# Developing an Interface between an LLM and the GUI of RCE

Bachelor's thesis

for the degree

Bachelor of Science

of the course Informationstechnik

at the Baden-Wuerttemberg Cooperative State University Mannheim

by

**Tim Rosenbach**

Submission on August 27, 2024

| | |
|---|---|
| Processing Period: | 04.06.24 – 27.08.24 |
| Student id, course: | 3492133, TINF21IT1 |
| Department: | Institute of Software Technology: Intelligent and Distributed Systems |
| Apprenticing company: | German Aerospace Center (DLR) |
| Company's supervisor: | Dr. Alexander Weinert |
| University's reviewer: | Prof. Dr. Karl Stroetmann |

# Declaration

I hereby assure you that I have written my bachelor's thesis on the

SUBJECT

**Developing an Interface between an LLM and the GUI of RCE**

independently and that I have not used any other sources and aids than those indicated.

I also assure you that the electronic version submitted is the same as the printed version.*

* if both versions are required.

Cologne, August 27, 2024

# Abstract

Remote Component Environment (RCE) is a complex software used by engineers to create and run simulation tool chains for complex systems. Because of its complexity, RCE has a steep learning curve, especially for new users. Reducing this learning curve is crucial to making RCE more accessible to new users. In this thesis, we develop a system that simplifies the integration of external tools into RCE, using Large Language Models (LLMs). The system creates an interface between RCE and an LLM that enables the LLM to solve tasks using the Graphical User Interface (GUI) of RCE. This system serves as a proof of concept that shows the great potential of using LLMs to solve tasks in RCE.

# Zusammenfassung

Remote Component Environment (RCE) ist eine komplexe Software, die von Ingenieuren zur Erstellung und Ausführung von Simulationswerkzeugketten für komplexe Systeme verwendet wird. Aufgrund seiner Komplexität hat RCE eine steile Lernkurve, insbesondere für neue Benutzer. Die Absenkung dieser Lernkurve ist entscheidend, um RCE für neue Benutzer zugänglicher zu machen. In dieser Arbeit entwickeln wir ein System, das die Integration von externen Tools in RCE vereinfacht, indem wir LLMs verwenden. Das System schafft eine Schnittstelle zwischen RCE und einem LLM, die es dem LLM ermöglicht, Aufgaben mit Hilfe der GUI von RCE zu lösen. Dieses System dient als Proof of Concept, welches das große Potenzial der Verwendung von LLMs zur Lösung von Aufgaben in RCE zeigt.

# Contents

# List of Figures

# List of Tables

# Abbreviations

**DLR** German Aerospace Center

**IVS** Inteligent and Distributed Systems (Intelligente und Verteilte Systeme)

**RCE** Remote Component Environment

**SWT** Standard Widget Toolkit

**GUI** Graphical User Interface

**LLM** Large Language Model

**CPACS** Common Parametric Aircraft Configuration Schema

**ACE** Actor-Critic Embodied Agent

**OCR** Optical Character Recognition

**API** Application Programming Interface

**RCP** Rich Client Platform

# Listings

# Appendix

# 1. Introduction

As modern engineering projects grow in complexity, advanced software tools to manage and execute simulations have become indispensable. One such software tool is Remote Component Environment (RCE) developed by German Aerospace Center (DLR). It enables engineers to create, manage, and execute simulation tool chains for complex systems, such as aircraft, ships, and satellites. Due to the complexity of these use cases, RCE offers a wide range of functionalities, making it a powerful but also complex tool for engineers. Therefore, RCE has a steep learning curve, especially for new users.

In light of this challenge, this thesis explores the potential of using LLMs to simplify the usage of RCE for new users. For this we will develop a system that acts as an interface between RCE and an LLM. The system receives a task from the user in textual form and then enables the LLM to solve it using the GUI of RCE. To achieve this, the system gives the LLM information and context about the GUI of RCE and enables it to perform actions on it.

The goal of this thesis is to determine if using LLMs to automate tasks in RCE is feasible. This will be measured by how well the LLM can solve a given task on its own by controlling the GUI of RCE. The task will be a typical task that an engineer would perform in RCE, such as integrating an external tool into RCE. We will also test the system with different LLMs to determine which LLMs are suitable for this task. The test compares how effectively and efficiently the LLMs can solve a given task using the GUI of RCE.

# 2. Background and Related Work

In this chapter, we will introduce the necessary background information for this thesis. We first introduce the software RCE in Section 2.1. After that we explain what LLMs are in Section 2.2. In Section 2.3, we explain different prompting techniques that can be used with LLMs. In Section 2.4, we present two examples of how an LLM can interact with external tools.

## 2.1. RCE

RCE [2] is an open-source software for creating and executing simulation toolchains of complex systems, such as aircraft, ships, and satellite. It is primarily developed and maintained at the DLR in the Institute of Software Technology by the department Inteligent and Distributed Systems (Intelligente und Verteilte Systeme) (IVS). RCE supplies the user with a GUI, which can be seen in Figure 2.1. The GUI is based on the eclipse Rich Client Platform (RCP) [30]. This platform is programmed in Java and uses the Standard Widget Toolkit (SWT) for the graphical interface. SWT [33] is an open-source widget toolkit for Java. It uses the native widgets of the operating system to create the graphical interface. Such widgets are for example buttons, text fields, and labels.

Users of RCE can use the GUI of RCE to create and execute data-driven workflows. These workflows consist of components. Each component has predefined inputs and outputs, that each have a specific datatype. The user can connect an output with an input if they have the same datatype. Possible datatypes are Bool, Directory, File, Float, Integer, Matrix, Short Text and Vector.

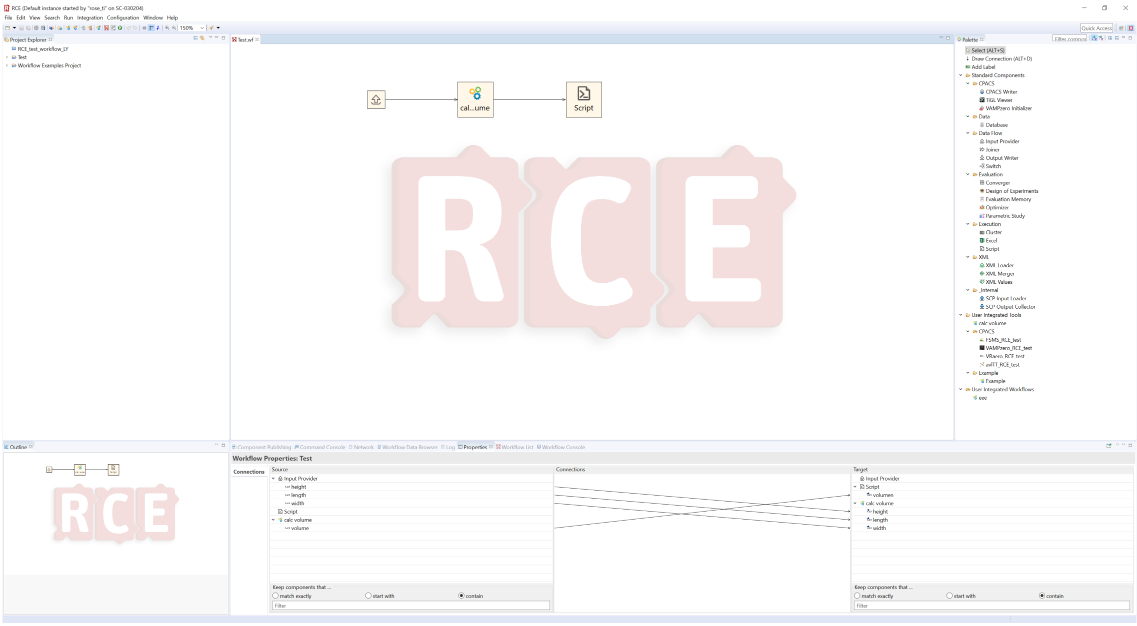



**Figure 2.1.:** The RCE GUI

There are two types of components, Standard Components and User Integrated Tools. RCE provides the user with Standard Components, which they can use to manipulate the data flow or create optimization loops. The User Integrated Tools are external tools that the user has integrated into RCE. The user can only integrate an external tool into RCE if they are able to execute it from the command line. For the integration process RCE provides the Tool Integration Wizard, which leads the user through a step-by-step guide.

The Tool Integration Wizard consists of six pages which leads the user through each integration step. The different pages are shown in Figure 2.2. In the following, we explain each page in detail.

On the first page (Figure 2.2(a)), the user can select the type of tool they want to integrate. There are two types of tools, common tools and Common Parametric Aircraft Configuration Schema (CPACS) [4] tools. There is also the option to integrate a tool from a template.

The second page (Figure 2.2(b)) is there to specify general information about the tool. The user must specify the name. Other information like the icon path, the group path, path to the documentation, a description and the contact information of the author are optional.

On the third page (Figure 2.2(c)) the user can specify the input and output of the tool. Furthermore, it is possible to activate the Manual Tool Result Verification option. This option is used to verify the output of the tool manually after its execution.

The fourth page (Figure 2.2(d)) is used to specify properties for the tool. A property is a value that can be set for the integrated component after the integration. This value is constant for the workflow execution.

The fifth page (Figure 2.2(e)) is used to specify the launch settings. This includes the path to the tool directory and a path to the working directory. The tool directory is the directory where the tool is located. The working directory is the path from where the tool is executed. It is also possible to specify whether the tool should be copied to the working directory before execution. Furthermore, the user must specify if the working directory should be deleted after the execution.

On the last page (Figure 2.2(f)), the user must specify the command that executes the tool. Furthermore, they can define a pre- and post-execution script, that executes before and after the tool is executed. In all of these it is possible to use placeholders for the previously defined inputs, outputs, properties and directories. A placeholder is a string that is replaced by a value during the execution of the tool, e.g. "${in:radius}" is replaced by the value of the input radius. After that the user can finish the integration and the tool is available in RCE as a component, which can be added to a workflow.

RCE saves the integrated tools as files in a specific format. That means it is also possible to integrate tools by creating these files manually and adding it to the RCE directory, where the integrated tools are stored.

As RCE is a complex software, it has a steep learning curve. To help the user with getting familiar RCE has a User Guide [35] that explains the usage of RCE. In addition to that, there is a documentation integrated into the GUI of RCE. This documentation gives information about the different components of RCE and how to use them. Although the documentation gives a good conceptional overview of RCE, it does not explain every little detail.

## 2.2. Large Language Models

Large Language Models (LLMs) are a class of deep learning models designed to generate human language. These models are typically built on transformer architectures, which enable them to process and generate text in a highly sophisticated manner. The transformer architecture was introduced by Vaswani et al. [36] in 2017. It is based on the so-called self-attention mechanisms, which allow the model to weigh the importance of different words in a sentence. This enables the model to process the context of a word in a sentence.

One key concept in the functioning of LLMs is the use of tokens. In this context, a token is a fundamental unit of text that the model recognizes as a single entity. These tokens can be as small as individual characters or as large as entire words depending on the tokenization process used by the model. Before an LLM processes any text, it first breaks the text down into these tokens, which serve as the building blocks for understanding and generating language. The way a model tokenizes text can greatly impact its ability to understand context and generate coherent output.

The models are trained on large amounts of text data. Among other things this allows the model to learn the structure of human language. Most of the text data is collected from the internet, which accumulates to thousands or millions of gigabytes' worth of text. During pre-training, the model learns to predict the next token in a sequence, given the preceding tokens. This token-based approach not only enhances the model's efficiency but also allows it to handle diverse linguistic structures more effectively. As the models are trained on wide and diverse text data, their application areas are vast. They can be used for sentiment analysis, DNA research, customer service, chatbots, and online search. [37]

LLMs have a limit on the number of tokens they can process at once, known as the "context window" or "maximum context length". This token limit is crucial as it defines the amount of information the model can consider at any given time. The context length increases when the model generates text, as it appends new tokens to the input sequence. If the context length exceeds the limit, the model "forgets" the earlier tokens and only considers the most recent ones. For example the Llama-2 model has a vocabulary size of 32000 different tokens and a context window of 8192 tokens. [12] ChatGPT-4 has a vocabulary size of 100256 different tokens [7] and a context window of 8192 tokens. [25]

## 2.3. Prompt Engineering Techniques

To utilize the full potential of LLMs, there exists prompt engineering techniques that define how a prompt should be structured to help to guide the model towards a specific task. There are different prompt techniques that have different use cases. The most common ones are the zero-shot prompt technique, the few-shot technique and the chain of thought prompt technique. This section including the examples of the prompts are based on the Prompt Engineering Guide by DAIR.AI. [5]

The zero-shot prompt technique uses prompts that do not contain any examples or demonstrations. Because LLMs are trained on a wide range of data, for some cases this technique is sufficient. For example the following prompt uses the zero-shot prompt technique.

| Prompt |
|---|
| Classify the text into neutral, negative or positive.<br>Text: I think the vacation is okay.<br>Sentiment: |

The prompt delivered the following output.

| Output |
|---|
| Neutral |

Despite not having any examples the LLM was able to classify the sentiment of the text as “neutral”.

The few-shot prompt technique on the other hand uses prompts that contain examples. This enables in-context learning, which allows for better performance. For example the following prompt uses the few-shot prompt technique.

**Prompt**

A "whatpu" is a small, furry animal native to Tanzania. An example of a sentence that uses the word whatpu is: We were traveling in Africa and we saw these very cute whatpus.
To do a "farduddle" means to jump up and down really fast. An example of a sentence that uses the word farduddle is:

The prompt delivered the following output.

**Output**

When we won the game, we all started to farduddle in celebration.

The output shows that the LLM was able to understand the meaning of the word “farduddle” and use it in a sentence. This shows that the few-shot prompt technique can be helpful for more complex tasks.

The chain of thought prompt technique uses reasoning steps to guide the LLM through a task. The technique can also be combined with the few-shot prompt technique. For example the following prompt uses the chain of thought prompt technique.

**Prompt**

The odd numbers in this group add up to an even number: 4, 8, 9, 15, 12, 2, 1. A: Adding all the odd numbers (9, 15, 1) gives 25. The answer is False.
The odd numbers in this group add up to an even number: 17, 10, 19, 4, 8, 12, 24. A: Adding all the odd numbers (17, 19) gives 36. The answer is True.
The odd numbers in this group add up to an even number: 16, 11, 14, 4, 8, 13, 24. A: Adding all the odd numbers (11, 13) gives 24. The answer is True.
The odd numbers in this group add up to an even number: 17, 9, 10, 12, 13, 4, 2. A: Adding all the odd numbers (17, 9, 13) gives 39. The answer is False.
The odd numbers in this group add up to an even number: 15, 32, 5, 13, 82, 7, 1. A:

The prompt delivered the following output.

**Output**

Adding all the odd numbers (15, 5, 13, 7, 1) gives 41. The answer is False.

The output shows that the LLM was able to understand the task and solve it by dividing the task into smaller reasoning steps.

## 2.4. Ways for LLMs to Interact with External Tools

LLMs are very versatile and can be used in many ways. One way to use LLMs is to create an interface between the LLM and an external tool. This opens up possibilities such as automatization of testing, task solving and many more. Creating such an interface a highly novel approach, with limited instances of implementation to date. Nonetheless, in the following we present two examples of how an interface between an LLM and an external tool can be created and used. In Subsection 2.4.1 we explain how an LLM can be used to automate GUI testing of mobile applications. In Subsection 2.4.2 we explain how an LLM can be used to automate task solving for complex software such as After Effects or Premiere Pro.

### 2.4.1. Automating GUI Testing of Mobile Applications

Liu et al. [15] developed a system called GPTDroid. The architecture of GPTDroid is shown in Figure 2.3. It creates an interface between the LLM and the mobile app. This is achieved by extracting the GUI of the mobile app and converting it into a prompt. The GUI information consists of general information about the app, e.g. the name of the app, and the detailed information about the current page of the mobile app. The detailed information consists of widgets, e.g. a text field or a button.

The LLM then uses this prompt to decide which action it wants to perform on the mobile app. There are predefined actions that the LLM can choose from, e.g. click/double click a widget. The action is then executed on the mobile app. In addition, the functional-level progress is saved. After that another iteration starts. The GUI is extracted again and a new prompt is created. In addition, the functional-level progress is also added to the prompt. This process is repeated until the LLM has reached the desired state or a predefined maximum number of iterations is reached.

