a complex series of commands for debugging, or a system administrator requiring immediate assistance in resolving a system issue – a network outage could be devastating.

Therefore, this thesis aims to evaluate local LLMs as a viable solution for command-line assistance. By running these models directly on the user's machine, we eliminate the dependency on external services, reduce latency, and address concerns regarding data privacy. This approach allows for a focused investigation into the performance and usability of LLMs in a low-latency, secure environment.

The main goal of this thesis is to describe the process of design, development, and to evaluate a Linux CLI assistant powered by different local LLM models (differing in architecture, size, and capabilities and under different constraints) to see how well they perform in various usage scenarios.

1 Theoretical Background

This section provides the necessary theoretical background for understanding the design, implementation of the tools we will be using. Its main goal is to provide the reader with sufficient information to understand the rest of the thesis, as well as to provide a right guidelines for usage of the tools in the implementation phase.

1.1 Terminal Interfaces and Applications. CLI, TUI

This section describes the interfaces and programs that are relevant to our solution. It defines two interfaces: Command Line Interfaces (CLIs) and Text-based User Interfaces (TUIs). It also describes the programs that leverage these interfaces: shells, CLI utilities, and TUI applications.

For the beginning, we need to understand what Command Line Interfaces (CLIs) are, as well as the utilities that are used within them.

The definition of the interface is adopted from Branden Hookway's book *Interface*, where he defines it as follows:

„In computing, an interface is a shared boundary across which two or more separate components of a computer system exchange information. The exchange can be between software, computer hardware, peripheral devices, humans, and combinations of these.“

[1]

1.1.1 Terminal, Terminal emulator. Terminal Application

The article focuses on text-based **terminal** interfaces, which are a specific type of interface used in computing environments. Here is how we define terminals, terminal emulators and terminal applications.

Terminal is a „an electronic or electromechanical hardware device that can be used for entering data into, and transcribing data from, a computer or a computing system“. This definition defines main functionality of terminals and terminal emulators, which is to provide an elementary, which is CLI, interface for user to interact with the computer system.

Terminal emulator / Terminal application is a software application that replicates the functionality of a traditional hardware terminal as a software program. Here are some examples of terminal emulators:

- iTerm2 (macOS)
- Windows Terminal (Windows)
- Alacritty (Cross-platform)
- tty (Linux)

Terminal applications are software programs that run within terminal emulators and utilize their capabilities to provide text-based user interfaces — either CLIs or TUIs. We will discuss it further in the following sections.

In most cases, always, when the application is POSIX-compatible, we can define text streams, used by terminal application for communication. [2] [3]

1. STDIN — used for input from user; initially as file descriptor 0.
2. STDOUT — used for application output; initially as file descriptor 1.
3. STDERR — used for error output; initially as file descriptor 2.

[4]

POSIX on default communication streams for application: (hidden)

...

1.1.2 CLI. Shells and CLI Utilities

The definition of CLI extends previous definition to the specific case of command line interfaces, which are text-based interfaces between the user and the computer system, where user inputs commands in the form of text, to manipulate the system or perform specific tasks, and gets responses in the same form.

Shells are special programs that provide relatively wide and direct access on system on which it runs. Most shells are CLI programs. They interpret user commands, execute them, and return the results.

CLI utilities are programs that are executed from within the CLI environments.

Here are some popular examples of shells and CLI utilities:

Name	System	Description
Bash, Zsh	Linux, macOS	Popular Unix-like shells, compliant with POSIX standard. ¹
Cisco IOS Shell	Cisco devices	Shell used in Cisco network devices.
bluetoothctl	Linux, macOS	CLI utility for managing Bluetooth devices. ²
git	Cross-platform	CLI utility for version control using Git system.
nmcli (Network-Manager)	Linux	Application for controlling network settings.

Tabulka 1: Examples of shells and CLI utilities

1.1.3 TUI. TUI Applications

Text-based User Interfaces (TUIs) are a type of user interface that is a further extension of CLIs. They provide a more visually structured and intuitive way for users to interact with applications by using libraries such as ncurses, which use the **terminal capabilities** emulators to create rich text-based interfaces.

Terminal capabilities is a special functionality of the terminal, which gives the application more control over its output (i.e. move cursor in all directions, change the color of the output, rewrite/clean characters and so on).

