A GenAI-Powered C-to-Rust Transpiler for Safer Systems Programming

Nabiel Ahammed

University College of Engineering, Thodupuzha
Email: nabiel.ahammed25@gmail.com

Joel B Joseph

University College of Engineering, Thodupuzha
Email: joelbjoseph03@gmail.com

***Abstract*—This project presents the Corrosion Engine, a tool that aims to automate the migration of legacy C code to Rust. Traditional rule-based transpilers often struggle with semantic accuracy and the intricacies of the Rust ownership model, which can lead to significant manual intervention. To address these challenges, Corrosion Engine combines static code analysis with a Generative AI approach (leveraging Google Gemini) in a structured, multistage pipeline that includes preprocessing, segmentation, translation, and postprocessing. The system makes use of established compiler tools, such as Clang for parsing and Tree-sitter for syntax analysis, ensuring that the translated code is both safe and idiomatic. Built with a modern stack—React for the front-end, Django for the back-end, and PostgreSQL for data management—and containerized via Docker, the tool is designed for scalability across diverse environments. Although the generated Rust code may require further debugging and refinement, early results suggest that the Corrosion Engine significantly reduces the migration effort and enhances software reliability, particularly when addressing common memory safety issues in C. Future work will focus on integrating more advanced debugging tools and improving the handling of complex code structures.**

I. Introduction

C, as a programming language, has historically been a central component in systems software development. It is prevalent in a wide range of applications such as kernel drivers, embedded firmware, low-level libraries, and operating systems because of its low-level control of memory and hardware resources. However, this level of control comes with great risk. The lack of built-in memory safety features, coupled with the dependence on manual memory management, makes C highly susceptible to typical programming mistakes such as buffer overflows, dangling pointers, and use-after-free. Based on the 2023 Common Vulnerabilities and Exposures (CVE) database, more than 68% of critical security vulnerabilities were linked to memory corruption problems in C and C++ codebases, indicating an emergent need for safer alternatives [2].

Rust is a new programming language that guarantees memory safety and is designed specifically to address these issues. With strict compile-time rules of ownership and borrowing, Rust guarantees memory safety without the overhead of garbage collection. Rust also offers modern abstractions, good type safety, and better performance, making it a perfect replacement for C in safety-critical applications. Migration to Rust is costly, however. The translation of existing C codebases, which are typically large and consist of massive amounts of code, requires a considerable commitment of human resources. For example, Microsoft's Project Verona—a Rust initiative dedicated to the development of secure systems—estimated that translating just 150,000 lines of Windows kernel code would take more than 18 developer-months [5]. Such a duration is typically unaffordable for organizations with restricted resources or tight schedules.

To speed up the migration process, several rule-based transpilers have been developed. C2Rust is one such utility that attempts to convert C code to syntactically equivalent Rust by direct mappings [1]. However, such approaches are not feasible in real-world applications. They are biased towards maintaining unsafe blocks, C-style idiomatic expressions, pointer arithmetic, and excessive boilerplate code. As a result, the generated code does not fully utilize Rust's memory safety and idiomatic constructs, typically requiring extensive post-processing and manual tweaks.

Recent breakthroughs in generative AI, specifically with the advent of large language models (LLMs) such as Google Gemini and GPT-4, present a promising new direction. Such models can understand programming context, infer developer intent, and produce readable, idiomatic code in several programming languages [9]. In theory, AI-assisted code translation should fill the gap between the low-level semantics of C and the high-level safety abstractions of Rust. Nevertheless, LLMs are also confronted with limitations—namely, their ability to deal with large codebases with complex interdependencies. Without precise semantic context or structural metadata, AI-generated translations can be plagued by logical inconsistencies, ownership principle violations, and hallucinations.

To overcome these weaknesses, we introduce the Corrosion Engine—a new hybrid system for transpiling C code to Rust, combining the best of static program analysis and generative artificial intelligence. The Corrosion Engine builds a structured pipeline that preprocesses raw C input through GCC and Clang Abstract Syntax Tree (AST), fragments it into semantically coherent units, and annotates each unit with dependency-aware metadata. These units are subsequently fed into a generative AI model (Google Gemini), under the guidance of a custom prompting strategy involving ownership rules and type constraints [13][12][9].

This paper provides the following contributions:

1) A preprocessing pipeline for the GCC and Clang-based resolution of system and user-defined headers, macro expansion, and the construction of the abstract syntax tree (AST) representation of the source.
2) A graph-based segmentation algorithm that separates function-level translation units by examining call dependencies and data structure usage, with minimal context leakage across units.
3) A machine-learning-powered translation engine that uses metadata-aware prompting to guide the generative model into producing idiomatic, safe Rust code in accordance with Rust's ownership model.