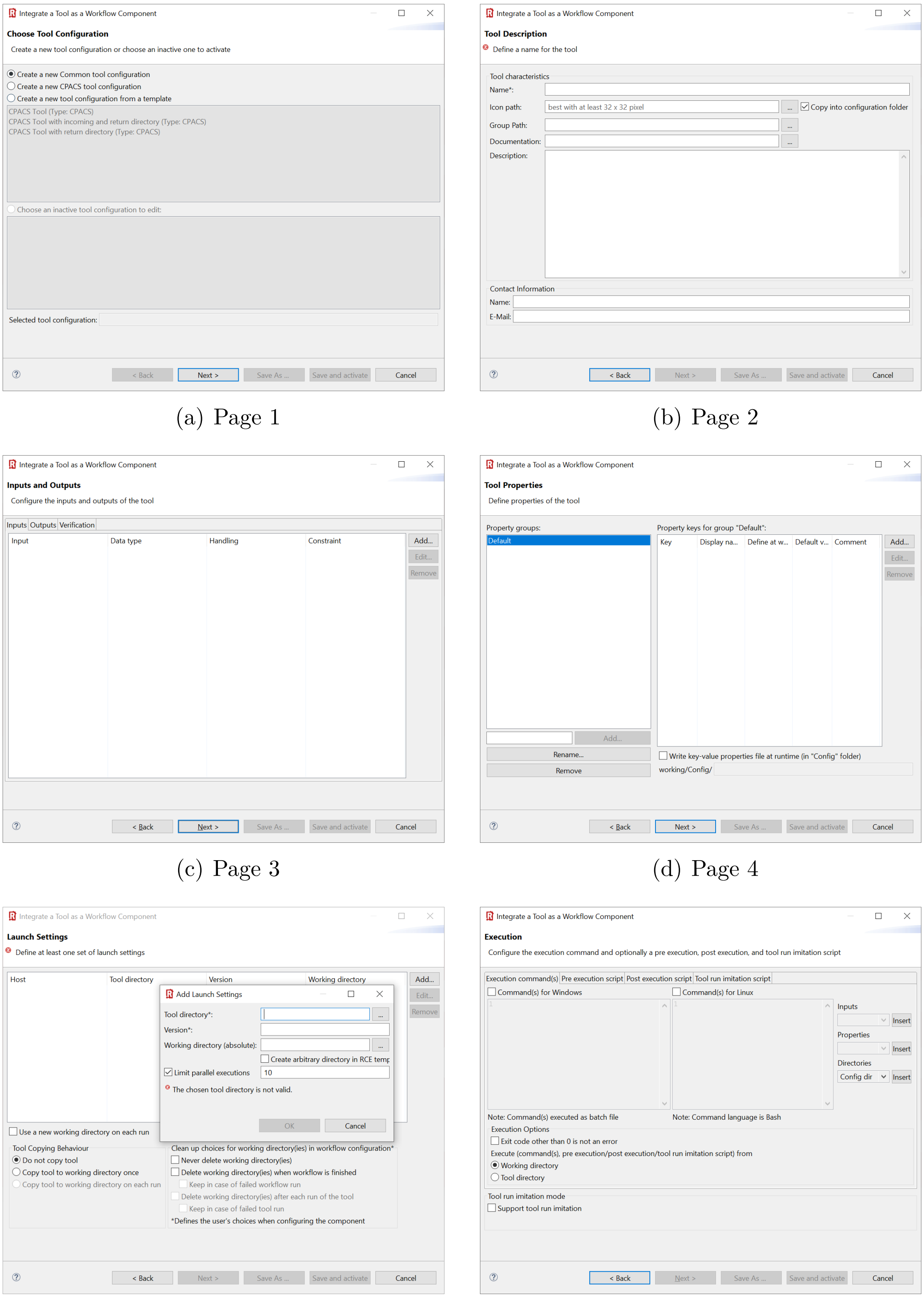
With this technique GPTDroid covers on average far more widgets and activities than the baseline (Ape [1] with QTypist [34]). More precisely, GPTDroid has an activity coverage of 75% which is 32% higher than the baseline. The activity coverage is the number of triggered activities on the stack during exercising, which are registered in the AndroidManifest.xml. [11] Furthermore, GPTDroid was able to detect 95 bugs for the 93 tested apps. This shows that a testing-system based on LLMs can be very effective.

### 2.4.2. Automating Task Solving for Complex Software

Gao et al. [6] developed a framework called Actor-Critic Embodied Agent (ACE). The framework uses an LLM to control the GUI of a software to solve a given tasks. The architecture of the framework is shown in Figure 2.4. The ACE consists of three agents who communicate with each other.

The input of the ACE is a query with the task it should solve and an instructional video. The video gives information about how to solve different tasks in Premiere Pro. The query and the video are given to the planner agent. This agent creates sub-tasks that need to be accomplished to achieve the given task. These sub-tasks are then passed on to the actor and critic agents. They are also provided with information about the desktop environment or more precisely the current state of the UI. For this a screenshot of the UI is translated into a textual description by a GUI parser. The GUI parser uses Google OCR [22] for extracting text, Yolo-v8 [38] to coarsely localize objects, and LangSAM [8] to obtain the precise object contours. With the current sub-task and the state of the UI the actor agent decides on actions to execute. The system then executes these actions, using the Python library PyAutoGUI. After that the critic agent evaluates the updated state of the UI and decides whether the sub-task has been completed. If it has been completed the actor agent works on the next sub-task. Otherwise, it continues to work on the current sub-task. This cycle continues until all sub-tasks are completed.

Gao et al. [6] tested different LLM combinations of planner, actor and critic agents on the ACE. They tested the LLMs GPT-4, GPT-3.5 and Llama2. The result showed that only when GPT-4 is used for all agents, a satisfactory result was delivered. Furthermore, they showed that their approach notably outperforms existing methods in GUI automation.




(a) Page 1

(b) Page 2

(c) Page 3

(d) Page 4

(e) Page 5

(f) Page 6

**Figure 2.2.:** Tool Integration Wizard

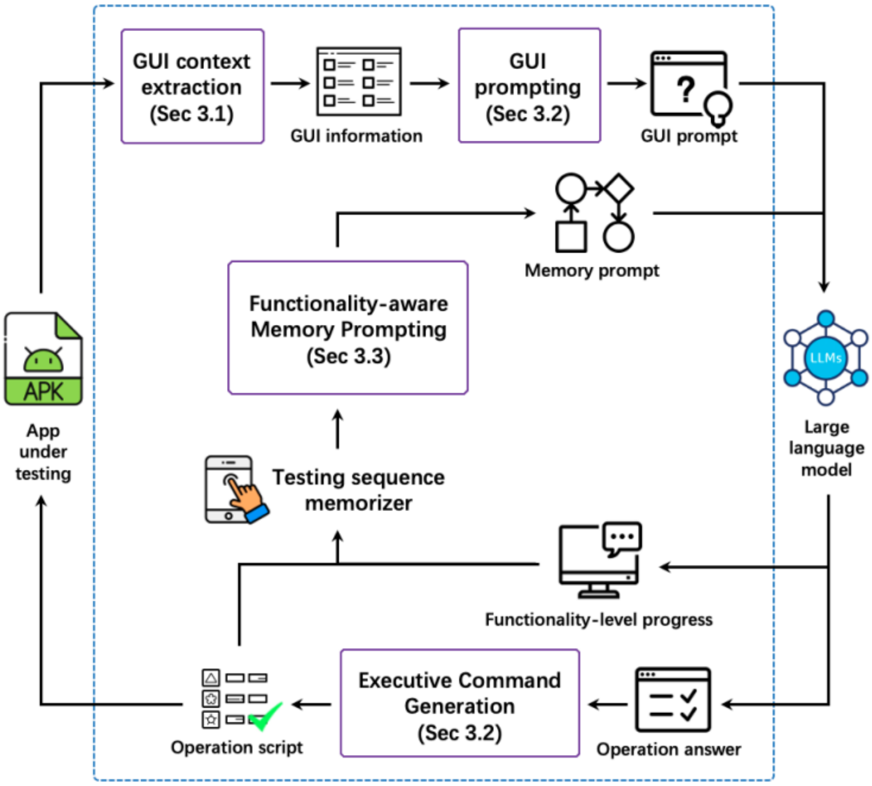



**Figure 2.3.:** Architecture of GPTDroid [15, Figure 2]

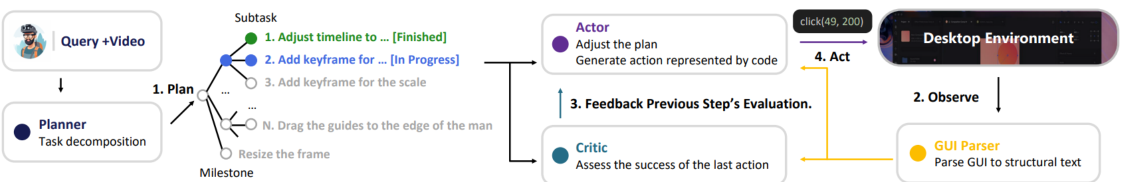



**Figure 2.4.:** Architecture of Actor-Critic Embodied Agent [6, Figure 4]

# 3. Controlling Software via LLMs

In the last couple of years there have been rapid developments in the field of LLMs. [39] These developments lead to new possibilities for their application in areas such as automatization. For example, Liu et al. [15] showed that it is possible to automate GUI testing with LLMs. LLMs can also be used to simplify the usage of software. Gao et al. [6] showed that it is possible to use LLMs to simplify the usage of a complex software such as After Effects. In these cases an interface between the LLM and the software was established.

RCE is a software which is comparably complex to After Effects. It is also a software that undergoes regular GUI testing, which at this point is not automated. [21] Therefore, there is potential to use LLMs similarly as Liu et al. [15] and Gao et al. [6] did. So creating an interface between RCE and LLMs would be beneficial for future developments of RCE.

Rosenbach explored the usability of RCE for new users in his T3_2000 report [31]. The report showed that RCE has potential to simplify the usage of RCE for new users. It was shown that the integration of external tools into RCE is one of the most difficult tasks for new users. The report developed a concept for a Tool Integration Wizard that is easier to use than the current one. In this thesis we want to explore a different approach to simplify the integration of external tools into RCE. This approach uses LLMs to assist the user with the integration process.

So the goal of this thesis is to create a system that simplifies the integration of external tools into RCE by using LLMs. We want to enable users to describe the external tools they want to integrate and let the LLM execute the integration. As this reduces the need of understanding how RCE operates, this can potentially simplify the usage of RCE. The execution of the LLM should be done on the GUI of RCE rather than just creating the tool integration files. This way the integration process is more comprehensible for the user, because the user can see how the LLM interacts with the GUI of RCE. In addition, it offers more potential for future applications that use LLMs for the automatization of RCE, e.g. GUI testing.

To achieve the goal we develop and implement a system that acts as an interface between RCE and an LLM. Such a system was never developed before. Therefore, this thesis acts as a proof of concept for the idea of using LLMs to simplify the usage of RCE. This proof of concept will also show the potential of using LLMs for other tasks related to RCE, for example the automatization of GUI testing. We will also evaluate the system by giving it a test case. As the system heavily depends on the LLM used we will also evaluate the system with different LLMs. This will also show what LLMs are suitable for this task and what LLMs are not.

# 4. Concepts and Implementation

For the implementation of the system, we combine the two existing concepts of Liu et al. [15] and Gao et al. [6] and apply them to the problem. Both concepts deal with the creation of an interface between an LLM and a target system. Therefore, their insights can be used to solve the problem of creating an interface between an LLM and the GUI of RCE to automate the integration of external tools into RCE. In Section 4.1 we will present the overall architecture of our system and explain how we combine the two concepts to solve this problem.

## 4.1. Overall Architecture

We want to build an interface between the GUI of RCE and an LLM. We have a similar goal as Gao et al. [6] namely to automate the usage of a complex software. However, the architecture of the ACE is very complex and not feasible for this thesis. Therefore, we want to build a simpler system that is more similar to GPTDroid. To achieve this we will base our system on the architecture of GPTDroid and add some elements of the ACE.

First we have to decide on which level our system should establish the interaction between the LLM and the GUI of RCE. We have two options. The first option is to use Optical Character Recognition (OCR) and other image recognition tools to extract the GUI information similar to the approach of Gao et al. [6]. This option has the advantage that it is more flexible. It does not depend on the underlying technology of the GUI or tools that extract the GUI information. On the other hand, it is very complex to implement and needs a lot of computing capacity. The second option is an interaction with the GUI on the operating system level similar to the approach of Liu et al. [15]. This option has the advantage that the functionality of the individual elements and their possible actions can be extracted. However, with this option there is the risk that the underlying structure of the GUI might be difficult to interpret.

As the GUI of RCE is based on the Eclipse RCP with the SWT, the second option is more suitable. This is because the SWT uses the native widgets of the operating system to create the graphical interface. Therefore, the risk that the underlying structure of the GUI is difficult to interpret is low. Moreover, the functionality and potential actions of individual elements are well-defined, as native operating system widgets come with predetermined capabilities. This immensely helps to give the LLM a better understanding of the GUI. Therefore, we will create a system that interacts with the GUI on the operating system level.

The next step is to create a system that is capable of simplifying the integration of external tools into RCE. The architecture of our system is shown in Figure 4.1. We use a similar architecture as GPTDroid. The system interacts with three components. In addition to the LLM and the target system like GPTDroid, we also have the user. The user provides the system with a task that should be solved. The task consists of a description of the external tool that should be integrated into RCE.

RCE provides the system with a GUI. The GUI parser translates the GUI into a textual form. How the GUI is translated into a textual form is explained in Section 4.2. The ACE has shown that for the use-case of automating complex software, it is important to add explanatory information about the software. In the approach of Gao et al. [6] this is done with an instructional video. For RCE such videos do not exist. However, RCE has a textual documentation of itself mentioned in Section 2.1. This will be used in our system.

The task, the documentation and the textual form of the GUI are then combined into a prompt. How exactly the prompt is structured is explained in Section 4.4. In addition, the prompt contains a list of possible actions that the LLM can use to control the GUI. This is similar to the approach of ACE, where there are predefined actions that can be called on a position, e.g. click(481, 924). However, the actions

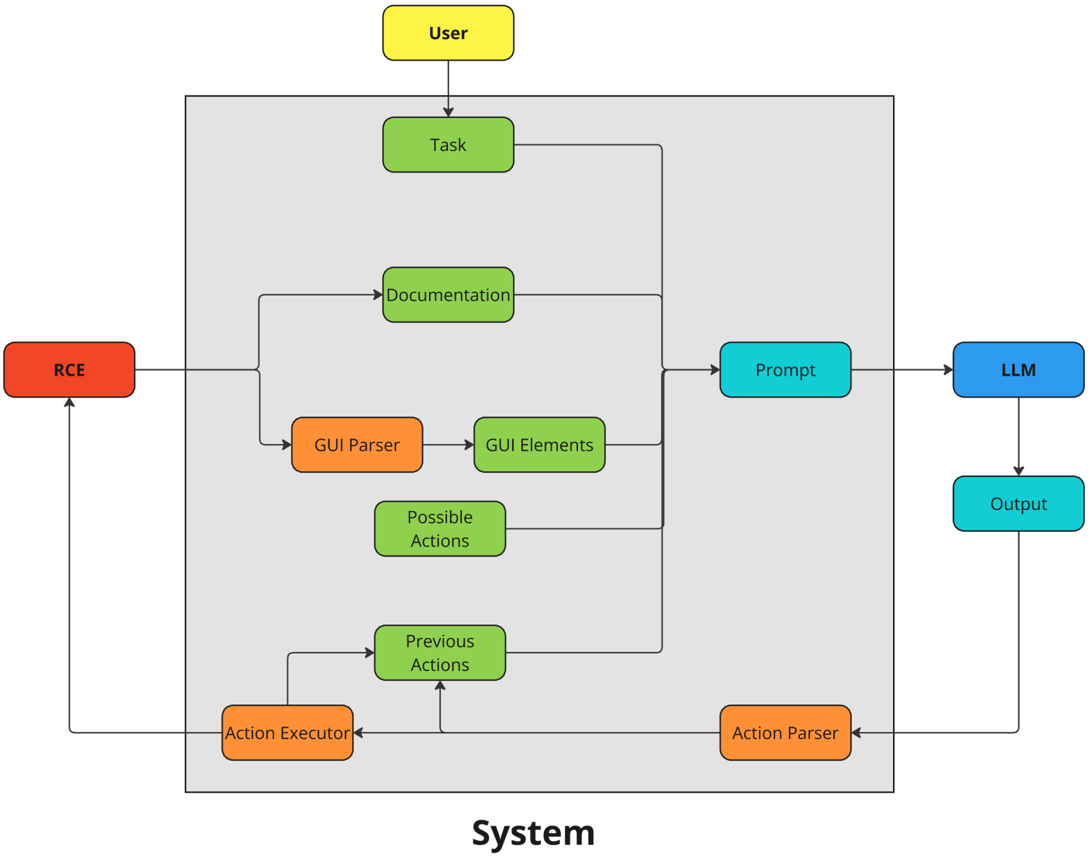



**Figure 4.1.:** Overview of our System Architecture

are more specific at our system, because we can extract which actions are possible for each control element and also execute the actions on these elements. Which specific actions are possible and how they are executed is explained in Section 4.3. If the LLM has already performed some actions, these are also included in the prompt. The prompt is then passed to the LLM.

