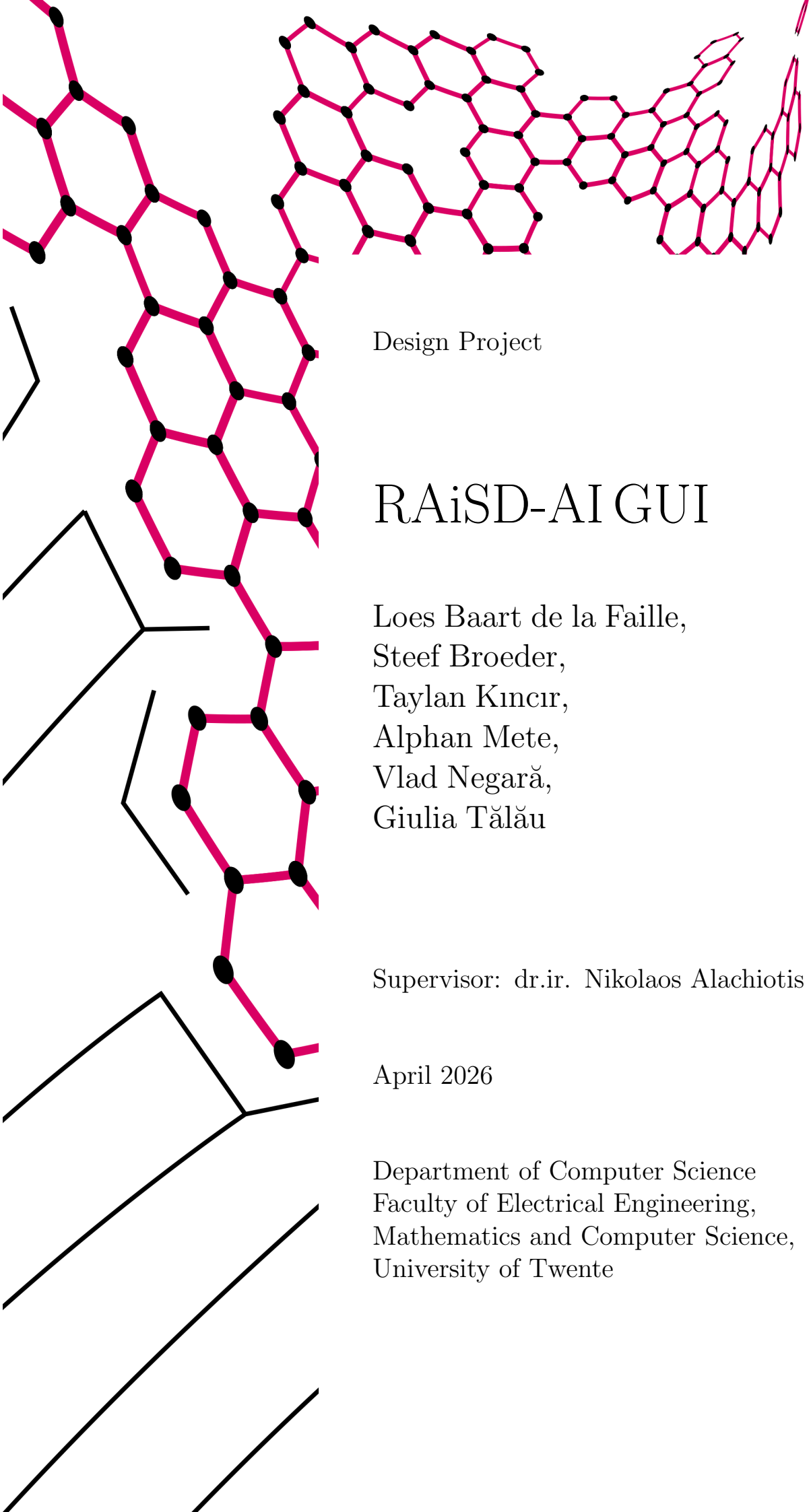


Design Project

RAiSD-AI GUI

Loes Baart de la Faille,
Steef Broeder,
Taylan Kıncır,
Alphan Mete,
Vlad Negară,
Giulia Tălău

Supervisor: dr.ir. Nikolaos Alachiotis

April 2026

Department of Computer Science
Faculty of Electrical Engineering,
Mathematics and Computer Science,
University of Twente

Contents

1 Introduction

RAiSD (Raised Accuracy in Sweep Detection) [1] is a command-line tool that analyzes DNA sequences to identify recent selective sweeps, i.e. regions of the genome that have undergone recent positive selection. RAiSD-AI (Raised Accuracy in Sweep Detection using AI) [2] extends the tool with the option to train a convolutional neural network on genomic data and use it for the detection of selective sweeps. The tool also includes utilities for file conversion and common outlier analysis.

Despite RAiSD-AI's role as an impactful tool in the field of genetics, its command-line interface implies a high barrier to entry for researchers who have only used desktop applications. The goal of this project is to create a graphical user interface that facilitates the use of RAiSD-AI by biologists without programming experience. Such an interface can make using RAiSD-AI for selective sweep detection accessible to a considerably broader audience, potentially leading to scientific discoveries and medical developments.

This report documents the design and development process of the RAiSD-AI GUI project, placing emphasis on significant design decisions and their justifications. Section 2 of the report lays out the planning and deadlines of the project, followed by a thorough risk analysis of the development process. Section 3 presents the artifacts produced by the requirements engineering phase, followed by an account of the initial design and user-tested mock-up of the system in Section 4. Next, Section 5 covers several additional aspects which needed to be carefully considered before commencing the implementation stage. Section 6 showcases the design and implementation of the final product, including the differences from the initial vision. Finally, Section 7 offers insight into the testing strategy employed in this project, and Section 8 contains the team's reflection on task division and the project as a whole. Section 9 explains the ways the ways in which artificial intelligence tools were used. Copies of various additional documents created as part of the development process can be found in the Appendices.

Source code

The source code of the repository, including user and developer manuals, can be found in the GitHub repository.

2 Project organization

Project organization is an important part of each project, as it aids in smooth cooperation and progress. Therefore, we carefully considered project organization during the initial phase of our design project. This section discusses the organization of our project. Firstly, it outlines how we communicated during the project and what deadlines we set. Secondly, it also includes a risk analysis of the software development process.

2.1 Planning & Communication

In order to ensure an efficient execution of our project, we created a plan with clear deadlines at the start, and we committed to following it. Each deadline was divided into tasks that were tracked using Trello and GitHub Issues. We used WhatsApp to communicate and scheduled daily meetings with the entire team. Moreover, we had weekly meetings with our supervisor on Wednesdays. This meant that our internal deadlines were often on Tuesdays. At the end of the 7th week, we held a presentation for the chair of the Computer Architecture for Embedded Systems research group of the Faculty of Electrical Engineering, Mathematics and Computer Science. Table 1 presents an overview of the team's planning, including both intermediate targets we set for ourselves and deadlines from our supervisor and from the module coordinator.

Week	Internal goal	External deadline
1	Meet supervisor; make planning.	-
2	Finalize project proposal.	-
3	Create initial design; begin user testing.	-
4	Finalize design; start implementation.	-
5	Finalize minimum viable product.	-
6	Finalize working product.	-
7	Prepare for supervisor presentation.	Supervisor presentation
8	Implement improvements; finalize report and poster.	-
9		-
10		Report and poster deadline

Table 1: Weekly planning and deadlines

2.2 Risk Analysis

We conducted an analysis of the possible risks we could face during the software development process. Table 2 details the identified factors that could negatively impact the project, both during the development process and the post-development stage. It also includes preventive steps taken to mitigate these identified risks.

Software development risk	Preventive steps
Poor time management	Track project progress; consider value-effort trade-offs.
Incomplete or insufficient requirements	Write a clear project proposal and discuss it with the client.
Flawed software architecture	Draw diagrams to visualize and validate design; follow sensible design patterns.
Unforeseen change in requirements by the client	Include a time buffer in the planning; clearly agree on requirements and deliverables.
Scope creep	Only take on additional requirements that are feasible and add tangible value.
Low team productivity	Hold regular meetings; always have tasks assigned to every team member; hold each other accountable.
Unsuitable or outdated tech stack	Research the relevancy, scalability, integration, performance of widespread tech stacks; write well-structured and well-documented code easily adaptable to new technologies.
Low-quality code or bugs	Adhere to coding standards; hold code reviews after adding functionality; write tests for new functionality; test thoroughly and regularly.
Lack of compatibility with user systems	Test the system on several prominent Linux distributions.

Table 2: Software development risks and corresponding preventive steps

3 Project Requirements

The overall objective of the project is to design and implement a user-friendly interface for the RAiSD-AI tool that is intuitive and visually appealing, to simplify input, execution, and output representation. The RAiSD-AI GUI is meant to lower the entry barrier to using the RAiSD-AI tool and facilitate more efficient and accurate data analysis, enabling researchers and biologists with limited programming experience in using the tool.

This section opens with an analysis of the stakeholders of the system, both direct and indirect. An exploration of existing solutions is followed by a collection of user stories, which encode the functionality of the system in an informal manner. Finally, an overview of the requirements of the RAiSD-AI GUI is provided. The requirements are divided into functional and non-functional, both of which are in turn divided into necessary and optional.

3.1 Stakeholder analysis

The stakeholders with the greatest influence on the design of the system are the end users—biologists with limited technical expertise who are interested in using RAiSD-AI to perform sweep scans. The focus of the project lies in making the existing RAiSD-AI tool more accessible to them, and all design decisions must be taken with this aspect in mind.

Additionally, the design of the project is shaped by the interests of the development team. The team's time investment in developing the system and possible long term-support make code clarity and readability a high priority matter.

Furthermore, the project's supervisor also has a stake in its outcome. They had the main role in writing the original RAiSD-AI tool. Improving the usability of the tool is anticipated to increase its adoption in the world of genetics research, leading to greater recognition from other academics in the field. Additionally, our supervisor will also be the long term maintainer of the system, which means they highly value the maintainability of the code. Moreover, if they decide to change the underlying RAiSD-AI tool, the GUI should be extensible and adaptable to those new changes.

Finally, many indirect stakeholders could conceivably be affected by the project. Authors of competing tools may be negatively impacted if the GUI causes users to migrate from their platform to RAiSD-AI. Entities that finance genetics research (e.g. large pharmaceutical companies) may have an interest in the resulting product, as a tool to increase productivity.

3.2 Analysis of existing systems

In order to avoid preventable, known mistakes, we investigated existing projects that implemented similar applications.

3.2.1 Previous design project

A particularly relevant project is the RAiSD-AI GUI and web service [3] previously implemented as a Bachelor Design Project at the University of Twente. This project's major shortcoming relates to its accessibility and ease of use. As a consequence, we decided to

carefully consider the accessibility we would offer, as well as to make the interface as user friendly as possible. Our project supports an extensive suite of features (such as submitting multiple operations in sequence), and at any point our interface should be clear and the user should know which options are available to them. Another challenge that the previous team encountered is security. Since our project only involves the development of a locally run graphical user interface with no web service, the security risks are reduced. Nonetheless, we consider security very important, and have therefore been mindful of it throughout all steps of the process, considering the possible risks and mitigation measures.

3.2.2 Web scanner

Another relevant project is the RAiSD-AI Web Scanner [4]. Although the GitHub page states the goal of the RAiSD-AI Web Scanner as providing a ready-to-use web interface with pre-trained models, the system did not provide the necessary data files to be able to run the web interface with all of the intended functionalities. Consequently, we decided to pay close attention to making the GUI usable immediately after setup, without many required additional steps. In order to achieve this goal, the GUI has been validated at the end of the development lifecycle by evaluating it on a clean installation of the operating system and resolving any issues that came up. On the other hand, the web interface in question is a simple utilitarian interface that provides some of the most important functionality of RAiSD-AI. We appreciate the simplicity of the interface, yet think that it misses some crucial functionality. We would like our product to provide all functionalities of RAiSD-AI, to aid in understanding RAiSD-AI, and to simplify the process of using RAiSD-AI for the users. For example, this is be done by allowing them to correct mistakes while filling out the parameters without needing to start over again.

3.2.3 PhyloSuite

Lastly, PhyloSuite [5] is a software system that is not directly related to RAiSD or RAiSD-AI, yet provides similar functionality, related to DNA sequences. On PhyloSuite's GitHub page, the software is described as "an integrated and scalable desktop platform for streamlined molecular sequence data management and evolutionary phylogenetics studies." The installation and execution of the software—at least on a Windows system—is made effortless by the provided video tutorial. The application initially asks for a workspace, with an option to set it as default and not ask again. Afterwards, the user is presented with the landing page which includes three sections. The top menu consists of several options for the user to choose such as flowchart, file, alignment, phylogeny, mitogenome, settings and workplace. The left menu consists of a directory-style view of files and some more functionality. One confusing aspect of the tool is that there are several ways to reach the same functionality. Moreover, several icons were animated, which made them feel distracting and out-of-place. On the other hand, the system embedded the file selection system within the software while preserving the look of the native Windows file browser. This was appreciated, and was taken as inspiration for our project. The general design language of the application also followed common practice, which made it easier for users to navigate the application. Overall, PhyloSuite application provided the group with an important example with particular strengths and weaknesses.

3.3 User stories

This subsection lists the expected functionality of the system from the perspective of the end user. Each user story is accompanied by its corresponding use case.

Run RAiSD As a user, I want to use RAiSD to conduct a sweep scan.

Run RAiSD-AI As a user, I want to use RAiSD-AI to prepare input data, train a model, evaluate its performance and/or run a sweep scan using the generated model.

Pick input files As a user, I want to choose the input files on my computer.

Fill in parameters As a user, I want to fill in the mandatory parameters and adjust the values of optional parameters.

Check input As a user, I want to verify the files and parameters I have selected before running the operations.

Inspect output As a user, I want to be presented with the output of my RAiSD-AI executions.

Inspect history As a user, I want to be provided with a list of my previous RAiSD-AI executions.

Reuse output As a user, I want to easily run RAiSD-AI using the output of a past operation as input.

Re-run execution As a user, I want to easily tweak the parameters of a previous execution and run it again.

3.4 Functional requirements

In this subsection, the previously established features are formalized into functional requirements. The requirements are divided into necessary and optional, prioritized by the utility they provide to the end user.

Necessary

- The system shall, in all modes, support the mandatory `-n` and `-I` parameters and all optional parameters. These include parameters relating to SNP and sample handing (`-L`, `-S`, `-m`, `-M`, `-y`, `-X`, `-B`, `-o`), sliding window and μ statistic (`-w`, `-c`, `-G`, `-VAREXP`, `-SFSEXP`, `-LDEXP`), standard output and reports (`-f`, `-s`, `-t`, `-p`, `-O`, `-R`, `-P`, `-D`, `-A`), accuracy and sensitivity evaluation (`-T`, `-d`, `-k`, `-l`), FASTA to VCF conversion (`-C`, `-C2`, `-H`, `-E`), VCF to ms conversion (`-Q`), common-outlier analysis (`-CO`, `-COT`, `-COD`) and other (`-frm`, `-gpu`, `-useTF`, `-b`, `-a`).
- The system shall support five modes of operation and allow the user to choose between them:

- Standard RAiSD:
  - The system shall allow the user to input data in one of the following file formats: ms, (compressed) VCF, FASTA.
  - The system shall allow the user to perform a full scan of the input data using a standard RAiSD execution.
- Data generation:
  - The system shall allow the user to input data in one of the following file formats: ms, (compressed) VCF, FASTA.
  - The system shall allow the user to input the mandatory `-icl` and `-its` parameters, as well as the optional `-poc`, `-ips`, `-iws`, `-bin`, `-typ` and `-w` parameters.
  - The system shall allow the user to generate images that can be used for training and/or inference.
- Training:
  - The system shall allow the user to input training data in the PNG file format.
  - The system shall allow the user to input the optional `-arc`, `-e`, `-cl4`, `-cl4_1x_label`, `-cl4_x1_label` and `-g` parameters.
  - The system shall allow the user to generate a neural network model from training data.
- Inference:
  - The system shall allow the user to input testing data in the PNG file format.
  - The system shall allow the user to input a neural network model in the PT file format.
  - The system shall allow the user to input the mandatory `-mdl` and `-clp` parameters.
  - The system shall allow the user to test a neural network model and shall report various metrics, including accuracy and F1-score.
- Sweep scan:
  - The system shall allow the user to input testing data in one of the following file formats: ms, (compressed) VCF, FASTA.
  - The system shall allow the user to input a neural network model in the PT file format.
  - The system shall allow the user to input the mandatory `-mdl` and `-pci` parameters.
  - The system shall allow the user to perform full scans for selective sweeps.

- The system shall allow the user to select the output of a previously executed operation to use as input for a new execution.
- The system shall allow the user to run a sequence of operations consecutively. When more than one operation is submitted, latter operations will use the output of previous operations as input. The user will provide only the necessary inputs, e.g., input data for the first selected operation, testing data for inference, or data to be scanned in the sweep scan.
- The system shall provide a description of each parameter and the required input.
- The system shall provide default values for optional parameters.
- The system shall correctly handle the contingency of optional parameters on the presence or value of other parameters, i.e. the user should be allowed to fill in values only for those parameters that are appropriate for the selected operation and file format.
- The system shall notify the user if any input—data or parameter—is incorrectly formatted.
- The system shall present the user with a confirmation dialog before commencing execution, where the user can review and change the selected files and parameter values.
- The system shall use RAiSD-AI to compute and generate the results of operations specified by the user.
- The system shall display the results of an operation to the user upon completion.
- The system shall allow the user to view the execution details, input and output of past operations.
- The system shall allow the user to specify the location where the input of each operation is stored.
- The system shall allow the user to specify the location where the output of each operation will be stored.
- The system shall notify the user in the case of errors during operation and provide explanations of the errors.
- The system shall offer the user an indication of progress during computationally intensive tasks.

