Empovering Program Control with Natural Language using Large Language Models like Chat GPT.

Petr Talla

petr.talla@gmail.com

DRAFT

Terms Used

fine tunning is a technique in the field of deep learning, where we modify an already trained model for a new task or dataset by freezing most of its layers and then retraining it with a lower learning rate.

DSL - domain specific language - is a specialized language designed for a specific domain, which facilitates expressing and solving problems within this domain with a higher level of abstraction and comprehensibility for experts in the respective field.

LLM - large language model - is deep neural network trained based on a large amount of data and specific text prompts, which allows it to provide answers or generate texts corresponding to given input prompts.

backend - is a part of the software system that processes data, manages logic, and communicates with databases and other services. It provides functionality for the frontend.

frontend - is a part of the software system that is directly visible and interactive for users. It ensures the presentation and visual interface through which users communicate with the part of the application called the backend.

command - is a software pattern that separates the frontend from the backend. This improves modularity and keeps the application easily extensible.

Command - is an object used to implement the pattern command.

text command - is the object Command converted into textual form..

front controller - is a design pattern that centralizes the control of request flow in an application to a single point.

FrontController - is an object used to implement the front controller pattern.

Introduction

Recently, an AI application called Chat GPT was released, which can be described as a universal translator. Besides translating various foreign languages, it can also translate spoken language into programming languages. It is capable of translating even relatively complex tasks, for example:

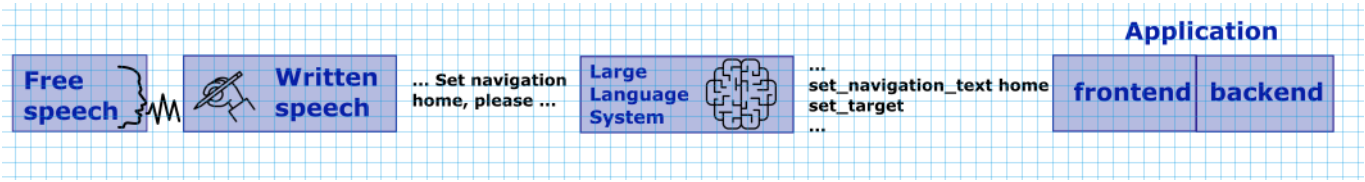
"In the shell language, write code that reads a text string from the input. Check if it is a valid Czech personal identification number. If it is, then verify if the number is present in the file pins.txt, where each number is on a new line. If the number is not found in the file, add the number to the file."

And here we have the generated code:

```
#!/bin/bash
#inputting the string from the user
read -p "Please enter a Czech personal identification number: " pin
# verification if it is a Czech personal identification number
if [[ $pin =~ ^[0-9]{6}/[0-9]{3,4}$ ]]; then
 # checking if the number is in the file
 if ! grep -q "^$pin$" pins.txt; then
 # adding a number to the file
 echo "$pin" >> pins.txt
 echo "The personal identification number has been added to the file."
 else echo
 "The personal identification number already exists in the file."
 fi
else echo
 "The given string is not a valid Czech personal identification number."
fi
```

Similar progress, as in the above example of translating spoken language into a list of commands, has also been made in the field of converting spoken language into written text. Additionally, Chat GPT is very tolerant of inaccuracies in input.

This creates a unique opportunity to control any program and, therefore, any device through freely spoken language. All that is required is for this program/device to be controlled through a set of text commands like shell, SQL, etc. The translation of spoken language into this set of commands can be provided by an LLM such as Chat GPT.



Chat GPT contains knowledge about a variety of extended command domains, such as shell, SQL, etc. If we were to look for examples of programs that have their own command domains and are included in the knowledge of Chat GPT, it would be beneficial to search in the field of CAD. In CAD, the use of the command line is a tradition that has never been abandoned, with programs like Autocad and Microstation being prominent examples.

Fine Tuning

But everything doesn't end there.

