Globant Code Fixer Agent

Globant Enterprise AI

November 2024

Contributors:

Martin Alejandro Bel - martinalejandro.bel@globant.com

José Lamas Ríos - jose.lamas@globant.com

Rodolfo Anibal Lobo Carrasco - rodolfo.lobo@globant.com

Juan Michelini - juan.michelini@globant.com

Gastón Milano - gaston.milano@globant.com

German Milano - german.milano@globant.com

Marcelo Pérez - ext-marcelo.perez@globant.com

Guillermo Pasquero - guillermo.pasquero@globant.com

Table of Contents

Abstract

In the context of the software industry, code repair or “bug fixing” is a complex problem that demands analysis, planning, and significant time investment. Naturally, this task plays a critical role in determining the speed and efficiency of engineering teams. AI systems endowed with significant agency, particularly those based on Large Language Models (LLMs), and LLM-based multi-agent systems have achieved state-of-art results in code-fixing tasks.

In this report, we propose a multi-agent system called **Globant Code Fixer Agent** (GCFA, or simply Code Fixer Agent) developed on top of **Globant Enterprise AI** (GEAI) platform. In GCFA, multiple agents collaborate in two stages: fault localization and bug fixing, to effectively address the problem at hand.

We’ve developed an agentic architecture that is intuitive for developers while prioritizing accuracy, speed, and cost-effectiveness. The system achieved an average of **2.65 minutes** per bug at an average cost below **$1 USD** per bug. Moreover, we achieved **48.33%** of resolved tickets in the SWE-Bench Lite benchmark. This positions our model as state-of-the-art when considering runtime, accuracy, and cost compared to the solutions available in the SWE community.

Overview

Context and Motivation

In today's fast-paced software development landscape, the demand for rapid and reliable code delivery is ever-increasing. Code issues not only slow down development but also impact software quality and user satisfaction. Traditional debugging and code review processes, while essential, can be time-consuming and require significant human expertise. This has paved the way for innovative solutions that leverage artificial intelligence to streamline and enhance these processes.

Agentic AI Systems, characterized by their decision-making capabilities and high agency levels, have emerged as powerful tools for addressing complex challenges across various domains. The agency level of these systems determines the degree to which they can independently manage processes and workflows, thereby minimizing human intervention.

In the context of the Software Development Life Cycle (SDLC), these systems can significantly boost efficiency. Particularly in the debugging process, they can aid in automating and optimizing it. Identifying patterns, analyzing code, and proposing fixes were all part of the Agentic AI System we wanted to build, reducing reliance on manual intervention, and freeing developers to focus on more strategic tasks such as system design. This shift not only accelerates the development cycle but also enhances code quality by minimizing human error.

Objectives

The primary objective of our AI Agentic System is to revolutionize the way code issues are handled in software development. By harnessing the power of AI, we aim to:

- **Implement automatic fault localization:** Develop advanced models and tools capable of automatically identifying the exact location of bugs within the codebase. Recognizing that localization is the crucial first step in successful bug resolution, the effort focuses on enhancing the precision and speed of error detection.

- **Enable automatic bug fixing:** Leverage foundational models and agentic workflows to automatically generate fixes for localized bugs. The goal is to produce effective and efficient code corrections that can be seamlessly integrated into real-world software environments without creating new bugs.

- **Optimize time and cost efficiency:** Ensure that both the automatic localization and fixing processes are optimized for time and cost, making them practical and viable solutions for businesses. This includes reducing the time developers spend on debugging and lowering the overall costs associated with software maintenance.

- **Validation using industry standards but grounded in our expertise:** Set clear, measurable standards for evaluating the effectiveness of our solutions, ensuring a high success rate while maintaining time constraints and avoiding excessive brute-force methods. Utilize industry-standard benchmarks like SWE-Bench-Lite as tools for objective validation, but being aware of their many limitations[1], and including our data science expertise to evaluate results.

- **Advance the SOTA in code fixing: A**pplied research is at the core of Globant's AI Studio, and it's deeply embedded in our data scientists. We want to position the Globant Code Fixer Agent, based on Globant Enterprise AI, at the forefront of automatic code correction technologies. By pushing the boundaries of what's currently possible, we aim to set new industry standards in software development practices.

1 SWE-Bench+: Enhanced Coding Benchmark for LLMs: https://arxiv.org/pdf/2410.06992

The Journey of Development

Genesis

Early this year, Google released a **demo** of Gemini 1.5 showcasing automatic code editing for a project with a substantial number of code lines. Significant advancements in context handling facilitated this.

Our initial, naive approach to bug fixing involved inserting as much code as possible into the context window and then applying various prompting techniques such as Chain-of-Thought (CoT) and Tree-of-Thoughts (ToT) to generate solutions. This method occasionally proposed acceptable solutions for real bugs. However, we quickly encountered several issues that set us back to the drawing board and kept us in the lab for months:

1. **Insufficient context window size:** Even with a 1.5 million token context window, it was inadequate for real-world codebases. Being aware of the “Needle in a Haystack” **test results** and the “Lost in the Middle” effect potentially getting in the way, we recognized that this wasn’t the correct approach.
2. **Performance and cost problems:** Zero-shot and few-shot prompting with such a large context led to performance bottlenecks and increased costs.
3. **Limited output context window:** While the input context window was large, the output context window was not proportionally expansive.
4. **Language variability limitations:** Gemini 1.5 did not perform well across the board when we started working with different programming languages.
5. **Challenges in code editing:** Editing code (still) is a significant challenge for LLMs, requiring specialized editing strategies.