Optional

- The system shall provide a simplified graphical interface as an alternative to the full-featured interface. The simplified interface shall allow users to make use of the full RAiSD-AI pipeline while abstracting away from the specific operations.
- The system shall allow the user to enter a small amount of raw data in a text box, as an alternative to selecting an input file.
- The system shall visualize intermediate results at runtime.
- The system shall notify the user when operations are completed.

- The system shall perform input validation to prevent malicious input from being executed.
- The system shall provide a way for users to access the locations of input and output files in the operating system's file browser.
- The system shall allow the user to delete operation results.

3.5 Non-functional requirements

Beyond measuring the fulfillment of functional requirements, the quality of the system can be judged by a number of criteria relating to ease of use, compatibility, maintainability etc. Just as the functional requirements, the non-functional requirements are classified into necessary and optional.

Necessary

- The system shall be intuitive and easy to understand.
- The installation of system dependencies shall be performed using an automated tool.
- The system shall be locally executed.
- The source code of the system shall be well-documented.
- The installation of system dependencies shall be well-documented.
- The system shall be easily extensible with extra parameters and new features.
- The system shall support Linux operating systems.
- The system shall only store necessary data.
- The system shall not send any data outside of the user's machine.
- The system shall leave the input data unchanged.

Optional

- The system shall be packaged as a stand-alone executable.
- The system shall comply with widely adopted accessibility guidelines.
- The system shall support other popular operating systems, namely Windows and macOS.
- The system shall support localization.

4 Initial system design

This section describes in detail the initial design of the system as devised in the second and third weeks. The design is conveyed through a series of UML diagrams, which each formalize specific flows, structures or functionalities of the system. In this section we first describe the main GUI functionality in Section 4.1 using a sequence diagram. Next, we focus on the interaction between the GUI and RAiSD-AI in Section 4.2 using an activity diagram. Section 4.3 describes the states of the GUI in a state machine diagram and Section 4.4 highlights the classes of the future system in a class diagram. Finally, in Sections 4.5 and 4.6 we highlight the UI design through an iterative process of mock-ups and user testing.

4.1 Main GUI functionality

The main loop of the graphical user interface is represented in the UML sequence diagram shown in Figure 1. A sequence diagram is suitable to describe the interaction between the user and the high-level entities of the system.

The four lifelines in the diagram correspond to the user, the GUI application itself, the file system of the machine and the RAiSD-AI tool, respectively. The process begins with the user opening the GUI, at which point the GUI checks for a known recent workspace. If no recent workspace is known, the user is prompted to select a directory as the current workspace. Next, the GUI must fetch the data of past executions from the machine's file system and display it to the user. From this point on, the GUI is in the main loop of responding to user's input until the user decides to close the window. Whenever the user is faced with the main menu, they have a choice between four primary options—reading general information about the tool, changing the current workspace, submitting a new operation to be run using RAiSD-AI or inspecting an operation that has already been run.

First, the general information is displayed as a static page that explains the main purpose of the tool, credits the developers and links to the public repository where the code can be found. The user can return to the main menu after reading the page.

If the user intends to change the current workspace, a directory selection dialog is presented which allows them to select the new location.

As submitting one or more operations to be run using RAiSD-AI is a complex process that can be initiated in different sections of the GUI, it has been extracted to a separate diagram (Section 4.2).

Should the user opt to inspect a previous execution, the GUI application fetches the appropriate execution details and results to display. Afterward, the user may choose to submit a new operation using the output as input (if applicable, depending on the specific RAiSD-AI operations involved), re-run the execution with adjusted parameters, or simply close the details and return to the main menu. Once again, the options that lead to starting a new execution reference the dedicated diagram for that process.

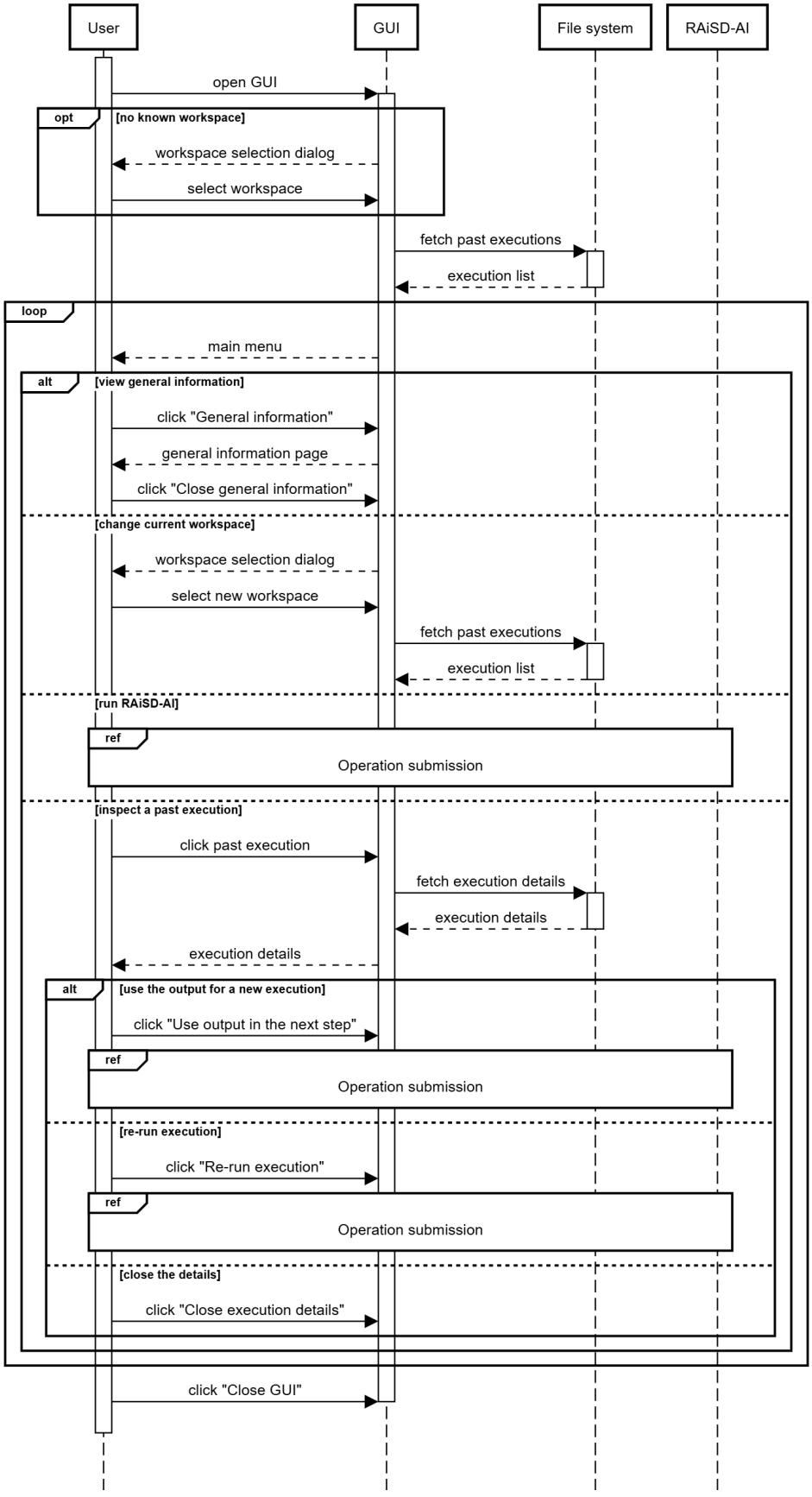


Figure 1: Sequence diagram—GUI loop

4.2 Running RAiSD-AI

The succession of steps involved for the user to run RAiSD-AI operations using the GUI is best depicted through a UML activity diagram. Shown in Figure 2, this diagram involves the user, the GUI application, and the RAiSD-AI backend. Each of these entities is modeled as a swimlane in the diagram.

The process begins with the user selecting the operations they want to perform, which can be either a single operation or a sequential chain of operations. Selected operations are then evaluated by the system, and the user is instructed to re-adjust them in case of an incompatible selection.

The user is then requested to select input and output locations; a default workspace location is used for output in case the user does not select one. After the file selection, applicable parameters to the selected operations are displayed to the user, and the user is instructed to fill in their values. Similarly to operation selection, a validation step takes place for the parameters and feedback is shown to the user. This step is followed by displaying both the selected files and parameter values for final confirmation. If the user wants a change in either file locations or parameter values, they can return to previous pages to make these changes.

After receiving final confirmation from the user, an operation record is created, and the RAiSD-AI process starts while the interface offers the user an indication of progress. If execution is interrupted by an error, the explanation of the error is displayed to the user, and the process finishes. If the process terminates successfully, the output is collected, and the application either loops back and calls RAiSD-AI again for the next operation, or finalizes the operation record and displays the results to the user.

4.3 System state overview

The system tracks the state of a RAiSD-AI execution in the GUI, from initialization to termination. A UML state machine diagram naturally models the states and transitions involved in this process.

Figure 3 shows such a diagram, which follows the lifecycle of a RAiSD-AI execution. An execution is created in the `Initialized` state, after which the user is prompted to select operations, input/output file locations and parameter values, in this order. The entry of data at each of the aforementioned steps constitutes a transition to the subsequent state in the diagram. Afterwards, the user is presented with a confirmation dialog with the option to either change the input files, change the parameter values or confirm the selection. These choices lead to regressing to the `Files selected` state, remaining in the `Parameters entered` or advancing to the `Executing` state, respectively. An execution can finish due to an error, or a successful completion, which lead to the `Error` and `Completed` states, respectively, which completes the diagram.

4.4 Software architecture

A structurally sound codebase is integral to the ease and momentum of development by the project team, as well as its comprehensibility and extensibility by future developers. Consequently, this point in the design process is of the utmost importance for the long-term utility that the final product provides. Relevant design patterns should be employed

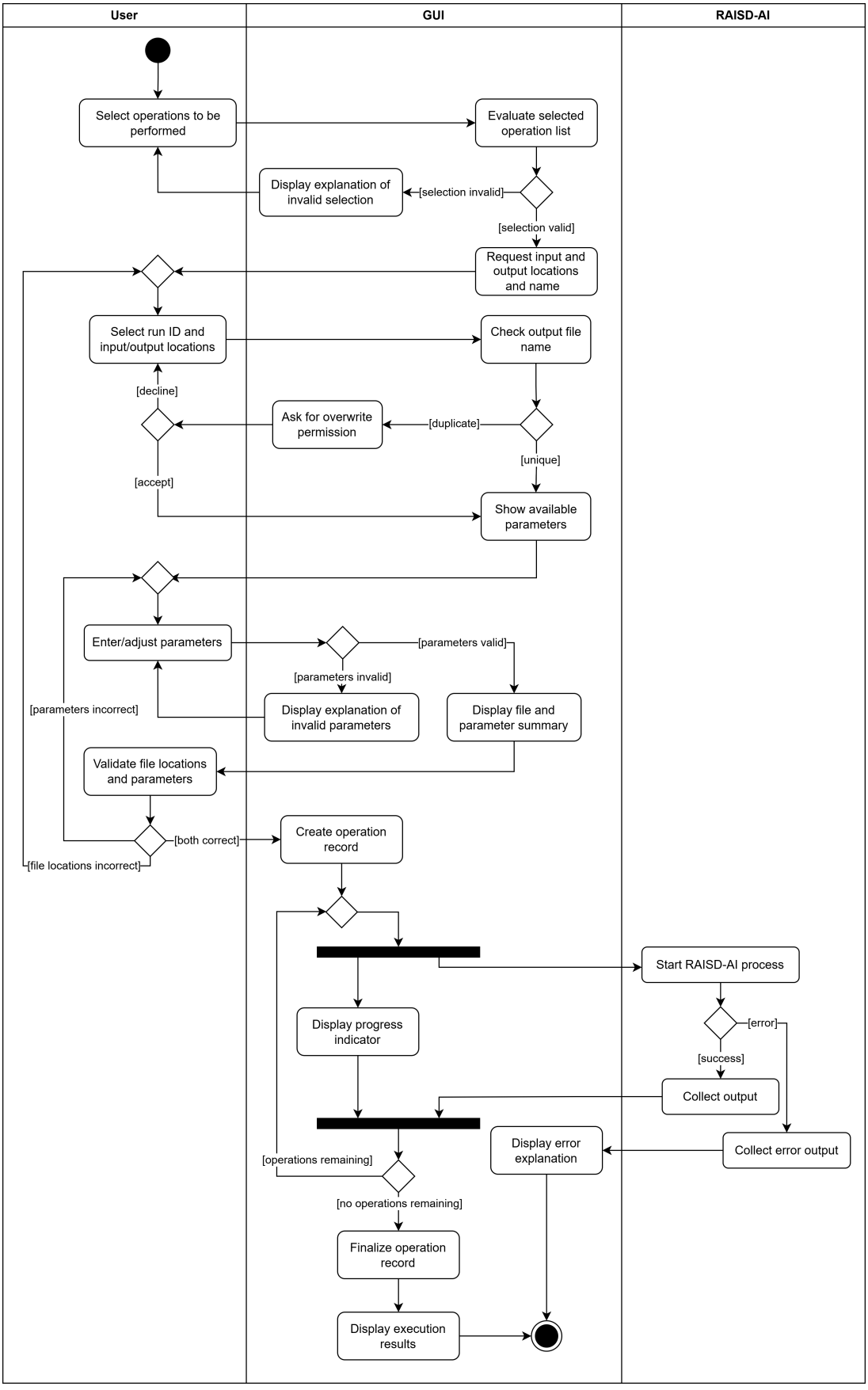


Figure 2: Activity diagram—Operation submission

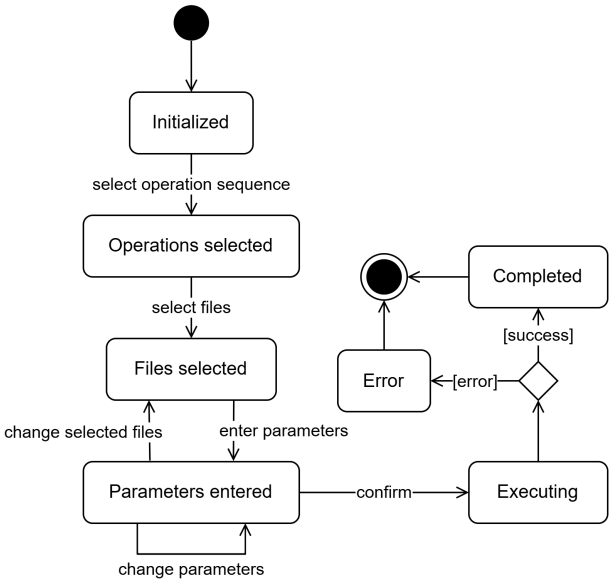


Figure 3: State machine diagram—Execution lifecycle

where applicable, in order to produce more modular and maintainable code.

Figure 4 represents the envisioned class structure of the system in a UML class diagram. The classes are structured by applying a variation of the Model-View-Controller design pattern. The classes corresponding to the Model are located roughly in the left half of the diagram, while the combined responsibility of the View and the Controller is placed on those on the right.

The model represents the RAiSD-AI execution parameters in a hierarchical structure: parameters are grouped by related function, and the groups are all contained within a list. This hierarchy is used for several reasons. First, for checking validity; the object at each level of the hierarchy queries its children and returns the conjunction of their validity. Additionally, it is used for converting the parameters to their command-line representation, where each object uses the representations of its descendants to produce its own. Lastly, the structure is useful in the GUI presentation, where related parameters are grouped together into sections based on the hierarchical structure.

Starting at the bottom of the hierarchy, the system must handle multiple types of parameters: booleans, integers, floating-point numbers, strings, files, enumerated values and combinations thereof. Although they have different specific behaviors, these parameter types also share many common traits: they each have a name, a description, a default and current value, and a flag to represent them in the command line. The abstract, generic `Parameter` class contains this shared functionality, and is parameterized by the type of value it stores. The different types of parameters subclass `Parameter` and override the command-line representation and the check for validity, possibly introducing constraints on allowed values.

