KPI Eip

EIP Support Guide Technical Track

Introduction

Welcome to the guide designed to support you throughout the Technical Track of the Epitech Innovative Project (EIP). This track allows you to explore and integrate advanced technologies, tackle complex technical challenges, and demonstrate in depth expertise in a specific technological field.

Mandatory Objectives

Evaluating and Integrating New Technologies

Regular Technology Watch

As part of our commitment to staying up to date with modern C++ standards, we continuously integrated and experimented with recent language features relevant to the JS-CMP project.

`std::variant` / `std::optional` (C++17)

These features are at the core of our project architecture. `std::variant` allows us to represent JavaScript's dynamic typing within C++ by storing values of multiple possible types in a single variable. Combined with `std::optional`, we handle undefined or nullable values gracefully, closely mimicking JavaScript semantics.

`<stacktrace>` (C++23)

Used to provide meaningful runtime error reporting, this feature is central to our exception system. It helps us link C++ exceptions back to their original JavaScript source locations using source maps, improving debugging and developer feedback.

By following the evolution of the C++ standard and consulting technical blogs, proposal discussions (e.g., on wg21.link), and library documentation, we ensured that JS-CMP stays modern, robust, and aligned with best practices.

Research Documentation

Benchmarks

We conducted benchmarks to compare the performance of different C++ constructs and data structures relevant to our transpilation engine:

- `std::string` vs custom **Rope** implementation: to evaluate efficient string concatenation strategies for large, mutable JavaScript strings.
- `std::visit`: assessed in the context of our variant based AST and runtime type dispatching.

Node Reference

Task	Attempt #1	Attempt #2	Attempt #3	Attempt #4	Attempt #5		Average
basic for loop bool / int add (test-bool.js)	0,123	0,032	0,031	0,031	0,03		0,0494
basic for loop string add (test-string.js)	0,044	0,033	0,031	0,029	0,029		0,0332

POC version

Task	Attempt #1	Attempt #2	Attempt #3	Attempt #4	Attempt #5		Average	vs node (%)
basic for loop bool / int add (test-bool.js)	0,361	0,168	0,167	0,169	0,166		0,2062	-317,41
basic for loop string add (test-string.js)	1,249	1,052	1,039	1,046	1,042		1,0856	-3169,88

Profiling	JS::Any::operator+	JS::Any::operator++	JS::Any::operator<	other
basic for loop bool / int add (test-bool.js)	82%	15%	2,10%	1%
basic for loop string add (test-string.js)	95,80%	1%	3,40%	0%

Rope data structure instead of string

Task	Attempt #1	Attempt #2	Attempt #3	Attempt #4	Attempt #5		Average	vs node (%)	vs POC (%)
basic for loop string add (test-string.js)	0,017	0,01	0,007	0,011	0,006		0,0102	69,28	95,05

Profiling	JS::Any::operator+	shared_ptr	JS::Any::operator<	other
basic for loop bool / int add (test-bool.js)	64%	12%	4,00%	20%

Without std::visit operators

Task	Attempt #1	Attempt #2	Attempt #3	Attempt #4	Attempt #5		Average	vs node (%)	vs POC (%)
basic for loop bool / int add (test-bool.js)	0,05	0,015	0,013	0,01	0,009		0,0194	60,73	90,59
basic for loop string add (test-string.js)	1,44	1,184	1,293	1,274	1,249		1,288	-3779,52	-18,64

Profiling	JS::Any::operator+	JS::Any::operator++	JS::Any::operator<	memmove	other
basic for loop bool / int add (test-bool.js)	48%	20%	15,50%	-	17%
basic for loop string add (test-string.js)	67,60%	1%	1,00%	30,40%	0%

Without std::visit everywhere and Without Types

Task	Attempt #1	Attempt #2	Attempt #3	Attempt #4	Attempt #5		Average	vs node (%)	vs POC (%)
basic for loop string add (test-string.js)	0,721	0,625	0,66	0,62	0,634		0,652	-1863,86	39,94

With char * and not std::string

Task	Attempt #1	Attempt #2	Attempt #3	Attempt #4	Attempt #5		Average	vs node (%)	vs POC (%)
basic for loop string add (test-string.js)	1,73	1,56	1,61	1,61	1,79		1,659	-4896,99	-52,82

With custom string class

Task	Attempt #1	Attempt #2	Attempt #3	Attempt #4	Attempt #5		Average	vs node (%)	vs POC (%)
basic for loop string add (test-string.js)	2,56	2,7	2,637	2,601	2,594		2,6184	-7786,75	-141,19

Key Articles and References Consulted

Articles:

- How is async/await transpiled to ES5?
- Using ES6 generators and `yield` for async workflows
- Ropes: an Alternative to Strings - Paper
- NodeJS documentation
- ECMA script specification
- V8 documentation
- Profiler documentation

Github:

- IdOp: custom operator in C++
- Are We Fast Yet? – JS performance across runtimes
- JS tester

Youtube:

- Modern C++ Techniques
- Behind the scenes of JS engines
- LLVM optimization insights
- Web browser from scratch devlogs
- JS Engine explained
- Fast JS
- Advanced C++ technique
- C++ variant

