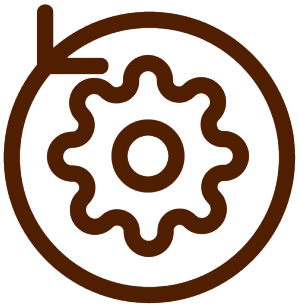
Introduction of a Plugin System for ReSet

Authors: Fabio Lenherr / Felix Tran

Project Advisor: Prof. Dr. Frieder Loch



School of Computer Science

OST Eastern Switzerland University of Applied Sciences

Abstract

In this thesis, a plugin system for the existing ReSet application is developed. ReSet is a settings application for Linux that aims to provide support for multiple desktop environments and window managers/compositors. Therefore, ReSet offers only a small set of core features due to its focus on universal environment support. As such a plugin system is needed in order to offer additional functionality.

The architecture of the plugin system was developed by analyzing existing solutions for other software and by creating various prototypes to prove the viability of each system on ReSet and its potential plugins.

The plugin system was ultimately developed with shared libraries which allows for resource sharing in both the daemon and the user interface of ReSet without the additional overhead of an interpreter.

To prove the plugin system, two exemplary plugins were developed in this thesis. The first is a monitor plugin, which allows users to change the individual settings of each monitor and rearrange their monitors. The second is a keyboard plugin that allows users to add, remove and rearrange keyboard layouts. Combined with the plugin system is a testing framework that also allows plugin developers to include their tests within ReSet in order to allow integration tests.

In summary, the plugin system was successfully implemented, with both plugins expanding the functionality as expected. Additionally, the plugin system offers ReSet the opportunity to offer limitless potential in both environment and hardware support, while also giving users the option to choose their options within ReSet.

Acknowledgments

We would like to express our sincerest gratitude to Prof. Dr. Frieder Loch for advising us throughout our project. His guidance and support have been invaluable to us. We are also grateful to our testers who took the time to test our product and share valuable feedback. Their input has been instrumental in helping us make improvements to our product. We would also like to acknowledge the OST for providing us with the opportunity to work on this project.

Table of Contents

1 Introduction

ReSet is a settings application for Linux made with GTK (formerly Gimp toolkit) and the Rust programming language. The application aims to provide functionality for all user interfaces (Desktop Environments) on Linux, reducing the need for first-party applications by the environments themselves or the reliance on smaller single-utility applications.

ReSet was a semester project for the OST, Eastern Switzerland University of Applied Sciences. This thesis will expand on the original work, solving existing issues and expanding its feature set.

1.1 Objectives

The focus of this thesis is on the introduction of a plugin system for the existing ReSet application, which was omitted from the original work due to time constraints.

The objective of a plugin system for ReSet is to provide a framework for developers to create their widgets for ReSet in order to provide additional setting functionality. Currently, ReSet only offers basic features such as Networking, Bluetooth and Audio. These features are as generic as possible and are expected to be usable on all Linux environments (considering the hardware supports the functionality). Further functionality is often implemented on a per-environment basis. Attempting to apply all these specific implementations to ReSet would lead to an unmaintainable and likely non-functional product. Therefore, the plugin system can cover both individual implementations, and providing dynamic settings for specific hardware or user preferences.

Hardware examples are drawing tablets, virtual reality headsets, accessibility solutions and more. User preference examples cover the reality that not every user requires all settings, therefore a plugin system offers the users the possibility to omit subjectively superfluous settings.

In order to expand the original ReSet application, this thesis will also develop exemplary plugins. This will ensure the functionality of the plugin system and allows for testing and potential revisions of the system.

1.2 Optional Objectives

The original work has limitations which this thesis intends to build upon. As an example, in the original work, code was often suboptimal with certain parts being duplicated due to complexity constraints. Another critical section is the testability of the program, which was omitted from the original work due to time constraints.

As such, this thesis will include work on the original implementation in order to facilitate the basis for a reliable plugin system.

1.3 Challenges

- **Consistency**
  With the advent of the plugin system, the importance of a consistent user experience is more relevant than creating an application with multiple technologies and use cases are not trivial. For this requirement, all plugins need to be able to represent themselves as integrated functionality of ReSet in order for the user to not notice the plugin system at all.

  This ensures that users will be able to switch from one plugin to another without being introduced to new design paradigms or differing concepts.
- **Ease of Use**
  Creating a good user experience and making the application accessible to all users requires a deep understanding of the user base which could require a lot of time.
- **Maintainability**
  With increasing features, the technical debt might increase and become a potential slowdown in development or release cycles.
- **Developer Experience**
  Plugin systems require rigorous documentation and communication. Developers should always be informed about changes without needing to frequently visit the codebase.
- **Stability**
  Plugin systems should be stable in both “execution” and “environment”. Stable execution refers to the system not crashing, while the stable environment refers to the plugin system API, which should not break compatibility with existing plugins on every update.

1.4 Methodology

In order to provide necessary information about technologies used by plugin systems, various approaches and technologies are analyzed and explained in Section 2. Further, differing plugin system implementations were analyzed in Section 3. This information is then used within Section 3.6 in order to evaluate the best plugin system architecture for this thesis.
Explicit implementation details for the resulting plugin system are discussed in Section 4.

As explained in Section 1.1, this thesis will include exemplary plugins. For these plugins, existing solutions for specific environments are analyzed in Section 5, which is used to create a mockup in Section 6.1. This information is used to create the plugins in Section 6.2 and Section 6.3, with the results being tested by users in Section 14.5.

At last, the conclusion of this thesis and potential future work is found in Section 8.

1.5 Potential

This thesis has the potential to introduce new functionality for users of minimal environments who still wish to use a unified program with both a graphical and command line interface.

With further work, more plugins can be introduced that could bridge the gap to full desktop environments as explained in the original ReSet paper. [3]

2 Prelude

In order to properly analyze potential plugin system implementations, it is necessary to introduce required concepts for potential implementations. In this section, these concepts are discussed.

2.1 Mangling

Mangling is the act of renaming designators by the compiler. Mangling can have different strategies and reasons. For example, a compiler for languages such as C++, C and Rust use mangling in order to remove namespaces and conflicts. Consider the Rust code in Listing 1:

```rust
pub mod namespace1;
pub mod namespace2;

fn main() {
  let _ = namespace1::add(5, 2);
  let _ = namespace2::add(5, 2);
  // both of these get renamed
}

// in namespace1.rs
pub fn add(left: i32, right: i32) -> i32 {
  left + right
}

// in namespace2.rs
pub fn add(left: i32, right: i32) -> i32 {
  left + right
}
```

Listing 1: Namespaces in Rust

This code will be reduced to one namespace during compilation, this means the concept of “namespace1” or “namespace2” is lost. Without mangling, this would not work, as both functions have the same name, however, with mangling, functions will have a seemingly random name and will therefore not clash with each other.

In Listing 2, the difference between regular compilation with mangling and compilation with the annotation no_mangle is visualized.

```
; mangled add function
.section .text._ZN4main4main17h51f2041274cfd0bdE,"ax",@progbits

; function without mangle
.section .text.add,"ax",@progbits
```

Listing 2: Mangling in assembly

This approach worked fine with one add function, however trying to achieve the same with both functions will result in the compiler error visualized in Figure 1.

```
error: symbol `add` is already defined
 --> src/namespace2.rs:2:1
  |
2 | pub fn add(left: i32, right: i32) -> i32 {
  | ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

error: aborting due to 1 previous error
```

Figure 1: Mangle Error example

Mangling has other applications than just avoiding compilation issues, one of these is code obfuscation. This is often used for languages that cannot be reduced to binary files, with an example of this being the obfuscation of JavaScript applications or websites.

JavaScript is an interpreted language, meaning that the code needs to be read by an interpreter, this disallows the compilation to binary form and requires it to stay as code. If the source code of this application should still be hidden from the public, then developers will have to change the code to be intentionally as unreadable as possible. This obfuscation is often done with existing tools such as javascript-obfuscator [4].

In Listing 3, an example obfuscation of JavaScript code is visualized.

```
// unobfuscated code
function penguin() {
console.log("I like penguins.");
}
penguin();

// obfuscated code
(function(_0x29b584,_0x52a642){var _0x2430f6=_0x4396,_0x5aaa83=_0x29b584();
while(!![]){try{var                                                    _0xeb9c7d=-
parseInt(_0x2430f6(0x88))/0x1*(parseInt(_0x2430f6(0x8d))/0x2)
// multiple lines of unreadable code omitted
penguin();
```

Listing 3: Example obfuscated code

There is a second common use case for obfuscation, namely compression. A lot of JavaScript applications and websites have large codebases that would need to be shipped to the user. This size can have an impact if the user operates with a reduced download speed. In this case, the codebase will be compressed to as few files and lines as possible.

The resulting code will again be unreadable for developers but unlike pure obfuscation, the size of the entire package has been significantly reduced.

2.2 Dynamic Libraries

Dynamic libraries are an interpretation of a binary that can be loaded into memory and accessed at runtime by another binary. This means that developers can provide libraries that can be loaded into a program at runtime.

Due to the read-only nature of dynamic libraries, it is possible to use the same instance of a dynamic library for multiple programs. In other words, the library is loaded into memory and then accessed by multiple processes. This benefit can also be used within ReSet with the multiprocess paradigm via DBus.

2.2.1 Versioning

In order for dynamic libraries to be sharable, they need to be compatible with each program using the library. Depending on the system that loads the library, different tactics are used to load dynamic libraries. Under Linux, the loading behavior depends on the installation variant. System native installations will always share one instance of a specific library, there may not be any other version of this specific library. This approach ensures maximum re-usage of resources and keeps the footprint low. However, at the same, it requires that all applications installed on this system must be compatible with this version of the library. Simple versioning such as incrementing the version number for each small change would make this system infeasible. For example, simple bug fixes would break compatibility.

In order to solve this issue, semantic versioning is used. This system creates several guarantees for a library by using multiple different version numbers. [5]

In Listing 4 the semantic version of the library Glib is visualized. [6]

```
# name of library, change would mean different library!
/usr/lib/libglib-2.0.so
# major number, incompatible change
/usr/lib/libglib-2.0.so.0
# minor numbers, compatible change
/usr/lib/libglib-2.0.so.0.7800.4
```

Listing 4: Semantic versioning

The minor version of a shared library can have multiple numbers, usually, it will be two or three. For two, there is no difference between compatible feature enhancements and compatible bug fixes. With three numbers, the first number is a compatible feature enhancement, and the last number is a compatible bugfix.

For the Linux system native packages, this ensures the feasibility of a single shared library, even if the version might be different from the expected one.

For ReSet, Flatpak is used as well, this system also tries to re-use libraries if possible, however, due to the sandboxed nature of Flatpaks, it is also possible to install multiple versions of a specific library, ensuring that each program receives the necessary library. [7]

2.2.2 Virtual Memory and Global Offset Table (GOT)

Operating systems do not offer processes direct access to physical memory. This ensures that processes do not access random memory that is used by other processes. Virtual memory address mapping enforces this paradigm by creating a pointer map to physical memory. On this map, the operating system can control the allowed memory space of the application with the Memory Management Unit. Should the process try to access physical memory which is not offered to this process, then the MMU will cause an MMU fault signal. [8]

When loading a potential shared library plugin for ReSet, this would result in two virtual address mappings, one for the ReSet user interface and one for the daemon. Figure 2 visualizes both mappings.

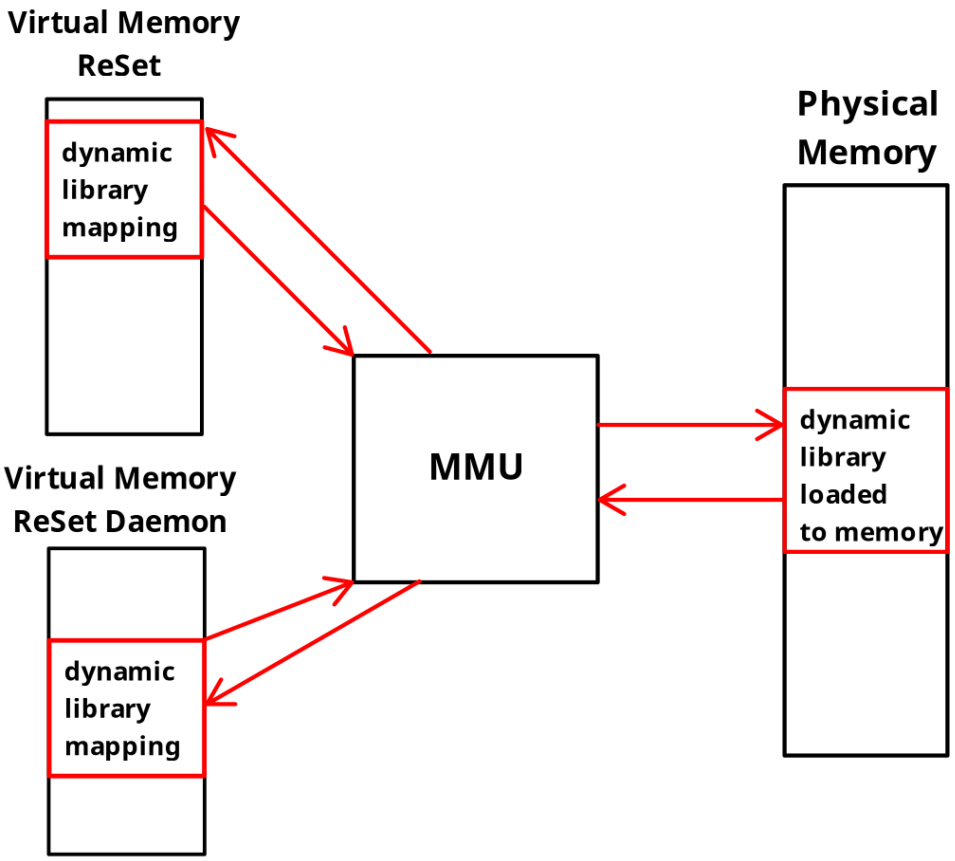


Figure 2: Virtual to physical memory mapping

Loading a shared library can either be done immediately, meaning all functions of a library are loaded on startup, or the library functionality can be loaded lazily. Lazy functions will only be loaded into memory when they are called. This can be achieved with the global offset table.

The global offset table stores pointers to loaded libraries which will redirect to the library functions. If the function is not yet loaded, then the global offset will load the function into memory first.

In Figure 3 an example function call with the global offset table is visualized. The function to be called will provide an output parameter which will be set by the library function. This means that the library needs a way to also access memory from the executable, which requires two-way access.

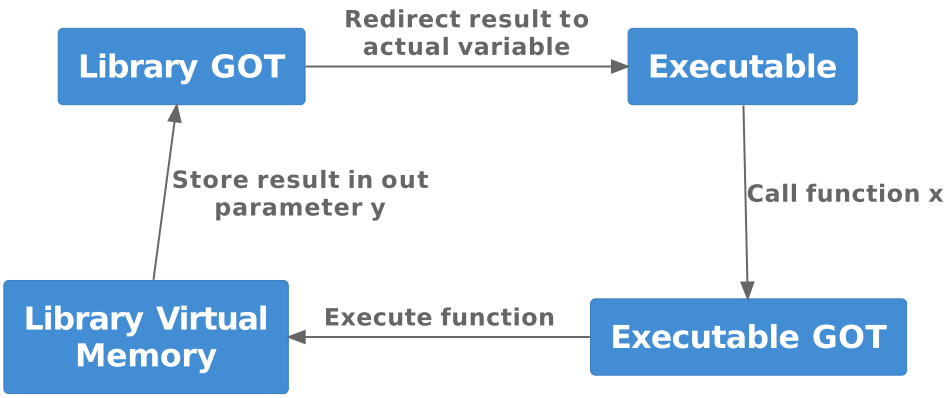


Figure 3: Global offset table usage example

2.3 Application Binary Interface (ABI)

Plugin systems based on dynamic libraries require that the plugins themselves are built against the current version of the ABI. For each specific change to ReSet and its respective daemon, the ABI might be changed as well.

Compared to an interpreted language, this is different from the fact that an API-compatible change is not necessarily ABI-compatible. For example, changing a parameter from i64 to i32 would not require a change for the programmer using the API. However, for the ABI user, this change would likely result in a crash as the compiled ABI changed.

Changes to function signatures which add or remove parameters or change the return type to a non-automatic cast would break API as well, meaning plugins based on an interpreted system would also need to be rewritten.

2.3.1 Exhibit ABI Compatibility

Due to ABI instability and specific interpretation of memory, it is often difficult if not impossible to provide interaction between languages if not used on a stable ABI such as the C language. In this section, an example of C++ to Rust compatibility is analyzed based on the Hyprland plugin system. [9] A working proof of concept plugin without functionality can be examined in Listing 78. Hyprland offers plugins via the C++ ABI, meaning no compatibility with any other language is offered out of the box.

Consider the C++ struct in Listing 5.

```
typedef struct {
 std::string name;
 std::string description;
 std::string author;
 std::string version;
} PLUGIN_DESCRIPTION_INFO;
```

Listing 5: C++ struct

Despite both Rust and C++ offering the same datatype, they are not properly compatible. A single Rust String to C++ std::string is transferrable over a shared library, however, a struct containing multiple strings is not converted properly and results in a double free, meaning memory is attempted to be freed twice by C++.

In order for this struct to be consistently compatible, it would need to be rewritten to use the C ABI, which can be seen in Listing 6.

```
extern "C" typedef struct {
  char* name;
  char* description;
  char* author;
  char* version;
} PLUGIN_DESCRIPTION_INFO;
```

Listing 6: C ABI compatible struct in C++

The same struct can then be configured in other languages as well, in Listing 7 the Rust equivalent is visualized.

```rust
#[repr(C)]
struct PLUGIN_DESCRIPTION_INFO {
  name: *mut libc::c_char,
  description: *mut libc::c_char,
  author: *mut libc::c_char,
  version: *mut libc::c_char,
}
```

Listing 7: C ABI compatible struct in Rust

2.3.2 Hourglass pattern

Using the hourglass pattern it is possible to provide a generalized ABI for which any language can be used to interact with a shared library-based plugin system. In order to achieve this goal, the application implementing the plugin system needs to provide the entire API as a C API. This API can then be targeted by other languages without knowing the implementation details. For proprietary applications, this is specifically interesting, since they do not need to provide any other code apart from the C header file.

In Figure 4 and Figure 5 the architecture of the hourglass pattern is visualized.

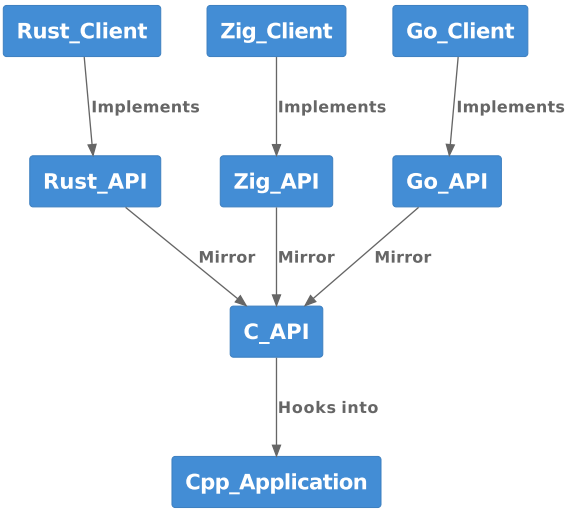


Figure 4: Hourglass architecture

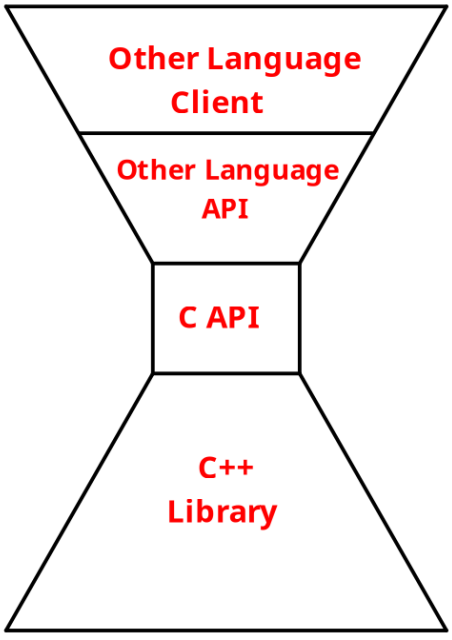


Figure 5: Hourglass visualization

A potential shared library plugin system for ReSet could also implement this C API in order to provide users of ReSet the possibility to use languages other than Rust. However, it is important to note that this would also mean including C bindings to DBus and GTK, which could increase the difficulty.

2.4 Macros

Macros are a way to change the code at compile time. In languages like C or C++, this is often used in order to differentiate different environments, prohibit duplicate imports or define constants. Rust macros are inherently different from this. Rust offers a macro system where the entire language is supported at compile time. While this does increase the overall complexity of the system, it also results in a system that does not end in simple text replacement. In C, all that macros do is replace text, in Rust, the macros will manipulate tokens instead, this guarantees that invalid tokens are prohibited, and operator precedence is not invalidated.

The book "The Rust Programming Language" offers a valuable example of operator precedence, and visualizes why C macros are often avoided. Consider the excerpt in Listing 8 taken from the book. [10]

```c
#define FIVE_TIMES(x) 5 * x

int main() {
 printf("%d\n", FIVE_TIMES(2 + 3));
 return 0;
}
```

Listing 8: C Macro

If this was a regular function in C, then the expectation would be 25 as the result: $5*(2+3)$. However, with C macros, the token x is not used as an actual token, instead, it is just text, so the result is: $5*2+3$. Without the parenthesis, the expectation of the result changes from 25 to 13. Bugs like these are incredibly hard to debug as they happen at compile time.

For comparison, the same macro in Rust in Listing 9 results in the expected 25.

```rust
macro_rules! five_times {
  ($x:expr) => (5 * $x);
}

fn main() {
  assert_eq!(25, five_times!(2 + 3));
}
```

Listing 9: Rust Macro

With the Rust version, it is clear that x is not just text, but an expression, which will be evaluated in full before entering it into the macro. In other words, the entire expression is multiplied by five, not just the first part of it.

A different approach to Rust would be to offer a flag to use code at compile time. This approach is used within C++ by adding constexpr or consteval to code. The difference between constexpr and consteval is the enforcement of compile time. If the code cannot be run at compile time, constexpr will not generate an error and instead run the code regularly, consteval would cause a compile time error in this case. [11], [12]

In Listing 10 the same functionality is visualized in C++.

```cpp
auto consteval five_times(int x) -> int { return 5 * x; }
int main() { std::cout << five_times(2 + 3) << std::endl; }
```

Listing 10: C++ Consteval

3 Plugin System Analysis

Plugin systems allow both the users and the developers of an application to provide specific implementations for use cases. Notably, it lessens the burden of development on the application developers, while also providing users with the functionality they explicitly want to use. The notable downside to this is the development and performance overhead the system itself has on the application and the newly created burden on the plugin developers.

For ReSet, the use case for a plugin system is the wide variety of features that either a system supports, or the user wants. As an example, it makes no sense to provide a VR Headset configuration for a system that does not support such devices. At the same, it is also counterproductive to offer settings for users that will never be used, for example, touchpad settings on a desktop. Here the system could either detect available devices and load plugins respectively or simply give users the ability to gradually control their used plugins.

Specifically for hardware support, a plugin system also covers potential future hardware, which would require new features to be added to ReSet in order to support new hardware. A plugin system can alleviate this problem, by offering a simpler extension mechanism.

3.1 High Level Architecture

ReSet follows a multi-process paradigm. This ensures that users have the option to avoid the graphical user interface of ReSet if they wish to. The parts of ReSet is therefore split into the daemon, which handles the functionality as a constant running service, while ReSet itself refers to the graphical user interface which will interact with the daemon via DBus (Inter-process Communication).

In Figure 6 the intended architecture of the plugin system is visualized.

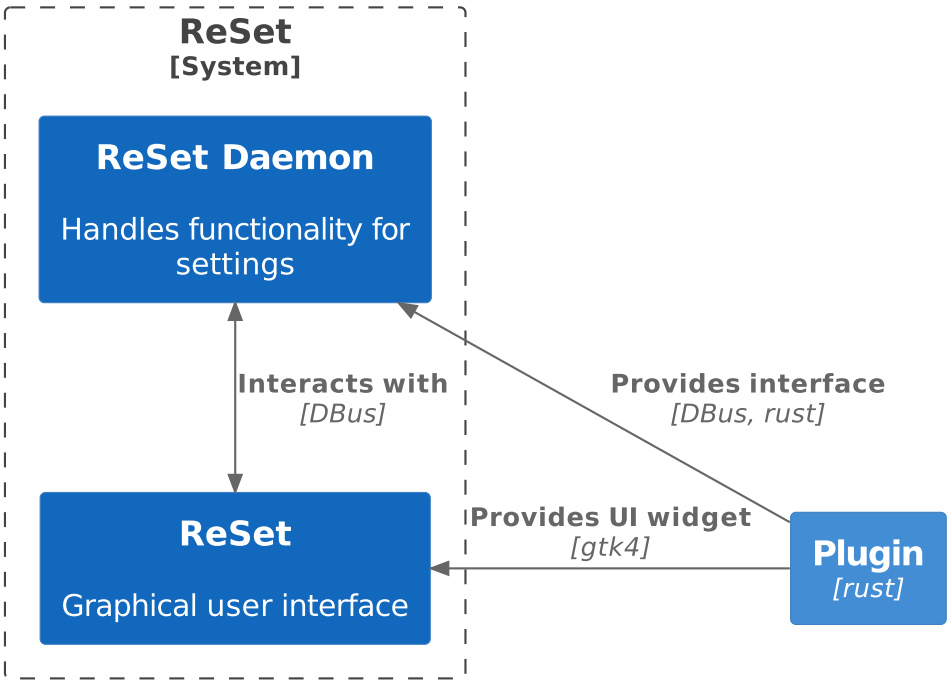


Figure 6: Architecture of ReSet

The intention is that each plugin can offer three parts for ReSet. The first part is the functionality itself, as an example, a monitor configuration plugin would need the ability to apply various resolutions to monitors. This functionality would then be offered by the plugin via a function or similar.

The next part of the plugin is the DBus interface, this interface could be directly injected into the existing DBus server provided by the ReSet daemon, providing users with a seamless user interface, meaning users would not see the difference between a plugin interface, and the core interfaces provided by ReSet out of the box.

The third part of a plugin is a user interface widget, which could be integrated into the ReSet user interface application. Notably, this is most likely optional, as users can choose to not use the user interface, and instead interact with ReSet via DBus.

3.2 Plugin System Variants

In this section, different variants of plugin systems are discussed.

3.2.1 Interpreted Languages

Interpreted languages can be run on top of the original application in order to provide on-the-fly expansion of functionality. In this case, the included interpreter uses functions within the application when certain functions are called by the interpreted language.

3.2.1.1 Custom Scripting Language

Creating a custom language just for a plugin system serves two potential use cases. The first would be security concerns which are explained in Section 3.3. The idea is that with custom scripting languages, it is possible to limit the functionality of the language, making it infeasible or harder for malicious developers to abuse the plugin system. The second use case is the simplification of functionality for potential plugin developers. For ReSet, this could mean creating ReSet-specific user interface elements with a single function call. However, this could also be potentially implemented with a library for an existing language.

3.2.1.2 Turing Complete

A simplified version of the meaning of Turing completeness for a programming language is whether the language can simulate the Turing machine (simplest possible computer). If the language can simulate a Turing machine, it would imply that the language has access to potential infinite loops and can therefore not be restricted in terms of functionality. In other words, if the language is Turing complete, any functionality that any other language can create is possible to be implemented. Turing incomplete languages on the other hand only offers a specific and limited set of functionalities, which cannot under any circumstance be extended without changing the language specification itself. [13]

The security aspect is likely the biggest factor in choosing to create a custom scripting language. Here the use of Turing incomplete languages is the most effective. It enforces limited functionality, which can severely limit the attack vectors compared to a Turing complete language. The downside of this approach is that only plugins with a supported use case can be created.

An example of a Turing incomplete language is the markup language HTML. It only offers specific tags which cannot create any functionality beyond the documented functionality. [14]

3.2.1.3 Error Handling

A big benefit of this system is the abstraction between the original application and the interpreted language. It allows the two parties to exist relatively independently of each other. This includes errors, which ensures that an error in the interpreted language does not lead to a full crash of the application.

As an example, browsers use the same independent error handling for web pages, hence when a webpage encounters issues, the browser itself is still usable.

3.2.1.4 Language Requirements

A soft requirement of an interpreted language for plugins is the simplification of creating a plugin for potential plugin developers. If this requirement is not fulfilled, then the interpreted language does not offer any benefit other than slight security improvements if the language is non-Turing complete.

The second requirement is interoperability with both the programming languages of the base system and any technology that is used within the base system. For ReSet, this would be Rust and GTK as the main technologies.

3.2.1.5 Architecture

in Figure 7 the architecture of a plugin system with interpreted languages are visualized.

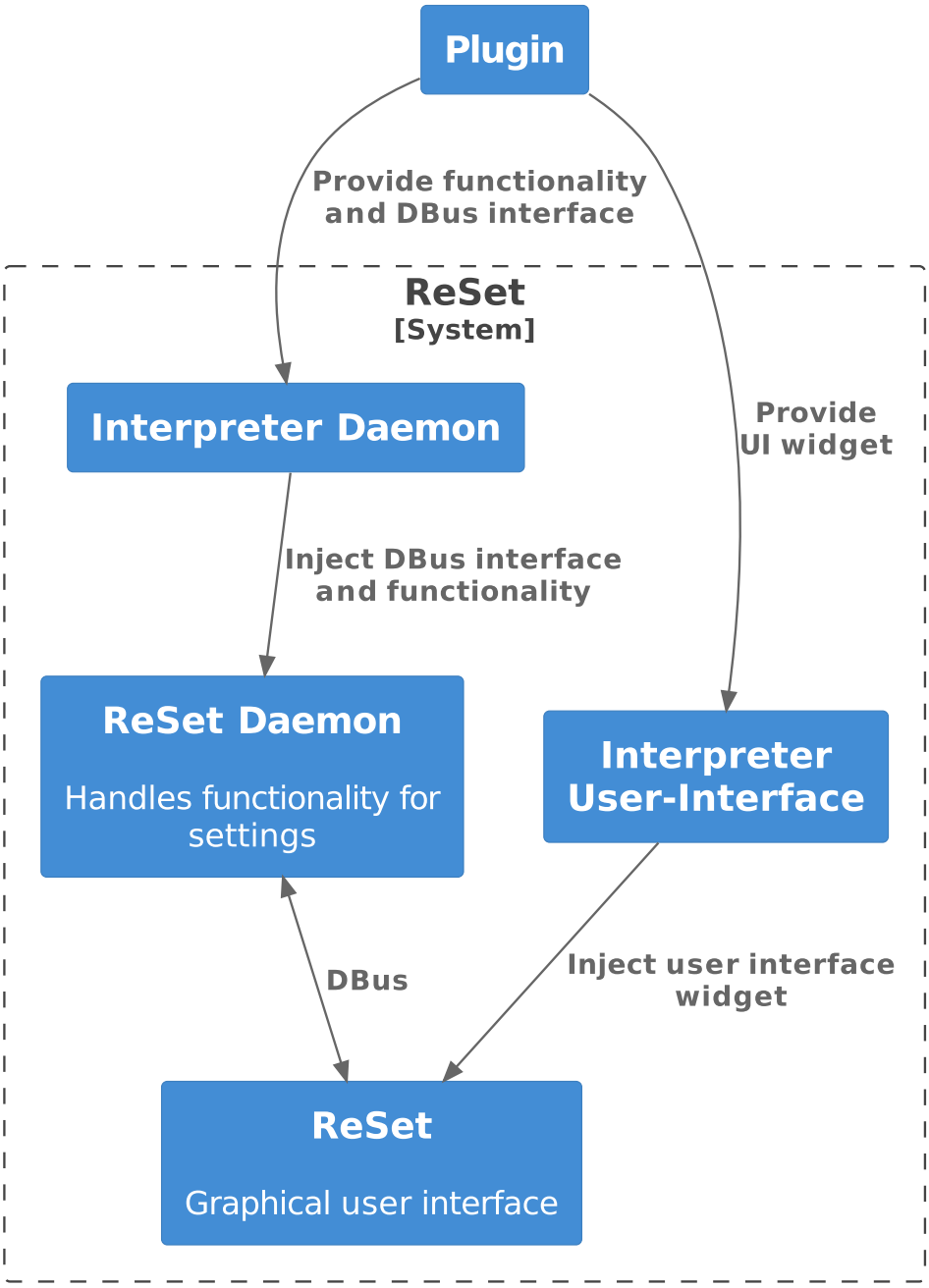


Figure 7: Architecture of a potential interpreted plugin system.

