Google Summer of Code 2026

CScout - Visual Studio Code plugin for CScout

GFOSS - Open Technology Alliance

Project size: 350 hrs

Applicant Name	Ujjwal Aggarwal
Email Address	ujjwal264a@gmail.com
Github username	ujjwx1
Primary Language	English
Country of residence	India
Timezone	Indian Standard Time (UTC + 05:30)
Repository	https://github.com/dspinellis/cscout
Mentor	Diomidis Spinellis (dspinellis)

Table of Contents

1. Your Motivation

1.1 My work in CScout [Links]

PR #86: Fixed the `-q` (quiet) option: documented in usage text, fixed code style (spaces → tabs), addressed PICO_QL flag collision in help output. (Merged).

PR #93: Removed obsolete GCC 3.2 compiler workaround in `pdtoken.cpp`, a named temporary variable that was only needed to avoid an internal compiler error in a 20-year-old GCC version. (Merged).

PR #95: Fixed fan-in/fan-out metrics to appear only as post-cpp values: they were incorrectly marked as both pre-cpp and post-cpp, causing duplicate columns in the web UI. Fixes Issue #75 filed by Professor Spinellis. (Open, pending review).

PR #94: Evaluate `sizeof` and `alignof` as constant expressions in the parser AST: addresses a long-standing `XXX` comment in `parse.y`. (Open, pending review).

REST API branch (`feat/rest-api`): Added 10 JSON endpoints + refactoring preview to CScout's SWILL web server. (Branch)

1.2 Why did you choose Open Technologies Alliance (GFOSS)?

I chose GFOSS because it supports technically deep open source projects rather than simple or short-lived ones.
CScout is a 20-year-old C analysis engine used on large codebases like the Linux kernel, and other projects like the GlossAPI handle large volumes of Greek text. These are substantial tools built by experienced developers.
GFOSS provides the community and structure to keep such projects alive and evolving. I wanted to contribute to an organization where the technical bar is high, and the work has a lasting impact, not just a one-off summer project.

1.3 Why do you want to work on this particular project

I chose this project after studying both the current codebase and its history. CScout had a GSoC project in 2019 (by Dimitrios Styliaras under GFOSS) that added a Node.js-based REST server as a separate process sitting alongside CScout. That approach required running

two processes, introduced a separate technology stack (Node.js + C++ REST SDK), and added coordination overhead between the analysis engine and the server.
This project takes a simpler, more integrated approach: embedding REST/JSON endpoints directly into CScout's existing SWILL web server. The same process serves both analysis and data, with no additional dependencies.
I've already implemented and tested this architecture with 10 working endpoints that pull real data from CScout's internal structures. It feels like the natural evolution of the 2019 effort.
My goal is to finish what that earlier project started, delivering a native REST API inside CScout and building a practical VS Code extension that makes its powerful analysis immediately usable inside the editor.

1.4 What are your expectations from us during and after the program

During the program, I expect regular feedback from my mentor on key design decisions and help with some undocumented parts of the codebase. I also welcome early and honest feedback if I'm heading in the wrong direction. After GSoC, I plan to continue maintaining the VS Code extension and responding to issues and pull requests on both repositories.

2. Project Details

2.1 What are you building?

I'm building a VS Code extension that brings the power of CScout's whole-program C analysis directly into the editor. My goal is simple, let developers understand large C codebases more effectively without ever leaving VS Code.
Using this extension, one will be able to browse identifiers, find all references, explore call graphs, check metrics, and even preview refactorings in the IDE itself. We will have to extend CScout by adding a clean REST/JSON API on top of the SWILL web server. This API will be the bridge to let external tools (like my VS Code extension) query the data from CScout in a structured way.
I've already built a working prototype, which has 10 REST endpoints added to CScout's C++ codebase, along with a companion VS Code extension with 36 passing integration tests. This helped me to get a solid understanding of the codebase.
At the very core of it, I'm working on two pieces that fit together to solve this task:

1. Adding REST/JSON API to CScout [Repo]

   I'm adding a set of REST endpoints directly to CScout's embedded SWILL web server. These endpoints will return clean JSON instead of HTML, exposing the internal analysis data without touching the existing web UI or CLI at all.
   As I mentioned above, I've tested 10 endpoints. Here's what they cover:

   - **Identifier endpoints**: list all identifiers with attributes, or look up a single identifier and get every location it appears across the workspace.
   - **File and metrics endpoints**: all analyzed files, per file metrics, per function metrics (nesting, fan-in/ fan-out, cyclomatic complexity).
   - **Call graph endpoints**: retrieve callers or callees for any function as JSON.
   - Project endpoints - list projects and their files, exposing CScout's multi-project workspace structure.
   - **Refactoring preview**: pass an identifier and a new name, CScout walks the equivalence class and returns every file, line, and offset where the replacement would occur.