Here are some examples of applications using TUI:

Name	System	Description
vi/vim/nvim	Linux, macOS, Windows	Best ever TUI visual text editor.
mc (Midnight commander)	Linux, macOS, Windows	TUI file explorer.
htop	Linux	Task manager. nmtui (NetworkManager)
Linux	Application for controlling network settings.	

Tabulka 2: Examples of shells and CLI utilities

1.2 Large Language Models (LLMs)

This section provides a brief overview of the aspects of Large Language Models (LLMs), relevant for this article. It describes the architecture, types, limitations of LLM, the main protocols for the interaction, as well as the advanced concepts, like extension and fine-tuning.

1.2.1 Architecture Overview. Functionality

Large language models (LLMs) refer to Transformer language models that contain hundreds of billions (or more) of parameters, which are trained on massive text data, such as GPT-3, PaLM, Galactica, and LLaMA. LLMs exhibit strong capacities to understand natural language and solve complex tasks (via text generation).

Currently, LLMs are mainly built upon the Transformer architecture [5], where multi-head attention layers are stacked in a very deep neural network. They are based on the Transformer architecture.

The model tokenizes the input text into smaller units, to be able to represent and process it effectively. The set of possible tokens is called the vocabulary, and can include words, subwords, or even individual characters. It is important to note that the vocabulary is fixed after training, which means that any new or rare words or combinations of characters (for example, ciphertexts) may not be processed accurately or at all.

The transformer architecture is based on finding relationships between elements in a sequence using self-attention mechanisms [6]. This enables the model to recognize patterns and dependencies in the input, which may be helpful for decrypting ciphers that rely on patterns in the text.

1.2.2 Transformer Architecture. Encoder and Decoder

In original article, previously mentioned Transformers consist of two main parts — **Encoder** and **Decoder**. For the purpose of this article, it is enough to know that nowadays LLMs are mainly based on Decoder-only architecture (also called Causal Decoder), which means they only use the Decoder part of the original Transformer architecture. And most of the time, they don't use the Encoder part at all.

The usage of Decoder-only architecture implies that the model generates the output text sequentially, one token at a time, while attending to all previously generated tokens. This can be a limitation for long prompts, as the model can only attend to a limited number of previous tokens (this limit is called context length or window size).

Furthermore, the longer the prompt, the more computationally expensive it becomes to generate the output, as the model has to process all previous tokens for each new token.

Additionally, Decoder-only models are typically easier to train and fine-tune, as they require less complex training data and objectives compared to Encoder-Decoder models.

1.2.3 Mixture of Experts (MoE) Extension

While the architectures above define how information flows through the layers, the **Mixture of Experts (MoE)** is an important extension that changes *how* the feedforward layers operate, with the goal to increase scalability and efficiency.

In a traditional dense Transformer, which can be referred to as **Dense**, the whole decoder is a monolith, meaning that while processing, every neuron is used. The Mixture of Experts (MoE) architecture splits the massive, singular feedforward network into multiple smaller, quasi-specialized (as everything is learned) *expert* subnetworks and introduces a *router* (or gating network) to orchestrate them. The router acts as a traffic controller, dynamically selecting only a tiny subset of experts—typically one or two—to process it.

When it comes to comparing MoE and dense models in scope of smaller devices with certain limitations, MoE architectures introduce the changes in two main aspects compared to dense models: computational efficiency and memory usage.

- **Computational Efficiency:** MoE models are more computationally lighter as the selective activation leads to significant reductions in computation time and energy consumption. However, such step makes the model less intelligent, especially on smaller scales. In contrary, a dense model involves every parameter in the computation for each token, leading to higher computational costs, but also more consistent performance.
- **Memory Usage:** The major drawback of MoE models is their high memory requirements. Since all experts must be loaded into memory to allow the router to select them dynamically, the total memory footprint is much larger than that of a dense model. Additionally, such architecture allows for knowledge duplication, thus less efficient memory usage.

Overall, MoE models tend to be more efficient in terms of computation, but have worse performance over dense models with same number of parameters. However, if VRAM is not a constraint, MoE models can achieve better performance than dense models within the same computational budget.

1.2.4 AI Agent Systems

The evolution of LLMs has progressed beyond simple text generation to the concept of **AI Agents**, which utilize LLMs with the capacity to plan, act, and interact with external environments. AI agents are autonomous entities, capable of planning, decision-making, and performing actions to achieve complex goals. [6] [7]

Two major changes that AI Agents bring compared to traditional LLMs are:

- **Tool Use:** AI Agents can leverage external tools, RAG pipelines or APIs to retrieve information or perform actions beyond text generation.
- **Iterative Generation:** AI Agents can generate multiple steps of reasoning or actions iteratively, allowing them to break down complex tasks into manageable sub-tasks.

