Reoxidizer: Idiomizing Rust Code Using Recursive LLMs

Addison Goolsbee
Addison.goolsbee@yale.edu

Advisor: Lin Zhong
lin.zhong@yale.edu

Department of Computer Science
Yale University
May 1, 2025

Reoxidizer: Idiomizing Rust Code Using Recursive LLMs

Addison Goolsbee

Abstract

Can a large-language-model (LLM) agent autonomously migrate legacy Rust code to the fully idiomatic, unsafe-free form that makes the most out of the language's advantages? To probe that question we built Reoxidizer, an open-source CLI that drives GPT-4o through (i) strategy generation, (ii) code patch generation via recursive prompting, and (iii) automated evaluation with compile, test, and idiomatic evaluation steps. The architecture is designed around five criteria—effectiveness, efficiency, adaptability, security, and transparency—and logs every prompt, diff, and verdict for auditability. We benchmark Reoxidizer on three progressively harder artifacts: Quicksort (60 LoC, direct C2Rust translation), RobotFindsKitten (1.5 k LoC CLI game), and the Theseus OS e1000 driver (700 LoC kernel module). Within 15 seconds the agent purged all 24 unsafe lines in Quicksort across two iterations, preserving semantics. Given 15 minutes, it only removed four of 460 unsafe lines in RobotFindsKitten while keeping gameplay intact, but on e1000 it failed to perform a single positive change. These results delineate today's frontier: LLM agents are surprisingly potent for smaller codebases, and Reoxidizer can run on its own, perfecting any given file that is simple enough. However, as codebases get larger and more complex, and the challenge of idiomizing the code begins to require hundred-line changes, LLM agents still struggle to generate code that even can compile. Our contributions are (i) the first end-to-end agent architecture for autonomous Rust idiomization, (ii) an empirical boundary-case analysis of current LLM limits, and (iii) a public codebase. We conclude that autonomous idiomization is possible in principle but not yet feasible at scale, and closing that gap will demand a concerted team effort of language-tool researchers and LLM engineers.

1 Introduction

Rust has emerged as a powerful systems programming language, celebrated for its ability to enforce memory safety, concurrency, and performance without incurring the overhead of a garbage collector. Central to Rust's promise is its ownership model and linear type system, which eliminate common programming errors such as data races, null pointer dereferencing, and buffer overflows at compile time. These benefits have positioned Rust as an ideal candidate for developing performance-critical applications like operating systems, embedded systems, and networking software.

While writing safe Rust code (code that lacks *unsafe* blocks) guarantees many of these safety benefits, fully realizing Rust's ability to guarantee correctness demands writing especially idiomatic Rust code—a task requiring deep understanding and deliberate thought. Idiomatic Rust involves practices like encoding invariants into types and structuring programs to leverage Rust's compile-time checks optimally. Such code enhances correctness, maintainability, and robustness, reducing the likelihood of bugs and undefined behavior. Unfortunately, generating idiomatic Rust automatically remains elusive, as it typically involves high-level reasoning that cannot be automated by any sort of programmatic tool.

Recent advancements in Large Language Models (LLMs) present a promising avenue for automating the complex cognitive task of Idiomizing Rust code. Specifically, the potential for recursive or self-prompting LLMs to iteratively refine code opens possibilities for creating autonomous systems capable of improving codebases without human intervention, turning what might be an arduous and often infeasible task into one that requires minimal human intervention. Leveraging this concept, this research investigates the feasibility of autonomous Rust idiomization by introducing Reoxidizer, a tool designed to iteratively refactor Rust programs into more idiomatic code.

Reoxidizer accepts a Rust source file, a build command, and a test script, autonomously processing these inputs through iterative cycles of strategy generation, code transformation, and evaluation feedback loops. By repeating these cycles autonomously, Reoxidizer aims to progressively produce cleaner, safer, and more idiomatic Rust code.

This paper evaluates Reoxidizer's effectiveness through three distinct test cases of varying complexity: Quicksort, a low-complexity sorting algorithm directly translated from C; RobotFindsKitten, a medium-complexity command-line game; and the TheseusOS e1000 network driver, a high-complexity systems-level component. While initial tests demonstrated promising results with simpler cases such as Quicksort, significant challenges emerged with more complex cases, particularly concerning the generation and application of large-scale idiomatic changes and effectively managing iterative compilation errors.

