LLM-Integrated Declarative Program Analysis

Sara Baradaran
University of Southern California
Los Angeles, CA, USA
sbaradar@usc.edu

Amirmohammad Nazari
University of Southern California
Los Angeles, CA, USA
nazaria@usc.edu

Mukund Raghothaman
University of Southern California
Los Angeles, CA, USA
raghotha@usc.edu

Abstract

Program analysis tools such as CodeQL enable programmers to express their questions about codebases in the form of declarative queries, which are then evaluated over structured representations of the code. These versatile tools have broad applications in bug finding, vulnerability discovery, and codebase exploration. Still, they are limited in their ability to answer questions that rely on semantic judgments which cannot be expressed or decided using program analysis tools alone, e.g., identifying string literals that contain private information or violations of naming conventions.

In this paper, we present SemQL, a system which extends declarative program analysis frameworks with the ability to invoke an LLM as an external oracle. SemQL allows developers to write queries which combine structural reasoning with semantic ("*extra-analytic*") judgments. We show the real-world value of such a system by collecting a set of analytic questions that require semantic reasoning beyond what is deducible simply from the structure of the code. We also present an algorithm which efficiently evaluates these queries while minimizing costly oracle invocations, and demonstrate its effectiveness in practical program analysis tasks.

CCS Concepts: • **Theory of computation → Program analysis**; • **Software and its engineering → Automated static analysis**.

Keywords: CodeQL, declarative program analysis, static analysis, code question answering, large language model

ACM Reference Format:
Sara Baradaran, Amirmohammad Nazari, and Mukund Raghothaman. 2026. LLM-Integrated Declarative Program Analysis. In *Proceedings of the 15th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis (SOAP '26), June 15–19, 2026, Boulder, CO, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3814987.3814993

1 Introduction

Program analysis tools have long played a foundational role in helping developers reason about the behavior and security of software systems. Frameworks such as CodeQL [14], Semgrep [6], and Comby [2] provide structured representations of program syntax, which enable automated analyses over large codebases, including data flow tracking and code scanning. These tools are indeed effective at answering questions about codebases that can be expressed purely in terms of program structure and well-defined rules. Answering many questions, on the other hand, requires subjective judgments that depend on domain-specific knowledge, intent, or nuanced human reasoning beyond the program structure.

A representative example arises in the analysis of URL handling code. To prevent vulnerabilities such as server-side request forgery [7] and malicious redirection [10], developers commonly sanitize untrusted URLs by checking whether their host belongs to an allowed set [5]. In practice, this check is often approximated using string-based heuristics, such as testing for the presence of an allowed hostname as a substring of the URL. These heuristics are nevertheless error-prone. For example, malicious URLs can embed trusted hosts in unexpected locations, which may lead to unintended behavior even in non-security-critical contexts.

At the same time, large language models (LLMs) have demonstrated strong capabilities in tasks that require semantic understanding, commonsense reasoning, and access to external knowledge [20, 27]. One can reasonably imagine an LLM oracle that, given a URL and a description of a validation routine, assesses whether the URL should be considered trustworthy. However, LLMs alone are ill-suited for systematically analyzing large codebases: they lack a structured view of the program and struggle to scale to millions of lines of code [28, 29].

This paper explores a synthesis of these two paradigms. We extend CodeQL [24], a program analysis framework that supports expressive queries over large codebases, with a new construct that allows queries to invoke external oracles such as LLMs to resolve predicates requiring reasoning beyond the code structure. In this setting, CodeQL is responsible for identifying relevant program facts and their relationships, while the oracle is used to evaluate whether these facts match a given condition that cannot be expressed or decided within the analysis framework alone.

This combination allows one to write a query that identifies untrustworthy URLs within a codebase. When evaluating such a query, the CodeQL engine identifies URLs that are used in the codebase and delegates to an oracle the task of determining whether each URL is considered trustworthy.

SOAP '26, Boulder, CO, USA

ACM ISBN 979-8-4007-2709-2/2026/06
https://doi.org/10.1145/3814987.3814993

This integration of CodeQL with LLM oracles presents two fundamental challenges: First, the CodeQL engine is closed-source, which in turn constrains how interactions with the external oracle can be performed. Second, invoking LLMs as the oracle is both computationally and financially expensive. To maintain efficiency at scale, it is therefore essential to minimize the number of oracle queries while ensuring that the analysis remains sound.