3.2.2 Code Patching

Code patching can be considered the simplest, although uncommon form of plugins. With code patching, users need to change the code of the application themselves in order to achieve the expected functionality. This type of extensibility is found in a specific set of free and open-source applications called “suckless”. [15]

Code patching technically means explicitly not implementing a plugin system and relying on pure code instead, however, it is still important to note that this approach requires a soft API/ABI stability. If the developers of the main applications often make radical changes, then these patches need to be rewritten for each change, which would make this approach completely infeasible.

Suckless specifically targets the Unix philosophy of “do one thing and do it well”.[16], [17] While ReSet is not opposed to this philosophy, ReSet does intend to offer more than one functionality by utilizing a plugin system in the first place. ReSet will therefore not pursue this approach.

3.2.3 Inter-Process Communication (IPC)

Inter-Process Communication can be seen as a soft version of a plugin system. While it does not allow users to expand the functionality of the program itself, it does allow users to wrap the program by using the provided IPC and expanding it with new functions.

ReSet itself is made with this idea in mind, expanding on existing functionality for Wi-Fi, Audio and more.

IPC has a major limitation, while the backend can be implemented solely with IPC by creating a new process that will handle the new functionality, the frontend cannot be expanded by just using IPC, hence this is not a system that can be fully applied to ReSet. On top of this, requiring a new process for each functionality would break the seamless user interface which was defined as a requirement for ReSet in Section 14.3.2.

3.2.4 Dynamic Libraries

As explained in Section 2.2, dynamic libraries can be used in order to load specific functions during runtime. This allows developers to load either specific files, all files within a folder or similar, which will then be used to execute specified functions for the application.

In order for this interaction to work, the plugin must implement all functions that the application requires, meaning the code has to be contained within the framework of the developer of the application.

As an example for Rust, consider the crate “libloading” which handles the mapping of C functions to Rust in a simple fashion. This allows a straightforward usage of dynamic libraries. Listing 11 visualizes a simple dynamic library with a single function.

```rust
// code in the calling binary
fn main() {
    // interactions with C are always unsafe
    unsafe {
        // open library file
        let lib = libloading::Library::new("./target/debug/libtestlib.so")
            .expect("Could not open library.");
        // fetch function from library
        let func: libloading::Symbol<unsafe extern "C" fn(i32) -> i32> =
            lib.get(b"test_function").expect("Could not load function.");
        // use the function
        assert_eq!(func(2), 4);
        println!("success");
    }
}

// code in the dynamic library
#[no_mangle]
pub extern "C" fn test_function(data: i32) -> i32 {
    data * data
}
```

Listing 11: Dynamic library loading in Rust

In Listing 11, the dynamic library has the annotation “no_mangle”. This flag tells the compiler to not change the identity of the specified designator. Without this, functions and variables will not be found with the original names. Mangling is further explained in section Section 2.1.

Additionally, in both the dynamic library and the calling binary, the flag “extern C” is used. This is required as Rust does not guarantee ABI stability, meaning that without this flag, a dynamic library with compiler version A might not necessarily be compatible with the binary compiled with compiler version B. Hence, Rust and other languages use the C ABI to ensure ABI stability.

The lack of a stable Rust ABI is also the reason why there are no Rust native shared libraries. Crates, as found on crates.io, are static libraries that are compiled into the binary, and all shared libraries are created with the C ABI using “extern C”. Further information about ABI along with examples can be found in Section 2.3. [18], [19]

3.2.4.1 Containerization of dynamic libraries

Plugin systems have a variety of points to hook a dynamic library into the application. The easiest is to just execute plugin functions at a certain point in the application. As an example, loading various settings in ReSet would mean looping through dynamic libraries and loading their respective user interfaces to show in ReSet. This approach is plausible and proven to work by a variety of existing plugin systems, however, it also has a major shortcoming. The moment the plugin crashes, the application has to either handle this unknown error or worse, if the error is not recoverable, the entire application crashes. This can lead to potential instability with plugins of different versions using different ABIs or simply due to bugs in a plugin.

To handle this case, a plugin system can also containerize plugins to run in their thread or process to provide independent error handling. This allows an application to recover even when a plugin exits abnormally while using resources.

In Listing 12 and Figure 8 the use of simple Rust threads guarantees the continuation of the invoking thread, meaning the underlying application can continue to run even though the spawned thread encountered a fatal error.

```rust
fn main() {
    thread::spawn(|| {
        library_function();
        // encapsulated functionality
    });
    // program should still get this input even if the thread crashes
    let mut buffer = String::new();
    println!("Still works?");
    io::stdin().read_line(&mut buffer);
}
```

Listing 12: Thread panic example

```
thread '<unnamed>' panicked at src/main.rs:18:38:
Could not load function.: DlSym { desc: "./testlib/target/debug/libtestlib.so: undefined symbol: non_existant" }
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
test
Still works?
```

Figure 8: Thread panic result

While the runtime guarantee is a benefit of this system, it also requires thread-safe synchronization of resources. This would incur a performance penalty on the entire system as even the native application services would now need to use synchronization when accessing data.

On top of this, when a plugin encounters a fatal error, this should not only be communicated to the user, but potential interactions with the now crashed plugin need to be removed. This might happen when plugins communicate with each other, or when multiple plugin systems are in place.

3.2.4.2 Architecture

In Figure 9, the architecture of a plugin system with dynamic libraries is visualized.

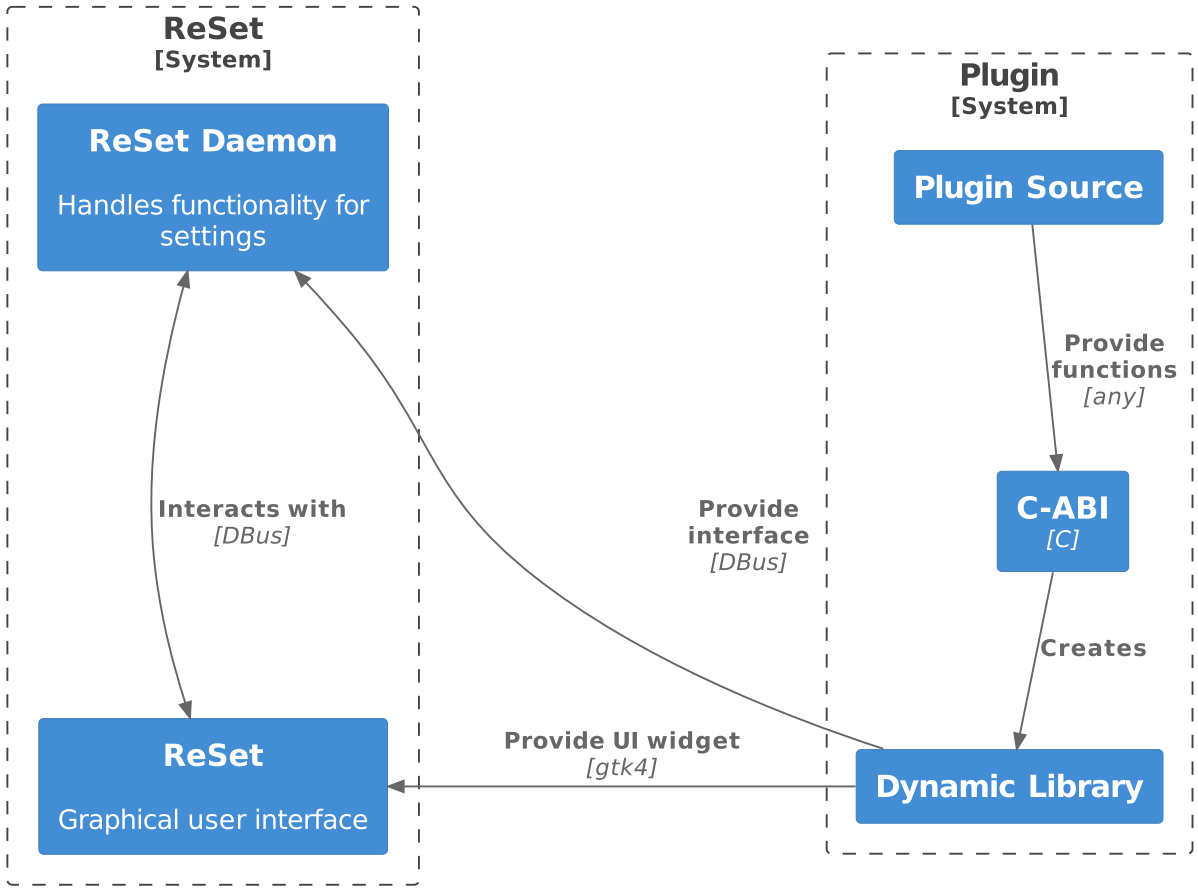


Figure 9: Architecture of a dynamic library plugin system.

3.2.5 Function Overriding

Function overriding is a variant of dynamic plugins which will override existing functionality instead of just expanding it. This type of functionality provides more control to the plugin developers, but also requires more maintenance from the plugin system developers and is more prone to breaking changes, as plugins interact more closely with the original system.

An existing plugin system with this variant is the GNOME Shell. This application is used on top of the GNOME Compositor to provide users with various user interfaces such as a status bar, notifications, and more. As GNOME-Shell is written in JavaScript, the overriding of functions is comparatively straight forward, meaning there are no type issues with extensions. Compared to using the ABI for compiled languages, this variant also means that just changing the integer variant does not break compatibility with existing extensions.

Using JavaScript for this use case also creates a bind, namely, it is no longer possible to split extensions, as JavaScript is a single-threaded system. This means that each extension that can crash, would also take down the GNOME-Shell as collateral.

To visualize the concept, Listing 13 provides an example of function overriding as a parameter in Rust.

```rust
use once_cell::sync::Lazy;
static mut G_PLUGIN_SYSTEM: Lazy<PluginSystem> = Lazy::new(|| PluginSystem {
    function: Box::new(regular_function),
});

fn main() {
    // The reason for unsafe is the mutability with different threads.
    // With static variables it is possible to access the same Plugin System
    // at the same time.
    // For a real system with different threads,
    // it would be required to put mutable access behind a locking mechanism.
    unsafe {
        G_PLUGIN_SYSTEM.call(5);
        // this override can be done with a dynamic library or IPC
        G_PLUGIN_SYSTEM.override_function(second_function);
        G_PLUGIN_SYSTEM.call(5);
    }
}

fn regular_function(data: i32) {
    println!("This is the first function: {}", data);
}

fn second_function(data: i32) {
    println!("This is the second function: {}", data);
}

struct PluginSystem {
    function: Box<fn(i32)>,
}

impl PluginSystem {
    fn override_function(&mut self, new_function: fn(i32)) {
        self.function = Box::new(new_function);
    }

    fn call(&self, data: i32) {
        (self.function)(data);
    }
}
```

Listing 13: Function overriding example in Rust

The output of this program is the regular_function and the second_function after this, both functions get the dummy data 5 passed to it. In a real-world example, this data could potentially be of the “any” type provided by the Any pattern, which would allow plugins to use custom-defined types, even with a statically typed language such as Rust. Important to note, however, is that even with the Any pattern, Rust would still break the ABI compatibility, as memory access depends on the size of a type.

3.2.5.1 Example Any pattern

In Listing 14 an example Any pattern implementation is visualized.

```rust
fn main() {
    let penguin = Example {
        name: "penguin".to_string(),
        age: 29,
    };
    let any_penguin = penguin.to_any();
    let restored_penguin = Example::from_any(any_penguin);
    assert_eq!(penguin, restored_penguin);
    println!("Success");
}

trait AnyImpl {
    fn to_any(&self) -> Any;
    fn from_any(any: Any) -> Self;
}

struct Any {
    // holds the data of all fields
    data: Vec<u8>,
    // determinant splits data vector back to fields
    determinants: Vec<usize>,
}

#[derive(PartialEq, Eq, Debug)]
struct Example {
    name: String,
    age: u32,
}

impl AnyImpl for Example {
    // This is just an example, could also be done with a Hashmap or similar
    fn to_any(&self) -> Any {
        let data = self.name.as_bytes();
        let determinants = vec![data.len()];
        let data = [data, &self.age.to_ne_bytes()].concat();
        Any { data, determinants }
    }

    fn from_any(any: Any) -> Self {
        let (name, age) = any.data.split_at(*any.determinants.last().unwrap());
        let name = String::from_utf8(name.to_vec()).unwrap();
        let age = u32::from_ne_bytes(age.try_into().unwrap());
        Self { name, age }
    }
}
```

Listing 14: Any pattern example in Rust

3.3 Security

Developers want to rely on the plugin system in order to focus on their core systems, however, this puts the burden of development on other developers, which could potentially have malicious intentions.

Similar concerns can be seen with browser extensions which are also just plugins, just for the browser itself. Some organizations require reviews from developers before publishing an extension to a web-based plugin "store", making it harder for malicious code to be published as an extension. [20]

3.3.1 Licensing

One way to approach this topic is to simply enforce plugins to use an open license. This could be done with a copy-left license which would enforce that any code utilizing the code of ReSet would also need to provide their source code with the same license. Currently, ReSet is already distributed under the GNU General Public License V3-or-later, which would apply the copy-left nature. Issues with this approach occur with legal questions, as enforcing copy-left licensing is not trivial, nor could code realistically be enforced to be used while utilizing shared libraries as the plugin system. It is also important to note that this would not make malicious plugins impossible, but it would signal to users that any proprietary plugin is by default against the terms of the license and therefore not to be trusted. [21]

3.3.2 Permission System

This approach would require the users' permission in order for a plugin to be included in ReSet. This could be done by creating a hash of the plugin, encrypting it with a password chosen by the user and storing this hash within a database (usually a secrets wallet). Should the plugin hash not be within the database upon startup, the user would be prompted for permission before loading the plugin. The challenge with this approach is the inclusion of an authentication mechanism. In order to facilitate this mechanism, ReSet could utilize the keyring functionality. This ensures that a user has a singular database that is not controlled by ReSet. The downside to this would be the dependency on keyrings themselves. [22], [23]

3.3.3 Signing

The Android ecosystem allows developers of apps to sign their application bundles with a cryptographic key. This allows users to be reasonably certain, that the package originates from the actual developer and not from a potentially malicious actor. It is however important to note, that signing plugins does not inherently mark plugins as safe, the only guarantee is that the key is from the developer. In other words, it does not protect against stolen keys that might be used by someone else, nor does it protect against malicious actors publishing their applications with their keys. Similar to the Licensing approach, users of ReSet would have to decide for themselves whether the developer of the application can be trusted in the first place. [24]

3.4 Testing

ReSet does not yet have a testing system implemented, only regular Rust tests are currently implemented, and those do not cover the full usage of the DBus API specified for ReSet. This is due to a lack of consistency with the features that ReSet provides. For example, without a mock implementation of the NetworkManager, it would not be possible to consistently connect to an access point. For this reason, all DBus interfaces must offer a mock implementation in order for it to be tested.

The second issue comes with the user interface, here regular Rust tests are meaningless. ReSet would need to use a GTK-compatible UI-testing toolkit.

After building this testing system, plugins can then also make use of this system by offering integration and unit tests for their use cases. This ensures that the plugin does not just work standalone, which would include specific functionality but also works in the entire system, which would cover the use inside ReSet by DBus and user interfaces.

In Figure 10, the architecture of the plugin system integrated into the testing framework is visualized.

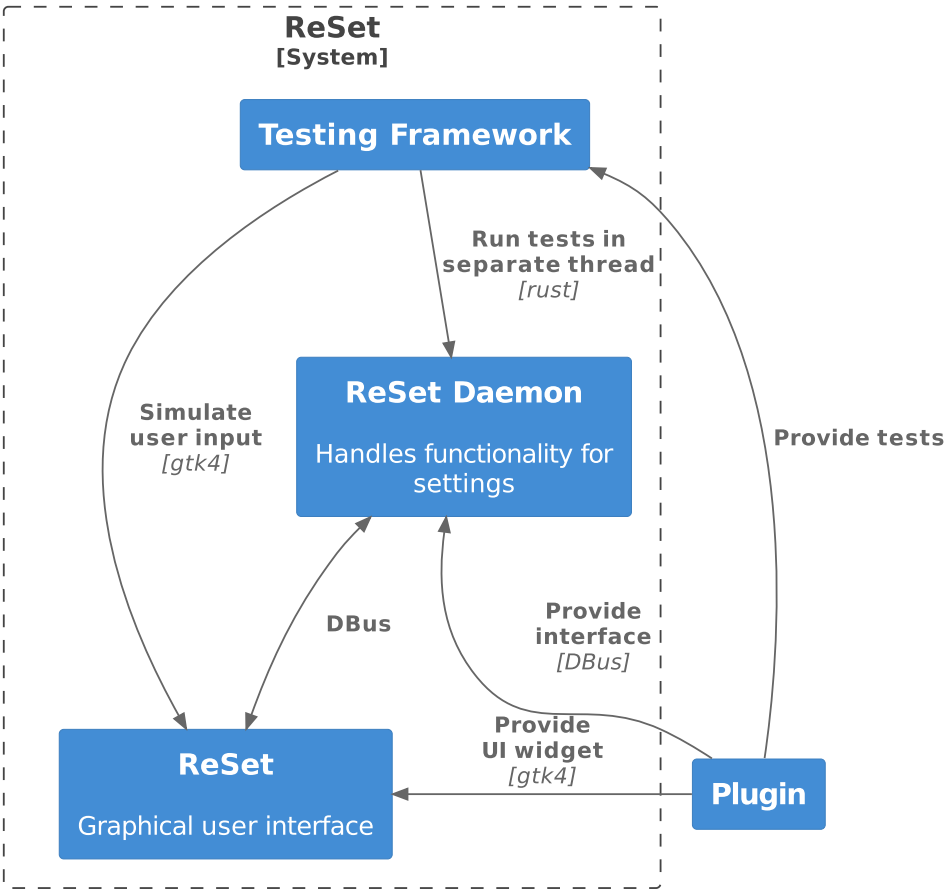


Figure 10: Architecture of the ReSet testing framework

3.4.1 GTK Tests

There is a GTK testing framework for Rust, which is called “gtk-test”. This crate allowed for an easy way of creating tests for the user interface of ReSet. As an example, the code snipped in Listing 15 is taken from the repository page and shows how to test the change of a string in a label. [25]

```
fn main() {
    let (window, label, container) = init_ui();

    assert_text!(label, "Test");
    window.activate_focus();
    gtk_test::click(&container);
    gtk_test::wait(1000);
    assert_text!(label, "Clicked");
}
```

Listing 15: gtk-test example

Unfortunately, that crate does not seem to be maintained anymore and is not compatible with GTK4 anyway. The general idea behind it was still useful and could be used to implement a new solution. Instead of returning each UI element in a tuple, saving it into a singleton would be much easier, especially when there are many UI widgets or dynamically generated widgets. These can then be easily accessed and manipulated during the tests.

```
static mut SINGLETON: Lazy<TestSingleton> = Lazy::new(|| TestSingleton {
    window: None,
    buttons: HashMap::new(),
    labels: HashMap::new(),
    checkbox: HashMap::new(),
    entryRow: HashMap::new(),
});
```

Listing 16: Structure of singleton

The idea is to save a reference to each widget that is going to be tested in a hashmap of its corresponding class with a string to identify it. The special case is the window because there is only one instance of it.

In Listing 17 and Listing 18, an example implementation of the singleton with GTK widgets alongside an example test is visualized.

```
let main = gtk::Box::new(Horizontal, 5);
let entryRow = EntryRow::new();
let button2 = Button::new();
let button1 = Button::new();
let label = Label::new(Some("nothing"));

entryRow.connect_changed(move |entry| {
    match Ipv4Addr::from_str(entry.text().as_str()) {
        Ok(_) => entry.add_css_class("success"),
        Err(_) => entry.add_css_class("error")
    }
});

let entryRow_ref = entryRow.clone();
let label_ref = label.clone();
button1.connect_activate(move |_| { entryRow_ref.set_text("192.168.1.100"); });
button2.connect_activate(move |_| { label_ref.set_text("button clicked"); });

let window = Window::builder().application(app).child(&main).build();
unsafe {
    SINGLETON.buttons.insert("button1".to_string(), button1.clone());
    SINGLETON.buttons.insert("button2".to_string(), button2.clone());
    SINGLETON.labels.insert("testlabel".to_string(), label.clone());
    SINGLETON.entryRow.insert("entryrow".to_string(), entryRow.clone());
    SINGLETON.window = Some(window.clone());
}
```

Listing 17: Setting up a simple UI

```
#[test]
#[gtk::test]
    let func = || unsafe {
        // test label text change after button click
        let label = SINGLETON.labels.get("testlabel").unwrap();
        assert_eq!(label.text(), "nothing");
        let button = SINGLETON.buttons.get("button2");
        button.unwrap().activate();
        assert_eq!(label.text(), "button clicked");

        // test entryRow css class change after button click
        let entryRow = SINGLETON.entryRow.get("entryrow").unwrap();
        assert_eq!(entryRow.css_classes().len(), 2); // 2 default css classes
        let button1 = SINGLETON.buttons.get("button1");
        button1.unwrap().activate();
        assert_eq!(entryRow.has_css_class("success"), true);
        SINGLETON.window.clone().unwrap().close();
        exit(0);
    };
    setup_gtk(func);
}
```

Listing 18: UI test

This code creates a few UI widgets with some binding to some signals. These signals can be activated by calling specific functions. The tests then get the references in the singleton and call functions that imitate click actions on buttons.

This approach unfortunately suffers from inefficiency due to the amount of boilerplate code and unnecessary compilation overhead it creates. Storing references to UI widgets in a singleton may simplify access for tests but is not needed at all by users.

3.4.2 GTK Test Macros

Macros as elaborated in Section 2.4 can be used to abstract the testing framework from different compilation targets. This means ReSet can drop the testing apparatus for release binaries which will benefit from better performance without the debug version suffering from reduced testing capabilities.

In Listing 19 an example macro for the test introduced in Listing 18 is visualized.

```
// debug version
#[cfg(debug_assertions)]
macro_rules! TestButton {
    ( $button_name:expr ) => {{
        let button = gtk::Button::new();
        unsafe {
            SINGLETON.buttons.insert($button_name, button.clone());
        }
    }};
}

// release version
#[cfg(not(debug_assertions))]
macro_rules! TestButton {
    ( $button_name:expr ) => {
        // drop name -> unused
        gtk::Button::new()
    };
}

// usage
let button3 = TestButton!("macro_button".to_string());
unsafe {dbg!(SINGLETON.buttons.clone());}
```

Listing 19: Macro Implementation

After compiling both versions of this test, the results in Figure 11 and Figure 12 show that the additional button reference in the testing singleton can no longer be found within the release version. Similarly, it is possible to also remove the entire singleton, or at least the underlying data for the release version.

```
    Running `target/debug/gtk_test`
[src/main.rs:78] SINGLETON.buttons.clone() = {
    "macro_button": Button {
        inner: TypedObjectRef {
            inner: 0x0000591d8275c6e0,
            type: GtkButton,
        },
    },
}
```

Figure 11: Macro Debug Version

```
    Running `target/release/gtk_test`
[src/main.rs:78] SINGLETON.buttons.clone() = {}
```

Figure 12: Macro Release Version

3.5 Plugin System Implementations

In this section, existing plugin systems are analyzed and evaluated. The results of this section will directly influence the choices for the plugin system ReSet will introduce. Evaluation of results can be found in Section 3.6.

3.5.1 Dynamic Library Plugin Systems

This section covers plugin systems utilizing dynamic libraries. Dynamic libraries are explained in detail in Section 2.2. [26]

3.5.1.1 Hyprland

Hyprland is a dynamic tiling compositor that can be extended via a plugin system. Both the plugin system and the compositor itself are written in C++. Notably, the plugin systems rarely use the "extern C" keyword, which results in only C++-compatible plugins. An example of this exact system can be seen in Section 2.3.1.

Using C++ ABI directly does offer both the plugin developer and Hyprland itself a benefit, namely the C++ classes, structs and functions do not need to be converted to C-compatible types. Depending on the complexity of the type, this can lead to a sizable overhead.

Hyprland also offers a hooking system besides regular additional plugins. The difference is that hooks allow direct modification of code execution, instead of execution at a specific location in code. With the hooking system, it is possible to replace any existing function within Hyprland with your own, or append functionality to it. In Listing 20, an example of the hooking system of Hyprland is analyzed.

```
// make a global instance of a hook class for this hook
inline CFunctionHook* g_pMonitorFrameHook = nullptr;
// create a pointer typedef for the function we are hooking.
typedef void (*origMonitorFrame)(void*, void*);

// define new/additional functionality
void hkMonitorFrame(void* owner, void* data) {
  // call the original function
  std::cout << "I bring additional functionality";
  (*(origMonitorFrame)g_pMonitorFrameHook->m_pOriginal)(owner, data);
}

// Find function listener_monitorFrame and return address of it
APICALL EXPORT PLUGIN_DESCRIPTION_INFO PLUGIN_INIT(HANDLE handle) {
  // Bind our hkMonitorFrame function to listener_monitorFrame
  static const auto METHODS =
    HyprlandAPI::findFunctionsByName(PHANDLE, "listener_monitorFrame");
  g_pMonitorFrameHook = HyprlandAPI::createFunctionHook(handle,
    METHODS[0].address, (void*)&hkMonitorFrame);
  // init the hook
  g_pMonitorFrameHook->hook();
}
```

Listing 20: Hyprland Hook Example [27]

The createFunctionHook API function takes a source and destination function. The source function is the address of the Hyprland function to be overwritten. Hence, a plugin developer would first need to get the address of that function. Hyprland provides a find function for this use case, making it easy to create hooks. The destination function is the function you would like to implement instead of the original functionality. Note that the original function can still be called at any point within your new function. In the official example from Hyprland, the additional functionality was added before the call to the original function. [27]

In Listing 21, the copying of the plugin-defined function is analyzed.

```
// alloc trampoline
// populate trampoline
// first, original but fixed func bytes
memcpy(m_pTrampolineAddr, PROBEFIXEDASM.bytes.data(), HOOKSIZE);
// then, pushq %rax
memcpy((uint8_t*)m_pTrampolineAddr + HOOKSIZE, PUSH_RAX, sizeof(PUSH_RAX));
// then, jump to source
memcpy((uint8_t*)m_pTrampolineAddr + HOOKSIZE + sizeof(PUSH_RAX),
  ABSOLUTE_JMP_ADDRESS, sizeof(ABSOLUTE_JMP_ADDRESS));

// fixup trampoline addr
*(uint64_t*)((uint8_t*)m_pTrampolineAddr + TRAMPOLINE_SIZE -
  sizeof(ABSOLUTE_JMP_ADDRESS) + ABSOLUTE_JMP_ADDRESS_OFFSET) =
  (uint64_t)((uint8_t*)m_pSource + sizeof(ABSOLUTE_JMP_ADDRESS));

// various code omitted: fixing pointers, NOP remaining code, etc.
// set original addr to trampo addr
m_pOriginal = m_pTrampolineAddr;
```

Listing 21: Hyprland Hook Creation [9]

Hyprland simply patches the memory itself by creating a trampoline function, which will point to the plugin function. This means that the instructions of the original function are turned to “No Operation (NOP)”, which will be skipped by the CPU. Instead, the trampoline will point to the new plugin function, which will be executed.

This system is both extremely powerful and potentially dangerous. It allows plugin developers to override any behavior of the existing application, providing potentially new and expected functionality, but also allows overriding any security implementations the program might have. For ReSet, this would likely not make much sense, as ReSet does not offer much functionality at its core, instead, ReSet intends to provide extend-ability.

A mock example Hyprland plugin written in Rust can be seen in Section 2.3.1, specifically Listing 78.

3.5.1.2 Anyrun

Anyrun is a Wayland application launcher similar to Launchpad for Macintosh computers. It is written in Rust and GTK3 and offers users the ability to load plugins with shared libraries using the stable ABI crate which automatically converts Rust structs to C structs.

Figure 13 shows the regular launch mode for Anyrun which allows users to find and launch desktop applications.

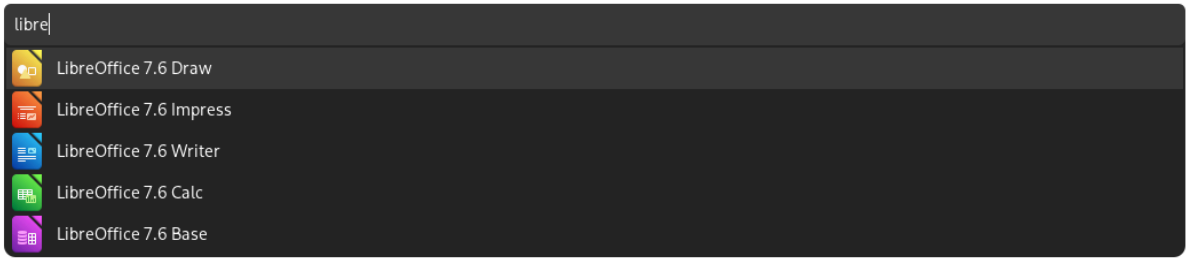


Figure 13: Anyrun Program search/launch

Figure 14 and Figure 15 show two plugins for Anyrun, a calculator plugin and a translate plugin.