   This proves that this approach works, but we still have to take it further:

   - **Filtered and paginated queries**: The current endpoints dump everything at once. Works for 1,723 identifiers in the awk example, won't scale to hundreds of thousands in the Linux kernel.
   - **Refactoring apply**: preview exists, but there's no way to actually perform the rename through the API. Requires integrating with CScout's file_refactor() machinery and modification_state tracking.
   - **Function argument refactoring**: CScout can reorder function arguments across an entire codebase via RefFunCall. Not exposed through the API yet.
   - **Should-be-static endpoint**: CScout identifies identifiers with project scope used in only one file. Valuable analysis the web UI shows but the API doesn't yet expose.
   - **Error handling**: proper responses for invalid EIDs, nonexistent files, and malformed parameters.

2. VS Code extension [Repo]

The extension connects to a running CScout instance and surfaces its analysis through standard VS Code patterns. I've already prototyped some features:

- **Sidebar panels** for browsing identifiers (grouped by kind), files, and functions with fan-in/out counts
- **Go-to-definition** (Ctrl+Click) resolving through equivalence classes across the entire workspace. This is a big deal for real-world C codebases where macros make things tricky.
- **Find all references** (Shift+F12) using the same equivalence class resolution across all files.
- **Hover tooltips** showing identifier kind, scope, unused status, and cross-file status.
- **CodeLens annotations** above functions showing caller/callee counts
- **Unused identifier diagnostics** in the Problems panel from CScout's whole-program analysis
- **Rename preview command**, **call graph viewer**, and **function metrics viewer** in the webview panels

All of this is backed up by 36 integration tests, which I had mentioned earlier. But for production quality, we still have a lot of work to do:

- **Proper F2 rename support**: integrating with VS Code's RenameProvider so you can just hit F2, see the changes in a proper diff preview, and apply them safely through the new refactoring endpoint.
- A proper **interactive call graph using D3.js**: replacing the current static list with actual nodes and edges, click-to-expand, zoom, and pan, all inside a VS Code webview.
- **"Should-be-static" diagnostics**: surfacing those alongside the unused identifier warnings so developers get a more complete picture.
- **Lazy loading and caching**: right now, the extension loads everything upfront, which will choke on large codebases. I need to move to on-demand fetching with proper server-side filtering and caching on the extension side.
- **Better cross-platform path handling**: my current fixPath() only works for WSL. I'll need solid support for native Linux, macOS, and Cygwin paths.

- Real-world **testing on bigger projects**: so far, I've only tested on awk example workspace (28 files). I want to push it against SQLite-sized codebases to find and fix scaling issues early.
- Finally, marketplace packaging, proper documentation, and a short demo video to make it easy for people to try