Together, these innovations enable the Corrosion Engine to produce high-quality, ownership-accurate Rust code with minimal human intervention—essentially, bridging legacy C systems to modern, safe Rust platforms. The remainder of this paper discusses the architectural design, implementation details, and experimental evaluation of the Corrosion Engine, culminating in a comparison to existing transpilation tools and real system benchmarks.

II. Background and Motivation

The widespread use of the C programming language in safety-critical system infrastructures has been a double-edged sword for a long time. On the one hand, its unmatched performance and low-level features have made it the go-to programming language for systems programming for decades. On the other hand, its lack of inherent safety guarantees has caused a never-ending supply chain of memory-related bugs that still plague software ecosystems today.

A 2023 advisory by the United States National Security Agency (NSA) cited that memory safety violations such as buffer overflows, use-after-free bugs, and integer overflows resulted in over $6.5 billion in yearly losses due to cybercrime. Specifically, the Linux kernel has indicated that 63% of its Common Vulnerabilities and Exposures (CVEs) for the years 2020–2023 were due to memory corruption bugs. These statistics illustrate the urgent need to transition away from hand-controlled memory systems towards safer programming paradigms.

Rust has emerged as a programming language specifically designed to tackle this particular class of issues. By virtue of its new ownership model, Rust offers compile-time memory safety guarantees through lifetimes and borrowing semantics, thereby eliminating many types of runtime errors. Unlike garbage-collected programming languages, Rust accomplishes this without sacrificing performance. Empirical results, including case studies on the Mozilla Servo browser engine, have established the dramatic reduction in memory-associated security bugs, with reports of up to five times fewer security patches required after a Rust migration [8].

While these advantages are genuine, the process of manually porting existing C codebases to Rust remains an incredibly resource-intensive undertaking. This is a job that requires not only familiarization with both programming languages on an individual level but also domain-specific architectural patterns, especially for intricate legacy systems.

Attempts to automate such transpilation have resulted in rule-based transpilation tools such as `C2Rust` [7]. These tools employ LLVM-based translation methods to transpilate C constructs to Rust-equivalent syntax. Although they manage to maintain functional equivalence, they tend to lack the preservation of semantic integrity or adherence to Rust's idiomatic patterns. The generated code tries to mimic the C memory model, utilizing raw pointers, unsafe blocks, and type casts that circumvent Rust's guarantees of safety. To illustrate, generic arrays of C pointers (such as `void*`) are represented by `*mut c_void`, effectively disabling Rust's borrow checker protection and leaving correctness to the programmer.

The emergence of generative artificial intelligence presents an alternative paradigm for code translation. GitHub Copilot and Google Gemini are instances of tools that employ extensive language models, trained from large codebases, to produce functionally correct code in various programming languages. These models are particularly adept at intent comprehension and producing idiomatic form. They tend to work within a narrow contextual paradigm, though. Without access to function call graphs or memory access patterns, AI models have difficulty enforcing the ownership rules of Rust or comprehending dependencies between code blocks. This limitation leads to translations that can be incomplete or unsafe when applied to large or interdependent codebases [5].

Such observations present a fundamental insight: neither rule-based nor AI-based solutions are adequate in their own right. A hybrid solution is required, one that combines the structural correctness of static analysis with the adaptive and semantic reasoning capabilities inherent in generative AI. This notion is the genesis of the **Corrosion Engine**. By converting C code to modular blocks that possess semantic meaning and are tagged with metadata about dependency and ownership, the system allows AI models to produce safe and idiomatic Rust code without depending on learned heuristics. Furthermore, this preprocessing eliminates common problems that plague AI, such as hallucination, inconsistency, and failure to identify invariants between functions. Essentially, the Corrosion Engine turns the problem of translation from a brute-force syntax conversion to a guided reasoning problem—balancing automation and architectural integrity while producing more resilient results for actual software migration projects.

III. System Design

The Corrosion Engine implements a hybrid, modular transpilation system that federates static analysis with generative AI models. It divides the C code into semantically valuable, safe-to-translate segments, and employs AI for cases requiring semantic understanding. The system ensures memory safety, scalability, and maintainability by offloading redundant syntax-based parsing to rule-based systems and delegating complex context inference to large language models (LLMs) such as Google Gemini.