Realizing that, despite impressive demos, the problem could not be solved with a single call to Gemini 1.5 in its current state, we initiated an intensive research effort.

Development Phases

Phase 0: Zero-Shot Tactic

As described before, the initial phase of development focused on exploring the potential of zero-shot bug-solving. This phase was primarily a proof-of-concept (PoC) aimed at demonstrating the value of providing a system with sufficient agency to solve bugs but without the complexity of orchestrating agentic workflows with tool usage. We were looking to determine whether the capabilities of existing LLMs could meet our expectations for bug-solving efficacy. At this stage, the goal wasn't to find the most efficient solution but rather to establish that a viable solution was achievable with the current technology. This foundational step was crucial in validating our hypothesis and setting the stage for more sophisticated development efforts. At this phase, it was essential to test different LLM model vendors. To achieve this, we leveraged our GEAI platform, accelerating our development process. In the following sections, we delve deeper into the description of this platform and how it fits into our solution, facilitating fundamental processes in both experimentation and development.

Phase 1: Divide & Conquer

Building on the insights gained from Phase 0, we embarked on Phase 1 with a deeper understanding of the components necessary for a successful AI Agentic System. We recognized that the most effective agent systems typically incorporate several key elements: task division, dedicated phases for reasoning, planning, execution, and evaluation, powerful tools, and feedback mechanisms[2]. With this knowledge, we set out to refine the system by clearly separating responsibilities.

Inspired by other innovative solutions such as the **RepairAgent**, we adopted a "divide and conquer" strategy. This approach involved splitting the bug-solving process into two distinct stages: (a) localization of the bug and (b) the actual fix. For us, this division meant we had to transition to a multi-agent AI system, where different agents had specialized expertise and access to codebase navigation tools (e.g., gathering project dependencies). By leveraging the strengths of specialized agents, we achieved a significant breakthrough, particularly in bug location, where we achieved more than 90% accuracy at the file level. Our E2E bug-fixing capability was around 20%, though.

This phase not only marked a quantum leap in our bug-solving capabilities but also established the baseline architecture for subsequent phases. The multi-agent approach provided a robust framework that allowed us to efficiently allocate tasks and optimize workflows, paving the way for further advancements in the system.

2 The Rise and Potential of Large Language Model Based Agents: A Survey: https://arxiv.org/pdf/2309.07864

Phase 2: In Search of Experts

During this phase, we embarked on an active “search of experts” exploration, further partitioning the fix stage and incorporating more expert agents and control checkpoints into the workflow. We approached this with an open exploration mindset, embracing the idea that almost any suggestion could have value and recognizing that while there might not be any magical solutions, there could be cleaner and more efficient ones.

We experimented with different LLMs and combinations thereof, mindful of the potential benefits of scaling laws to enhance our system’s performance with minimal effort. A key difference between Phases 1 and 2 was that in Phase 1, only the expert agents in the localization stage had access to tools, whereas, in Phase 2, agents primarily relied on specific prompting techniques to propose solutions, evaluate them, and edit the source files. We realized that this tool-free approach had limitations and would eventually plateau, but it was essential to understand these boundaries.

The step of editing the file with the proposed fix proved to be challenging. We tried a number of approaches, including (a) editing the entire source file, (b) creating a diff file, and (c) creating a Python script to edit the file accordingly. While exploring these methods, we encountered significant issues with accuracy, precision, and avoidance of unrelated changes. This challenge led us to develop a specialized tool capable of performing code edits, setting the foundation for the next phase of development. This tool-based approach not only addressed the limitations we encountered but also enhanced the system’s overall efficacy and robustness.

Phase 3: Current Solution

Our current solution implements a set of specialized agents dedicated to the fixing stage. These agents are responsible for proposing solutions, which are then evaluated by a critic agent acting as a “LLM-as-a-judge.” This approach is complemented by code editing tools designed to create and apply diffs, ensuring that any changes are precise and effective. Additionally, a retry mechanism is in place to handle instances where a fix fails, adding a layer of robustness to the workflow. This iterative and adaptive strategy has refined our system’s ability to resolve bugs efficiently and consistently, positioning it as a cutting-edge solution in the realm of AI-driven software development.

Key Decisions and Trade-offs

During the development of GCFA, our main driver was the desire to iterate rapidly and learn quickly—a principle that should be at the core of any data science team. This agile approach allowed us to adapt swiftly to new insights and challenges, ensuring that our architecture remained robust and responsive to evolving needs.

To validate our solution, we tested it across various teams within the company, gathering feedback not only on its performance but also on its usability. This user-centric evaluation was crucial in shaping a solution that aligns with real-world demands and enhances the user experience.

Several key decisions were pivotal in our development process:

- **Architecture type:** Selecting the right architecture was fundamental. We explored various configurations to determine which would offer the optimal balance between complexity and functionality.

- **Conversation patterns:** We considered different conversation patterns for agent interactions. Finding the right communication strategy was critical to ensuring proper collaboration among agents, enabling them to perform their tasks as best as possible.

- **Tool selection and utilization:** Another critical decision was which tools to integrate and how many each agent should use. The effectiveness of each development stage depended on equipping agents with the right tools to perform their specialized tasks.

- **Integration of AI Agents by using GEAI:** This approach was essential to avoid spending excessive time on inference, model hosting, and associated complexities. Regarding FinOps capabilities, GEAI enables you to set spending limits and execution time constraints, which in our case, ensured that our Code Fixer Agent remained cost-effective and efficient. By implementing these controls, we could manage resources effectively while maintaining high performance. GEAI ensures that data flows remain within the boundaries of Globant, adhering to strict data privacy and security policies. This containment is crucial for maintaining trust and responsibility in our AI applications, as it prevents sensitive information from leaving the organization's controlled environment. On top of GEAI, numerous agents are being developed for various specific industries. Particularly for the SDLC, where one key application is our Code Fixer Agent. By leveraging GEAI, these agents can be customized to meet the specific needs of different industries and processes, enhancing efficiency and effectiveness.