Higher in the model hierarchy are the `ParameterGroup` and `ParameterGroupList` classes. A `ParameterGroup` serves to group related parameters, and contains a name and the operations with which the parameters are associated. The `ParameterGroupList` holds a reference to each parameter group. It also presents a method to set an operation as enabled or disabled, which will hide the corresponding parameter groups. Furthermore, the

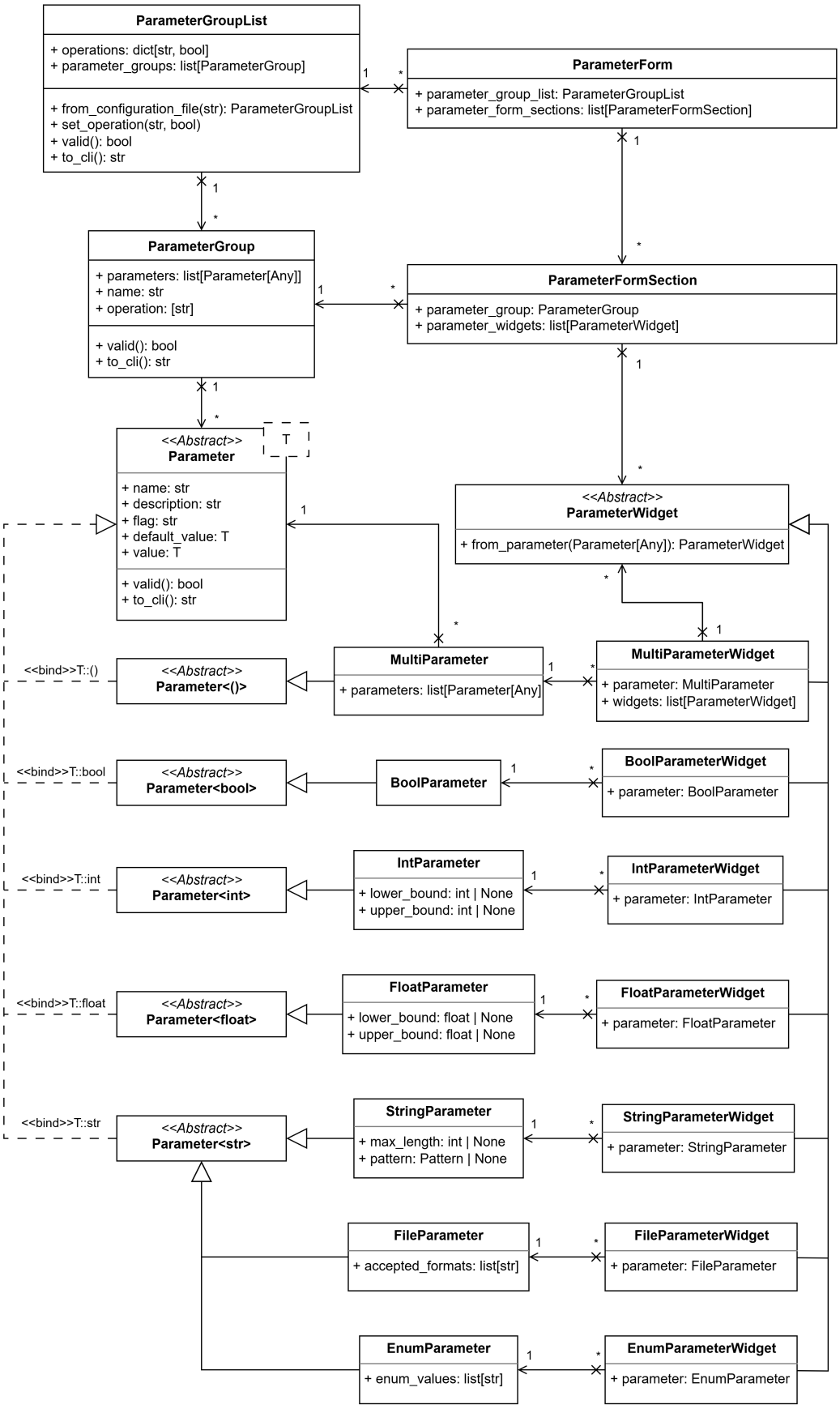


Figure 4: Class diagram

class exposes a factory method named `from_configuration_file`. As the name suggests, this method allows a `ParameterGroupList` object to be initialized from a configuration file by providing its path. The aforementioned file dictates the available operations, information about each parameter, as well as dependencies between them. The existence of this configuration file is what makes the GUI extensible and adaptable to future changes.

The view consists of various widget classes, which can be instantiated programatically based on the data in the model. Each class in the model has a correspondent in the view, which will display its contents in a suitable fashion. At the top of the view hierarchy is the `ParameterForm`. A `ParameterForm` object is initialized with a `ParameterGroupList`. For each parameter group contained within that list, it initializes a `ParameterFormSection` widget. Within a section, a widget is created for each parameter in accordance with its type. The different types of widgets (e.g. `IntParameterWidget`, `StringParameterWidget`) all subclass the abstract `ParameterWidget` class, which enables their creation by means of the factory method `from_parameter`.

4.5 Initial Mock-up

The general objective of this project is to design and implement a user-friendly interface. To ensure this is achieved, user testing is an important step. Therefore, we developed a mock-up that allowed us to user test early, before the implementation phase. Our initial mock-up supports one main sequence of actions, which allows for the demonstration of all important features of the graphical user interface.

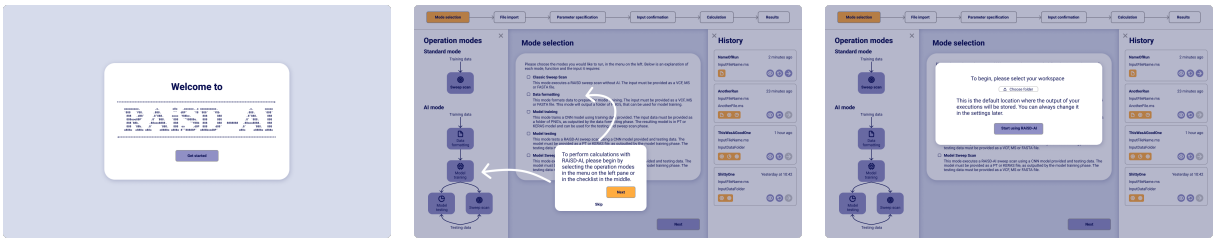
4.5.1 Design

For the design of the mock-up, our main focus was on finding a balance between allowing the user to choose and specify everything themselves, and at the same time keeping the design simple, intuitive and user-friendly, even for users with a minimal understanding of neural networks. We intentionally abstract away from the structure of the RAiSD-AI command-line interface in places where doing so improves usability.

Introduction pages Before the main loop of the graphical user interface commences, as described in Section 4.1, the user encounters a series of introductory pages, shown in Figure 5. These are meant to help the user get started with the system and teach them how to use it. When a user opens the graphical user interface, the first page they see is the landing page, seen in Figure 5a. This page serves as a welcome to the user, informing them the system is ready and inviting them to start using it. Optionally, this page could offer the choice to open a simplified version of the user interface, as described in one of the optional functional requirements in Section 3.4.

After the landing page, a short tutorial is shown to the user. The first page of this tutorial can be seen in Figure 5b. The tutorial is meant to increase usability and learnability by guiding the user through the elements of the graphical interface and their purposes. The tutorial is only shown to the user the first time they open the graphical user interface, and users have the choice to bypass it if desired.

The last introductory page asks the user to select a workspace, as seen in Figure 5c. This workspace will be remembered the next time the user opens the graphical user interface.



(a) Landing page (b) Tutorial (c) Workspace selections

Figure 5: Introduction pages

Main interface layout Once the main loop of the system begins, the pages follow a consistent layout which can be seen in Figure 6.

The first important feature of this layout is the top bar. It functions as an overview of the steps that the user must fulfill for a RAiSD-AI execution. Throughout the process, the top bar can also be used for navigation between pages.

The left panel of the layout contains a visualization of the operation modes. The purpose of the diagram is to aid understanding of the relationships between the different modes of operation and the different forms of input necessary. This is is an important part of attempting to find the balance between user freedom and user-friendliness, by communicating to the user what their options are.

The panel on the right contains the history of past executions. For each execution, a tile with some summarizing information is shown, including the name, input files and operations that were run. The buttons on the tiles allow the user to view the results of a past execution, re-run it after optionally changing some input, and use the output of the execution as input for a new run.

The middle section of the interface is where the user follows the main flow of actions of a RAiSD-AI execution. For this, the mock-up has a separate page for each of the steps. We aimed to include as much information and explanation as possible, while keeping the views clean and uncluttered. Both side panels are closable, which makes the view less cluttered and more focused on the middle section, according to the user's preferences.

Mode selection Figure 6 shows the interface for the first step of the RAiSD-ai execution process, mode selection. This page works closely together with the left panel to give the user the ability and understanding to choose the operation modes they prefer. In the mock-up, the modes are shown as a multiple-selection list. A user can select modes in both the diagram and the multiple-selection list, and with every choice both items are updated. This visual connection is meant to help the users understand how the list relates to the diagram. Only when a valid combination of operation modes is selected can the user move to the next step of the process.

File import The next step of the RAiSD-AI execution process in the mock-up design is the file import page as seen in Figure 7. On this page, the user fills in the name for this run, inputs the necessary files, and selects a location for the result to be stored. Similarly to the mode selection page, we aim for this page to explain all the elements so that the user understands what is expected of them. Once the user has entered all the necessary information, they can move on to the next page.

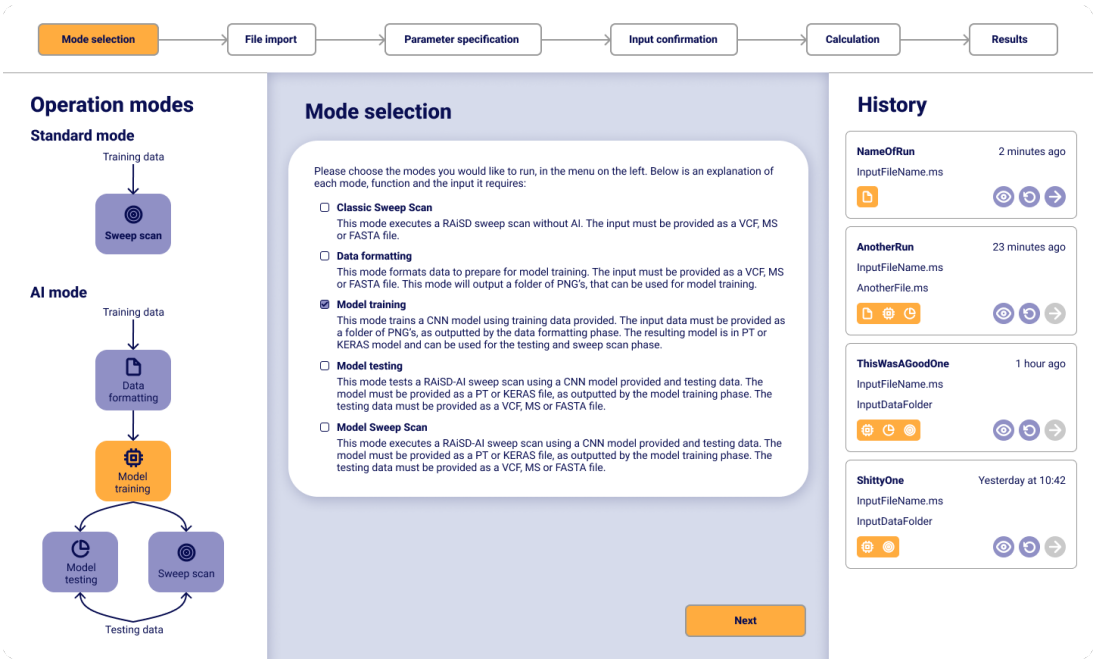


Figure 6: Mode selection interface

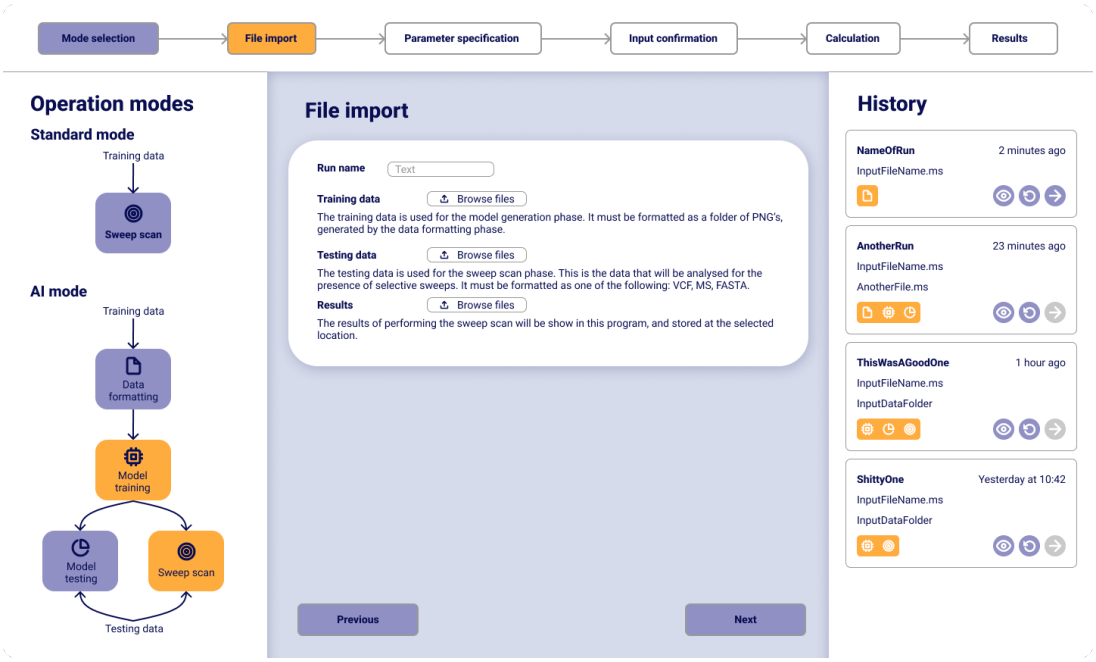
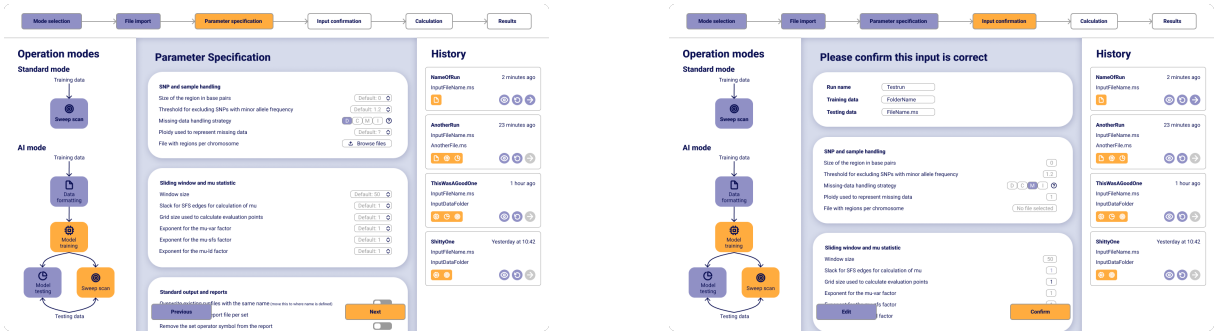


Figure 7: File import interface

Parameter specification After files have been imported, the next page is the parameter specification page, as can be seen in Figure 8a. This page lists all the parameters with meaningful names and shows different input boxes dependent on the input type. Whenever possible, the default value and/or range are shown to give the user a reference value of sensible inputs. Additionally, extra information is available, for example through hover-able elements. For the final implementation, we would like each parameter to have an explanation when you hover over the name, but this was not included in the mock-up.

Input confirmation Once all input has been provided by the user, they are prompted to review their input and confirm that it is correct. Because calculation can take some time, this step offers the user a chance to ensure their input is as desired, and prevents unnecessary and time-consuming mistakes and re-runs. Visually, this page, as seen in Figure 8b, closely resembles the parameter specification page. The main difference is that this page does not allow the user to make changes, as it is meant to validate and confirm. To make changes, the user must move backward by one or more steps. Once all input is confirmed, the system proceeds to the calculation phase.



(a) Parameter specification (b) Input confirmation

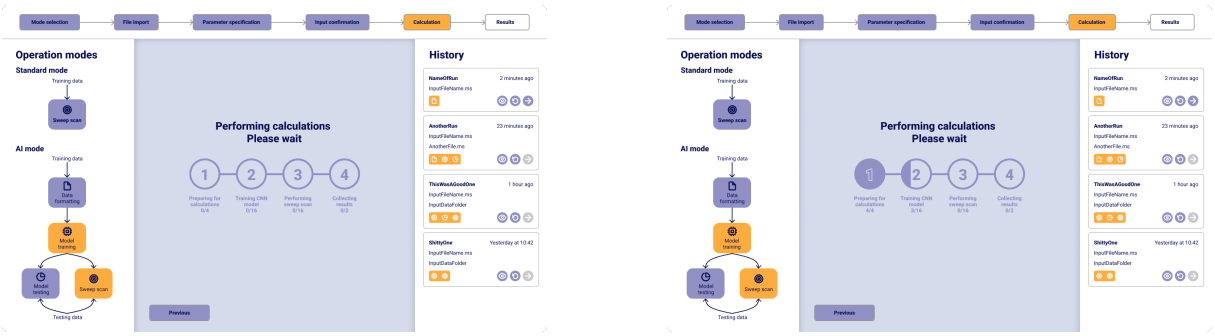
Figure 8: Parameter specification and input confirmation interfaces

Calculation After input confirmation, the user is shown an intermediate page, while the desired operations are executed with RAiSD-AI. The page, as shown in Figure 9, includes a visual indicator of progress. The indicator consists of several steps, each with the number of tasks that must be done, and gradually fills as shown in Figure 9b. The purpose of the animation is to show the user computational progress and make it clear that they must wait.

Results The final interface of the process is the results page. This interface can be seen in Figure 10. In this mock-up, the results page is not entirely accurate with the actual RAiSD-AI output yet. Nevertheless, this page is intended to show the results of running RAiSD-AI to the user. Additionally, it includes all the parameters that were filled in to reach this result, so that the user does not have to go back to view those. The "done" button takes the user back to the mode selection page, where they can begin the process over again.

4.5.2 User Testing

After finishing the high fidelity prototype of the GUI, user tests were conducted to better understand the expectations of the intended users and make the required changes in the



(a) Animation initial stage

(b) Animation in progress

Figure 9: Calculation interface with waiting animation

Figure 10: Results interface

design before starting the implementation of the system.

Testing Setup User testing for the initial design was conducted in the form of interview sessions. The participants were first asked three screening questions to understand their background in biology and their experience with scientific tools. Afterwards, they were briefly introduced to the concept of selective sweeps and the functionality of the RAiSD-AI tool in accordance with their prior knowledge. Following this, they were given a mock task of performing a calculation using the GUI with minimal help from the interviewer. The given task involved using two RAiSD-AI modes, model training and sweep scan, with parameters specified in the task description. The users were observed as they navigated through the interface and completed the given task step by step. After the task was completed, they were asked several questions to reflect on their experience with the GUI mock-up.

The user tests were conducted with 14 participants with varying educational backgrounds. To reflect our intended users, we focused on finding participants with a biology background, but people with other backgrounds were also included. The complete questionnaire can be seen in the Appendix A.

Conclusion of Reflection Questions

1. **What is your overall impression of the GUI in terms of features and layout?** The overall impressions were mostly positive. Most of the participants liked the structure and visual language, although some of the participants found the mode selection step slightly confusing and mentioned it was not clear from the UI that they would have been able to do multiple operations on a single run. Some participants mentioned that the name difference between the side panel and the main window ("Operation modes" and "Mode selection") decreased clarity and also asked for better separation between standard RAiSD and RAiSD-AI operations. One participant mentioned that the Parameter Selection page was text heavy and therefore separating it into multiple pages would be helpful.