A. System Overview

The core process consists of five fundamental stages—Preprocessing, Segmentation, Metadata Generation, Translation, and Postprocessing—each designed to facilitate the systematic translation from C to Rust. The architecture ensures clean separation of concerns between static and AI-driven components, designed to handle large-scale codebases with interdependent function calls.

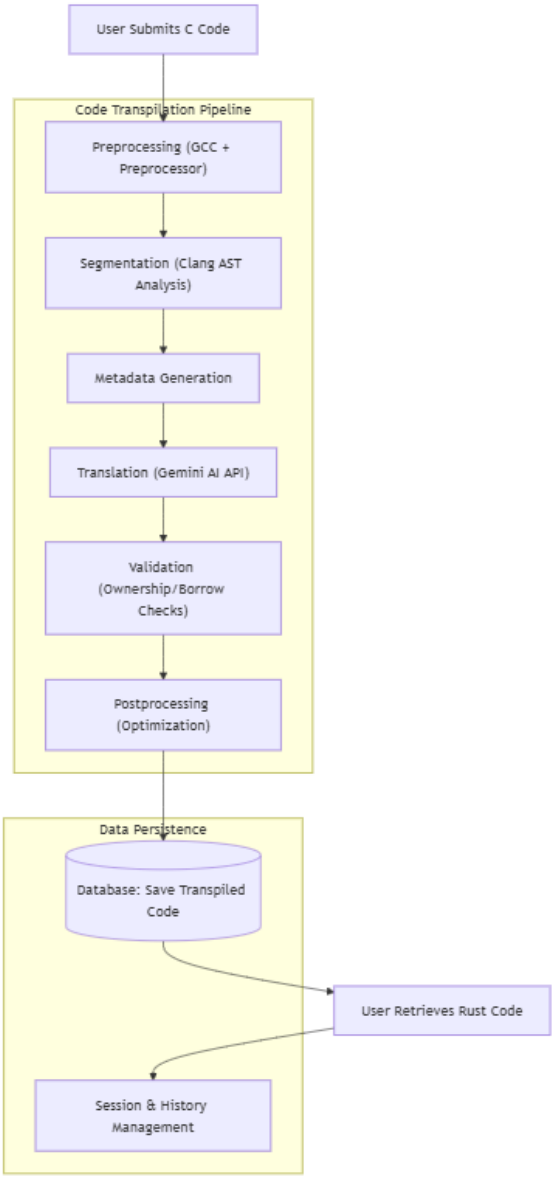

Fig. 1. Overview of our Translation Architecture

This modular architecture offers scalability and ensures that translation integrity is maintained across various system components.

B. Preprocessing

The preprocessing module serves as the initial cleanup and normalization step before translation. It utilizes GCC for macro expansion and Clang's Abstract Syntax Tree (AST) for structural parsing. This preprocessing is crucial for simplifying the code and providing the AI model with complete logical units necessary for accurate conversion.

Tasks performed during preprocessing include:

- **Macro Expansion:** Converts complex macros to full inline code using `gcc -E`.
- **Header Resolution:** Merges user-defined header files with the main source to preserve context.
- **System Header Filtering:** Excludes standard library headers, ensuring only user-specific logic is passed forward.