As with any AI Agentic system, the selection of a foundation model (particularly LLMs) was also part of the key decision-making process. We leveraged GEAI to test our agent strategy with several models from different providers, including Gemini 1.5, Gemini 1.5 Flash, Claude 3.5 Sonnet (v1 and v2), GPT-4, GPT-4o, o1-mini, o1-preview, and the LLaMA family of models. GEAI provided the much-needed flexibility, which saved us the burden and overhead of managing multiple models manually.

We found that our current system implementation yields the best results using Claude 3.5 Sonnet[3] for our specific use case.

Even with guidelines from the broader AI community, the solution space is just too vast, and it is impossible to traverse it all. So, we conducted extensive research to understand what might best suit our use case, examined what others were doing in the field, and last but not least, tested (and failed!) a lot. Each discarded approach provided information and brought us closer to our current solution.

We remain committed to this iterative, research-driven approach. As we continue to develop and enhance our system, applied research will remain a cornerstone of our strategy and, hopefully, allow us to evolve our solution to meet the challenges of AI-driven software development.

Globant Enterprise AI

Globant Enterprise AI (GEAI) serves as the umbrella under which Globant's strategy for the adoption, integration, and creation of AI Agents is established. It is the foundational platform that supports and accelerates the iterative development, evaluation, and deployment of Generative AI solutions across the organization and its clients.

One of GEAI's most important features is its multi-cloud capability. This allows you to install AI Agent solutions on any cloud provider—be it Google Cloud Platform (GCP), Amazon Web Services (AWS), Microsoft Azure, or even on-premises. This is an essential aspect because the platform we envision is designed to run in any enterprise environment, regardless of where their infrastructure is hosted. This flexibility ensures that our clients can adopt our platform without being constrained by their existing cloud setups.

3 Claude 3.5 Sonnet (2024-10-24 snapshot)

System Architecture and Design

High-Level Architecture

The Code Fixer Agent system is a two-stage, multi-agent system. The following describes the different stages of the multi-agent system that enable code repair.

1. Localization Stage

Task: Diagnose the bug.
Generate Asset: A localization report.

In this stage, a set of agents with access to tools navigates the codebase and searches for candidate files that may be causing the bug. These are the files that will be passed to the next stage as input. Once the candidates are found, a report that describes the main causes of the bug is generated. This report serves two purposes: it provides the user with insight into the model's functionality and acts as input for the fix stage.

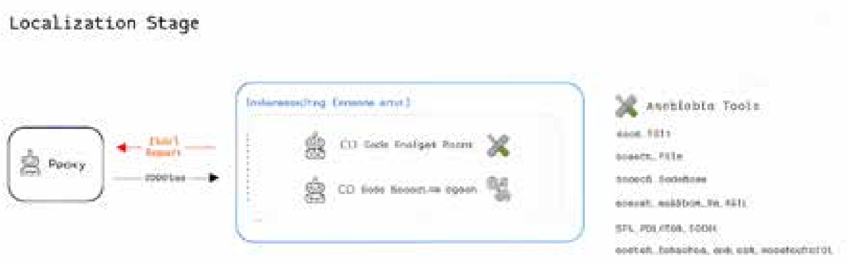


Figure 1. High-level Agentic architecture of automatic bug localization stage of Globant Code Fixer Agent.

Available tools:

- **read_file:** Reads the entire file and returns its contents as a list of lines.
- **search_file:** Searches for a specific file in a directory and returns the path.
- **search_methods_in_file:** Searches for methods content in the specified file.
- **get_related_files:** Get related files to the specified file. Useful when you find a suspect file to get more related files.
- **search_codebase:** Recursively searches for specific terms in all files within a directory, options to search_option “exact” or “fuzzy” or “all words” or “some word.”
- **search_code_in_file:** Searches for a specific term in a file and returns the matching tokens.
- **detect_language_and_get_dependencies:** Detect the programming language of the project and extract dependencies.

2. Fixing Stage

Task: Fix the bug using the localization report.
Generate Asset: A patch file containing the code fixes.

A robust configuration of three specialized agents is employed at this stage, working collaboratively to address the identified issues. The agents are defined as follows:

- **Architect Agent:** Generates solution proposals based on the bug localization report and contextual understanding of the codebase. This agent utilizes foundational models to draft fixes aligned with coding best practices and project specifications.
- **Editor Agent:** Implements the proposed solution by modifying the source code. Equipped with tool integrations, the Editor Agent ensures high accuracy in executing the Architect Agent's proposal, addressing both syntactical and logical correctness.
- **Critic Agent:** Evaluates the implemented solution for correctness and alignment with the intended fix. If the solution is deemed unsatisfactory, it triggers a retry loop, prompting the agents to refine the fix.

The retry mechanism is a cornerstone of the system, enabling iterative improvements until an acceptable solution is achieved. Each retry leverages feedback from the Critic Agent to guide the Architect and Editor Agents in refining their outputs.

Such a multi-agent, multi-stage approach ensures efficient and accurate bug detection and resolution, streamlining the software development process and enhancing code quality.