Ultimately, this research highlights both the potentials and inherent challenges of automating code idiomization with LLMs, underscoring critical barriers in automated strategy generation, reliable code transformation, and the robustness of evaluation feedback mechanisms. Addressing these challenges may pave the way for practical tools capable of significantly reducing the manual effort required in transitioning legacy or non-idiomatic Rust codebases to safer, idiomatic Rust.

2 Background

2.1 The Rust Programming Language

Rust is a modern systems programming language released in 2015 that ensures memory safety and high performance without relying on a garbage collector. Its unique ownership model enforces strict rules around resource borrowing and ownership, eliminating common runtime errors such as null pointer dereferencing, data races, and buffer overflows—with no performance penalty. Rust achieves performance comparable to C while offering significantly stronger safety guarantees. These properties make it particularly well-suited for performance-critical and concurrent systems, including operating systems, embedded systems, and network drivers. The language's linear type system further strengthens safety by enforcing proper resource management and preventing logic errors at compile time.

Theseus OS is a novel operating system written entirely in Rust, and it demonstrates Rust's advantages in a systems context. Theseus benefits from Rust's compile-time memory safety, runtime-persistent bounds checking, and lack of undefined behavior, even in low-level components like device drivers and kernel modules. Unlike traditional OSes, Theseus uses a single address space, reducing context switching overhead and simplifying inter-component communication—enabled in part by Rust's safety guarantees. Its architecture also supports dynamic, modular evolution of components, allowing live updates without compromising stability or isolation [1]. None of these features would exist if not for Rust.

Writing code in an idiomatic Rust style maximizes these benefits. Rust code is idiomatic if it aligns with the language's design philosophy: encoding invariants into types, minimizing unsafe blocks, and leveraging the ownership and borrowing system to enforce correctness at compile time. Idiomatic Rust code is not only more readable but also can ensure correctness and reduce the surface for bugs. The concept of intralingual design captures the essence of idiomatic Rust. As described by Ijaz et al., intralingual design means using the Rust type system itself as a proof mechanism, structuring code such that the compiler enforces correctness properties directly. Rather than relying on external specifications or runtime checks, programmers encode constraints and relationships into type definitions, trait bounds, and ownership lifetimes. This approach was successfully applied to the memory management subsystem of Theseus OS, leading to a design that significantly reduced the formal verification burden while uncovering a previously undetected bug [2]. These results highlight that idiomatic Rust is not just stylistic preference—it is a practical path to simpler, more robust, and verifiably correct systems.

In short, adopting Rust over traditional languages like C is advantageous due to its strong safety guarantees and performance. Moreover, it is essential for Rust code to be idiomatic rather than merely correct, as idiomatic code fully exploits Rust's compiler-enforced safety and type-driven design features, resulting in simpler, safer, and more maintainable software. However, writing idiomatic Rust code is challenging because it requires deep familiarity with Rust's intricate type system, ownership rules, and best practices—demands that exceed those of conventional languages like C. Given this complexity, there is substantial motivation to automate the idiomization process, enabling developers to reliably produce idiomatic Rust without extensive manual effort.

2.2 Rust Idiomization and Challenges

Improving Rust code to make it more idiomatic involves addressing various aspects of code quality and safety. Broadly speaking, Rust idiomization can be categorized into three primary approaches:

1. **Reducing usage of unsafe keyword**: Rust has a special *unsafe* keyword which allows one to write blocks of code that don't need to adhere to the various compiler restrictions that standard, safe Rust code usually needs to, such as raw pointer manipulation. By simply reducing or eliminating unsafe blocks with safe alternatives, we can significantly or entirely regain the safety guarantees inherent to Rust.

2. **Making unsafe blocks sound**: Sometimes, using the unsafe keyword is necessary, and other times, avoiding using the unsafe keyword is so complex that it's not worthwhile. Just because code is in an unsafe block, doesn't mean the code is going to introduce problems—it merely means that the Rust compiler can't guarantee that the code won't introduce problems. If the block does introduce problems, it is called *unsound.* To make unsafe blocks sound, developers must ensure through careful reasoning or formal verification that all invariants expected by the compiler are manually upheld.