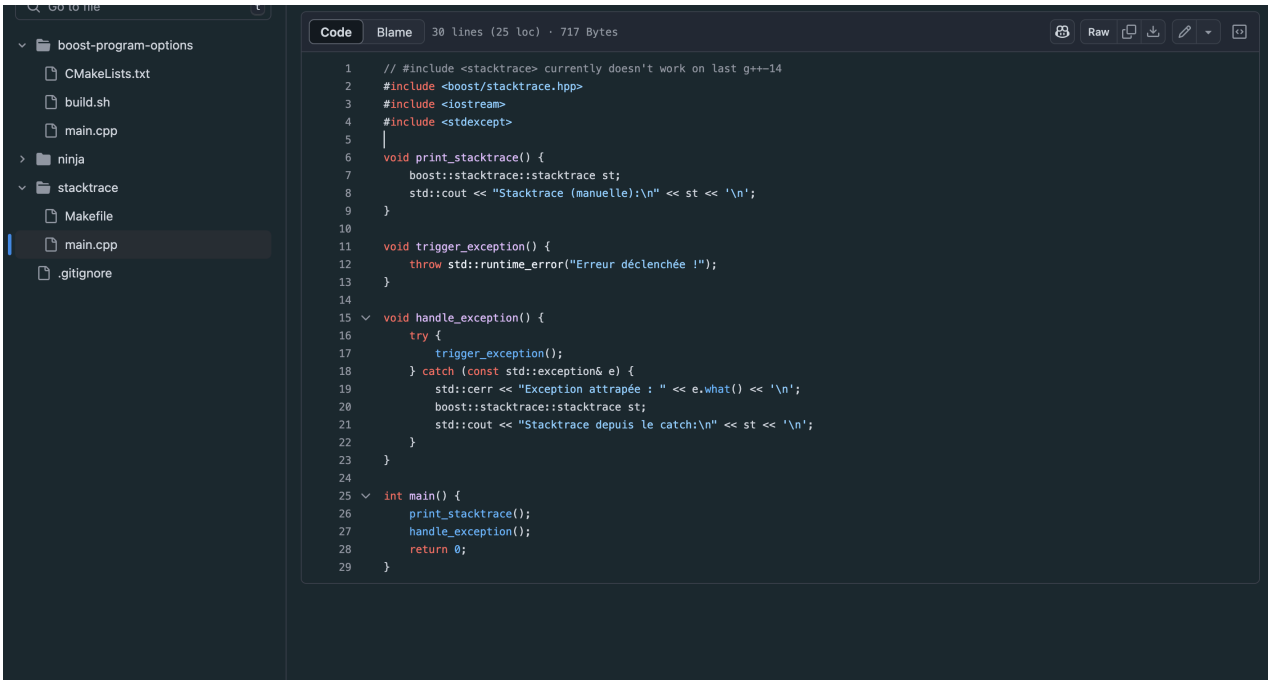
These resources helped us understand how modern JS runtimes work internally, how asynchronous control flow can be lowered to simpler constructs, and how C++ can be used effectively to replicate similar patterns.

Practical Application of New Technologies

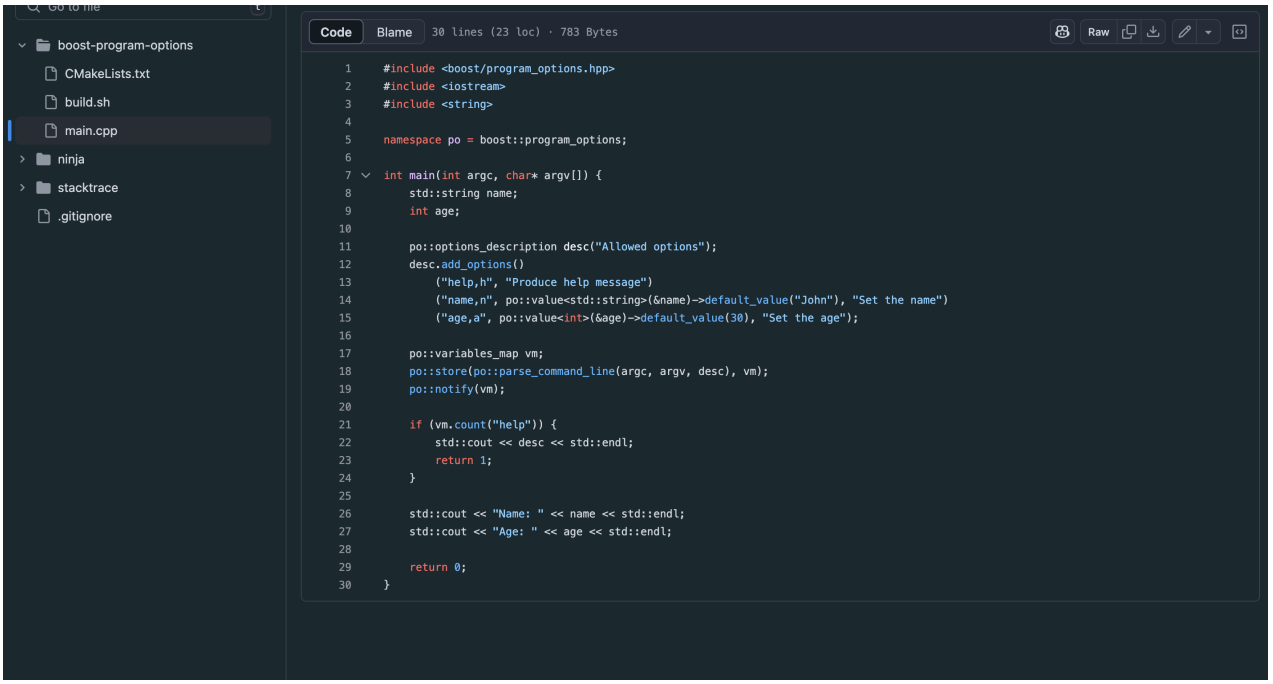
Throughout the JS-CMP project, we regularly explored and tested new technologies by developing various Proofs of Concept (POCs) and experimental modules. These experiments allowed us to evaluate libraries, compiler features, and language capabilities relevant to our transpiler.