As of March 2023, Chat GPT allows users to use a technique called fine-tuning with a user-supplied set of prompt-requests. Facebook is now going further in that sense by releasing their Llama application for free usage, including the use of the fine-tuning technique.

What does fine-tuning actually mean? It's essentially retraining LLM. It allows injecting emphasis on knowledge that is specific to a certain domain into an already trained LLM. For example, it can be the help for your program, detailed knowledge about a particular location, or **knowledge of the specific DSL of your application.**

The training dataset for fine-tuning is simple, and it contains pairs following this schema:

```
...
{'prompt': '<prompt text>'}, 'completion': '<ideal generated text>'}
{'prompt': '<prompt text>'}, 'completion': '<ideal generated text>'}
{'prompt': '<prompt text>'}, 'completion': '<ideal generated text>'}
...
```

So, for example, for code generation in the shell, it would be:

```
...
{'prompt': 'in the shell language, execute: change the current directory to:
c:/home'},
 'completion': 'cd c:/home'}
{'prompt': 'display the file readme.txt in the shell'},
 'completion': 'cat readme.txt'}
{'prompt': 'Using the shell, append the text "hallo Charles" to the end of the
file readme.txt'},
 'completion': 'echo "hallo Charles" >> readme.txt'}
...
```

This set of training pairs is then used for fine-tuning a suitable LLM. You can then easily use it as a converter from spoken language to your DSL.

Commandline

From the previous text, it is evident that the only thing required to control a device using a freely spoken voice command is for the device to be controllable through a set of commands from the command line.

Controlling programs via the command line was the standard in archaic times of computing technology development. However, that is no longer valid in the present time, as programs are controlled through GUIs. As a result, they lost the ability to be controlled using technologies like LLM.

Let's consider the following situation when operating a car. The text provides both the control through the device's GUI description and then the voice command equivalent.

Pressing the start button. - "start yourself"
Switching to navigation mode. Entering the destination into the navigation through a text field. Activating the navigation. - "activate navigation target to home"
Switching to radio station tuning mode. Entering the station name into the text field. Activating the entered station. - "tune to Radio 3 station"

From the example, it's relatively easy to imagine how voice control can make operating devices more convenient and safer. For instance, setting a navigation destination while driving is almost impossible, and we would have to do it before the journey, whereas speaking while driving is a common occurrence. A person can talk while performing other tasks. However, they are not easily and safely able to drive a car while using the car's GUI or various buttons.

When using LLM, there is no need to use exact phrases as it was previously required for voice-controlled devices. This is probably the main reason why such voice control didn't spread widely. LLM ensures the translation of any variant of the command, even in any language. It can be, for example, "activate home navigation target", "set navigation to home", "navigation - home", "naviguj - domu.

In fact, there is no need for any precise knowledge of how the device works. It is sufficient to have a rough idea of what the device is intended for. Taking the example of a mobile phone, commands such as "call Peter," "phone number for Lojza is 658896785," "surname of Robert is Novak" can be used. Then, anyone can control the device without any prior knowledge, even people for whom operating such a device was previously inaccessible. LLM serves as a guide here, eliminating the need to browse through help instructions. Interestingly, in this case, it will probably still be necessary to use the GUI button to end the call.

What is necessary for us to voice-control devices like automobiles? It is only required that they can be controlled using the following text commands or similar. In the following text, there will always be a textual description followed by the corresponding text command

“start yourself up” - start
“set the text in the field to 'navigate to'” - set_navigation_text home
“enter the active destination from the 'navigate to' field” - set_target
“enter the text into the field 'radio station'” - set_radio_text “radio 3”
“set the active radio station from the 'radio station' field” – set_radio

LLM can then, for example, generate a script for the command "activate navigation target 'home'" as follows:

```
set_navigation_text home
set_target
```

Here, it is necessary to pause and reflect on why the script is not constructed like this:

```
set_target home
```

The reason is simple. LLM needs to be taught the sequence of text commands, and for that, it is beneficial to use a GUI. Therefore, it is a good idea to design a set of text commands that can be used in exactly the same form for both GUI and non-GUI interactions. GUI will include a field where we input the target of navigation. Thus, equivalent to entry in this field must be included in the text commands set.