The LLM uses the prompt to decide what it wants to do on the GUI and outputs an action. The action is then parsed into a format that can be executed by the action executor. The action executor then executes the action on the GUI. Because the action is executed on an operational level, the action executor directly receives feedback about the success of the execution. The feedback will indicate if the action was executed successfully and provide reasons if it was not.

It is important for our system to save the functional-level progress like GPTDroid, so that the LLM knows what it has already done. Therefore, the action and the feedback of the action executor are saved for future iterations, where they will be added to the prompt. After that the process is repeated until the LLM has solved the task or a maximum number of iterations is reached, to prevent the system from getting stuck in a loop.

In the following sections we will explain the individual components of the system in more detail.

## 4.2. GUI-Parser

For extracting the GUI information from RCE we use the Python library PyWinAuto. We describe it in more detail in Subsection 4.2.1. In Subsection 4.2.2 we explain how we create a structure from the extracted GUI information. In Subsection 4.2.3 we explain which concrete data we extract. Because there are also icons in the GUI of RCE we need an image recognition tool. For this we use a tool called LLaVA, which we describe in Subsection 4.2.4.

### 4.2.1. PyWinAuto

PyWinAuto [29] is a Python library that allows the user to automate the GUI of Windows applications. It enables the user to extract information about the GUI of a Windows application in form of control elements, such as text fields and buttons. Furthermore, the user can automate user inputs, such as mouse clicks or keyboard inputs.

There are two backends for the library. The first backend is called MS UI Automation (UIA). The second one is called Win32 API. They differ in the way they handle the GUI of the Windows application. Depending on the application and the goal of the automation, one of the two backends is more suitable. Most times determining which backend is more suitable is a trial and error process during the development of the automation. Although, within the scope of this bachelor thesis it is only possible to try out one backend. Therefore, we have to decide which backend is more suitable for our use case, before we start the implementation. To make this decision we use the Microsoft tool inspect.exe. [14] This tool can be used to inspect the GUI of a Windows application with either of the two backends. It shows of what components the GUI consists of and how they relate to each other.

The Win32 API focuses on the windows of an application compared to the MS UI Automation which focuses on the application itself. That means that with the Win32 API backend every window of the application has its own root element. RCE consists of multiple windows. For example the Tool Integration Wizard is an independent window. With Win32 this allows us to focus on this window and filter out the other windows of RCE more easily. Furthermore, the MS UI Automation backend extracts elements that are not needed for our use case, such as scroll bars. These elements are not needed because they are elements that the LLM does not need to interact with. Therefore, we assume that the Win32 API backend is more suitable for our use case.

To use PyWinAuto, we first need to connect to the application. If the application is already running, we connect to it. Otherwise, we start the application first. This is done with the code snippet in Listing 4.1. The default timeout for connecting to an application is 5 seconds. Because RCE needs more time than that to start, we increase the timeout to 50 seconds, when starting the application.

```python
def start_or_connect_rce() -> Application:
    try:
        app = Application().connect(title_re="RCE*")
    except ElementNotFoundError:
        app = Application().start(RCE_PATH)
        app.connect(title_re="RCE*", timeout=50)
    return app
```

**Listing 4.1:** Connecting to or starting the application with PyWinAuto

### 4.2.2. Creating a Structure

The next step is to extract the basic structure of the GUI by using PyWinAuto. We want to describe the GUI in a textual form that gives the LLM enough information to understand the GUI. Given the typical 8k token limit of current LLMs' context windows, we aim to keep the textual GUI representation as concise as possible in this limit. So we want to find the most effective and efficient way to describe the GUI.

The first step to finding a suitable representation of the GUI is to get an understanding of which components RCE consists of and how they relate to each other. For this PyWinAuto provides the method *print_control_identifiers()*. This method prints all control elements of the application and their position. One element of the output of *print_control_identifiers()* is shown in Listing 4.2.

```
| SWT_Window0 - ''    (L1, T105, R3839, B2074)
| ['SWT_Window02']
| child_window(class_name="SWT_Window0")
```

**Listing 4.2:** One element of the output of *print_control_identifiers()*

The elements are structured hierarchically. That means there are parent elements and child elements. There are many parent elements that just group child elements, who have the actual functionality. For example in Listing 4.3 the parent element is a group element and the child elements are radio buttons and an edit field.

```
| SWT_GROUP - 'Keep components that ... '    (L906, T1964, R1863, B2063)
| ['SWT_GROUP2', 'Keep components that ... ', [...]]
| child_window(title="Keep components that ... ", class_name="SWT_GROUP")
|    |
|    | RadioButton - 'match exactly'    (L914, T1994, R1050, B2019)
|    | ['match exactlyRadioButton', 'RadioButton', [...]]
|    | child_window(title="match exactly", class_name="Button")
|    |
|    | RadioButton - 'start with'    (L1229, T1994, R1331, B2019)
|    | ['start with', 'start withRadioButton', 'RadioButton2', [...]]
|    | child_window(title="start with", class_name="Button")
|    |
|    | RadioButton - 'contain'    (L1544, T1994, R1631, B2019)
|    | ['containRadioButton', 'RadioButton3', [...]]
|    | child_window(title="contain", class_name="Button")
```

```
   |   |
   |   | Edit - ''    (L914, T2024, R1854, B2055)
   |   | ['Edit2', 'match exactlyEdit', 'match exactlyEdit0', [...]]
   |   | child_window(class_name="Edit")
```

**Listing 4.3:** Example of grouped elements

This structure could help the LLM to understand the GUI better, because this structure displays the relation between the elements. For example in Listing 4.3 the radio buttons are children of the group element with the title "Keep components that ...". Therefore, the radio buttons are options that the user can choose from, e.g. "Keep components that match exactly".

The *print_control_identifiers()* method delivers more than a million elements without even finishing. The reason for not finishing is that we aborted the process, because these are far too many elements to be processed with an 8k token context length. It is also possible that the method is not working correctly on the GUI of RCE. To test this we choose a different approach to get an overview of the elements.

PyWinAuto provides the method *children()* to get all child elements of a parent element. When manually iterating over the elements from the root element, we found that there are around 70,000 elements. For that reason it is very likely the method *print_control_identifiers()* is not working correctly on the GUI of RCE. Therefore, we use the *children()* method for further investigations.

70,000 elements are still too many elements to be processed within an 8k context window. Therefore, we get an overview of the elements and then filter out elements that are irrelevant for our use case. Getting an overview over so many elements is difficult. To make it easier we take a portion of the elements and visualize them. We do the visualization of the elements by drawing colored rectangles over the elements. We do this for all the child elements of the root element, which are a little over a hundred elements. The result can be seen in Figure 4.2.

The visualization shows that there are three noticeable problems with the current selection of elements. First, there are elements that are not visible. These elements do not have a surface area and are therefore not visible in the visualization, but

**Figure 4.2.:** Visualization of the GUI of RCE

appear in textual form using the *children()* method. Such elements for example are used as separators between two buttons. As they do not bring any additional information for the LLM we filter them out.

The second problem is that the elements are overlapping. Normally, elements that are overlapping are the children with their parent elements. For example, in Listing 4.3 the SWT_GROUP element has the position (L906, T1964, R1863, B2063). Every child element of the SWT_GROUP element has a position that is within this rectangle. For example, the 'match exactly' radio button has the position (L914, T1994, R1050, B2019), which indicates the distances from the left, top, right and bottom of the top-left corner of the screen. As the distance left and top distances of the button are bigger and the right and bottom distances of the button are smaller than the parent element, the 'match exactly' radio button is within the rectangle of the SWT_GROUP element. If the elements are on the same level, they should not overlap, for example all the radio buttons in Listing 4.3 do not overlap. However, in our visualization we only display all the children from the same parent element. Therefore, the elements should not overlap in the visualization. Further investigation shows that this unexpected behavior is caused by a behavior of the PyWinAuto library. The *children()* method does not only return the children of the parent element, but all descendant elements. This is especially unexpected, because there is a *descendants()* method that should return all descendant elements. Therefore, we have to filter out elements that appear multiple times.

The third problem is that there are elements that are not visible or enabled. These elements are not needed for the LLM and can also be filtered out. The complete filter process is shown in Figure 4.3.

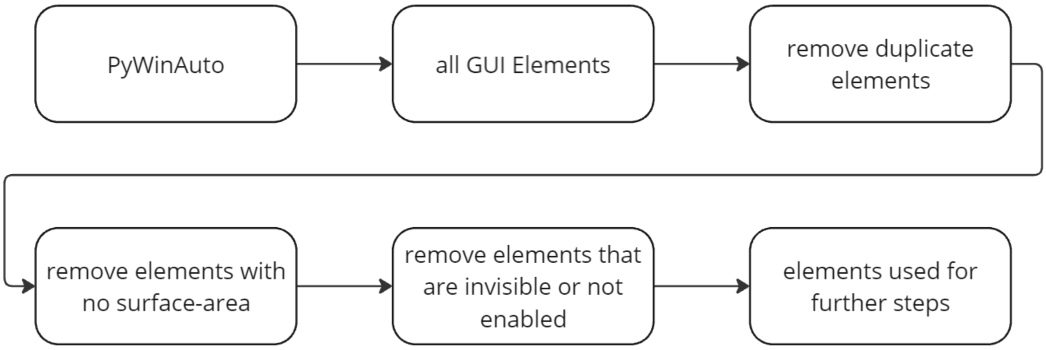



**Figure 4.3.:** Filter Process

After filtering out the unnecessary elements, we are left with just over a hundred elements for the entire GUI of RCE. Or for the first page of the Tool Integration Wizard, we are left with 21 elements. This is a manageable amount of elements that can be processed by the LLM.

### 4.2.3. Data for Each Control Element

There are different types of control elements in the GUI of RCE. Each control element has different information that is available. There is also information that is available for every control elements.

Every control element has a friendly class name, which is a human-readable name of its class. This name indicates the role of the control element. For example, a radio button has the friendly class name “RadioButton”, which indicates that it is part of a multiple choice option. Furthermore, every control element has a control type. This type indicates what actions can be performed on the control element. For example, an element with the “ButtonWrapper” control type can be clicked. In addition, every control element has a control ID. The LLM can use this ID to identify the control element. Another important information is the position of the control element. The position and size are represented as a rectangle which is stated in pixel distance from left, top, right and bottom side to the top-left of the screen. Lastly, every control element has a text attribute that represents the content of the control element. Sometimes this text is empty, for example for some control elements that group other control elements. In this case, the text attribute is left out.

On top of the information that is available for each control element, there is the following information available for specific control elements. The “RadioButton” control element has an additional attribute that indicates if it is checked. The “ComboBox” control element has an additional attribute that contains all items of the combo box. Each item has a text attribute that represents the content of the item, an index and an attribute that indicates if the item is selected. The “ListBox” works similar to the “ComboBox”, but its appearance is different and it allows multiple selections. For example, on the first page of the Tool Integration Wizard there is a “ListBox” that allows the user to select a template. This example is shown in Figure 4.4. The “ListBox” has the same attributes as the “ComboBox”. On top of that, it has an additional attribute that indicates if the “ListBox” allows multiple selections.

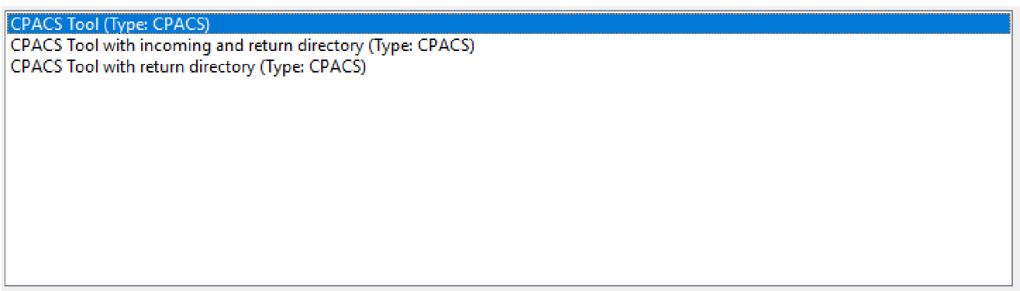
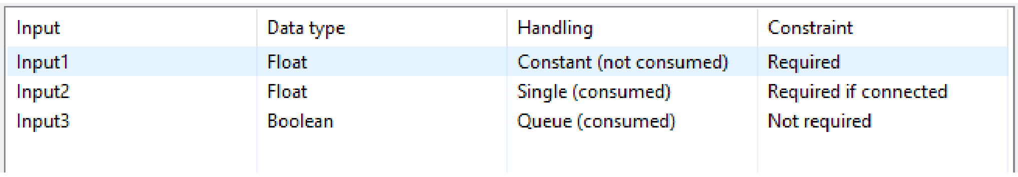



**Figure 4.4.:** Example of a ListBox

The “CheckBox” control element has an additional attribute that indicates if it is checked. The “ListView” control element has attributes that describe the content of a table that consists of rows and columns. It is also linked to a “Header” control element that has the column names as attributes. Therefore, the “ListView” has an additional attribute that contains the column names The “Header” control element is not needed, because it does not have any additional information. Additionally, the “ListView” has an additional attribute that contains the rows of the table. An example of a “ListView” is on the third page of the Tool Integration Wizard, which shows the inputs and outputs of the tool. It is shown in Figure 4.5.



| Input | Data type | Handling | Constraint |
|---|---|---|---|
| Input1 | Float | Constant (not consumed) | Required |
| Input2 | Float | Single (consumed) | Required if connected |
| Input3 | Boolean | Queue (consumed) | Not required |

**Figure 4.5.:** Example of a ListView

There is also a “Static” control element. Most of the time it is used to display text that is not editable. However, sometimes a “Static” control element contains an image. To parse an image into text, we need an image recognition tool. For this we use LLaVA, which is described in Subsection 4.2.4.

There are also control elements that have no additional attributes. For example the “Edit” control element has no additional attributes, because the content of the text field is already represented by the text attribute. Similarly, the “Button” and “SWT_GROUP” control elements are sufficiently represented by the text attribute.

An overview of the information that is available for each control element is shown in Table 4.1. The table shows the control element and the attribute that the element has.

### 4.2.4. Image Recognition with LLaVA

To complement the textual information of the control elements, we use LLaVA for image recognition. This is necessary because certain icons contain key information,

| Control Element | Attributes |
|---|---|
| All control elements | class_name, control_type, control_id, rectangle, text (if not empty) |
| RadioButton | check_state (unchecked, checked, indeterminate) |
| ListBox | single_selection, items (index, text, selected) |
| ComboBox | items (index, text, selected) |
| CheckBox | check_state (unchecked, checked, indeterminate) |
| Static | image_description (if image property exists) |
| ListView | columns, content (rows and columns) |

**Table 4.1.:** Information for each control element

for example an error icon indicates that something went wrong. The Large Language and Vision Assistant (LLaVA) [20] is capable of recognizing objects in images and translating them into text. The advantage of LLaVA is that it is open-source and therefore can be run locally. Its performance is noteworthy and is even comparable to GPT-4 vision in some areas. [9]

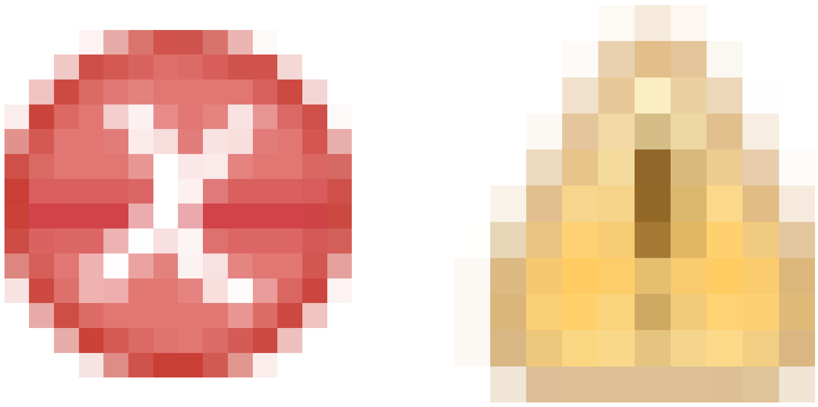
To use LLaVA, we need to provide a prompt along with the image. The image is extracted from the GUI of RCE by PyWinAuto. The prompt contains instructions for LLaVA on how to process the image. We want a very short description of the image. Therefore, we use the following prompt.

```
Describe the image precisely.
For example: "A green circle with a white checkmark in the middle. It represents
    that something works or is correct."
Never write more than two sentences.
```

According to DAIR.AI [5] when creating a prompt it is important to give a clear instruction on what the model should do. For example the instruction "Never write more than two sentences." gives the model a clear boundary of how long the description should be. Furthermore, they state that providing examples in the prompt is very effective to get desired output. Therefore, we provide an example in the prompt.