The second point is especially important for the purpose of this research, as it allows the agent to decompose complex tasks into steps small enough to be handled by local LLMs with limited capabilities.

We can define the workflow of AI Agents as follows [8]:

```
function AI_Agent(task, context):
    subtasks = LLM.decompose_task(task) # Break down the main task into smaller sub-tasks
    for subtask in subtasks:
        switch subtask.type:
            case "tool":
                result = use_tool(subtask.tool, subtask.tool_input)
            case "further_decomposition":
                result = AI_Agent(subtask.task, context) # Recursive call for further decomposition
            case "generate":
                result = LLM.generate_response(subtask.task)
        results.append(result)
    final_output = LLM.summarize_results(results)
    return final_output
```

In essence, AI Agents perform tasks, dividing complex objectives into manageable steps, and iteratively prompting themselves to execute these steps using various tools.

1.2.5 Retrieval-Augmented Generation (RAG)

Retrieval-Augmented Generation (RAG) is a technique that allows LLMs to utilize external knowledge bases, significantly reducing hallucinations and increasing accuracy. It uses vector databases to store data in a way that allows for efficient search and retrieval of relevant information.

When a user submits a query, the system retrieves relevant documents from the vector database based on semantic similarity to the query. These documents are then

added to the prompt sent to the LLM, providing it with additional context to generate a more accurate response.

1.2.6 Communication protocols. OpenAI API. MCP

Agents require robust communication protocols to allow agents to interact with each other and with external services. This is usually achieved through APIs or Model Context Protocol (MCP), which defines how to request these services.

The OpenAI protocol refers to the set of APIs and communication standards developed by OpenAI for interacting with their AI models. It typically uses RESTful HTTP endpoints with JSON payloads for requests and responses. Authentication is handled via API keys, and communication is secured over HTTPS. The protocol supports various endpoints for tasks such as completions, chat, embeddings, and file uploads, enabling integration with a wide range of applications.

MCP is a protocol designed for structured message exchange between clients and servers, often used in MUDs (Multi-User Dungeons) and similar networked applications. It defines a standardized way to encapsulate commands and data within messages, supporting extensibility through modules. MCP messages are typically text-based, with clear delimiters and key-value pairs, allowing for flexible and efficient communication of commands, events, and data between connected systems.

2 Review and Analysis of Existing Solutions

This section provides a comprehensive review of existing solutions, tools and applications. The information from this section will be used in further design and implementation of our solution.

This section provides a comprehensive review of existing solutions, tools and applications. The information from this section will be used in further design and implementation of our solution.

2.1 General Solutions Overview

This section contains an overview of already existing tools. This will help us to concentrate on essential points and may serve as good consideration points when we will be working on the design of our solution. There will also be provided a comparison table at the end of this section, which will summarize the main properties of each solution.

2.1.1 Warp Terminal

This is a development-oriented and general-purpose AI-enabled terminal emulator, which uses remote Agent LLM System to enhance the experience of users. The most of its code remains closed-source. It is not adapted for executing on small LLM's and it provides no support for local LLM's. Additionally, it is not a CLI/TUI application, but a full graphical terminal emulator.

This solution is popular and commercially successful, but it does not completely fit into the scope of our work, as it is not designed to work with local LLM's. However, it is worth mentioning, as it provides good AI-based services for terminal users [9] [10].

2.1.2 gptme

This is general-purpose CLI utility with breadth AI functionality. It is compatible with OpenAI API, which allows it to use both remote and local LLMs. It uses Agentic architecture and implements MCP protocol. This enables it to use various tools (terminal, browser, code execution, etc). It is also context-aware, which means it can read and understand the files in the current working directory and acknowledge host OS [11] [12].