Figure 14: Anyrun Calculator Plugin

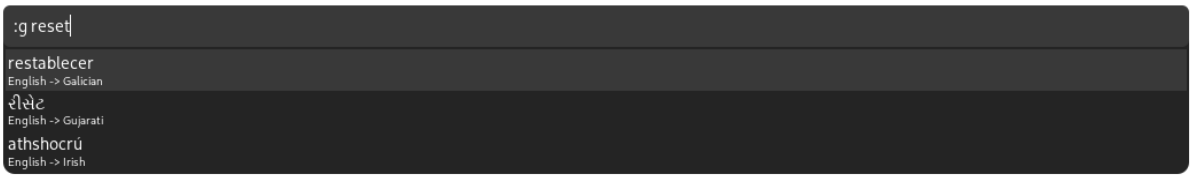


Figure 15: Anyrun Translate Plugin

Anyrun abstracts the plugin implementation behind several different crates. First is the ABI crate which handles the conversion of Rust-specific types into C stable ABI-compatible ones. [28]

In Listing 22, the plugin info struct of Anyrun is visualized.

```rust
#[repr(C)]
#[derive(StableAbi, Debug)]
pub struct PluginInfo {
    // RString is the stable ABI compatible string
    pub name: RString,
    pub icon: RString,
}
```

Listing 22: Anyrun PluginInfo [29]

The next part of the Anyrun plugin structure is found within the macro crate. Macros allow plugin developers for Anyrun to create plugins without the need to worry about certain restrictions like special types, mutexes and thread synchronization, as this can be handled automatically by the macro. More information about Macros can be found in Section 2.4.

In Listing 23, the simple macro that transforms the plugin-provided info function into the required format is visualized.

```rust
#[proc_macro_attribute]
pub fn info(_attr: TokenStream, item: TokenStream) -> TokenStream {
    let function = parse_macro_input!(item as syn::ItemFn);
    let fn_name = &function.sig.ident;

    quote! {
        #[::abi_stable::sabi_extern_fn]
        fn anyrun_internal_info()
          -> ::anyrun_plugin::anyrun_interface::PluginInfo {
            #function
            #fn_name()
        }
    }
    .into()
}
```

Listing 23: Anyrun Info Macro [29]

With these two crates, it is now possible to create a function within a potential plugin for Anyrun. The macro in Listing 23 is an attribute macro, this means the info function within the plugin will need to be annotated with the #[info] flag.

In Listing 24, an example info function is visualized.

```rust
#[info]
fn info() -> PluginInfo {
    PluginInfo {
        name: "YourPlugin".into(),
        icon: "YourIcon".into(),
    }
}
```

Listing 24: Anyrun Info Plugin Function [29]

This plugin function will now be used within Anyrun after loading the plugin.so file. In Listing 25, the usage of the function is visualized.

```rust
// load plugin first
let plugin = if plugin_path.is_absolute() {
    // load file
    abi_stable::library::lib_header_from_path(plugin_path)
} else {
  // omitted
}
.and_then(|plugin| plugin.init_root_module::<PluginRef>())
.expect("Failed to load plugin");

// omitted initialization etc

// handle file and run functions within the dynamic library
if !runtime_data.borrow().config.hide_plugin_info {
  plugin_box.add(&create_info_box(
      &plugin.info()(),
      runtime_data.borrow().config.hide_icons,
  ));
  // omitted
}
```

Listing 25: Anyrun Plugin Usage [29]

3.5.2 Interpreted Language Plugin Systems

This section covers plugin systems utilizing an interpreted language on top of their system in order to provide expandability. The paradigm used for such systems is explained in Section 3.2.1.

3.5.2.1 Neovim

Neovim is a fork of the ubiquitous VIM editor. [26], [30] It offers both VIMscript and lua support, as well as a Remote Procedure Call API, providing users with multiple ways to expand functionality. VIMscript is converted to lua, meaning Neovim only needs a single interpreter for lua. This interpreter is tightly coupled to Neovim itself, providing plugin developers with an easy way to access core functionality.

While the implementation of the interpreter itself is too large for a brief analysis, the usage of this system can be analyzed. Neovim provides solid documentation about their natively supported functionality which one can use within the Neovim lua interpreter. Note that this means the documented functionality is only supported in Neovim and not in any other Neovim interpreter. [31]

For the example plugin, the Neovim plugin package manager “lazy” is used. [32]
In Listing 26, a Neovim test plugin is visualized. [33]

```lua
-- define local variables
local opts = {
  val = 0,
}

local test_plugin = {
  opts = opts,
}

-- include variables in user configuration
function test_plugin.setup(user_config)
  test_plugin.opts = user_config
  vim.tbl_deep_extend("force", opts, user_config)
end

function test_plugin.config(user_config)
  test_plugin.opts = user_config
  vim.tbl_deep_extend("force", opts, user_config)
end

-- define function which the user can use
function test_plugin.test()
  if test_plugin.opts.val == 0 then
    vim.cmd("echo 'val is 0'")
  else
    vim.cmd("echo 'val is not 0'")
  end
end

return test_plugin
```

Listing 26: Example Neovim Plugin

3.5.2.2 Helix

Helix is a post-modern modal text editor written in Rust. [34] It currently does not offer a plugin system, however, as of February 2024, there is an open pull request on the helix repository. [35] This addition would introduce the lisp-based steel scripting language as a plugin system. [36]

The difference between Neovim with steel besides the paradigm of the scripting language is the integration with the Rust programming language. Steel offers a first-party virtual machine for its parent language. Creating plugins with steel would therefore require less work as a large portion would already be covered by steel.

In Listing 27, a simple usage of the steel engine in Rust and GTK is visualized.

```rust
use steel::steel_vm::engine::Engine;
use steel::steel_vm::register_fn::RegisterFn;

fn main() {
    // gtk initialization
    let vm = Rc::new(RwLock::new(Engine::new()));

    let button_func = ActionEntry::builder("test")
        .activate(move |_, _, _| {
            label_ref.set_text("button clicked");
        })
        .build();
    window.add_action_entries([button_func]);

    vm.register_fn("test-function", test_function);
    vm.run("(test-function)").unwrap();

    // due to both the VM and GTK being limited to one thread
    // it is necessary to provide thread synchronization
    let wrapper = Arc::new(ArcWrapper {
        window: window.clone(),
    });
    let gtk_action_function = move || {
          gtk::prelude::ActionGroupExt::activate_action(&*wrapper.window, "test", None);
    };
    vm.write()
        .unwrap()
        .register_fn("test-function", gtk_action_function);
    let button = Button::new();
    let newvm = vm.clone();
    button.connect_clicked(move |_| {
        newvm.write().unwrap().run("(test-function)").unwrap();
    });

    // run gtk
}
```

Listing 27: Example usage of the Steel Engine

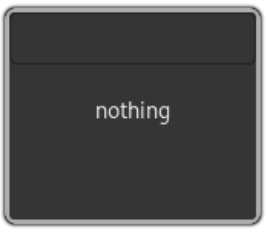


Figure 16: Steel GTK example

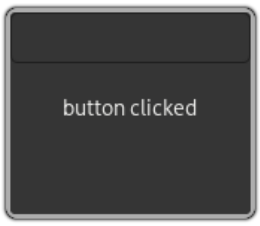


Figure 17: Steel GTK example (clicked)

A challenge to utilize any scripting language with GTK is that both GTK and the scripting language has its thread, and both are single-threaded environments. Hence, it requires manual synchronization and also requires special wrapping of GTK structs as they are not marked as Sync/Send, which is required in Rust for multithreading.

3.5.2.3 Usage in Games

Various game engines and modding frameworks also utilize lua as a scripting language in order to provide extendability.

Roblox for example uses lua to provide players with a way to create their game modes. [37]

There are also modding frameworks like UE4SS, which uses lua as the scripting language for modifications to Unreal Engine games. [38]

The difference between the Roblox system and the UE4SS system is that Roblox provides a specific binding for exposed functionality. This ensures that developers are unable to modify crucial systems like rendering, anti-cheat protection and more. The UE4SS system provides hooks, meaning it is possible to attach to existing functions and override or expand them. Hooking functionality is described in Section 3.5.1.1.

3.5.2.4 GNOME Shell

The GNOME shell is written in C and JavaScript. Specifically, the user interface part is written in JavaScript, meaning extensions can fully rely on this language. Usage of an interpreted language for this allows plugin developers to simply override existing functions at runtime without requiring a complicated plugin/hooking system.

In Listing 28, the override function is visualized. This function allows extension developers to override existing functionality within GNOME shell similarly to using hooks.

```
/**
 * Modify, replace or inject a method
 * @param {object} prototype - the object (or prototype) that is modified
 * @param {string} methodName - the name of the overwritten method
 * @param {CreateOverrideFunc} createOverrideFunc
 *        - function to call to create the override
 */
overrideMethod(prototype, methodName, createOverrideFunc) {
  const originalMethod = this._saveMethod(prototype, methodName);
  this._installMethod(prototype, methodName, createOverrideFunc(originalMethod));
}
```

Listing 28: GNOME Extensions Override Function [39]

```
// import the extension js from GNOME shell -> dependency
import {Extension} from './path/to/shell/extensions/extension.js';

export default class Example extends Extension.Extension {
  enable() {
    // override functions, add widgets, etc
  }

  disable() {
    // disable plugin
  }
}
```

Listing 29: GNOME shell example extension

Another benefit of the GNOME shell extensions is the fact that the regular GNOME shell has already created classes, functions and more within JavaScript. This allows plugin developers to just use said functions to create widgets and more using the same styling as the native GNOME implementation. For ReSet this could also be possible, even when using native Rust code, as ReSet-specific widgets can be exported to a crate, allowing plugin developers to utilize similar functionality.

3.6 Plugin System Evaluation

This section covers the chosen plugin system paradigm for ReSet. Paradigms are evaluated using a value table, which defines a score between 0 and 2 for each category. A score of 0 indicates an infeasible requirement, 1 indicates that the system could work, but not optimally, while a score of 2 indicates an optimal system for the requirement. Each category is also given a constant weight, in order to evaluate which paradigm is chosen.
The weights are as follows: low -> 1, medium -> 2, high -> 3

The following categories are evaluated for programming languages:

- **Testability** | weight: medium
  This category defines how well the system can be tested and whether plugins can also be tested using this system.
  As testability was problematic with the original work, this thesis intends to improve upon this situation. Therefore the weight medium is chosen as it represents an important but not critical aspect.
- **Language Conformity** | weight: low
  Language conformity defines how well a particular system would work with the Rust programming language. This category is evaluated by creating small prototype implementations of potential systems.
  The weight low is used due to the impact of this category. While it would be beneficial for the system to be easily integrated, it is not a crucial requirement.
- **Use Case Overlap** | weight: medium
  Specific paradigms will favor specific use cases, this point will solely cover how well the paradigm would suit the need of ReSet.
  The weight medium is used as the applicability of the system to the use case should overall fit the requirements. More so than integration with Rust, if the system does not fit into the architecture of ReSet, substantial work might be needed to fulfill the requirements.
- **Expected Workload** | weight: medium
  This is the evaluation of the authors on how much work is to be expected for specific paradigms based on the research of existing plugin systems.
  The weight medium is used as the workload must overall be in line with the time constraints of this thesis.
- **Compatibility** | weight: high
  Compatibility covers the need for the plugin system to interact with all existing third-party libraries. For ReSet, this would include the GTK user interface toolkit and the dbus-rs crate, for which the plugin must be suitable. This category is considered important, as rewriting ReSet with different libraries is not feasible for this thesis. [40]

Special requirement: All tools used in this project **must be published under a free and open-source license**, as ReSet will be published under the GPL-3.0 license.

3.6.1 Comparison

This section covers the comparison of each plugin system implementation when applied to ReSet.

For this test, only two approaches are considered, the first is the dynamic library paradigm and the second is the interpreted language paradigm. Paradigms like function hooks and function overriding are usually just slight variants of either dynamic libraries or interpreted languages, and would therefore suffer from similar benefits and downsides.

3.6.1.1 Testability

ReSet targets not only to offer testability for the core functionality but also for the plugins themselves. Considering the architecture of ReSet, this would require integration into both DBus and GTK.

For all plugin systems, tests would require the dynamic loading of plugins specified within a configuration file or similar. As Rust allows tests to execute any arbitrary code, loading of either dynamic libraries or virtual machines for interpretation is feasible.

For this category, there is no difference between any specific plugin system. Therefore, both systems receive the result of 1 as neither system works particularly well.

3.6.1.2 Language Conformity

All plugin systems use some form of abstraction compared to just using Rust. Meaning there is no outright winner. This is especially the case when looking at differing stable ABI usages and different scripting languages. Rust native structs, functions and more usually transfer without issue to both a stable ABI or a scripting language.

For this category, the analysis has not resulted in any meaningful differences in the architecture. Note, however, that this does not include specific usages of an architecture. For example, the steel scripting language offers better language conformity than using lua as the scripting language.

Both systems receive the baseline result of 2.

3.6.1.3 Use Case Overlap

ReSet targets solely the expansion of functionality without restrictions. This means that ReSet neither intends to offer custom hooking functionality, as there is no underlying system to hook into, nor does ReSet intend to prevent the plugin developers from using the full range of features of any tooling.

A potential plugin system should also consider the architecture of ReSet, as ReSet includes a multiprocess architecture. When comparing interpreted languages to dynamic libraries on this specific point, there is a considerable difference. As the name suggests interpreted languages require an interpreter, and as ReSet uses multiple processes, this would require two interpreters, adding an abstraction to the already distributed architecture of ReSet. With dynamic libraries, a library loaded into memory can be re-used, this means there is only one instance of a plugin within memory, which is then used by multiple processes.

The analysis concluded a significant benefit to using dynamic libraries for this category. As such, dynamic libraries receive the result of 2 while interpreted languages receive the result of 1.

3.6.1.4 Expected Workload

Dynamic libraries would likely require the use of macros to ensure the simple creation of plugins. Considering Section 3.5.1.2 as an example, this would require significant overhead for the implementation. However, interpreted languages could also require additional overhead when considering the integration of individual parts of the system. In Section 3.5.2.2, an example usage of an interpreted language is shown. Even with this simple example without integration of DBus, it already required additional thread synchronization.

For this category, no system has shown any substantial benefits. Both systems receive the baseline result of 1.

3.6.1.5 Compatibility

ReSet uses two libraries that a plugin system must be able to support, namely GTK and DBus.

For a plugin system utilizing dynamic libraries written in Rust, C or C++, this would not be an issue. All of these languages offer solid bindings for the language, or the entire library is already written with that language.

However, for interpreted languages, this would make it more difficult to use. ReSet could provide predefined widgets for the interpreted language in order to build a UI, however, this would conflict with Section 3.6.1.3, making it unsuitable. The secondary option is to use an interpreted language that offers bindings for GTK. This however would limit ReSet to a small amount of languages, none of which are written with Rust in mind, further complicating integration as mentioned in Section 3.6.1.2.

For this category using dynamic libraries offers a substantial benefit. As such, dynamic libraries receive the result of 2 while interpreted languages receive the result of 1.

3.6.2 Results

Table 1: Plugin System paradigms

Category	Dynamic Libraries	Interpreted Languages	Weight
Testability	**1**	**1**	**2**
Language Conformity	**2**	**2**	**1**
Use Case Overlap	**2**	**1**	**2**
Expected Workload	**1**	**1**	**2**
Compatibility	**2**	**1**	**3**
Total	**16**	**11**	

For Reset, the usage of dynamic plugins is the most suitable option for creating a plugin system. Not only will this guarantee that any Rust functionality will work, but it will also ensure that resources are shared between the daemon and the user interface.

4 Plugin System Results

This section covers the implementation of the ReSet plugin system.

The user tests for the results can be found in Section 14.5.

4.1 Resulting Architecture

The end resulting architecture is similar to the architecture in Section 3.2.4.2. The only difference is the absence of the C-ABI, which was omitted in favor of direct usage of Rust due to the tedious conversion of Rust to C. In Figure 18, the architecture is visualized.

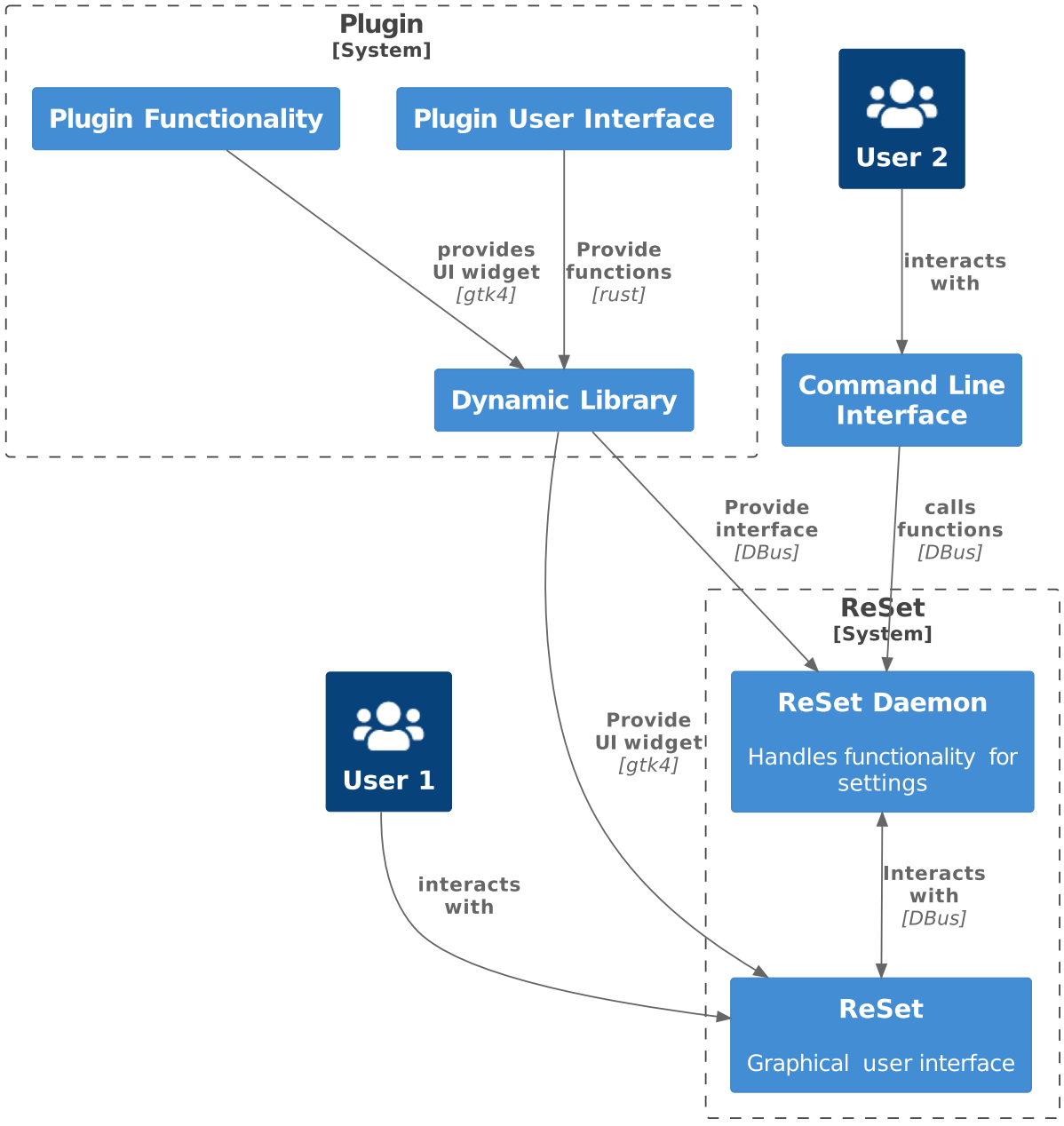


Figure 18: Resulting architecture of ReSet and its plugin system

4.2 Security

The initial idea of the ReSet plugin system was to reduce the attack vector by enforcing a narrow definition for the plugin system. As the ReSet daemon uses DBus, each plugin can provide its own DBus interface. For DBus, there are three different levels that a program can and must provide. The first is the DBus object, these objects then implement interfaces which in return provide functions for calling programs. ReSet plugins should only provide a name for the plugin and functions that will be implemented for the interface. This ensures that the daemon is the sole authority for adding any interface or object, which ensures that no plugin can override an existing interface, potentially shadowing a common interface with a malicious one.

The registration and insertion of interfaces are handled with the crossroads DBus context shown in Listing 30.

```
let interface = cross.register(interface_name, |c| {
  // functions for DBus
  // provided by plugin -> (plugin.dbus_interface)()
}
cross.insert(dbus_path, &[interface], data);
```

Listing 30: Example interface registration for DBus

During the implementation of this system, it was found that the DBus functions would not be able to be provided by the shared library. Attempting to call functionality provided by shared libraries would result in the object not being available. The only way to solve this issue would be to provide plugins with a reference to the crossroads DBus context, which would enable plugins to insert and register their interfaces.

However, it was possible to wrap the reference in a new type, meaning the overall goal of separating plugins from both existing functionality and other plugins is still guaranteed. The DBus object is created with the plugin name which must be defined as a separate plugin function.

In Listing 31, the injection of the interface is visualized.

```
pub fn setup_dbus_interface(
  cross: &mut RwLockWriteGuard<CrossWrapper>,
) -> dbus_crossroads::IfaceToken<SomePluginData> {
  cross.register::<SomePluginData>(
    "org.Xetibo.ReSet.SomePlugin",
    |c: &mut IfaceBuilder<SomePluginData>| {
      // define methods
    },
  )
}
```

Listing 31: Refined interface registration for DBus

While the cross variable in Listing 31 has the same name as the variable in Listing 30, the types are different. The original implementation used the Crossroads type which allows for both register and insert functionality, the latter of which allows creating new objects and also overwriting existing objects. The new CrossWrapper only allows plugins to register interfaces for their assigned object, meaning overriding is not possible.

4.3 Plugin Testing

Rust tests are handled by a specific test macro, this flag tells the compiler that this function is to be used when testing the specific project. This works well for a single project without dynamic loading of additional functionality, which for ReSet is the plugin system. While it is possible to call arbitrary functions within a function marked as a test function, it would also bundle all functions within the test function into one. If ReSet were to call all plugin tests from one test function, this would enforce that all plugin tests utilize the same thread and any error would cancel all remaining plugin tests.

In order to solve this issue, ReSet utilizes a different thread-spawning mechanism with a separate printing functionality in order to provide feedback about each plugin test function. This mechanism also allows plugins to be shown as separate entities with their tests, ensuring that developers receive appropriate feedback.

```
#[tokio::test]
async fn test_plugins() {
    use re_set_lib::utils::plugin::plugin_tests;
    setup();
    unsafe {
        for plugin in BACKEND_PLUGINS.iter() {
            let name = (plugin.name)();
            let tests = (plugin.tests)();
            plugin_tests(name, tests);
        }
    }
}
```

Listing 32: Custom plugin test framework

To write tests for a plugin, the functions backend_tests() and frontend_tests() can be implemented as seen in Listing 33. In there, a vector of PluginTestFunc must be returned, which contains a list of function references, which contain the test logic and a string, which acts as a test name. This works in the same way for the frontend tests.

```
#[no_mangle]
#[allow(improper_ctypes_definitions)]
pub extern "C" fn backend_tests() -> Vec<PluginTestFunc> {
    vec![PluginTestFunc::new(dbus_end_point, "Test DBus endpoint")]
}
```

Listing 33: Example backend test

In Listing 34, the dbus_end_point() function tests the SetMonitors and GetMonitors end points of the monitor plugin. If the DBus call fails, a PluginTestError will be returned that will be printed to the console.

```rust
pub fn dbus_end_point() -> Result<(), PluginTestError> {
    let conn = Connection::new_session().unwrap();
    let proxy = conn.with_proxy(
        "org.Xetibo.ReSet.Daemon", "/org/Xetibo/ReSet/Plugins/Monitors",
    );
    let monitors = vec![
        Monitor { id: 1, ..Default::default() },
        Monitor { id: 2, ..Default::default() },
    ];

    // Call set monitors
    let res: Result<(), Error> =
        proxy.method_call(MONITOR_PATH, "SetMonitors", (monitors, ));
    if let Err(error) = res {
        return Err(PluginTestError::new(format!(
            "DBus call returned error: {}", error
        )));
    }

    // Call get monitors
    let res: Result<(Vec<Monitor>, ), Error> =
        proxy.method_call(MONITOR_PATH, "GetMonitors", ());
    if let Err(error) = res {
        return Err(PluginTestError::new(format!(
            "DBus call returned error: {}", error
        )));
    }
    let len = res.unwrap().0.len();
    if len != 2 {
        return Err(PluginTestError::new(format!(
            "Result was not filled with 2 mock monitors instead got: {}", len
        )));
    }
    Ok(())
}
```

Listing 34: DBus Monitor Endpoint test

The plugin tests can be run with “cargo test – –nocapture”. Normally, Rust does not print logs when running tests, but with the nocapture flag, this behavior can be disabled. In Figure 19 an output of the plugin test can be seen. Figure 20 shows what the output looks like if a test has failed.

```
---------- Plugin Tests for Monitors ----------

running 1 tests:
running test Test DBus endpoint

0 tests crashed:

0 tests failed:

1 tests successful:
Success: Test DBus endpoint

---------- Plugin Tests end ----------
```

Figure 19: Plugin test output success

```
---------- Plugin Tests for Monitors ----------

running 1 tests:
running test Test DBus endpoint

0 tests crashed:

1 tests failed:
Test Test DBus endpoint failed with error: DBus call ret
urned error: Unknown interface org.Xetibo.eSet.Monitors

0 tests successful:

---------- Plugin Tests end ----------
```

Figure 20: Plugin test output fail

4.4 Plugin System Implementation

In this section, the specific implementation of the plugin system in ReSet is discussed.

As discussed in Section 3.6, the chosen paradigm is the dynamic library loading variant. With this paradigm, ReSet is required to provide a set of functions that a plugin needs to implement.

For the daemon, the specific list of functions is visualized in Listing 35.

```
extern "C" {
    // functions to create or clean up data
    pub fn backend_startup();
    pub fn backend_shutdown();

    // Reports the capabilities that your plugin will provide.
    // These capabilities will also be reported by the daemon.
    pub fn capabilities() -> PluginCapabilities;

    // Reports the name of the plugin
    // Mostly used for duplication checks
    pub fn name() -> String;

    // Inserts your plugin interface into the DBus server.
    pub fn dbus_interface(cross: &mut Crossroads);

    // import tests from plugins
    pub fn backend_tests();
}
```

Listing 35: ReSet Daemon plugin functions

A plugin that implements these functions can now be compiled and copied into the plugin folder. Additionally, in order to run the plugin, the configuration of ReSet would need to be updated to include the name of the plugin binary. This is to ensure that only user-defined plugins are loaded. Note that this does not offer proper security, it only offers protection against unintentional loading by the user.

In Listing 36, the TOML configuration in order to load the plugin is visualized.

```
plugins = [ "libreset_monitors.so", "libreset_keyboard_plugin.so" ]
```

Listing 36: ReSet TOML configuration for loading plugins

The loading of the plugins themselves is handled by the ReSet library which offers this for both the daemon and the user interface. It also covers the potential duplication of memory when loading the plugin twice. In other words, the library will only load the plugin into memory once, as defined in Section 2.2. The only difference is the fetching of functions from a plugin, which will be different depending on the daemon or the user interface.

This loading will also be done lazily. This means that the plugin files will only be loaded when either the daemon or the user interface explicitly call the plugin.

In Listing 37, the structures for loading plugins are visualized.

```
pub static LIBS_LOADED: AtomicBool = AtomicBool::new(false);
pub static LIBS_LOADING: AtomicBool = AtomicBool::new(false);
pub static mut FRONTEND_PLUGINS: Lazy<Vec<FrontendPluginFunctions>> = Lazy::new(||
{
    SETUP_LIBS();
    setup_frontend_plugins()
});
pub static mut BACKEND_PLUGINS: Lazy<Vec<BackendPluginFunctions>> = Lazy::new(|| {
    SETUP_LIBS();
    setup_backend_plugins()
});
static mut LIBS: Vec<libloading::Library> = Vec::new();
static mut PLUGIN_DIR: Lazy<PathBuf> = Lazy::new(|| PathBuf::from(""));
```

Listing 37: Plugin structures within ReSet-Lib

Because both the daemon and the user interface are started up in a non-deterministic fashion, the plugin library loading will also potentially be non-deterministic. This enforces an atomic check that both the daemon and the user interface can see. If the check is already either in loading or loaded, the other process would simply wait or immediately move on to using the plugin respectively.

Loading of plugins themselves in both the daemon and the user interface can be done by iterating over the BACKEND_PLUGINS or FRONTEND_PLUGINS global respectively. Both of these globals are both static and constant after initialization, this ensures that plugins cannot be changed later on, while also being initialized after startup.

In Listing 38, the plugin loading within the daemon is visualized.

```
unsafe {
    thread::scope(|scope| {
        let wrapper = Arc::new(RwLock::new(CrossWrapper::new(&mut cross)));
        for plugin in BACKEND_PLUGINS.iter() {
            let wrapper_loop = wrapper.clone();
            scope.spawn(move || {
                // allocate plugin specific things
                (plugin.startup)();
                // register and insert plugin interfaces
                (plugin.data)(wrapper_loop);
                let name = (plugin.name)();
                LOG!(format!("Loaded plugin: {}", name));
            });
        }
    });
}
```

Listing 38: Plugin loading within the ReSet daemon

4.5 Any-Variant

The Any-Variant was originally intended to be used for uniform plugin data which can then be sent across DBus interfaces. However, this complexity turned out to not be necessary for plugin data and is hence not used for this use case. Instead, the Any-Variant is now used to handle command line flags for the ReSet-Daemon as it allows arbitrary expansion and processing of flags as whole tokens by utilizing the Any-Variant as tokens.

In Section 3.2.5.1 an Any-Variant via byte vectors is covered. For ReSet, a different route was taken to implement the Any variant. Instead of converting byte vectors, ReSet utilizes polymorphism with types that implement a specific trait. For ReSet this would be implementing the TVariant trait visualized in Listing 39.

```rust
pub trait TVariant: Debug + Any + Send {
    // enables the type to be converted to the Variant struct
    fn into_variant(self) -> Variant;
    // converts the type to a polymorphic unique pointer
    fn value(&self) -> Box<dyn TVariant>;
}
```

Listing 39: TVariant trait

This trait is combined with the struct shown in Listing 40.

```rust
#[derive(Debug)]
#[repr(C)]
pub struct Variant {
    // converted value
    value: Box<dyn TVariant>,
    // rusts type id used to check for valid casts
    kind: TypeId,
}
```

Listing 40: Variant struct

Comparing this to a language like Java highlights both the complexity of Rust and the clear difference in paradigm. In Java, all reference types are linked to a garbage collector as well as equipped with a virtual table, which means that lifetimes, allocation and de-allocation, as well as casting, are handled automatically and enforced for all safe Java code. With Rust, a garbage collector does not exist, and virtual tables are an opt-in feature, as using it by default would introduce a performance overhead. For the Any variant, this virtual table is desired, as we would like the original value back when calling the “value” function.

```java
public interface IAny<T> {
  // public T into_variant();
  // This is not necessary as Java does not support arbitrary interface
  // implementation for existing types.
  public T value();
}

public class IntAny implements IAny<Integer> {
  private Integer value;

  public IntAny(Integer value) {
    this.value = value;
  }

  @Override
  public Integer value() {
    return this.value;
  }
}
```

Listing 41: Any variant in Java

When comparing Listing 40 and Listing 41, there is also the use of the unique pointer Box<T> in Rust. This unique pointer is used to ensure that the Any variant will always have the same allocation size. Omitting this pointer would enforce that the values within the variant are stored inside the variant struct without indirection. Given different possible value types, Rust could no longer determine the size of a specific variant at compile time, hence a pointer is used to mimic the behavior of Java (enforcement of references).

As mentioned at the start of this section, the Any-Variant is used to tokenize command line flags. This is stored into a lazily evaluated global accessible constant which stores the values in key-value pairs.