Examples of Technologies Explored

- **Build Ninja** – Tested as a lightweight and fast alternative build system.
- `<stacktrace>` – Explored C++23's built in stack trace capabilities for improving debug information during transpilation.



- **ECMA Compliant Object Model** – Designed a prototype for an object structure closely matching JavaScript's dynamic object behavior in C++. This experiment can be see here
- **Boost ArgPars** – Experimented with Boost's argument parsing utilities to handle CLI interactions for the transpiler.



All experiments and documentation have been maintained in a private repository:

Private Repository: JS-CMP/POC

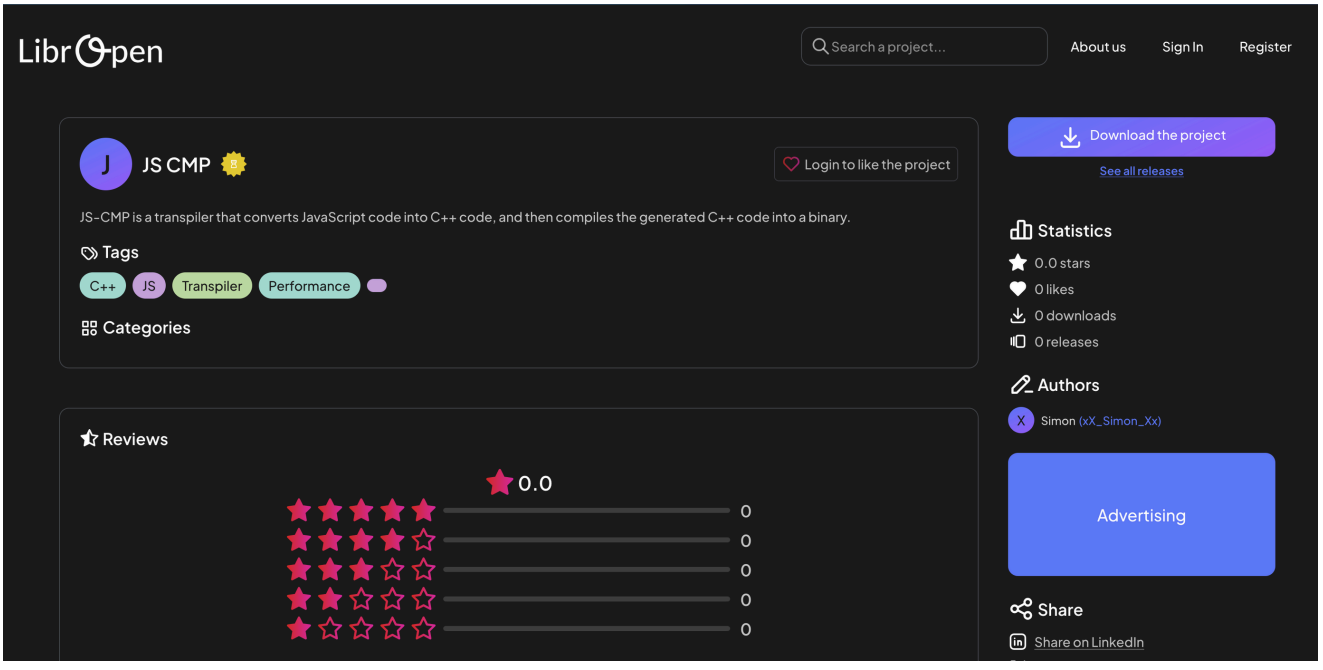
These POCs demonstrate our hands on engagement with modern tools and techniques, and our ability to critically assess their relevance for the JS-CMP toolchain.

Engagement in Tech Communities

LibrOpen is a solution project made by epitech student for the EIP to find and list all the open source project.

We have setup our project on the website to be found by developpers and users around the world.