2.2 How will it impact the GFOSS community?

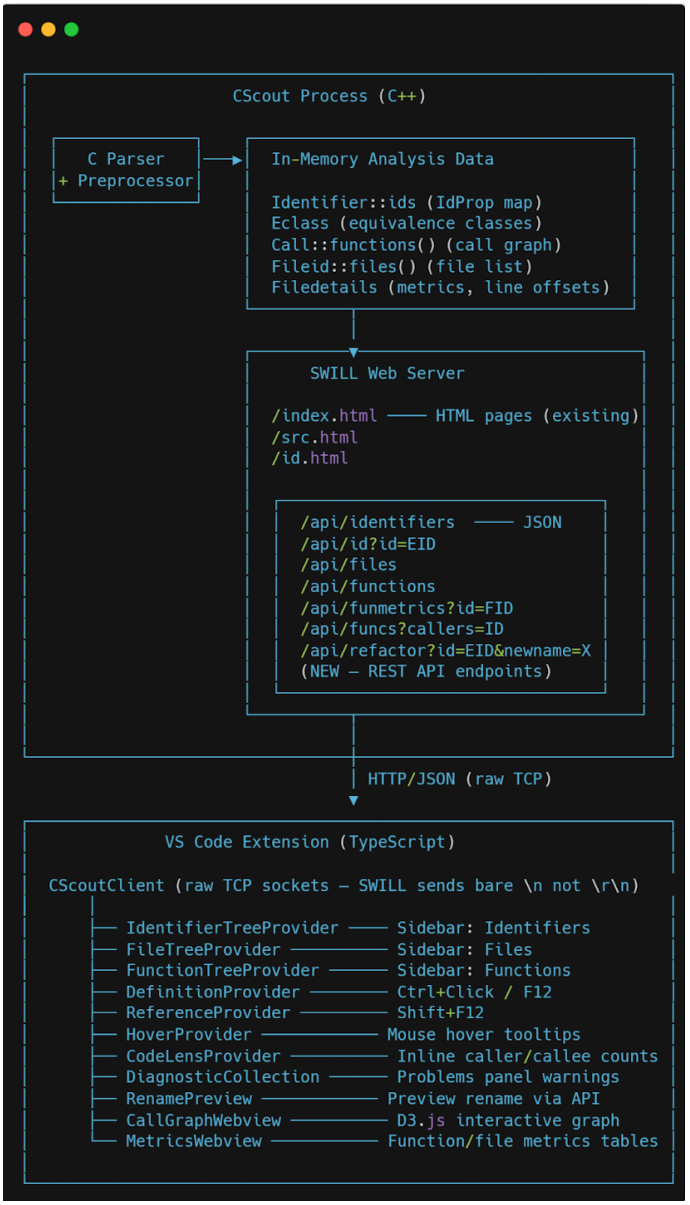
CScout is a powerful tool; it does whole-program analysis with precision. It's been applied to the Linux kernel, FreeBSD, and Apache, but the limitation has always been its interface. The old web UI works, but it feels separate from normal development work. You analyze the code in one window and then go back to your editor to act on what you saw.
This project bridges that gap. The VS Code extension should make Cscout's analysis feel more natural inside the editor, things like inline warnings for unused identifiers and Ctrl+Click navigation that actually understands the whole program.
On top of that, REST API makes it possible for other tools to use CScout's data more easily. Whether it's a CI check, a reporting script, or plugins for other editors, having a clean JSON interface opens up options without breaking the existing web UI.
I hope it lowers the barrier enough that more people give it a serious try, especially those working on larger C codebases. It's also a practical example of how we can modernize established open-source tools by adding a thin API layer instead of rewriting everything from scratch.

2.3 Technologies used

- C++ (CScout REST endpoints)
- TypeScript (VS Code extension)
- VS Code extension API
- Raw TCP sockets (`net.createConnection`): CScout's SWILL server sends HTTP/1.0 with non-standard bare `\n` line endings. Node.js's `http` module rejects these, so the extension uses manual HTTP request/response parsing. O(1) header detection via `indexOf('\n\n')` fallback.
- D3.js (interactive call graph visualisation)
- REST/JSON (API design: communication layer between CScout and extension)

2.4 Technical Architecture & Implementation Plan

I. The Macro Architecture: CScout ↔ VS Code



Key decision: The REST API endpoints are registered alongside the existing HTML handlers using `swill_handle("api/identifiers", api_identifiers, NULL)`. This means a single `cscout` process serves both interfaces. The existing HTML Web UI is completely unchanged.

II. The Micro Architecture: REST API Endpoint Internal Flow

Key decision: The refactoring preview never calls `file_refactor()` or modifies `modification_state`. It walks the equivalence class members read-only and returns the replacement map as JSON.

CScout uses a global `modification_state` to distinguish between unmodified, substitution active, and hand-edit active modes. Once a substitution happens, hand-edits are blocked, and vice versa.

To keep the preview safe, it stays strictly read-only; it only queries `e->get_members()` and `Filedetails::get_line_number()` without touching the substitution pipeline. This prevents any accidental state changes.

The actual apply endpoint (to be implemented during GSoC) will handle the state update. This separation is intentional.