In Listing 42 and Figure 21, an example custom flag is visualized.

```rust
for flag in FLAGS.0.iter() {
  match flag {
    Flag::Other(flag) => {
      LOG!(format!(
        "Custom flag {}
         with value {:?}",
        &flag.0,
        flag.1.clone()
      ));
    }
  }
}
```

Listing 42: Custom flag in ReSet-Daemon

```
LOG: Custom flag --custom with value Variant {
    value: "flag",
    kind: TypeId {
        t: (
            16568496113410436123,
            11908032666093659918,
        ),
    },
}
```

Figure 21: Output of the Custom Flag in Listing 42

This flag can now be converted into a proper Rust type with the to_variant function on the Any-Variant. With this, it is extremely easy to create constraints for command line flags which ensure stability on use.

4.6 Mock System Implementation

The original work did not offer any testing potential, this resulted in a requirement for manual testing. This work implements a testing system for plugins covered in Section 4.3, and the implementation of mock DBus endpoints for the existing features covered in this section.

The implementations of Bluetooth and Network are straightforward to encapsulate into a testing interface as the original implementation is also done with DBus. In contrast, Audio is not implemented with DBus, which complicates the creation of a mock system significantly. However, Audio is also the only server that does not require hardware to be present in order to use basic functionality. When no input or output device is found, PulseAudio automatically creates dummy devices, which allow changing volume or changing the mute status. For ReSet, no further mock implementation for Audio will be created, as the default dummy output covers the vast majority of use cases.

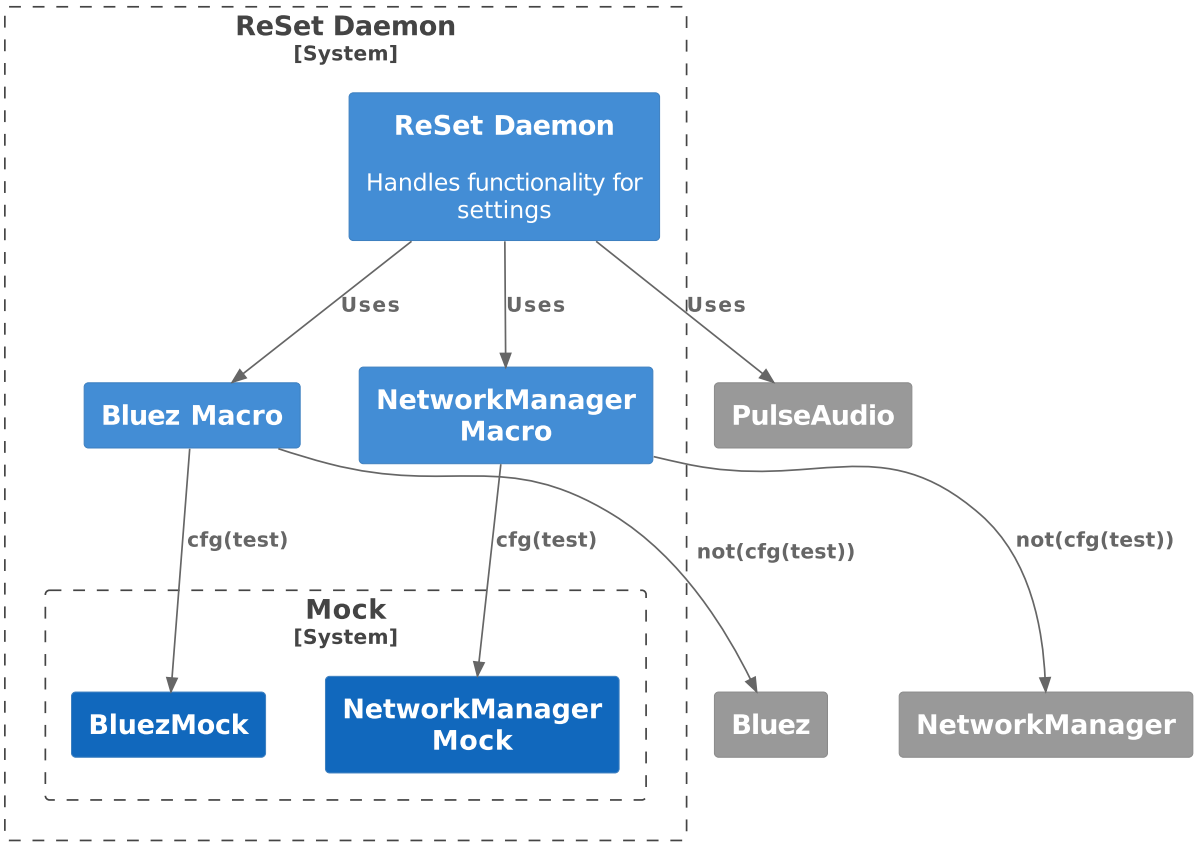


Figure 22: Architecture of the mock implementation

The cfg(test) annotation can be used to tell the compiler to include differing code in test environments. For ReSet, this is used to inject different interface names for DBus which will be handled at compile time. This means there is no performance cost to this system as everything is handled before running the binary. A user with the release version of ReSet would not even be able to see this system by decompiling it, as it does not exist within this version.

Further usage of macros with the mock system is covered in Section 4.6.2.

4.6.1 Testing the Mock System

In order to provide tests for the mock system, both the daemon and the mock implementations need to be available during the testing phase. However, Rust does not provide an easy way to ensure systems are available on test start, especially with the fact that tests are by default multithreaded and therefore run in a non-deterministic fashion. This forces developers to create their synchronization chain in order to create a proper setup for each test. For ReSet, this was handled by calling the same setup function with all threads and waiting for both the mock and the daemon with atomic booleans.

In Listing 43, the setup function for the mock tests is visualized.

```
fn setup() {
    // only the first test is starting the endpoint and the server
    if COUNTER.fetch_add(1, Ordering::SeqCst) < 1 {
        thread::spawn(|| {
            // create a thread and spawn the mock endpoint
            let rt2 = runtime::Runtime::new().expect("Failed to create runtime");
            rt2.spawn(start_mock_implementation_server(&READY));
            while !READY.load(Ordering::SeqCst) { hint::spin_loop(); }
            // omitted daemon creation
            rt.shutdown_background();
        });
    };
    // wait until the mock endpoint is ready
    while !READY.load(Ordering::SeqCst) { hint::spin_loop(); }
    // omitted wait for daemon
}
```

Listing 43: Mock testing system setup function

An example test case that uses the setup function alongside the mock system tests output can be seen in Listing 44 and daemon-tests-output respectively.

```
#[tokio::test]
async fn test_mock_connection() {
    setup();
    let conn = Connection::new_session().unwrap();
    let proxy = conn.with_proxy(INTERFACE, DBUS_PATH, DURATION);
    let res: Result<(), Error> = proxy.method_call(INTERFACE, "Test", ());
    assert!(res.is_ok());
}
```

Listing 44: Mock test example

```
test tests::test_plugins ... ok
test tests::test_mock_connection ... ok
test tests::test_get_default_source ... ok
test tests::test_get_output_streams ... ok
test tests::test_get_input_streams ... ok
test tests::test_bluetooth_get_devices ... ok
test tests::test_list_connections ... ok
LOG: Wrong password entered for connection: /org/Xetibo/ReSet/Test/Connection/100.
test tests::test_connect_to_new_access_point_wrong_password ... ok
test tests::test_add_access_point_event ... ok
test tests::test_get_default_sink ... ok
test tests::test_connect_to_new_access_point ... ok
test tests::test_get_sinks ... ok
test tests::test_get_sources ... ok

test result: ok. 13 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.05s
```

Figure 23: Daemon tests output

4.6.2 Macro Usage

The mock system will be used within the session(userspace) DBus, which does not require any special permissions. However, the actual implementation of NetworkManager or bluez both exist in the system-wide DBus, as they require hardware access. This situation requires ReSet to differentiate between session DBus and system DBus depending on the configuration. Just like in Section 4.3, Rust allows the differentiation of testing, debug or release configuration via compiler flags.

In Listing 45, the Rust macro for differentiation between session and system DBus is visualized.

```
#[cfg(test)]
macro_rules! set_dbus_property {
    (
    $name:expr,
    $object:expr,
    $interface:expr,
    $property:expr,
    $value:expr,
) => {{
    let conn = Connection::new_session().unwrap();
    let proxy = conn.with_proxy($name, $object, Duration::from_millis(1000));
    use dbus::blocking::stdintf::org_freedesktop_dbus::Properties;
    let result: Result<(), dbus::Error> = proxy.set($interface, $property, $value);
    result
  }};
}
```

Listing 45: DBus method macro for tests

This macro alone would not work if used in regular code, Rust requires a duplicate macro with the same name without the test configuration. In that macro, the new_session call would be changed to new_system.

The same configuration can be used to target different endpoints of DBus. This would mean targeting org.Xetibo.Test.NetworkManager instead of org.freedesktop.NetworkManager. In Listing 46, both the regular and the testing macro are visualized.

```
#[cfg(not(test))]
macro_rules! NM_INTERFACE {
  () => {
    "org.freedesktop.NetworkManager"
  };
}

#[cfg(test)]
macro_rules! NM_INTERFACE {
  () => {
    "org.Xetibo.ReSet.Test.NetworkManager"
  };
}
```

Listing 46: NetworkManager macros

5 Exemplary Plugin Analysis

5.1 Plugin Ideas

For this thesis, two common use cases with settings applications were chosen. This guarantees that all environments offer at least basic configuration of this functionality which can be implemented as a plugin for ReSet.

The first plugin is a monitor configuration plugin, which would allow users to create monitor arrangements, change their resolution/refresh rate and more.

The second plugin is a keyboard layout plugin, which allows users to add, remove and rearrange layouts to their preferred order.

Both of these plugins should offer both a daemon implementation and a graphical user interface.

5.2 Monitor Plugin

In this section, the implementations of monitor configurations are discussed.

For the monitor plugin, the following environments are considered: KDE, GNOME, wlroots-based compositors and KWin-based compositors. This selection covers a large section of the Wayland compositors except the cosmic desktop, which will be released after this thesis. [41]

When discussing implementations from other environments, they were tested on an EndeavourOS virtual machine with their respective dark themes applied. [42]

5.2.1 Environment Differences

The introduction of Wayland complicates the fetching of the data needed for configuring monitors. This is because many Wayland environments have their custom implementation of applying monitor configuration and will hence not be compatible with each other. For example, the wlroots implementation of applying a monitor configuration is defined by the Wayland protocol extension zwlr_output_manager_v1. [43] As the name suggests, this is made solely for environments using wlroots as their library. Similarly, KDE also offers its protocol, while GNOME chose the DBus route, providing a handy DisplayConfig endpoint. [44], [45]

For the X11 protocol, there is only one endpoint for fetching, as close to every environment uses the same display server (Xorg) in order to implement their stack.

5.2.1.1 Hyprland Implementation

Hyprland's monitors can be configured by three different approaches. The first would be to just use the inbuilt hyprctl tool, which provides a monitor command that can either display monitors in a human-readable way or output it directly to JSON. For this, it would be necessary to spawn the tool within the plugin and convert the output with serde, a serialization/deserialization framework for Rust. In Listing 47, the conversion from JSON to the generic monitor struct is visualized.

```
// The tool is always installed for hyprland.
pub fn hy_get_monitor_information() -> Vec<Monitor> {
    let mut monitors = Vec::new();
    let json_string = String::from_utf8(get_json());
    if let Ok(json_string) = json_string {
        let hypr_monitors: Result<Vec<HyprMonitor>, _> =
          serde_json::from_str(&json_string);
        // omitted error handling
        for monitor in hypr_monitors.unwrap() {
            let monitor = monitor.convert_to_regular_monitor();
            monitors.push(monitor);
        }
    }
    // omitted error handling

    monitors
}
```

Listing 47: Hyprland monitor conversion

The second approach is to directly use Hyprland's Unix sockets, which are fully replicated in hyprctl, meaning both solutions will lead to the same outcome. For sockets, the same conversion as with hyprctl would be required.

The third approach is to use the zwlr_output_manager_v1 protocol in order to apply the configuration. [43] Hyprland uses a fork of wlroots as a foundation library. A benefit with this would be the automatic support for any other environment that supports this protocol, the downside is that this protocol might not fully replicate Hyprland's features in the future, as this protocol specifically targets wlroots.

5.2.1.2 Wlroots Implementation

As mentioned in Section 5.2.1.1, the wlroots implementation can only be implemented via Wayland protocols and is as such also limited to the used protocol. Hence, the wlroots implementation does not offer persistent storing of monitor configurations, instead only offering the application of a specific configuration.

However, unlike the first and second Hyprland implementations, it is compositor independent and will work as long as the zwlr_output_manager_v1 protocol is implemented.

In order to connect to a compositor using Wayland protocols, a wayland-client is used. This client would then connect to the "server" which is the compositor. Wayland-client implementations are provided either directly by the Wayland project as a C library or by third parties such as smithay which implements the client and server part for Rust. [46], [47]

In Figure 24, the base Wayland architecture is visualized.

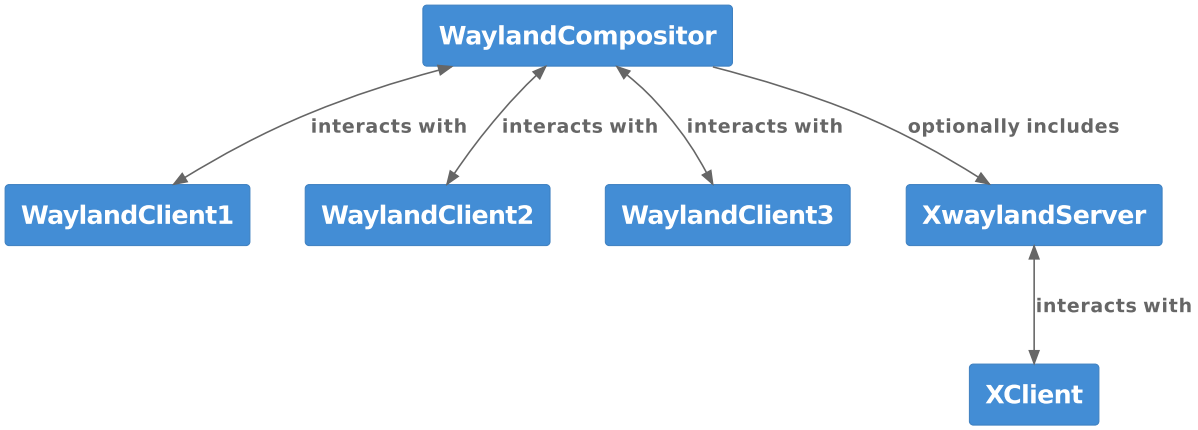


Figure 24: Wayland architecture

As such, it is now possible to create a client for a Wayland server and generate requests for it. The server will then respond with events that the client can process and interact with.

In Listing 48, an example client that requests all implemented protocols from the Wayland compositor is visualized.

```
struct AppData(pub String);
impl Dispatch<wl_registry::WlRegistry, ()> for AppData {
    fn event(
        obj: &mut Self,
        registry: &wl_registry::WlRegistry,
        event: wl_registry::Event,
        data: &(),
        conn: &Connection,
        handle: &QueueHandle<AppData>,
    ) {
        if let wl_registry::Event::Global { interface, .. } = event {
          // print the protocol name
          println!("{}", &interface);
        }
    }
}
pub fn get_wl_backend() -> String {
    let backend = String::from("None");
    let mut data = AppData(backend);
    // connection to wayland server
    let conn = Connection::connect_to_env().unwrap();
    let display = conn.display();
    // queue to handle events
    let mut queue = conn.new_event_queue();
    let handle = queue.handle();
    // creates an event for each wayland protocol available
    display.get_registry(&handle, ());
    queue.blocking_dispatch(&mut data).unwrap();
    data.0
}
```

Listing 48: Example Wayland connection with Smithay

5.2.1.3 KDE Implementation

Similar to Hyprland, KDE offers both a custom tool and a Wayland protocol to handle monitor configuration.

Also like with Hyprland, both solutions were implemented, for KDE the reason is simply to include support for both the X11 implementation of KDE and the Wayland version. If this plugin were to target the KWin protocol exclusively, then X11 support would not be included and would have to be implemented separately.

The protocol variant requires the implementation of two protocols that will interact with each other. The first is the kde_output_device_v2 protocol, which defines the data structure responsible for holding the necessary data for each monitor. This protocol closely resembles the Wayland core protocol "wl_output" which offers a way to fetch the currently active monitors with all their data. The only noticeable difference is the lack of data for potential changes a user can make. In other words, it only offers data about what is currently active, including refresh rate, resolution and more, but does not provide any data about what other refresh rates or resolutions, etc. the monitor supports. [48], [49]

The combinations of the kde_output_device_v2 protocol and the KDE output management v2 protocol enables the same functionality as with the implementation of wlroots in Section 5.2.1.2.

The KDE environment also offers an optional monitor module which provides the Kscreen-doctor tool. This tool allows for a quick fetch of data that can be output via JSON and then deserialized into a monitor data structure. [50]

5.2.1.4 GNOME Implementation

GNOME separates hardware monitors from logical monitors. The logical monitors represent the representation of the real-world monitor within GNOME with the x and y coordinates of the position added. [45] These coordinates will define which monitor will be considered "on the left", or "on the right" within GNOME.

The conversion for GNOME displays is trivial other than the challenges with experimental features. As of GNOME 46, features such as fractional scaling and variable refresh rates are still experimental features within GNOME. In order to still accommodate the features for GNOME users, ReSet has to check for the availability of these features not only within the monitor capabilities but also within the GNOME configuration. GNOME stores its configuration within a custom binary blob file which stores key/value pairs. ReSet already uses GTK, which can also read DConf variables, meaning there is no additional dependency for ReSet.

DConf is further explained in Section 5.3.1.2.

In Listing 49 the value for fractional scaling is read via DConf.

```rust
fn get_fractional_scale_support() -> bool {
    let settings = gtk::gio::Settings::new("org.gnome.mutter");
    let features = settings.strv("experimental-features");
    for value in features {
        if value == "scale-monitor-framebuffer" {
            return true;
        }
    }
    false
}
```

Listing 49: GNOME fetching of Fractional Scale support

5.2.2 Visualization

For the visual representation, ReSet aims to be aligned with other configuration tools within the Linux ecosystem in order to provide users with a seamless transition. Notable for this visualization is the use of drag-and-drop for monitor positioning. This paradigm allows users to quickly place monitors on their preferred side or height.

To accommodate users of other applications, the environments GNOME and KDE are discussed.

In Figure 25, the KDE variant of the monitor configuration is shown.

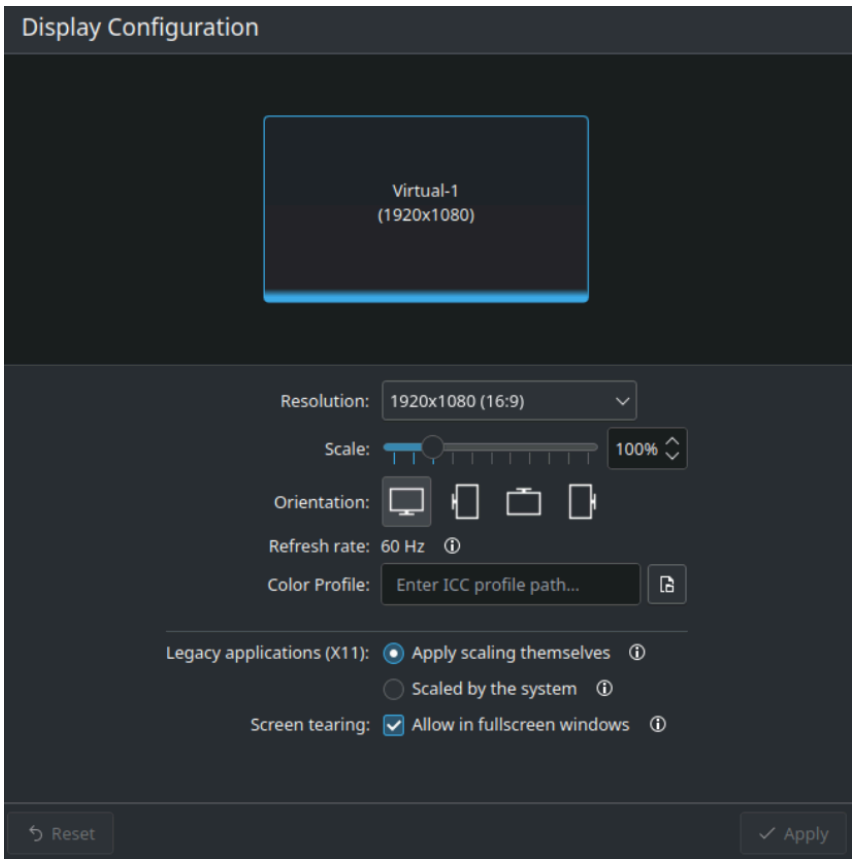


Figure 25: Monitor configuration within KDE systemsettings

KDE opted for a per-monitor paradigm, meaning settings are shown for the current monitor, with global settings being separated by a visual separator. The indication of which monitor is currently selected is shown by a blue indicator while other monitors are uncolored. The selected color is suitable for color blindness as different shades of blue are less susceptible to color blindness. People with monochromatic sight (full colorblindness) will also still be able to tell the difference based on shade, making this an optimal color defined in figure 16 Colorblind barrier-free color pallet by Color Universal Design (CUD). [51]–[53]

In Figure 26, the GNOME variant of the monitor configuration is shown.

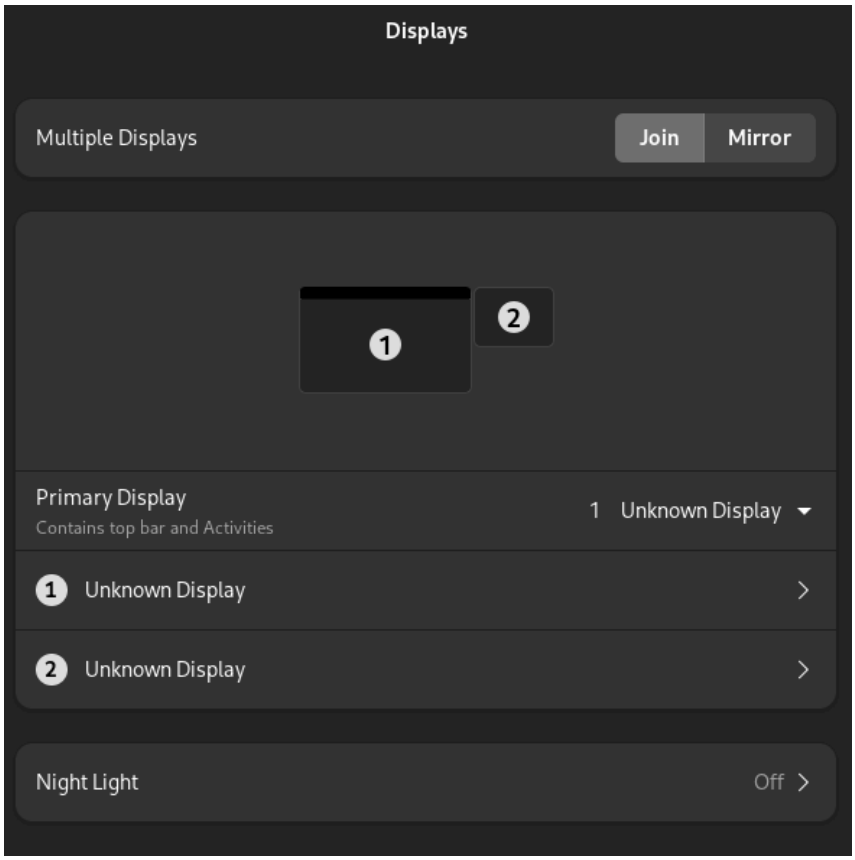


Figure 26: Monitor configuration within GNOME settings

Visible for GNOMEs implementation is the lack of direct configuration with multiple monitors. Instead, GNOME relies on submenus in order to change values such as resolution or refresh rate. At the same time, monitor independent settings like primary monitor or joining/mirroring displays are also shown in the overall menu. Noteworthy is also the lack of an apply or reset button when no actions have been taken. This is in contrast to the KDE implementation in Figure 25, which disables the buttons instead while still showing them visually. Another difference is the placing of the buttons when visible, KDE opted to show the reset and apply buttons at the bottom, while GNOME shows them at the top.

Figure 27 displays the configuration of per monitor specific settings.

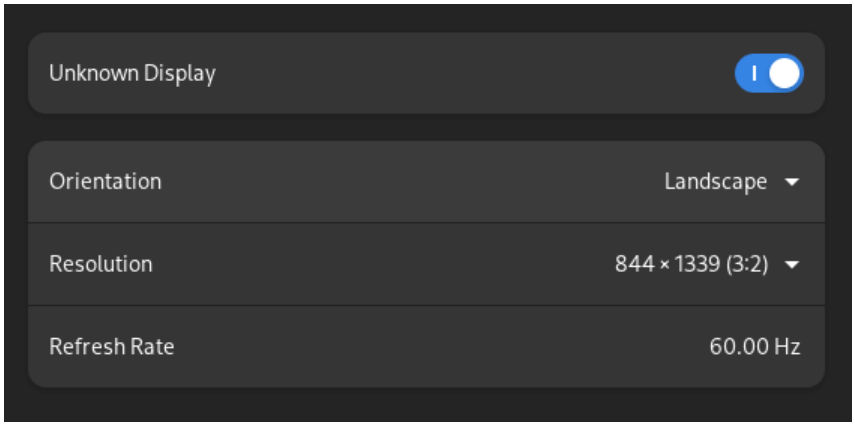


Figure 27: Configuration of a specific monitor within GNOME settings

5.2.2.1 Gaps in Configuration

While testing for environment differences, a significant difference between wlroots-based compositors to both KDE and GNOME was detected. Using a compositor like Hyprland, it is possible to create gaps between monitors to disallow direct mouse movement from one monitor to another. However, in both KDE and GNOME, a configuration with gaps results in an error which makes the configuration not applicable.

For ReSet, this would mean offering the same feature with restrictions to GNOME and KDE users, while still allowing other users to take advantage of their features set.

In Figure 28 and Figure 29, the error messages for gaps in GNOME and KDE are visualized.

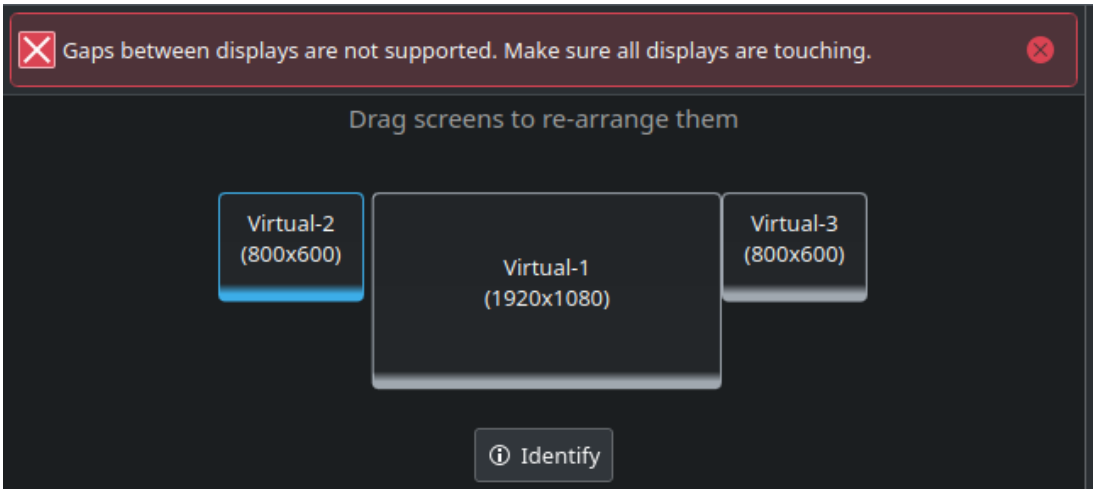


Figure 28: Monitor gaps error within KDE systemsettings

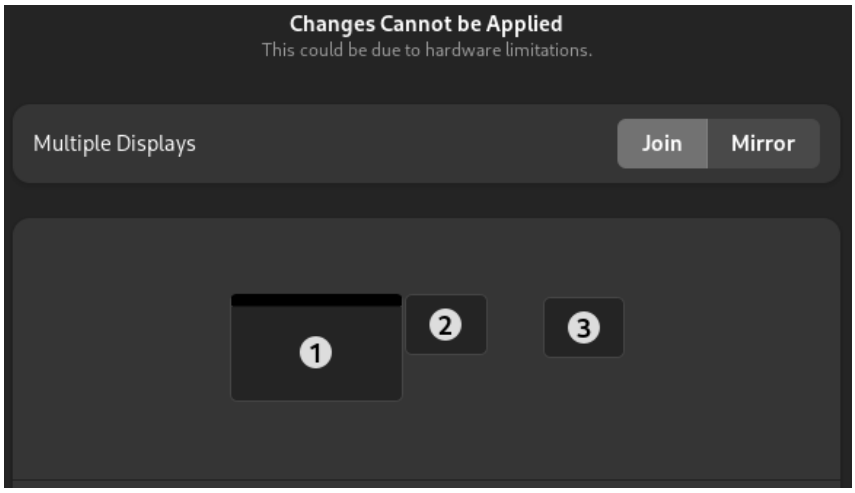


Figure 29: Monitor gaps error within GNOME settings

5.3 Keyboard Plugin

5.3.1 Analysis of different environments

In this section, the analysis of the keyboard plugin on various environments is discussed.

5.3.1.1 Hyprland

The keyboard layouts in Hyprland are stored in a hypr.conf file, which is a configuration file and can be modified using multiple ways.

One way to use this is hyprctl, a command-line tool that can be used to modify the setting. The problem with it is that the settings set by hyprctl are not persistent. That means that those changes disappear after a restart [54].

The other method is to write it directly into the hypr.conf file. Because the configuration file uses a custom configuration language called hyprlang, it is very hard to make changes in a specific area there because currently no parser exists for it that is written in Rust.

5.3.1.2 GNOME

In GNOME, it is possible to change keyboard layouts with gsettings or DConf. There are other tools like gconf, but they are replaced by DConf.

GSettings is a high-level API for application settings and serves as a front end to DConf. The command line tool gsettings (not to be confused with GSettings) can be used to access the GSettings API [55]. Another possibility is to use GIO, which is a library that provides multiple general-purpose functionality such as bindings to the GSettings API among others [56]. Because GIO is a dependency of GTK4, it can already be used from the code without any further setup. The advantage of using GIO instead of gsettings is that it can validate input and log error messages if something does not work. Another advantage is that gsettings returns a string that needs to be manually parsed with Regex or similar. GIO on the other hand returns a Variant object that can be casted into the desired type as long as it is correct.

DConf serves as a low-level configuration system for gsettings that stores key-based configuration details in a single compact binary format database [57]. Because gsettings is a layer for DConf, the keyboard plugin should uses DConf directly for setting the keyboard layouts. Combined with the DConf crate, which provides Rust bindings to DConf, the plugin can read the keyboard layouts [58]. This returns a string that needs to be parsed to get the keyboard layout and variant, which can be done with a simple Regex.