3. **Making already safe Rust code more idiomatic**: Even safe Rust code can often be made more idiomatic by adopting best practices such as type-driven development, better leveraging Rust's trait system, and structuring code to explicitly encode invariants into types. [2] demonstrates why this is desirable

Despite their benefits, these three approaches to idiomization are difficult for human developers to apply consistently. Reducing the use of unsafe often requires deep familiarity with safe alternatives and a willingness to significantly restructure code—something that is

both time-consuming and error-prone. Making unsafe blocks sound demands meticulous reasoning about low-level memory invariants, and even small mistakes can compromise program safety. Finally, transforming safe Rust into idiomatic Rust requires a strong understanding of best practices, type-level design, and long-term maintainability trade-offs, all of which are hard to learn and easy to overlook. For large or legacy codebases, this makes manual idiomization a prohibitively expensive task, even for experienced Rust developers.

2.3 Existing Tools and Limitations

One notable existing tool in this domain is C2Rust, which automatically translates C code into Rust. C2Rust first generates a semantically equivalent but highly unsafe Rust translation. It then provides scripts designed to refactor this generated Rust code into safer constructs. However, these scripts have significant limitations, primarily in their inability to fully eliminate unsafe blocks. They result in roughly one-to-one translations of C to Rust and therefore do a very poor job at being idiomatic, as C does not share Rust's linear type system. Because removing unsafe constructs typically requires semantic reasoning beyond basic syntactic patterns, C2Rust's approach is inherently limited [5].

The ACM analysis of C2Rust highlights the key issues faced by such automated tools. Common errors introduced by naive transformations include incorrect handling of memory safety, improper use of Rust's ownership model, and flawed logic arising from direct semantic translations. This underscores the critical need for human-like reasoning in refactoring processes, as purely programmatic approaches are often insufficient to achieve idiomatic standards and can inadvertently introduce new safety vulnerabilities [3].

2.4 Large Language Models for Code

Large Language Models (LLMs), which have seen significant advancements recently, are powerful generative models capable of producing contextually appropriate text and code. Their applicability in programming extends beyond mere syntactic transformation—they can mimic human cognitive processes, capturing semantic nuances and context in ways that traditional programmatic tools cannot.

The preference for LLM-based approaches in idiomatic Rust refactoring stems from their ability to simulate human-like "thinking." Rather than relying solely on predefined rules and syntactic patterns, LLMs can contextualize, reason about the codebase, and generate semantically meaningful transformations. This flexibility and adaptability make them particularly suitable for idiomatic refactoring tasks, as they can dynamically adjust strategies based on previous outcomes and contextual insights.

Furthermore, LLM-driven systems can iteratively refine their outputs through recursive prompting, enabling continuous improvement and adaptability to diverse

scenarios. Combining recursive prompting with code, one could make a system that runs for hours at a time, iteratively improving code. This method provides a marked advantage over traditional tools, offering not only greater effectiveness in achieving idiomatic standards but also better visibility and adaptability during the refactoring process.

3 Methodology

3.1 Reoxidizer Overview

Reoxidizer is an autonomous Rust idiomizer built with Python and OpenAI's API, specifically using GPT4o [7]. It is a CLI tool that takes a single Rust file, a build command, and a test script, an optional Cargo.toml file (if you want to have Reoxidizer add packages), and runs for up to several hours, iteratively improving the Rust file by making it safer and more idiomatic.

The goals of Reoxidizer can be summarized as follows, in decreasing order of importance:

1. **Effective**: The program can successfully idiomize code. The more difficult the idiomized code, the more effective. It is more effective if it can keep idiomizing code until there is little or none left to idiomize.