III. Implementation

The REST API Handler Pattern (`cscout.cpp`)

Every endpoint follows the same pattern: set JSON content type, parse parameters from the SWILL query string, iterate the relevant in-memory data structure, and emit JSON. Here is the refactoring preview handler (most complex endpoint):

```
// GET /api/refactor?id=EID&newname=NAME
// Refactoring preview — no files are modified
static int
api_refactor(FILE *of, void *)
{
    swill_setheader("content-type", "application/json");

    Eclass *e;
    if (!swill_getargs("p(id)", &e)) {
        fprintf(of, "{\"error\":\"Missing or invalid id parameter\"}");
        return 0;
    }

    IdProp::iterator idi = ids.find(e);
    Identifier &id = idi->second;

    // Walk the equivalence class — every location across every file
    const setTokid &members = e->get_members();
    set<Fileid> affected_files;
    for (setTokid::const_iterator j = members.begin(); j != members.end(); j++)
        affected_files.insert(j->get_fileid());

    // For each file, emit replacement locations as JSON
    for (set<Fileid>::const_iterator fi = affected_files.begin(); fi != affected_files.end(); fi++) {
        for (setTokid::const_iterator j = members.begin(); j != members.end(); j++) {
            if (j->get_fileid() != *fi) continue;
            int line = Filedetails::get_line_number(j->get_fileid(), j->get_streampos());
            fprintf(of, "{\"line\":%d,\"offset\":%lu,\"length\":%d}",
                line, (unsigned long)j->get_streampos(), e->get_len());
        }
    }
    // NO FILES MODIFIED — pure read-only preview
    return 0;
}
```

The key line is `const setTokid &members = e->get_members()`. This is where CScout's equivalence class gives us every single location, across all files, where that identifier appears. No other tool can do this quite like CScout, because none of them properly track identifiers through the C preprocessor.

The Raw TCP HTTP Client(`cscoutClient.ts`)

SWILL sends HTTP/1.0 responses with bare \n instead of the spec-required \r\n. Node.js's `http` module rejects these. The extension uses `net.createConnection` with manual header parsing:

```
    private get(path: string): Promise<string> {
        return new Promise((resolve, reject) => {
            const net = require('net');
            const socket = net.createConnection({ host: this.host, port: this.port });
            let raw = '';
            let settled = false;

            const fail = (err: Error) => {
                if (settled) { return; }
                settled = true;
                socket.destroy();
                reject(err);
            };

            socket.setTimeout(10000);
            socket.setEncoding('utf-8');

            socket.on('connect', () => {
                socket.write(
                    `GET ${path} HTTP/1.0\r\n` +
                    `Host: ${this.host}:${this.port}\r\n` +
                    `Connection: close\r\n` +
                    `\r\n`
                );
            });

            socket.on('data', (chunk: string) => { raw += chunk; });

            socket.on('end', () => {
                if (settled) { return; }
                settled = true;

                // Skip HTTP headers, find body after blank line
                let bodyStart = raw.indexOf('\r\n\r\n');
                if (bodyStart !== -1) {
                    bodyStart += 4;
                } else {
                    bodyStart = raw.indexOf('\n\n');
                    bodyStart = bodyStart !== -1 ? bodyStart + 2 : 0;
                }

                resolve(raw.substring(bodyStart));
            });

            socket.on('timeout', () => fail(new Error(`Timeout fetching ${path}`)));
            socket.on('error', (err: Error) => fail(err));
        });
    }
```

The WSL Path Conversion (`extension.ts`)