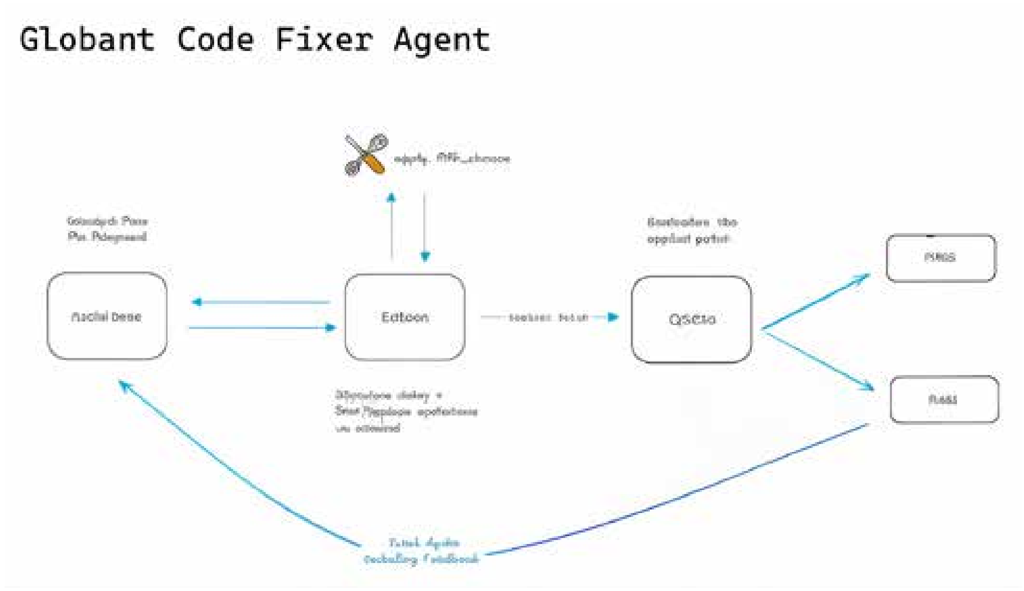


Figure 2. High-level Agentic architecture of the fixing stage of Globant Code Fixer Agent.

3. Flow Engineering

Code Fixer's performance hinges on its flow engineering—the orchestration of data and task execution among agents—and the strategic use of tools designed to augment the agents' capabilities. Key aspects include:

- **Return values design:** Tools are configured to provide structured and actionable feedback to the agents, enabling them to make informed decisions.
- **LLM-enhanced tooling:** Traditional software tools are seamlessly integrated with LLMs to optimize the localization and fixing process, ensuring scalability and precision.

Designed with a modular and extensible structure, the architecture allows for the seamless integration of new tools, techniques, or agents to tackle evolving challenges in software development.

Core Technologies

The implementation of GCFA relies on foundation models and robust frameworks to ensure high performance and adaptability across a range of tasks. By integrating SOTA technologies, we have developed a system that not only meets the demands of complex problem-solving but also provides an intuitive user experience.

Foundation Models

As mentioned before, we've tested several foundation models in different combinations. These are an essential part of the research and development process. Here's a non-comprehensive list of the ones we evaluated:

- **Claude Sonnet 3.5**: a SOTA language model available in two versions, is designed for efficient text understanding and generation, enabling nuanced comprehension and communication. It has been utilized for most of the agents described in **System Architecture and Design** since it has shown the best coding capabilities given our workflow.
- **GPT-4o**, **o1-preview**, **Gemini 1.5**, **Llama-3.1**: These models were incorporated at various stages of our research. Each model was selected and evaluated based on its specific strengths, contributing to different aspects of language processing, code analysis, and iterative problem-solving. Some of them were tested out of curiosity about their performance, knowing that they may not be ready for a productive environment. Currently, we use GPT-4o as a fallback for Sonnet. This setup is designed to handle situations where Sonnet may be unavailable, such as when a rate limit is reached, thereby enhancing system robustness at the cost of some performance.
- **GPT-4o mini**: Used primarily in GCFA's CLI, the model excels in quick and precise code modifications, supporting the system's debugging and code refinement processes.

Frameworks and Libraries

To support the diverse functionalities of our system, we have integrated several awesome frameworks and libraries. The ones to highlight are the following:

- **AutoGen**: A framework that facilitates automated workflows, Autogen enables seamless execution of tasks such as testing, debugging, and report generation with minimal manual intervention. It emerged as our top pick among several possible agent frameworks. It allows us to create conversation patterns between agents fairly easily. We're currently using v0.2 but are planning to migrate to v0.4 once it's stable enough.

- **Tree-sitter**: Since we're dealing with codebases, we needed to create tools for our agents that could create and traverse ASTs—tree-sitter was our top-of-mind for this. As a parser generator tool and incremental parsing library, it is essential for code analysis and manipulation. Tree-sitter provides the structural analysis needed to understand and process source code efficiently, forming a backbone for the system's code-related tasks.

- **Chameleon**: A **GeneXus** library of white-label, highly customizable, and reusable web components, Chameleon serves as the foundation for visualization and end-user interaction. It supports the creation of a dynamic and adaptable user interface, facilitating a more intuitive and accessible experience for users interacting with AI agents.

This combination of foundation models and frameworks underpins the system's ability to deliver high performance and adaptability, ensuring that it remains at the forefront of AI-driven innovation.

Results

SWE-bench

SWE-bench is a benchmark designed to evaluate codebase problems using verifiable in-repo unit tests. The full dataset contains 2,294 issue-commit pairs across 12 Python repositories, offering diverse and challenging evaluation tasks for long-term assessments of language models (LMs). However, the complexity and computational demands of the SWE-bench have posed challenges for systems aiming for short-term progress. To address this, **SWE-bench Lite** was introduced as a streamlined, canonical subset of 300 instances focused on functional bug fixes. This subset maintains the diversity and distribution of repositories in the original dataset, covering 11 of the 12 repositories.