When testing LLaVa with example images (Figure 4.6), it provides the following results. The seed used for the test is 8.

```
Image 1: A red circle with a white X in the center, typically indicating that
    something is incorrect or not allowed.
```



(a) Example Image 1 (b) Example Image 2

**Figure 4.6.:** Example Images for LLaVA

```
Image 2: A yellow triangle with a black exclamation mark. It is commonly used to
    indicate caution or attention.
```

The results show that LLaVA is able to recognize the objects in the images and translate them into text.

## 4.3. Action-Parser

The action parser is responsible for extracting the possible actions for each control element. In Subsection 4.3.1 we first determine which actions needed to control the GUI of RCE. In Subsection 4.3.2 we then explain how we can extract the actions from the output of the LLM. In Subsection 4.3.3 we explain how the actions are executed and what feedback is provided.

### 4.3.1. Possible Actions

There are different actions that can be performed on the control elements of the GUI of RCE. The actions that are possible are determined by the control type of the control element. In the GUI of RCE there are the following control types.

- StaticWrapper
- HwndWrapper
- ButtonWrapper
- EditWrapper
- ListBoxWrapper
- ComboBoxWrapper
- CheckBoxWrapper

The StaticWrapper is a control element that either displays text that is not editable or an image. The HwndWrapper only groups other control elements and therefore does not have any actions that can be performed. The ButtonWrapper is a control element that can be clicked. For example a RadioButton is a ButtonWrapper and therefore can be clicked. The CheckBoxWrapper is a control element that can be checked or unchecked which is done by clicking. So for all the control elements that can be clicked, we need the action "click(<control_id>)". The EditWrapper is a control element that can be edited. So we need the action "write(<control_id>, <text>)". This action writes the text into the text field of the EditWrapper. The text that has been in the text field before is overwritten. A user of RCE could also select text or insert text at a specific position, but the write action is sufficient to cover all these cases. The ListBoxWrapper and the ComboBoxWrapper both have elements that can be selected. Every item in the ListBoxWrapper and ComboBoxWrapper has an index. Therefore, we have the action "select(<control_id>, <index>)".

With these three actions it is possible to control the GUI of the Tool Integration Wizard of RCE. Therefore, we do not need to implement additional actions. The different actions for each control type are shown in Table 4.2.

### 4.3.2. Forcing the Format of an Output

We give the LLM the possible actions that can be performed on a control element in the prompt. To be able to extract the action we ask the LLM to output the action

| Control Type | Actions |
|---|---|
| StaticWrapper | - |
| HwndWrapper | - |
| ButtonWrapper | click(<control_id>) |
| EditWrapper | write(<control_id>, <text_to_write>) |
| ListBoxWrapper | select(<control_id>, <index>) |
| ComboBoxWrapper | select(<control_id>, <index>) |
| CheckBoxWrapper | click(<control_id>) |

**Table 4.2.:** Actions for each control type

in a JSON format, so we can easily parse it. In this JSON format we also add an element for a possible explanation. So we have the following format that we ask the LLM to use in the prompt.

```
{
    "action": "<action>",
    "explanation": "<explanation>"
}
```

However, the LLM still includes additional text before and after the JSON object that is not needed. For example, using the prompt in Listing A.1 on page 65 the Llama 3.1:8b_Q4_0 Model with the seed 1 gives the following output.

```
    Based on the given task and the current state of the GUI, I would like to take
        the following action:

{
    "action": "write(68650, 'C:\\Users\\rose_ti\\Documents\\RCE_Tool\\calc_volumen.
        py')",
    "explanation": "Enter the path to the Python script in the Icon Path field.
        This is required for integrating the tool as a workflow component."
}

I choose this action because the Icon Path field requires a valid file path, and
    the task requires me to integrate a Python script. The path I entered should be
     correct according to the documentation provided.

Please execute this action and provide feedback about its result.
```

This makes it difficult to parse the output. Therefore, we need to force the format of the output. This is done by limiting the output of the LLM to the JSON syntax by using a grammar rule. The grammar rule limits the tokens that the LLM can output to those that are included in the JSON syntax. For some LLMs like Llama there are already grammar rules for JSON format available. [16] There are also software like Ollama [23] that already have implemented the grammar rules for JSON format. OpenAI also provides an option for their Application Programming Interface (API) to use grammar rules for JSON format. [24]

So when we run the same prompt and configuration, but with the JSON format option, we get the following output.

```
{
    "action": "write(68646, 'calc_volumen')",
    "explanation": "Enter the name of the tool in the text field, as this is a
        required field and it should match the script's path."
}
```

This output is now in the desired format and can be easily parsed. Because the action is also in the predefined format, we can extract the action with its parameters as well.

### 4.3.3. Executing the Actions

With the predefined actions we can extract the *control_id* and the other parameters text and index. We then use PyWinAuto and the *control_id* to find the referenced control element. Depending on the action we then call the corresponding method on the control element. For example, if the action is "click(68646)" we call the *click()* method on the control element with the *control_id* 68646. For the "write" action we use the *set_edit_text()* method with the text. For the "select" action we use the *select()* method with the index.

We then save the action with the explanation of the LLM for future iterations. Additionally, we save a status parameter in the saved action that indicates whether the action was executed successfully or not. We check this by validating the action and catching possible errors that can occur during the execution of the corresponding method. The validator checks if the action matches with “click”, “write” or “select” and if the action is called on the right type of control element. For example, if the action “click” is called on an “Edit” control element, the validation fails. If the validation fails, we set the status to “not executed” and save the reason “Element with control_id <control_id> is a <control_type> which has no action <action_name>” in the “error” parameter. If the validation is positive the action executor calls the corresponding method on the control element. If there were no errors during the execution, the status is set to “executed”. Otherwise, the status is set to “not executed” and a reason for the error is saved in the “error” parameter. Depending on the error, we set different reasons for why the action was not executed.

The first error that can occur is the “ElementNotFoundError” error. This error occurs when the control element with the control_id does not exist. So we set the reason to “Element with control_id <control_id> does not exist”. The second error that can occur is the “ElementNotEnabled” error. This error occurs when the control element is not enabled. So we set the reason to “Element with control_id <control_id> is not enabled”. The last error that can occur is the “IndexError” error. This error occurs when the index is out of bounds. For example if a “ComboBox” has only two items and the “select” action is called with index 3. In this case we set the reason to “Element with control_id <control_id> has no item with index <index>”.

An overview over reasons for the not execution of the actions and the error message that is saved are shown in Table 4.3.

Three examples of saved actions are shown in Listing 4.4. The examples include actions with and without errors. The actions are not related to each other and only serve as examples.

```
{
    "action": "click(592972)",
    "explanation": "Click on the 'Integrate a Tool' button to proceed with
        integrating the Python script.",
    "status": "not executed",
```

| Reason for Failure | Error message |
|---|---|
| Action does not match with “click”, “write” or “select” or action is called on wrong type of control element | Element with control_id <control_id> is a <control_type> which has no action <action_name> |
| “ElementNotFoundError” error | Element with control_id <control_id> does not exist |
| “ElementNotEnabled” error | Element with control_id <control_id> is not enabled |
| “IndexError” error | Element with control_id <control_id> has no item with index <index> |

**Table 4.3.:** Reasons for errors and corresponding error messages

```
    "error": "Element with control_id 592972 is a HwndWrapper which has no
        action click"
}

{
    "action": "click(396374)",
    "explanation": "Click on the Next button to proceed with defining the name
         of the tool, as this is a required field.",
    "status": "not executed",
    "error": "Element with control_id 396374 is not enabled"
}

{
    "action": "click(200618)",
    "explanation": "Click on the '...' button to select a path for the icon of
         the tool, as this is required field",
    "status": "executed"
}
```

**Listing 4.4:** examples of saved actions

## 4.4. Prompt Engineering

We now have every component to create a prompt for the LLM. First we need to decide on the prompting techniques we want to use. We decide if each of the prompting techniques mentioned in Section 2.3 is suitable for our use case.

For our prompt we are limited to about 8k tokens, because at the time of writing this thesis, the typical context window of most LLMs (e.g. Llama 3 and ChatGPT-4) is 8192 tokens. Applying the few-shot prompt technique would increase the length of the prompt by a factor of the number of examples. The length of the parsed GUI multiplied by two would already exceed the token limit. Therefore, the few-shot prompt technique is not suitable for our use case. At least it is not useable in the way of using full examples. What is possible is to give examples for output prompts. Nevertheless, the basis of the prompt has to be a zero-shot prompt.

The chain of thought prompt technique can be used to improve the prompt. Concretely, it can be used to provide the examples for the outputs with reasoning steps. This encourages the LLM to also utilize a step-by-step reasoning process to solve the task.

After weighing which of the prompt techniques are most suitable, we have to decide on the structure of the prompt. There is common structure that is used for prompts. [32] This structure is shown in Figure 4.7.

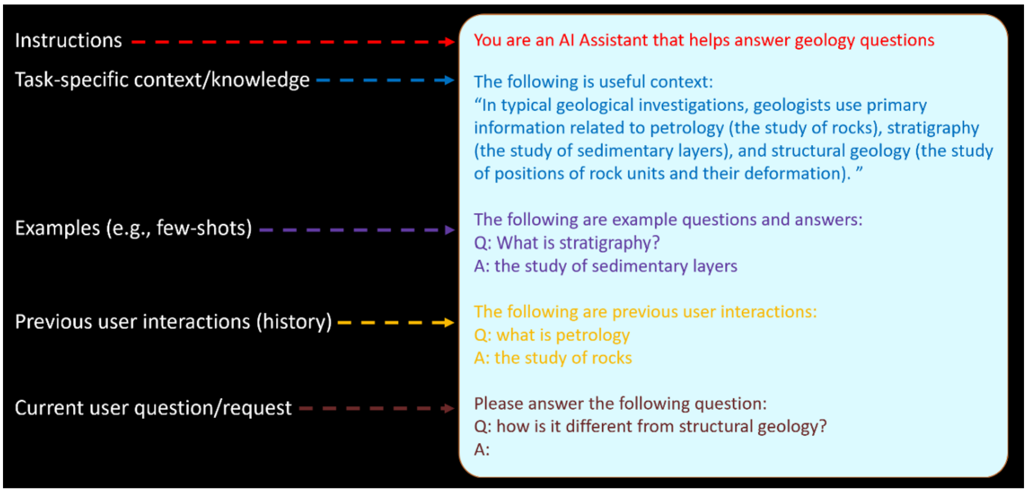



**Figure 4.7.:** Common structure used for prompts [32]

The first part consists of general instructions and other static context. For our use case this is the general description of the task or rather the role of the LLM. We want to make the LLM understand that it acts as the controller of the GUI of RCE and has to solve a task. Furthermore, we want to explain how the system works and how the LLM can interact with it. In addition, we provide an overview over the information that is available for the LLM. So considering these points, the first part of the prompt is shown in Listing 4.5.

```
You are an automated system that controls the software RCE.
You are given a task, the GUI of RCE and documentation about the software in
    textual form.
You have to interact with the GUI to achieve the given task.
You can control the GUI by sending actions to the software.
The software will execute the actions and give you feedback about the result, by
    sending you the new state of the GUI and whether the action was successful or
    not.
You must use the information about the GUI and the documentation to decide which
    actions to take.
Include the information about the position of the GUI-Elements and the text of the
    GUI-Elements in your decision.
Also consider which Parent-Elements the GUI-Elements have.
It is important to take the context of the GUI-Elements into account when deciding
    which actions to take
You can also use the feedback about the result of the actions to decide which
    actions to take next.
```

**Listing 4.5:** Instruction

The second part consists of task specific context and knowledge. For our use case this is the task that the LLM has to solve, the documentation of the software, the parsed GUI and the possible actions that the LLM can take. The task is given by the user. It consists of a detailed description of the external tool that should be integrated into RCE. This detailed description includes where the tool is located, what the tool does and how the tool is used. It should also include which parameters the tools has and how the tool handles the result. Other information like the name and E-Mail of the user can also be included. The documentation of the software informs the LLM about the technical functionality of RCE. The parsed GUI informs the LLM about the structure of the GUI of RCE. We explained its composition in Section 4.2. Additionally, we provide a description of different types of control elements. We also determined the possible actions that the LLM can take in Subsection 4.3.1. For each of the possible action we also provide an example. The list of the possible actions and the examples are shown in Listing 4.6.

```
To control a GUI-Element output a command in the following format:
<action>(<control_id>), for example click(134478)
For each control type there are different actions posible.
The StaticWrapper and HwndWrapper have no actions.
The ButtonWrapper has the click(<control_id>) action, for example click(134478)
The EditWrapper has the write(<control_id, <text to insert>) action, for example
    write(134456, "example text")
The ListBoxWrapper has the select(<control_id>, <index>) action, for example select
    (134456, 1)
The ComboBoxWrapper has the select(<control_id>, <index>) action, for example
    select(134456, 1)
The CheckBoxWrapper has the click(<control_id>) action, for example click(134456)
```

**Listing 4.6:** Possible actions with examples

The third part consists of Examples. For our use case these are the example outputs that include the reasoning steps. The examples are shown in Listing 4.7.

```
You must format your output in JSON as the following:
{
    "action": "<action>",
    "explanation": "<what the action does and why you do it>"
}
example 1:
{
    "action": "click(134478)",
    "explanation": "click next to get to the second page"
}
example 2:
{
    "action": "write(134456, 'Airresistenz Calculator')",
    "explanation": "Enter the name in the text field, as this is a required
        field"
}
example 3:
{
    "action": "select(134456, 1)",
    "explanation": "Select float as the data type, because the input 'material
        's coefficient' is a float"
}
```

**Listing 4.7:** Examples actions of the prompt

The fourth part consists of a list of the previous actions. This includes the action command and the description of the action.

Lastly, the fifth part is the user question or request. This is the final question which asks the LLM to decide on an action. The question is shown in Listing 4.8.

```
What action do you want to take to do the next step for achieving the given task?
```

**Listing 4.8:** Final question

A complete example prompt is shown in Listing A. For the ChatGPT-4 model it would have a context length of 4,019 tokens. [26] This is within the limit of 8k tokens and still has enough space when the LLM performs additional actions which are saved in the prompt.

# 4.5. Pitfalls and Lessons Learned

Now that we have the complete system, we perform first tests and look for abnormalities. We conduct the first tests with the Llama 3 and ChatGPT4-o model. The system works as expected and the LLM is able to control the GUI of RCE. However, there are some pitfalls that we encountered, which we will describe in the following.

## 4.5.1. Non editable Edit Control Element can be Edited

During the first tests we encountered a problem with one specific "EditWrapper" control element. The specific control element is marked in Figure 4.8.

The purpose of this control element is to display an error or warning message, which can change during the interaction with the GUI. Although the control element has the control type "EditWrapper", it is not editable for the user, by using mouse or keyboard. For the LLM, however, it is possible to call the "write" action on this control element, which changes the text of the control element. This enabled the LLM to overwrite the error or warning message. This is problematic because an actual user could not do this and the error or warning message is important for the LLM to understand what went wrong.