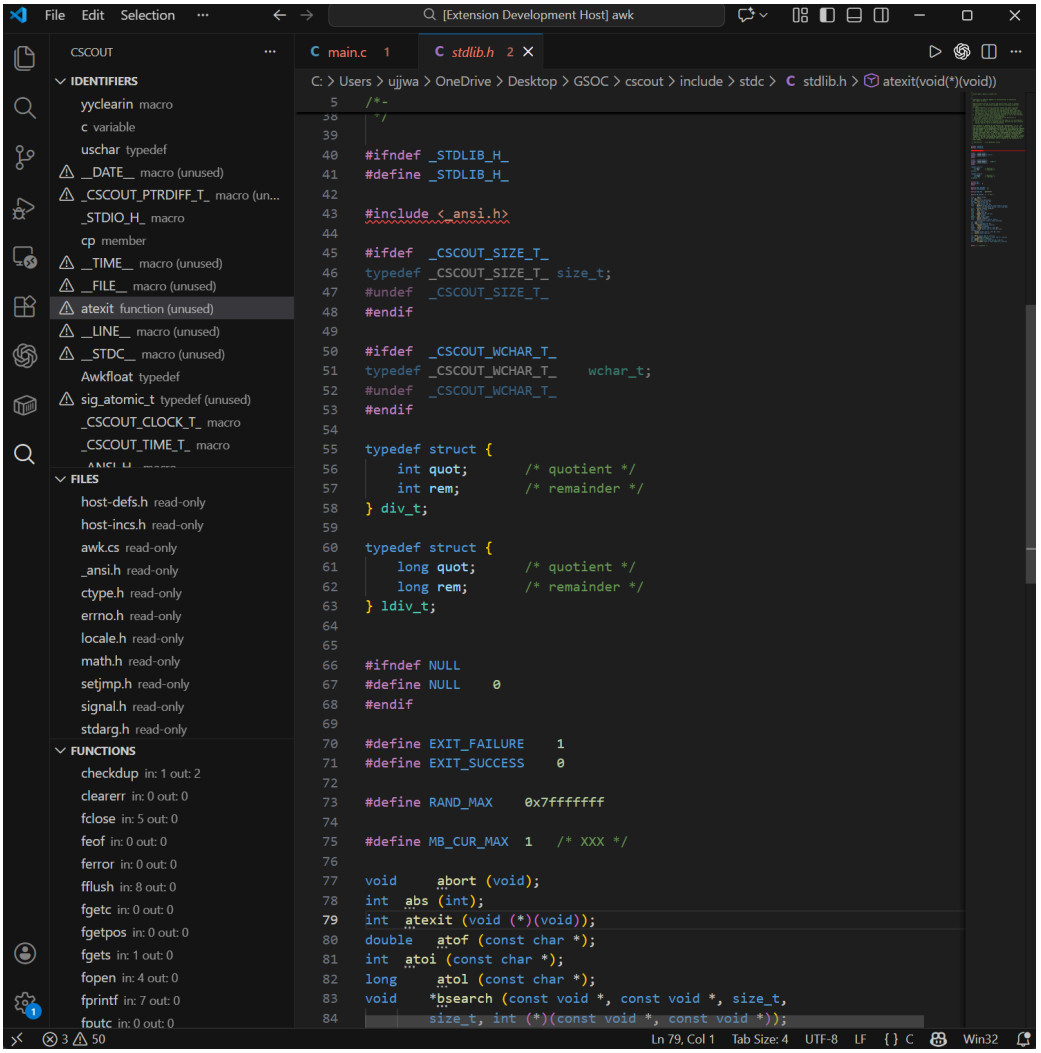
CScout runs in WSL and returns Linux paths (/mnt/c/Users/...). VS Code on Windows needs Windows paths (C:/Users/...). The `fixPath()` function handles this conversion.