An additional 23 development instances were curated to aid active development on SWE-bench tasks. Our results are being evaluated using SWE-bench Lite,
consistently achieving outcomes that position us among the top-performing or SOTA solutions globally, with **48.33% of solved tickets.**

The submission of results in SWE-bench requires the publication of the agents' internal states to verify the logical process of ticket resolution through logs and trajectories.
For more information, you can refer to the results published by competitors on the **SWE benchmark website**.

Real-world Applications and Client Integration

Our work extends beyond achieving high performance on benchmarks; we are actively deploying our solution in real-world client scenarios. The **localization phase** is extremely important because it guides our teams to the precise areas in the codebase where issues reside. Accurate fault localization empowers developers to focus their efforts effectively, significantly reducing the time spent on debugging and enhancing overall productivity.

We have integrated our system with industry-standard tools like **Jira**, incorporating a preliminary phase of problem understanding before initiating localization. This integration allows us to extract contextual information from issue reports, enabling a more informed and targeted localization process. By understanding the problem's context through Jira tickets, our solution can provide more accurate and relevant localization results, streamlining the transition from issue identification to resolution.

We believe that **automatic fixing** will transform many workflows within the software development lifecycle. To accommodate diverse user needs and integrate seamlessly into existing processes, we have developed three modes of interaction with our code fixer:

1) Batch Mode: Used primarily for running benchmarks, this mode allows for the automated processing of multiple fixes without user intervention. It's ideal for large-scale codebases and continuous integration pipelines where efficiency is paramount.

2) Chat Command Line Interface: This mode facilitates integration with tools like **Copilot** and other IDE assistants. Developers can interact with the code fixer through command-line prompts within their development environment, making it a convenient option for those who prefer a more hands-on approach.

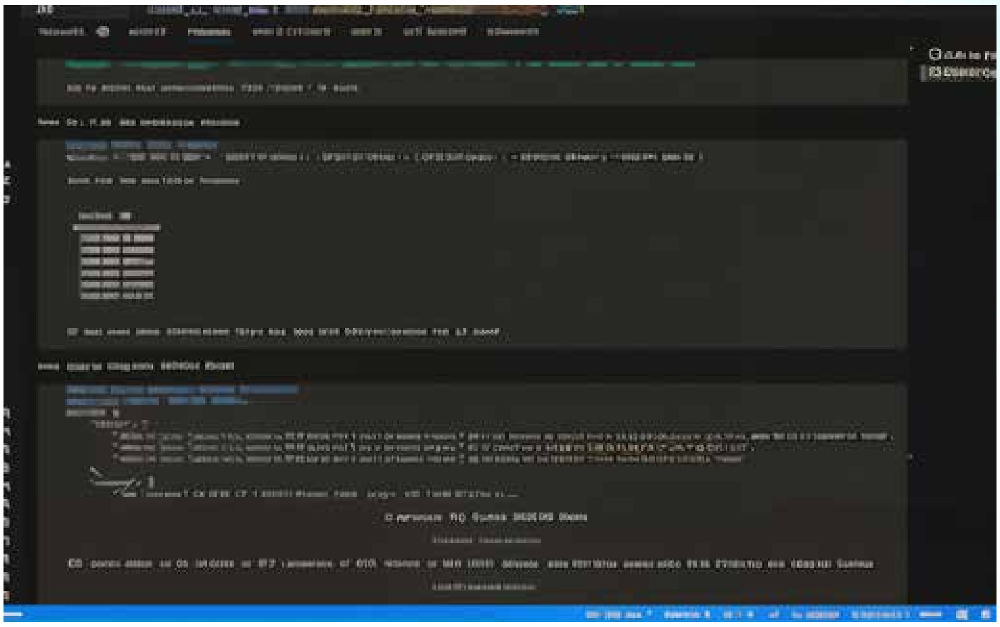


Figure 3. Screenshot of GCFA's chat command line interface.

3) Web Interface: We have created a web-based interface that enables users to monitor the progress of code fixing in real time. This interface combines a visual client with conversational capabilities, allowing for iterative interactions and providing a transparent view of the fixing process.