Below the short interaction example with gptme is shown:

```
\$ gptme --tools=browser "How many calories are there in one spoon of sugar?"
[11:07:21] WARNING Skipping all confirmation prompts. cli.py:186
[11:07:21] Using browser tool with lynx
Found openai API key, using openai provider
```



```

Using model: openai/gpt-5
[11:07:22] Using logdir: ~/.local/share/gptme/logs/2025-12-10-running-sleepy-robot
Using workspace: ~/Workspace/bachelor/project/document
Skipped 2 hidden system messages, show with --show-hidden
--- ^^^ past messages ^^^ ---
User: How many calories are there in one spoon of sugar?
Assistant: Short answer: about 16 calories per level teaspoon.

Details:
- Level teaspoon ( 4 g sugar): ~15 16 kcal
- Heaped/rounded teaspoon ( 57 g): ~20 28 kcal
- Tablespoon ( 1215 g): ~48 60 kcal

Sugar has ~4 kcal per gram (sucrose ~3.87 kcal/g), so you can estimate by weight if needed.

```

- Uses OpenAI API for accessing remote/local LLMs
- Uses Agent architecture and MCP tools
- Is context-aware and is able to use tools (terminal, browser, code execution, etc)
- Implements MCP protocol

When tested manually with local LLMs (Gemma3 4b, Qwen2.5 1.5b), it did not perform well, as the models were not able to handle the complexity of the tasks. It usually due to too long prompts which made LLMs hallucinate on too many details. This shows, that the solution is not fully adapted for local LLMs, which is one of the main goals of our work. This should be considered when designing our solution to use more fine-grained prompts for local LLMs and optimize the architecture for local LLMs.

2.1.3 gemini-cli

This is general-purpose CLI chat-like application by Google. It uses Gemini LLMs to provide AI capabilities in terminal. It is useful for various tasks, including code generation, data analysis, and general Q&A. It also supports MCP protocol and custom tool integration. Though, it is dependable on remote LLM and cannot be used with local LLMs [13] [14].

2.1.4 Yai (Your AI Assistant)

This application is an interactive TUI chat application, which uses OpenAI to provide user with AI capabilities in terminal. It supports tools interaction and basic agentic architecture. In current form, it is not able to use local LLMs. Though, it is possible to

adapt it for local LLMs with some modifications in its source code. [15] [16].

2.1.5 Aider

This is an AI-powered code agent with focus on software development tasks and terminal integration. It supports various tools and is context- and project-aware. It is able to use both remote and local LLMs. It is also open-source. [17] [18].

2.1.6 ai-terminal-assistent (github)

The definition

(temporary) References:

2.1.7 cmd-ai (github)

The definition

(temporary) References:

2.1.8 termai

The definition

(temporary) References:

2.1.9 AiTerm

The application that utilizes [language=bash]tmux

(temporary) References:

- <https://github.com/aki-colt/AiTerm>

(temporary) References:

- <https://github.com/aki-colt/AiTerm>

3 Design and Implementation

3.1 Design Considerations

3.2 System Architecture

Based on previously reviewed solutions, as well as small local LLM limitations, the best approach to system architecture is to divide the whole system into smaller components, each one responsible for one simple task. These defined small tasks will (1) reduce the possibility of LLM hallucination and (2) simplify the testing by allowing us to test and evaluate each component in separate. The modular design also makes future maintenance and expansion more manageable.

1. **Natural Language vs. Shell Command Classifier** — will allow the system to distinguish between user commands and questions, thus simplifying the interaction for the user. This is critical for seamless operation; the system should intelligently decide whether to execute a command directly or to engage the LLM for a more conversational response.
2. **Task splitter** — this component will split complex tasks into smaller ones, thus dividing the work for other nodes. This is inspired by multi-agent architecture and will follow similar principles. It will be responsible for taking a user's request (after classification) and breaking it down into sequential or parallel sub-tasks that can be handled by the Information Retriever or other subsequent components.
3. **Information Retriever** — for specific tasks related to information retrieval, the RAG-enabled component will be designed. It should be able to retrieve information from manuals or the web to provide accurate answers. This component will leverage a knowledge base (local documents, indexed websites, etc.) and utilize retrieval techniques to provide contextually relevant information to address user inquiries.

The interface will remain as close as possible to CLI with certain TUI elements where needed. We will prioritize a minimal, predictable user experience.

3.3 Development Process

3.3.1 Component 1. Natural Language vs. Shell Command Classifier

Idea. The first component is a simple proof-of-concept implementation of the proposed idea. It uses a local LLM hosted by Ollama to distinguish between shell commands and

natural language input. If the input is a shell command, it would execute it directly; if it's natural language, it would respond with an answer or suggestion.

