Systematic Triage and Resolution of High-Impact clang-tidy & Clang Static Analyzer Diagnostics

CppAlliance Fellowship @ Compiler-Research 2026

Peiqi Li
Shanghai Jiao Tong University, Shanghai

Mentor:
Vassil Vassilev

---

1. Abstract & Motivation

This proposal focuses on resolving high-impact diagnostic issues in `clang-tidy` and the Clang Static Analyzer, including false positives, unsafe fix-it hints, and other sources of diagnostic noise. In modern C++ engineering, a static analyzer with even a low false-positive rate is frequently bypassed or disabled by development teams. By addressing these pain points conservatively, this project aims to reduce noise and improve the reliability of Clang-based diagnostics in real-world usage.

The foundation of this proposal builds on my active experience navigating the LLVM codebase, specifically resolving a `Sema/CodeGen` state desynchronization issue in Clang's REPL mode (Issue #146770, PR #192588).

2. Expected Deliverables

To provide a concrete measure of success, this 6-month project targets the following total outputs:

A Prioritized Triage Matrix: Categorizing 15-20 high-impact open issues.
8 to 12 Merged PRs: Addressing frontend semantics, AST matchers, or checker-level states.
Robust Test Coverage: Comprehensive `llvm-lit` regression tests (both positive triggers and negative false-positive prevention) for all resolved diagnostics.
Documentation & Follow-ups: Issue updates and documentation describing the resolutions, technical limitations, and any remaining edge cases.

3. Prioritization Heuristic

In a large issue tracker, target selection is critical. Prioritization in this project will follow these guidelines:

Report Frequency: Volume of identical failures reported across downstream projects
Code Pattern Ubiquity: Issues involving widely used C++ constructs (e.g., `std::` containers, templates, perfect forwarding)
Suppression Rate: Diagnostics that frequently lead to `NOLINT` usage
Fix Safety: Likelihood of implementing a local fix without introducing regressions

4. Execution Methodology

To make patches acceptable upstream, I will follow a consistent workflow:

Extract a minimal reproducer for each issue;
Apply fixes at the narrowest possible layer (e.g., matcher refinement vs check-level logic) to avoid unintended regressions;
Handle fix-it hints conservatively: only emit or modify them when correctness is clear; otherwise keep diagnostics as warnings

5. Target Issue Identification (Case Studies)

The following examples illustrate the technical depth required to resolve these issues, moving beyond superficial generic fixes.

Case Study 0: Frontend State Desynchronization (Active Work: Issue #146770 / PR #192588)

Bug Trigger: Implicit template instantiation inside a statement that later fails to parse in REPL mode.
Root Cause: `CodeGen` drops the failed module, while `Sema` retains instantiated bodies (e.g., `std::pow`). Subsequent valid calls reuse the cached body and skip emission, leaving the JIT without symbols.
Resolution Strategy: Implemented a targeted bypass in `Sema::MarkFunctionReferenced` to force re-emission of valid implicit instantiations in `IncrementalExtensions` mode, relying on `GlobalDeclMap` for safe symbol deduplication.

Case Study 1: Perfect Forwarding Blind Spots (e.g., Issue #179090)

Observation: `performance-unnecessary-value-param` incorrectly flags cases involving `std::map::try_emplace` with `std::function` rvalues.
Resolution Strategy: A likely direction involves refining the parameter usage matchers to better account for value category propagation through nodes like `CXXConstructExpr` and `MaterializeTemporaryExpr` within template instantiations and validating the behavior on minimal reproducers before considering a fix.

Case Study 2: Opaque Smart Pointer Mutations (e.g., Issue #179941)

Observation: `readability-use-anyofallof` does not treat dereferences through `std::shared_ptr` as mutations of the underlying object.
Resolution Strategy: A likely direction is to extend the mutation matcher to penetrate overloaded dereference operators (e.g., `CXXOperatorCallExpr`), ensuring state mutations reach the underlying pointee while strictly avoiding false flags on standard iterator traversals.

6. Detailed Project Timeline (6 Months / 24 Weeks)

Given the nature of the LLVM upstreaming process, the timeline assumes concurrent workflows. While one patch is blocked waiting for upstream code review, triage and implementation for the next target will proceed.

Weeks 1 - 2: Triage Finalization & Infrastructure Setup

Process the clang-tidy and CSA backlog using the prioritization heuristic. Extract 5-8 primary target issues with the highest impact. **Deliverable:** A finalized priority matrix. Setup of automated scripts for rapid local reproducer testing.

Weeks 3 - 4: First Blood - AST Matcher Refinement

Address Case Study 1 (forwarding blind spots). Write minimal reproducers, update the AST matchers, and author `llvm-lit` negative tests. **Deliverable:** Submit PR #1 & PR #2.

Weeks 5 - 6: Fix-it Safety & Iterative Review

Address code review feedback for earlier PRs. Concurrently begin work on Case Study 2 (opaque smart pointer mutations). **Deliverable:** Submit PR #3.

Weeks 7 - 10: Scaling clang-tidy Resolutions (Mid-term)

Expand focus to additional high-noise `clang-tidy` issues identified during triage. Audit the safety of their associated fix-it hints; downgrade to warnings if safety across macro expansions cannot be guaranteed. **Deliverable:** Submit PRs #4 - #6. Prepare the Mid-term evaluation summary.

Weeks 11 - 14: Pivot to Clang Static Analyzer (CSA)

Iterate on pending `clang-tidy` code reviews. Begin the CSA phase by targeting issues with high frontend overlap (e.g., pointer decay or implicit casts confusing the analyzer state). **Deliverable:** Submit initial CSA PRs. Ensure all earlier `clang-tidy` PRs are in final review.

Weeks 15 - 18: CSA Deep Dive & False Positive Reduction

Focus on specific high-noise CSA checkers (e.g., memory leak false positives). Modify checker-level logic or lightweight state transitions, focusing on cases where the root cause overlaps with frontend semantics, while explicitly avoiding deep changes to core path-sensitive analysis or Exploded Graph evaluations. **Deliverable:** Submit PRs targeting core CSA noise reduction.

Weeks 19 - 22: The Long-Tail & Edge Cases

This period acts as a buffer for prolonged code reviews (which are standard in LLVM). Address complex edge cases, regressions reported by upstream CI, and reviewer requests for broader test coverage. **Deliverable:** Achieve merge consensus on all outstanding PRs (Targeting 8-12 total merges).

Weeks 23 - 24: Wrap-up & Final Documentation

Finalize any remaining code changes. Document the limitations of current AST matchers/checkers discovered during the project to guide future contributors. Update upstream issues with resolutions. **Deliverable:** Final Project Report submitted.

7. Biographical Information

I am currently a dual-degree student in Mathematics and Artificial Intelligence, with a deep interest in C++ infrastructure. Through building high-performance systems—such as a deterministic high-frequency matching engine using zero-allocation object pools—I've spent a lot of time wrestling with C++17 memory models. That hands-on experience taught me firsthand why reliable and quiet compiler tooling is so critical for developers. I genuinely enjoy diving into massive codebases like LLVM, writing strict regression tests, and learning from the upstream open-source review process**.**