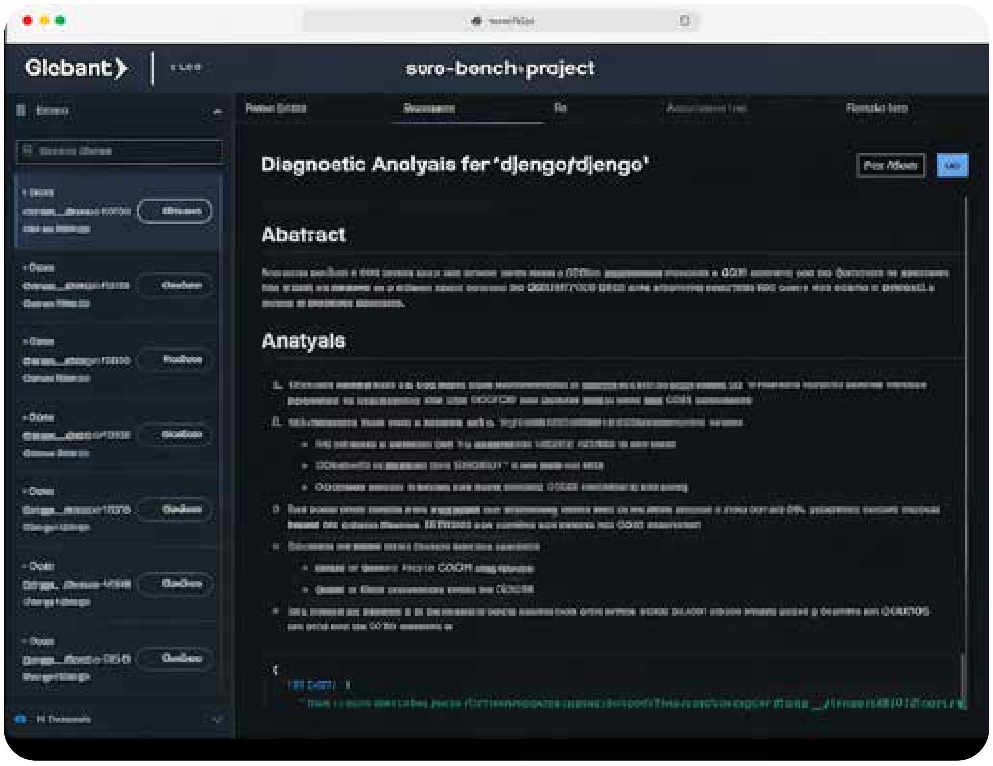


Figure 4. Screenshot of GCFA's web interface.

- By offering multiple interaction methods, we ensure that our solution is flexible and adaptable to various development workflows. Whether through automated batch processing, command-line interactions, or a user-friendly web interface, our code fixer integrates seamlessly into the developer's toolkit. This versatility not only enhances productivity but also facilitates efficient bug resolution in real-world applications, affirming our commitment to delivering practical and impactful agents for our clients.

Challenges and Lessons Learned

Throughout the development of GCFA, we faced several significant challenges:

1. **Context window limitations:** Managing various context windows was a considerable hurdle. Despite advances that allow for larger context windows—even those exceeding a million tokens—they quickly become insufficient for real-world projects with extensive codebases. Utilizing these large contexts can be inefficient and costly, particularly concerning response times, even after implementing optimizations like prompt caching.

2. **Necessity of an agentic solution:** Ensuring the need for an agentic solution, where iterative loops take precedence over simple few-shot prompting, became an area of continuous evolution. Relying solely on few-shot prompting proved inadequate for complex tasks. Developing a loop-based, agent-driven approach allowed for more nuanced and effective problem-solving but required significant research and refinement.

3. **Integrating diverse agent frameworks with various models:** We encountered challenges in harmonizing different agent frameworks with the specific requirements of each language model. Some models, like o1, do not support system prompts, while others have limitations regarding message role sequences or tool usage. This required meticulous adjustments and customizations to ensure compatibility and optimal performance across different models and frameworks.

4. **Navigating model guardrails:** Dealing with the guardrails implemented in certain models was also a challenge. These built-in safety features sometimes restricted the models from performing desired actions, requiring us to find workarounds without compromising ethical guidelines or model integrity. This involved careful prompt engineering and, at times, selecting alternative models better suited to our needs.

5. **Generating successful code edits:** Facilitating successful edits to code files proved difficult. In their current state, large language models (LLMs) without specialized tools struggle to perform precise code editing tasks. The challenge underscored the need for strategies and tools to enhance LLMs' capabilities in code manipulation, such as incorporating external code editors or developing specialized editing prompts.

6. **Cross-language support:** Another significant challenge was developing a solution that is cross-language compatible. Our goal was not only to target Python but to support a wide range of programming languages. The solution we achieved supports numerous languages, including:

   - Primary Languages Tested: Python, Java, C#, JavaScript, TypeScript.

   - Additional Supported Languages: Delphi, ELisp, Go, Guile, Haskell, Julia, Kotlin, Lua, OCaml, Odin, Perl, R, Ruby, Rust, Swift.

7. **Tooling:** Developing tools to handle the syntactic and semantic nuances of diverse programming languages required significant effort. Our models and frameworks needed to parse, analyze, and modify code across various paradigms. While testing focused on Python, Java, C#, JavaScript, and TypeScript, expanding support involved addressing challenges such as language-specific syntax, standard libraries and frameworks, and adapting tooling and compilation pipelines for different languages.

These challenges underscored the complexity of developing an effective automatic code-fixing solution that is versatile and scalable across multiple programming languages.

Roadmap

Current Lines of Work

Our ongoing efforts are focused on advancing the capabilities of the Code Fixer Agent. These initiatives aim to optimize collaboration, improve accuracy, and enhance the overall user experience through the following key areas:

- **Test Agents:** We are developing new stages and workflows that include environment (i.e., sandbox) management and unit test creation and execution. These enhancements enable the Code Fixer Agent to validate its candidate fixes more effectively, thereby reducing reliance on the LLM-as-a-judge approach.
- **Evaluation metrics:** We analyze, experiment with, and develop metrics to optimize the flow of information. We are exploring combinations of qualitative and quantitative metrics to validate and refine the internal processes within the multi-agent workflow.
- **Accessibility and ease of use:** We aim to improve the tool's speed, interactions, and feedback capabilities for our users, with a primary focus on enhancing the user experience.

Research Opportunities

Besides our current endeavors, we have numerous open lines of research that work on the problem of fault localization and automatic code fixing. One important area is the work we are conducting to develop proprietary benchmarks using real-world cases. We want to establish controlled environments where developers can engage in coding exercises that mimic actual client scenarios. This approach allows us to test in authentic settings, evaluating our architectures and tools in environments that closely resemble those of our clients, enhancing the applicability of our AI Agents. Having our own benchmarks ensures that our AI Agents are grounded in practical applicability rather than solely relying on standard benchmarks.