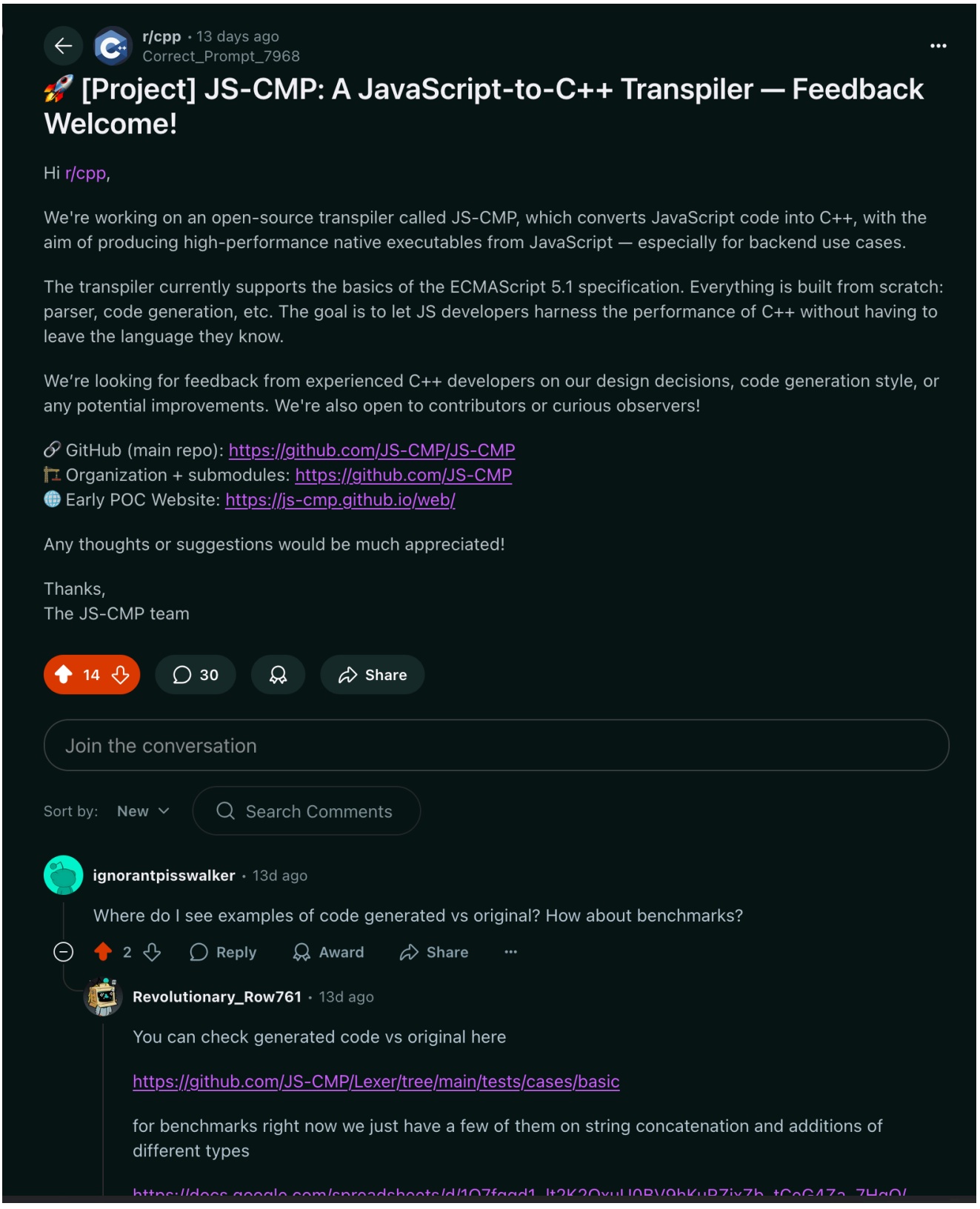
The project can be seen here.



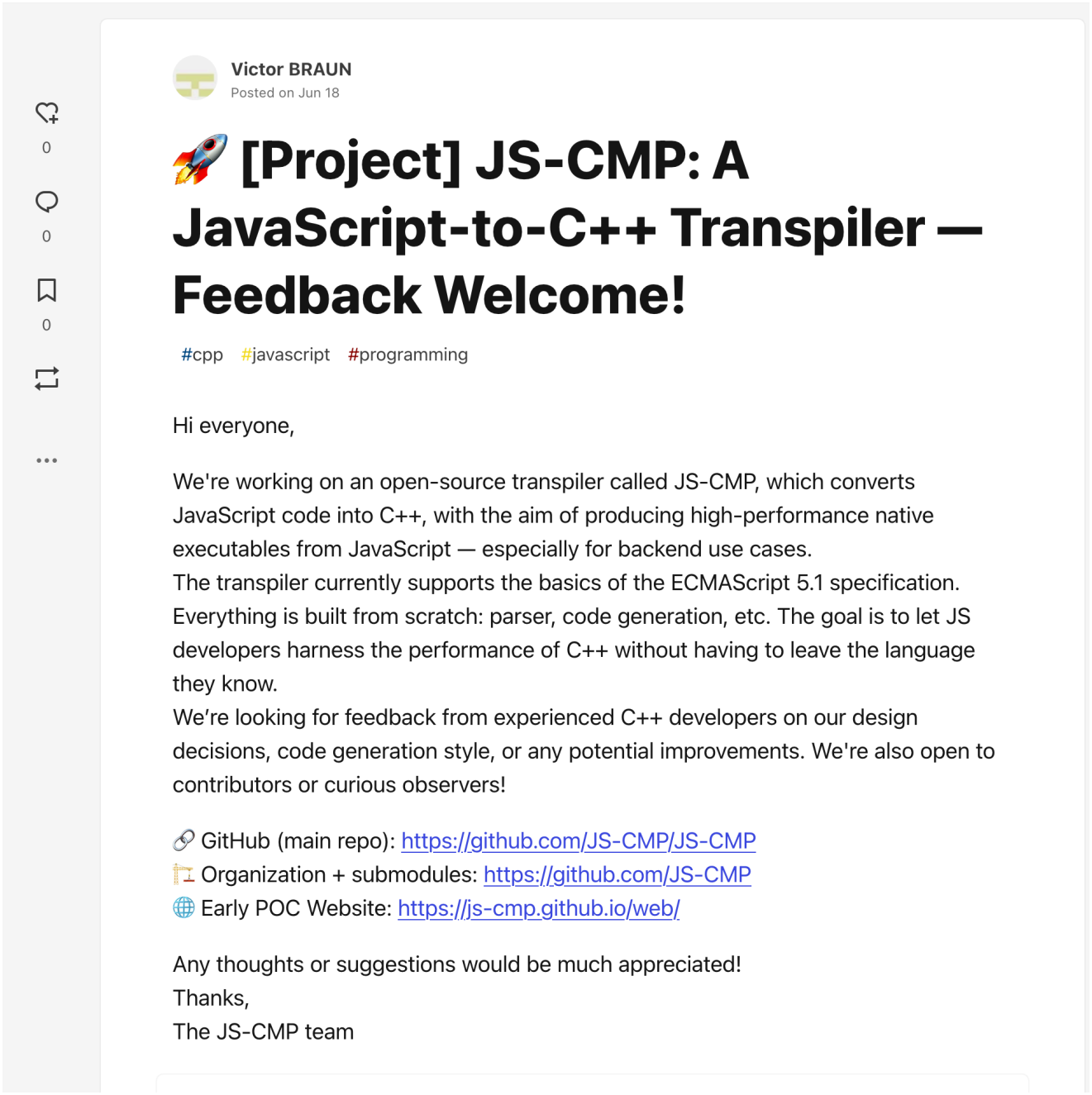
We have also post our project on the c++ discord. And we receive question and remark about our project.