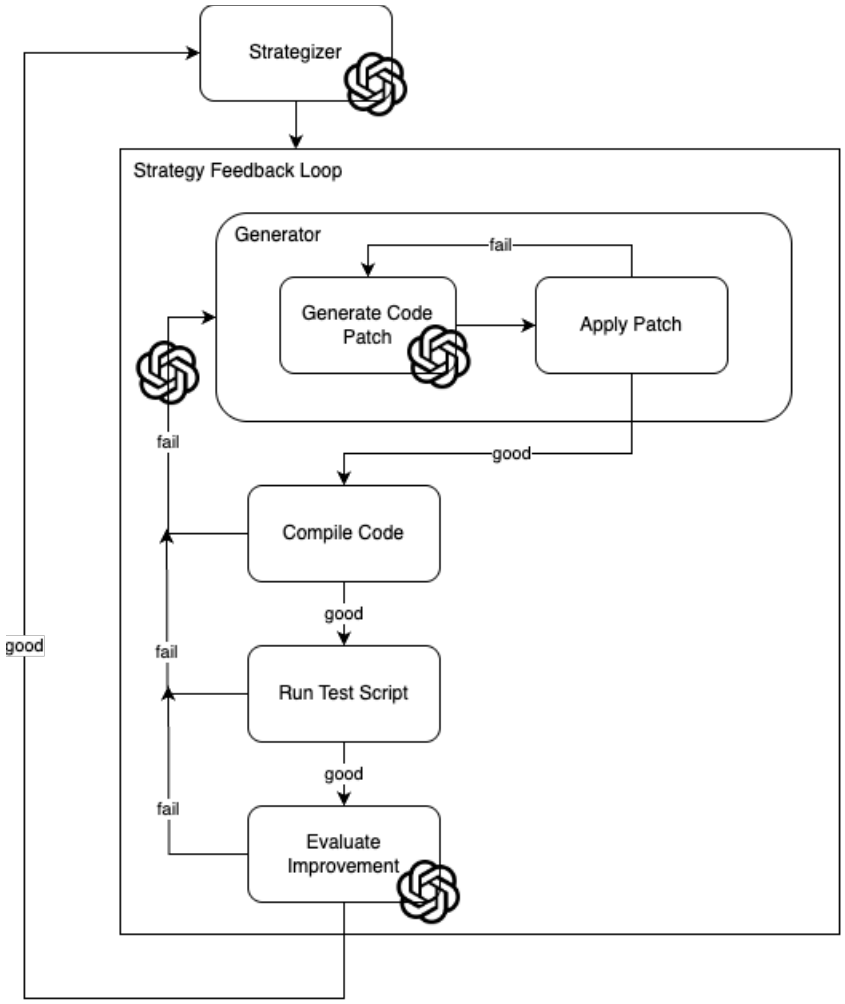
2. **Efficient**: The program can perform its operations (code generation, testing, etc.) quickly and in a cost-efficient manner.

3. **Adaptable**: The program can work on a variety of inputs and the code is easily able to accommodate input changes.

4. **Secure**: The program cannot mess up the original codebase or run dangerous commands on the host computer.

5. **Transparent**: It is clear to the user what the program is doing at any given point, and how it made its decisions. Runs of the program are auditable.

The design of Reoxidizer is as follows:



System Diagram of Reoxidizer. The OpenAI logo is present in areas where an LLM is used

Reoxidizer can be thought of as having three separate looping system components:

1. **Strategizer**: This is where the program begins. The strategizer looks at the code and generates a strategy for improving the code, then sends that strategy to the strategy feedback loop where the other two systems reside. Once the strategy feedback loop ends, the strategy is added to an ongoing list of successful and unsuccessful strategies, and a new strategy is generated based on them.

2. **Generator**: With a given code improvement strategy, the generator looks at the code and the Cargo.toml file (if it exists) and generates a set of edits to apply. The edits are applied, and if the application step failed, the generator will try again. If application fails enough times, the generator will exit the generation loop, add a message on why generation/application failed, and re-enter the generation loop, using that message, alongside the other inputs, to generate new code

3. **Evaluator**: Once new code is generated and applied successfully from the generator component, the evaluator runs a series of tests on the new code to verify its quality. First, the build script is run. This is mostly to check for linting errors (which are numerous in Rust). If the build script has an error output, it is evaluated, and Reoxidizer returns to the generator with a message on what went wrong in the build step. If the build step succeeds, the evaluator moves to the test script step. The user supplies the test, which can vary from a basic integration test to a full unit test suite, and if the script output matches the desired output, it passes the step. Otherwise, Reoxidizer will again return to the generator. Finally, the evaluator evaluates 1) whether there are less unsafe lines than previously, and if that doesn't succeed, then 2) whether it thinks the new code is more idiomatic than the old code. If either succeed, then this strategy worked, and we now loop back and generate a new strategy.