5.3.1.3 KDE

KDE stores its keyboard configurations in a file called kxkbrc. This text file is located in the config folder of the user and can be read using kreadconfig6 and written to using kwriteconfig6. These are tools provided by KDE to modify settings. [59] Unfortunately, no Rust crate provides bindings to these tools, so the plugin would have to use the command.

There was also a DBus interface that could be used to fetch keyboard layouts, but there was no way to set them.

5.3.2 Keyboard Limit

XKB is a system part of the X Window System and provides an easy way to configure keyboard layouts. On Wayland, XKB is the recommended way of handling keyboard input as well. [60] The problem with XKB is that it does not support more than four keyboard groups at the same time. A group in XKB consists of symbols, which are a collection of character codes and a group type that defines the type of symbols, e.g. Latin letters, Cyrillic letters etc. and make up a keyboard layout.

In Figure 30 the structure of the XKBState can be seen. The structure is 16 bits long and contains various information about the active state of a keyboard (or keyboards). Because KEYMASK and BUTMASK are not responsible for saving keyboard layouts, they are not explained any further. The important parts are bits 13 and 14 as they represent the keyboard group. It is not possible to just increase that number because the XKBState field is only 16 bits long and there are no bits left that could be used. Any attempt to increase the number of groups would require a change in the representation schema in XKB and other changes that would not be backwards compatible. [61]

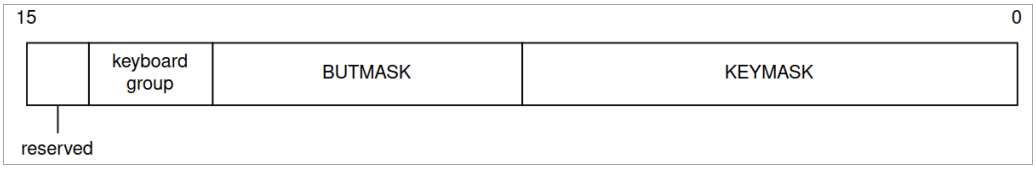


Figure 30: XKBState structure

5.3.3 Visualization

In this section, the different applications are analyzed. The main goal is to keep ReSet aligned with other settings in ReSet as well as other applications that use the GTK4 library.

5.3.3.1 GNOME

In Figure 31 the UI for the keyboard settings from GNOME control center is shown. They use a drag-and-drop list to order the keyboard layouts around and mark it with the six dots on the left. These grip indicators are commonly used in software as well as in physical products (ex. battery compartments) to indicate that something is draggable. The list is then grouped in a PreferenceGroup that provides a title and subtitle that explains what it is doing. Multiple of these PreferenceGroups groups are stacked vertically in a ScrollWindow that allow vertical scrolling.

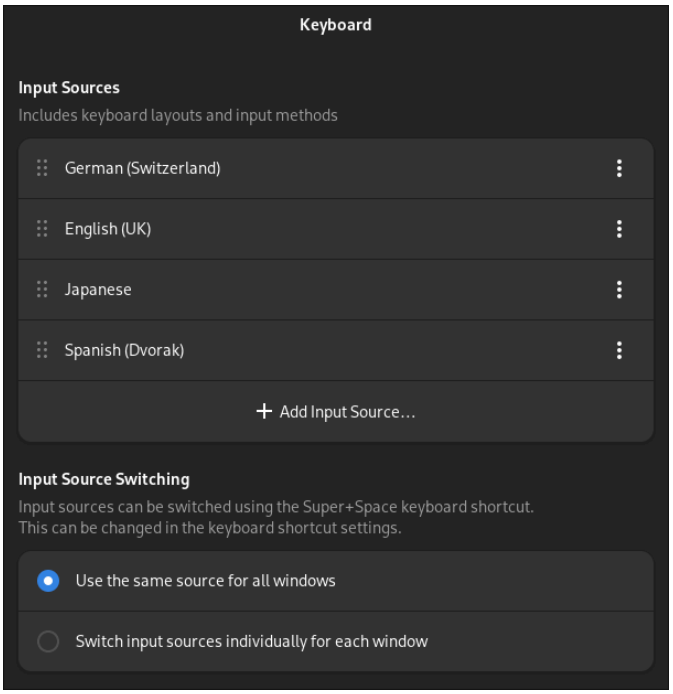


Figure 31: Gnome keyboard settings

New keyboard layouts can be added with the button at the very bottom of the list. It opens a dialog menu as seen in Figure 32 that contains a list of every keyboard layout available in the system. After selecting a keyboard layout, it can be added with the Add button on the top right.

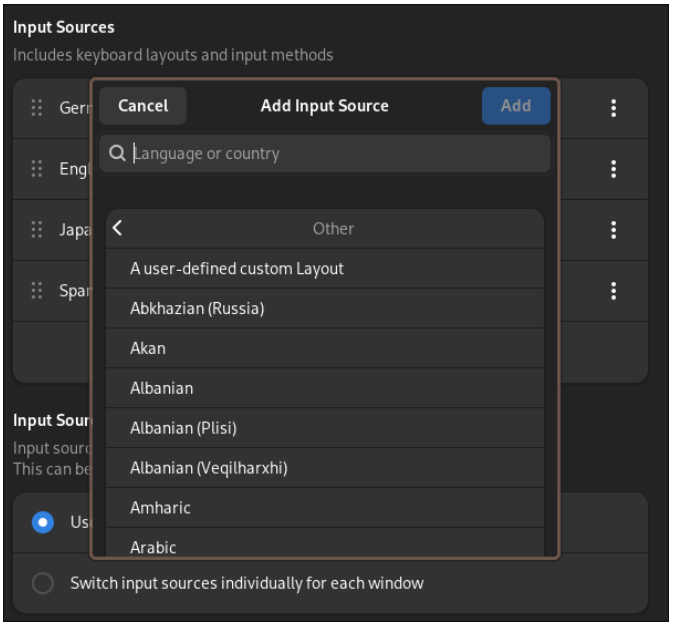


Figure 32: Gnome add keyboard layout settings

5.3.3.2 KDE

In Figure 33 the UI for KDE system settings can be seen. The keyboard layout list is a list that can only be modified using the few buttons above it. It still allows the same behavior as a drag-and-drop list but is more explicit because each feature is written out.
The keyboard list is placed inside a border and is located in a grid-like fashion with other settings.

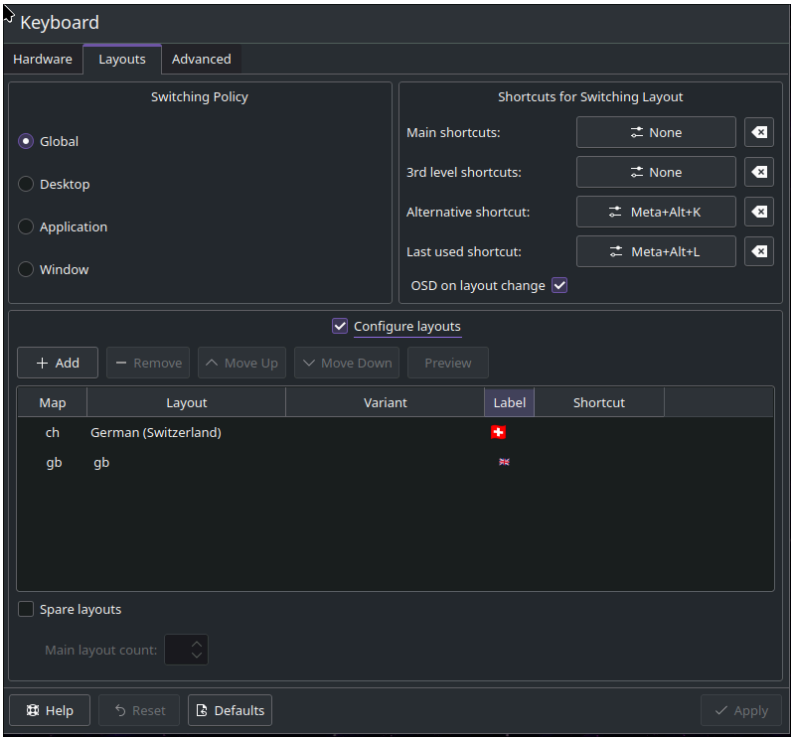


Figure 33: KDE keyboard settings

The Add button opens a new dialog as seen in Figure 34 in which users can search and insert new keyboard layouts from the list. It also has a preview functionality that allows users to see each key mapping visually.

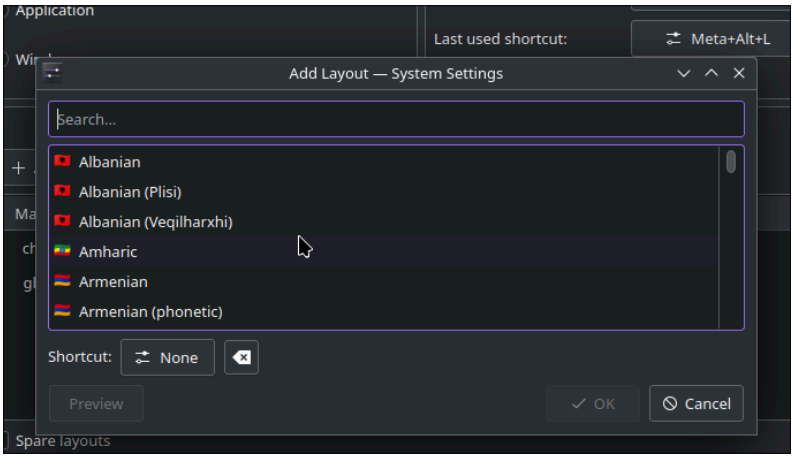


Figure 34: KDE add keyboard layout settings

6 Exemplary Plugin Results

6.1 Plugin UI Mockups

In this section, two mockups of potential plugins are created. The first plugin allows the user to change monitor settings and the second plugin allows keyboard settings. The mockups were created using Mockflow, which is an online tool for creating mockups. [62]

6.1.1 Monitor Plugin Mockup

While settings applications offer many different approaches and user interfaces, the monitor settings look very similar across practically all applications that were referenced. There is always a drag-and-drop area where the user can align monitors. Because there are near-infinite variations on how monitors can be ordered, it makes sense to use a widget where the user can visually see the result. This makes it very intuitive compared to having users set the offset as a number. A user is most likely to be familiar with a drag-and-drop feature, which makes it easy to understand how to use it. [63]
The "Cancel" and "Apply" buttons are placed at the bottom of the drag-and-drop area because it follows the natural downward flow of the user. A feature that can be implemented into that area is the ability to change commonly used monitor settings inside the area like resolution and orientation. Many applications only offer these settings after scrolling a bit, which is not very user-friendly.

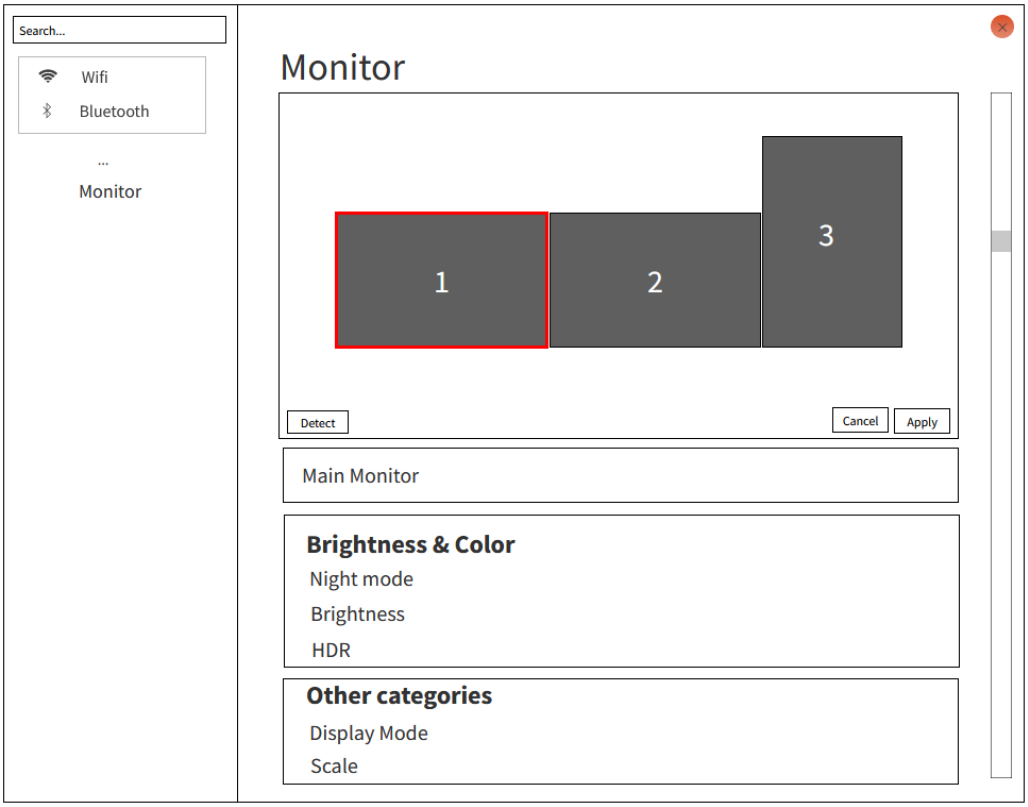


Figure 35: Monitor plugin mockup

Below this area, the user can set the main monitor and change other settings such as the brightness and night mode. The order is inspired by Windows 11. The further down the user scrolls the less common and more advanced the settings should become. This should allow users to easily find settings that are often changed. The order seen in Figure 35 does not hold much meaning and should be taken only as a filler. The specific form controls for each setting are not considered for now.
There are a few settings that are handled differently or are just plainly not available in certain desktop environments. To address these variations, the UI will check which desktop environment is currently running and display the correct widgets accordingly.

For reference Figure 36 visualizes the Windows 11 monitor configuration.

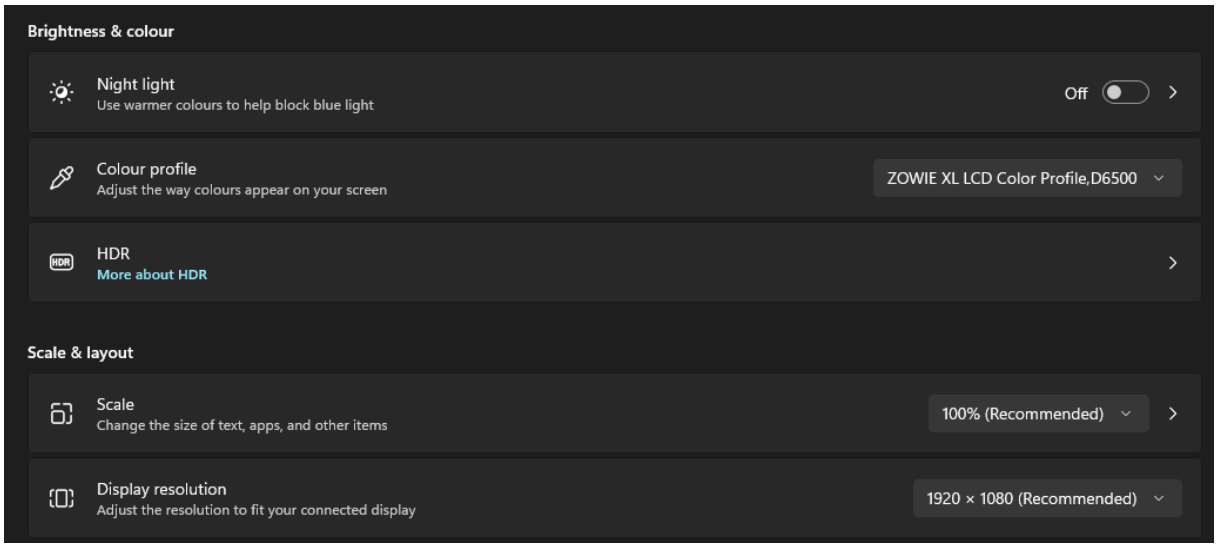


Figure 36: Windows monitor settings

6.1.2 Keyboard Plugin

While many keyboard settings can be implemented, only the very basic ones will be in the mockup. Any further settings can be implemented further down the navigation page or a group of settings can be moved into a tab to keep the main page clean and simple.

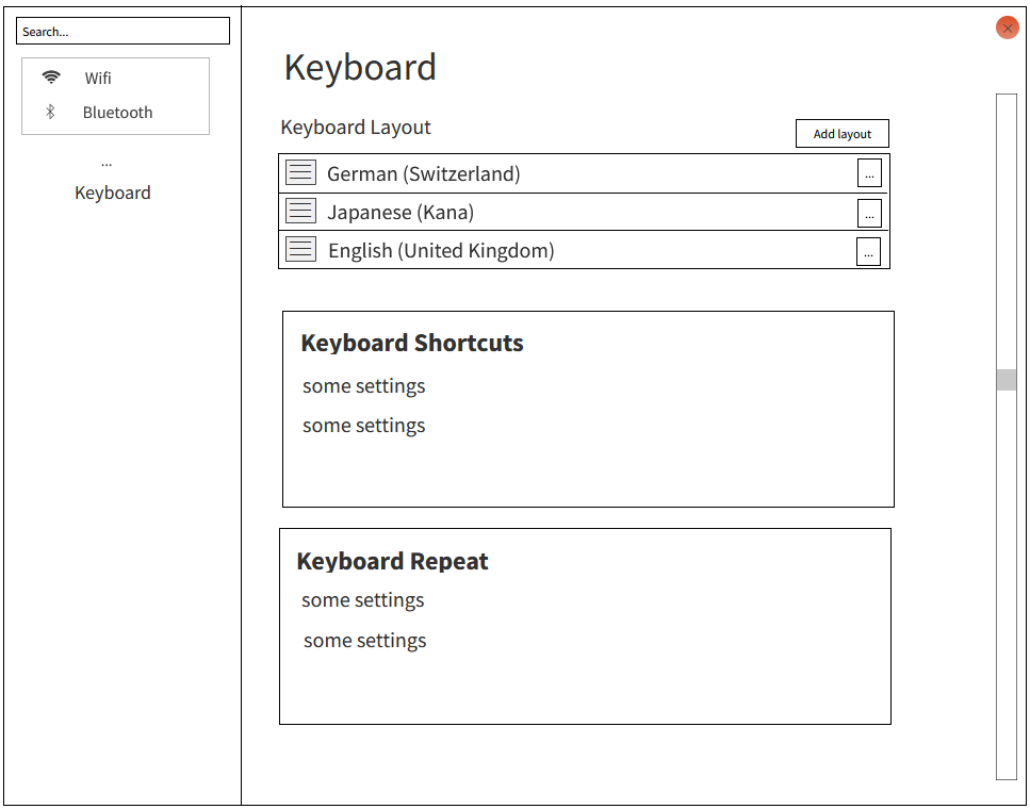


Figure 37: Mock of keyboard plugin

The keyboard plugin UI starts with a list of currently available keyboard layouts. Because the user can have multiple keyboard layouts which have to be sorted by importance, the best form of control for this is a sortable list. This is because, at the same time, the layout in the first place is also the default keyboard layout. The usage of such form control can be seen in Windows and GNOME Settings.

As defined in Section 5.3.2, there is a maximum of keyboard layouts a system can have configured at the same time. For example, if a user has ten keyboard layouts configured, only the first four can be switched to. The switchable keyboard layouts will be colored to indicate if they are active or not.

In Figure 38, GNOME's keyboard list can be seen.

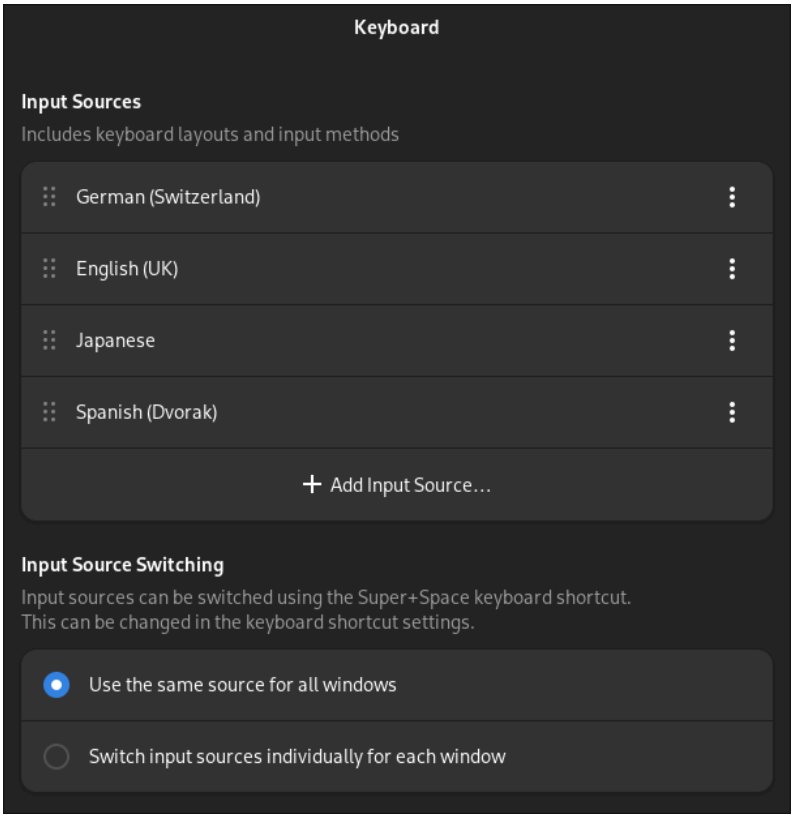


Figure 38: GNOME keyboard layouts

The user can also add a new layout with "Add layout" button, which places the new layout at the bottom of the list. By clicking on the three dots on the right, the user can open a context menu to remove the layout.

The user is shown a list of all available keyboard layouts in the system. The arrow icon indicates that there are multiple variations of the same keyboard such as "de-us", "de-dvorak", "de-mac" and many more. To reduce cluttering, only the main layout is shown, and all other variations are hidden until the user clicks on the row. This immediately hides all other layouts and only shows the layout and its variations. A new back button will be prepended to the list to allow the user to go back to the main list. This feature is inspired by GNOME Settings and has been expanded upon. GNOME Settings only has this feature for "English (UK)" and "English (US)", while all other layouts are shown at the same time. This makes it harder for the user to find a specific layout because filtering the layouts with the search bar can still show layouts of another language, which have a similar name. In Figure 39 a mockup of this feature can be seen.

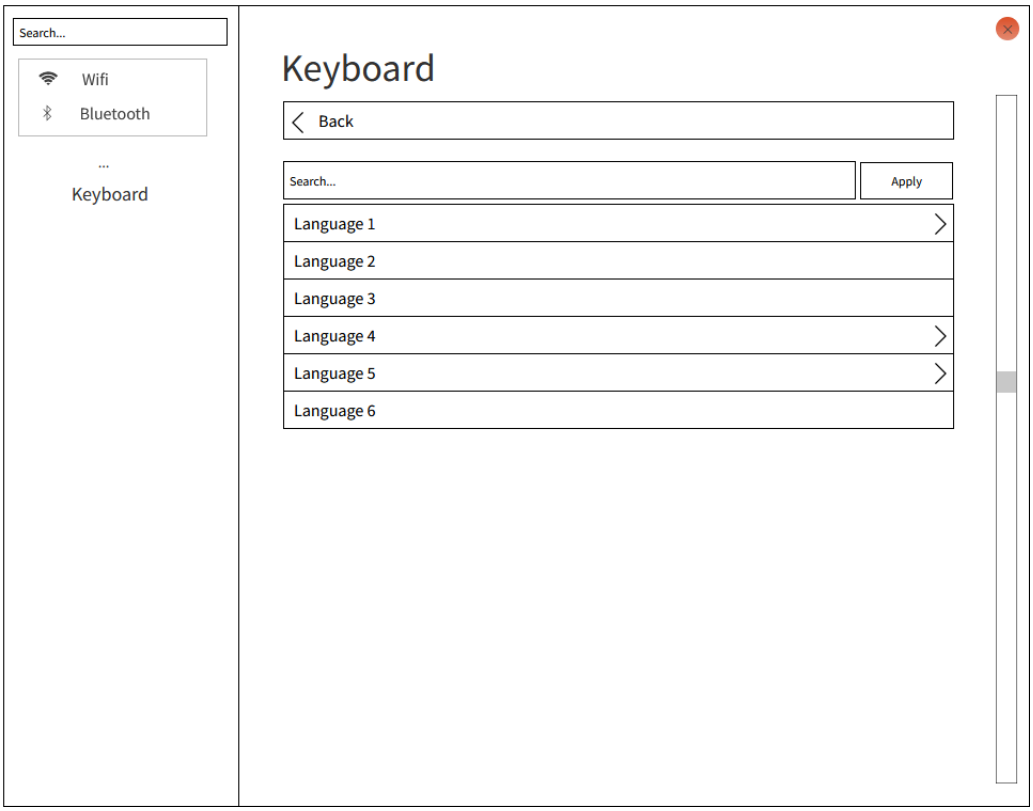


Figure 39: Mock of keyboard add layout plugin

6.2 Monitor Plugin Implementation

This section covers the implementation of the monitor plugin for ReSet.

6.2.1 Data Transfer Object (DTO)

In order to facilitate the usage of all the different fetching methods and different structures, ReSet needs a way to convert the fetched structures to universal versions in order to provide a consistent, API-compatible structure as a DBus endpoint. Just as with ReSet itself, the goal of this plugin is to offer both a DBus endpoint for functionality and a graphical user interface for human interaction. Should a user not be satisfied with the provided interface, they can just use the endpoint to create their own.

In Listing 50 the fields of the display struct used for the DBus connection and the user interface is visualized.

```rust
#[repr(C)]
#[derive(Debug, Clone, Default)]
pub struct Monitor {
    pub id: u32,
    pub enabled: bool,
    pub name: String,
    pub make: String,
    pub model: String,
    pub serial: String,
    pub refresh_rate: u32,
    pub scale: f64,
    pub transform: u32,
    pub vrr: bool,
    pub primary: bool,
    pub offset: Offset,
    pub size: Size,
    pub drag_information: DragInformation,
    pub mode: String,
    pub available_modes: Vec<AvailableMode>,
    pub uses_mode_id: bool,
    pub features: MonitorFeatures,
}
```

Listing 50: Display struct

The monitor struct holds all important information within a single indirection, while user interface information (DragInformation), environment feature flags and monitor modes are combined into separate data structures. This allows every environment to be converted, while they still share differing approaches to monitors.

6.2.2 Fractional Scaling

Fractional scaling is implemented according to the fractional-scale-v1 Wayland protocol. [64] This protocol defines how scaling values will be interpreted by the environment. The specification defines that supported scales must be of a fraction with a denominator of 120. In other words, incrementing a scaling value would mean multiplying the base value with 120, then proceeding to increase or decrease this number before dividing by 120 again. The result will be a new scaling fraction within the constraints of the protocol.

In Figure 40, an example is visualized.

$$1.00 * 120 = 120$$
$$120 + 6 = 126$$
$$\frac{126}{120} = 1.066666...$$

Figure 40: Example for a valid fractional scale

For fractional scaling, there is another requirement that a user-selected scale must adhere to. This requirement defines that a chosen scale must be a divider for both the width and the height of the resolution. In other words, the division must result in a full integer and may not leave any fractions. Using scaling with noninteger resolutions would mean slightly different aspect ratios for every scale, which creates an inconsistent user experience.

The challenge with this approach is that a user may no longer enter any random number as a scale. ReSet would either need to provide a check for every entered scale or provide a range of valid scales to choose from. Existing solutions such as GNOME and KDE resorted to using a selection of scales by dropdown and slider respectively.

In Figure 41 the options from KDE and GNOME are visualized.

Figure 41: GNOME and KDE scaling

The GNOME variant offers a simple dropdown with a percentage value, without any arbitrary scale. This ensures the user cannot under any circumstance enter an invalid scale, removing a potential error. KDE offers both a slider that snaps to predefined percentages, while also offering a user input for arbitrary percentages. Noteworthy is that the user input automatically changes to a supported value when applying the configuration. In Figure 41 the scaling was adjusted to 132% up from 130%, in this case, the apply button is disabled as 132% is not a possible scale and the nearest possible scale (130%) is already applied.

As ReSet offers scaling for multiple environments, the plugin should also support arbitrary scales depending on the environment. For ReSet, the implementation was handled with a libadwaita SpinRow, this widget provides both arbitrary user input and an increment and decrement button.[65] As ReSet implements arbitrary scaling, it would also require a check for valid scales with proper user feedback.

The chosen method was to simply snap to the closest possible valid scale and provide an error banner if no scale can be found.

The implementation itself is a loop that increments the by 120 divided scale until either a valid scale is found, or the maximum iterations (0..amount) have been reached.

In Listing 51, the body of the search_nearest_scale function within the plugin is visualized.

```
// reverse x for the second run
let reverse_scale = if reverse { -1.0 } else { 1.0 };
for x in 0..amount {
    // increment here does not equal to increment of 1, but 1/120 of an increment
    // specified at: https://wayland.app/protocols/fractional-scale-v1
    let scale_move = if direction {
        (*search_scale + (x as f64) * reverse_scale) / 120.0
    } else {
        (*search_scale - (x as f64) * reverse_scale) / 120.0
    };

    let maybe_move_x = monitor.size.0 as f64 / scale_move;
    let maybe_move_y = monitor.size.1 as f64 / scale_move;
    if maybe_move_x == maybe_move_x.round()
        && maybe_move_y == maybe_move_y.round()
        && scale_move != monitor.scale
    {
        *search_scale = scale_move;
        *found = true;
        break;
    }
}
```

Listing 51: Search the nearest scale function

6.2.3 Drag-and-Drop

A common configuration is the arrangement of monitors. A user might have a physical setup where the leftmost monitor is considered the second monitor within the operating system/environment. This requires the user to either re-configure the physical cable arrangement, or preferably, just drag the monitor to the correct position with a user interface.[63]

GTK does not offer a direct way to draw arbitrary shapes, however, it does offer cairo integration, which is a low-level drawing framework that can be used to draw pixels onto a GTK DrawingArea. [66]

In order to both draw the shapes and calculate the eventual user offsets, a coordinate system is required. For cairo this is a top-left to bottom-right system. This means that x increases towards the right and y increases towards the bottom.

In Figure 42, the monitor cairo coordinate system is visualized.

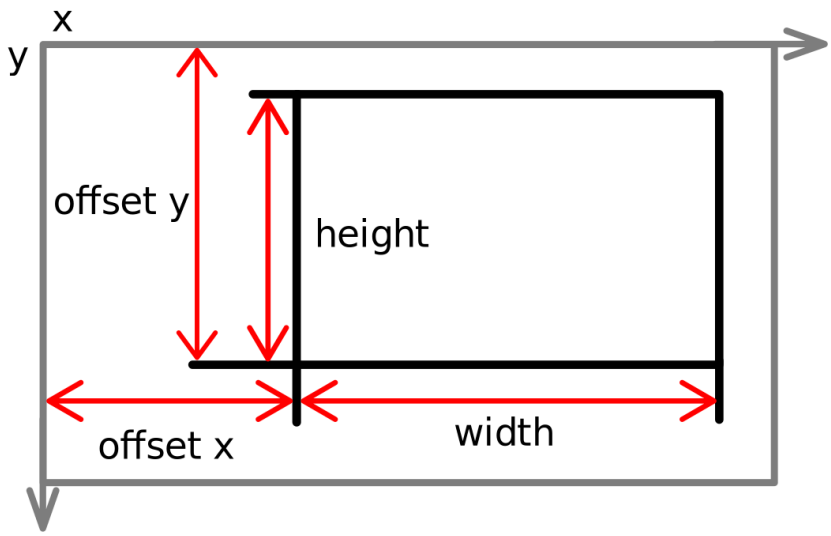


Figure 42: Visualization of the monitor cairo coordinate system

For a simple drawing of the monitor, this coordinate system would be trivial, however, it is important to understand it in detail when providing drag-and-drop operations, which have to constantly apply transforms to these shapes. At the same time, any potential monitor overlaps have to be handled, as well as providing snapping functionality in order to auto-align monitors.