2. **What do you think of the help tutorial?** Most of the participants found the help tutorial clear and straightforward. Only one participant mentioned that the arrows that point to sections of the screen were unclear and suggested highlighting the section instead. One participant mentioned that they would like to automatically skip the tutorial after their first time using the GUI.

3. **Would you want to be able to close left and right panes of operation modes and history?** Most of the participants replied that they would like to be able to close the side panels. While some participants suggested displaying the history panel on the first and last steps and automatically hiding them during the rest of the process, others mentioned having an operation mode panel always present was useful. Three participants replied that they did not mind having the side panels and did not feel the need to close them.

4. **What do you think of selecting operation modes?** Several participants have mentioned that they were confused about the fact that there were two "selective scan" modes. Therefore, a better separation between the two sweep scan modes might be useful. Additionally, one participant mentioned that being able to select

multiple operations in a single run was not clear enough. Other participants found the mode selection easy to understand.

5. **What is your impression of the page for selecting the operation name and files?** All of the participants mentioned that they were satisfied with this section and did not have problems while using it. Two participants mentioned that some readability improvements, such as more whitespace in between list elements and larger font size, would be useful when the side panels are collapsed.

6. **What do you think of the parameter overview?** The participants mostly liked the default values for the parameters in this section, although some mentioned that informing the user explicitly that the default values are correct might be useful. Around half of the participants have mentioned that they would want some changes with the parameter specification view. Recurring suggestions included making the page less text-heavy by separating parameters into multiple pages or collapsible sections, adding explanatory hints to all parameters, increasing font size, and allowing direct number input instead of button clicks for values like grid size.

7. **How is your experience of confirming files and parameters? How do you find its usefulness?** Most of the participants found the confirmation step useful for detecting mistakes before a potentially time consuming process. However, several participants suggested improvements in this section, such as showing only the parameters that were changed from their default values, making it visually distinct from the parameter input page so it clearly reads as a confirmation rather than an input form. Additionally, allowing changes on the confirmation page itself instead of navigating to previous view was suggested, for example using a pop-up dialog instead of a separate page for confirmation. One participant mentioned that they found this step unnecessary since it was similar to the previous view in which they just entered the values for parameters.

8. **What do you think of the animation that showed the progress of operations?** All participants responded positively to the progress animation. Participants appreciated being able to see which step the calculation was on and how far along the process was.

9. **What do you think of the results page after the operation is finished?** The results page received mixed feedback. While some participants liked having an overview of the results and parameters together, the most common change suggested was increasing the plot size or adding the ability to zoom in. Multiple participants mentioned graph readability was problematic for a detailed analysis. Suggestions also included adding explanations for what the plots represent and renaming "Parameters" to "Parameters used" to better indicate that it is a read-only summary.

10. **What do you think of your general experience while navigating through the GUI? Was proceeding to the next step intuitive for you on each step of operation?** The majority of participants found that navigation through the GUI was intuitive and they appreciated the step-by-step structure. The top bar and Previous/Next buttons were specifically mentioned as helpful navigation aids. Some participants noted that the interface became clearer after the first use. Recurring issues included difficulty locating specific parameters in text-heavy screens and one participant experiencing missing navigation buttons on a tablet.

11. **What surprised you while using GUI (if any)?** Most participants were not surprised by any particular thing in the GUI, indicating the interface mostly met their expectations. A few notable surprises included the colorful visual design (which they found unexpected for a scientific tool), the progress animation (positively surprising), and the amount of information and options presented on screen (which felt overwhelming to some).
12. **If you could change anything about the GUI, what would you change?** Participants suggested a variety of improvements. The most frequently mentioned changes involved the results page (larger plots, clearer output file locations), the parameter selection screen (less cluttered, more readable), and some suggested changes in visual design elements (less rounded corners for a more scientific feel, different color scheme, more professional step indicators). Other suggestions included hiding side panels by default, adding a dark mode, improving accessibility with text-to-speech, making clickable elements bold, adding the mode selection side panel to the mode selection page, and providing a more detailed tutorial.
13. **How did the GUI compare to other programs you have used previously?** Participants who had experience with other scientific tools generally found the RAiSD-AI GUI simpler and more intuitive in comparison. Multiple participants noted that it lowers the entry barrier compared to other, more complicated, scientific software. The simplified and clean interface was seen as an advantage rather than a limitation, with participants appreciating that it made a complex task accessible. Some participants had no comparable experience to draw from.
14. **What final comments do you want to make before ending this interview?** Most of the participants did not have additional comments. Those who did offered encouragement and expressed positive impressions. A notable final suggestion was to automatically close the left panel after mode selection step is complete.

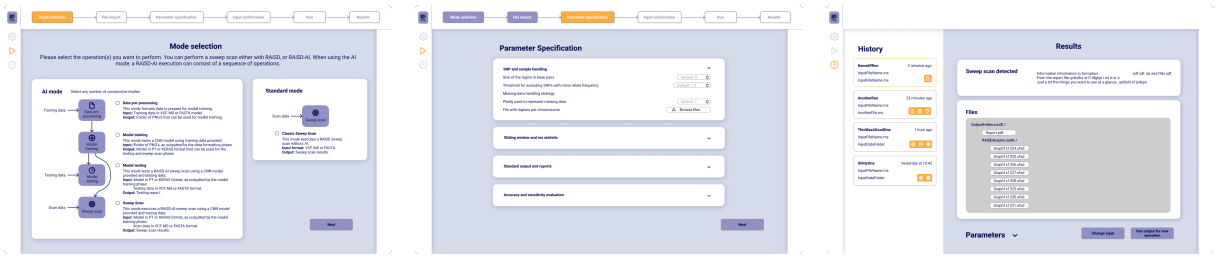
General Conclusions To conclude, the user testing sessions have revealed that the GUI design was generally well received in terms of structure, visual design, and step-by-step workflow. The navigation bar at the top, the help tutorial, and the progress visualization were appreciated by the participants as effective features. The most significant usability issue identified was in the mode selection step, where multiple participants struggled to grasp that operations such as model training and sweep scan could be run in combination at a single execution. Additionally, the naming difference between "Operation modes" and "Mode selection", between the side panel and the main page, added to this confusion. After this, the parameter specifications page was the most commonly criticized, as participants found it too text-heavy and suggested collapsible sections, separation into different pages, explanatory hints for all parameters. The results page received the feedback that plot sizes were too small for a detailed analysis.

4.6 Revised Mock-up

4.6.1 Design

Based the feedback gathered in the first round of user testing, we revised our mock-up, and created new designs for pages that needed improvements. These improved designs can be seen in figure 11. This section will highlight the changes that were made based on

the feedback, and thereby show the revised mock-up that was user tested again.



(a) Mode selection (b) Parameter specification (c) History page

Figure 11: Revised mock-up design

Mode selection The mode selection page was one that, according to feedback from the user tests, needed much improvement in terms of clarity. To do so, we made several changes as seen in Figure 11a. To begin with, we removed the side panels from all pages, which greatly decreased the amount of visual clutter and overwhelm. The history was moved to a separate page, accessible through an icon on the left sidebar. Additionally, since the operation selection side panel was only useful during the first step, operation selection, we moved it to the page designated for that. As such, the connection between the clickable list and the blocks of the diagram became a lot clearer. Additionally, based on feedback from expert RAiSD-AI users, the diagram was changed slightly. To clarify the separation between RAiSD and RAiSD-AI, and thereby the standard sweep scan and the AI sweep scan, they were moved to separate containers, beside each other rather than after each other. Lastly, we added text to clarify that it is possible to select multiple operations for a single run. Through these changes, we improved the connection between the diagram and the text and clarified which operations can be run together.

Parameter specification The parameter specification page shown in Figure 11b was improved by making the separate sections collapsible. Because of this, when a user opens the page, most of the content is hidden and the page is much less overwhelming. Users can open and close sections as they wish, and optionally an "Expand all"/"Collapse all" button could be added.

History User testing participants found the history feature useful, but would like the ability to close the side panel. This was because the persistent side panel was not necessary during most of the process, and instead provided visual clutter. Thus, we removed the side panel and moved the history to a separate page, which can be seen in Figure 11c. The history page contains a history list similar to the initial mock-up. When a user clicks on an item in the list, more information about the run is shown in a results view, similar to the normal Results page. This results view was changed to have a list of files, rather than the contents of the files. We chose this because runs can have thousands of files as output, and showing those all would be too much. The list shows the user an overview, and when an item in the history is double clicked, it is opened and shown in more detail.

4.6.2 User Testing

The revised design was tested again to ensure that it better suited our intended users' wishes. We focused on experienced RAiSD-AI users because they could provide us with

more insight into regular use of the system.

Testing Setup The user test for the revised design has been conducted in the form of a semi-structured interview with a PhD student in biology. This user had extensive experience with RAiSD-AI, thanks to utilizing the tool during their master's thesis. During the interview the participant was introduced to the group and the project and subsequently answered some questions regarding their experience with RAiSD-AI. Afterwards, the user was shown a walkthrough of the GUI using a combination of the initial and the revised versions. Lastly, the participant was asked questions to gather their feedback on the GUI layout.

The complete questionnaire can be seen in the Appendix A.

Conclusion of the User Test for Revised Design The information gathered during this user test was very useful to understand what makes for an user-friendly interface of RAiSD-AI. For example, the participant explained that much of their normal use was repetitively performing scans for selective sweeps, and tweaking the parameters. The participant gave much positive feedback. They stated they would use the team's tool over using RAiSD-AI through the terminal. Additionally, they liked the way the workspace worked and the mode selection, input confirmation, history and settings pages. On the other hand, the participant also provided some constructive negative feedback that provided additional insight for the team. For file selection, they mentioned that some operation mode can require more than one file as input. Although they liked the parameter selection page, they recommended adding a help-menu that appears for each parameter when you hover over the its text. The participant also suggested to display the command line output of the GUI to the user, such that if a user wants to, they can run it themselves. Moreover; for the results display page, the participant recommended to convert the hard to read .txt files into more readable .csv files. The participant also recommended serving the UI as a website instead of having a local app. Their reasoning was that biologists already need to download a lot of tools while conducting their work and making the tool a website would increase the reach of the tool by making it easier to access for biologists. Overall, the second user test for the revised design provided the team with useful insight regarding the usability of the tool and further improvements.

5 Implementation considerations

Before realizing the envisioned system design, several important aspects of implementation had to be carefully considered. This section reports on the decisions that were made, as well as the rationale behind each.

5.1 GUI framework

Given that the main goal of the project is to develop an intuitive, user-friendly GUI, the choice of GUI framework undoubtedly has a bearing on the development process, the usability of the final product and the maintainability of the codebase.

As a team, we researched a selection of available frameworks, weighing their benefits and drawbacks to find the one most suitable for the project. This subsection lists what we took into account for each of these frameworks.

5.1.1 PySide6/PyQt6

PySide6 and PyQt6 are both wrappers around Qt, a cross-platform application development framework, in the Python programming language. They are very similar to one another, having nearly identical syntax and feature sets. Notable differences between the two are the more permissive license of PySide6, and the official support it receives from the Qt Group.

Advantages Both PySide6 and PyQt6 rely on the mature and feature-rich Qt framework, which allows applications to be seamlessly deployed on Linux, Windows and macOS. The Qt ecosystem comprises various developer tools, such as the WYSIWYG[1] editor Qt Widgets Designer. This tool can ease the transition between prototyping software (e.g. Figma) and the actual implementation.

Disadvantages Using Qt through Python wrappers introduces an additional abstraction layer compared to native C++ Qt applications. While it is generally not significantly problematic for typical GUI workloads, this may result in slightly lower performance.

5.1.2 QT with C++

Qt can be used directly in C++, its native language. Using C++ for the development of the app means that the result will be a fully native desktop application that doesn't require Python bindings.

Advantages A C++ implementation offers high performance and minimal runtime overhead as the application is compiled into a native binary. Qt provides mature tooling and strong cross-platform support. This approach enables tight integration with system resources and full access to Qt's native feature set.

[1] "What You See Is What You Get"

Disadvantages Developing in C++ increases implementation complexity and development time compared to Python. Additionally, memory management and debugging are more demanding. The development team is more experienced in working with Python compared to C++ as well.

5.1.3 GTK

GTK is an open source graphical toolkit widely used in Linux environments, particularly within the GNOME desktop ecosystem. Python bindings are available through PyGObject, enabling GTK based applications to be developed in Python.

Advantages This system integrates well with Linux systems and provides a native look and feel, especially on GNOME-based distributions. It is open source and free of licensing concerns. For an application targeting Linux users, GTK can be a viable and stable option.

Disadvantages The tooling ecosystem surrounding GTK in Python is less extensive and less beginner friendly compared to Qt. Visual design tools comparable to Qt Designer are less mature. Since the design of RAiSD-AI GUI is nearly fully custom, the native feeling of GTK will not bring significant advantages as well.

5.1.4 Conclusion

After considering the alternatives listed above, it was decided that PySide6 would be the preferred framework for this project. The cross-platform compatibility, robustness of the Qt framework, and comprehensive set of tools for developing a modern and responsive GUI while maintaining a clean Python based implementation made it the preferred option considering the previous experiences of the development team as well. PySide6 is preferred over PyQt6 because of its official support from the Qt Group.

5.2 Distribution

In addition to selecting an appropriate GUI framework, careful consideration must be given to how the application is distributed to end users. Since the RAiSD-AI GUI depends on external libraries and the RAiSD-AI executable itself, effective distribution must ensure reproducibility, ease of installation, and compatibility across different Linux distributions.

5.2.1 Conda

Conda is a widely used package and environment management system. It includes a lot of packages and it allows applications to be distributed together with a fully specified set of dependencies defined in an `environment.yml` file.

Advantages Conda provides strong dependency resolution and is well suited for our use case. It allows reproducible environments and simplifies the installation of complex libraries.

Disadvantages Conda installations can be relatively large and may introduce additional overhead. Users must have conda installed before creating the environment, unless additional configuration steps are automated.

5.2.2 Miniconda

Miniconda is a miniature installation of conda that includes only conda, Python, the packages they both depend on, and a small number of other useful packages. It provides the core functionality of conda while being more lightweight.

Advantages Miniconda allows creation of reproducible environments while requiring less initial installation size compared to the full conda distribution.

Disadvantages Miniconda still requires a separate installation step prior to environment creation.

5.2.3 Micromamba

Micromamba is a lightweight reimplementation of Conda written in C++. It provides similar functionality while being significantly smaller and faster.

Advantages Micromamba is lightweight compared to conda and miniconda. It does not require a pre-installed conda distribution and offers fast environment creation. It is well suited for automated installers and can be bundled directly with the application setup process.

Disadvantages Although largely compatible with conda environments, micromamba has a smaller user base and less documentation. In some cases, debugging environment issues may require additional familiarity with conda style dependency management.

5.2.4 PyInstaller

PyInstaller is a tool that packages a Python application and its dependencies into a single executable. It packages the Python interpreter with required libraries.

Advantages PyInstaller enables distribution of the GUI as a single executable, simplifying the installation process.

Disadvantages Applications can become large, especially when including frameworks such as Qt. Additionally, PyInstaller only bundles the GUI's Python dependencies and does not manage external build requirements needed for compiling RAiSD-AI, limiting its suitability for this project.

5.2.5 Conclusion

After evaluating the available distribution options, an environment-based approach was selected as the most feasible solution for the RAiSD-AI GUI. RAiSD-AI requires specific dependencies and custom compilation, in addition to those needed for the GUI itself, making PyInstaller insufficient on its own. While micromamba is preferred during the

development for its lightweight nature and suitability for automated installations, the client has requested support for conda due to its widespread adoption. Fortunately, since micromamba and conda environments are interchangeable and use the same storage format, both will be accommodated.

5.3 Security

Security is a critical part of software development and it should be ensured that our application is secure by design. To achieve this, a structured threat analysis is performed to identify what threats the application faces and how these can be mitigated. This allows for informed decision making in the subsequent design, development, testing, and post-deployment phases [6][7].

5.3.1 Scope

The scope of the assessment is the application as a whole. This includes the GUI and how it uses RAiSD-AI. However, this does not include RAiSD-AI itself.

The application is assumed to run locally on a user's machine. Authentication and authorization are handled by the host operating system and are therefore considered out of scope.

5.3.2 Threats

The STRIDE model is used to systematically identify threats. One important factor to note is that the source code of the application is open source. This increases the risk of an attack that exploits a weakness in the source code.

Spoofing/Authentication The application does not have user accounts or other forms of authentication mechanisms. The host operating system could be compromised, giving an unauthorized user the ability to execute the application or access data used or generated by the application. This is an accepted risk, since the responsibility of authentication is delegated to the host operating system.

Tampering/Integrity While a malicious user of the system could change or modify persistent data stored by the application, this falls outside responsibility of our application. As stated before, we assume that the system has handled the authentication and authorization of the user correctly.

Correctness does fall within scope of our application. The generated data must be accurate, consistent and trustworthy. Next to this it is important that the application does not corrupt data and does not alter data on the system that it shouldn't.

Repudiation/Non-Repudiation The application does not maintain logs of user actions. This means that users cannot prove or trace performed actions. However, this is not a requirement for our local, single-user application.

Information Disclosure/Confidentiality The application processes data locally and does not transmit data over a network. However, sensitive data might be exposed through certain files. It is inside the scope of our application to prevent this.

Denial of Service/Availability Although the application runs locally, it can still become unavailable due to multiple reasons. For example, it can crash due to certain input, use an excessive amount of memory or CPU, or freeze because of large files. It is part of the responsibilities of our application to handle this.

Elevation of Privilege/Authorization The application runs with the privileges of the user that uses the application. If the application executes unsafe code, it could perform unintended actions within the user's privilege level. It is the job of our application to prevent this from happening.

5.3.3 Mitigation

To mitigate the identified threats, the application has to incorporate several measures to ensure security. The application has to only interact with user-specified input and output files. To prevent unintended data loss, users have to be prompted before overwriting existing files. Additionally, input validation has to be implemented to reduce the likelihood of crashes and unintended results. Error handling has to be implemented to ensure that failures are handled gracefully and communicated clearly to the user.