While using such architecture is feasible, the problem that arises is how should the wrapper interact with the shell. It cannot attach to an existing bash process, as it is not possible to determine when the shell is waiting for user input (e.g. after running a TUI program) and when it is executing commands (this will be visible when using TUI applications inside the wrapper).

Another approach is to create new shell process for each command, but this will lead to loss of context (e.g. current working directory, environment variables, etc.) and will make the wrapper less useful.

Testing.

3.3.2 Component 2. Information retriever

Idea. This component is responsible for retrieving information from various sources to answer user queries. It's built around a Retrieval-Augmented Generation (RAG) architecture, meaning it combines information retrieval from a local knowledge base with LLM generation to produce informative and accurate responses. The task of this module is to summarize the relevant context and pass it to user or requesting node, which then generates a response based on the retrieved information and its own knowledge.

Testing.

3.3.3 Component 3. Task splitter

Idea. This component serves as the central coordinator for more complex requests. It receives the user input and decomposes it into a series of manageable sub-tasks. It will use the strategy discussed in 1.2.4.

Testing.

4 Evaluation

Záver

One finds only when one seeks.

Poznámka autora: Autorské dielo nesmie obsahovať odseky vytvorené generatívnou umelou inteligenciou.

Literatúra

1. HOOKWAY, B. The Subject of the Interface. In: *Interface*. MIT Press, 2014, s. 1–58. ISBN 9780262525503.
2. *Computer Terminal*. [B.r.]. Dostupné tiež z: https://en.wikipedia.org/wiki/Computer_terminal. Accessed: 2026-01-31.
3. *Terminal Emulator*. [B.r.]. Dostupné tiež z: https://en.wikipedia.org/wiki/Terminal_emulator. Accessed: 2026-01-31.
4. *System Interfaces, 2.5 Standard I/O Stream*. [B.r.]. Dostupné tiež z: <https://pubs.opengroup.org/onlinepubs/9799919799/>. Accessed: 2026-01-31.
5. POLOSUKHIN, A. V. N. S. N. P. J. U. L. J. A. N. G. L. K. I. Attention Is All You Need. In: Curran Associates, Inc., 2017, s. 6000–6010. Dostupné tiež z: <https://arxiv.org/abs/1706.03762>. Accessed: 2026-01-31.
6. NAVEED, H. et al. A Comprehensive Overview of Large Language Models. *arXiv preprint*. 2023, **2307.06435**. Dostupné tiež z: <https://arxiv.org/abs/2307.06435>. Accessed: 2026-01-31.
7. ZHAO, W. X. et al. A Survey of Large Language Models. *arXiv preprint*. 2023, **2303.18223**. Dostupné tiež z: <https://arxiv.org/abs/2303.18223>. Accessed: 2026-01-31.
8. BLOG, N. D. *Building Your First LLM Agent Application*. 2024. Dostupné tiež z: <https://developer.nvidia.com/blog/building-your-first-llm-agent-application/>. Accessed: 2025-11-31.
9. *Warp Website*. [B.r.]. Dostupné tiež z: <https://www.warp.dev/>. Accessed: 2025-11-31.
10. *Warp Documentation*. [B.r.]. Dostupné tiež z: <https://docs.warp.dev/>. Accessed: 2025-11-31.
11. *gptme Website*. [B.r.]. Dostupné tiež z: <https://gptme.org/>. Accessed: 2025-11-31.
12. *gptme Documentation*. [B.r.]. Dostupné tiež z: <https://gptme.org/docs>. Accessed: 2025-11-31.
13. *Gemini CLI Source Code*. [B.r.]. Dostupné tiež z: <https://github.com/google-gemini/gemini-cli>. Accessed: 2025-11-31.
14. *Gemini CLI Documentation*. [B.r.]. Dostupné tiež z: <https://geminicli.com/docs/>. Accessed: 2025-11-31.

15. *YAI Website*. [B.r.]. Dostupné tiež z: <https://ekkinox.github.io/yai/>. Accessed: 2025-11-31.
16. *YAI GitHub Repository*. [B.r.]. Dostupné tiež z: <https://github.com/ekkinox/yai>. Accessed: 2025-11-31.
17. *Aider Website*. [B.r.]. Dostupné tiež z: <https://aider.chat/>. Accessed: 2025-11-31.
18. *Aider GitHub Repository*. [B.r.]. Dostupné tiež z: <https://github.com/Aider-AI/aider>. Accessed: 2025-11-31.

Použitie nástrojov umelej inteligencie