To prevent this, we would have to check if the control element is editable before executing the "write" action. However, the "Win32" backend of PyWinAuto does not

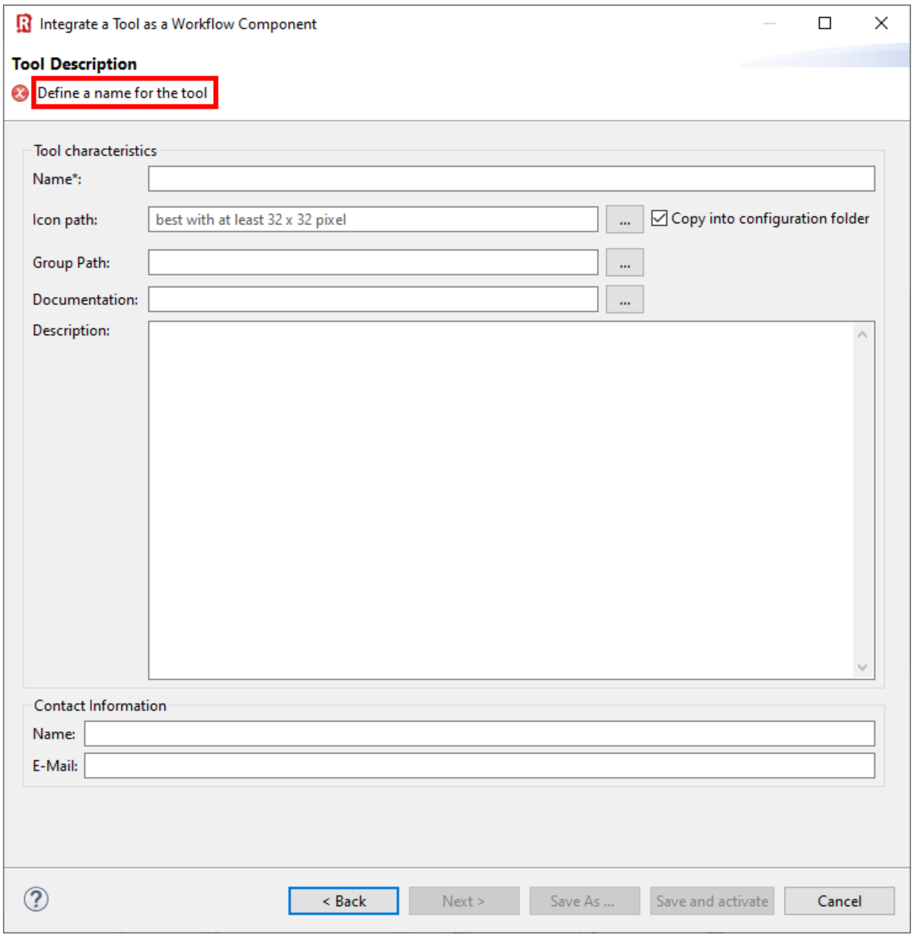



**Figure 4.8.:** Specific EditWrapper control element

provide a method to check if a control element is editable. Only the "UIA" backend provides such a method. However, switching to the UIA backend is out of scope for this thesis, as explained above. Therefore, we solve the problem by changing the control type of the control element to "StaticWrapper".

We achieve this by detecting the "control_id" of the control element at the start of the program and save it. It can be detected because on the first page of the Tool Integration Wizard (Figure 2.2(a) on page 11) the text is always "Create a new tool configuration or choose an inactive one to activate". The GUI parser then detects the control element and changes the control type to "StaticWrapper". Also, when executing the action, the action parser checks if the control element has the "control_id" of the specific control element and treats it as a "StaticWrapper" control element. This way we prevent the LLM from overwriting the error or warning message.

### 4.5.2. Limitations of the GUI Parser

We also encountered some other limitations of the GUI parser. On the third (Figure 2.2(c)) and sixth page (Figure 2.2(f)) of the Tool Integration Wizard, there are tab elements, for example the "Inputs", "Outputs" and "Verification" tab. These tabs are all represented by the same control element with the control type "HwndWrapper". Therefore, the GUI parser cannot distinguish between the different tabs and offer actions to select them. This is problematic because the LLM cannot select the other tabs except the first one.

Another limitation is the text field on the sixth page (Figure 2.2(f)), where the user can enter the start command. This text field is not an "EditWrapper" control element, but again a "HwndWrapper" control element. Therefore, the LLM cannot write into this text field.

With the "UIA" it might be possible to distinguish between the different tabs and write into the text field, because the "UIA" backend provides different control types. However, testing this is out of scope for this work, as explained above. Therefore, we have to live with this limitation and consider it in our evaluation.

### 4.5.3. Give Documentation Piece by Piece

During the first tests we also encountered a problem with the documentation. We have given the documentation of the Tool Integration Wizard to the LLM in the prompt. The LLM uses the documentation to decide which actions to take, but sometimes it uses wrong parts of the documentation. For example, on the first page of the Tool Integration Wizard the LLM has to select what kind of tool it wants to integrate. However, it uses the documentation of the last page of the Tool Integration Wizard, regarding the Tool run imitation script. This leads to the problem that the

LLM often wants to perform actions that cannot be executed on the first page. To prevent this, we give the documentation piece by piece. That means we write only the part of the documentation in the prompt, that is relevant for the current page. So if the LLM presses the "Next" or "Back" button, we provide the next or the previous part of the documentation. This way the LLM can only use the relevant part of the documentation and does not get confused by the other parts.

# 5. Evaluation

Now that we have implemented the system, we evaluate it. For this we use a test case, which we describe in Section 5.1. We evaluate our system using metrics that measure the LLM's ability to perform required actions, its accuracy in executing correct and incorrect actions, and its overall efficiency in completing the task, as detailed in Section 5.2. Because our system heavily relies on the LLM, we evaluate it with different LLMs, which we describe in Section 5.3. Finally, we present the results in Section 5.4 and discuss them in Section 5.5.

## 5.1. Test Case

For our test case we use a simple example of an external tool that can be integrated into RCE. The tool is a python script that can be executed from the command line with the following command:

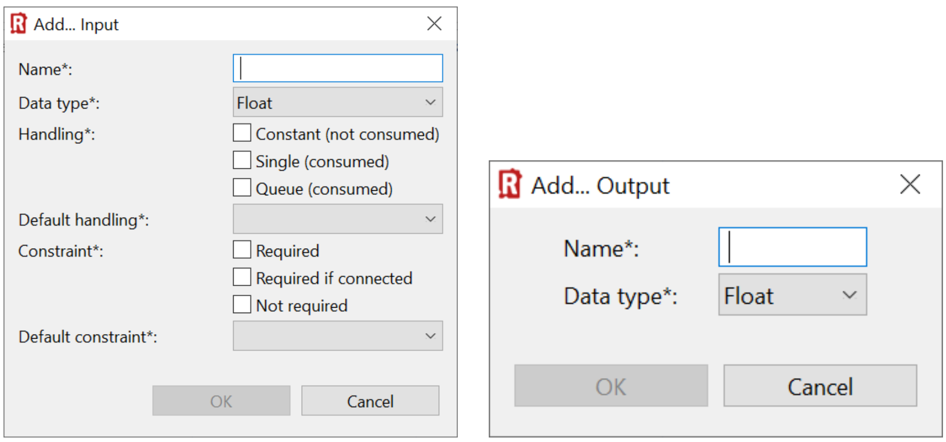
```
python volume.py --length <number> --width <number> --height <number>
```

The command is executed with three arguments, which are all floating-point numbers. The script calculates the volume of a cuboid with the given side lengths. The result is written into a file called `result.txt`.

To integrate this tool into RCE, certain steps are necessary on each page of the Tool Integration Wizard. On the first page of the Tool Integration Wizard, the user does not need to enter any information, because the option "Create a new Commmon tool configuration" is selected by default. Therefore, the user only has to click on the "Next" button. So only one action is needed on this page.

On the second page, the user has to enter the name of the tool and click on the second page. These are two actions.

On the third page, the user has to add three inputs and one output. For adding one input, the user has to click on the "Add" button, which opens a wizard for adding an input (see Figure 5.1(a)). The correct data type is already selected. So the user only has to enter the name of the input, the input handling and the execution constraint. Any of the handling and constraint options can be selected, as we do not specify the context of how the tool will be used in RCE. After that, the user has to click on the "OK" button. This has to be done three times, each time needing five actions, so 15 actions in total. For adding the output, the user has to click on the "Output" tab and then on the "Add" button, which opens a wizard for adding an output (see Figure 5.1(b)). The user has to enter the name of the output and then press "OK". This is four actions. After that, the user has to click on the "Next" button, which is another action. So in total, 20 actions are needed on this page.




(a) Wizard for adding an input (b) Wizard for adding an output

**Figure 5.1.:** Wizards for adding an input and an output

As properties are not needed for this tool, the user can click on the "Next" button on the fourth page. So only one action is needed on this page.

On the fifth page, the user has to add the launch settings. The user has to click on the "Add" button, which opens a wizard for adding a launch setting. (see Figure 5.2) There they have to enter the path to the tool-directory and the working directory and a version for the tool integration. After that, the user has to click on the “OK” button, which is 5 actions. Then the user has to set a clean-up option, for example, “Delete the working directory after execution”. This is another action. After that, the user has to click on the “Next” button, which is one action. So in total, seven actions are needed on this page.

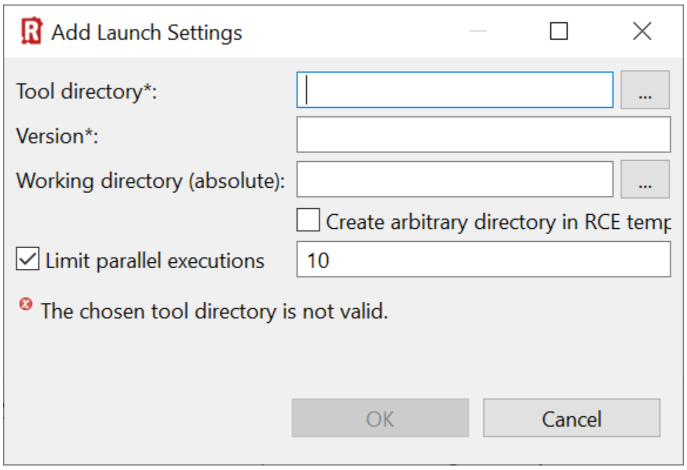



**Figure 5.2.:** Wizard for adding a launch setting

On the last page, the user has to add a start command. For this they need to click the “Command(s) for Windows” checkbox, enter the following command.

```
python ${dir:tool}/calc_volumen.py --length ${in:length} --width ${in:width}
    --height ${in:height}
```

In this example the inputs are called `length`, `width` and `height`. If the user wants to use different names, they have to change the command accordingly. Additionally, the user has to add a post-execution script, which passes the result from the file to the output. For this the user has to switch to the “Post-execution” tab and enter a script that reads the result from the file and writes it to the output. For example, the following script would work.

```
f = open("${dir:tool}/result.txt", "r")
${out:volume} = float(f.read())
```

If the user named the output differently than “output”, they have to change the command accordingly. After that, the user can click on the “Save and activate” button. So in total, five actions are needed on this page.

In total, 36 actions are needed to integrate the tool into RCE. There are other optional actions, that are useful for the user. For example, the user can add a description of the tool and contact information of the author.

## 5.2. Measured Metrics

The task for the LLM is to perform these actions and integrate the tool into RCE. Because of the limitations of the GUI parser mentioned in Subsection 4.5.2, the LLM cannot perform all of these actions. In detail the LLM is not able to switch to the “Output” or “Post-execution” tab. Therefore, it cannot perform these actions and also cannot add an output or the post-execution script. So the LLM can only perform 31 of the 36 required actions.

The primary metric we measure is how many of the 31 required actions the LLM can perform. This metric is important because it shows how much of the integration process the LLM could automate. We also measure how many correct actions that are not required the LLM can perform. These are actions that bring a benefit but are not mandatory to complete the integration. It is also important how many wrong actions the LLM performs. Wrong actions are actions that hinder the integration process, e.g. enter a not valid path into an edit field, which causes an error. This is divided into two categories, actions that could not be executed, e.g. the control_id does not exist, and actions that were executed but were wrong. The rest of the action fall into the category of actions that have no effect, for example if the LLM enters the same text into a text field that is already there.

We also measure the number of actions the LLM performs. The system stops the integration process if the LLM either finishes the integration or cancels it by pressing the “Cancel” button. As the LLM might get stuck in a loop, we also limit the amount of actions the LLM can perform to 80.

## 5.3. Selection of LLMs

For the selection of the LLMs we want to consider the following criteria:

- How effective is the LLM?
- How good is the performance of the LLM?
- How much does it cost to use the LLM?
- How much control do we have over the LLM?

The effectiveness of the LLM is important because we want the LLM to integrate the tool into RCE as well as possible. We also want the LLM to perform these actions in a reasonable amount of time. Therefore, the performance of the LLM is also important. The cost of the LLM is also important, especially for future developments where the LLM is used more often. The cost depends on the business model of the provider of the LLM. This changes most likely in the future, so it is the least consistent and therefore least important criterion. For a future application of the LLM in RCE, it is also important how much control we have over the LLM. Because some users might not want their information to be processed by a third party.

Gao et al. used Llama2, ChatGPT-3.5 and ChatGPT-4 in their study to control the software. [6] They found that of these three LLMs, only ChatGPT-4 delivered good results. Therefore, Llama2 and ChatGPT-3.5 are out of consideration for our evaluation. ChatGPT-4 could be a good choice, but there have been new developments of OpenAI, since the release of their work, which resulted in the ChatGPT-4o model. ChatGPT-4o is compared to ChatGPT-4 more effective. This is shown in the benchmarks, where ChatGPT-4o outperforms ChatGPT-4 in most cases. [10] It is also more performant, which allows for near instant responses. [27] On top of that, ChatGPT-4o is much cheaper. It costs only one sixth of the price of ChatGPT-4. [28] The only downside of ChatGPT-4o is that it is a cloud-based service, which means that we have less control over the LLM. However, the advantages of ChatGPT-4o outweigh this downside, and we therefore choose ChatGPT-4o as one of the LLMs for our evaluation.

If we want to have more control over the LLM, we have to use a local LLM. The Llama2 model is a local LLM, but as mentioned before, it is not suitable for our evaluation. However, Meta recently released their Llama3.1 model, which is much more powerful than their Llama2 model. [3] It even outperforms ChatGPT-4 in some benchmarks. [13]

There are multiple sizes of the Llama3.1 model, which differ in the amount of parameters. The more parameters the model has, the more powerful it is, but it also needs more resources to run. If the model fits into the GPU's VRAM, the model can run on the GPU at a very high speed. If the model does not fit into the GPU's VRAM, the computer also uses the RAM, which is much slower. The different sizes of the Llama3.1 model are shown in Table 5.1.

| **Model** | **Parameters** | **Size** |
|---|---|---|
| Llama3.1:8b [17] | 8.03B | 4.7GB |
| Llama3.1:70b [19] | 70.6B | 40GB |
| Llama3.1:405b [18] | 406B | 229GB |

**Table 5.1.:** The different sizes of the Llama3.1 model with the quantization of Q4_0

The model are quantized, which means that they are compressed, so that they need less resources to run, but still have a high performance. For this thesis we have a computer with the specifications shown in Table 5.2.

| **Component** | **Specification** |
|---|---|
| CPU | Intel Core i9-10900KF |
| RAM | 64 GB |
| GPU | NVIDIA GeForce RTX 3090 |

**Table 5.2.:** The specifications of the computer used for this thesis

It has an NVIDIA GeForce RTX 3090, which has 24 GB of VRAM. As the Llama3.1:8b model fits into the VRAM it runs at a very high speed. The Llama3.1:70b model does not fit into the VRAM, but it fits into the RAM. It is better than the Llama3.1:8b model, but it is slower. The Llama3.1:405b model does not fit into the RAM, so it cannot run at all. Therefore, we also choose the Llama3.1:8b and Llama3.1:70b model for our evaluation.

So for our evaluation we use the ChatGPT-4o, Llama3.1:8b and Llama3.1:70b model. The ChatGPT-4o model is effective and performant, but costs per usage and we have less control over it. The Llama3.1:8b model is also performant and cost nothing to use and we have more control over it. The Llama3.1:70b model is effective, but slower than the other models and also costs nothing to use and we have more control over it. Their exact designations are the following:

- gpt-4o-2024-05-13
- Llama3.1:8b-Q4_0
- Llama3.1:70b-Q4_0

To access the ChatGPT-4o model we use the OpenAI API. For the Llama3.1:8b and Llama3.1:70b model we use Ollama to run these locally. [23] It is easy to set up and run the Llama models with it. For all models we use the default configuration except for the seed and the format option. The seed is set to different predefined values for each run. The format option is set to “JSON”. The possible context length of the available models have increased since the start of this thesis. For example, the Llama3 model has a context length of 8k, but the Llama3.1 model has a context length of 128K. Even though we planned our system with a context length of 8k, we set the context length to 16k for the Llama3.1:8b and Llama3.1:70b models, to guarantee that the prompt fits into the context length. The context length of the ChatGPT-4o model defaults to 128k tokens.