In addition to preventing and handling incorrect behavior, the application must be designed to avoid unnecessary exposure of sensitive information. Data has to be processed locally and not be transmitted over a network. Any files generated by the application have to be created at user-defined locations.

However, the responsibility of our application has boundaries. It assumes a host system that correctly enforces authentication. Additionally, the handling of potentially malicious input files processed by RAiSD-AI is considered outside the scope of this project. As such, any consequences arising from the execution of RAiSD-AI on unsafe data are not the responsibility of this application.

6 Final design and implementation

Although the initial design discussed in Section 4 provided a good scaffolding for the GUI application, many aspects surfaced during the implementation process which had an effect on various aspects of the project. This section discusses the details of the final design and implementation, placing particular emphasis on the points that differ from the initial design.

In accordance with the initial design, the classes are divided according to the Model-View-Controller architectural design pattern. The model classes contain the logic of data manipulation and checking for validity, while the view-controller classes display the underlying data to the user. This separation of concerns leads to high code readability and reusability—multiple views can be instantiated for the same model object and all of them will update when the referenced object changes.

As is customary in web and UI frameworks, the code is event-driven: the user's interaction with a visual component triggers an event on that component, which is broadcast to all objects that have previously subscribed to it. This is the Observer behavioral design pattern, implemented in Qt through the Signal and Slot mechanism. To make use of this feature, most classes in the application inherit either from the `QObject` base class or a more specialized subclass of it.

6.1 Run records

The `RunRecord` class is at the core of the GUI's functionality, as it records all details about a particular RAiSD-AI run. Namely, a run record holds the selected RAiSD-AI operations, the input files, the values of all parameters and, once the run has been completed and details about the success status.

Figure 12 depicts a portion of the system's class diagram centered around the `RunRecord` class. Each `RunRecord` instance holds references to one or more `OperationTree` objects, of which one is selected at any given time. The `OperationTree` class is presented in more detail in Section 6.2. A RunRecord also contains a number of `ParameterGroup` objects, which are containers for related parameters more closely inspected in Section 6.3.

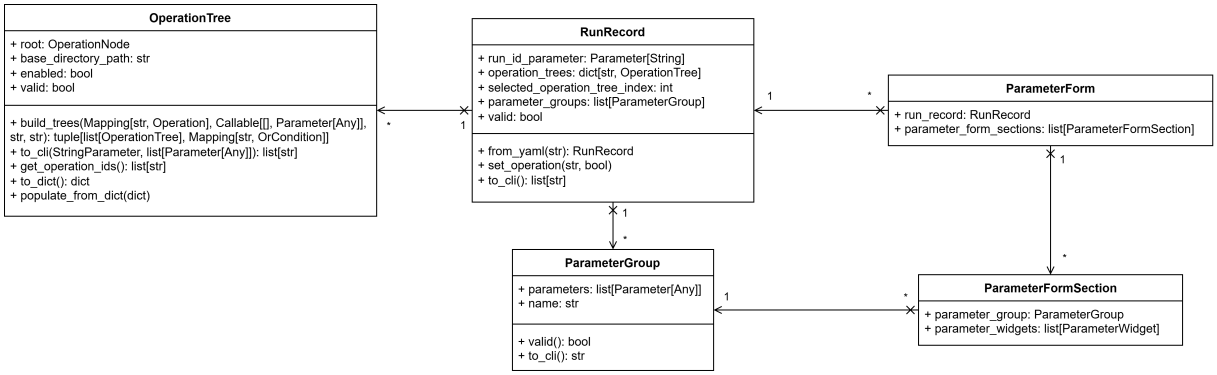


Figure 12: Class diagram—Run Record

Given the high amount of information stored by a run record, no single UI component displays one in its entirety. Instead, different view classes take on the responsibility of rendering certain parts of the data stored in the run record: the `OperationTab` displays

the run ID parameter, selected operations and input files; the `ParameterForm` displays the values of parameters; finally, the `ResultsWidget` displays the success status and output files of an execution on the `ResultsTab` and `HistoryPage`.

The operations and parameters in a run record are not statically defined in the source code, but instead loaded from a configuration file at runtime. Section 6.7 describes this procedure in more detail.

6.2 Operation selection

One important characteristic of typical RAiSD-AI workflows is running multiple operations one after another and using the output of one as input for the next. The initial interface design included a simple checkbox for each existing operation, but during the development process it became clear that such a structure is inadequate for fulfilling the users' needs.

By taking into account the relationship between one operation's output and another one's input and the fact that there is no cyclic structure within this relationship, we found that RAiSD-AI workflows can be represented through a tree structure.

A diagram of the classes involved in the arborescent representation of operation selection is presented in Figure 13. As mentioned in Section 6.1, a run record holds a list of operation trees, along with the index of the currently selected tree.

Operation trees An operation tree, encoded by the `OperationTree` class, has an `OperationNode` object at the root. The tree is structured in alternating layers, where every second layer—beginning with the root—is made of `FileProducerNode` objects, while in between are layers of `FileConsumerNode`s. Below, the specifics of these classes are discussed.

File producers The `FileProducerNode` class is an abstract base class for nodes which can provide a path to a file, either by having the user select it on the file system or by running one or more operations to generate it. A file producer node advertises the kind of file it will provide upon running the commands generated by its `to_cli` method, and provides properties to check the node's enabled status and the path to its output file. This class has three subclasses, all of which serve a specific purpose.

Operation nodes The first subclass of `FileProducerNode` is `OperationNode`, which holds all information related to an operation instance. The output file path of an operation node is constructed dynamically, in order to match the predicted output location of the corresponding RAiSD-AI operation. An operation node also holds an overwrite confirmation parameter and an arbitrary number of other parameters as part of the operation it represents, and has a `FileConsumerNode` child for each input file required by the operation. The validity of an operation node is tied to the validity of its parameters and child nodes, as well as to the overwrite parameter being set if an overwrite is detected.

File pickers The `FilePickerNode` class is a second subclass of `FileProducerNode`. It corresponds to the user selecting an input file already present on the system. A file picker node is a leaf node, since it requires input from no other node to produce its file.

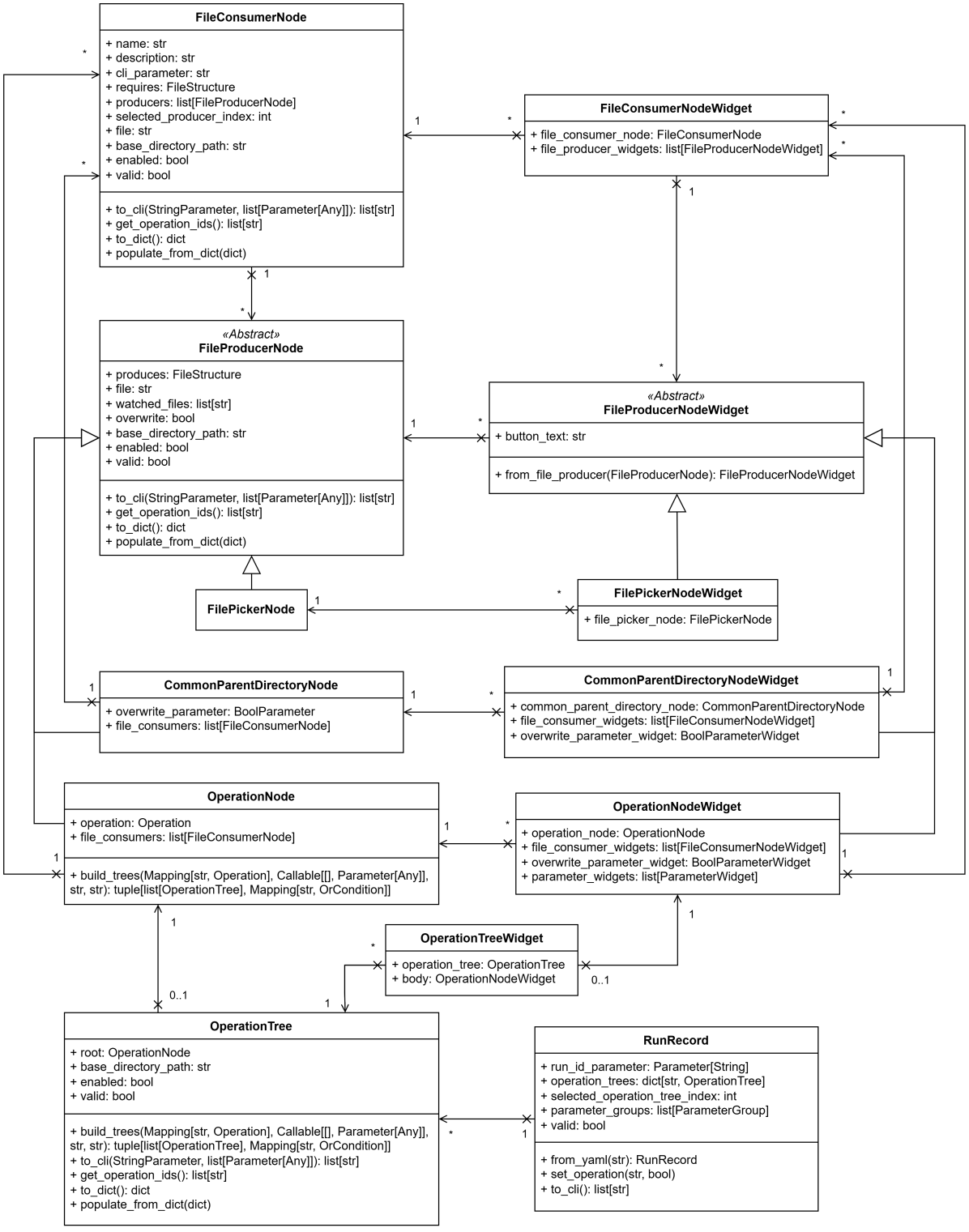


Figure 13: Class diagram—Operation tree

Its output path is the path selected by the user. Validity is assessed by checking whether the path points to a regular file or a directory, based on the required type.

Common parent directories Finally, the typical RAiSD-AI workflow presents one more challenge not properly addressed by the aforementioned `OperationNode` and `FilePickerNode` class. Namely, the expected input of the "Model training" and "Model testing" operations consists of a directory produced by two consecutive executions of the "Data preparation" operation. This relationship is modeled through the `CommonParentDirectoryNode` class, which represents an intermediate node that can be instantiated when a directory of files is required. The common parent directory node has a `FileConsumerNode` child for each file in the directory structure it requires. Its validity depends on the validity of each of its children and whether their output is located in a common directory. If that is the case, that common directory is also the output path of the node.

File consumers As previously explained, `FileConsumerNode`s exist as an intermediate between some `FileProducerNode`s and the descendants which can provide their input. In many instances, there are multiple possible ways of obtaining the necessary input files. In this context, a `FileConsumerNode` is responsible for holding the currently selected file producer.

Figure 14 provides an example of how an operation tree is built. While the example showcases the construction of a tree for the Model testing operation, the same working principles would apply if considering any other operation as the root.

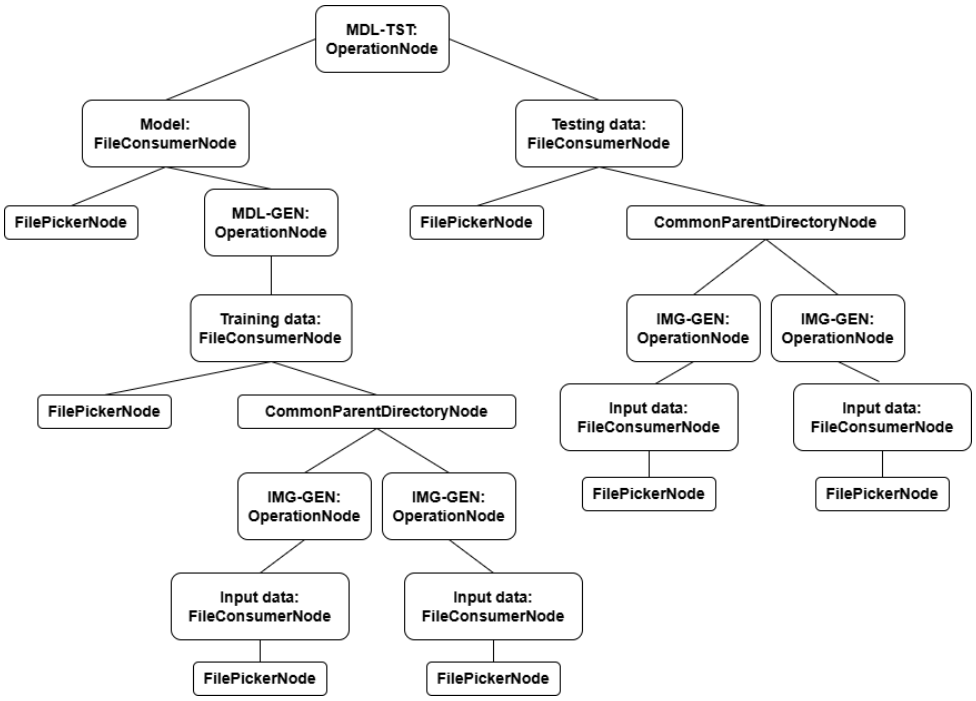


Figure 14: Sample operation tree—Model testing

Initially, the root node is created for the "Model testing" operation. This node has two `FileConsumerNode` children, one for each required input file: the Model and the Testing data. The Model can be provided through a `FilePickerNode`, which means

that the user selects a directory from their file system, or by running another operation. The specific operation which can generate the Model is determined by comparing the input and output file structures of the two operations. In this case, a match is found with "Model Generation". The single child of "Model Generation" is the Training data `FileConsumerNode`. This, in turn, can be obtained either from a `FilePickerNode` or be a directory in which the output of two "Image generation" operations will be stored. In the latter case, for each of them, there is a `FileConsumerNode` for the input data. In an almost identical way, the tree is build for Testing data.

In the corresponding `OperationTreeWidget`, the user has a choice of input at each step, and depending on their selection the suitable subtree is displayed. Figure 15 shows the operation selection tab in use.

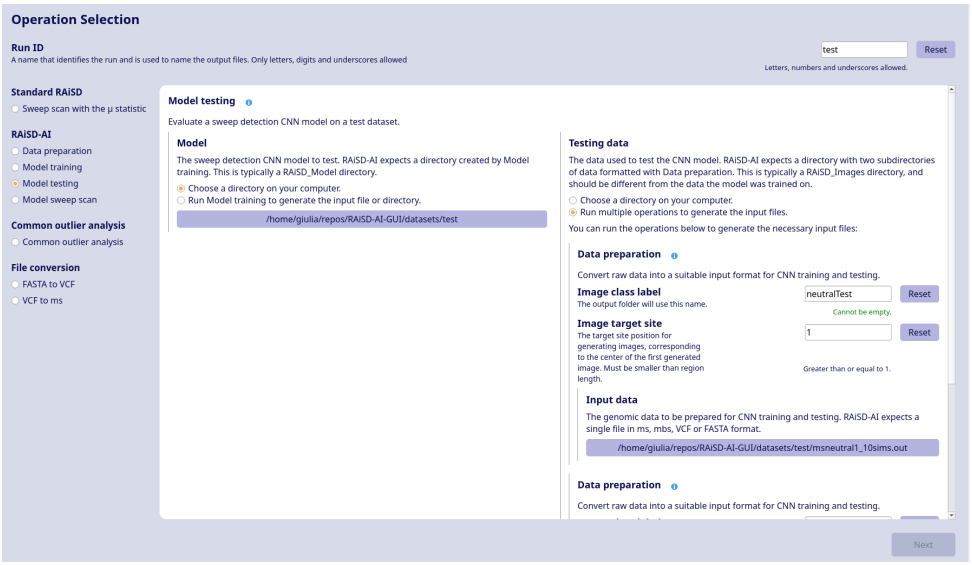


Figure 15: Operation selection tab

6.3 Parameter input and confirmation

Correctly modeling parameters is central to the GUI application's goal—one of the main purposes of the project is to enable researchers to fill in the parameters of RAiSD-AI with ease, without relying on the command line. While the initial design for the parameter class hierarchy generally held up, new parameter types had to be introduced after closer analysis of the usage of RAiSD-AI, and common functionality was factored out of subclasses where possible.

An overview of the classes involved in the parameter model is found in Figure 16. The rest of the section is dedicated to documenting the `Parameter`, `Condition` and `Constraint` class hierarchies and their usage.