Within the strategy feedback loop, if enough failures occur (as measured by number of restarts back to the generator loop), Reoxidizer will abandon the strategy and generate a new one. Previously failed strategies do not get repeated. Reoxidizer ends when the user stops it, or when the strategy loop fails enough times in a row, signaling that Reoxidizer is struggling to make any more progress.

```
(.venv) addisongoolsbee:thesis$ /Users/addisongoolsbee/Desktop/thesis/.venv/bin/python /Users/addisongoolsbee/D
esktop/thesis/src/main.py
Logs for this run can be found in  /Users/addisongoolsbee/Desktop/thesis/src/log/run002
Generating new strategy... (1.03s)
Prompt 1: Change the `swap` function to use a safe Rust implementation using slice indexing instead of raw poin
ters, and remove the `unsafe` keyword from the function and its calls.
Generating code... (7.56s)
Building... (1.97s)
Analyzing compilation... (3.33s)
Compilation ❌ (5.51s)
Attempt took 13.17s
Prompt 2: Import the standard library's slice module using `use std::slice;` at the top of the file. Change the
 `swap` function to use a safe implementation using slice indexing instead of raw pointers, and update the `par
tition` function to call `swap` using slice indexing. Ensure that you eliminate the use of `unsafe` in the `swa
p` function and its calls.
Generating code... (5.70s)
Building... (0.52s)
Analyzing compilation... (2.82s)
Compilation ✅ (3.55s)
Running test script... (0.10s)
Test script ✅ (0.21s)
Result: 24 unsafe lines -> 21 unsafe lines
Code safety improved ✅
Result: code safety improved in 2 attempts and 22.74s
Updated best code
Generating new strategy... (2.38s)
Prompt 1: Replace the use of raw pointers in the `partition` and `quickSort` functions with safe Rust slices, a
nd remove the `unsafe` keyword from these functions.
Generating code... (10.64s)
Building... (0.72s)
Analyzing compilation... (2.40s)
Compilation ✅ (3.32s)
Running test script... (0.10s)
Test script ✅ (0.21s)
Result: 21 unsafe lines -> 0 unsafe lines
Code safety improved ✅
Result: code safety improved in 1 attempt and 14.29s
Updated best code
No unsafe lines remaining. Exiting.
Reverted code to original state.
Logs saved to  /Users/addisongoolsbee/Desktop/thesis/src/log/run002
Best generated code saved to  /Users/addisongoolsbee/Desktop/thesis/src/log/run002/best.rs
```

A typical 14-second run of Reoxidizer on Quicksort

There is a config file to set up the various constants like the build script command and the location of the Rust file. There is also a detailed logging system that generates folders for

each strategy and files for each step within a strategy loop, as well as summary files for easier analysis. Finally, Reoxidizer runs as a CLI with useful, succinct messages as well as timings on every step.

3.2 The Strategizer

The Strategizer acts as Reoxidizer's "planning brain." Instead of jumping straight into patch generation, we first craft a high-level transformation goal that is small, isolated, and targeted at idiomizing Rust code. Separating planning from editing lets Reoxidizer converge incrementally: each strategy targets a single unsafe hotspot, minimizing collateral damage and making rollback trivial. It also yields a clean audit trail—every improvement can be traced back to one sentence explaining why it was attempted, boosting transparency and reproducibility.

To propose a new strategy, the Strategizer ingests the current Rust file, the (optional) Cargo.toml, and a log of all prior strategies with their outcomes. A fixed template instructs the LLM to produce a brief yet specific plan that (i) removes or isolates specific unsafe uses, (ii) touches as little code as possible, and (iii) avoids areas already deemed unhelpful. Each proposed plan is tagged with a unique ID and stored alongside its eventual result (SUCCESS, CODE_SAFETY_UNCHANGED, etc.), along with metrics such as attempts and run time. The prompt also explicitly enforces specific common behaviors, including a ban on attempting sweeping refactors, mandating removal of the unsafe keyword when no longer needed, and forbidding touching external files unless they are known to exist. Below is an example of a good prompt:

> Change the `swap` function to use a safe Rust implementation using slice indexing instead of raw pointers, and remove the `unsafe` keyword from the function and its calls.

3.3 The Generator

The generator is the component that generates new code based on the current strategy and its understanding of the current code. Some of the following methods went into the development of the generator:

Efficiency via patch-then-apply. Early prototypes asked the LLM to emit the entire new code on every generation, which for an original Rust file that was 700-1,500 lines long, costed ~2 minutes per generation and frequently introduced formatting drift. In any given strategy, the LLM tends to only update 3-100 lines, so there is a lot of wasted information slowing down Reoxidizer needlessly if the entire code file is generated each time. Instead, the current generator massively increases efficiency by splitting generation into two steps: the model now returns a compact change file—a JSON array of replace-this-with-that

snippets—while a deterministic Python routine splices those edits into the source. This reduced generation time to the 10-20 second range, and token usage dropped by an order of magnitude. Whenever a task can be expressed as pure text manipulation, handing it to ordinary code rather than the LLM proved the single biggest speed gain.