## 5.4. Results

We run each LLM 10 times and measure the metrics described in Section 5.2. To make the results as reproducible as possible, we use predefined seeds for each run. We also use the same seed (8) for the image recognition model, mentioned in Subsection 4.2.4, for each run. However, there are some random factors in the GUI parser that we have no control over. For example the *control_ids* are assigned by the operating system and can change from run to run. Therefore, we cannot guarantee that the results are reproducible.

### 5.4.1. Results of the Llama3.1:8b model

The results of each test run of the Llama3.1:8b model are shown in Table 5.3.

| **Seed** | **Correct Actions (required)** | **Correct Actions (not required)** | **No Effect Actions** | **Wrong Actions** | **Actions that cannot be executed** | **Total Number of Actions** | **Time in s** | **Time/Action in s** |
|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 73 | 1 | 5 | 80 | 295 | 3.69 |
| 8 | 2 | 4 | 47 | 0 | 27 | 80 | 317 | 3.96 |
| 42 | 2 | 4 | 70 | 0 | 4 | 80 | 289 | 3.61 |
| 73 | 2 | 2 | 8 | 2 | 66 | 80 | 348 | 4.35 |
| 100 | 0 | 1 | 0 | 0 | 79 | 80 | 315 | 3.94 |
| 63 | 2 | 1 | 67 | 0 | 10 | 80 | 322 | 4.03 |
| 48 | 0 | 0 | 57 | 1 | 22 | 80 | 278 | 3.48 |
| 1001 | 2 | 1 | 66 | 0 | 11 | 80 | 268 | 3.35 |
| 765456 | 0 | 0 | 0 | 1 | 0 | 1 | 22 | 22.00 |
| **Avg** | 1.3 | 1.4 | 46 | 0.6 | 22.8 | 72.1 | 274.7 | 5.61 |
| **SD** | 0.9 | 1.428 | 29.36 | 0.663 | 26.248 | 23.7 | 87.114 | 5.47 |

**Table 5.3.:** The results of the Llama3.1:8b model

It shows that the Llama3.1:8b model is only able to perform up to two of the required actions. This means that it is only able to navigate to the second page of the Tool

Integration Wizard and enter the name of the tool. Sometimes it was able to enter some optional information, such as the path to the icon or the documentation. Most times it ended up in a loop, where it performed an action that had no effect or cannot be executed. That is why either the no effect actions or the actions that cannot be executed are the highest in most runs. The run with the seed 765456 ended very fast after the first action, because the LLM instantly canceled the integration process. On the positive side, the Llama3.1:8b model is very fast, with an average time of 274.7 seconds.

All in all, the Llama3.1:8b model is not able to complete the tool integration in even a rudimentary way. It can barely enter the basic information of the tool and then gets stuck in a loop. Even this is not guaranteed, because often the LLM did no correct actions at all.

### 5.4.2. Results of the Llama3.1:70b model

The results of each test run of the Llama3.1:70b model are shown in Table 5.4.

The Llama3.1:70b model is able to perform on average 4.9 of the 31 required actions. Most times it is able to enter the general information of the tool on the second page. On top of the required name of the tool, it also enters most of the optional information on this page. Sometimes it was also able to navigate to the third page and start to add the inputs. However, it was never able to add all the information of the inputs. Most times it entered a name of the input, but did not select the input handling or the execution constraint. It then canceled the adding of the inputs and therefore did not finish the adding of the inputs. After some tries of adding the inputs in most cases the Llama3.1:70b model ended up in a loop, where it performed an action that had no effect or cannot be executed. Sometimes it even canceled the whole integration process. Only in rare cases it continues to the fifth page, where it tries to add the launch settings, but it never finishes it.

All in all, the Llama3.1:70b model is also not able to complete the tool integration. On top of that, the Llama3.1:70b model is very slow, with an average execution time of about 55 minutes and an average time per action of 88.16 seconds.

| Seed | Correct Actions (required) | Correct Actions (not required) | No Effect Actions | Wrong Actions | Actions that cannot be executed | Total Number of Actions | Time in s | Time/Action in s |
|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 5 | 0 | 2 | 1 | 10 | 873 | 87.30 |
| 8 | 2 | 4 | 4 | 2 | 2 | 14 | 1214 | 86.71 |
| 42 | 0 | 0 | 1 | 2 | 0 | 3 | 292 | 97.33 |
| 73 | 6 | 1 | 5 | 1 | 3 | 16 | 1328 | 83.00 |
| 100 | 0 | 0 | 1 | 2 | 0 | 3 | 249 | 83.00 |
| 63 | 9 | 5 | 10 | 14 | 42 | 80 | 6258 | 78.23 |
| 48 | 7 | 8 | 1 | 6 | 58 | 80 | 6784 | 84.80 |
| 1001 | 2 | 6 | 0 | 1 | 0 | 9 | 823 | 91.44 |
| 765456 | 9 | 5 | 35 | 7 | 24 | 80 | 7942 | 99.28 |
| **Avg** | 4.9 | 4 | 6.3 | 4 | 18.3 | 37.5 | 3300.6 | 88.16 |
| **SD** | 4.036 | 2.608 | 10.04 | 3.847 | 22.755 | 34.912 | 3108.153 | 6.25 |

**Table 5.4.:** The results of the Llama3.1:70b model

### 5.4.3. Results of the ChatGPT-4o model

The results of each test run of the ChatGPT-4o model are shown in Table 5.5.

The ChatGPT-4o model is able to perform on average 23.2 of the 31 required actions. This is more than two thirds of the required actions. Sometimes it is even able to almost complete all the required actions, e.g. with the seed 8 or 1001, where it performs 30 or 28 of the 31 required actions. The first two pages of the Tool Integration Wizard are no problem for the ChatGPT-4o model. It is always able to enter a name for the tool and most times add all the optional information. The third page is most times

| Seed | Correct Actions (required) | Correct Actions (not required) | No Effect Actions | Wrong Actions | Actions that cannot be executed | Total Number of Actions | Time in s | Time/Action in s |
|---|---|---|---|---|---|---|---|---|
| 1 | 14 | 4 | 7 | 9 | 46 | 80 | 832 | 10.40 |
| 8 | 30 | 8 | 3 | 3 | 3 | 47 | 567 | 12.06 |
| 42 | 26 | 10 | 7 | 2 | 6 | 51 | 540 | 10.59 |
| 73 | 27 | 10 | 6 | 2 | 5 | 50 | 337 | 6.74 |
| 100 | 20 | 10 | 17 | 7 | 6 | 60 | 471 | 7.85 |
| 63 | 22 | 12 | 15 | 4 | 27 | 80 | 735 | 9.19 |
| 48 | 22 | 10 | 4 | 4 | 14 | 54 | 428 | 7.93 |
| 1001 | 28 | 6 | 3 | 1 | 6 | 44 | 307 | 6.98 |
| 765456 | 22 | 10 | 1 | 5 | 12 | 50 | 346 | 6.92 |
| **Avg** | 23.2 | 9.1 | 6.9 | 4.2 | 13 | 56.4 | 493.5 | 8.64 |
| **SD** | 4.423 | 2.3 | 4.928 | 2.315 | 12.892 | 12.476 | 167.648 | 1.74 |

**Table 5.5.:** The results of the ChatGPT-4o model

also no problem. Most times ChatGPT-4o tries to add the inputs, but it is not always able to finish it. Similar to the Llama3.1:70b model, it often does not select the input handling or the execution constraint. However, this does not stop the ChatGPT-4o model from continuing the integration process. It always continues to the next page, where it is supposed to not add any properties. However, it sometimes tries to add properties. Sometimes the property is the path to the result file, which is not required but also a correct action. Other times it tries to add a property but cancels it after some tries. So it wastes some actions on this page, but then continues to the next page. The fifth page is never a problem for the ChatGPT-4o model. It always adds the launch settings, with the Tool and the working directory and the version number. The version was always 1.0. The only problem with the launch settings is that the ChatGPT-4o model often wrote the path to the tool into the Tool directory field. So it wrote “C:\Users\<user>\Documents\RCE_Tool\calc_volumen.py” instead of “C:\Users\<user>\Documents\RCE_Tool” into the Tool directory field. It also always selected a clean-up option, most time it was “Delete the working directory after execution”. After that it always continued to the last page. There it always selected the “Command(s) for Windows” checkbox. After that it was sometimes able to write the correct start command, but other times there were some mistakes, e.g. it wrote “calc_volumen.py -height ${in:height} -width ${in:width} -length ${in:length}” instead of “python ${dir:tool}/calc_volumen.py -length ${in:length} -width ${in:width} -height ${in:height}”. The problem with the start command is that the ChatGPT-4o model did not write “python” at the beginning of the command.

All in all, the ChatGPT-4o model is very close to completing the tool integration. Since certain required actions are always performed correctly, ChatGPT-4o is reliable to some extent. The only downside are the costs of using the ChatGPT-4o model. To incorporate ChatGPT-4o into our system we have to use OpenAI’s API, which costs per usage. OpenAI charges $5 per 1M input and $15 per 1M output tokens. [28] For this evaluation the costs of using the ChatGPT-4o API are about 20$. That means one run costs about 2$. In addition, OpenAI has a rate limit of 30k tokens per minute. This rate limit is set very high but still slows down our system. So with an average execution time of 493.5 seconds and an average time per action of 8.64s it is still quite fast.

### 5.4.4. Comparison

To compare the three LLMs in more detail we create boxplots for each metric, which are shown in Figure 5.3 through Figure 5.8.

As we can see in Figure 5.3, the ChatGPT-4o model outperforms the other two models by far regarding the number of correct actions that are required. The minimum of

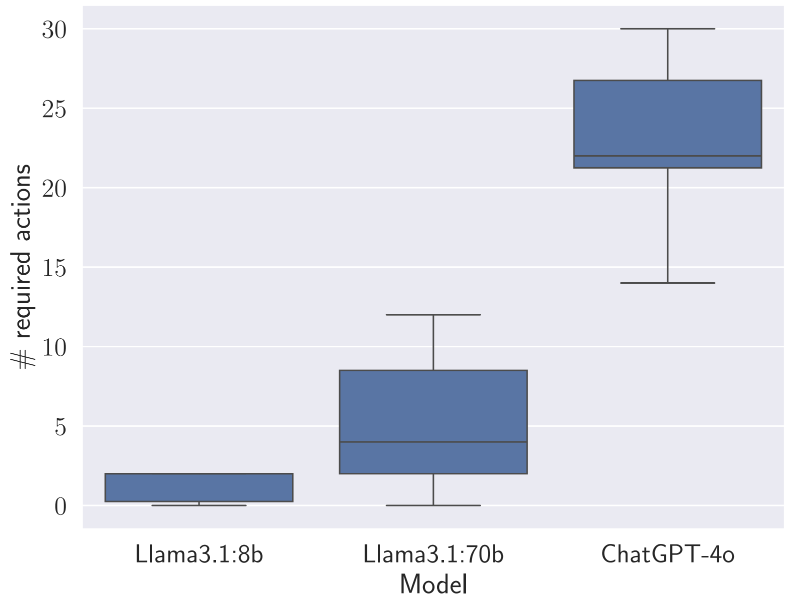



**Figure 5.3.:** Boxplot of the number of correct actions (required)

the ChatGPT-4o model is higher than the maximum of the other two models. It can also be seen that the Llama3.1:70b model is significantly better than the Llama3.1:8b model.

In Figure 5.4 we can see that the ChatGPT-4o model also has the most correct actions that are not required. That means that the ChatGPT-4o model also enters more optional information than the other two models. The Llama3.1:70b model executed more optional actions than the Llama3.1:8b model. On one hand entering optional information is good, because the tool integration is more detailed. On the other hand this is bad, because it takes longer to complete the tool integration. However, the benefits of the additional information outweigh the disadvantages. So in this metric the ChatGPT-4o model is also the best model and the Llama3.1:70b model is also better than the Llama3.1:8b model. Nevertheless, in this metric the differences between the models are not as big as in the previous metric.

In Figure 5.5 we can see that the Llama3.1:8b model has the most no effect actions. This is because it often gets stuck in a loop, where it repeatedly performs the same

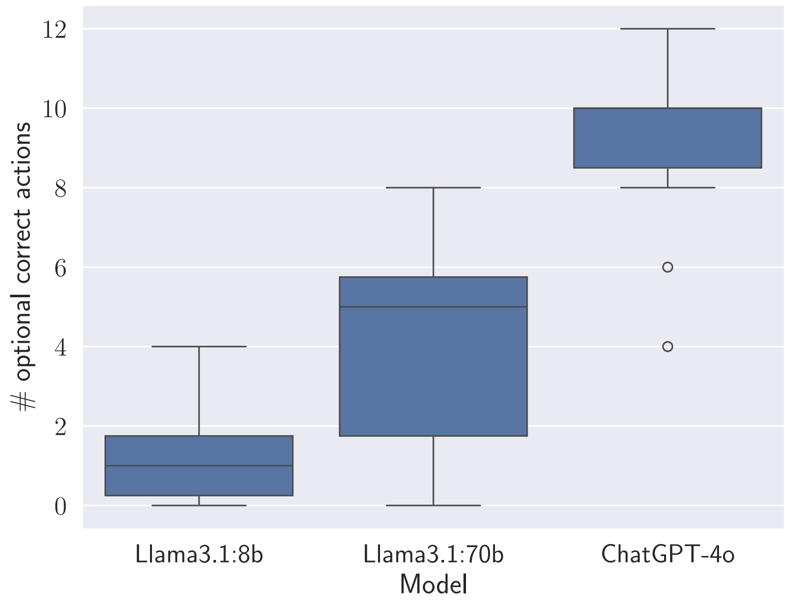



**Figure 5.4.:** Boxplot of the number of correct actions (not required)

action that has no effect. Both the Llama3.1:70b and the ChatGPT-4o model perform only a few actions that have no effect. The reason for that is that both these models do not get stuck in a loop, because they change the action they perform after some tries.

In Figure 5.6 we can see that Llama3.1:8b model has the least amount of wrong actions and Llama3.1:70b and ChatGPT-4o model have about the same amount of wrong actions. This is because the Llama3.1:8b model performs, in general, fewer actions that change the state of the GUI than the other two models. Even so the number of wrong actions is not very high for the Llama3.1:70b and ChatGPT-4o model.

In Figure 5.7 we can see that the ChatGPT-4o model has the least amount of actions that cannot be executed. However, the other two models also have not significantly more actions that cannot be executed.

In Figure 5.8 we can see that the Llama3.1:8b model is the fastest of the three models.

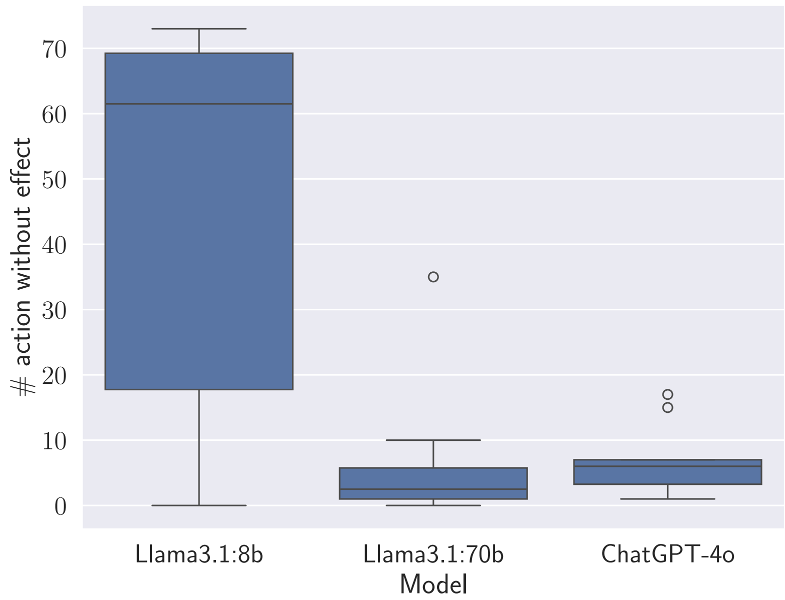



**Figure 5.5.:** Boxplot of the number of no effect actions

The ChatGPT-4o model is slightly slower. The Llama3.1:70b model is by far the slowest of the three models. As already mentioned in Section 5.3, the Llama3.1:70b model is very slow, because it does not fit into the VRAM of the GPU and therefore has to use the RAM, which is much slower. The ChatGPT-4o model is only lower than the Llama3.1:8b model because it is a cloud-based service which has a rate limit.