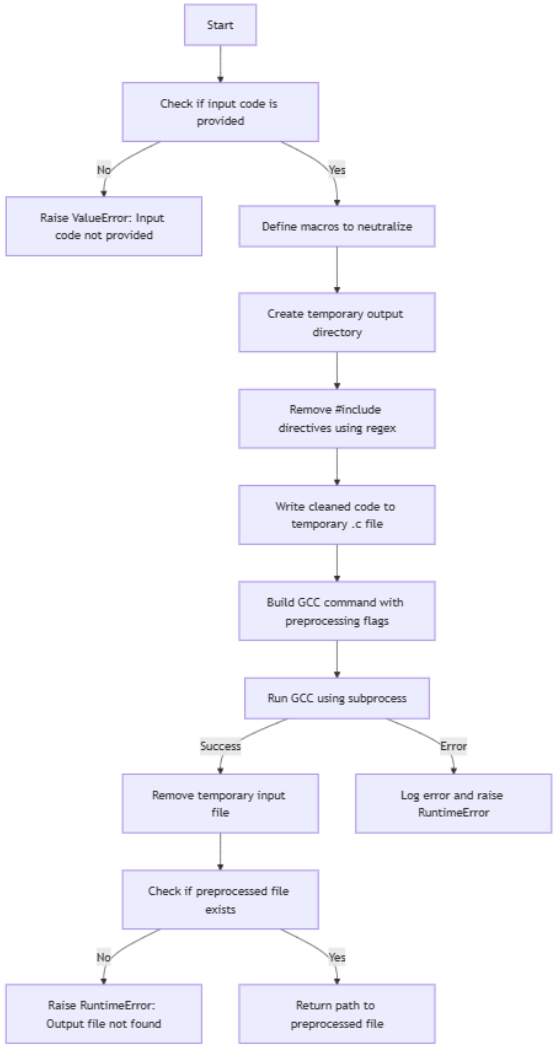

Fig. 2. Preprocessing Workflow

C. Segmentation

Once preprocessing is complete, the code is separated into independent modular parts, each representing a function, struc-

ture, or logical unit. This modularization technique reduces the cognitive load on the AI model and ensures compliance with Rust's strict memory management and function call conventions.

This operation is guided by the Clang AST and is managed through call and control graphs of the program. Segmentation allows the Corrosion Engine to split every unit and deliver it to AI processing in parallel using task queues, improving scalability and runtime efficiency.

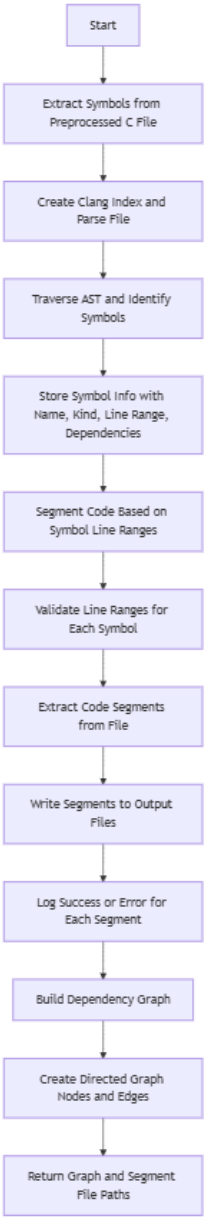

Fig. 3. Segmentation Workflow

D. Metadata Generation

The metadata generation module enriches each segmented unit with contextual hints, such as variable dependencies, call hierarchy, and ownership semantics. By leveraging libraries like NetworkX, the system forms a dependency graph, facilitating topological sorting and recursive loop detection.

This step ensures:

- **Function Execution Order:** Maintains logical flow by preserving the call chain.
- **Cycle Detection:** Prevents infinite loops in recursive function groups.
- **Ownership Semantics:** Adds metadata annotations to enable the AI to enforce Rust's strict memory model.

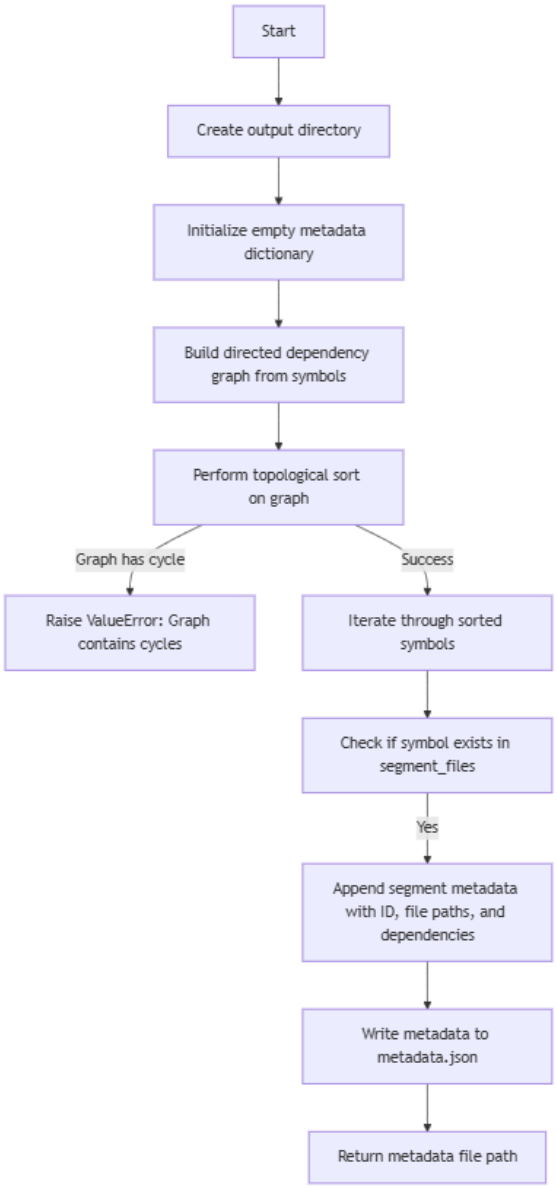

Fig. 4. Metadata Generation Workflow

With this enriched metadata, every AI request is better-informed, reducing the likelihood of hallucinations or unsafe suggestions during translation.