There is also a second reason. For example, the command:

```
set_set_color_rgb 255 300 0
```

can perform validation checks on the entered values. However, this should be done by the backend, not the frontend or not just by the command

```
draw_line --color 255 300 0
```

This leads us to the problem of paradigm of the statelessness of communication. However, in ordinary speech, statelessness does not exist. In speech, context is heavily emphasized, primarily for the sake of communication efficiency. Otherwise, we would talk ourselves to death. Text commands must also replicate this characteristic of speech.

Command

Everything you need to do for controlling an application using voice is to control the application through text commands. So, how do you implement the control through text commands in the application?

Controlling an application through text commands is, in fact, an application of the classical object pattern called command. The command pattern strictly separates the frontend/UI of the program from the backend, which is the executor of the program's functionality. The purpose of this separation between the frontend and backend is to enable us to replace the frontend with another one while keeping the same backend. As a frontend, we can then use either a command-line interface or a graphical user interface (GUI).

And yes,

the frontend can also become an LLM that can translate spoken language into text commands, similar to input from the command line.

Here we encounter a fundamental problem in almost all existing programs. The GUI control of buttons, text boxes, etc. is typically solved using a hook function, which is simply written and attached to the button, and that's it. As a result, these programs lose a reasonable possibility of exchanging the frontend for another. There is also a mixing of functionality frontend-backend, so these programs become one large intertwined monolith.

This is the state of software development in the vast majority of companies. However, besides all the other disadvantages, this also eliminates the possibility of exchanging the frontend and, therefore, the instant way of controlling the program through natural language using LLM.

A bit of an exception is in web systems which separate the backend/server from the client/frontend using HTTP, allowing natural text-based communication. However, there is an issue with the stateless paradigm applied here, which will be mentioned later in the text. Also, for communication efficiency, some tasks, like parameter checks, are solved locally - these are then handled solely within the code and are not subject to text-based communication. Hence, they are also beyond the scope of LLM, but can be resolved by dividing frontend-backend parts already within the client.

The classical way of implementing the command pattern in object-oriented programming is through the object Command, respectivelly its specialization. Each such object contains an execute(...) method that performs the corresponding action in the backend. The user interface only knows the specific instance of the object Command and its execute(...) method, ensuring the separation of the frontend and backend. These Command objects can be used by different frontends, such as the command line, where the main goal is to activate the appropriate Command using the text command.