Recovery methods. When Reoxidizer reaches a failed step in the strategy feedback loop, instead of abandoning the strategy, it alters it based on the type of failure and restarts the loop. Places it can fail include too many failed code generation attempts, build fail, test fail, or evaluation fail. Three recovery methods were tested:

1. **Reset to original code.** After a failure, reset the code and generate a new strategy using the same 'core strategy' as the initial strategy, but with extra details explaining how to implement the strategy such that it would work. This approach has the downside in that you may lose lots of progress when the loop fails—if the only thing that failed about the code was that there was a missing brace, you must regenerate 100s of lines of code changes, you waste lots of time. The upside is that Reoxidizer always stays focused on the task.

2. **Iterate on the mutated code.** After a failure, keep the new, non-functional code, and generate a new strategy to fix the code, making sure the end goal is still in line with the initial strategy. This way, the model is great at tackling smaller problems that arise after the first attempt, especially when it comes to syntax errors. The downside is that if the first generation went poorly, the model will spend the rest of the feedback loop trying to fix its implementation, instead of just restarting from scratch.

3. **Iterate on the change file.** After a failure, feed the model the original code plus the most recent patch. The strategy is updated by adding extra details, like in (1). Now it sees both the clean baseline and its latest diff, so it can refine the patch without losing history. This fixes the problem in (2) of one bad generation spiraling the whole feedback loop out of control, but it requires the model to understand what the previous change file was trying to accomplish, which can be difficult considering it is in a strange format and can have hundreds of lines of changes.

Surprisingly, recovery method (1) remained the most effective. Using the previous prompt as an example, a strategy iteration might look like this:

> Change the `swap` function to use a safe Rust implementation using slice indexing instead of raw pointers, and remove the `unsafe` keyword from the function and its calls.

Then, after code is generated that leads to a compile error, the strategy is modified:

Import the standard library's slice module using `use std::slice;` at the top of the file. Change the `swap` function to use a safe implementation using slice indexing instead of raw pointers, and update the `partition` function to call `swap` using slice indexing. Ensure that you eliminate the use of `unsafe` in the `swap` function and its calls.

3.4 The Evaluator

The evaluator runs the new code through a series of tests to ensure it still works and that it's more idiomatic. If any step fails, it analyzes why and generates the reason, along with specific details on how to fix it, which it then passes back to the generator to restart the feedback loop.

1. **Compilation.** This is the most basic test—can the code build? The compilation step mostly catches syntactical issues, and while it is a simple step, it is also the most frequently failed step.

2. **Testing.** If the program can build, the next thing we need to know is if it still maintains functionality. Rather than trying to infer correctness heuristically or analyzing the new code with the LLM, testing is offloaded to the user. While this incurs an additional layer of effort that comes at the cost of Reoxidizer's adaptability, it completely solves the issue of maintaining functionality. The test script can be as specific as it needs to be, from just a single integration test (see if quicksort can sort a list), to a full unit test suite.

3. **Evaluation.** Passing tests only proves non-regression; we also want the code to be more idiomatic. As touched on in background, we're looking to (1) reduce uses of the unsafe keyword, (2) make unsafe code sound, and (3) make safe code more idiomatic. Because (2) is relatively uncommon, Reoxidizer only focuses on (1) and (3). Quantity of unsafe lines is much easier to measure than idiomatic-ness, so the evaluation step follows a two-pass system. First, count the lines that reside inside unsafe blocks in the old code and the new code. This requires no interaction with the model and therefore is effectively instantaneous. Counting lines of unsafe code, while primitive, is also a surprisingly accurate proxy for idiomatic Rust code. However, in more complicated Rust files, removing even a single line of unsafe code may take multiple consecutive strategies, so often this evaluation step isn't enough. If this step passes, the feedback loop is done; the strategy worked, and now we return to generate a new strategy. If it didn't work, the evaluator asks the model "given the two versions, which is more idiomatic?" A positive verdict here will also complete the strategy.

3.5 Test Cases

To test Reoxidizer I gathered three test cases which each fill specific niches.

1. **Quicksort:** A Rust implementation of the quicksort algorithms. Originally written in C, then translated to unsafe and unidiomatic Rust using C2Rust. ~60 lines of code, low complexity. Quicksort is meant to be the base case for Reoxidizer, just to prove that it works.

2. **RobotFindsKitten:** A classic CLI game where the player is "robot", and moves around the screen, bumping into various objects until they find "kitten". Translated directly from C using C2Rust. ~1500 lines of code, moderate complexity. RobotFindsKitten is meant to stress Reoxidizer on long programs. It has some low hanging fruit idiomizations, but also some quite complex ones.