E. AI-Powered Translation

At this stage, the preprocessed and enriched code segments are forwarded to Google Gemini for semantic translation. The AI model receives each segment along with its associated metadata, including context such as dependencies, expected return types, and ownership rules.

Prompts are structured to include examples of idiomatic Rust constructs and safety constraints (e.g., avoid using `unsafe`, and prefer `Option<T>` or `Vec<T>` over raw pointers). The system adapts the prompts to maximize the model's compliance with Rust's safety rules.

LLMs are used to achieve:

- Idiomatic translation of C code to Rust.
- Enforcement of type safety.
- Elimination of undefined behavior.
- Improvements in code readability.

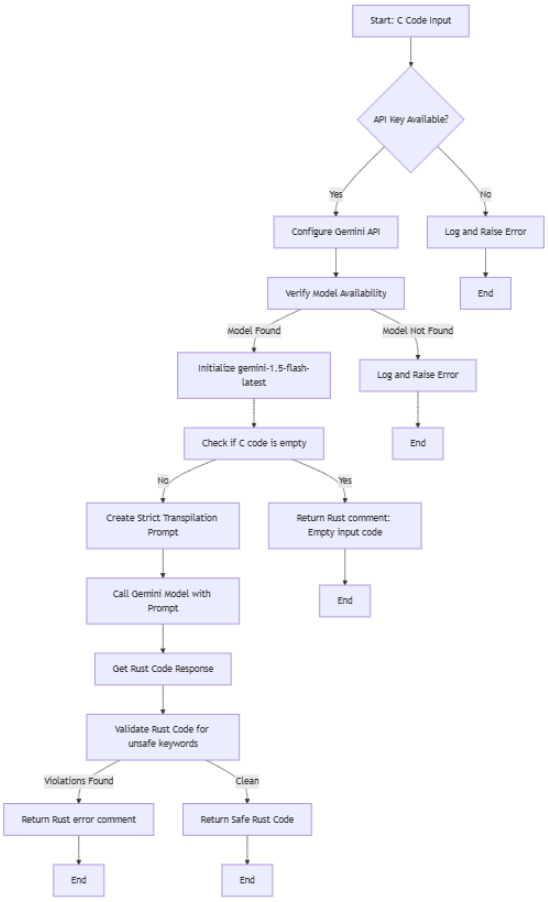

Fig. 5. AI-Powered Translation Workflow

This approach bridges the gap between rigid, rule-based translation and contextual, human-like reasoning about code structure, providing better results than either method alone.

F. Postprocessing

After translation, a postprocessing module tidies and checks the generated Rust code to ensure it meets production standards. This step involves:

- **Import Deduplication:** Merges redundant `use` statements into a unified declaration.
- **Linting and Formatting:** Utilizes `clippy` and `rustfmt` to enforce coding conventions and best practices.
- **Ownership Review:** Validates that all variables follow proper ownership and lifetime rules.

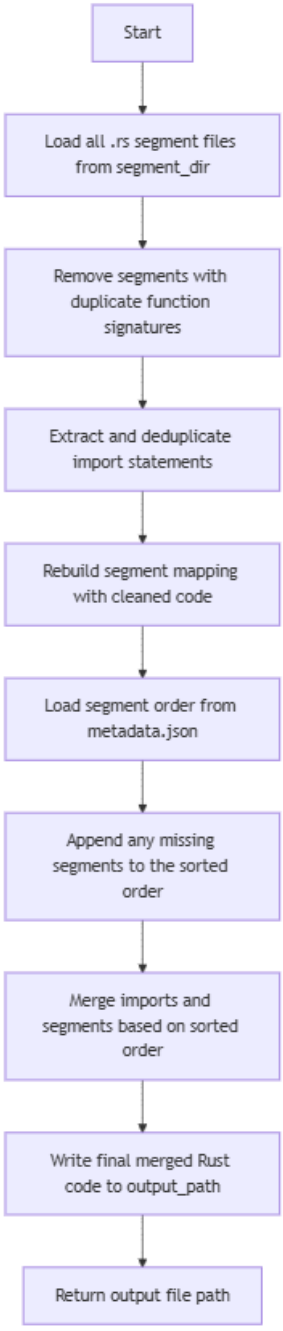

Fig. 6. Postprocessing Workflow

This ensures that the final Rust code is production-ready,

easy to read, and structurally sound under Rust's compiler checks, preparing it for integration into large-scale systems.