Another crucial area of research is exploring and experimenting with various agentic system architectures on different types of bugs, including one-line, multi-line, and multi-file errors. We recognize that the complexity and nature of bugs can vary significantly, and a system (or part of it) that performs well on one type may not be as effective on another. By testing diverse solutions across these different bug categories, we aim to:

- **Understand the solution's strengths and weaknesses:** Identify which architectures are best suited for specific types of bugs.
- **Optimize GCFA:** Tailor our agentic system to be more effective across a broader spectrum of issues.
- **Develop specialized strategies:** Create or adapt our system to address the unique challenges posed by different bug complexities.

These efforts aim to enhance the versatility and effectiveness of our Code Fixer Agent. By continuously expanding our understanding and capabilities, we are better positioned to tackle the diverse and complex challenges that arise in modern software development.

Long-Term Vision

Our future vision is that a **diverse array of agents** will undoubtedly collaborate on tedious and time-consuming tasks across various phases of the SDLC. Currently, we are making significant advances in areas such as requirements analysis, automatic bug reporting, localization and fixing, test generation, code generation, and finally, automatic UX generation. Our main goal is to create an SDLC suite for our users that enables them to work faster and with fewer errors, providing the appropriate environment to maximize their productivity while ensuring an enjoyable experience.

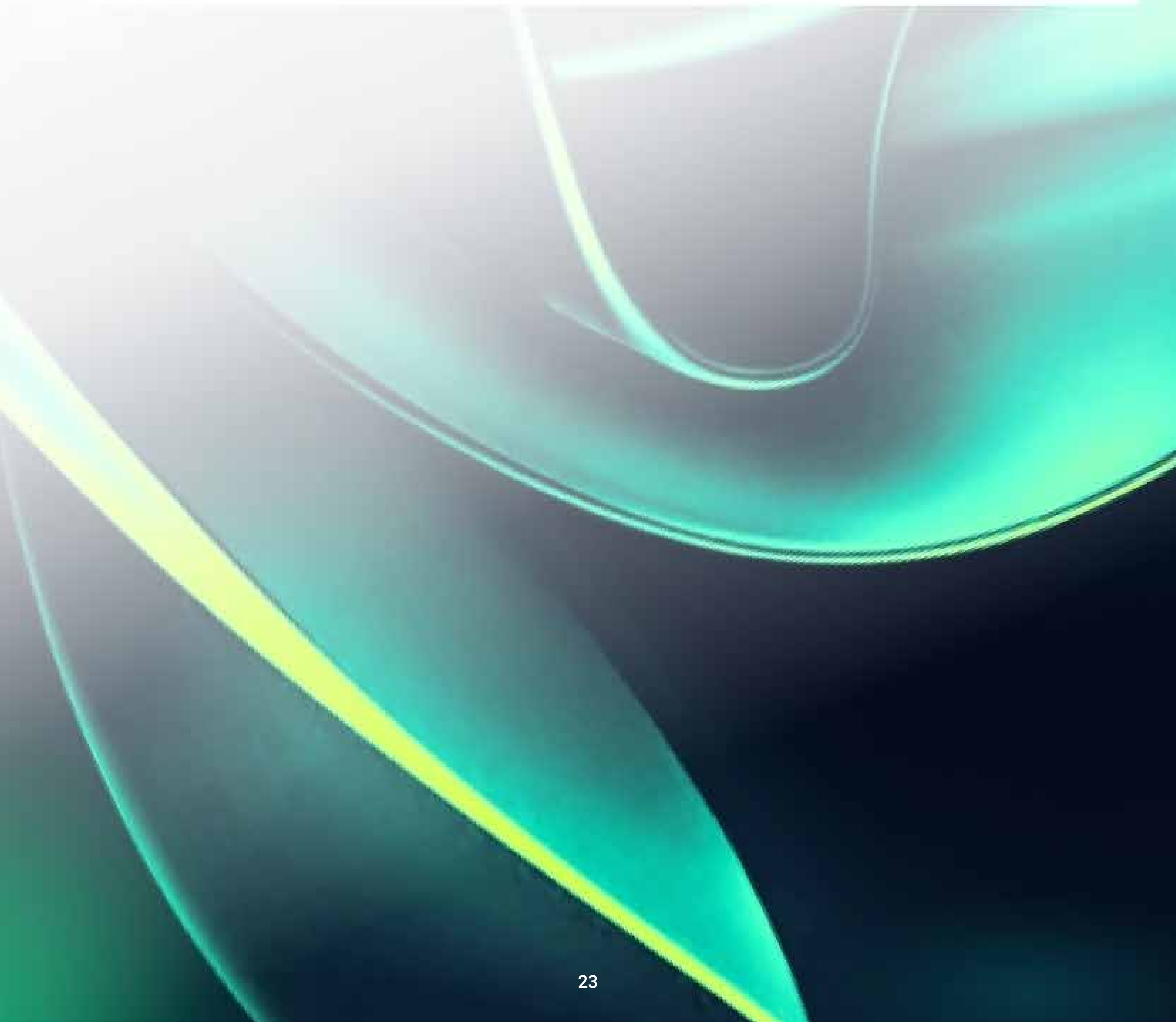
We firmly believe that by combining these diverse agents to work on a common knowledge base, companies' digital assets will finally have the longevity and adaptability to endure over time, evolving alongside technological advancements. By accelerating the most tedious maintenance tasks and streamlining others, such as code generation, we can explore and discover novel new agents and experiences across all industries.

We envision agents that remove accidental complexities in software development across all sectors. This will not only simplify the development process but also empower businesses to focus on innovation and deliver exceptional value to their customers. By harnessing the collective capabilities of specialized agents, we aim to revolutionize the software development lifecycle, making it more efficient, adaptable, and future-proof.