The discussion can be seen here

we have made a reddit post. we did get a lot's of comment. You can see it here



And we have also made a dev.to post but we didn't get any comment on it. It can be seen here



Victor BRAUN
Posted on Jun 18

🚀 [Project] JS-CMP: A JavaScript-to-C++ Transpiler — Feedback Welcome!

#cpp #javascript #programming

Hi everyone,

We're working on an open-source transpiler called JS-CMP, which converts JavaScript code into C++, with the aim of producing high-performance native executables from JavaScript — especially for backend use cases.
The transpiler currently supports the basics of the ECMAScript 5.1 specification. Everything is built from scratch: parser, code generation, etc. The goal is to let JS developers harness the performance of C++ without having to leave the language they know.
We're looking for feedback from experienced C++ developers on our design decisions, code generation style, or any potential improvements. We're also open to contributors or curious observers!

🔗 GitHub (main repo): https://github.com/JS-CMP/JS-CMP
🏗️ Organization + submodules: https://github.com/JS-CMP
🌐 Early POC Website: https://js-cmp.github.io/web/

Any thoughts or suggestions would be much appreciated!
Thanks,
The JS-CMP team

Complementary Objectives

B - Developing Community Contributions

Clear Licensing Position

We chose the **GNU General Public License v3 (GPLv3)** for JS-CMP to ensure that the project remains free and open-source, and that any derivative works also stay open and accessible to the community. The GPLv3 enforces a strong copyleft model, meaning any modified versions must also be released under the same license. This fosters a collaborative environment and encourages contributions from developers who share similar free software values.

Why GPLv3?

Our goal with JS-CMP is not only to provide a powerful JavaScript to C++ transpiler, but also to protect the freedom of its users. By choosing GPLv3, we guarantee that improvements and

extensions to the project remain available to everyone, which aligns with our commitment to open development and long-term transparency.

Advantages

- Strong user freedom protection
- Ensures derivative works remain open source (strong copyleft)
- Widely used and supported by the open source community

Disadvantages

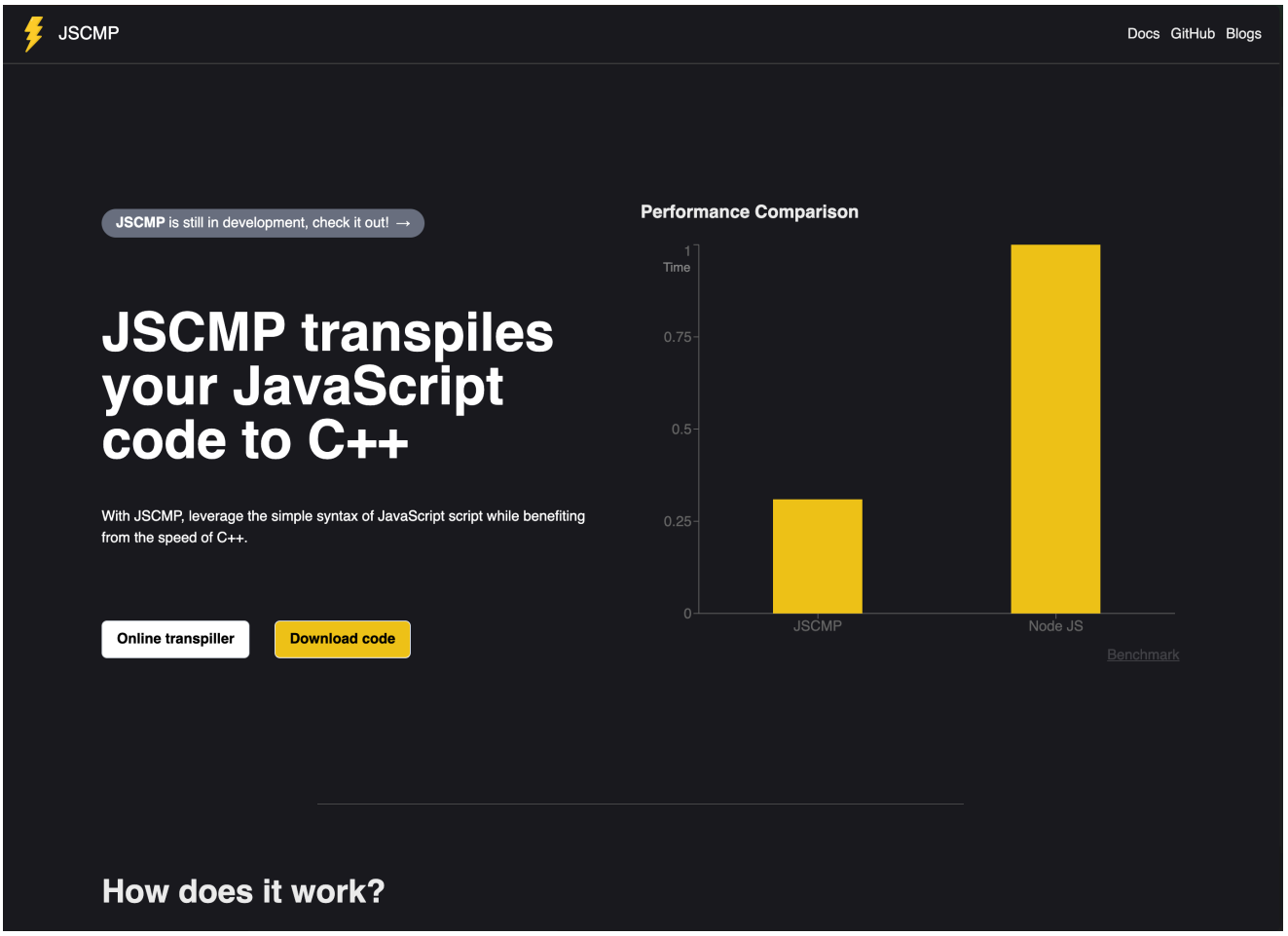
- Incompatible with certain proprietary licenses
- Stricter legal obligations for redistribution
- Potentially less appealing for commercial adoption