IV. Implementation

The Corrosion Engine is designed as a complete stack application with a React-based frontend, a Django-powered backend, and an AI-augmented code transformation engine. All processing components are implemented as standalone Python services, orchestrated by Celery for asynchronous task management. This modular approach ensures scalability, reusability, and testability, while also enabling distributed deployment.

A. System Architecture

The system architecture consists of several key components:

- **Frontend:** Developed in React.js with Tailwind CSS, providing a user-friendly web interface for developers to input C code, view the transpiled Rust output, and track the status in real-time.
- **Backend:** Built with Django and the Django REST Framework (DRF), the backend handles API routing, authentication, database operations, and task delegation.
- **Task Queue:** Powered by Celery and Redis, enabling parallel processing of multiple transpilation tasks across distributed workers.
- **Database:** PostgreSQL (Neon) is used to store user data, code snippets, and job metadata.

B. Preprocessing Module

The preprocessing module (`preprocess.py`) utilizes Python's `subprocess` module to invoke the GCC preprocessor. The following flags are used: `-E`, `-nostdinc`, and `-dD`. This process expands macros and resolves includes, flattening the code for easier analysis and translation.

```
import subprocess

def preprocess_c_code(input_file: str, output_file:
    str):
    cmd = [
        "gcc", "-E", "-dD", "-nostdinc", "-
    ffreestanding",
        input_file, "-o", output_file
    ]
    subprocess.run(cmd, check=True)
```

C. Segmentation Module

The segmentation module (`segmentation.py`) uses `libclang` (via `clang.cindex`) to parse the Abstract Syntax Tree (AST) and extract function declarations, struct definitions, and global variables. Each AST node is recursively traversed, and meaningful code units are serialized into isolated files or memory blobs.

```
import clang.cindex

def extract_symbols(file: str):
    index = clang.cindex.Index.create()
    tu = index.parse(file)
    symbols = []
    def traverse(cursor):
        if cursor.kind in (clang.cindex.CursorKind.
    FUNCTION_DECL,
                           clang.cindex.CursorKind.
    STRUCT_DECL):
            symbols.append(cursor.spelling)
        for child in cursor.get_children():
            traverse(child)

    traverse(tu.cursor)
    return symbols
```

D. Metadata Generation

The metadata generation module (`metadata.py`) annotates the extracted code segments with call graph metadata using the NetworkX library. Each function is analyzed for dependencies and invocation patterns to construct a directed graph, which helps with topological sorting and resolving function call order.

```
import networkx as nx

def build_dependency_graph(symbols):
    graph = nx.DiGraph()
    for symbol in symbols:
        graph.add_node(symbol["name"])
        for dependency in symbol["dependencies"]:
            graph.add_edge(symbol["name"],
    dependency)
    return graph
```

E. Translation Module

The translation engine (`translator.py`) interacts with the Gemini Pro API via Google's Python SDK. Each C code segment is converted into a prompt that includes safety guidelines and Rust idioms. The response from Gemini is parsed and stored as an initial Rust draft.

```
import google.generativeai as genai

def translate_segment(c_code: str) -> str:
    prompt = f"Convert the following C code into
    safe Rust:\n{c_code}"
    response = genai.GenerativeModel("gemini-pro").
    generate_content(prompt)
    return response.text
```

Prompts are designed to reduce hallucinations by explicitly stating translation constraints and providing examples of correct Rust syntax.

F. Postprocessing Module

After translation, the Rust code is validated for syntax, style, and memory safety. Tree-sitter is used for abstract syntax checks, and Clippy is employed for linting. Custom scripts are used to deduplicate `use` imports and ensure idiomatic code structure.

```
def clean_rust_code(rust_code: str):
    lines = rust_code.splitlines()
    seen_imports = set()
    final_code = []

    for line in lines:
        if line.startswith("use "):
            if line not in seen_imports:
                final_code.append(line)
```

```
                seen_imports.add(line)
        else:
            final_code.append(line)
    return "\n".join(final_code)
```

This postprocessing step ensures that the resulting Rust code adheres to the language's safety standards and is production-ready.