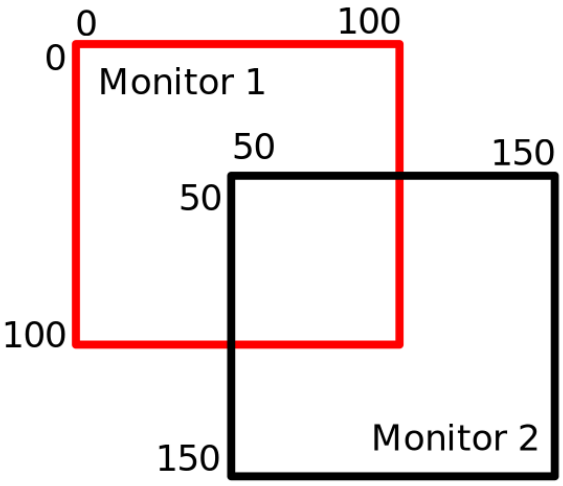
On top of this, the coordinate system of the monitor itself has the y-axis inverted, as all environments expect positive y to be “upwards”.

Intersections can be seen by two conditions per axis. If both axes have at least one condition evaluated as false, then an overlap has occurred. In Listing 52 and Figure 43 the conditions and an example overlap are visualized.

```rust
pub fn intersect_horizontal(&self, offset_x: i32, width: i32) -> bool {
    // current monitor left side is right of other right
    let left = self.border_offset_x + self.offset.0 >= offset_x + width;
    // current monitor right is left of other left
    let right =
        self.border_offset_x + self.offset.0 + self.width
            <= offset_x;
    !left && !right
}

pub fn intersect_vertical(&self, offset_y: i32, height: i32) -> bool {
    // current monitor bottom is higher than other top
    let bottom = self.border_offset_y + self.offset.1 >= offset_y + height;
    // current monitor top is lower than other bottom
    let top =
        self.border_offset_y + self.offset.1 + self.height
            <= offset_y;
    !bottom && !top
}
```

Listing 52: Simplified implemented overlap conditions



"Monitor 2" starts at x coordinate 50 which is before the endpoint of "Monitor 2",
this marks the condition left of the function intersect_horizontal as false.
"Monitor 2" starts at y coordinate 50 which is before the endpoint of "Monitor 2",
this marks the condition bottom of the function intersect_horizontal as false.
Figure 43: Visualization of the monitor overlap

The dragging itself without calculations is trivial, as GTK allows the use of event handlers including drag-and-drop handlers on each cairo shape which represent the monitors.

6.2.4 Snapping

A quality of life feature is the ability to allow users to be inaccurate with their monitor positioning and then automatically snapping the monitor towards adjacent ones.

For example, the Windows desktop paradigm offers desktop icons. A user can rearrange them via drag-and-drop, and in this action, the user does not need to be accurate, instead, they can approximately drag the icon to the target endpoint and drop it at this position. The desktop icon system will then reposition the icon towards the correct place within the grid system.

In Figure 44 and Figure 45, the dragging mechanism is visualized.

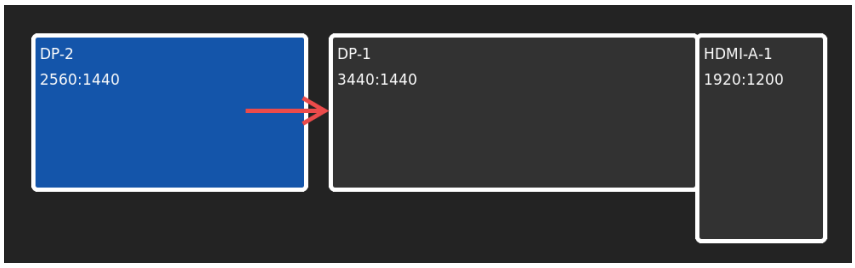


Figure 44: Monitor dragged towards the right

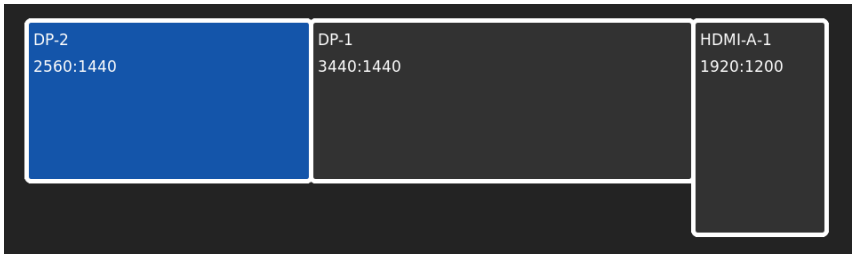


Figure 45: Monitor snapped to another monitor

To ensure this functionality works as expected, the potential snapping functionality checks for overlaps, as well as the no-gaps requirement in the KDE and GNOME environments. Specifically, the dragged monitor is checked against all other monitors, with the smallest gap in both vertical and horizontal being considered. In other words, a monitor can snap once in horizontal direction and once in vertical at the same time.

In Listing 53, the code to check for the smallest distance in for vertical snaps is visualized.

```
// find smallest difference
let top_to_bottom = endpoint_top.abs_diff(endpoint_other_bottom);
let bottom_to_top = endpoint_bottom.abs_diff(endpoint_other_top);
let top_to_top = endpoint_top.abs_diff(endpoint_other_top);
let bottom_to_bottom = endpoint_bottom.abs_diff(endpoint_other_bottom);
let min = cmp::min(
  cmp::min(top_to_bottom, bottom_to_top),
  cmp::min(top_to_top, bottom_to_bottom),
);
// snap to the smallest distance if smaller than SNAP_DISTANCE
if min < SNAP_DISTANCE {
  match min {
    x if x == top_to_bottom => {
      snap_vertical = SnapDirectionVertical::TopBottom(
        endpoint_other_bottom,
        endpoint_other_left,
        other_width,
      );
    }
    x if x == bottom_to_top => {
      snap_vertical = SnapDirectionVertical::BottomTop(
        endpoint_other_top,
        endpoint_other_left,
        other_width,
      );
    }
    x if x == top_to_top => {
      snap_vertical = SnapDirectionVertical::TopTop(endpoint_other_top);
    }
    x if x == bottom_to_bottom => {
      snap_vertical = SnapDirectionVertical::BottomBottom(endpoint_other_bottom);
    }
    _ => unreachable!(),
  }
}
```

Listing 53: Vertical snapping check within the monitor_drag_end function

The endpoints represent the edges of both monitors, with the “other” monitor being the monitor that is currently checked against.

6.2.5 Feature Disparities

Differing environments offer a range of features, this forces ReSet to offer dynamic feature checking in order to only show compatible features with the current environment. As an example, both GNOME and KDE have a concept of a primary monitor, which should be the default monitor for starting applications, widgets and more. However, tiling environments like Hyprland do not offer a primary monitor, as these types of environments handle these use cases with explicit focus. If your focus is currently on “Monitor 1”, then the application will also be started on this monitor. Similarly, widgets and panels are created explicitly, meaning you would need to define one or more monitors where your widget or panel should be.

The solution for this problem is a set of feature flags that are introduced during the conversion from environment-specific data to DBus-compatible generic monitor data. Within this data, the struct visualized in Listing 54 is included.

```rust
pub struct MonitorFeatures {
    pub vrr: bool,
    pub primary: bool,
    pub fractional_scaling: bool,
    pub hdr: bool,
}
```

Listing 54: Monitor feature flag struct

VRR/Variable-Refresh-Rate: This configures the monitor to automatically lower or increase the monitor’s refresh rate depending on the current performance of the graphics card. In games, this means coupling the monitor refresh rate to the frames per second generated by the graphics card/ core processing unit. This feature is currently supported by KDE, GNOME (experimental), Xorg and all wlroots-based compositors such as Hyprland or Sway.

HDR/High Dynamic Range: Luminosity ranges between the brightest and darkest areas within a scene. This setting is a longstanding issue within Linux environments. As such this feature is currently only available as an experimental feature on KDE.

Fractional Scaling: Scaling with a non-integer number. The challenge with this is that a full integer will always result in an integer resolution. However, as explained in Section 6.2.2, using a fractional number will result in fractional resolutions which is not applicable. Hence, a set of rules must be followed in order to avoid this issue. Currently, KDE, GNOME (experimental) and wlroots-based environments support fractional scaling.

6.2.6 Redraws

In order to show changes within the plugin, GTK needs to redraw the widgets. Depending on the change, this could be something insignificant such as a number, or replacing the entire set of settings when clicking on another monitor. Redrawing a singular number within a widget is covered by GTK, while the selection of the monitor must be handled by the plugin. For this problem, a singular function was chosen that repopulates the settings per monitor on selection. The function is visualized in Listing 55.

```rust
pub fn get_monitor_settings_group(
    clicked_monitor: Rc<RefCell<Vec<Monitor>>>,
    monitor_index: usize,
    drawing_area: &DrawingArea,
) -> PreferencesGroup {
    let settings = PreferencesGroup::new();
    // add settings
    settings
}
// each call to this function replaces the current group with a new one
```

Listing 55: Section of the get_monitor_settings_group function

It is important to note, that the covered feature differences in Section 6.2.5, lead to differing functions being used for different environments. As such ReSet checks for environment differences inside this function via the `$XDG_CURRENT_DESKTOP` environment variable.

In Listing 56, an example for a dynamic feature is visualized.

```rust
// inside "get_monitor_settings_group"
let scaling_ref = clicked_monitor.clone();
add_scale_adjustment(monitor.scale, monitor_index, scaling_ref, &settings);

// scale handler
pub fn add_scale_adjustment(
    scale: f64,
    monitor_index: usize,
    scaling_ref: Rc<RefCell<Vec<Monitor>>>,
    settings: &PreferencesGroup,
) {
    // Different environments allow differing values
    // Hyprland allows arbitrary scales, Gnome offers a set of supported scales
    match get_environment().as_str() {
        "Hyprland" => {}, // hyprland implementation
        "GNOME" => {}, // GNOME implementation
        "KDE" => {}, // KDE implementation
        _ => match get_wl_backend().as_str() {}, // match protocols for others
    };
}
```

Listing 56: Dynamic feature functionality for the ReSet monitor plugin

Another problematic section of the plugin is the drawing area, as this requires separate redraws as well. For this, redraws are queued for any action taken that changes the appearance within the drawing area. Possible actions include dragging, snapping, change of transform, change of scaling, change of resolution and monitor selection. Each of these actions will cause the drawing area to be redrawn in order to show the result of the chosen action.

The last piece of redraw is the arrangement of the monitors. Each time a user changes the resolution, the transform of a monitor or the scaling, this will be reflected on the interface. This requires the plugin to calculate a new arrangement for all monitors, as the constellation of monitors should not change simply because the user changed the resolution of a single monitor. As an example, consider a situation with three monitors aligned in a row. When the user changes the resolution or transforms either the first or the second monitor, the monitors to the right would need to be moved either towards or away from the changed monitor. In Figure 46 the example is visualized.

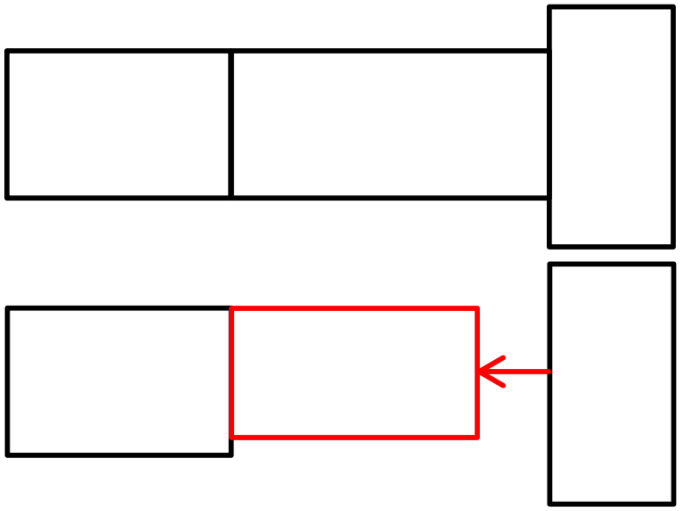


Figure 46: Example of a monitor resize within an arrangement

Further issues arise with the inclusion of the y-axis for potential resizes. Due to multiple axes being possible, it is now necessary to check for overlaps that can be caused by to shifting of other monitors after resizing. Consider the second example with four monitors shown in Figure 47.

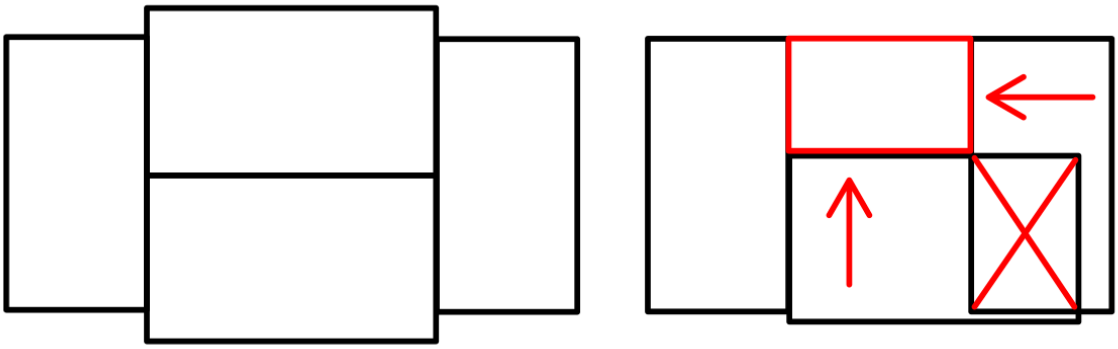


Figure 47: Example of an overlap caused by monitor resize

This overlap is not applicable as a configuration and needs to be resolved. There are multiple possible solutions to this problem. A very simple approach would be to just reorder the monitors as soon as the resolution or transform of any monitor happens. The drawback with this approach is the fact that it breaks the arrangement a user created. This would mean that the monitor previously situated in the center might now be on the left. In Listing 57 the solution is visualized.

```
for monitor in monitors.iter_mut() {
    // handles rotation in order to calculate x and y
    let (width, _) = monitor.handle_transform();
    monitor.offset.0 = furthest;
    // move all monitors to one axis -> hence no overlap possible
    monitor.offset.1 = original_monitor.offset.1;
    furthest = monitor.offset.0 + width;
}
```

Listing 57: Code snippet for the furthest calculation of monitors

For this plugin, the more complicated variant of checking for overlaps and only rearranging the overlapped monitors was chosen. With this, users only see monitors moving away from their assigned spot, if their size cannot fit in the same spot after rearranging.

This variant includes four incremental steps. The first step is calculating the resolution or transform difference of the changed monitor, as well as the right side of the rightmost monitor. Step two handles the movement of all monitors that exist on the right or the bottom of the changed monitor. Step three handles the checking for overlaps among all monitors. This is done by looping over all monitors and checking each monitor against the other.

In Listing 58, step three is visualized.

```
// Step 1 and 2: omitted
// (false, false)
// first: already used flag -> don't check for overlaps
// second: bool flag to indicate overlap
let mut overlaps = vec![(false, false); monitors.len()];
// check for overlaps
for (index, monitor) in monitors.iter().enumerate() {
  for (other_index, other_monitor) in monitors.iter().enumerate() {
    if monitor.id == other_monitor.id || overlaps[other_index].0 {
      continue;
    }
    let (width, height) = other_monitor.handle_transform();
    let intersect_horizontal = monitor.intersect_horizontal(
      other_monitor.offset.0 + other_monitor.drag_information.border_offset_x,
      width,
    );
    let intersect_vertical = monitor.intersect_vertical(
      other_monitor.offset.1 + other_monitor.drag_information.border_offset_y,
      height,
    );
    // don't rearrange the rightmost monitor -> otherwise gap between monitors
    let is_furthest = furthest == monitor.offset.0 + monitor.size.0;
    if intersect_horizontal && intersect_vertical && !is_furthest {
      overlaps[index].1 = true;
    }
  }
  overlaps[index].0 = true;
}
// Step 4: omitted
```

Listing 58: Overlap detection for the rearrangement-function of monitors

The last step is to remove all overlaps. This is done by moving the overlapping monitors to the rightmost side calculated in step one.

In Figure 48, the overlap shown in Figure 47 is fixed.

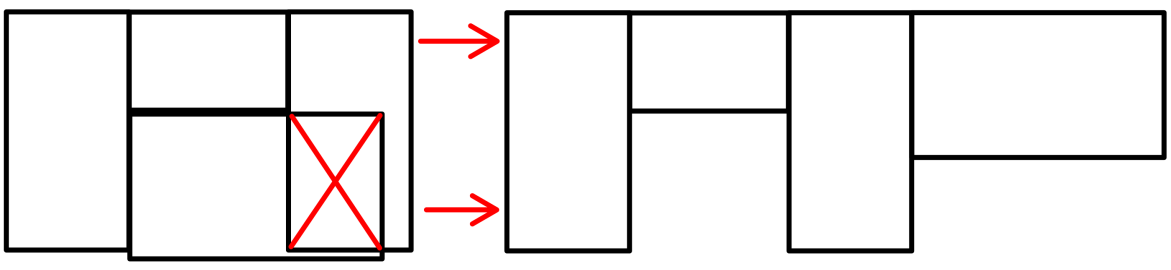


Figure 48: Example of a rearranged monitor configuration

6.2.7 Resulting User Interface

This section covers the resulting plugin user interface.

As a baseline, the plugin is shown in Figure 49.

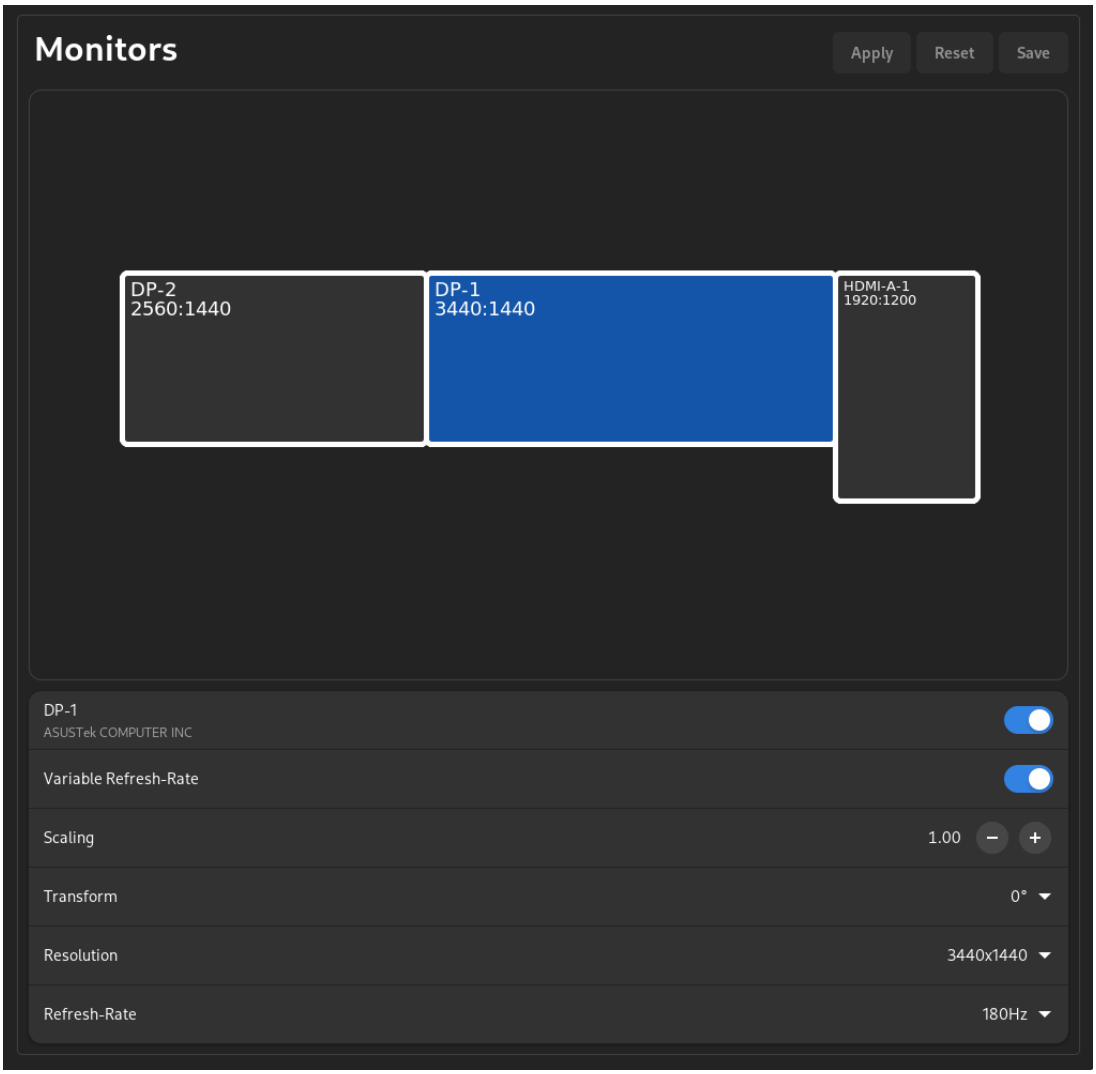


Figure 49: ReSet monitor plugin on Hyprland

The buttons for applying, resetting and saving were placed at the top in order to provide a visually pleasing appearance while also indicating to the user that these buttons are for the entire configuration of all monitors. Including the constraints provided by the implementation, this alignment is also cohesive to the positioning of similar buttons in the Bluetooth and Wi-Fi functionality of ReSet. Compared to the mockup in Section 6.1.1, this differs from placing the buttons within the drag-and-drop visualization.

Similar to Section 5.2.1.3, the plugin interface will always show the current monitor configuration visually, with a blue indicator showing the currently selected monitor. This ensures that users are always immediately aware of how their configuration will translate to the real world, including resolution changes and transformation changes.

Below the visual interface is a list of configurations for the currently selected monitor with the first bar showing the name and the make of the selected monitor. On the same bar, the selected monitor can also be turned off, if the overall configuration has more than one monitor available.

Options for features such as resolutions, refresh rates, fractional scaling and more were implemented with either a dropdown menu comparable to the GNOME implementation discussed in Section 5.2.1.4 as the same toolkit was used, while the scaling implementation differs depending on the environment. The scaling feature is discussed in detail in Section 6.2.2.

In Figure 50 and Figure 51, the GNOME and other environment specific scaling widgets are visualized.

Figure 50: ReSet monitor scaling on GNOME

Figure 51: ReSet monitor scaling on Hyprland

The results of this user interface will be tested in Section 14.5.2.

6.3 Keyboard Plugin Implementation

6.3.1 Implementations

In this section, the implementation of the keyboard plugin is discussed.

6.3.1.1 Hyprland Implementation

Because there is no hyprlang parser in Rust, a separate file is being created that holds the keyboard layout setting. This file has to be referenced from the main configuration file like in Listing 59.

```
source=~/Documents/dotfiles/hypr/input.conf
```

Listing 59: Include separate config file in hypr.conf

This is because with this solution, ReSet does not need to parse its content, but can just override the whole file each time a change is made. A string that follows the Hyprland syntax is built from the keyboard layouts in Listing 60 and is written to the file.

```
let mut layout_string = String::new();
let mut variant_string = String::new();
for x in layouts.iter() {
    layout_string += &x.name;
    layout_string += ", ";
    if let Some(var) = &x.variant {
        variant_string += &var;
    }
    variant_string += ", ";
};

layout_string = layout_string.trim_end_matches(", ").to_string();
variant_string = variant_string.trim_end_matches(", ").to_string();

let string = format!("input {{
    kb_layout={}
    kb_variant={}
}}", layout_string, variant_string);

input_config.set_len(0).expect("Failed to truncate file");
input_config.write_all(string.as_bytes()).expect("Failed to write to file");
```

Listing 60: Convert keyboard layouts to string

In Listing 61 the contents of the input config are shown. Each combination of same index from kb_layout and kb_variant represents a keyboard layout.

```
input {
    kb_layout=ch, jp, gb, fi
    kb_variant=, , , winkeys
}
```

Listing 61: Input configuration

ReSet has to know where the input config is located which has to be provided by the user. This path can be provided in the local config folder of a user in a file called Reset.toml. This file contains ReSet relevant configuration. The user has to provide the path to the input config there like Listing 62 or else ReSet would not know where to write the changes to. If the file does not exist, a new one will be created. If the path is not provided, the keyboard plugin will use a predefined path.

```
[Keyboard]
path = "/home/felix043/Documents/dotfiles/hypr/input.conf"
```

Listing 62: Path to input configuration in ReSet configuration

6.3.1.2 GNOME Implementation

The first try was using dconf_rs because GSettings is a frontend API to DConf and there was no reason to go through that indirection. In Listing 63 the setting string was fetched using DConf which then had to be matched using a Regex pattern. Unfortunately, it is not possible to use it to set a new keyboard config because dconf_rs wraps the value to be set with an apostrophe that cannot be parsed by DConf. The only solution is to use a command to set the keyboard layouts as seen in Listing 64.

```
let result = dconf_rs::get_string("/org/gnome/desktop/input-sources/sources");
// has format of: [('xkb', 'ch'), ('xkb', 'us'), ('xkb', 'ara+azerty')]

let pattern = Regex::new(r"[a-zA-Z0-9_+-]+").unwrap();
let matches = pattern.captures_iter(layouts.as_str())
```

Listing 63: Parsing GNOME keyboard layouts

```
Command::new("dconf")
  .arg("write")
  .arg("/org/gnome/desktop/input-sources/sources")
  .arg(all_layouts)
  .status()
  .expect("failed to execute command");
```

Listing 64: Write new layouts with dconf

Therefore dconf_rs was replaced by GIO. The advantages have already been discussed in Section 5.3.1.2. GIO provides very convenient bindings to the GSettings API. In Listing 65 the layouts are fetched, and its type is checked. If the layout_variant is not an array of two strings an empty array will be returned. Otherwise, a generic get function is called with type `Vec<(String,String)>>` to parse it into desired structure. This solution is a lot more elegant than the Regex because it is more human-readable and better performant as Regex processing can be slower due to its pattern matching which involves operations like backtracking.

```
let input_sources = gtk::gio::Settings::new("org.gnome.desktop.input-sources");
let layout_variant = input_sources.value("sources");

if layout_variant.type_() != VariantType::new("a(ss)").unwrap() {
  return kb;
}

let layouts = layout_variant.get::<Vec<(String, String)>>().unwrap();
```

Listing 65: Get keyboard layouts with GIO

In Listing 66 the code for writing the new keyboard layouts can be seen.

```
let variant = Variant::from(all_layouts);
let input_sources = gtk::gio::Settings::new("org.gnome.desktop.input-sources");
input_sources.set("sources", variant).expect("failed to write layouts");
```

Listing 66: Set GNOME keyboard layouts with GIO

Unfortunately, GIO cannot access the system schemas within the sandboxing of Flatpak, which means that using it results in a crash. GIO is therefore not suitable and a workaround was necessary. Flatpak

provides a command that allows running commands outside the sandbox called Flatpak-spawn. Subsequently, DConf was used again to get the layouts like on the first try. To find out if ReSet is running inside Flatpak's sandboxing, the environment variable "container" can be checked. If it exists, then ReSet is running inside the sandbox and the command can be spawned using flatpak-spawn. There is the possibility that a user can have the environment variable "container" defined, but this is currently the best way to check. If an alternative is found, then this will be replaced. Compared to the first try, Regex was not necessary anymore because with the knowledge from the second try, using the built-in variant was a much cleaner way and less error-prone. In Listing 67 the updated version be seen.

```
let result = if getenv("container").is_none() {
    Command::new("dconf")
        .args(["read", "/org/gnome/desktop/input-sources/sources"])
        .output()
} else {
    Command::new("flatpak-spawn")
        .args([
            "--host",
            "dconf",
            "read",
            "/org/gnome/desktop/input-sources/sources",
        ])
        .output()
};
if result.is_err() {
    return kb;
}
let output = result.unwrap();
let layout_variant = String::from_utf8(output.stdout).unwrap();

let layout_variant = Variant::parse(Some(VariantTy::new("a(ss)").unwrap()), &layout_variant);
if layout_variant.is_err() {
    return kb;
}
let layout_variant = layout_variant.unwrap();
let layouts = layout_variant.get::<Vec<(String, String)>>().unwrap();
```

Listing 67: Flatpak compatible fetching of keyboard layouts

6.3.1.3 KDE Implementation

To read the keyboard layouts and variants in KDE, kreadconfig6 has to be used. Unfortunately, no library provides bindings so reading and writing using Rust commands was necessary. In Listing 68 the command with all its arguments can be seen. Writing to the kxkbrc file works the same as writing with the only difference of adding the new keyboard layout string as an additional argument and using kwriteconfig6.

```rust
let output = Command::new("kreadconfig6")
  .arg("--file")
  .arg("kxkbrc")
  .arg("--group")
  .arg("Layout")
  .arg("--key")
  .arg("LayoutList")
  .output()
  .expect("Failed to get saved layouts");
let kb_layout = parse_setting(output);

let output = Command::new("kreadconfig6")
  .arg("--file")
  .arg("kxkbrc")
  .arg("--group")
  .arg("Layout")
  .arg("--key")
  .arg("VariantList")
  .output()
  .expect("Failed to get saved layouts");
let kb_variant = parse_setting(output);
```

Listing 68: Read KDE keyboard layouts

An issue with these is that the versions are part of the command. Currently, the newest version is kreadconfig6 and kwriteconfig6 and have deprecated version 5 for the most part. This also means that if version 7 is being released, the command needs to be adjusted so that it works for both versions while version 6 is not deprecated. But because version 6 just released, version 7 is still in the far future.

6.3.2 Nested listing

A quality-of-life feature to makes adding keyboard layouts easier is the addition of nested listing. There are many keyboard variants for the same layout, for example, German (US), German (Dvorak) etc. which can grouped into a single entry in the list marked with an arrow symbol because not all languages have variants. An example of how this looks can be seen in Figure 52.