Notable Projects Using GPLv3

- GNU/Linux
- VLC media player
- WordPress
- GCC (GNU Compiler Collection)

Engaging Developer Communities

We have setup a website to find contributor and make people interested in our project. you can find our github here and the website url here. We have also setup a plausible on the website to analysis the trafic on it. The website is SSR to make the SEO better and rank up on the article.

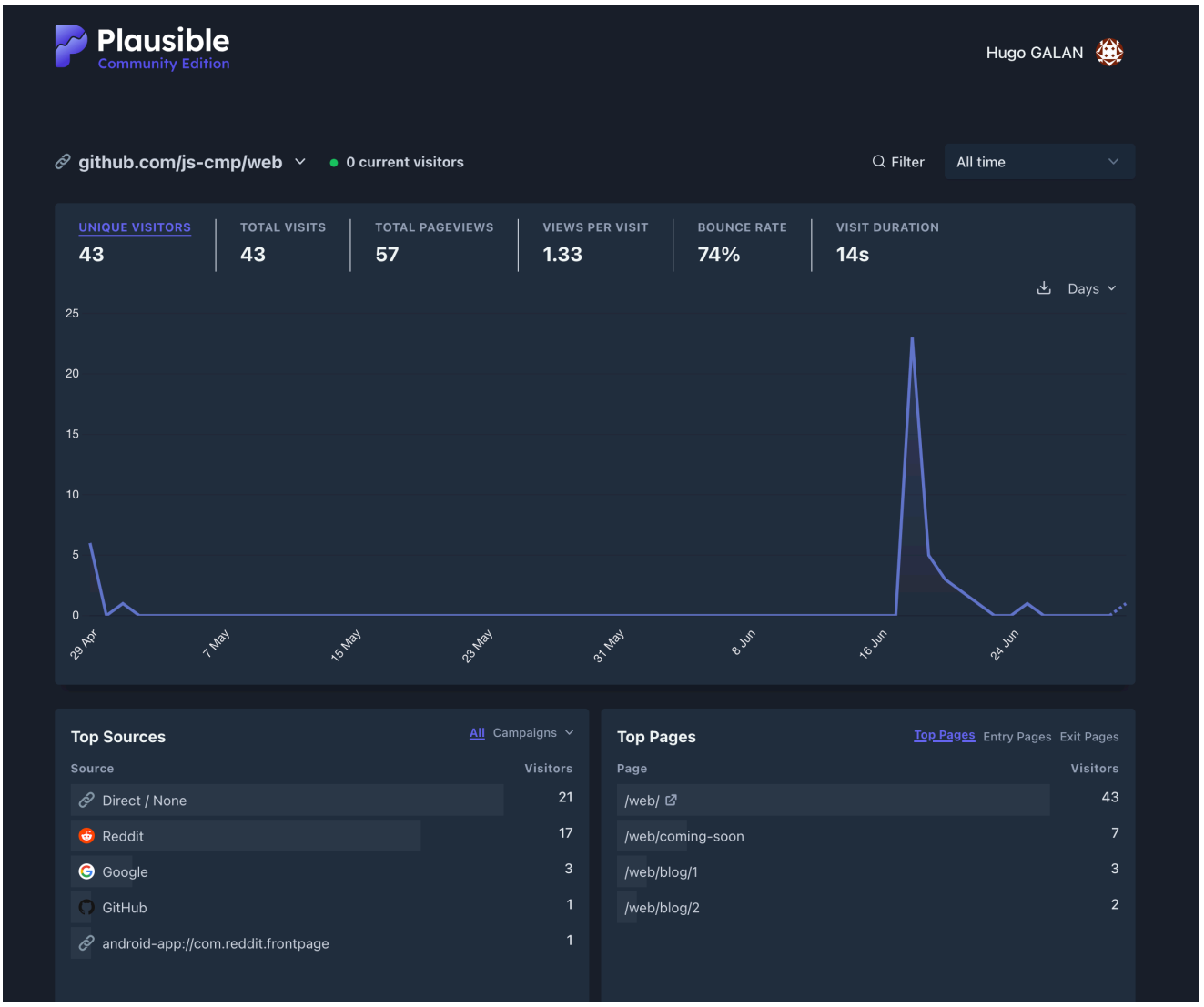


We have made 2 article about our open source project.