Parameter The generic `Parameter` class centralizes the common functionality of all parameters, and the classes for specific parameter types inherit from it and override the required functionality. Each parameter object holds a textual name and description, as well as a command-line flag, a set of operations it belongs to and a default value. All of the aforementioned values are parsed from the configuration file upon running the application, ensuring dynamicity in their definition. `ParameterWidget` is the base class

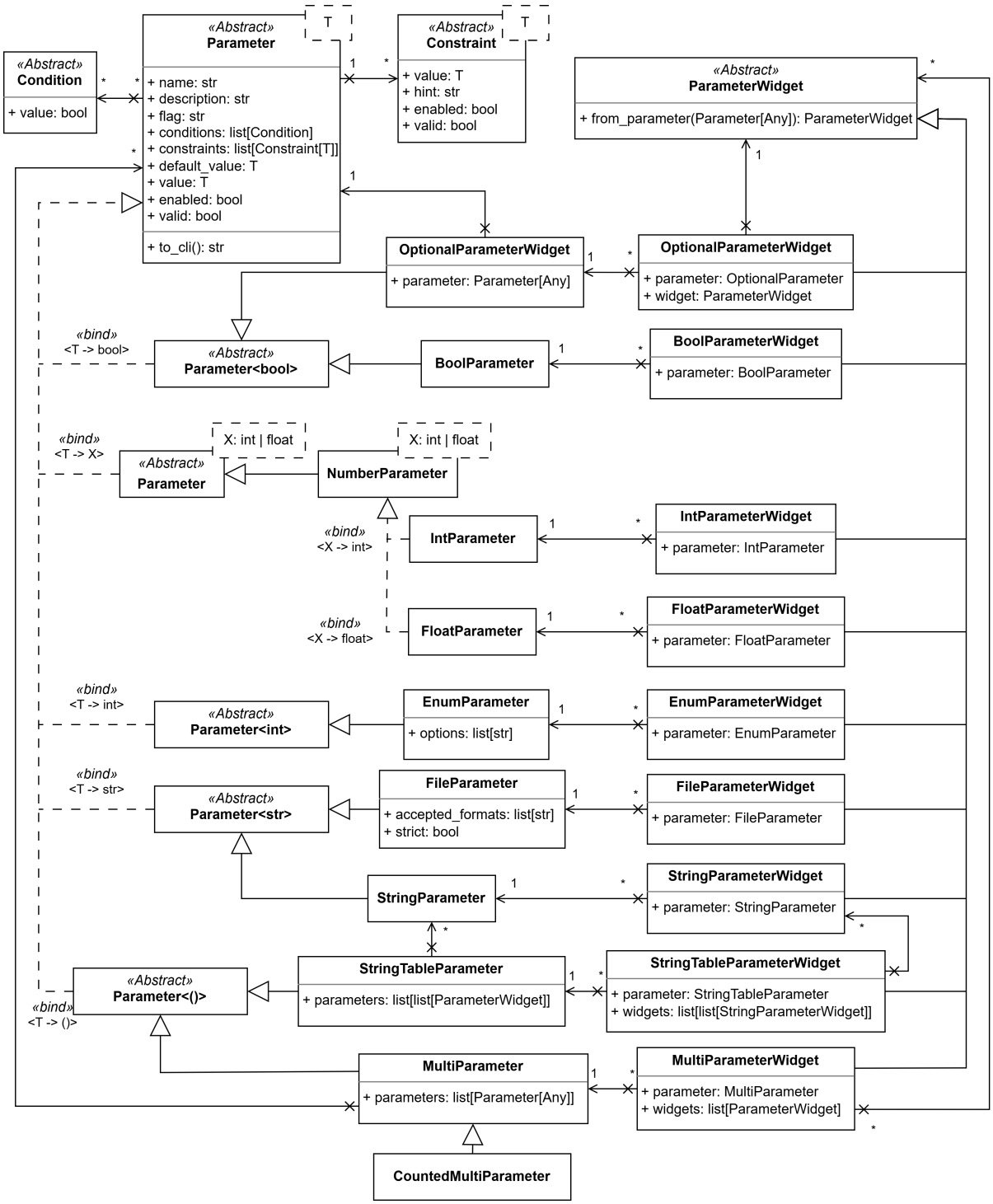


Figure 16: Class diagram—Parameter

for the UI components used to edit parameters, and instances of its subclasses can be constructed by the `from_parameter` factory method.

Boolean parameters The simplest kind of command-line parameters are those which take no value, but rather affect the behavior of RAiSD-AI if they are present. The `BoolParameter` class models the presence or absence of such a flag as a boolean value, and it is exposed to the user as a checkbox through the `BoolParameterWidget` class. A boolean parameter is represented in the command line by its flag if the value is set, or an empty string if it is not.

Number parameters Many parameters of the RAiSD-AI tool have an integer or floating-point value. Since the behavior of the two types is almost identical, the two classes `IntParameter` and `FloatParameter` inherit from the intermediate class `NumberParameter`. The UI components `IntParameterWidget` and `FloatParameterWidget` are used to allow the user to fill in values, and the command-line representation is formed by concatenating the flag with the value.

Text parameters Parameters which allow the user to provide a string are modeled using the `StringParameter` class and displayed in a `StringParameterWidget`. Similarly to number parameters, the flag and value are concatenated to form the command-line representation.

Enumerated parameters There are certain options of the RAiSD-AI command-line tool where only a fixed set of values are allowed. In those cases, the `EnumParameter` class is used, which holds a list of user-facing values associated with the form they take in the command line. An `EnumParameterWidget` contains a drop-down menu to allow the user to choose between the options. The selected option's command-line representation is consequently appended to the parameter's flag.

File parameters Besides the main input files of the operations, a path to a file can also be provided as a parameter to RAiSD-AI in certain contexts. To aid the user in providing a valid file path, the `FileParameter`, which holds a reference to a file, presents a file browser window in the user's operating system through the associated `FileParameterWidget`. The path of the file is concatenated to the flag in the command-line representation.

Optional parameters For parameters which provide a value, but whose absence leads to different execution behavior (e.g. `-T`), it is necessary to provide the user with the option of using or not using them, as well as prompt them for the required values if they choose to use them. This is represented using the `OptionalParameter` class, which internally holds a reference to another parameter. The optional parameter itself has a boolean value, which conditions the enabled status of the inner parameter—the contained parameter can only be displayed if the user has opted to use it.

Multi-value parameters Certain RAiSD-AI parameters must include multiple values, such as the `-B` parameter with two integers. A `MultiParameter` holds a list of inner parameters, which can be of any type. Their widgets are displayed in a vertical layout in

the `MultiParameterWidget`, and the command-line representation is built by concatenating those of the inner parameters and prepending the flag. A special case of the multi-value parameter is `-l`, which also includes the number of values after the flag. To accommodate this, the `CountedMultiParameter` class overrides the command-line representation method while still being displayed through the same widget class.

String pair parameter Another parameter which required special attention is `-clp`, which allows the user to provide pairs of strings. This led to the introduction of the highly specialized `StringTableParameter` and corresponding `StringTableParameterWidget`, which allow the user to choose the number of pairs before entering values for each string. The command-line representation is constructed following the rules of RAiSD-AI.

The behavior of parameters may be altered by conditions and constraints, both of which are described in detail below.

Conditions The enabled status of a parameter—which dictates whether it is shown to the user, checked for validity and used in the terminal command—is often determined by several external factors. The `Condition` base class stores a boolean value associated with such an external factor it tracks, and emits a signal when the value changes. Conditions can be added to parameters by calling the `add_condition` method, which makes the parameter's enabled status conditional on the respective external factor. The paragraphs below list the existing condition types and the motivation behind each.

Operation condition The immediately evident use case for the condition system is for enabling parameters when one of the operations they are part of is selected, and disabling them when none are. This check is performed by the `OperationNode.EnabledCondition` class.

Overwrite condition Overwrite confirmation parameters should only be offered to the user if the output path of an operation points to an already existing file. The class `FileProducerNode.OverwriteCondition` tracks the overwrite status of an operation for this purpose.

Parameter enabled condition Occasionally, it is helpful to involve a parameter's enabled status when determining the status of another, especially when complex composite conditions are involved. For this reason, the `Parameter.EnabledCondition` class is made available.

Parameter value conditions It is often the case, when working with RAiSD-AI, that a parameter is only allowed—or is required—when another parameter is given, or has a certain value. This is modeled through `BoolParameter.Condition` (tracking the value of a boolean parameter), `OptionalParameter.Condition` (tracking whether an optional parameter is used) and `EnumParameter.Condition` (tracking whether the value of an enumerated parameter is in a given list).

Conjunction/disjunction condition To allow for constructing more expressive conditions, the application allows taking the conjunction or disjunction of a number of existing conditions through the `AndCondition` and `OrCondition` classes, respectively. The former is also used internally by the `Parameter` class to accumulate conditions that are added.

Constraints Besides the enabled status of a parameter, its validity can also be subject to various factors. Restrictions such as a minimum and maximum value for numbers are common, as are a minimum length or a regular expression for strings. Such requirements are denoted by `Constraint` objects, which are added to parameters through the `add_constraint` method. A constraint exposes the `hint`, `enabled` and `valid` properties, which are used in the `ConstraintWidget` class to give informative indications to the user. The final constraint system constitutes a significant shift from the initial design, in that the specific constraint logic is placed outside of the `Parameter` classes and into its own specialized classes. This change brings about an improvement in maintainability, as all possible combinations of constraints no longer need to be explicitly defined in a parameter type.

Min/max constraint A restriction on a number's minimum and maximum allowed values can be put in place using the `IntervalConstraint` class, which also permits specifying whether each of the lower and upper bounds is inclusive or exclusive.

Parity constraint The window size given to RAiSD-AI must be even. This can be enforced using the `EvenConstraint` class.

Length constraint A string's maximum length can be set using the `MaxLengthConstraint` class.

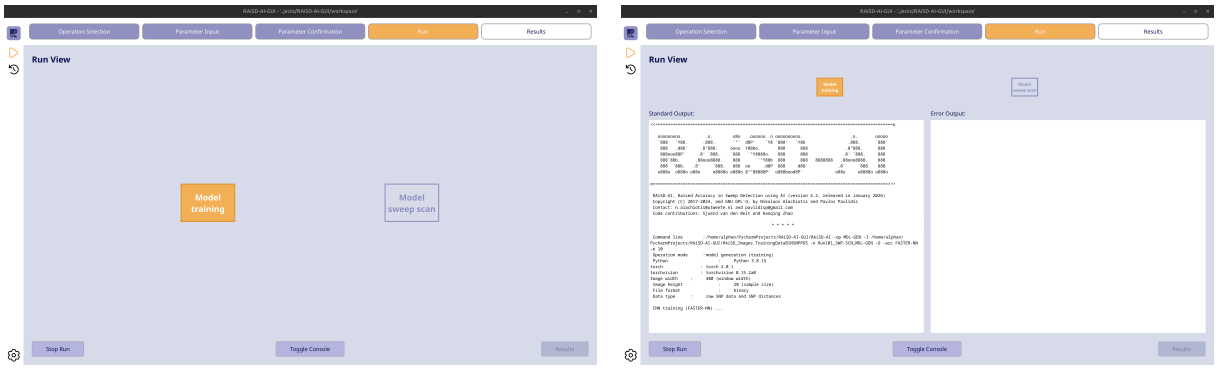
Regular expression constraint An arbitrary restriction can be placed on a string parameter's allowed values by adding a `RegexConstraint` object with a suitable pattern and hint.

Parameters are found within four parts of the run record. Firstly, the run record contains one run ID parameter, which has a bearing on several parts of the produced commands. Secondly, each operation must have an overwrite confirmation parameter, but may also define one or more parameters to be provided separately for each instance of that operation. Finally, the bulk of the parameters is found in `ParameterGroup` objects, which are visually mirrored by `ParameterFormSection` objects. These parameters are filled in only once for a given run, and automatically applied to all relevant operations.

Multiple `ParameterGroup` objects are stored in a single `RunRecord`, and they are visually shown in a `ParameterForm`. This widget combines the `ParameterFormSection` widgets in a vertical layout, and can either be editable or not. The two versions are used on the parameter input and parameter confirmation tabs, respectively.

6.4 Run view

The run view received several updates after the initial design and during development. The visual design can be seen in Figure 17. In this section, the design, functionality and



(a) Run view with consoles hidden

(b) Run view with consoles shown

Figure 17: Comparison of run views

implementation of the run tab will be discussed.

First of all, the run view looks as Figure 17a when the run starts. There are rectangles to display progress, with the name of the operation they represent inside them. In the initial design, the progress indicator was designed to be circular and to display partial progress. That is, give an indication of how much of the operation was done. However, this information regarding the progress of an operation is not available. Therefore, this feature is excluded and the progress indicators only display the overall progress of the operations. Furthermore, the circular design was replaced with rectangular indicators to allow for showing the corresponding operations. The `ProcessIndicatorWidget` creates progress indicators that each correspond to an operation of the execution that is being executed. These rectangles are initialized with a purple border with transparent background. When the corresponding operation is being executed, the indicator turns yellow. A purple background signifies a completed operation, a red background a failed operation, and gray a stopped operation.

At the bottom of the page, there are three buttons; `Stop Run`, `Toggle Console` and `Results`. While the execution is still running, the user can click the `Stop Run` button to stop the execution. After the execution is done, users are directed automatically to the results tab. From there, they can go back to the run tab using the `Back` button. To return to the results tab, they use the `Results` button. The last button available is the `Toggle Console` button. When that button is clicked, the output console and error console appear, making the run tab look as shown in Figure 17b. Here, the console output of RAiSD-AI is printed on the left console and any error output is displayed on the right console. Users have the freedom to resize the two consoles horizontally by dragging the separator between them to either side.

6.4.1 Command Executor

The run tab makes use of the `CommandExecutor` class to run RAiSD-AI. The `CommandExecutor` bundles commands together into "executions". These can be started and stopped with its two exposed methods, `start_execution` and `stop_execution`. `start_execution` takes a list of partial commands, only containing the command-line arguments. With each list item a command is built by prepending the environment activation and RAiSD-AI executable.

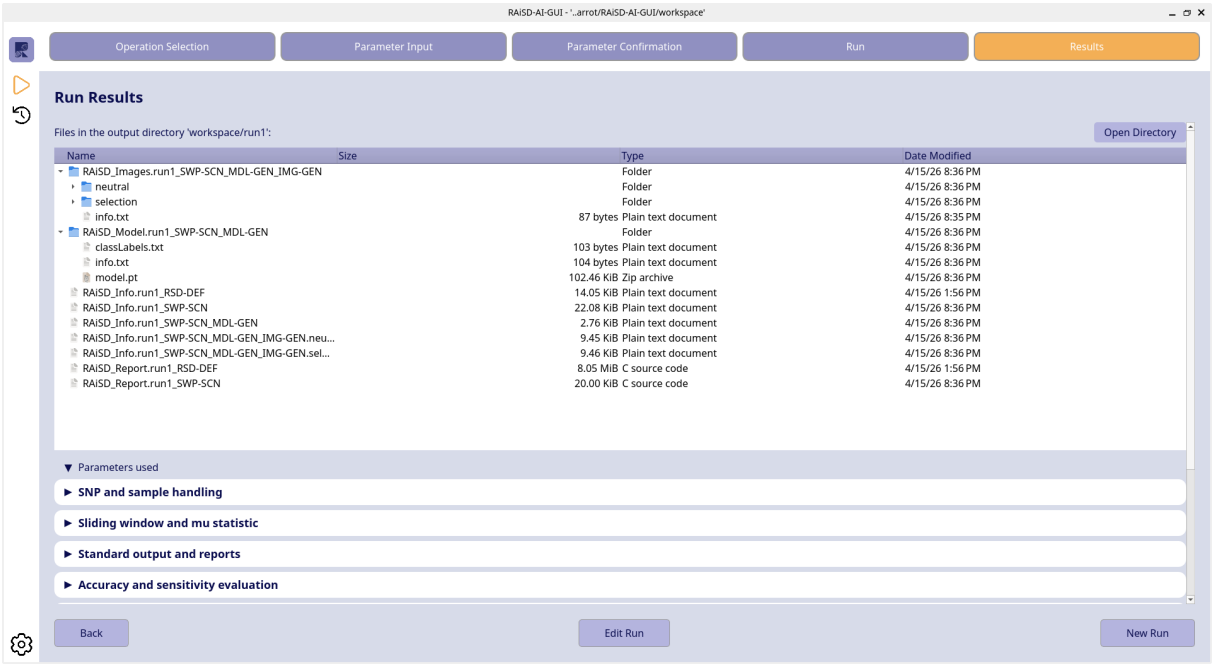


Figure 18: Results tab after a Sweep scan

Then, for each command a process is started inside the workspace folder. This process executes the command and finishes if the command is finished. If all processes finish successfully, the execution is successful. However, a process can also fail or be terminated. In that case the execution will also fail or be terminated. These exit states of the execution are broadcast using signals. These are used to determine what the GUI will show next.

While a process is running, the std_out and std_err are emitted with two signals. Each time a process exits, it's exit status is also emitted using signals that also carry the process index. This is used to visualize the progress of the execution with the `ProcessIndicatorWidget`s.

6.5 Results

The design of the results tab remained largely the same after the initial design. The front-end design can be seen in Figure 18. The data shown on the page comes from the data structure explained in Section 6.1, the `RunRecord`. This section will go through the elements on the page to explain their functionality and implementation.

Firstly, the view differs slightly depending on whether the RAiSD-AI execution finished successfully or if it was stopped or errored. In case of the latter two, a warning label appears, displaying the state of the execution to the user and what results they are viewing.

At the bottom of the page there are three navigation buttons that allow the user certain flows of actions. The `Back` button will take the user to the previous page in the execution process, the run page. The `Edit run` button takes the user to the operation selection tab and preserves the state of the `RunRecord`, such that the parameters and operations are still filled in. The `New run` button also takes the user to the operation selection tab, but instead resets the `RunRecord` and all pages so that the user can start from scratch with a new run.

The rest of the main layout, including a view of the directory with the results, and the parameters used, is created using a `ResultsWidget`, which is re-used in the history page.

6.5.1 Results widget

The results widget consists of two main parts. Firstly, there is the directory view. A header explains that the files shown are those in the output directory of the run, and includes the relative path from the workspace. To the right, there is a button that allows the user to open said directory in their default file browser. This allows them to make use of functionality like search or sort.

The directory is shown using a `QFileSystemModel` in a `QTreeView`. Visually, it mirrors what such directory views often look like in the native file browser, which helps the user understand what they are looking at. Each file can be double clicked to open it in the default file viewer.

The second part of the `ResultsWidget` is the list of parameters. This is made using a `ParameterForm` described in Section 6.3, similar to in the parameter specification and the input confirmation pages. The parameter form is locked, meaning that it is view-only, and parameters are not editable.

6.6 History

The history page is responsible for the history features of the GUI, which is an important added functionality of the GUI compared to the command-line RAiSD-AI tool.

The visual design of this page has remained largely the same as it was in the initial design. The left side of the page contains a list of items that represent runs successfully executed with the GUI. Each item is clickable, which opens the run's contents in the more detailed view on the right side. This consists of a `ResultsWidget` as described in Section 6.5.1.

On the bottom right there is a `Reuse` button, which takes the user back to the operation selection page, with the entire operation selection and parameter specification filled in from that run. This allows the user to execute a similar run, with some tweaked the parameters. This is similar functionality as that of the `Edit run` button on the results page (Section 6.5).