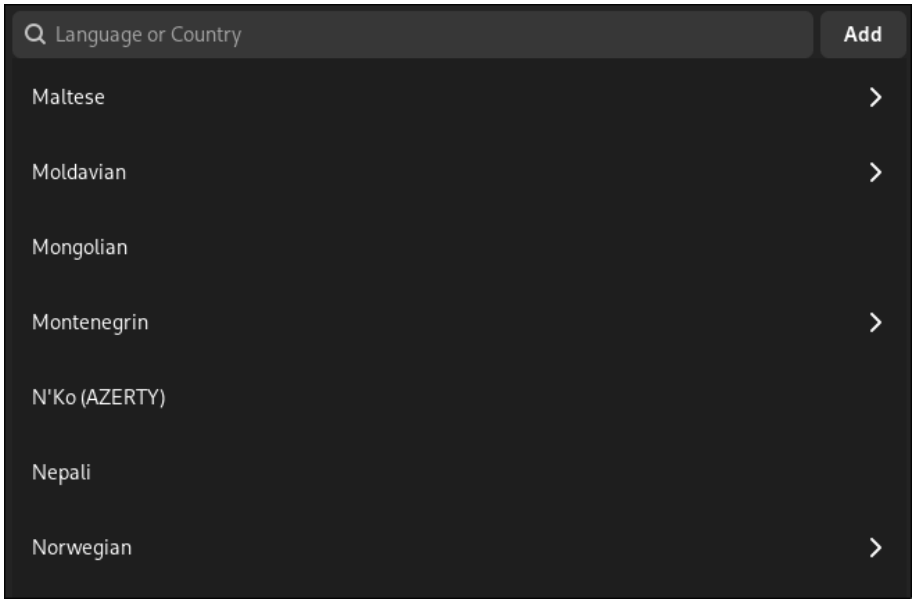


Figure 52: Keyboard layouts with variants

In GNOME control setting, every keyboard layout with its variations is shown in an alphabetically sorted list. The problem with this approach is that there are variants for layouts that have a very different name where it is not obvious to which layout it belongs to as seen in an example in Figure 53.

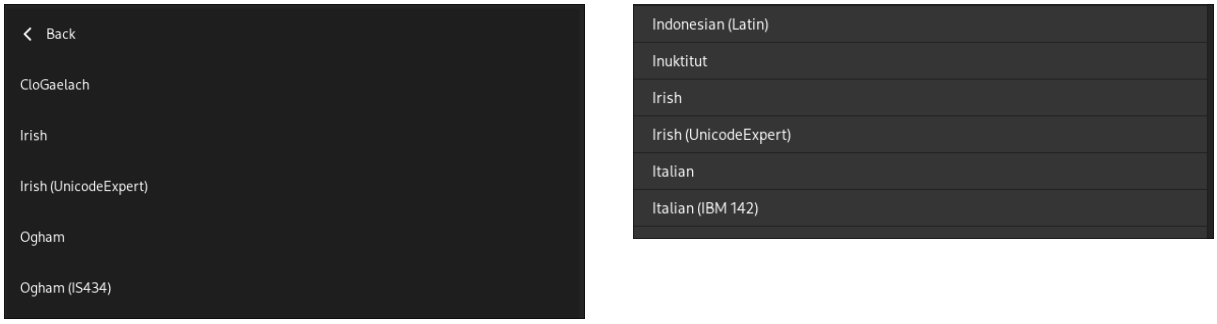


Figure 53: Irish keyboard layouts comparison (ReSet left, GNOME right)

This feature reduces the list by a significant amount and helps the user to narrow down the desired language first before looking more detailed at the specific variant. Clicking on such an entry removes all other keyboard layouts and only shows the variants for that language as seen in Figure 54.

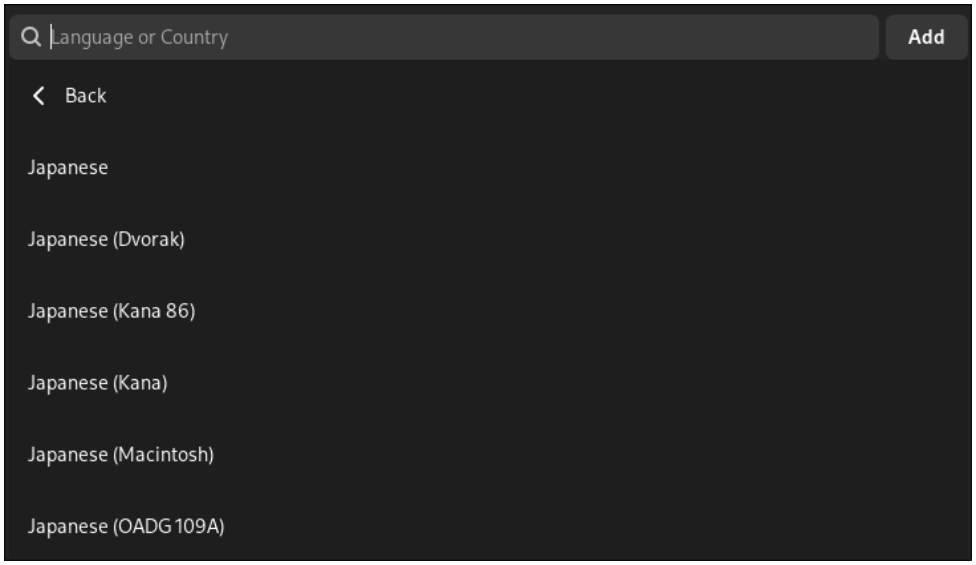


Figure 54: Keyboard layouts with variants

To achieve this, a layout that has variants needs to have some code that removes all other layouts and only shows its variants and a button to show all layouts again. This is shown in the on-click event listener in Listing 69. The back button removes every element in the list and inserts all layouts back into the list.

```
layout_row.connect_activate(clone!(@strong keyboard_layouts, @weak list, @strong back_row => move |_| {
    // add variants to list
    for keyboard_layout in keyboard_layouts.clone() {
        let layout_row = create_layout_row(keyboard_layout.description.clone());
        list.append(&layout_row);
    }

    list.prepend(&back_row);

    let mut last_row = list.last_child();
    let mut skip = keyboard_layouts.len();

    // remove all but first
    while last_row != None {
        // if we're at top of list, prevent selecting the back button and break
        if list.first_child() == last_row {
            list.grab_focus();
            break;
        }
        // skip rows because it's a variant we want to show
        if skip > 0 {
            last_row = last_row.unwrap().prev_sibling();
            skip -= 1;
            continue;
        }
        // remove row from list
        let temp = last_row.clone().unwrap().prev_sibling();
        list.remove(&last_row.unwrap());
        last_row = temp;
    }
}));
```

Listing 69: Code to only show variants of selected layout

An example on how the removal works visually can be seen in Figure 55.

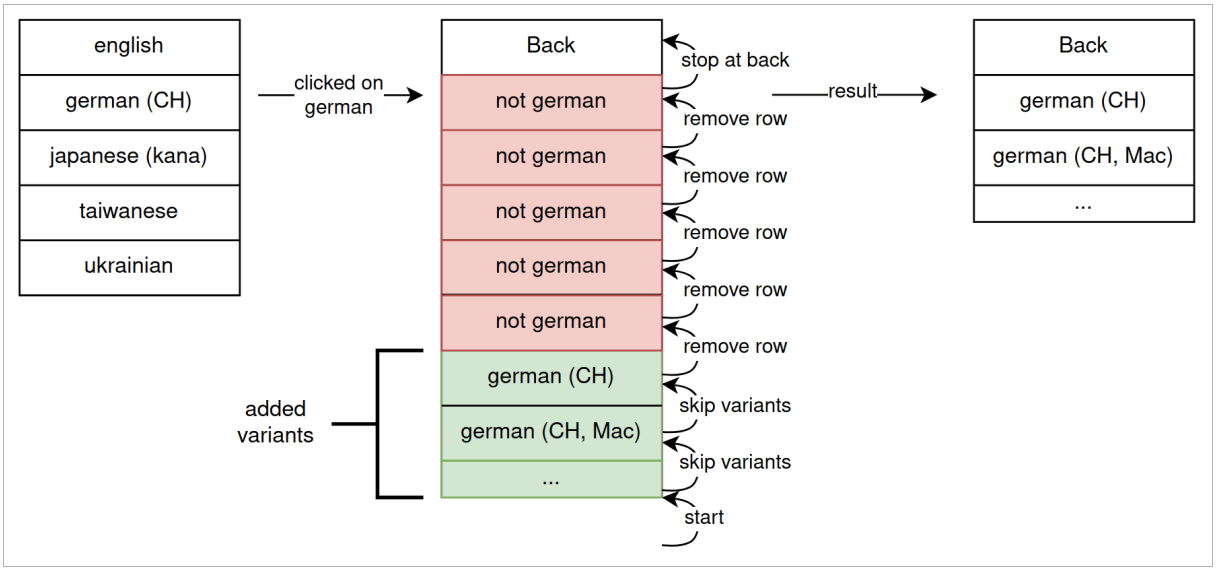


Figure 55: Removal of all layouts except german

6.3.2.1 Highlight active layouts

To show users the limitation of Section 5.3.2 visually, the first few rows are highlighted, while the rest have system colors. This number is set depending on the desktop environment because some do not use XKB and therefore could allow more than four keyboard layouts. To further clarify it a subtitle for this setting group that explains the highlight is added.

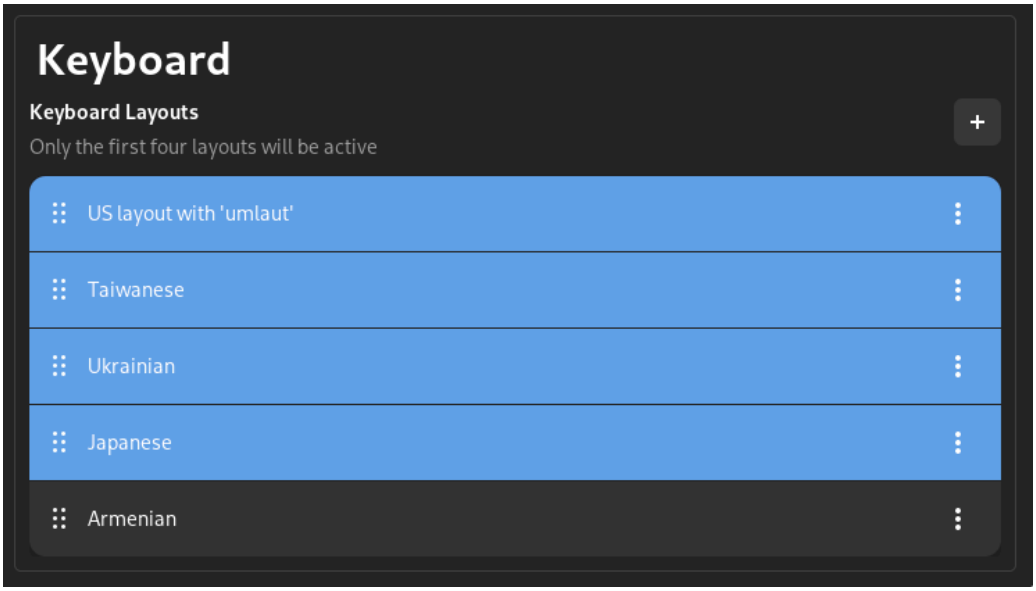


Figure 56: First few keyboard layouts are colored differently

Because there are users that use many different themes, it is not possible to hardcode a single color because it might not fit with other themes. A GTK theme is defined in a CSS file that contains a list of key-value like pairs of names and a color as can be seen in Listing 70. These can be accessed from ReSet with an @css-name like Listing 71. Because using the color definition is bound to clash with other UI elements with the same coloring, color expressions can be used to slightly change color without having to hardcode it. [67]

```
@define-color accent_color #a9b1d6;
@define-color accent_bg_color #a9b1d6;
@define-color accent_fg_color rgba(0, 0, 0, 0.87);
@define-color destructive_color #F28B82;
/* many more colors */
```

Listing 70: Excerpt from GTK default dark theme

```
row.activeLanguage {
    background-color: darker(darker(darker(@window_fg_color)));
}
```

Listing 71: Setting the highlight color

7 Usage and Distribution

This section covers how users can install and configure the ReSet plugin system along with their preferred plugins.

7.1 Distribution

7.1.1 Flatpak

Flatpak is a generic packaging solution that solves the issue of fragmentation across the Linux ecosystem by creating sandboxed environments for each application. [68]

Flatpak is a developer-first approach, meaning the distribution is done directly via developers who can request their application to be hosted on Flathub or on their own server. This is in contrast to system packaging solutions, which were created with Linux distributions in mind, in other words, while packages installed by apt and similar can be installed directly from the developer, it will likely cause dependency issues, as apt packages define a single source for shared libraries. Therefore, distributions host packages themselves, meaning a Debian user would install packages from the Debian repository and not from the developer directly.

In order to create a Flatpak package, ReSet needs to provide a Flatpak manifest, which defines the environment for Flatpak to build ReSet with. Within this sandbox, ReSet will then be built and packaged with all necessary dependencies in one, meaning the finished solution can then be used on any Linux system with Flatpak installed.

In Listing 72 the Flatpak manifest of ReSet is visualized. Note that several options were omitted in order to keep the listing concise.

```
{
  "app-id": "org.Xetibo.ReSet",
  "runtime": "org.gnome.Platform",
  "runtime-version": "45",
  "sdk": "org.gnome.Sdk",
  "sdk-extensions": [
    "org.freedesktop.Sdk.Extension.rust-nightly"
  ],
  "command": "reset",
  "build-options": {
    "append-path": "/usr/lib/sdk/rust-nightly/bin"
  },
  "modules": [
    {
      "name": "reset",
      "buildsystem": "simple",
      "build-commands": [
        "cargo --offline fetch --manifest-path Cargo.toml --verbose",
        "cargo --offline build --verbose",
        "install -Dm755 ./target/release/reset -t /app/bin/",
        "install -Dm644 ./src/resources/icons/ReSet.svg /app/share/icons/hicolor/scalable/apps/org.Xetibo.ReSet.svg",
        "install -Dm644 ./flatpak/org.Xetibo.ReSet.desktop /app/share/applications/org.Xetibo.ReSet.desktop"
      ],
    }
  ]
}
```

Listing 72: ReSet Flatpak manifest

Plugins for the Flatpak package cannot be distributed within Flatpak itself. This is a limitation of Flatpak, as one package cannot access another without consequences. Hence, plugins would need to be compiled separately and placed within the folder structure manually. Specific installation steps are shown in Section 7.2.

7.1.2 NixOS

NixOS is the Linux distribution of the Nix project, which promises three substantial benefits for users: Reproducibility, Declarativeness and Reliability. For ReSet, the interesting claim is the declarative aspect of Nix. With Nix, it is possible to create a system configuration, with which a user can define both ReSet to be installed and define the plugins for ReSet within the same configuration. This means that no additional system is required in order to have automatic handling of plugins. This is in contrast to other packaging solutions, which would require custom scripts in order to install and configure plugins via the package manager. [69]

In order to provide this functionality for Nix, ReSet would either need to provide a flake file, which can be used by Nix users to install ReSet and their respective plugins directly from the source, or ReSet could create a pull request on the official Nix repositories.
In order to not depend on third-party arbitration, this thesis will focus on creating a flake, which will include the possibility of installing and configuring plugins automatically.

Configuration in Nix is done with the functional and identically named Nix language. This language is Turing-complete and therefore offers an infinite amount of options for developers. ReSet uses this to create modules for users to configure ReSet directly via Nix instead of changing configuration files.

In Listing 73 the Nix options for the ReSet module are visualized.

```
# omitted setup
options.programs.ReSet = with lib; {
  # define if ReSet is installed
  enable = mkEnableOption "reset";

  # define plugins to be installed
  # this is defined with specifying Nix packages
  config = {
    plugins = mkOption {
      type = with types; nullOr (listOf package);
      default = null;
      description = mdDoc ''
        List of plugins to use, represented as a list of packages.
      '';
    };

    # define additional toml values to configure plugins
    plugin_config = mkOption {
      type = with types; attrs;
      default = { };
      description = mdDoc ''
        Toml values passed to the configuration for plugins to use.
      '';
    };
  };

};
# omitted applying of configuration
```

Listing 73: ReSet Nix options

Within the system configuration, a user can now enable ReSet and their plugins via the code in Listing 74.

```
# within flake inputs
# defines the origin for ReSet and plugins
reset.url = "github:Xetibo/ReSet";
reset-plugins.url = "github:Xetibo/ReSet-Plugins?ref=dashie";


# In nix/home-manager module
# enables ReSet and allows for plugins to be specified
programs.ReSet.enable = true;
programs.ReSet.config.plugins = [
  inputs.reset-plugins.packages."x86_64-linux".monitor
];
```

Listing 74: ReSet Nix usage

7.1.3 Arch and Ubuntu

For Arch Linux and Ubuntu, native packages are provided within the repository of the exemplary plugins. Users can download the plugins and install them via their system package manager which places them in `/usr/lib/reset/pluginname.so`. This means users of Arch Linux or Ubuntu would only be required to confirm their plugin selection as defined in Section 7.2.

7.1.4 Other Distributions

For other Linux distributions, ReSet would need to provide a generic plugin manager in order to achieve a consistent and cross-distribution installation experience. A plugin manager is not within the scope of this thesis, which requires users of ReSet to instead copy binaries manually into the plugin folder.

7.1.5 Cargo

Cargo is the package manager for the Rust language, which allows both the management of libraries within a project and the installation of binaries for a user. Installed binaries are located within the user home directory in `~/.cargo/bin/`. This means that a user must then manually handle the installation of plugins by copying them into the right directory or directly passing a different installation path for cargo during the plugin installation.

7.2 Usage

If manual installation is required, the plugins must be placed within `~/.config/reset/plugins/` after compiling them individually.

After installing the plugins, users are required to confirm their plugin selection within the ReSet configuration file by adding the exact filename of the plugin. By default, the configuration file is located at `~/.config/reset/ReSet.toml`. Confirmation of plugins is handled by the “plugins” key within the TOML file. In Listing 75, an example ReSet.toml file is visualized.

```
plugins = ["libreset_monitors.so", "libyour_other_plugin.so"]
```

Listing 75: ReSet TOML

8 Conclusion

In this section, the final result of this thesis is discussed.

This thesis aimed to implement a reliable plugin system for the existing ReSet application, in order to provide arbitrary extensibility for users. As a result, this thesis implemented both the plugin itself and two exemplary plugins in order to prove and test the implementation.

The thesis mentioned goal of a plugin system for the ReSet application has been achieved. Alongside this were two exemplary plugins, as well as underlying work for the base ReSet application such as refactoring and testing systems. The underlying work proved to be necessary for the extension of the plugin system as well as the continuing maintainability of the ReSet application.

The exemplary plugins prove that the plugin system works as expected and can hold arbitrary extensions for differing functionality. Alongside the plugin itself, the plugins prove the viability of the used testing framework for plugins to be injected into the base application. This results in the expected behavior of consolidated testing mentioned in Section 4.3.

8.1 Limitations

The implementation of the plugin system lacks ABI stability, as explained in Section 2.3, this means that plugin developers are limited to the Rust programming language. On top of this, any change in the compiler version might break the compatibility of a plugin. While this can also occur with major changes to the API conversion, it is at least decoupled from the compiler itself.
For this limitation, further work could implement ABI stability with the ABI stable crate in order to provide a C-compatible layer, which can then be used by any programming language. [28]

8.1.1 ReSet Leftovers

In the original work, multiple potential improvements were mentioned. Some of these points were addressed in this thesis, such as the testing environment for DBus, or refactoring several parts of the codebase. However, due to the goal of this thesis, no work was done to implement further features for the base application, such as completing the missing parts of both Bluetooth and network connectivity. Future work could focus on implementing a fully-fledged Bluetooth and networking solution for ReSet.

Further improvements that were not featured in this thesis are accessibility concerns for the base application, as well as keyboard movement and animation improvements.

8.1.2 Testing

Similar to the original work, the inclusion of DBus endpoints and GTK complicated testing heavily. Several special approaches had to be taken in order to allow testing at all. Overall with more time, a proper GTK testing framework and mock endpoints for environments such as Wayland protocols and more could be implemented after careful consideration of the cost-to-gain ratio.

8.1.3 Monitors

Due to the current development status of High Dynamic Range within the Linux desktop sphere, the HDR option for monitors is not implemented. This option could be added once the development of HDR on Linux stabilizes.

Time constraints and further necessary research pushed color profiles into an optional feature for the end product, which was omitted in favor of more testing and further environment support. This feature is preferably implemented alongside HDR as both options revolve around the same type of functionality.

The current implementation is targeted towards multiple Linux environments with their specific implementation. This forces the plugin to provide multiple endpoints to connect to the differing environments, as well as offer multiple conversions from the environment data types to the plugin data types. A new Wayland protocol, offering universal management of monitors, could simplify the implementation to a single endpoint for all Wayland implementations. The challenges for this endeavor lie with the universal acceptance of such a protocol, which would be required in order to gain universal status.

Support for environments is currently limited to Wayland implementations. This is mainly because Wayland is the current standard, with X11 being phased out as time passes. Due to the drastic difference in paradigm, a different plugin implementing the X11 solution would be optimal.

8.1.4 Keyboard

As mentioned in Section 5.3.2, the current protocol used within various Linux environments limits the active keyboard layouts to four. Hence, future work could implement a new protocol in order to increase this number and hence, improve functionality for users.

Even though features such as keyboard shortcuts, keyboard repeats and user-defined custom layouts were initially planned, they had to be removed because of time constraints.

8.2 Potential

This thesis offers users of ReSet the ability to inject plugins into the base application, which offers them theoretically unlimited extensibility. Hence, the biggest potential is likely to be seen in potential plugins for various use cases. Examples include online account management, device configuration (controllers, VR-Headsets, etc.), MIME-type configuration, and more.

9 Glossary

Compositor	A combination of display server(not a true server with Wayland) and window composition system. Often used to describe standalone Wayland environments like Mutter, KWin, Hyprland, Sway and River.
Daemon	Background process, most commonly used to handle functionality for a frontend.
DBus	Low-level API providing inter-process communication (IPC) on UNIX operating systems.
MIME	Multi-purpose Internet Mail Extensions, a description of a file that is used within Linux in order to specify a target program to open the file with. Example: text/html
Desktop Environment	A collection of software, enabling a graphical user interface experience to do general computing tasks. This includes most basic functionality like starting programs, shutting down the PC or similar.
GTK	A free software widget toolkit for creating graphical user interfaces. The name is an acronym for "GIMP ToolKit" as a reminder of the toolkit legacy, however, today GTK is trademarked and developed by the GNOME foundation and no longer refers to GIMP.
GNOME	A Linux desktop environment.
KDesktop Environment (KDE)	A Linux desktop environment. The K has no particular meaning.
Wayland	The current display protocol used on Linux. It replaces the previous X11 protocol, which is no longer in development. (it is still maintained for security reasons)
Window Manager	Provides window management without compositing them. With the X11 protocol, implementing an entire display server is not needed, therefore one can choose to simply provide, window spawning and window management. Examples include Mutter, KWin, dwm, Xmonad, i3 and herbstluftwm.
X11	A network transparent windowing system used by a variety of systems. It is usually used with the reference implementation Xorg.
IPC	IPC or Inter Process Communication is the idea of communication between differing processes using a specific protocol. Examples of IPC are DBus, sockets, message passing, pipes and more.

10 Bibliography

[1] feathericons, *settings.svg*. [Online]. Available: https://github.com/feathericons/feather/

[2] OST, *OST.svg*. [Online]. Available: https://www.ost.ch/de/die-ost/organisation/medien

[3] Fabio Lenherr and Felix Tran, "ReSet." [Online]. Available: https://eprints.ost.ch/id/eprint/1193/

[4] javascript-obfuscator, "javascript-obfuscator." [Online]. Available: https://github.com/javascript-obfuscator/javascript-obfuscator

[5] "Semantic Versioning 2.0." [Online]. Available: https://semver.org/

[6] "GLib." [Online]. Available: https://docs.gtk.org/glib/

[7] "Basic Concepts." [Online]. Available: https://docs.flatpak.org/en/latest/basic-concepts.html

[8] "Memory Management Unit." [Online]. Available: https://en.wikipedia.org/wiki/Memory_management_unit

[9] HyprWM, "Hyprland Plugin System." [Online]. Available: https://github.com/hyprwm/Hyprland/tree/main

[10] Steve Klabnik and Carol Nichols, *The Rust Programming Language*. [Online]. Available: https://doc.rust-lang.org/stable/book/

[11] "consteval specifier." [Online]. Available: https://en.cppreference.com/w/cpp/language/consteval

[12] "constexpr specifier." [Online]. Available: https://en.cppreference.com/w/cpp/language/constexpr

[13] "On computable numbers, with an application to the Entscheidungsproblem." [Online]. Available: https://www.cs.virginia.edu/~robins/Turing_Paper_1936.pdf

[14] "HTML: HyprText Markup Language." [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTML

[15] suckless, "Suckless Philosophy." [Online]. Available: https://suckless.org/philosophy/

[16] Wikipedia, "Unix philosophy." [Online]. Available: https://en.wikipedia.org/wiki/Unix_philosophy#cite_note-volkerding-2012-10

[17] Peter H. Salus, *A Quarter-Century of Unix*.

[18] "crates.io." [Online]. Available: https://crates.io/

[19] steveklabnik, "Define a Rust ABI." [Online]. Available: https://github.com/rust-lang/rfcs/issues/600

[20] "Publish in the Chrome Web Store." [Online]. Available: https://developer.chrome.com/docs/webstore/publish#upload-your-item

[21] "GNU General Public License." [Online]. Available: https://www.gnu.org/licenses/gpl-3.0.en.html

[22] "GNOME Keyring." [Online]. Available: https://wiki.gnome.org/Projects/GnomeKeyring

[23] "Keyring." [Online]. Available: https://docs.rs/keyring/latest/keyring/

[24] "Application Signing." [Online]. Available: https://source.android.com/docs/security/features/apksigning

[25] "GTK test." [Online]. Available: https://github.com/gtk-rs/gtk-test

[26] Bram Moolenaar, "Vim - the ubiquitous text editor." [Online]. Available: https://www.vim.org/

[27] HyprWM, "Hyprland Wiki Hook Example." [Online]. Available: https://wiki.hyprland.org/Plugins/Development/Advanced/

[28] rodrimati1992, "abi_stable_crates." [Online]. Available: https://github.com/rodrimati1992/abi_stable_crates/

[29] Kirottu, "Anyrun." [Online]. Available: https://github.com/Kirottu/anyrun

[30] "Neovim." [Online]. Available: https://neovim.io/

[31] Neovim, "Neovim lua documentation." [Online]. Available: https://neovim.io/doc/user/lua.html

[32] folke, "lazy package manager." [Online]. Available: https://github.com/folke/lazy.nvim

[33] DashieTM, "testplugin neovim." [Online]. Available: https://github.com/DashieTM/test_plugin

[34] "Helix." [Online]. Available: https://helix-editor.com/

[35] mattwparas, "Steel plugin system for helix." [Online]. Available: https://github.com/helix-editor/helix/pull/8675

[36] mattwparas, "Steel." [Online]. Available: https://github.com/mattwparas/steel

[37] Roblox, "Introduction to Scripting." [Online]. Available: https://create.roblox.com/docs/tutorials/scripting/basic-scripting/intro-to-scripting

[38] UE4SS, "UE4SS lua API." [Online]. Available: https://docs.ue4ss.com/lua-api.html

[39] GNOME, "GNOME Shell Extensions." [Online]. Available: https://gitlab.gnome.org/GNOME/GNOME-shell/-/blob/main/js/extensions/extension.js?ref_type=heads

[40] diwic, "dbus-rs." [Online]. Available: https://github.com/diwic/dbus-rs

[41] "COSMIC: More Alpha, More Fun!." [Online]. Available: https://blog.system76.com/post/cosmic-more-alpha-more-fun

[42] "EndeavourOS." [Online]. Available: https://endeavouros.com/

[43] "wlr output management." [Online]. Available: https://wayland.app/protocols/wlr-output-management-unstable-v1

[44] "KDE output management v2." [Online]. Available: https://wayland.app/protocols/kde-output-management-v2

[45] GNOME, "DisplayConfig.xml." [Online]. Available: https://gitlab.gnome.org/GNOME/mutter/-/blob/main/data/dbus-interfaces/org.gnome.Mutter.DisplayConfig.xml?ref_type=heads

[46] "Wayland." [Online]. Available: https://gitlab.freedesktop.org/wayland/wayland

[47] "wayland-rs." [Online]. Available: https://github.com/Smithay/wayland-rs

[48] "Kde Output Device V2." [Online]. Available: https://wayland.app/protocols/kde-output-device-v2

[49] "wl_output compositor output region." [Online]. Available: https://wayland.freedesktop.org/docs/html/apa.html#protocol-spec-wl_output

[50] "Kscreen doctor gitlab of KDE." [Online]. Available: https://invent.kde.org/plasma/libkscreen/-/tree/master/src/doctor?ref_type=heads

[51] "Color Universal Design(CUD)." [Online]. Available: https://jfly.uni-koeln.de/color/

[52] "Color Blindness." [Online]. Available: https://en.wikipedia.org/wiki/Color_blindness

[53] Joachim Goedhart, "Data Visualization with Flying Colors." [Online]. Available: https://thenode.biologists.com/data-visualization-with-flying-colors/research/

[54] "hyprctl." [Online]. Available: https://wiki.hyprland.org/Configuring/Using-hyprctl/

[55] "gsettings." [Online]. Available: https://wiki.gnome.org/Projects/dconf

[56] "GIO." [Online]. Available: https://crates.io/crates/gio

[57] "dconf." [Online]. Available: https://wiki.gnome.org/Projects/dconf

[58] "dconf-rs." [Online]. Available: https://crates.io/crates/dconf_rs

[59] "gsettings." [Online]. Available: https://userbase.kde.org/KDE_System_Administration/Configuration_Files

[60] "wayland-keyboard." [Online]. Available: https://wayland-book.com/seat/xkb.html

[61] "xkb." [Online]. Available: https://www.x.org/releases/X11R7.7/doc/kbproto/xkbproto.html

[62] "MockFlow." [Online]. Available: https://www.mockflow.com/

[63] "Drag and Drop." [Online]. Available: https://en.wikipedia.org/wiki/Drag_and_drop

[64] "Fractional Scale." [Online]. Available: https://wayland.app/protocols/fractional-scale-v1

[65] "Adwaita SpinRow." [Online]. Available: https://gnome.pages.gitlab.gnome.org/libadwaita/doc/1-latest/class.SpinRow.html

[66] "Cairo Graphics." [Online]. Available: https://www.cairographics.org/

[67] "GTK CSS properties." [Online]. Available: https://docs.gtk.org/gtk4/css-properties.html

[68] "Flatpak." [Online]. Available: https://flatpak.org/

[69] "Nix." [Online]. Available: https://nixos.org/

[70] Xetibo, "ReSet-Lib." [Online]. Available: https://github.com/Xetibo/ReSet-Lib

[71] tokio-rs, "tokio." [Online]. Available: https://github.com/tokio-rs/tokio

[72] jnqnfe, "pulse-binding-Rust." [Online]. Available: https://github.com/jnqnfe/pulse-binding-rust

[73] crossbeam-rs, "crossbeam." [Online]. Available: https://github.com/crossbeam-rs/crossbeam

[74] matklad, "OnceCell." [Online]. Available: https://github.com/matklad/once_cell

[75] nagisa, "libloading." [Online]. Available: https://github.com/nagisa/rust_libloading/

[76] palfrey, "serial_test." [Online]. Available: https://github.com/palfrey/serial_test/

[77] toml-rs, "toml." [Online]. Available: https://github.com/toml-rs/toml/tree/main/crates/toml

[78] Xetibo, "ReSet-Daemon." [Online]. Available: https://github.com/Xetibo/ReSet-Daemon

[79] GNOME, "libadwaita-rs." [Online]. Available: https://gitlab.gnome.org/World/Rust/libadwaita-rs

[80] gtk-rs, "gtk4-rs." [Online]. Available: https://github.com/gtk-rs/gtk4-rs

[81] gtk-rs, "gtk-rs-core." [Online]. Available: https://github.com/gtk-rs/gtk-rs-core

[82] immortal, "fork." [Online]. Available: https://github.com/immortal/fork

[83] achanda, "ipnetwork." [Online]. Available: https://github.com/achanda/ipnetwork

[84] xdg-rs, "dirs." [Online]. Available: https://github.com/xdg-rs/dirs/tree/master/directories

[85] whitequark, [Online]. Available: https://github.com/whitequark/rust-xdg

[86] serde-rs, "serde." [Online]. Available: https://github.com/serde-rs/serde

[87] serde-rs, [Online]. Available: https://github.com/serde-rs/json

[88] jonasc, [Online]. Available: https://github.com/jonasc/xkbregistry.rs

11 List of Tables

12 List of Figures

13 List of Listings

14 Appendix

14.1 Management Summary

ReSet is a settings application developed with Rust for Linux-based systems to be compatible with different graphical desktop interfaces. Its core functionalities of Wi-Fi, Bluetooth and Audio are already implemented, but additional functionalities require desktop-specific implementation with their own set of rules that need to be adhered to. Hence, a plugin system was created for the ReSet application, alongside a testing framework and improvements to the original application.