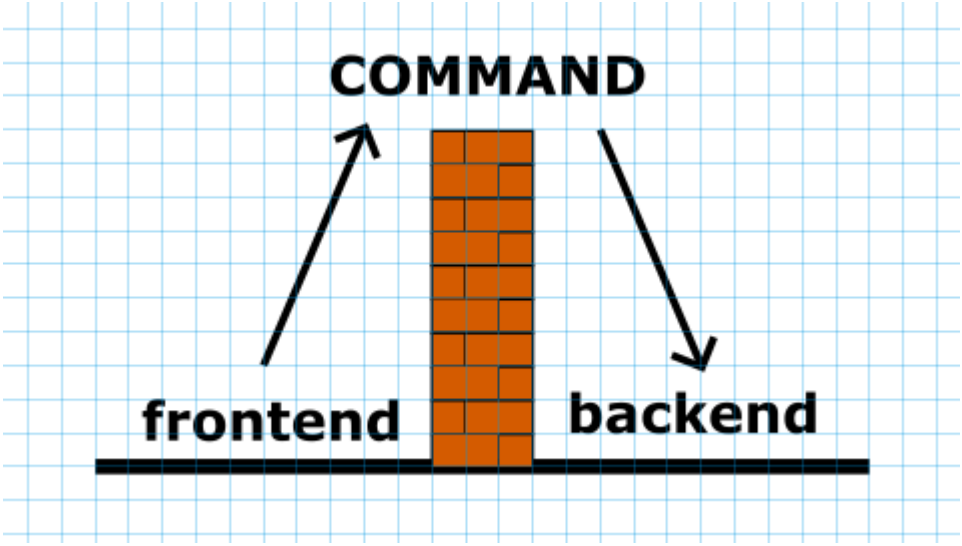


Image: Illustration of the separation of tasks between the frontend and backend using commands. We can imagine it like this: we have a wall, with the frontend on one side of the wall and the backend of the program on the other side. These two parts are completely unaware of each other's existence. At the top of the wall, there is a set of instances of Command objects. The frontend can see and communicate with these Command object instances. The Command objects, in turn, can see the backend and operate with it. Essentially, Command represents a specialized form of the Mediator pattern without any extra infrastructure.

To enable Command to mediate variables, it is good to equip it with a set of generic arguments. Additionally, it is beneficial to add a pointer to the context with which the Command will work, because it is not always appropriate to transfer the entire context solely through arguments. In such cases, the Command instance that switches the context must also be present, allowing the entire program to be controlled from the command line.

If we give the Command object a textual identifier and rules for converting the text into the Command object's arguments, we can easily activate the Command from the command line. We will refer to this text string as the text command, and it is essentially an exact equivalent of the Command object. The text command is what the LLM can manipulate.

To incorporate Command objects into the language system, it is necessary to assign them one additional property, apart from the textual ID and arguments that can be converted to text. Specifically, the translation of Command objects into spoken words. So far, we have discussed the reverse translation - from spoken words to text command. That is an amazing technology that has fascinated the whole world since Chat GPT's publication. However, the reverse process, i.e., from Command objects to spoken words, is straightforward - we can easily equip each Command object with a text generator that generates sentences in natural language, commenting on what is happening within it. This functionality is easily achievable through standard programming practices.

For each text command, we can generate an associated description:

x_set_navigation_text home → “In program X, set the text in the 'navigate home' field.”
x_set_target → “In program X, enter the current destination from the 'navigate' field.”

For this purpose, it is good to create a special instance of the object Command, whose sole function is to add additional textual commentary to this log.

By using this approach, it is possible to generate an adequate number of prompt-completion pairs, which can then be used for fine tunning LLM on our command domain.

You will notice that both the text commands and the generated spoken descriptions above are accompanied by the contextual prefix 'x_' and 'In program X ...'. This contextual prefix must also be applied to the voice control of the machine. Due to the nature of LLM functionality, it is sufficient to provide it in context only at the beginning and does not need to be used in every sentence separately.

"Further generate commands in language X."

Next, the patern command provides the possibility of easy documentation of program functionality. Simply describe the functionality of individual instances of the Command object. It is ideal for each implementation of the command to return the relevant help, for example, in Markdown language directly from each Command object. The help should be well-structured to enable to generate a sufficient amount of additional educational material for the fine tunning process.

The Pattern Command Implemented in the Code.

The command pattern described in the previous text can be implemented in C++ using the following class:

```
class Command {
public:
   string id() {return id_;}
   virtual int execute ( ArgumentList& args, void* context = nullptr ) = 0;
   virtual string describe ( ArgumentList& args, void* context = nullptr )
             // it describes in natural language what the command executed
   virtual string help();
protected:
   string id_;
}
```

ArgumentList is intentionally used as a reference here. Command execution can extend it with additional information, such as error state descriptions, and so on. The return value of execute(...) is then the count of these new arguments.

You can also use the extension of the argument list for returning a value from the system if the Command has some value to return. This is not the primary purpose of the Command class - its main purpose for us is to give commands to the backend, not to control how the backend presents itself. However, using the Command pattern, you can create some basic presentation in this way.