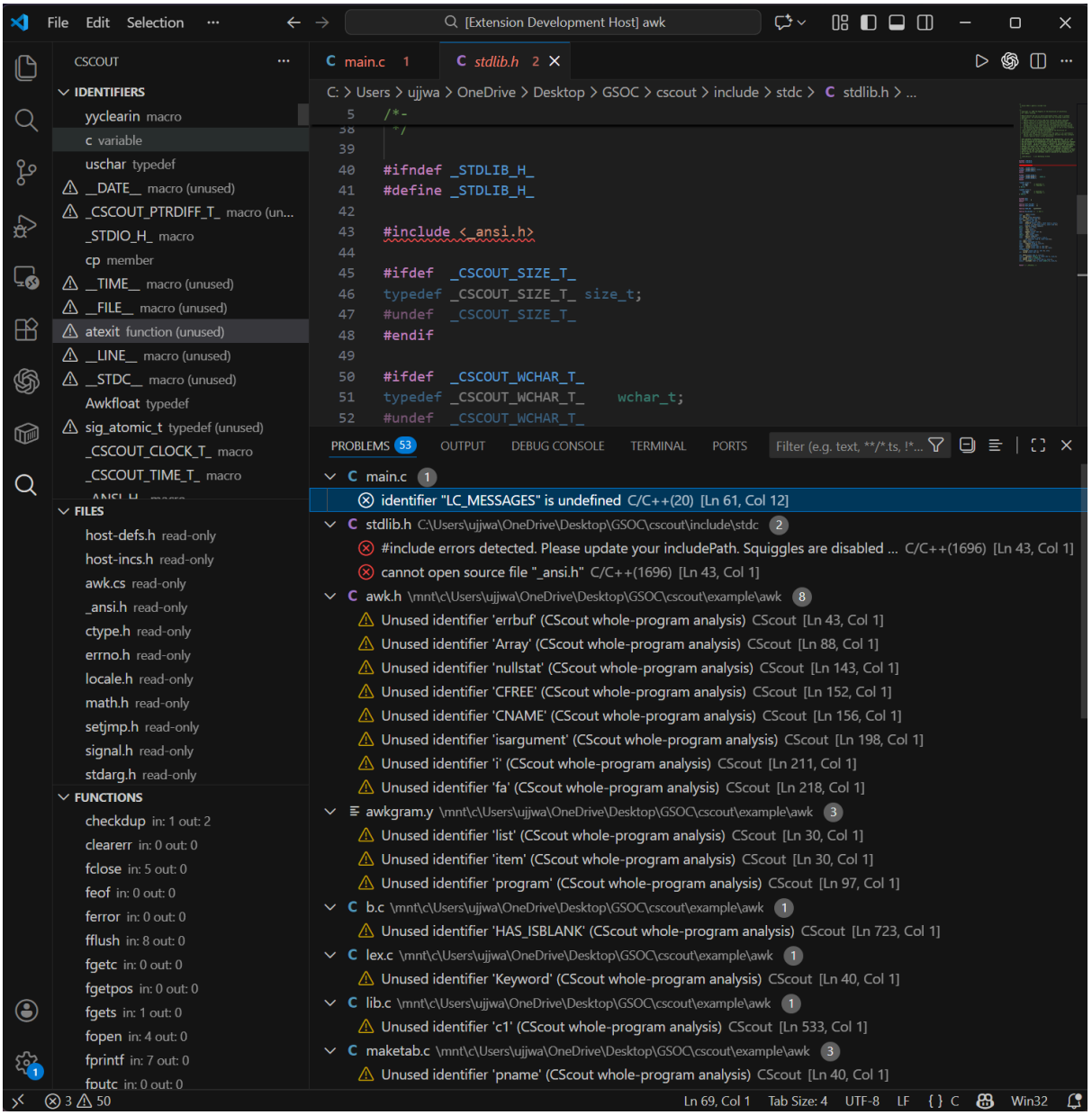
```ts
function fixPath(p: string): string {
    const m = /^\/mnt\/([a-zA-Z])\/(.*)$/.exec(p);
    if (m) { return `${m[1].toUpperCase()}:/${m[2]}`; }
    return p;
}
```

This is currently WSL-only. During GSoC, I will expand this to detect and handle native Linux, macOS, and Cygwin paths.

2.5 Output Verification

Output 1: Live Extension Screenshot





These screenshots demonstrate that the REST API serves real data from a live CScout process, navigation through equivalence classes works across files, and whole-program unused analysis surfaces as VS Code diagnostics.

Output 2: Refactoring Preview JSON

The refactoring is the endpoint Professor Spinellis specifically described wanting. Here is the actual output for renaming `checkdup → check_duplicate` in the awk project:

```
{
  "eid": "0x5b1b17537440",
  "old_name": "checkdup",
  "new_name": "check_duplicate",
  "affected_files": 1,
  "total_replacements": 3,
  "changes": [
    {
      "fid": 4,
      "file": "awk/awkgram.y",
      "replacements": [
        { "line": 30,  "offset": 1227,  "length": 8, "old": "checkdup", "new": "check_duplicate" },
        { "line": 430, "offset": 12829, "length": 8, "old": "checkdup", "new": "check_duplicate" },
        { "line": 477, "offset": 13660, "length": 8, "old": "checkdup", "new": "check_duplicate" }
      ]
    }
  ]
}
```


CScout found all 3 locations, the function declaration (line 30), and both call sites (lines 430, 477). No files were modified. The extension can display this as a diff preview before the user commits.