Acknowledgements

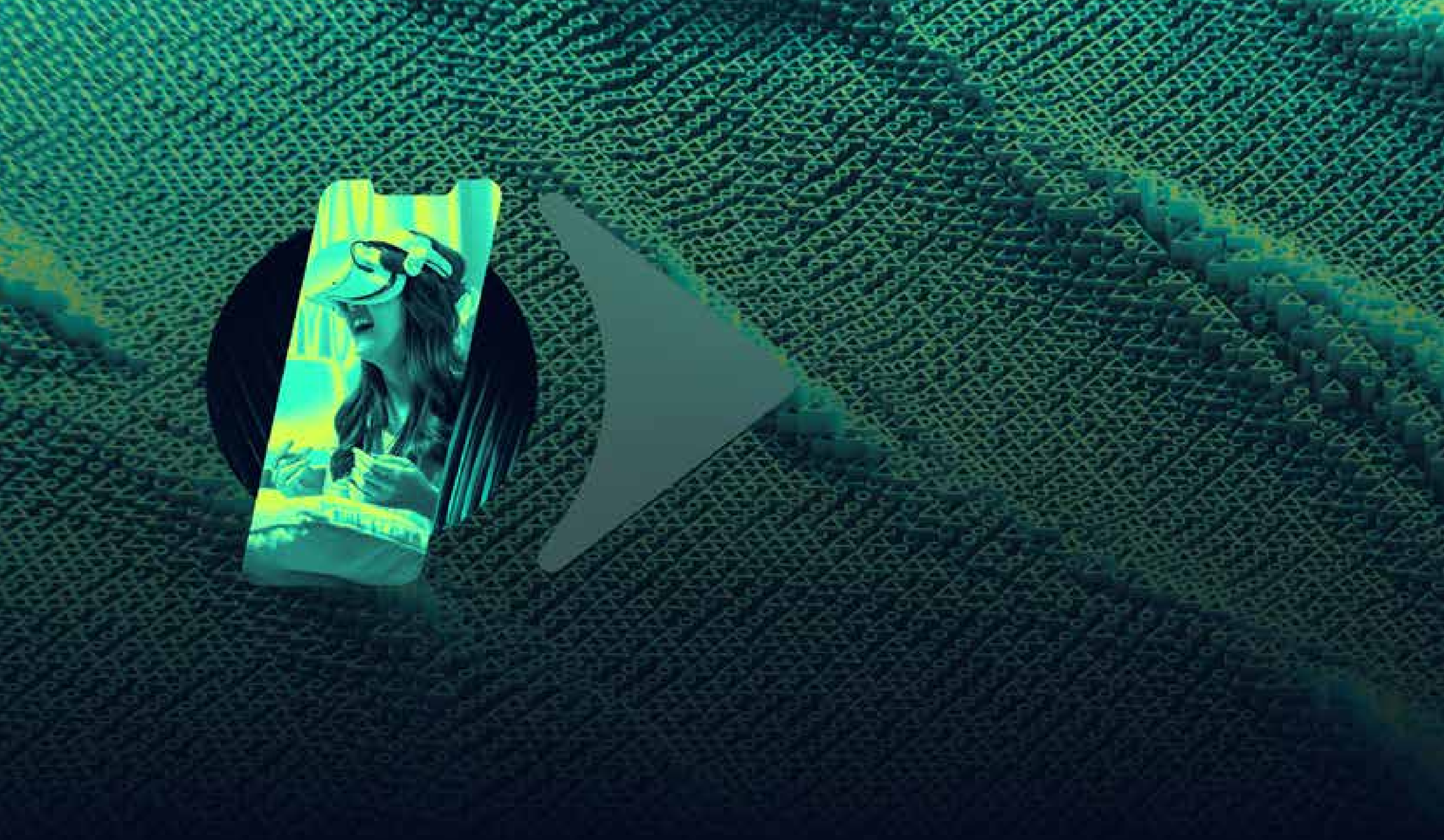
We draw inspiration from various agents, whether open source or accompanied by published research. Here are a few notable mentions:

- **RepairAgent**: Following their Agent design, we adopted a multi-stage strategy (localization and fix) and were inspired by some tools related to code search.
- **Aider**: We followed the research shared in the community in relation to editing **benchmarks** and decided to try how dividing the roles of “architect” and “editor” worked in GCFA.



Glossary

Terminology	**Definition**
AI	Artificial Intelligence
LLM	Large Language Model
GCFA	Globant Code Fixer Agent
GEAI	Globant Enterprise AI
SDLC	Software Development Life Cycle
CoT	Chain-of-Thought
ToT	Tree-of-Thought
PoC	Proof-of-Concept
E2E	End to End
SOTA	State-of-the-Art
AST	Abstract Syntax Tree



About **Globant**

At Globant, we create the digitally-native products that people love. We bridge the gap between businesses and consumers through technology and creativity, leveraging our expertise in AI. We dare to digitally transform organizations and strive to delight their customers.

- We have more than 29,900 employees and are present in 34 countries across 5 continents, working for companies like Google, Electronic Arts, and Santander, among others.
- We were named a Worldwide Leader in AI Services (2023) and a Worldwide Leader in CX Improvement Services (2020) by IDC MarketScape report.
- We are the fastest-growing IT brand and the 5th strongest IT brand globally (2024), according to Brand Finance.
- We were featured as a business case study at Harvard, MIT, and Stanford.
- We are active members of The Green Software Foundation (GSF) and the Cybersecurity Tech Accord.

For more information, visit **www.globant.com**