Compared to the initial design, the button that allowed the user to use the output of a past run as input for a new run was omitted. This decision was made because we found that the button left room for ambiguity. The output of a run can often be used as the input of a run in many different ways. Additionally, that input could then be incomplete and need to be supplemented with other input. Since there were so many options the button could lead to, we establish that it would confuse more than be useful. The requirement was therefore removed.

6.6.1 History classes

Figure 19 depicts the classes used to construct the history page. Similarly to before, the left side of the diagram is the model side, and the right shows the view. When a run is

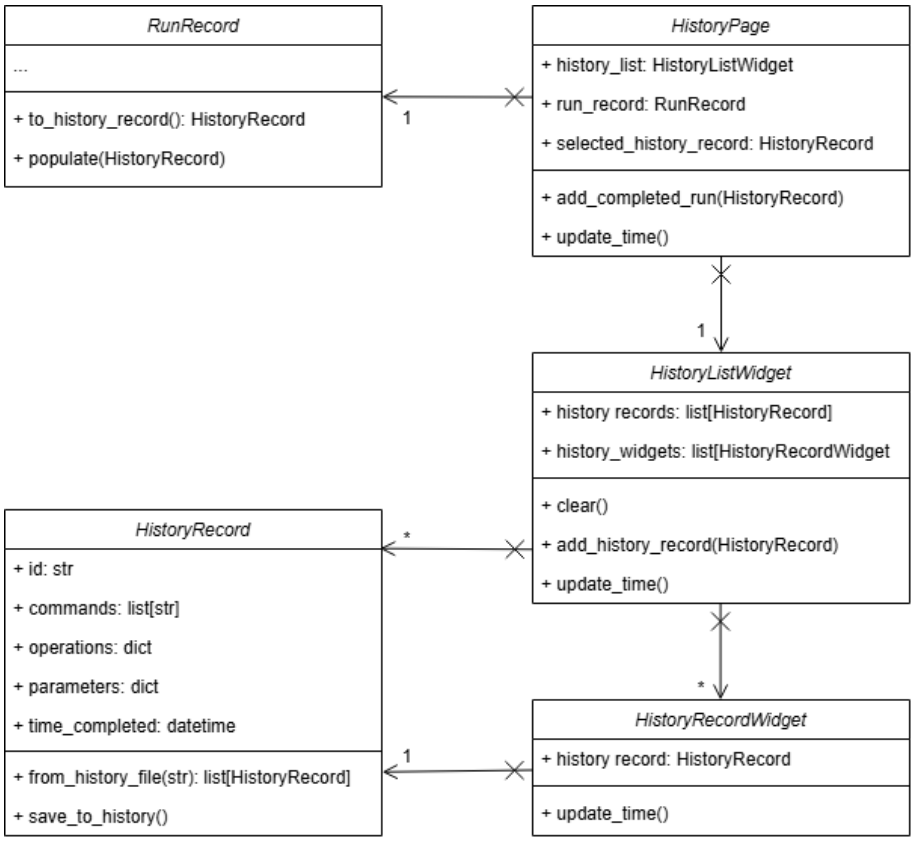


Figure 19: History class diagram

completed, its data is stored in a `RunRecord` object. This is made into a `HistoryRecord` through its `to_history_record()` function. That `HistoryRecord` class is the basis of the history page. It stores all necessary information about a run, but in a format closer to the history file. Thus, when a run is completed, the created history record is both saved to the history file and added to the history page.

For the view, a `HistoryRecord` is visualized through a `HistoryRecordWidget`. This widget shows the name of the run, as well as the amount of time ago the run was completed. Several of them are combined in a `HistoryListWidget`, which shows them in a vertical layout, and manages added or removed widgets.

When a history item is clicked, its `HistoryRecord` is used to populate a `RunRecord`. This includes setting the operation trees to the right values, and filling the parameters as they were. That `RunRecord` is used in the `ResultsWidget` where the details of the run are shown.

6.7 Configuration file

In order to anticipate future changes in the underlying RAiSD-AI tool, run records are not statically defined in the source code. Instead, the details of operations and parameters are loaded from a configuration file at runtime. This single source of truth for the GUI ensures that prospective modifications to RAiSD-AI—such as new parameters, or even new operations—can be effortlessly accompanied by corresponding updates in the graphical interface.

Parsing the configuration file is carried out in the `from_yaml` class method of the `RunRecord` class, which thoroughly checks all values for validity and reports errors with clear messages in the case of violations. The configuration file is expected to be in the YAML file format, and an overview of its structure is laid out below.

Run ID The name, description and command-line flag of the run ID parameter are defined in the root object of the configuration file. Its type is constrained to string parameter. This corresponds to the `-n` option of the RAiSD-AI tool.

Operations The configuration file subsequently defines the operations supported by RAiSD-AI. For each operation, the file contains its name, description, input files, output files, the path to the output location and the path to the info file that it will produce. This data is used to construct the operation trees, as explained in Section 6.2. Each operation also defines its own output overwrite confirmation parameter—of boolean type—and, optionally, parameters that are defined per operation instance rather than globally. This is specifically used for the `-icl` and `-its` parameters of RAiSD-AI, in order to present them on the "Operation selection" separately for each instance of the "Data preparation" operation.

Parameters Furthermore, most of the parameters of RAiSD-AI are defined within parameter groups. Each parameter can be augmented with conditions and constraints, in addition to specifying the operations to which it belongs, either at the parameter group or parameter level. Every parameter has a name, a description and a type. Based on the parameter's type, various other fields are expected in the object.

Common directory overwrite confirmation Finally, the configuration file also contains the description of the confirmation parameter for when a common output directory will be overwritten, corresponding to the `-frm` parameter.

6.8 Settings

The settings page allows the user to view and update the application's configuration. It is divided into two sections: one for the settings of the GUI, and one with additional information about the RAiSD-AI and the GUI. The current look of the settings page can be seen in Figure 20.

Each configurable setting is displayed using a `SettingsItemWidget`, which shows the setting's name alongside its current value. These settings also have a `Change` button that opens a dialog or file picker to update the value. When a value is updated, the widget's label refreshes automatically by connecting to the corresponding signal emitted by the `Settings` model.

The `Settings` class manages all configuration states and uses a `yaml` file to keep values persistent between different sessions. Each setting is exposed as a property, the changes are written to the `yaml` file and a signal is emitted to notify the rest of the application.

Workspace The workspace is a folder selected by the user that is used to organize the outputs from different RAiSD-AI runs. When a run is completed, its results are saved

inside a subfolder named after the run id. This makes runs easier to manage and also enables history functionality, which is described in its own section.

On startup, the application loads the `yaml` file and populates `app_settings`. If no settings file is found on startup, for example on first launch, a setup dialog appears asking the user to select their workspace. The other settings, the location of their RAiSD-AI executable, their preferred environment manager and their environment name are shown as well, with default values, so that the user has the option to change them. If the user closes the dialog without making a selection, the application uses the current directory as the workspace.

Figure 20: Settings page

6.9 Manuals

The system includes documents with information about how to install, use, and understand the application. For the user, these documents are there to clarify in case anything is unclear. For a developer, they aim to give insight into how to repair, extend or adapt the system. Both of these are done using multiple documents which we will now discuss.

6.9.1 General manual

The first document is the `README.md` file, which is already part of the `RAiSD-AI` repository. It outlines how to install and use `RAiSD-AI`. We have added a section which instructs the user on how to run the installation script which sets up either only the GUI, or the GUI with the environment of RAiSD-AI. After that, there is a small section that instructs the user on how to run the GUI and it links to the user manual for more information about the usage of the app.

Here are the links to the chapters we added to the `README` of the original project: Installing the GUI, Using the GUI, Developing the GUI.

6.9.2 User manual

The next document is the `gui/USER.md` file, the user manual. This file begins with some general information about the GUI. Then, there is an explanation of the complete interface of the GUI, informing the user about the functionality of each page and tab. This includes screenshots to support the explanation and show what the pages are supposed to look like. The user manual can be found here.

6.9.3 Developer manual

The last document is the `gui/DEVELOPER.md` file, the developer reference. This document explains how to change the configuration file which is used throughout the entire app for any information or structure related to RAiSD-AI. Next to this, the reference explains the structure of all packages that make up the `gui` package. Lastly, it explains how to test the application. The developer reference can be found here.

7 Testing

To ensure the system works as intended and does not contain bugs, we have tested it thoroughly and regularly. This section provides an overview of the test strategy for RAiSD-AI GUI, divided into functional and non-functional testing parts.

7.1 Non-functional testing

7.1.1 Usability testing

Usability testing was conducted in week 3 as user testing sessions with low fidelity prototypes. Several versions of mock-ups of the GUI were displayed to users, who received simulated assignments which involve navigating the interface and finishing certain tasks. The testing sessions are described detail in Section 4.5.2 and Section 4.6.2. After the testing sessions, both the feedback of the users and team's observations were used to understand the tool's degree of usability and make improvements to the final design.

7.1.2 Compatibility testing

The system has been tested on several Linux distributions to ensure compatibility, including Ubuntu, Xubuntu, and the Arch distribution CachyOs. On each distribution, the application was installed from scratch on a clean machine and tested thoroughly afterwards. We did this to ensure that not only the GUI, but also all setup files and functionality worked as intended on the different distributions. On all systems, the tool functioned as required, and did not show problems.

7.1.3 Performance testing

Performance testing was only carried out to a limited degree. On a Virtual Machine with a single core and 2048 MB of RAM, the GUI has no visible delay or performance issues. Additionally, high sensitivity, reactivity and stability, although appreciated, are not the most essential features, nor are they a part of our necessary requirements. Therefore, due to time constraints, we have decided to prioritize other forms of testing over further performance testing.

7.2 Functional testing

7.2.1 Unit testing

Each component of the model and the command executor was tested individually to ensure it works as intended in an isolated setting. These unit tests were implemented using the `pytest` testing framework. To ensure the isolated setting, any related components are mocked using `pytest`'s mocking functionality.

For the unit tests, we aimed to ensure that for each component, all functionality worked as expected. To ensure this, we used `pytest`'s coverage functionality, aiming for high coverage. Although this does not guarantee that everything is tested adequately, it gives some insight into untested parts of code. The coverage can be found in Table 3.

[2]The coverage for the RunRecord class is lower because this file contains the parsing logic of the entire configuration file. Although we do test this logic, testing it is very text heavy and the code has many

File	Coverage
history_record.py	100%
file_structure.py	100%
operation.py	100%
operation_tree.py	91%
condition.py	100%
constraint.py	86%
parameter.py	97%
parameter_group.py	100%
run_record.py	60%[2]
settings.py	100%
Total	85%

Table 3: Test coverage of model files

All testing code is inside the project directory, and instructions can be found inside the `README.md` file and the developer reference.

7.2.2 Integration testing

Unit tested components were tested together to validate their correct functioning while they interact with each other. This is done in a similar manner to our unit tests, using `pytest`. However, instead of mocking components outside the one we are testing, we instead focus on the interaction between those components.

For our integration tests, we have three test files. The first focuses on the structures that store the parameters. These include the `RunRecord`, `ParameterGroup` and `Parameter` classes. The second integration test combines the `RunRecord` with the `OperationTree` and all classes that it uses. Lastly, we focus on the functionality of the conditions, between parameters and operations.

7.2.3 System testing

The system as a whole was thoroughly tested, both during and after development. For this, we have written an extensive testing protocol that can be found in Appendix B. The protocol details the intended functionality of the entire GUI, and what outcomes should result from certain actions. The protocol uses the Black Box Testing technique, describing the functionality of the GUI without knowledge of the internal structures.

7.2.4 Regression testing

Throughout the process, each addition to the system was tested before it was accepted. For this, we made use of GitHub pull requests. For a pull request to be merged, at least two reviews were required. Because of this, each addition is tested by at least three people, the developer and two reviewers, which ensured that everything worked as expected and previously working functionality was not broken.

[2] control flow branches to check for invalid values. Not all of these branches are checked which leads to a lower coverage.

8 Evaluation and reflection

8.1 Requirements

At the start of the project, the requirements were divided into 4 different sections as it can be seen in Section 3. Functional and non-functional requirements were both further divided into necessary and optional requirements.

8.1.1 Functional requirements

Starting with the functional requirements, all of the necessary functional requirements that were specified during the requirements specification stage were fulfilled with one exception: the -pci parameter specified in the requirements is replaced by the -pcs parameter in the RAiSD-AI tool and in the GUI as well. Moreover, this updated requirement was also fulfilled. Conversely, from the optional functional requirements only "The system shall provide a way for users to access the locations of input and output files in the operating system's file browser." has been partially achieved. For output files, the GUI has a button that allows the user to view them in the default file browser of the OS. However, this functionality is missing for input files. The rest of the optional functional requirements could not be achieved due to unforeseen feature creep during implementation and time constraints.

8.1.2 Non-functional requirements

Additionally, all of the necessary non-functional requirements were also fulfilled. The optional non-functional requirements such as packaging the system as a stand-alone executable, complying with widely adopted accessibility guidelines and supporting localization could not be achieved due to time constraints. The last optional non-functional requirement, "The system shall support other popular operating systems, namely Windows and macOS." has also not been achieved. This is due to the fact that RAiSD-AI itself is currently only compatible with Linux. However, the GUI itself is portable and not operating system dependent, so if RAiSD-AI supports other operating systems it can be easily used without major changes.

8.2 Task division

Throughout the project, we divided the work between the members of our team. Much of the design process was a shared responsibility, and everyone contributed. In general, everyone contributed to the initial design and much of it was decided in collaboration. Additionally, we all user tested and contributed to the different forms of documentation, the report and the final presentation.

For the implementation, responsibilities were not separate, and instead we all worked on issues that overlapped. Good communication helped us make sure this worked well. If we divide responsibilities in a general way, below is a list of what each member focused on.

Loes Baart de la Faille Loes took responsibility for the interface design, including the mock-up and the poster. Additionally, she worked with Steef on the structure of pages of

the GUI, specifically the history, results and settings pages. Additionallly, Loes and Steef took on a large part of the testing.

Steef Broeder Steef worked with Loes on the structure of the pages of the GUI, specifically the run page and its ability to run RAiSD-AI. He also took responsibility for the setup scripts, manuals, and testing set-up. Additionallly, Loes and Steef took on a large part of the testing.

Taylan Kıncır Taylan took responsibility for the styling of all the pages in the GUI, including a custom splash screen.

Alphan Mete Alphan worked on the file parameter and the visual feedback that is given when parameters are invalid.

Vlad Negară Vlad worked with Giulia on a large part of the model and its connection to the configuration file, including the parameters, conditions, constraints and operation trees. Additionally, they made the operation selection page.

Giulia Tălău Giulia worked with Vlad on a large part of the model and its connection to the configuration file, including the parameters, constraints and operation trees. Additionally, they made the operation selection page. She also wrote the system testing protocol.

8.3 Conclusion

In conclusion, we successfully designed and built an adaptable, intuitive user interface for RAiSD-AI. Throughout the process, our team faced several challenges and reached several successes. This section will evaluate on those and the design process itself.

8.3.1 Challenges

One of the first challenges we ran into is that we did not have a good enough understanding of RAiSD-AI from the beginning. This led to changes to the design that could have been prevented if we had a better understanding of the system. For example, the operation selection page of our initial design turned out to be overly simple and not sufficient for the amount of combinations of operations that can be run. This meant we had to completely change our design for both the system and the UI after we had already implemented it. This was compounded by inconsistencies in the client's clarifications. It regularly happened that something that had already been discussed was reconsidered and the client came to a different conclusion. Although this is expected during a design process, it meant that we often had to change what we had done and discuss the same issue multiple times. Fortunately, the adaptable design based on a configuration file meant the application could be swiftly adapted to fit a change in parameters.

Additionally, once we had a well thought-out design, we focused a lot on the implementation of it. However, this meant that testing fell behind a bit. There was some automated testing, but this was often incomplete and/or outdated. Even though we were able to progress our testing later in the process, it would have been better if we were more thorough with it early on.

As a team, we also faced the challenge of differences between team members, both cultural and in personality traits. These differences often created tensions. For example, some group members valued structurally meeting in the morning and punctuality, while others preferred joining later. Similarly, our ways of giving and receiving feedback sometimes clashed. This was a difficult situation to handle, but we scheduled a reflection meeting to address it. This was a good step, and although we could not change our differences, it resolved much of the tension surrounding them.

Lastly, a challenge we experienced during this project was the fact that our client was also our supervisor. As a system designer, one must evaluate what the client proposes and try to find something that meets their actual goals, rather than what they say they want. In turn, a supervisor would provide an objective view, that is centered on how we handle such situations, rather than the client's opinion. We felt that that distinction was not clear for us, meaning that satisfying our client was a main part of satisfying the supervisor. For an external supervisor, the main focus would have been on the design process.

8.3.2 Successes

First, one structural success we experienced in our teamwork is that we had a clear work process. This included daily meetings and several methods to keep track of progress. This structure was very useful for staying on track with our planning and knowing what other team members are working on.

Additionally, we held weekly meetings with our client/supervisor. At the beginning of the module we set a fixed time frame during the week for this, and during the module it varied sometimes according to our availabilities. These meetings were highly appreciated because it gave our client/supervisor a good view on our progress and it gave us regular feedback to improve our system.

Another positive point of our design process is that we learned a lot. RAiSD-AI is a gene analysis tool, and given that none of us had any significant background in biology, this project taught us much new. However, even within computer science we have explored new ideas, such as how to make a system this extensible and adaptable.

Our last positive point is that our client is happy with the system we have built. The system works according to its requirements, and has two features very important to our client: the user-friendliness and the adaptability.

8.4 Future work

Due to the limited scope of the project, several promising directions remained unexplored in the project. This section aims to outline features that could be implemented as contributions to the open source project, or as a future Design Project at the University of Twente.