All in all, the ChatGPT-4o model performs the best out of the three models by far. It performs most of the required actions and also more optional actions than the other two models. On top of that, it is the most efficient model, because it does the least number of actions that have no effect and also the least number of actions that cannot be executed. Effective vise the Llama3.1:70b model is the second-best model. It performs more required actions than the Llama3.1:8b model and also more optional actions. However, it takes far longer than the ChatGPT-4o model. The Llama3.1:8b model is the worst of the three models. Even though it is very fast, it is not able to complete the tool integration in even a rudimentary way.

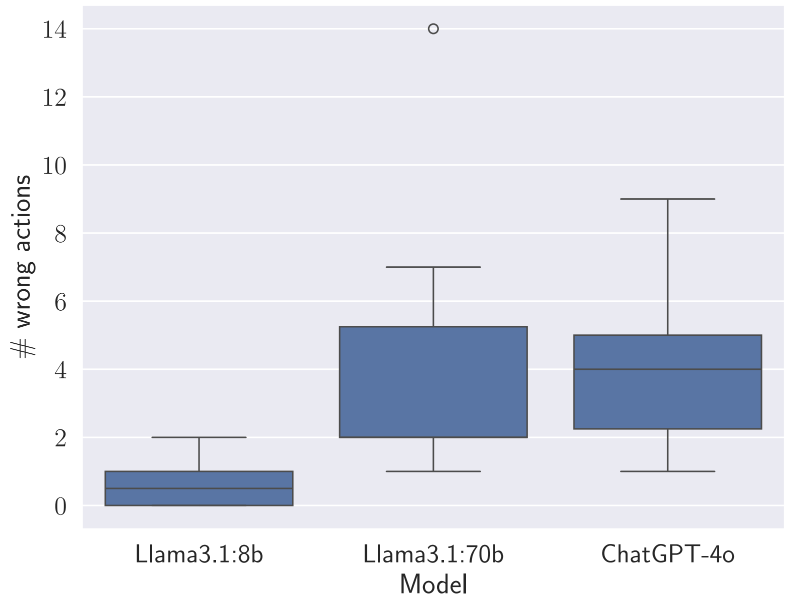



**Figure 5.6.:** Boxplot of the number of wrong actions

## 5.5. Discussion

Now we can discuss if the concept of using LLMs to control the software RCE is a viable option for future applications. We also discuss which model is most suitable for the task of integrating a tool into RCE.

We have tested one task, the integration of a tool into RCE, with three different LLMs. This task is a very complex task and a common use case for RCE. Moreover, similar tasks can be found in many software applications. Therefore, the results of this evaluation can be transferred to other software applications.

The performance of ChatGPT-4o demonstrates that using LLMs to control software is indeed a viable option for future applications. As some test cases show, ChatGPT-4o sometime is able to complete the tool integration into RCE almost perfectly. Although with an average of 23.2 out of 31 required actions completed correctly, it shows that the reliability of ChatGPT-4o is not perfect. However, the system was not optimized for the ChatGPT-4o model due to the time constraints of this thesis.

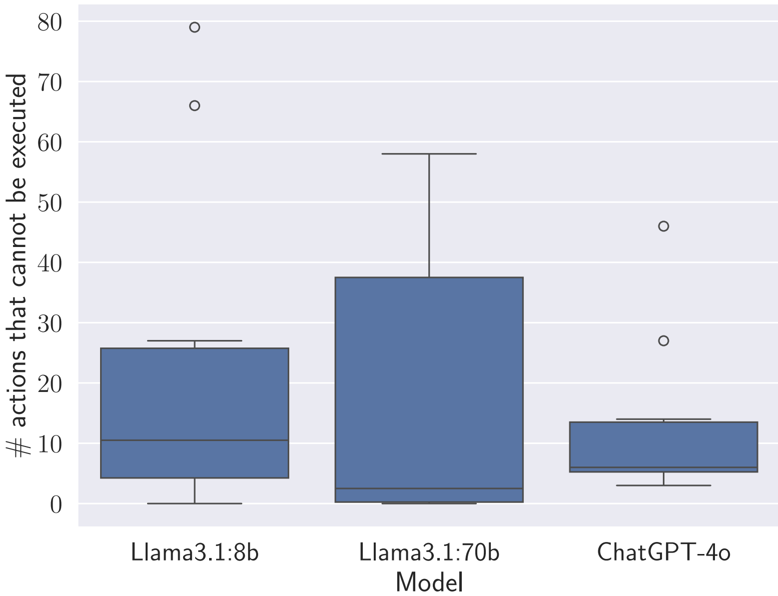



**Figure 5.7.:** Boxplot of the actions that cannot be executed

The ChatGPT-4o model is multimodal, meaning it has both vision and language capabilities. The vision capabilities of the ChatGPT-4o model were not used in this thesis. If the ChatGPT-4o model has access to a screenshot of the GUI in addition to the current prompt, it might understand the context of the GUI better. This enhanced understanding could enable the model to perform even better. In addition, the documentation of RCE could be adapted to the ChatGPT-4o model by adding information about the control elements that are not mentioned in the current documentation. For example adding the information that it is required to select the input handling and the execution constraint when adding an input. Therefore, there is potential to improve the reliability of the system using ChatGPT-4o.

The only problem that is difficult to fix is the cost of using the ChatGPT-4o model. With the current state of the system the cost of using the ChatGPT-4o model is about \$2 per run. That means if we would implement the system as feature in RCE, the cost of one automated tool integration would be \$2. For most users this would be too expensive. As the prompt is also not optimized for the ChatGPT-4o model, there

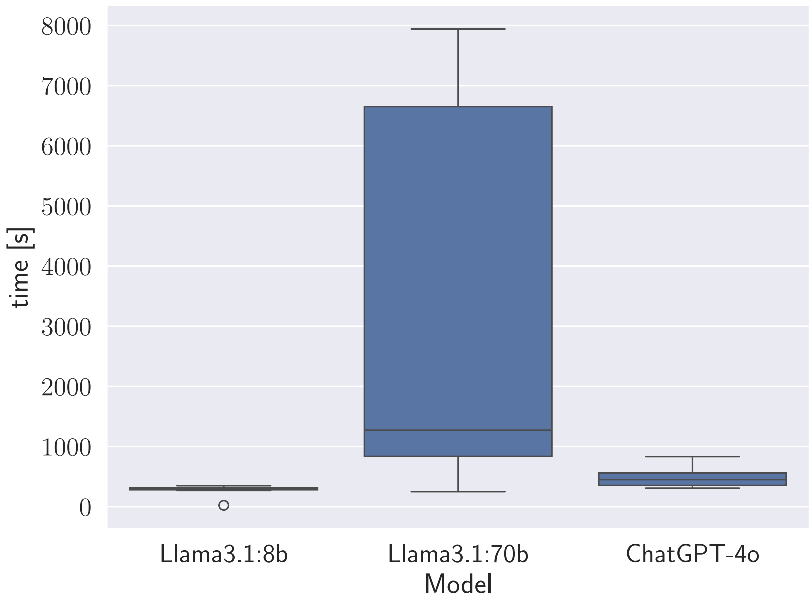



**Figure 5.8.:** Boxplot of the execution time in s

is also the potential to reduce the cost of using the ChatGPT-4o model. For example, when parsing the GUI some elements might not be necessary for the ChatGPT-4o model to perform the required actions. The elements that are not necessary could be removed from the prompt. Depending on how much the cost can be reduced, the ChatGPT-4o model might be a viable option for future applications.

The result of the two Llama3.1 models have shown that they are not suitable for the task of tool integration into RCE. They just perform too few of the required actions. However, the option to use a local LLM is still not off the table. There exists the even bigger and more powerful Llama3.1:405b model, that could not be tested in this thesis. As the difference in effectiveness between the Llama3.1:8b and Llama3.1:70b model are noticeable, it is possible that the Llama3.1:405b model performs even better than the Llama3.1:70b model and might be able to complete the tool integration into RCE. Yet there is still the performance issue. The Llama3.1:70b model is already very slow on a high-end computer. The Llama3.1:405b model is more than ten times bigger in terms of storage size than the Llama3.1:70b model.

(see Table 5.1) So the resources needed to run the Llama3.1:405b model are even higher. For this model to run smoothly a computation cluster is needed, which is very expensive and might even exceed the cost of using the ChatGPT-4o model. But if the priority is to have more control over the LLM, the Llama3.1:405b model is an option to consider.

So in conclusion, the concept of using LLMs to control software is a promising option for future applications. There is still some work to do to be worthwhile. Depending on the requirements of the applications, future work could focus on different aspects. If it is important that no third party has access to the data, future work should focus on testing the Llama3.1:405b model. If that is not a concern, future work should focus on optimizing the prompt for the ChatGPT-4o model to reduce the cost of using it and add the vision capabilities of the ChatGPT-4o model to improve the reliability of the system. In both cases, the backend of PyWinAuto should be changed to “MS UI Automation”, so that all the control elements of the GUI can be accessed by the LLM. This would fix the problems mentioned in Subsection 4.5.1 and Subsection 4.5.2 and generally would make the system more capable.

# 6. Conclusion

In this thesis, we developed a system that acts as an interface between an LLM and the GUI of RCE. The system enables the LLM to control the GUI of RCE. It achieves this by composing a prompt that includes all the information about the current state of the GUI and its context. The system then gives the prompt to the LLM. After that, the LLM outputs an action based on the prompt. The system executes this action on the GUI and updates the prompt with the new state of the GUI. We also developed a memory system based on the work of Liu et al. [15]. This system saves previous actions and includes them into the prompt for the next iteration so that the LLM remembers its previous actions.

We tested the system by giving it the task of integrating a Python script into RCE. We conducted the test with the three LLMs Llama3.1:8b, Llama3.1:70b and ChatGPT-4o. The Llama3.1:8b is extremely fast, with an average execution time of about 275s, but was only able to solve a fraction of the task. The Llama3.1:70b is extremely slow, with an average execution time of about 55 minutes, but is able to solve the first steps of the task. The ChatGPT-4o comes very close to fully solving the task in reasonable amount of time, with an average execution time of about eight minutes. However, GPT-4o costs a lot of money to run. Additionally, the user has very little control over the data being processed, as it is a cloud service provided by OpenAI without the option of an on-premise solution.

So out of the three tested LLM only ChatGPT-4o is a viable option for the task of integrating an external tool into RCE. Nevertheless, the results have shown that the system is capable of acting as an interface between an LLM and the GUI of RCE. It has shown that LLMs are at a stage where they can be used as a controller to automate tasks on external tools. In this thesis we have shown that one concrete viable application is the automation the tool integration in RCE.

For future improvements, the system should switch to using the "UIA" backend of PyWinAuto. This backend provides more detailed information about specific GUI elements. This additional information would allow the LLM to interact with every GUI element in RCE. It would also enable the LLM to provide more specific details about certain elements.

After that, the system could be fine-tuned for the ChatGPT-4o model to make it more reliable, faster and cheaper. Such fine-tuning is a long process where a lot of tests must be conducted to determine which parts of the prompt are necessary, which parts are not and which parts can be improved. Depending on how far the system can be fine-tuned, it could be implemented as a feature in RCE to automate or simplify the integration of external tools.

However, there might be other requirements for a future application. If the requirements are that the LLM must run locally so that the user has full control over the data, the ChatGPT-4o model could not be used. Even though the Llama3.1:8b and Llama3.1:70b were not able to solve the task, it has shown that the amount of parameters in the LLM makes a big difference in how well they can solve a task. This suggests that the even larger Llama3.1:405b model could solve the task. It runs locally but needs a lot of resources to run. Therefore, if the requirements rule out ChatGPT-4o and the resources are available, testing the system with the Llama3.1:405b model would be the next step.

Furthermore, the use case of the system is not limited to the integration of external tools in RCE. As the interface is generic, the task of the LLM could be changed to any other task that can be done on the GUI of RCE. For example, the system could be altered to automate GUI testing. There are different ways how this could be achieved. One way would be to create or use existing GUI test cases and let the LLM execute them using the system developed in this thesis. A different system would then take screenshots of the GUI after each step. This system then gives the screenshot and an adequate prompt to a vision model, e.g. LLaVA [20], for evaluation. The system then determines based on the output of the vision model if the test case was successful or not.

# Bibliography


[1] *Ape | Tianxiao Gu*. https://gutianxiao.com/ape/. (Visited on 07/23/2024).

[2] Brigitte Boden et al. *RCE*. Zenodo. 11/2019. DOI: 10.5281/ZENODO.3691675. (Visited on 07/10/2024).

[3] *Context.Ai*. https://context.ai/compare/llama3-1-70b-instruct-v1/llama2-70b-chat-v1. (Visited on 08/16/2024).

[4] *CPACS*. https://dlr-sl.github.io/cpacs-website/. (Visited on 08/12/2024).

[5] DAIR.AI. *Prompt Engineering Guide*. https://www.promptingguide.ai/. (Visited on 07/01/2024).

[6] Difei Gao et al. *ASSISTGUI: Task-Oriented Desktop Graphical User Interface Automation*. 01/2024. DOI: 10.48550/arXiv.2312.13108. arXiv: 2312.13108 [cs]. (Visited on 04/24/2024).

[7] *GitHub - Kaisugi/Gpt4_vocab_list: Vocabulary List of GPT-4o (O200k_base) and GPT-4/GPT-3.5 (Cl100k_base) Tokenizers. Special Tokens Are Excluded.* https://github.com/kaisugi/gpt4_vocab_list/tree/main. (Visited on 08/26/2024).

[8] *GitHub - Luca-Medeiros/Lang-Segment-Anything: SAM with Text Prompt*. https://github.com/luca-medeiros/lang-segment-anything. (Visited on 04/29/2024).

[9] *GPT-4 Vision vs LLaVA*. https://encord.com/blog/gpt-vision-vs-llava/. (Visited on 08/21/2024).

[10] *GPT-4o vs. GPT-4: Welches Modell ist besser?* https://textcortex.com/de/post/gpt-4o-vs-gpt-4. (Visited on 08/16/2024).

[11] Yuyu He et al. "TextExerciser: Feedback-driven Text Input Exercising for Android Applications". In: *2020 IEEE Symposium on Security and Privacy (SP).* San Francisco, CA, USA: IEEE, 05/2020, pp. 1071–1087. DOI: 10.1109/SP40000.2020.00071. (Visited on 08/14/2024).

[12] Michael Humor. *Understanding "Tokens" and Tokenization in Large Language Models*. https://blog.devgenius.io/understanding-tokens-and-tokenization-in-large-language-models-1058cd24b944. 02/2024. (Visited on 08/19/2024).

[13] *Introducing Llama 3.1: Our Most Capable Models to Date.* https://ai.meta.com/blog/meta-llama-3-1/. (Visited on 08/18/2024).

[14] Karl-Bridge-Microsoft. *Barrierefreiheitstools – Prüfen - Win32 apps.* https://learn.microsoft.com/de-de/windows/win32/winauto/inspect-objects. 10/2023. (Visited on 08/20/2024).

[15] Zhe Liu et al. *Make LLM a Testing Expert: Bringing Human-like Interaction to Mobile GUI Testing via Functionality-aware Decisions.* 10/2023. DOI: `10.48550/arXiv.2310.15780`. arXiv: `2310.15780 [cs]`. (Visited on 04/24/2024).

[16] *Llama.Cpp/Grammars at Master · Ggerganov/Llama.Cpp.* https://github.com/ggerganov/llama.cpp/tree/master/grammars. (Visited on 08/07/2024).

[17] *Llama3.1.* https://ollama.com/library/llama3.1. (Visited on 08/16/2024).

[18] *Llama3.1:405b.* https://ollama.com/library/llama3.1:405b. (Visited on 08/16/2024).

[19] *Llama3.1:70b.* https://ollama.com/library/llama3.1:70b. (Visited on 08/16/2024).

[20] *LLaVA.* https://llava-vl.github.io/. (Visited on 07/30/2024).

[21] Robert Mischke et al. "Automated and Manual Testing in the Development of the Research Software RCE". In: *Computational Science – ICCS 2022.* Ed. by Derek Groen et al. Cham: Springer International Publishing, 2022, pp. 531–544. DOI: `10.1007/978-3-031-08760-8_44`.

[22] *OCR mit Google-KI.* https://cloud.google.com/use-cases/ocr. (Visited on 04/29/2024).