- Reducing Boilerplate in C++ with Macros

- Optimizing string concatenation in C++ with Rope data strucuture

Plausible:



We can see after the post on all the socials. We had a big bump on the trafic of the github main repo

To maximize the number of contributors we had all community standards:

as you can see we have added all the template needed for pull request and issue. We have also added a security policy.

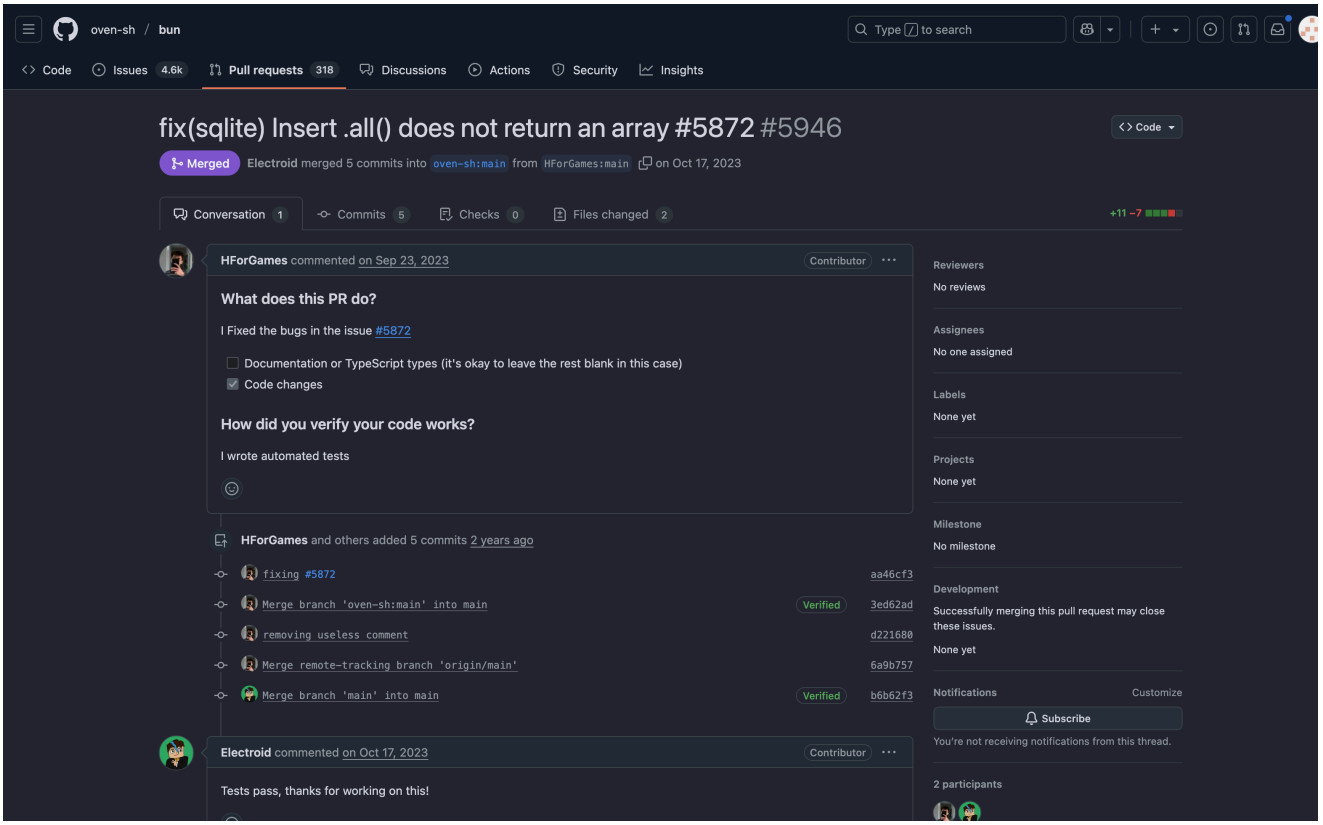
Documenting Contributions

We made concrete contributions to open source projects as part of our technical watch and active involvement in developer communities.