Output 3: Test Suite Validation

```
ujjwal@UjjwalLenovo:/mnt/c/Users/ujjwa/OneDrive/Desktop/GSOC/cscout-vscode$ node out/test.js
✓ server is reachable
✓ GET /api/identifiers returns array
✓ identifiers have required fields
✓ identifiers include known awk functions
✓ unused identifiers exist
✓ function identifiers have fun=true
✓ macro identifiers have macro=true
✓ GET /api/id returns locations
✓ identifier locations have required fields
✓ unused identifier has locations
✓ GET /api/files returns array
✓ files have required fields
✓ files include .c and .h files
✓ GET /api/filemetrics returns metrics
✓ file metrics include standard fields
✓ file metrics values are non-negative
✓ GET /api/functions returns array
✓ functions have required fields
✓ functions include known awk functions
✓ checkdup has correct fan-in/fan-out
✓ GET /api/funcs?callers returns array
✓ GET /api/funcs?callees returns array
✓ checkdup callers include yyparse
✓ checkdup callees include strcmp
✓ GET /api/projects returns array
✓ projects include awk
✓ identifier count matches across endpoints
✓ file count matches across endpoints
✓ function count matches
✓ GET /api/funmetrics returns metrics
✓ GET /api/refactor returns preview
✓ refactoring preview does not modify files
✓ refactoring preview with invalid EID returns error
✓ refactoring preview without newname returns error
✓ invalid identifier EID returns error
✓ invalid file metrics ID returns error
36 tests: 36 passed, 0 failed
```

Test names read as a specification: `checkdup has correct fan-in/fan-out` verifies the call graph data. `refactoring preview does not modify files` verifies the safety invariant. `checkdup callers include yyparse` verifies cross-function call resolution.

3. Timeline

GSoC 2026 Dates:

- Community bonding: May 1 – May 24
- Coding begins: May 25
- Midterm evaluation deadline: July 10
- Coding ends: August 18
- Final evaluation: August 24 – 31

Community Bonding Period (May 1 - May 24)

During community bonding, I want to focus on getting aligned with my mentor and making sure we're on the same page before heavy coding starts:

- Discuss and finalize the overall API design, things like consistent JSON structure, error handling format, and whether we want any URL versioning.
- Talk through the refactoring apply endpoint, how we should handle the actual changes, what "apply" semantics should look like, and how to deal with concurrency if someone edits files while a rename is happening.
- Set up a basic CI pipeline that automatically validates the output of all endpoints so we catch regressions early.
- Spend some time diving deeper into CScout's refactoring internals, so I'm ready to implement the apply part properly.

Phase 1: Completing the REST API

Week 1 (May 25 - May 31)

I'll focus on making the list endpoints more practical for larger codebases:

- Add server-side filtering to the identifier endpoint (by unused, writable, scope, kind, etc.).
- Introduce text search and pagination (offset + limit) for all list-type endpoints.

This should take care of the biggest scaling issues I've seen so far when dealing with real projects.

Week 2 (June 1 - June 7)

The main goal is to get the refactoring apply endpoint working:

- Implement the actual rename operation through the API.
- Handle modification state so API renames and hand-edits don't conflict
- Also, expose "should be static" analysis as a JSON endpoint

Week 3 (June 8 - June 14)

I want to extend the refactoring capabilities a bit further:

- Add preview and apply support for function argument reordering
- Add a dedicated endpoint for querying identifier attributes to support more advanced filtering on the extension side.

Week 4 (June 15 - June 21)

This week will mostly be about polishing the API and making sure it's reliable:

- Error handling, proper JSON errors for invalid inputs, missing parameters, and edge cases
- Comprehensive endpoint test suite
- Verify all endpoints against awk and at least one larger project
- Document every endpoint's request/response format

Phase 2 - VS Code Extension

Week 5 (June 22 - June 28)

- Integrate rename with VS Code's native F2 using RenameProvider
- Show changes in diff editor instead of text output
- Apply on user confirmation through the apply endpoint

Week 6 (June 29 – July 5)

- Implement lazy loading, names on connect, details on demand
- Hook into filtered query endpoints so only needed data is fetched
- Add “should be static” diagnostics alongside unused warnings in Problems panel

Midterm evaluation (July 10). Expected state: complete REST API with filtering, pagination, refactoring apply. Extension with F2 rename, lazy loading, and both types of diagnostics.