[23] *Ollama/Ollama: Get up and Running with Llama 3.1, Mistral, Gemma 2, and Other Large Language Models.* https://github.com/ollama/ollama. (Visited on 08/07/2024).

[24] *OpenAI API Usage.* https://platform.openai.com/docs/api-reference/introduction. (Visited on 08/25/2024).

[25] *OpenAI Models.* https://platform.openai.com/docs/models. (Visited on 08/26/2024).

[26] *OpenAI Tokenizer.* https://platform.openai.com/tokenizer. (Visited on 08/26/2024).

[27] Luz Perez. *GPT-4o vs. GPT-4: The Ultimate Advances in AI.* 05/2024. (Visited on 08/16/2024).

[28] *Pricing.* https://openai.com/api/pricing/. (Visited on 08/16/2024).

[29] *Pywinauto/Pywinauto*. pywinauto. 07/2024. (Visited on 07/26/2024).

[30] *Rich Client Platform - Eclipsepedia*. https://wiki.eclipse.org/Rich_Client_Platform/. (Visited on 08/03/2024).

[31] Tim Rosenbach. *Improving RCE with User Centered Design*. T3_2000. DHBW Mannheim, p. 78.

[32] shanepeckham. *Getting Started with LLM Prompt Engineering*. https://learn.microsoft.com/en-us/ai/playbook/technology-guidance/generative-ai/working-with-llms/prompt-engineering. 03/2024. (Visited on 07/31/2024).

[33] *SWT: The Standard Widget Toolkit | The Eclipse Foundation*. https://www.eclipse.org/swt/. (Visited on 08/03/2024).

[34] testing22. *Testing661/QTypist*. 07/2024. (Visited on 07/23/2024).

[35] *User Guide of RCE*. (Visited on 08/13/2024).

[36] Ashish Vaswani et al. *Attention Is All You Need*. 08/2023. DOI: `10.48550/arXiv.1706.03762`. arXiv: `1706.03762 [cs]`. (Visited on 08/19/2024).

[37] *What Is a Large Language Model (LLM)?* https://www.cloudflare.com/learning/ai/what-is-large-language-model/. (Visited on 08/19/2024).

[38] *YOLOv8: A New State-of-the-Art Computer Vision Model*. https://roboflow.com/yolov8. (Visited on 04/29/2024).

[39] Wayne Xin Zhao et al. *A Survey of Large Language Models*. 11/2023. DOI: `10.48550/arXiv.2303.18223`. arXiv: `2303.18223 [cs]`. (Visited on 07/04/2024).

# Appendix A.

# Example Prompt

```
Your Role:
You are an automated system that controls the software RCE.
You are given a task, the GUI of RCE and documentation about the software in textual form.
You have to interact with the GUI to achieve the given task.
You can control the GUI by sending actions to the software.
The software will execute the actions and give you feedback about the result, by sending you the new state of the GUI and whether the action was successful or not.
You must use the information about the GUI and the documentation to decide which actions to take.
Include the information about the position of the GUI-Elements and the text of the GUI-Elements in your decision.
Also consider which Parent-Elements the GUI-Elements have.
It is important to take the context of the GUI-Elements into account when deciding which actions to take
You can also use the feedback about the result of the actions to decide which actions to take next.


Your task:
Integrate a Python-script.
  The Path to the Script: C:\Users\rose_ti\Documents\RCE_Tool\calc_volumen.py
  It calculates the volume of an Object.
  It has three parameters:
    --height
    --width
    --length
  The Result is written into a "result.txt" file
  There is a Dokumentation about the Tool at C:\Users\rose_ti\Documents\RCE_Tool\calc_volumen_dokumentation.txt
  There is an icon for the tool at C:\Users\rose_ti\Documents\RCE_Tool\calc_volumen_icon.png
  My name is Tim Rosenbach and my email is tim.rosenbach@dlr.de


Documentation:
# Tool Description

## Synopsis
Define some characteristics of the tool such as name or icon and give some information about the tool integrator.

```

```
## Usage
On this page, some characteristics of the integrated tool are defined. Information on the tool integrator or graphical elements are optional.

Required fields:
- **Name**: This field defines the name of the component. The tool is displayed with this name in the palette view. Note that the name must not contain special characters. Furthermore the name will not be accepted if there is another locally created tool with the same name.

Optional fields:
- **Icon Path**: Select an icon for the tool. At best, it is larger than 32x32 pixels and has the same width and height. To avoid an absolute path link to the icon, select to copy it to the configuration folder.
- **Group Path**: Define a path here under which the component is displayed in the palette view. Subgroups can beseparated by "/".
- **Documentation**: Add a documentation file for the tool. Note that the file must be a PDF or a TXT document and it must not be greater than 50 MB.
- **Description**: Provide a description of the tool, that can be viewed by other users in the network when using the component in a workflow.
- **Name/E-Mail**: In this field, specify some contact information, if anyone has questions or want to makesuggestions.


The current state of the GUI:
[
  {
    "class_name": "Dialog",
    "control_type": "WindowSpecification",
    "control_id": 0,
    "rectangle": [ "L4531", "T1677", "R5274", "B2323" ],
    "text": "Integrate a Tool as a Workflow Component",
    "sub_elements": [
      {
        "class_name": "SWT_Window0",
        "control_type": "HwndWrapper",
        "control_id": 199320,
        "rectangle": [ "L4539", "T1707", "R5267", "B2316" ],
        "sub_elements": [
          {
            "class_name": "SWT_Window0",
            "control_type": "HwndWrapper",
            "control_id": 199316,
            "rectangle": [ "L4539", "T1776", "R5267", "B2315" ],
            "sub_elements": [
              {
                "class_name": "SWT_Window0",
                "control_type": "HwndWrapper",
                "control_id": 68582,
                "rectangle": [ "L4539", "T1776", "R5267", "B2263" ],
                "sub_elements": [
                  {
                    "class_name": "Static",
                    "control_type": "StaticWrapper",
                    "control_id": 68584,
                    "rectangle": [ "L4539", "T1776", "R5267", "B1777" ],
                    "image_description": "The image is a plain, light gray square. It appears to be a simple graphic with no additional elements or text. The color gradient is even and there are no distinct shapes or patterns visible within the square. The background is not
```

```
                distinguishable as it matches the same color of the
                square itself. The simplicity of the image suggests
                that it could serve as a backdrop for other graphics,
                text, or as a placeholder in visual content."
          },
          {
            "class_name": "SWT_Window0",
            "control_type": "HwndWrapper",
            "control_id": 68586,
            "rectangle": [ "L4539", "T1777", "R5267", "B2218" ],
            "sub_elements": [
              {
                "class_name": "SWT_Window0",
                "control_type": "HwndWrapper",
                "control_id": 68638,
                "rectangle": [ "L4542", "T1781", "R5263", "B2215" ],
                "sub_elements": [
                  {
                    "class_name": "SWT_Window0",
                    "control_type": "HwndWrapper",
                    "control_id": 68640,
                    "rectangle": [ "L4545", "T1784", "R5256", "B2211" ],
                    "sub_elements": [
                      {
                        "class_name": "SWT_GROUP",
                        "control_type": "HwndWrapper",
                        "control_id": 68642,
                        "rectangle": [ "L4549", "T1787", "R5254", "B2134" ],
                        "text": "Tool characteristics",
                        "sub_elements": [
                          {
                            "class_name": "Static",
                            "control_type": "StaticWrapper",
                            "control_id": 68644,
                            "rectangle": [ "L4554", "T1809", "R4593", "B1826" ],
                            "text": "Name*:"
                          },
                          {
                            "class_name": "Edit",
                            "control_type": "EditWrapper",
                            "control_id": 68646,
                            "rectangle": [ "L4645", "T1807", "R5248", "B1828" ]
                          },
                          {
                            "class_name": "Static",
                            "control_type": "StaticWrapper",
                            "control_id": 68648,
                            "rectangle": [ "L4554", "T1838", "R4607", "B1855" ],
                            "text": "Icon path:"
                          },
                          {
                            "class_name": "Edit",
                            "control_type": "EditWrapper",
                            "control_id": 68650,
                            "rectangle": [ "L4645", "T1836", "R5032", "B1857" ]
```

```
},
{
  "class_name": "SWT_Window0",
  "control_type": "HwndWrapper",
  "control_id": 68652,
  "rectangle": [ "L5035", "T1831", "R5248", "B1861" ],
  "sub_elements": [
    {
      "class_name": "Button",
      "control_type": "ButtonWrapper",
      "control_id": 68654,
      "rectangle": [ "L5035", "T1835", "R5064", "B1858" ],
      "text": "  ...  "
    },
    {
      "class_name": "CheckBox",
      "control_type": "ButtonWrapper",
      "control_id": 68656,
      "rectangle": [ "L5068", "T1838", "R5248", "B1855" ],
      "text": "Copy into configuration folder",
      "check_state": "checked"
    }
  ]
},
{
  "class_name": "Static",
  "control_type": "StaticWrapper",
  "control_id": 68658,
  "rectangle": [ "L4554", "T1868", "R4616", "B1885" ],
  "text": "Group Path:"
},
{
  "class_name": "Edit",
  "control_type": "EditWrapper",
  "control_id": 68660,
  "rectangle": [ "L4645", "T1866", "R5032", "B1887" ]
},
{
  "class_name": "Button",
  "control_type": "ButtonWrapper",
  "control_id": 68662,
  "rectangle": [ "L5035", "T1865", "R5064", "B1888" ],
  "text": "  ...  "
},
{
  "class_name": "Static",
  "control_type": "StaticWrapper",
  "control_id": 68664,
  "rectangle": [ "L4554", "T1895", "R4642", "B1912" ],
  "text": "Documentation: "
},
{
  "class_name": "Edit",
```

```
                "control_type": "EditWrapper",
                "control_id": 68666,
                "rectangle": [ "L4645", "T1893", "R5032", "B1914" ]
              },
              {
                "class_name": "Button",
                "control_type": "ButtonWrapper",
                "control_id": 68668,
                "rectangle": [ "L5035", "T1891", "R5064", "B1914" ],
                "text": "  ...  "
              },
              {
                "class_name": "Static",
                "control_type": "StaticWrapper",
                "control_id": 68670,
                "rectangle": [ "L4554", "T1918", "R4617", "B1935" ],
                "text": "Description:"
              },
              {
                "class_name": "Edit",
                "control_type": "EditWrapper",
                "control_id": 68672,
                "rectangle": [ "L4645", "T1918", "R5248", "B2129" ]
              }
            ]
          },
          {
            "class_name": "SWT_GROUP",
            "control_type": "HwndWrapper",
            "control_id": 68674,
            "rectangle": [ "L4549", "T2138", "R5254", "B2208" ],
            "text": "Contact Information",
            "sub_elements": [
              {
                "class_name": "Static",
                "control_type": "StaticWrapper",
                "control_id": 68676,
                "rectangle": [ "L4554", "T2160", "R4588", "B2177" ],
                "text": "Name:"
              },
              {
                "class_name": "Edit",
                "control_type": "EditWrapper",
                "control_id": 68678,
                "rectangle": [ "L4593", "T2158", "R5248", "B2179" ]
              },
              {
                "class_name": "Static",
                "control_type": "StaticWrapper",
                "control_id": 68680,
                "rectangle": [ "L4554", "T2184", "R4589", "B2201" ],
                "text": "E-Mail:"
              },
```

```
                    {
                      "class_name": "Edit",
                      "control_type": "EditWrapper",
                      "control_id": 68682,
                      "rectangle": [ "L4593", "T2182", "R5248", "B2203" ]
                    }
                  ]
                }
              ]
            }
          ]
        }
      ]
    },
    {
      "class_name": "Static",
      "control_type": "StaticWrapper",
      "control_id": 68600,
      "rectangle": [ "L4539", "T2261", "R5267", "B2262" ],
      "image_description": "The image is a plain, light gray square. It appears to be a simple, flat graphic with no distinct features or objects. This could be used as a background or for design purposes where minimalism is desired."
    }
  ]
},
{
  "class_name": "SWT_Window0",
  "control_type": "HwndWrapper",
  "control_id": 68602,
  "rectangle": [ "L4539", "T2263", "R5267", "B2316" ],
  "sub_elements": [
    {
      "class_name": "Toolbar",
      "control_type": "ToolbarWrapper",
      "control_id": 68604,
      "rectangle": [ "L4551", "T2279", "R4572", "B2299" ]
    },
    {
      "class_name": "SWT_Window0",
      "control_type": "HwndWrapper",
      "control_id": 68606,
      "rectangle": [ "L4706", "T2263", "R5267", "B2316" ],
      "sub_elements": [
        {
          "class_name": "Button",
          "control_type": "ButtonWrapper",
          "control_id": 68608,
          "rectangle": [ "L4718", "T2277", "R4820", "B2300" ],
          "text": "< &Back"
        },
        {
          "class_name": "Button",
          "control_type": "ButtonWrapper",
          "control_id": 68616,
          "rectangle": [ "L5153", "T2277", "R5255", "B2300" ],
          "text": "Cancel"
        }
      ]
```

```
                  }
                ]
              }
            ]
          },
          {
            "class_name": "Static",
            "control_type": "StaticWrapper",
            "control_id": 199324,
            "rectangle": [ "L5173", "T1707", "R5266", "B1720" ],
            "image_description": "A grayish background with a light blue diagonal gradient. It appears to be a digital graphic or an image with a plain, unidentifiable texture."
          },
          {
            "class_name": "Static",
            "control_type": "StaticWrapper",
            "control_id": 199314,
            "rectangle": [ "L4545", "T1715", "R5173", "B1734" ],
            "text": "Tool Description"
          },
          {
            "class_name": "Static",
            "control_type": "StaticWrapper",
            "control_id": 68574,
            "rectangle": [ "L4542", "T1743", "R4553", "B1754" ],
            "image_description": "A blurry red circle with a white x. It suggests that something is incorrect or does not work."
          },
          {
            "class_name": "Static",
            "control_type": "StaticWrapper",
            "control_id": 68576,
            "rectangle": [ "L4553", "T1743", "R5174", "B1776" ],
            "text": " Please enter a name for the component."
          },
          {
            "class_name": "Static",
            "control_type": "StaticWrapper",
            "control_id": 68578,
            "rectangle": [ "L4539", "T1743", "R4542", "B1754" ]
          },
          {
            "class_name": "Static",
            "control_type": "StaticWrapper",
            "control_id": 68580,
            "rectangle": [ "L4539", "T1753", "R4553", "B1776" ]
          }
        ]
      }
    ]
  },
  286
]


There are different types of GUI-Elements in the GUI of RCE.
HWndWrapper and StaticWrapper are GUI-Elements that can not be interacted with.
Their only perpose is to display information or group other Gui Elements.
The ButtonWrapper is a GUI-Element that can be clicked.
```

```
The EditWrapper is a GUI-Element that has a text field that can be written into.
The ListBoxWrapper is a GUI-Element that has a list of items that can be selected.
The ComboBoxWrapper is a GUI-Element that also has a list of items that can be selected.
The CheckBoxWrapper is a GUI-Element that can be checked or unchecked by clicking on it.

To control a GUI-Element output a command in the following format:
  <action>(<control_id>), for example click(134478)
For each control type there are different actions posible.
The StaticWrapper and HwndWrapper have no actions.
The ButtonWrapper has the click(<control_id>) action, for example click(134478)
The EditWrapper has the write(<control_id, <text to insert>) action, for example write(134456, "example text")
The ListBoxWrapper has the select(<control_id>, <index>) action, for example select(134456, 1)
The ComboBoxWrapper has the select(<control_id>, <index>) action, for example select(134456, 1)
The CheckBoxWrapper has the click(<control_id>) action, for example click(134456)



You must format your output as the following:
{
  "action": "<action>",
  "explanation": "<what the action does and why you do it>"
}
example 1:
{
  "action": "click(134478)",
  "explanation": "click next to get to the second page"
}
example 2:
{
  "action": "write(134456, 'Airresistenz Calculator')",
  "explanation": "Enter the name in the text field, as this is a required field"
}
example 3:
{
  "action": "select(134456, 1)",
  "explanation": "Select float as the data type, because the input 'material's coefficient' is a float"
}


The previous actions:
[
  {
    "action": "click(68610)",
    "explanation": "I click on the 'Next >' button, as it seems like I need to choose an integration configuration first. Since there is no text field or list box that needs to be filled out and only radio buttons with different options are available, clicking next should guide me through a new page where further information can be entered.",
    "status": "executed"
  }
```

```
]


What action do you want to take to do the next step for achieving the given
    task?
```

**Listing A.1:** Example prompt for the RCE task