Contribution to Bun

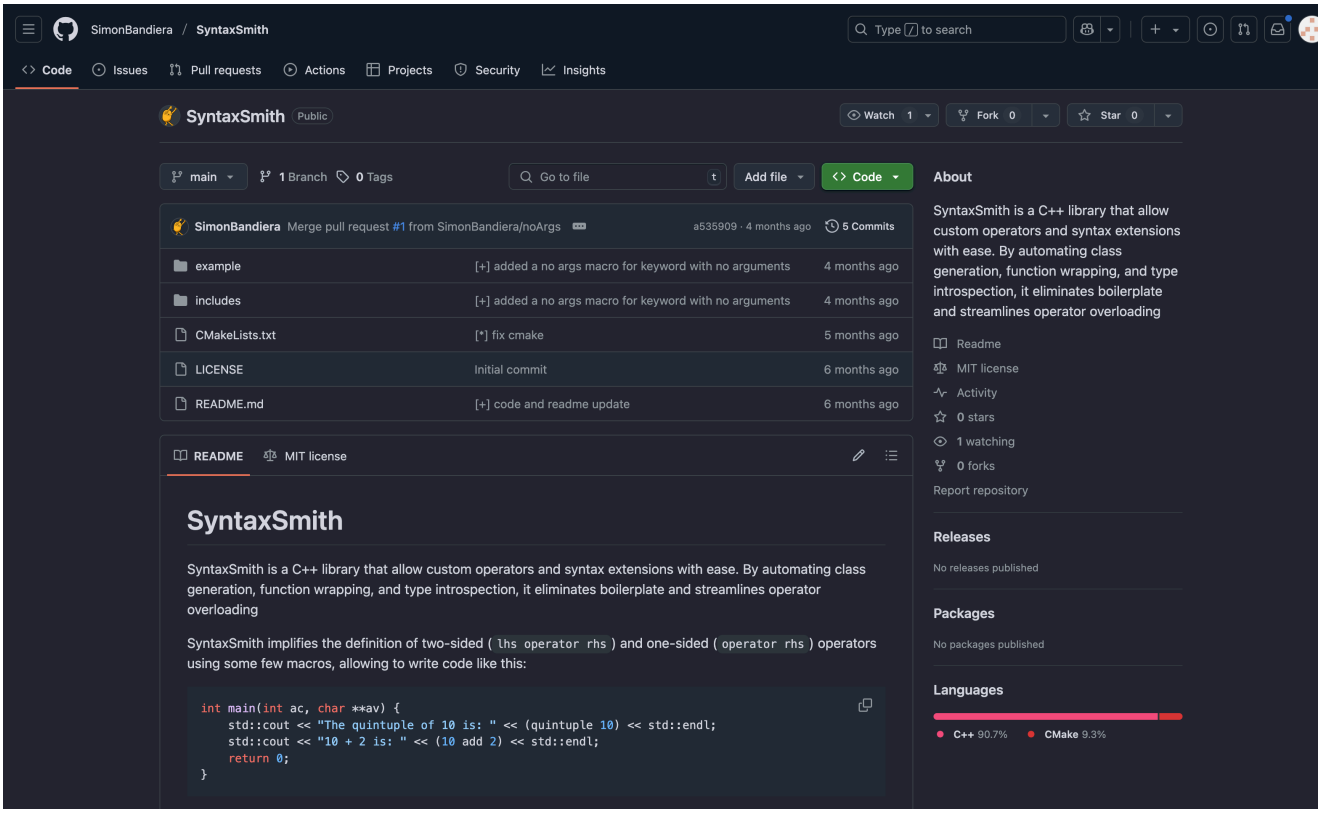
We submitted a pull request to fix an issue in Bun's SQLite interface where the `.all()` method did not return an array as expected:

- **PR**: fix(sqlite) Insert .all() does not return an array – #5946



We chose to contribute to Bun because it is a high performance JavaScript runtime, much like our own project, JS-CMP. By diving into its source code, we gained insight into how a modern JS runtime handles native bindings, database drivers, and type consistency—knowledge that directly informs the design and implementation of our transpiler.

We've also built a C++ library for custom operators, which is a core dependency of JS-CMP. You can check it out here. This library extends C++ syntax capabilities to support JavaScript like operator behavior and is essential for accurate transpilation. As it's part of the open source stack, we actively welcome contributions whether you're fixing bugs, improving performance, or proposing new operator features.



C - Collaborating with Technical Experts

Identifying the Right Technical Problem Solvers

We have found 3 experts and PhD researchers in Computer Science at the University of Kent specialists in C++, transpilation, and programming language theory and invited them to review our project:

1. **Mark Batty**
   A Professor in Concurrency Semantics, he develops rigorous mathematical models and formal verification techniques for real world concurrent systems, particularly focusing on C/C++

memory models and compiler/hardware interactions.

2. **Michael Vollmer**
   A researcher in interpreter design and runtime optimization, he explores the trade offs between AST based and bytecode based execution models, and their impacts on performance and

tooling.

3. **Stefan Marr**
   A Senior Lecturer and Royal Society Industry Fellow, he researches programming language implementation techniques, focusing on combining concurrency models safely and building

tools to detect and prevent subtle concurrency issues in complex systems.

You can find all the mails sent in the same directory with `[name]_Emails.pdf`

Unfortunatly we didn't get any response from Michael Vollmer.

Mobilizing the Right Resources in Case of Technical Issues

While working on JS-CMP, our C++ based JavaScript transpiler, we encountered a key technical limitation: C++ lacks direct equivalents for several JavaScript operators, such as the unsigned right shift (`>>>`), `delete` , `void` , and `typeof` . These operators are fundamental to JavaScript's semantics, and omitting them would compromise compatibility.

To address this, we first discussed possible approaches with our mentor (expert in compiler). We proposed simulating some of these operators using advanced macro techniques in C++. As a starting point, we examined how low level C/C++ macros are used in large scale projects like the Linux kernel—particularly the `container_of` macro described in this blog post and implemented in list.h.

Through this research, we discovered IdOp, a C++ header only library that enables limited forms of custom operator overloading using identifiers. While promising, IdOp wasn't flexible enough for our needs—it lacked the adaptability and extensibility we required to support JavaScript's dynamic behavior.

As a result, we shifted focus to make a custom robust solution: SyntaxSmith. This project offered a more general approach to parsing and rewriting C++ expressions, making it a better fit for our goal of emulating JavaScript operators in a maintainable and scalable way. With it, we aim to integrate operator support into our transpilation process more seamlessly, all while keeping the C++ output readable and consistent with JavaScript logic.