Week 7 (July 6 - July 12)

- Interactive call graph webview using D3.js
- Force-directed layout, nodes for functions, edges for calls
- Click to expand, zoom and pan for large graphs

Week 8 (July 13 - July 19)

- File and function metrics webview panels, sortable tables
- Click any row to navigate to that function or file
- Add grouping and filter options to sidebar panels

Week 9 (July 20 - July 26)

- Cross-platform path handling, support WSL, native Linux, macOS, Cygwin
- Add function argument refactoring preview to extension UI
- Performance profiling on a medium-sized codebase

Week 10 (July 27 - August 2)

- Buffer week for catching up on anything that slipped
- Fix bugs from weeks 5-9
- Address midterm feedback from mentor

Phase 3 - Testing, Documentation, Ship

Week 11 (August 3 - August 9)

- Test against a real-world C project with hundreds of files
- Find and fix scaling issues and edge cases
- Package extension for VS Code marketplace

Week 12 (August 10 - August 18)

- Write README with screenshots, user guide, API reference
- Record demo video showing full workflow end to end
- Final cleanup and address remaining mentor feedback

Final evaluation (August 24 - 31) Deliverables: CScout PR with complete REST API, published VS Code extension, documentation, and demo video.

Availability: During the entire GSoC period, I'm available to work for 30-35 hours per week. I don't have any vacations or plans during the summer, and I want to dedicate it into expanding my skillset in working on production quality projects and open source projects.

Note: I have my college end semester exam from 11 May to 23 May and will be unavailable for this duration; however, I will make up for the time before and after this duration. My community bonding tasks will be front-loaded into May 1-10 and completed after May 24.

4. Motivation (Why me?)

I've spent significant time working inside CScout's codebase and am comfortable with its core internals. I understand key structures like `Eclass::get_members()` (which returns equivalence classes as `setTokid`), `Filedetails::get_line_number()`, and how SWILL's `swill_handle()` works with its `FILE*` and `fprintf` output model. I've also traced through the refactoring logic in `file_refactor()` and `get_refactored_part()`.

I've been actively engaging with Professor Spinellis through issue #82. I've shared screenshots in the thread and submitted several pull requests (#86, #93, #94, #95) that show that I can make clean, testable changes to the codebase.

Before writing this proposal, I built a working proof of concept. The REST API already extracts real data from CScout's internals, the VS Code extension displays, and I have 36 passing integration tests. The refactoring preview is functional and accurate. This is working, tested code, not just a promise of future work.

Beyond my work on CScout, I have experience collaborating in large teams and organisations and have been part of internationally awarded teams. These experiences have sharpened both my technical skills and ability to collaborate effectively with others. Furthermore, I recently filed a patent (Application No. 202611020879) which demonstrates my ability to identify technical problems, design novel solutions, and take ideas from concept to formal invention.

5. After GSoC

I want to continue working on the development and maintenance of my work and not just restrict myself to the duration of the Google Summer of Code. I will take responsibility for the codebase and will be actively available.
There are several features I would love to explore:

- **Language Server Protocol (LSP) wrapper**: It would be great to wrap the REST API in a proper LSP server. That way, CScout's analysis could be used from other editors like Emacs or JetBrains IDEs, not only VS Code.
- **CI integration**: Build a simple command-line tool that talks to the REST API and generates reports in common formats like SARIF. This would let people easily plug CScout into their CI pipeline for things like unused code detection.

6. Links and Contributions

Pull Request	Link	Status
PR #86 (quiet flag fix)	https://github.com/dspinellis/cscout/pull/86	Merged
PR #93 (remove compiler workaround)	https://github.com/dspinellis/cscout/pull/93	Merged
PR #94 (feat: evaluate SIZEOF and ALIGNOF)	https://github.com/dspinellis/cscout/pull/94	Open
PR #95 (Fix fan-in/fan-out)	https://github.com/dspinellis/cscout/pull/95	Open

- CScout REST API branch: https://github.com/ujjwx1/cscout/tree/feat/rest-api
- VS Code extension: https://github.com/ujjwx1/cscout-vscode
- Issue #82 discussion:
  https://github.com/dspinellis/cscout/issues/82#issuecomment-4063973243
  https://github.com/dspinellis/cscout/issues/82#issuecomment-4029748363