Tutorial & example run While the user manual describes the functionality of each page, a step-by-step tutorial inside the application, guiding the user through a complete run from start to finish could further lower the entry barrier for new users. A foundation for this already exists in the manual testing protocol described in Section B, which walks

through a full "Model testing" workflow using sample data from the RAiSD-AI repository. Transforming this into a more accessible, beginner focused, in-app tutorial with more explanation of why each step is taken rather than just what to do, would give new users a concrete reference point for understanding the system before applying it to their own data.

A web based version During the user testing session with a past user of the RAiSD-AI, making the interface as a web application was recommended. The reasoning behind this recommendation was that some researchers may only wish to use the tool once or occasionally in their workflow and would prefer not to go through the process of installing the application and its dependencies locally. A previous design project which is also mentioned earlier in this report has attempted to provide a web based GUI for the RAiSD-AI, though the scope and design of that project differed from ours. A web based solution remains a promising direction which could make the tool accessible to a broader audience.

Support for different operating systems The GUI itself, since being built on PySide6, is not inherently limited to one operating system and would largely function on Windows or macOS as well without significant changes. However, RAiSD-AI currently supports Linux only, which means that the application as a whole is limited to that platform. Adding support for different operating systems would require changes to the RAiSD-AI itself rather than the GUI. Expanding the supported platforms would meaningfully lower the entry barrier and make the tool available to a wider range of users.

Data visualization As the GUI project was focused on the user experience for providing the input of RAiSD-AI, little attention was paid to methods of visualizing data that make the extensive output of the tool more accessible. In combination with the current project, such a feature would form a very complete toolkit for biologists who wish to analyze selective sweeps in genomic data.

Configuration file versioning In the current version of the project, the version of the configuration file found in the application directory is used to instantiate a run record. It is likely that a change in the configuration file will lead to incompatibilities between it and an existing history file, which prevents past runs from being opened in the GUI. A potential improvement could be to store past versions of the configuration file with each update, as well as marking runs saved to history with the current configuration version in order to offer support for older versions.

Integration of other tools While the scope of the project only included the RAiSD-AI tool, it is conceivable that the GUI could also offer users access to other open source genetics software such as SweeD or OmegaPlus.

9 AI statement

Throughout the duration of this project, artificial intelligence tools have been used sparingly. There are three main ways in which we have used them, which will be discussed below.

First, we used large language models as conversational partners when stuck with certain issues where the documentation was not helpful enough. Here, we used them to verify what we had done and for suggestions on how to solve problems. Artificial intelligence was not used to generate code. Therefore, all code created as a result of this process was written by us and we take full responsibility for it.

Second, we used artificial intelligence tools for simple but tedious tasks, such as renaming variables. This usage was accompanied by manually checking the correctness of the code produced, and we thereby take responsibility for it.

Lastly, for the writing of this report, we used a spellchecker. Again, we have checked all suggestions to ensure their correctness, and thus take responsibility for the results.

References

[1] N. Alachiotis and P. Pavlidis. "RAiSD detects positive selection based on multiple signatures of a selective sweep and SNP vectors." In: *Communications Biology* 1.79 (2018). DOI: https://doi.org/10.1038/s42003-018-0085-8.

[2] N. Alachiotis and P. Pavlidis. *RAiSD-AI*. 2024. URL: https://github.com/alachins/raisd-ai.

[3] M. Posthuma et al. *RAiSD-AI GUI and web service*. 2024. URL: https://bachelorshowcase-eemcs.apps.utwente.nl/content/AQAGE5Sf/Design_Project.pdf.

[4] P. Papadopulous. *RAiSD-AI Web Scanner (with Pre-Trained Demographic Models)*. URL: https://github.com/prodromospapa/raisd-ai_web.

[5] Z. Dong. *PhyloSuite*. URL: https://dongzhang0725.github.io/.

[6] L. Conklin. *OWASP Foundation*. https://owasp.org/www-community/Threat_Modeling_Process. [Accessed 15-04-2026].

[7] *threatmodelingmanifesto.org*. https://www.threatmodelingmanifesto.org/. [Accessed 15-04-2026].

A User Testing

RAiSD-AI GUI User Testing Questionnaire for Initial Design

Screener Questions

1. What is your profession/study?
2. Do you have some background in biology?
3. Have you ever used a scientific tool with a GUI? What kind of scientific tools have you used?

Usability Test Instructions The goal of this user interface is that it makes it easy and intuitive for researchers to use RAiSD-AI. During this user test, we will give you a task, and your goal is to figure out how to do that, with as little help as possible. Afterwards, we will go through it to get any feedback on the system. If it helps, you can say out loud what you are doing and thinking while trying to complete the task. The goal is for the system to be as clear and self-explanatory as possible, so if there is anything you do not understand, that is something for us to improve.

Main task Perform a calculation with two modes: **model training** and **sweep scan**, with the following attributes (parameters):

- Grid size = 7
- Use a mask for the missing data handling strategy
- Generate a separate report file per set
- Generate a site report

Post-Task Interview Questions

1. What is your overall impression of the GUI in terms of features and layout?
2. What do you think of the help tutorial?
3. Would you want to be able to close left and right panes of operation modes and history?
4. What do you think of selecting operation modes?
5. What is your impression of the page for selecting the operation name and files?
6. What do you think of the parameter overview?
7. How is your experience of confirming files and parameters? How do you find its usefulness?
8. What do you think of the animation that showed the progress of operations?
9. What do you think of the results page after the operation is finished?
10. What do you think of your general experience while navigating through the GUI? Was proceeding to next step intuitive for you on each step of operation?

11. What surprised you while using GUI (if any)?
12. If you could change anything about the GUI, what would you change?
13. How did the GUI compare to other programs you have used previously?
14. What final comments do you want to make before ending this interview?

RAiSD-AI GUI User Testing Questionnaire for Revised Design

Introduction First, let's explain a bit about us. We are currently doing a design project at the University of Twente, for which we are designing a graphical user interface for RAiSD-AI. The goal of this user interface is that it makes it easy and intuitive for researchers to use RAiSD-AI. What we have done so far, in terms of the graphical design, is that we have made a high fidelity prototype. We used this to perform user tests, from which we drew conclusions that helped us improve the design. We currently have an improved version of the GUI that we would like to get your opinion on.

Our plan for this meeting is to go through the UI with you, and ask you some questions. Additionally, if you have any questions or remarks at any time, let us know, it will probably be valuable feedback.

Pre-Task Interview Questions

1. How much have you used RAiSD-AI and how much do you use it right now?
2. What operations do you perform most with RAiSD-AI? (combination of modes)
3. What limitations do you encounter when using it through the terminal and what are the needs that can not be satisfied in this way?

Post-Task Interview Questions

1. What do you think about the GUI? (in general terms)
2. Would you use this GUI application?
3. Run-through:
   (a) Workspace: Do you like this option? Do you normally do multiple runs in the same folder/topic?
   (b) Tutorial: No questions here.
   (c) Mode selection: Is the mode selection page clear enough? Can you think of anything that might cause confusion and make people unable to use the program how they want?
   (d) Files selection: How do you find the file selection page?
   (e) Parameters selection: How do you find this page? Do you like the option of collapsing sections? Is it clean enough? (not too crowded and overwhelming)
   (f) Input confirmation: How do you find the input confirmation page?
   (g) Results display: Does how the output is displayed make sense? How do you normally use the output of executions?

(h) History: what do you think about the page where you can see the history? Do you find it useful?

(i) Settings: Do you think our setting section is complete? Is there something you might want to change that we miss?

4. What surprised you while using GUI (if any)?
5. If you could change anything about the GUI, what would you change?
6. What final comments do you want to make before ending this interview?

B System testing

In order to ensure expected functionality of the system as a whole, we conduct manual testing. We use the Black Box Testing technique, which means examining the functionality of an application without knowledge of its internal structures. This section contains a test protocol that covers the the functionality of the entire product. The prerequisites are:

- A local clone of the application source code.
- A workspace directory with copies of the training data (*msneutral1_100sims.out* and *msselection1_100sims.out*) in the *train/* subdirectory and the testing data (*msneutral1_10sims.out* and *msselection1_10sims.out*) in the *test/* subdirectory from the RAiSD-AI repository. For the rest of the section, the workspace directory will be assumed to be *∼/workspace/*.

Select the workspace

The first step of using the app is selecting the folder where the output will be generated. By default, this folder is located inside the application's root directory. Follow the instructions below to change the workspace directory to the one where the training and testing data are located.

Go to settings In the left sidebar, click the "Settings" (cog symbol) button.

Expected behavior The settings page is displayed.

Set workspace Click the "Change workspace" button. In the file dialog, browse to the *∼/workspace/* directory and select it.

Expected behavior The workspace setting is updated to the *∼/workspace/* path. The name of the window is changed to show the workspace path.

Set the run ID

The next step of using the application is setting the run ID. On the first page, there is a text box where the run ID should be filled in. The run ID represents the folder where the output is generated. From it, the run IDs of the operations are created. A valid run ID is a string without spaces. Follow the instructions below to check that the run ID works as expected.

Go to run page In the left sidebar, click the "Run" (play symbol) button.

Expected behavior The run page is displayed. The five steps are displayed in the bar at the top, with "Operation Selection" highlighted. The empty "Run ID" parameter is presented, with clear indications of the validation criteria. The "Next" button is disabled, since a valid run ID has not been filled in.

Set the run ID Fill in the value *manualTest_01* for the run ID.

Expected behavior The run ID text box shows the validity of the given value. The operation selection form appears on the left side of the page, with the "Sweep scan with μ statistic" operation is selected by default. On the right side, the heading of the operation is shown with a blue "i" icon next to it. When the icon is clicked the output location of the "Sweep scan with μ statistic" is shown as *~/workspace/manualTest_01/RAiSD_Report.manualTest_01_RSD-DEF/*.

Select operations and input files

After the workspace is set, the next step is to select the desired operation and provide the necessary input files. The input files required by some operations can be produced by other operations. In these cases, the application prompts the user to choose between uploading a file or generating it. If the needed files are not provided, the user cannot proceed to the next step. Follow the instructions below to check that the selection of operations and input files works as expected.

Choose Model testing In the list on the left side of the page, select the "Model testing" operation under "RAiSD-AI" heading.

Expected behavior The right side of the page displays the name and description of the "Model testing" operation. The output location of the operation is shown as *~/workspace/manualTest_01/RAiSD_Info.manualTest_01_MDL-TST/* when information icon is clicked. The two input directories of the operation, "Model" and "Testing data", are displayed side by side below. For the "Model" input directory, the choice of either picking a directory in the file system or running the "Model training" operation are presented. For the "Testing data" input directory, the options are choosing a directory or running multiple operations. For both of the two inputs, the option to choose a directory on the machine is selected by default. The "Next" button is disabled.

Choose Model training Under the "Model" heading, select the "Run Model training" option.

Expected behavior The name and description of the "Model training" operation are displayed. The output location of the operation is *~/workspace/manualTest_01/RAiSD_Model.manualTest_01_MDL-TST_MDL-GEN/*. The operation requires the "Training data" input directory. The directory can be selected on the computer or obtained by running multiple operations.

Choose to generate training data Under the "Training data" heading, select the "Run multiple operations" option.

Expected behavior The name and description of the "Data preparation" operation are displayed twice. For each instance of the operation, the image class label, image target site input fields can be filled in and the input file can be selected.

Fill in training data options For the first instance of the "Data preparation" operation under "Training data", enter the image class label "TrainNeutral", enter the image

target site as 50000 and select the file *~/workspace/train/msneutral1_100sims.out*. For the second, enter the image class label "TrainSelection", enter the image target site as 50000 and select the file *~/workspace/train/msselection1_100sims.out*.

Expected behavior The output location of the first operation is *~/workspace/manualTest_01/RAiSD_Images.manualTest_01_MDL-TST_MDL-GEN_IMG-GEN/TrainNeutral/*. The output location for the second operation is *~/workspace/manualTest_01/RAiSD_Images.manualTest_01_MDL-TST_MDL-GEN_IMG-GEN/TrainSelection/*.

Choose to generate testing data Under the "Testing data" heading, select the "Run multiple operations" options.

Expected behavior Two more instances of the "Data preparation" operation are displayed on the right. For each instance, the image class label, image target site input fields can be filled in and the input file can be selected.

Fill in testing data options For the first instance of the "Data preparation" operation under "Testing data", enter the image class label "TestNeutral", enter the image target site as 50000 and select the file *~/workspace/test/msneutral1_10sims.out*. For the second, enter the image class label "TestSelection", enter the image target site as 50000 and select the file *~/workspace/test/msselection1_10sims.out*.

Expected behavior The output location of the first operation is *~/workspace/manualTest_01/RAiSD_Images.manualTest_01_MDL-TST_IMG-GEN/TrainNeutral/*. The output location for the second operation is *~/workspace/manualTest_01/RAiSD_Images.manualTest_01_MDL-TST_IMG-GEN/TrainSelection/*. The "Next" button is enabled.

Fill in parameters

Now that the operation has been selected, the next step is to specify the needed parameters. If the parameters are invalid, the user cannot proceed to the next step and there is an indication of what parameter is wrong. Moreover, each parameter can be reset with the "Reset" button on its right. Follow the instructions below to check that the parameter form functions as expected.

Enter invalid value Input −100000 for the "Data preparation - Region length" parameter under the "RAiSD-AI data preparation" parameter section.

Expected behavior The border of the parameter will turn red as well as the hint text under the input field and at the bottom of the page a message indicating the name of the invalid parameter will appear. When clicking "Next" the sections containing invalid parameters are highlighted with red borders and input fields inside all the sections are highlighted with green or red borders depending on their validity.

Reset parameter Click the "Reset" button on the right of the "Data preparation - Region length" parameter.

Expected behavior The "Data preparation - Region length" parameter is reset to 1 and the border and the hint text turn green as the value is now valid. The red border around the section disappeared.

Enter valid parameter Input 100000 for "Data preparation - Region length" parameter under the "RAiSD-AI data preparation" parameter section.

Expected behavior The border of the parameter will remain green.

Enter the rest of the parameters Under the "RAiSD-AI data preparation" parameter section, tick "Convert image data to .snp" and set "Data type" to "Derived allele frequencies and positions". Set "Epochs" from "RAiSD-AI model training" to 9. For the "Class tests" under "RAiSD-AI model testing" add the following two pairs: "TrainNeutral , TestNeutral" and "TrainSelection , TestSelection".

Expected behavior All parameters are valid and the "Next" button leads to the next page when clicked.

Inspect and edit parameters

After the user has entered their preferred parameter values, the GUI presents the terminal commands that are generated and prompts them to verify the correctness of the parameter values. At this stage, the user can either submit the run or return to edit any of the parameters. Follow the instructions below to check that the parameter confirmation screen functions as expected.

Go to parameter confirmation Click the "Next" button.

Expected behavior The "Parameter Confirmation" tab is displayed. The previously entered parameter values can be viewed at the top of the page. The values of the parameters cannot be edited. Below, six commands are listed accompanied by a "Copy" button. At the bottom of the page, the "Edit" and "Run" buttons are present.

Copy commands Click the "Copy" button located next to the terminal commands.

Expected behavior The generated commands are copied to the operating system's clipboard.

Return to parameter input Click the "Edit" button at the bottom of the page.

Expected behavior The GUI returns to the "Parameter Input" step. The previously entered values are still filled in, and can be freely edited.

Change number of epochs Change the value of the "Epochs" parameter under "RAiSD-AI model training" to 10. Click the "Next" button at the bottom of the page.

Expected behavior The "Parameter Confirmation" page is displayed again. Under the "RAiSD-AI model training" section, the "Epochs" parameter has the value 10.

View progress and inspect output

Once the parameters have been submitted, the user is brought to the "Run" tab. Here, the progress of the RAiSD-AI execution is displayed, along with the option to inspect the console output. Follow the instructions below to check that progress and output are displayed as expected.

Submit parameters Click the "Run" button at the bottom of the "Parameter Confirmation" page.

Expected behavior The "Run" tab is displayed. Six progress indicators are shown with operation names inside them, corresponding to the six submitted operations. Execution begins automatically, and the indicators show successive completion of the operations.

Show console output During execution, click the "Toggle console" button.

Expected behavior A panel appears which displays the standard output and the error output side by side. The left side contains the standard output of the RAiSD-AI tool. The right side of the panel, containing the error output, is empty.

Hide console output Click the "Toggle console" button again.

Expected behavior The output panel is hidden. Once execution is completed, the "Results" tab is displayed.

View results

After a RAiSD-AI execution is completed, the user is able to inspect the output files and the parameters that were used. Follow the instructions below to check that the results are shown as expected.

Inspect output files In the output file view, expand the contents of the directories.

Expected behavior The output directory *~/workspace/manualTest_01/* contains six info files and three directories. The two *RAiSD_Images* directories each contain two subdirectories with images, as well as an info file. The *RAiSD_Model* directory contains a class labels file, a model file and an info file.

Check history

The user can refer to the history page to view details of previous RAiSD-AI runs in the current workspace. Follow the instructions below to check that the overview and details of past runs are displayed as expected.

Go to history In the left sidebar, click the "History" (clock symbol) button.

Expected behavior The history page is displayed. The left side of the page contains the list of past runs. The only entry in the list is *manualTest_01*.

Inspect completed run Click the entry for *manualTest_01* in the history list.

Expected behavior The right side of the history page is populated with the details of the *manualTest_01* run. The output files are the same that were displayed on the "Results" page. A button to reuse the values of the parameters is present.

View parameters Click the "Parameters" button at the bottom of the details pane.

Expected behavior The parameters that were previously entered as part of submitting the *manualTest_01* execution are displayed. The parameters cannot be edited.

Reuse parameters

Within the history page, the user is provided with the option of reusing parameters. The same functionality is provided on the "Results" page. Follow the instructions below to check that reusing parameters works as expected.

Click edit Click the "Edit" button on the bottom right to reuse the values of the parameters.

Expected behavior The "Operation Selection" page is displayed instead of the history. The run ID, selected operations, parameters and input files are restored from the original execution. Each operation displays a warning about the output being overwritten, since they all have the same output locations as previously. The next button is disabled until the confirmation for overwriting the output is given for each operation.