Features:

- **Plugin system**:
  A plugin system for both processes of ReSet, which can be loaded dynamically at runtime and configured by the user.
- **Monitor plugin**:
  A monitor plugin that allows users to change the properties of their current monitor or rearrange their monitors.
  This plugin supports: KDE, GNOME, Hyprland, wlroots-based compositors and KWin-based compositors
- **Keyboard plugin**:
  A keyboard plugin that allows users to change their keyboard layout, and add or remove layouts.
  This plugin supports: KDE, GNOME, Hyprland
- **Plugin testing framework**:
  Plugins can provide tests to both the daemon and the graphical user interface, in order to test the entire system instead of just the plugin. (Integration tests)
- **Mock System**:
  The original ReSet application can be tested without hardware or software support enabled on the system. (excluding Audio)

Distribution:
Just like the original ReSet application, the plugins should be available on as many distributions as possible. For this reason, at least four different packages have been created: Arch package, Debian package (Ubuntu 24.04), NixOS module and the binaries themselves which were proven to work with Flatpak.

Risks:
During the initial stages, risks and its countermeasures were defined to respond fast to possible obstacles. The plugin system was completed in time and was not too ambitious. ReSet does not have any noticeable lag that would hinder the user experience. The plugin system was tested extensively with testers and developers and should be relatively stable. The chosen architecture fits very nicely into ReSet.

Non-Functional Requirements:
The project completed all but one required non-functional requirements. The user interface is responsive and fast, plugins feel like they are core features of the application and the plugins do not get loaded in case of a mismatch in API. The only problematic non-functional requirement is the handling of crashes in plugins. This was not achieved as plugins are not sandboxed, hence if a plugin calls unwrap on a None value, the application will crash.

Extensibility:
Further work can be done to create further plugins to increase the functionality of ReSet for specific use cases. Direct recommendations include theming for various toolkits, mouse configuration, controller configuration and more.

14.2 Planning methods

Project Management | ReSet is managed using Scrum.
Due to the small team size, no scrum master or product owner is chosen, the work of these positions is done in collaboration.

14.2.1 Project Plan

The broad project plan is created with a GANTT diagram which is visualized in Figure 57.

14.2.2 User Stories

User stories are handled via GitHub issues. This allows for simple handling of labels and function points, which can also be changed later on if needed.

14.2.3 Sprint Backlog/Product Backlog

Both the sprint backlog and the product backlog are handled using the GitHub project board. This allows for a clean and automatic handling of created user stories via GitHub issues.

14.2.4 Tools

Rust is an editor-agnostic language, hence both team members can use their tools without conflicts. For documentation typst is used, which also does not require a special editor, it just requires a compiler that works on all platforms. The user interface is made with Cambalache.

14.2.5 Time

As an agile methodology is used, long-term time management is only broad guidance and does not necessarily overlap with reality. In other words, with a two week sprint, only two weeks are considered well planned out. By week four, work on the backend implementation should be started. The minimal working UI milestone is in week seven, so UI Reviews can be assessed and corrections can be applied early. By week 12 the minimal viable product should be in production state, in order to leave time for reflection and documentation.

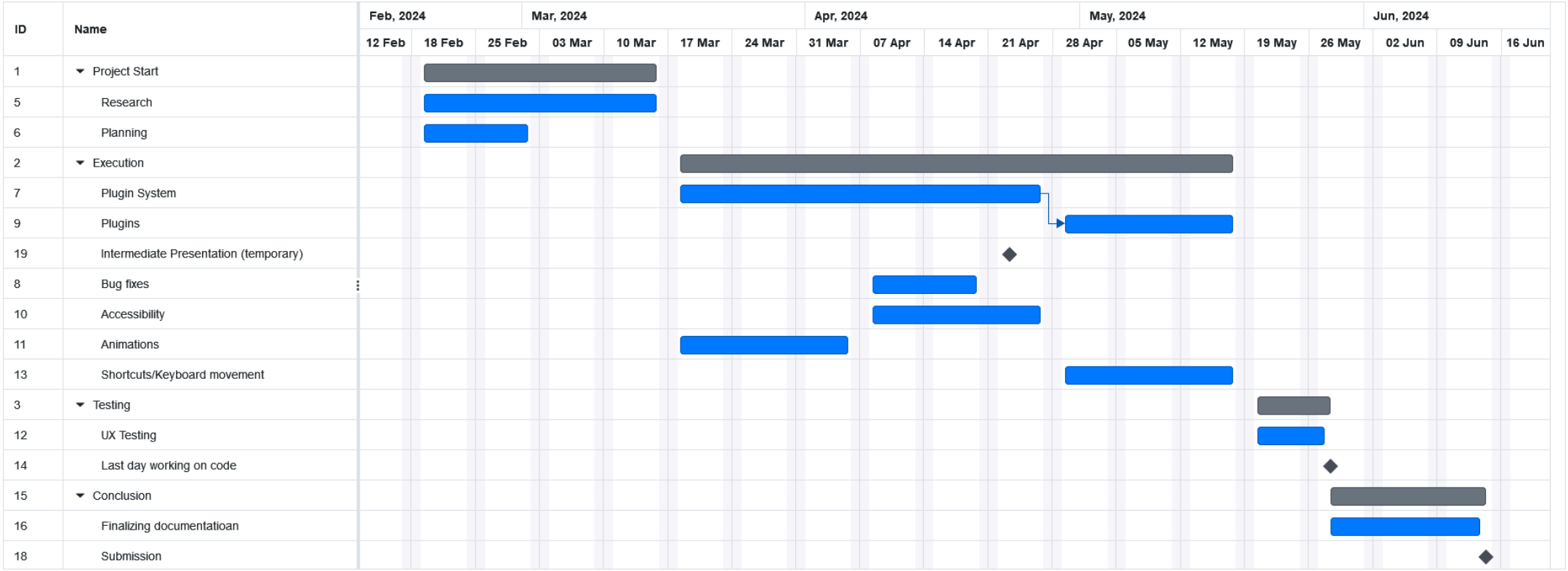


Figure 57: Time management

14.3 Requirements

14.3.1 Functional Requirements

As per Scrum, functional requirements are handled using the user story format which can be found in the ReSet and ReSet-Daemon repositories respectively.

14.3.2 Non-Functional Requirements

Subject	Seemless UI
Requirement	Performance
Priority	High
Description	Settings should load quickly, so the user does not feel like the application is lagging
Measures	• Lazy loading of settings • Use spinner or similar to indicate loading • Use async function in order to keep UI responsive
Subject	Seemless UI
Requirement	Usability
Priority	Medium
Description	Users should not easily differentiate between Plugin and Core functionality.
Measures	• Provide a seamless user experience for the user by integrating plugin settings within core settings (e.g add plugin into sidebar)
Subject	User Interface
Requirement	Maintainability
Priority	Medium
Description	In case of a plugin crash, the application should not crash. It should try to continue running or at least log the error.
Measures	• Try to disable the crashing plugin and continue the application • Provide a log file for the user to view in case of a crash
Subject	Test plugins
Requirement	Maintainability
Priority	Medium
Description	Plugins should be tested to ensure they are working correctly.
Measures	• Check version number before running it.

14.4 Risks

Risks are assessed according to the ISO standard with a risk matrix.

Certain	High	High	Very High	Very High
Likely	Medium	High	High	Very High
Possible	Low	Medium	High	Very High
Unlikely	Low	Medium	Medium	High
Rare	Low	Low	Medium	Medium
Eliminated	Eliminated	Eliminated	Eliminated	Eliminated
	Minor	Marginal	Critical	Catastrophic
Probability	Severity			

Subject	Plugin System
Risk	The plugin system might be too ambitious and could take too much time to realize or could be implemented faster than expected.
Priority	High
Probability&Severity	Likely & Marginal
Measures	In case it is too ambitious, instead of a plugin system, or socket connections can be used to realize a limited implementation of expected features. Otherwise more features like a system tray or custom bar can be implemented as well.
Subject	Plugin System performance issues
Risk	The plugin system might have performance issues that could negatively impact the user experience.
Priority	Medium
Probability&Severity	Unlikely & Marginal
Measures	The plugin system can be optimized. If this is not possible, consider a different architecture.
Subject	Plugin System stability issues
Risk	Because plugins can be written by third parties, they might not be as stable and cause problems like crashes.
Priority	High
Probability&Severity	Possible & Critical
Measures	Verify the plugin that it is compatible and add measures in case of a plugin crash.

Subject	Plugin System architecture not fitting
Risk	There might be some breaking problems that go unnoticed until development has already startet on the plugin system.
Priority	Medium
Probability&Severity	Rare & Catastrophic
Measures	Keep a second architecture in mind that could replace the first if it does not work out.
Subject	Exemplary Plugins
Risk	Planned plugins might be to ambitious and could take longer than expected.
Priority	Medium
Probability&Severity	Possible & Marginal
Measures	• Reduce the scope of individual plugins • Reduce the amount of plugins • Reduce the amount of supported environments

14.5 User Testing

In this section, the ReSet usability tests for the exemplary plugins are discussed.

14.5.1 Participants

In our usability test, a total of 12 participants were involved. In Figure 58 the participant's experience with PC and Linux is shown. Most participants are very experienced in using a PC and have a good understanding of Linux as most are studying computer science. This sample might skew the results towards more experienced users.

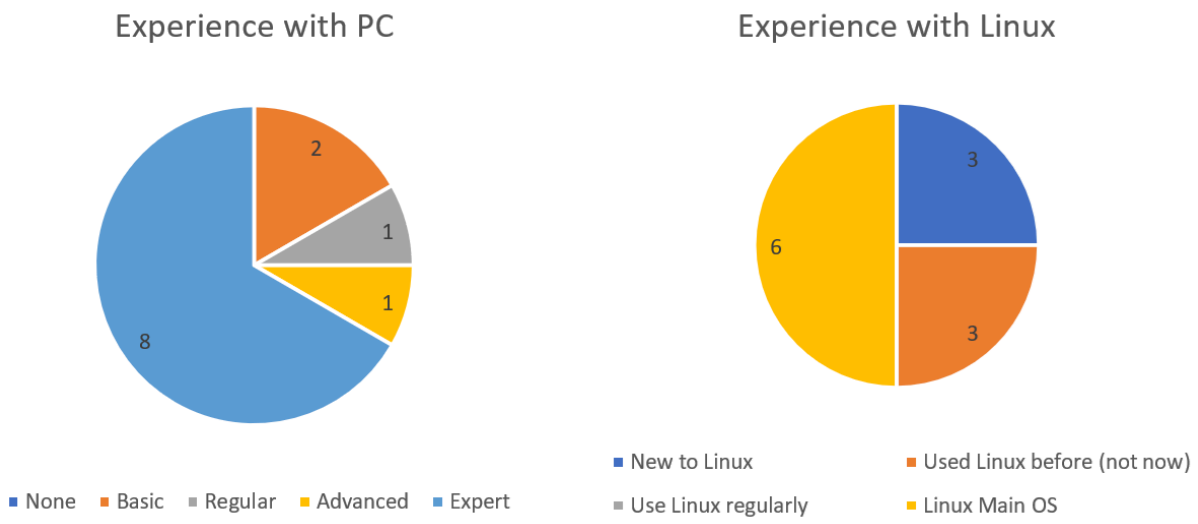


Figure 58: Participant's PC and Linux Experience

14.5.2 Monitor Plugin Tests

Subject	Monitor Plugin Base
Description	Did you find the monitor page? Did you encounter any issues opening the page?
Feedback	All participants found the monitor plugin page without any issues.
Suggestions	No improvements were suggested.
Subject	Monitor Plugin Single Settings
Description	Changing the resolution within the user interface and applying the configuration results in the chosen resolution being applied to the monitor.
Feedback	Most participants were able to change the resolution without any issues. One participant had issues with the resolution not being applied if the monitors were not adjacent. The confirmation dialogue with the revert function was praised. One user encountered an error on the GNOME environment, which was fixed before release.
Suggestions	• Name and resolution inside monitor clips through the border • The Apply button is not directly visible • Show countdown how many seconds before revert

Subject	Monitor Plugin Single Settings
Description	Changing the refresh rate within the user interface and applying the configuration results in the chosen refresh rate being applied to the monitor.
Feedback	Most participants were able to change their refresh rate. Issues were found on GNOME, which had the same source as in the last test, just like the issue with resolution, it is fixed.
Suggestions	No improvements were suggested.
Subject	Monitor Plugin Single Settings
Description	Changing the rotation (transform) rate within the user interface and applying the configuration results in the chosen rotation (transform) being applied to the monitor.
Feedback	Most participants were able to change their monitor rotation. The only changes proposed were cosmetic. The issue from resolution on GNOME was also applicable here.
Suggestions	• Add ° to numbers (90 -> 90°) or use words instead of degrees ‣ implemented
Subject	Monitor Plugin Single Settings
Description	Changing the scale rate within the user interface and applying the configuration results in the chosen scale being applied to the monitor.
Feedback	All participants were able to change their monitor scale. A few found the shown scale values and their steps a bit confusing.
Suggestions	• Show percentage values instead of numbers • Percentages should only show integers -> percentage showed 170.0000023421% • limit max scaling to sensible number -> implemented • limit offset of monitors in drawing area
Subject	Monitor Plugin Special Settings
Description	Toggling the variable refresh rate button within the user interface and applying the configuration results in the chosen setting being applied to the monitor.
Feedback	Most participants had no VRR support. The ones that had it could not verify the change.
Suggestions	• unclear if the feature works on the environment at all ‣ setting therefore did not apply sometimes

Subject	Monitor Plugin Multi-Monitor Settings
Description	Enabling/Disabling a monitor within the user interface and applying the configuration results in the chosen setting monitor being enabled/disabled.
Feedback	The participants that had a multi-monitor setup were able to enable/disable monitors without any issues. One user didn't find the option to enable/disable a monitor.
Suggestions	• Revert window time too short as some monitors take seconds to enable • button to enable/disable should have a text added
Subject	Monitor Plugin Multi-Monitor Settings
Description	Rearranging the monitors with the mouse and applying the configuration results in the chosen arrangement being applied to the monitors.
Feedback	The participants that had a multi-monitor setup were mostly able to rearrange their monitors. One participant didn't notice that the monitors could be dragged. Another mentioned that dragging and dropping needs to be too precise. The issue from resolution on GNOME was also applicable here.
Suggestions	• Snapping of monitors could be more lenient • Show that monitors can be dragged • Disable Apply button when monitors are overlapping instead of snapping
Subject	Monitor Plugin Multi-Monitor Special Settings
Description	Toggling a monitor as a primary monitor on environments such as KDE/GNOME and applying the configuration will set the primary monitor within that environment.
Feedback	Most participants were able to set their main monitor. One participant reported that it could only be applied once. GNOME did not always accept the configuration, causing a "configuration on stale information error", this error is fixed in the release.
Suggestions	No improvements were suggested.

14.5.3 Keyboard Plugin Tests

Subject	Keyboard Plugin Base
Description	Did you find the keyboard page? Did you encounter any issues on opening the page?
Feedback	All participants found the keyboard plugin page without any issues.
Suggestions	No improvements were suggested.
Subject	Keyboard Plugin Settings
Description	Opening the plugin page within ReSet will show the currently configured keyboard layouts.
Feedback	Most participants were able to see their current keyboard layouts. One participant had German shown as the active layout, but the US layout was active.
Suggestions	No improvements were suggested.
Subject	Keyboard Plugin Settings
Description	Adding a new keyboard layout with the user interface will show a new layout and will be saved to the environment.
Feedback	Some users were able to add a new layout without issues. Others had some crashes when adding a new layout or the layout was not saved, these crashes have been fixed for the release.
Suggestions	• Double-click to add a layout • Add multiple layouts at once
Subject	Keyboard Plugin Settings
Description	Rearranging the order of layouts will be saved to the environment. (A different layout will be active if the first layout is changed)
Feedback	Worked for most participants, with some having issues with the default layout not being set.
Suggestions	• Dropping zone is too harsh • click doesn't always register -> turned out to be a Hyprland issue
Subject	Keyboard Plugin Settings
Description	Removing a layout from the list will also remove the layout within the environment configuration.
Feedback	Most participants were able to remove a layout without any issues. One participant crashed the application when removing a layout, this issue is fixed in the release.
Suggestions	• Context menu pointless with only one option

14.5.4 Feedback

In general, the feedback from the participants was predominantly positive. Most participants did not encounter any major issues. The suggestions for improvements were mostly cosmetic or small usability improvements.

14.5.4.1 Ease of use

The feedback regarding the ease of use was very positive. Most participants found the plugins to be very intuitive to use because the user experience was familiar. Another point that was mentioned was it worked with different themes. One participant mentioned that the drag-and-drop feature for the keyboard plugin was not effortless. In Figure 59 the participant's review scores regarding ease of use are shown.

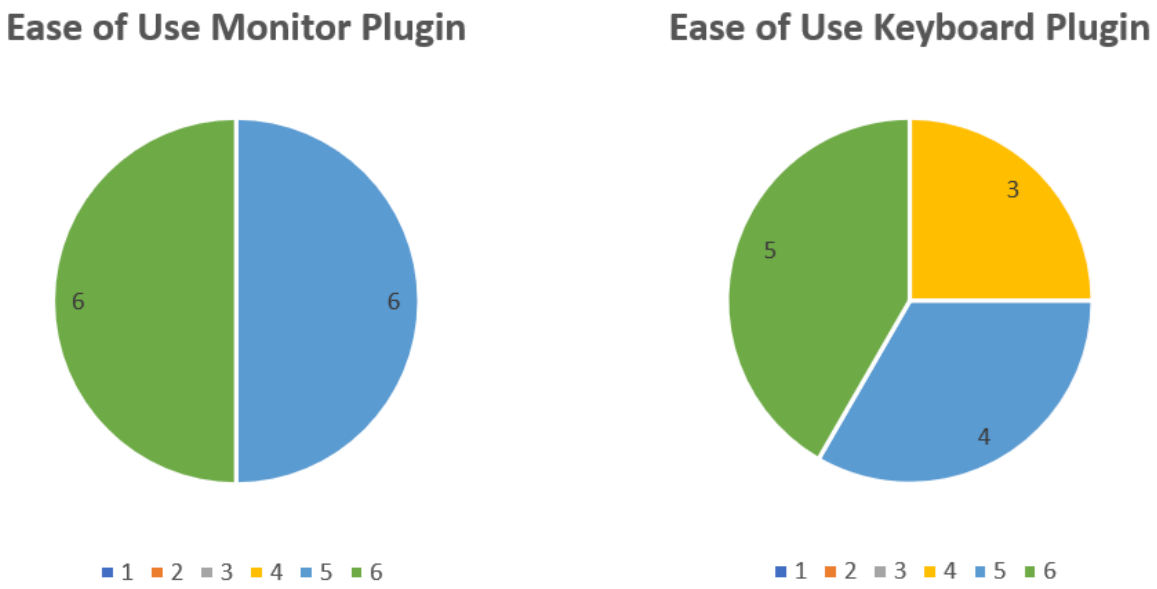


Figure 59: Participant's Ease of Use Score

14.5.4.2 Design

Participants praised the simple and clean design of the plugins which reminded them of the GNOME control settings. Some participants mentioned that the spacing could be improved to make the interface feel less cramped and allow for more customization. In Figure 60 the participant's review scores regarding the design are shown.

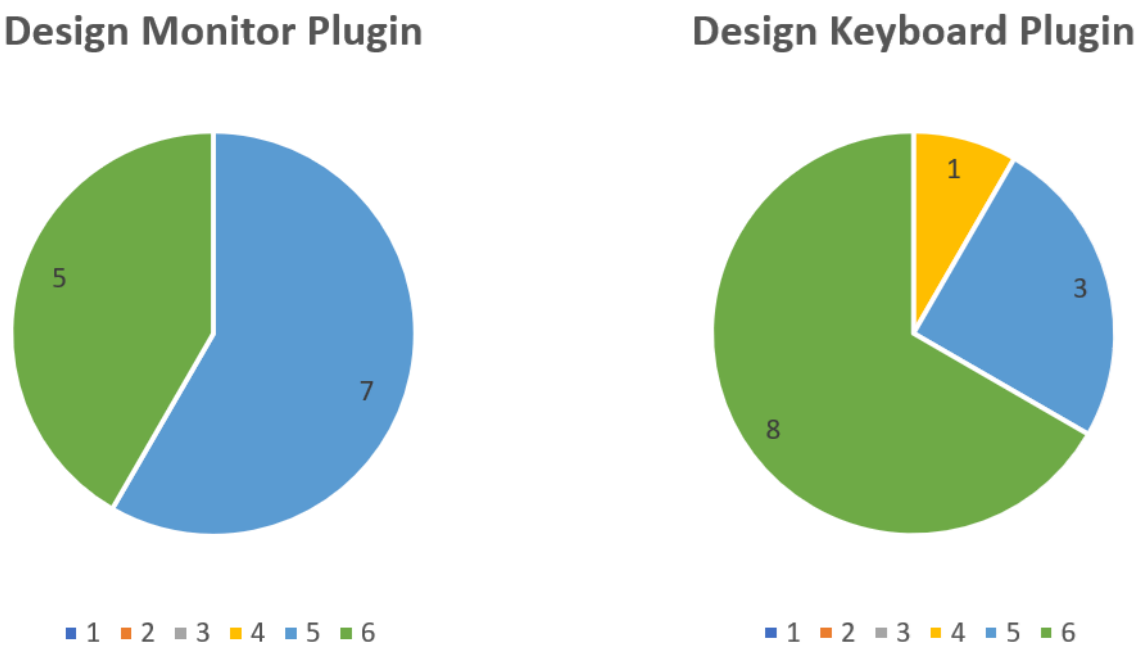


Figure 60: Participant's Design Score

14.5.4.3 Conclusion

In conclusion, most participants found the plugins to be very useful and easy to use. Some more general improvements that were suggested were for example translations and automatic installation scripts for plugins. Requested features were for example a notification manager and a system tray config. A few of the suggestions have already been implemented.

14.6 Plugin API

This section covers the API that a potential plugin must implement.

14.6.1 Plugin Backend API

```
// The startup function is intended to be used to allocate any required resources.
pub fn backend_startup();

// Cleanup any resources allocated for your plugin that aren't automatically removed.
pub fn backend_shutdown();

// Reports the capabilities that your plugin will provide, simply return a vector of
// strings.
pub fn capabilities() -> PluginCapabilities;

// Reports the name of the plugin, used for duplication detection and plugin tests.
pub fn name() -> String;

// Inserts your plugin interface into the dbus server. Provided as a parameter is the
// crossroads context, which you can use in order to insert your interfaces and objects
pub fn dbus_interface(cross: &mut Crossroads);

// Use this function to return any tests you would like to have run.
// This might be a bit confusing as this will force you to define your functions for
// testing outside of your typical rust tests.
pub fn backend_tests();
```

Listing 76: Backend Plugin API functions

14.6.2 Plugin Frontend API

```
// The startup function is intended to be used to allocate any required resources.
pub fn frontend_startup();

// Cleanup any resources allocated for your plugin that aren't automatically removed.
pub fn frontend_shutdown();

// Reports the capabilities that your plugin will provide, simply return a vector of
// strings.
pub fn capabilities() -> PluginCapabilities;

// Reports the name of the plugin, used for duplication detection and plugin tests.
pub fn frontend_name() -> String;

// Provides the information needed to embed the plugin UI into Reset.
// SidebarInfo contains the name, icon and hiearchy in the sidebar.
// Vec<gtk::Box> contains the widgets that will be shown in the main window.
pub fn frontend_data() -> (SidebarInfo, Vec<gtk::Box>);

// Use this function to return any tests you would like to have run.
// This might be a bit confusing as this will force you to define your functions for
// testing outside of your typical rust tests.
pub fn frontend_tests();
```

Listing 77: Frontend Plugin API functions

14.6.3 Keyboard Plugin DBus API

Type	Type Description
GetKeyboardLayouts	DBus signature: a(sss) `Vec<String, String, String>`
GetSavedLayouts	DBus signature: a(sss) `Vec<String, String, String>`
SaveLayoutOrder	DBus signature: a(sss) `Vec<String, String, String>`
GetMaxActiveKeyboards	DBus signature: u32 `u32`

14.6.4 Monitor Plugin DBus API

Type	Type Description
GetMonitors	DBus signature: a(ub(ssss)(udu)bb(ii)(ii)sa(s(ii)a(us)ad)b(bbbb)) Vec<Monitor> (see Listing 50)
SaveMonitors	DBus signature: a(ub(ssss)(udu)bb(ii)(ii)sa(s(ii)a(us)ad)b(bbbb)) Vec<Monitor> (see Listing 50)
SetMonitors	DBus signature: a(ub(ssss)(udu)bb(ii)(ii)sa(s(ii)a(us)ad)b(bbbb)) Vec<Monitor> (see Listing 50)

```rust
use std::ffi::{c_void, CString};

#[no_mangle]
pub extern "C" fn pluginAPIVersion() -> String {
    String::from("0.1")
}

#[no_mangle]
pub extern "C" fn pluginInit(_handle: *const c_void) -> PLUGIN_DESCRIPTION_INFO {
    println!("start plugin");
    PLUGIN_DESCRIPTION_INFO {
        name: CString::new("RustPlugin")
            .expect("Could not convert to CString.")
            .into_raw(),
        description: CString::new("This is insane!")
            .expect("Could not convert to CString.")
            .into_raw(),
        author: CString::new("Xetibo")
            .expect("Could not convert to CString.")
            .into_raw(),
        version: CString::new("0.1")
            .expect("Could not convert to CString.")
            .into_raw(),
    }
}

#[no_mangle]
pub extern "C" fn pluginExit() {
    println!("exit plugin");
}

#[allow(non_camel_case_types)]
#[repr(C)]
pub struct PLUGIN_DESCRIPTION_INFO {
    name: *mut libc::c_char,
    description: *mut libc::c_char,
    author: *mut libc::c_char,
    version: *mut libc::c_char,
}
```

Listing 78: Hyprland plugin in Rust

14.7 Libraries

This section covers all direct third-party libraries used within ReSet and its repositories.

14.7.1 ReSet-Daemon

- ReSet-Lib [70]: GPL-3.0-or-later
- DBus, DBus-crossroads, DBus-toktio [40]: Apache-2.0, MIT
- tokio [71]: MIT
- libpulse_binding [72]: Apache-2.0, MIT
- crossbeam [73]: Apache-2.0, MIT
- once_cell [74]: Apache-2.0, MIT
- libloading [75]: ISC
- serial_test [76]: MIT
- toml [77]: Apache-2.0, MIT

14.7.2 ReSet

- ReSet-Daemon [78]: GPL-V3-or-later
- ReSet-Lib [70]: GPL-V3-or-later
- libadwaita [79]: MIT
- gtk4 [80]: MIT
- glib [81]: MIT
- tokio [71]: MIT
- fork [82]: BSD-3-Clause
- ipnetwork [83]: Apache-2.0

14.7.3 ReSet-Lib

- directories-next [84]: Apache-2.0, MIT
- DBus, DBus-crossroads, DBus-toktio [40]: Apache-2.0, MIT
- libpulse_binding [72]: Apache-2.0, MIT
- once_cell [74]: Apache-2.0, MIT
- libloading [75]: ISC
- gtk4 [80]: MIT
- serial_test [76]: MIT
- toml [77]: Apache-2.0, MIT
- xdg [85]: Apache-2.0, MIT

14.7.3.1 Monitor Plugin

- ReSet-Lib Section 14.7.3: GPL-V3-or-later
- DBus, DBus-crossroads, DBus-toktio [40]: Apache-2.0, MIT
- libadwaita [79]: MIT
- gtk4 [80]: MIT
- glib [81]: MIT
- directories-next [84]: Apache-2.0, MIT
- wayland-protocols-plasma, wayland-protocols-wlr, wayland-client [47]: MIT
- serde [86]: Apache-2.0, MIT
- serde-json [87]: Apache-2.0, MIT
- once_cell [74]: Apache-2.0, MIT

14.7.3.2 Keyboard Plugin

- ReSet-Lib Section 14.7.3: GPL-V3-or-later
- DBus, DBus-crossroads, DBus-toktio [40]: Apache-2.0, MIT
- libadwaita [79]: MIT
- gtk4 [80]: MIT
- glib [81]: MIT
- directories-next [84]: Apache-2.0, MIT
- toml [77]: Apache-2.0, MIT
- xkbregistry [88]: WTFPL
- xdg [85]: Apache-2.0, MIT
- once_cell [74]: Apache-2.0, MIT

14.8 Retrospective

In this section, both team members will reflect on their experience in the project and elaborate on their takeaways.

14.8.1 Fabio Lenherr

This project reinforced the importance of documentation for me. We were stuck multiple times in this project, solely due to ambiguous documentation of an endpoint for our two plugins. It made it very clear that even the best-looking API cannot be used by developers if it does not tell you *how* to use it. And it goes even further than just using the API, if developers have to first experiment with the API in order to understand it, it is essentially impossible to create tests for the API before you already implemented what you needed. Especially with an API, the test-driven development could make sense, but it is only possible with the previously mentioned requirement.

Further, as the agile methodology states, users should test the application continuously. This was proven correct as the user tests resulted in edge cases being found and documented, which in return resulted in late fixes for said issues. While we did not have a customer for this project, and therefore could not rely on this, it did make it clear, that users need to be at the forefront of testing for a great result.

In the end, I am satisfied with the result of this project, and I am looking forward to maintaining the project after this thesis. It will be the ultimate test for the viability of both the plugin system and the original ReSet application.

14.8.2 Felix Tran

Overall, I am satisfied with the plugin system and the plugins we created over the last few months. This project has been a significant learning experience for me that will be useful even years down the line. While we did plan too much and had to cut some optional features, all the main features have been implemented successfully. This is again a reminder, that project planning can be easily overestimated like in the SA, and we're in a lucky position where we can just remove features at a whim.

I learned a lot in this project because it was the first time I had to work with a low-level language. Though I would not do it again willingly, it was nevertheless a good experience to have as the knowledge gained working on this project can also be applied to other languages.