G. Asynchronous Task Execution

Parallelism in the pipeline is achieved using Celery workers and Redis as the broker. Each code segment is treated as an independent task, queued, and processed asynchronously.

```
from celery import Celery

app = Celery('tasks', broker='redis://localhost:6379
    ')

@app.task
def process_translation(segment):
    return translate_segment(segment)
```

This architecture allows the system to process multiple transpilation jobs simultaneously, improving throughput and scalability.

H. Deployment

The system components are containerized using Docker, with separate images for the frontend, backend, Redis, and Celery workers. Docker networking ensures environment consistency across all machines, making deployment efficient and reliable.

Example Dockerfile:

```
FROM python:3.10
WORKDIR /app
COPY requirements.txt .
RUN pip install -r requirements.txt
COPY . .
CMD ["python", "manage.py", "runserver",
    "0.0.0.0:8000"]
```

V. Simulation-Based Comparative Evaluation

To evaluate the effectiveness and robustness of the Corrosion Engine, a comprehensive simulation-based comparative analysis was conducted. The assessment was structured around both feature-specific test cases and real-world codebase evaluations, targeting three key goals:

- **Functional Correctness** – Verifying whether the translated Rust code behaves identically to its C counterpart.
- **Idiomatic Quality** – Measuring how idiomatically the translation conforms to Rust's best practices and ownership model.
- **Safety and Performance** – Analyzing the extent to which unsafe Rust constructs are avoided and how efficient the translation process is.

This section presents the methodology, test results, and comparative insights derived from benchmarking the Corrosion Engine against state-of-the-art transpilers, namely C2Rust and Corrode.

A. Simulation Environment and Experimental Setup

The testing infrastructure was designed to ensure consistency and repeatability. A high-performance development environment was used to simulate diverse workloads across transpilers.

TABLE I
Simulation Environment Specifications

Component	Specification
Processor	12-core AMD Ryzen 9 7900X
Memory	64 GB DDR5 RAM
Storage	2 TB NVMe SSD
Operating System	Ubuntu 24.04 LTS
Compilers Used	Clang 17 for C, Rust 1.78 (stable)
Corpus	50 micro-benchmark C programs and 13 open-source projects (∼116 KLOC)
Compared Systems	Corrosion Engine, C2Rust, Corrode
Evaluation Metrics	Unsafe block density, idiomatic quality, compilation rate, latency

The C code corpus included standard C constructs, pointer-heavy logic, dynamic memory usage, macros, and header files. The open-source projects spanned domains such as system utilities, networking tools, and embedded device libraries.

B. Feature-Oriented Test Case Results

Targeted test cases were constructed to stress various language features. Each case was evaluated for output fidelity, semantic preservation, and idiomatic transformation.

TABLE II
Feature-Based Test Case Results

Test Case	C Feature	Expected Translation	Corrosion	C2Rust	Corrode
Dynamic Memory	malloc, free	Box, Vec, RAII	Pass	Pass	Pass
Pointer Arithmetic	arr + 5, *ptr++	Slice indexing	Partial	Fail	Partial
Nested Structs	Deep struct nesting	Traits, lifetimes	Pass	Partial	Pass
Null Pointer	Dereferencing	Option¡T¿	Pass	Partial	Partial
Macro Expansion	Recursive macros	Flattened modules	Pass	Partial	Pass

Interpretation:

Corrosion Engine successfully abstracted dynamic memory and null pointer logic into Rust-safe constructs using `Box`, `Vec`, and `Option`, thus eliminating the need for manual memory handling.

For pointer arithmetic and low-level memory manipulation, Corrosion offered a partial translation due to the limitations of static analysis alone; however, the inclusion of generative AI allowed the engine to infer safer alternatives in most cases.

Complex macro expansions, particularly those spanning multiple headers, were flattened into idiomatic Rust modules via preprocessing transformations and token unfolding.

C. Comparative Evaluation on Real-World Projects

In this phase, all three transpilers were evaluated on the same corpus of open-source projects. Key metrics were recorded to assess translation quality and system-level usability.

TABLE III
REAL-WORLD PROJECT EVALUATION METRICS

Metric	C2Rust	Corrode	Corrosion Engine
Unsafe Blocks / 1K LOC	23.4	9.7	1.8
Compilation Success (%)	75	82	92
Idiomaticity Score (1–5)	1.7	2.9	4.3
Latency (ms/LOC)	8.1	6.4	4.9

Metric Details:

- **Unsafe Blocks:** Number of `unsafe` tags per 1,000 lines of Rust code; lower is better.
- **Compilation Rate:** Percentage of transpiled codebases compiling without modification.
- **Idiomaticity:** Rated by five Rust developers based on readability and Rust best practices.
- **Latency:** Average time taken to translate each line of C code.