Front Controller

The Front controller is another programming pattern that is well suited to be used in conjunction with the command pattern. It essentially involves centralized processing of Command objects, which is necessary when using the command line, but it is also beneficial to use it in GUI applications, even though you could directly associate GUI elements with corresponding Command objects here.

The FrontController object can be implemented as a list of Command objects. The FrontController is requested to execute a specific Command object by using the identifier of the Command and a list of arguments.

The FrontController treats each Command object as if it were passing through it. Due to the generic nature of the Command object, various plugins can be executed on it during the processing in the FrontController object. The execution of the Command object itself can also be implemented as a plugin.

It is also possible to apply various other plugins, such as filtering, modifying arguments, recording, logging, and generating Command objects for undo purposes.

As a plugin, you can also apply recording of verbally commented records of invoked Command objects, which can then serve as training data for the fine-tuning of the LLM on the command domain of your program.

Quick Simulation of Using LLM to Control a Program.

LLM is limited only to commonly known command domains, such as SQL, shell, etc. If a command domain was not present in the original training set of LLM, it cannot translate it. However, you can partially simulate the corresponding DSL if you provide the relevant knowledge to it in context.

So, you will write some introduction. For example: "In my CAD system, I can set colors using the command 'set_color r g b', where the arguments r, g, b represent the color in the red-green-blue encoding. Furthermore, I have the command 'draw_line' that sets the mode for drawing lines. After specifying two points using the command 'xy x y', where x and y are the coordinates of the points, a line is drawn between them. To draw another line, I input the next pair of points using the 'xy' command."

Etc. In this way, you can describe a series of text commands of the system.

After completing the contextual instructions, you can write the verbal command: 'Write commands to draw a yellow line from coordinates 1 2 to coordinates 8 5.'

From Chat GPT, you get:

```
set_color 255 255 0
draw_line
xy 1 2
xy 8 5
```

So, based on natural speech, you obtained a list of text commands that the program can execute. The only thing you had to do for this is to implement the command pattern in the program. Everything else happens through other technologies outside of the program.

The above-mentioned approach should be kept only at the experimental level. Especially the scope of the given context will have its limitations. A typical program usually requires hundreds of text commands for functioning. A more serious approach is to perform fine-tuning of LLM on a set of commands from our command domain.

More Complex Use Cases.

Combining GUI and voice input is not a problem at all. All commands pass through the front controller, and the context can be relayed to LLM using the system logs.

It works well even when the data is supplied to the system by the machine itself, for example, from various sensors. In that case, the machine must also communicate with the system through a front controller. In general, there should be no piece of code that is executed outside the scope of the front controller. This ensures that LLM can orient itself in the current situation.

In this way, it is also possible to have a combination where you don't use LLM for directly controlling the machine, but it merely guides you on what to do when operating through command-line or GUI. LLM only observes the system's actions and serves as a sophisticated contextual help.

In Conclusion

Separate the frontend and backend and teach your program to use text commands. Generate prompt-response pairs and perform fine-tuning of a certain LLM. Then, you can control your program or device using natural speech in any language.

Or at least invest money in corporations manufacturing headsets and noise reduction systems. Avoid building more open spaces, or if you do, divide them with sound-isolated cubicles. In the foreseeable future, we will predominantly communicate with computers and machines using natural speech. Keyboards and mice will only be occasionally used as supplements.

In the TV series "Red Dwarf," an eccentric toaster plays one of the supporting roles. This article provides a realistic description of the process of creating it.

Lister: "No. Shhh. I'm busy."
Talkie Toaster: "Not busy eating toast though are you?"
Lister: "I don't want any."
Talkie Toaster: "The whole purpose of my existence is meaningless if you don't want toast."
Lister: "Good."
Talkie Toaster: "I toast, therefore I am."