3. **TheseusOS e1000 Networking Driver:** The network interface card driver for the e1000 for TheseusOS. Taken from the TheseusOS GitHub repository, this is the 7-year-old version of the e1000, before it was changed to be idiomatic and fully safe. This was then taken and modified enough to work with the current version of TheseusOS. This is the primary test for Reoxidizer: if Reoxidizer can pass this test, it proves beyond reasonable doubt that autonomous idiomization is possible. ~700 lines of code and high complexity, including lots of function calls unique to TheseusOS. There is no low-hanging fruit for the e1000, and every removal of unsafe code requires multiple steps. The test script is "ping 8.8.8.8".

Together these three cases span easy demo → realistic mid-size app → expert-level systems code, giving a balanced view of Reoxidizer's effectiveness, robustness, and scalability.

4 Results

Table 1 reports outcomes from running each of the three tests for up to 15-minutes on an M1 MacBook Pro. Reoxidizer removed 100 % of unsafe in Quicksort (24 → 0) in 2 iterations but only 0.9 % in RobotFindsKitten and 0 % in e1000.

Test	Unsafe line change	Evaluation	Duration
Quicksort	24 -> 0	Perfect	15 seconds
RobotFindsKitten	460 -> 456	Minimal improvement	15 minutes (capped)
TheseusOS e1000	6 -> 6	No effect	15 minutes (capped)

Quicksort succeeded where the other two test cases didn't simply because it was shorter and less complex. An LLM could have refactored the quicksort code in a single prompt with high accuracy; splitting quicksort into multiple strategy steps is an even simpler feat. While quicksort's success shows that Reoxidizer works as a proof of concept, Reoxidizer's failure to make meaningful improvements to more complicated code indicates its inability to iteratively idiomize complex, large code.

In the RobotFindsKitten run, out of 6 strategies with 52 strategy iterations, 12% failed to generate valid code changes, 86% failed the compilation test, and 2% succeeded. Looking through the logs manually, it's clear that the generator was the weakest link. For the strategizer, Reoxidizer rarely struggled to properly diagnose how to make the code more idiomatic, although sometimes it lacked specificity. For the evaluator, the build, test, and evaluation steps did an accurate job at judging the effectiveness and idiomatic-ness of the new code. For the generator, though, Reoxidizer would consistently fail to generate code that made it past the compilation stage. Even after modifying the strategizer to generate a preset great strategy, Reoxidizer still ran into lots of trouble idiomizing code. Noticing this, I tried to change the recovery method of the feedback loop to build on top of previous iterations (see 3.3), but regardless of the system, getting past compilation persisted as the toughest challenge.

On RobotFindsKitten, with a large codebase, generating a legitimate change file proved to be challenging as well. Out of 99 code change file generations, only 22% succeeded This can likely be attributed to there just being too much code for the model to output a set of changes with no mistakes.

The e1000 had the biggest challenge, though, because all idiomization attempts required sweeping changes to the codebase. When done by hand, one of the simpler

idiomization strategies—making better use of RAII when initializing the driver—required over 200 lines in changes. This was not feasible for Reoxidizer, and it never made it past the compilation step.

Reoxidizer shines when the target patch is small and self-contained—it can purge all unsafe code in Quicksort and even chip away at RobotFindsKitten under a 15-minute budget. But once unsafe is woven through multiple modules and type layers, and even the simplest idiomizations take hundreds of lines, the agent stalls, unable to generate code good enough to compile, and unable to effectively fix the code it generates.

5 Related Work

Research on Theseus OS demonstrated that writing OS code in Rust can unlock live-update modularity and state-preserving restarts—capabilities unattainable in a conventional C kernel [1]. Follow-up work demonstrated the value of making code more idiomatic (in the paper called intralingual design), showing that re-encoding invariants into Rust's type system slashed the proof burden for memory management subsystems and reduced bugs [2].

C2Rust and its pitfalls. Automatic C → Rust translators are the obvious baseline for any safety-upgrade workflow. The most mature is C2Rust, which parses C with Clang, outputs a one-to-one Rust twin that compiles but is wrapped almost entirely in unsafe, and then offers a refactoring script meant to chip away at the raw pointers and extern calls [5]. An empirical audit of 17 codebases translated by C2Rust found that the generated Rust remains dominated by raw-pointer dereferences, aliasing around malloc/free, and naïve void-pointer casts; even after the scripted passes, over 80 % of functions stayed unsafe and many new borrow-checker errors emerged when developers tried to hand-clean the code [3].