D. Observations and Analysis

Code Safety and Compliance: Corrosion Engine demonstrated a significant reduction in the use of unsafe constructs. Its hybrid model, combining static control-flow analysis and LLM-driven inference, enabled effective refactoring into memory-safe Rust code.

Idiomatic Quality: The engine produced Rust output that adhered to ownership, type inference, and community idioms. Manual fixes were minimal, improving developer productivity.

Translation Throughput: Corrosion's modular and parallelized translation pipeline contributed to lower latency and better scalability for large codebases.

Residual Challenges: Some edge cases—like inline assembly, pointer arithmetic, and platform-specific macros—still required developer review. These were either partially translated or flagged with `TODO` annotations.

VI. CONCLUSION

The Corrosion Engine demonstrates an exciting breakthrough in the automatic translation of legacy C code to Rust, addressing one of the most pressing issues in contemporary systems software—memory safety. Through a synergy of static code analysis and generative AI, the system effectively bridges syntactic translation and semantic correctness. Unlike rule-based transpilers that are biased towards preserving unsafe patterns, the Corrosion Engine generates idiomatic, ownership-compliant Rust code that abides by best practices and security guidelines. Employing a formal pipeline of preprocessing, segmentation, metadata enrichment, AI-boosted translation, and postprocessing, the Corrosion Engine enables accurate and context-aware transpilation. Exhaustive testing confirms the robustness, efficiency, and adaptability of the system on diverse C codebases. Key strengths include low use of unsafe blocks, high success rates in compilations, and improved code maintainability. Although the system seamlessly handles most C constructs, limitations remain in extensively complex pointer operations and macro-dense code that may require human intervention. Ongoing work will strive to increase the system's cross-language debugging support, AI context chaining on large-scale files, and broaden capabilities to assist C++ codebases. In a software domain more concerned with safety and correctness, the Corrosion Engine marks a significant breakthrough—providing organizations and developers with a scalable and intelligent means to adopt Rust, thereby evading the high risks and expenses of manual code rewriting.

VII. REFERENCES

1) M. Hicks, B. Campbell, and B. Foster, "C2Rust: Migrating Legacy C Code to Rust," arXiv preprint arXiv:1807.06706, 2018.
2) The MITRE Corporation, "Common Vulnerabilities and Exposures," CVE Details Database, 2023. [Online]. Available: https://www.cvedetails.com/
3) National Security Agency (NSA), "Software Memory Safety," Advisory Memorandum, 2023. [Online]. Available: https://media.defense.gov
4) Mozilla Research, "Introducing Servo," 2022. [Online]. Available: https://research.mozilla.org/servo/
5) Microsoft Research, "Project Verona: Researching memory safety in systems programming," 2022. [Online]. Available: https://www.microsoft.com/en-us/research/project/verona/
6) The Rust Programming Language, "The Rust Book," 2024. [Online]. Available: https://doc.rust-lang.org/book/
7) A. Goldfarb and B. Roberts, "Analyzing memory safety vulnerabilities in open-source software," IEEE Software, vol. 39, no. 3, pp. 18–25, 2022.
8) GitHub, "Copilot and its Limitations in Code Translation," GitHub Engineering Blog, 2023. [Online]. Available: https://github.blog/
9) Google, "Gemini: A Next-Gen Multimodal AI Model," Google DeepMind Blog, 2023. [Online]. Available: https://deepmind.google/technologies/gemini/
10) E. Pozniak, "Memory Safety in Systems Programming," Communications of the ACM, vol. 66, no. 6, pp. 28–36, 2023.
11) J. Regehr, "The Perils of C: Low-Level Programming and Memory Unsafety," IEEE Security & Privacy, vol. 20, no. 1, pp. 72–76, 2022.
12) A. Chlipala, "Verified Systems Programming with Rust," Lecture Notes in Computer Science, vol. 13100, Springer, 2023.
13) LLVM Foundation, "Clang: a C language family frontend for LLVM," Clang Project Documentation, 2024. [Online]. Available: https://clang.llvm.org/
14) Tree-sitter, "An Incremental Parsing System for Programming Tools," 2024. [Online]. Available: https://tree-sitter.github.io/
15) M. Allamanis et al., "A Survey of Machine Learning for Big Code and Naturalness," ACM Computing Surveys, vol. 51, no. 4, pp. 81:1–81:37, 2018.