Using LLMs for coding has been of interest since LLMs became widespread, and at the frontier of using LLMs for coding tasks is the idea of treating a prompt like a real program, given more precise control flow, deterministic output, and 30-80% fewer calls/tokens for the same outputs. Reoxidizer did not go down this route, but perhaps it would have tipped to scale and made Reoxidizer more effective [4].

Recent LLM agents specially designed for coding tackle the generator stage much more effectively than Reoxidizer, notably Cursor AI, which was tested manually in comparison. While hallucination and compiler errors are still common with Cursor AI, its ability to make large changes relatively quickly is impressive. Perhaps if Cursor AI had an API, it could replace the generator component, and Reoxidizer would go from infeasible to feasible [6].

6 Conclusion

Our central question was straightforward but ambitious: Can an LLM-driven agent autonomously refactor arbitrary Rust into safer, fully idiomatic Rust? With Reoxidizer we set out to answer that question while meeting five design goals—effectiveness, efficiency, adaptability, security, and transparency (Sec. 3.1):

1. **Effectiveness:** effective enough for the toy case of quicksort, but failed to work to a level where it could idiomize complicated or long code files, let alone do sweeping refactors. Reoxidizer is not very effective.

2. **Efficiency:** thanks to the patch-then-apply format of the generator, Reoxidizer can run through hypothesis relatively quickly, around 10-20 seconds per significant generation.

3. **Adaptability:** Reoxidizer is designed in a modular way that enables it to work for many types of files, but it is hindered by only working for a single file at a time, being limited by the size of the file, and requiring a test script.

4. **Security:** Because of the strictly defined ways in which the model can edit code, security is fully met.

5. **Transparency:** Log folders, timing data, live CLI output all clearly display what the program is doing and why it makes its choices

The headline result is mixed. Under a 15-minute budget Reoxidizer eliminated all 24 unsafe lines in Quicksort, trimmed only four lines in RobotFindsKitten, and failed to land a single compiling patch on the Theseus e1000 driver. The bottleneck is clear: today's LLMs still hallucinate or break syntax when asked for multi-hundred-line, semantics-preserving edits, and our recovery loops cannot overcome that drift fast enough. Yet the experiment yields two concrete contributions:

1. A full agent architecture for autonomous Rust idiomization—separated into strategizer, generator, and evaluator loops—along with open-source code, reusable prompts, and a rich audit trail.

2. An empirical boundary-case analysis showing where current self-prompting LLM agents succeed (localized refactors) and where they tend to collapse (cross-cutting, type-driven redesigns).

With Reoxidizer we demonstrated that autonomous idiomization is possible in principle—the system can plan, patch, compile, test, and judge its own work—but it is not yet feasible at scale. Closing that gap will require larger context windows, stronger static-analysis guidance, and, very likely, a concerted team effort of researchers and tool builders.

Bibliography

[1] K. Boos, N. Liyanage, and L. Zhong, “Theseus: an Experiment in Operating System Structure and State Management,” 2020. Accessed: Apr. 30, 2025. [Online]. Available: https://www.usenix.org/system/files/osdi20-boos.pdf

[2] R. Ijaz, K. Boos, and L. Zhong, “Leveraging Rust for Lightweight OS Correctness,” doi: https://doi.org/10.1145/3625275.3625398.

[3] M. Emre, R. Schroeder, K. Dewey, and B. Hardekopf, “Translating C to safer Rust,” Proceedings of the ACM on Programming Languages, vol. 5, no. OOPSLA, pp. 1–29, Oct. 2021, doi: https://doi.org/10.1145/3485498.

[4] Luca Beurer-Kellner, M. L. Fischer, and M. Vechev, “Prompting Is Programming: A Query Language for Large Language Models,” Proceedings of the ACM on programming languages, vol. 7, no. PLDI, pp. 1946–1969, Jun. 2023, doi: https://doi.org/10.1145/3591300.

[5] “Introduction - C2Rust Manual,” C2rust.com, 2025. https://c2rust.com/manual/

[6] “Cursor - Welcome to Cursor,” Cursor.com, 2023. https://docs.cursor.com/welcome

[7] A. Goolsbee. “Reoxidizer: An Autonomous Rust Idiomizer Using Recursive LLMs”, v1.0, 2025. https://github.com/AddisonGoolsbee/reoxidizer