To address the first challenge, we introduce a frontend query language, named SemQL, which extends CodeQL with a new oracle invocation construct. This allows programmers to write queries that incorporate semantic judgments beyond the native expressiveness of CodeQL. Queries written in this language are decomposed into a sequence of intermediate queries conforming to the standard CodeQL specification, which are used to extract the relevant program facts. These facts are then submitted to an LLM oracle to obtain semantic decisions. Finally, the oracle results are re-integrated by synthesizing a collection of auxiliary predicates, which allows us to rewrite the original query in standard CodeQL so that it can be evaluated by the CodeQL engine.

To address the second challenge, we rewrite the original query in disjunctive normal form (DNF) and decompose it into a set of conjunctive subqueries. This allows us to incrementally evaluate each subquery and prune the program facts which do not need oracle assessment, ultimately reducing the total number of queries submitted to the LLM.

To show the real-world application of SemQL, we constructed a set of benchmarks by consulting bug descriptions in the SpotBugs documentation [8] and CodeQueries [26], a dataset of semantic queries over code. Each benchmark specifies a bad coding practice in Java whose identification requires semantic judgments and asks, as an analytic question, whether such instances occur in a given codebase. We also crafted a set of SemQL queries that enable answering these questions over arbitrary codebases.

In summary, we make the following contributions:

(1) We introduce SemQL by extending the CodeQL language with a new construct that enables invoking external oracles over program facts.
(2) We demonstrate the applicability of the extended language by developing SemQL query detectors to identify a set of bad coding practices in Java codebases.
(3) We propose a DNF-based algorithm for efficiently evaluating SemQL queries and demonstrate its scalability on a large sample codebase from the SpotBugs test suite [9].

2 Motivating Example

Identifying misnamed fields. Consider the following coding guideline from the Oracle Java documentation [3], the violation of which is also identified as bad practice by SpotBugs [8]: «*Names of fields that are not* `final` *should be in mixed case with a lowercase first letter and the first letters of subsequent words capitalized. Also, names of* `final` *fields should be all uppercase, with words separated by underscores.*»

A programmer may want to identify violations of this guideline in her codebase. In this case, manual inspection is infeasible for all but the smallest codebases. One may, for example, write the following CodeQL query as a detector to identify `final` fields that violate the naming convention:

```
from Field f    /* Example Query 1 */
where f.isFinal() and
      not f.getName().regexpMatch("^[A-Z_]+$")
select f, "field violates naming convention"
```

Notice however that a field named `MAXSIZE` (as opposed to `MAX_SIZE`) cannot be detected by this query, as a more thorough analysis would require human judgment to determine how sequences of uppercase letters should be segmented into meaningful words (in this case, `MAX` and `SIZE`).

On the other hand, one may suggest using contemporary LLMs to identify such cases. This, however, presents several challenges: (*a*) Since LLMs lack a structured view of the code, the quality of their output is not guaranteed, and the results may contain false positives or misflagged code [25]. (*b*) Given that field declarations often spread across various files, the LLM must be provided with the entire codebase. At the same time, LLMs impose resource usage limits, e.g., on the number of uploaded files, which requires partitioning large codebases. (*c*) LLM-based analysis suffers from high latency and occasional non-termination. Queries may take a long time, fail to return results, or require repeated attempts.

SemQL overview. To address these problems, we extend the CodeQL language [4] with an oracle invocation construct that allows the programmer to write, for example, the following query to detect inappropriately named `final` fields:

```
from Field f    /* Example Query 2 */
where f.isFinal() and not LLMQuery(f.getName(), "name is
all uppercase with words separated by underscore"/* c1 */)
select f, "field violates naming convention"
```

Here, `LLMQuery(`e`,`s`)` is a two-arity predicate which takes a string-valued expression e and a question/specification s (in the form of a string literal), and uses the LLM to judge whether a given field `f` satisfies the property `c1` in question.

Given this new construct, one can similarly write a query to identify misnamed non-`final` fields:

```
from Field f    /* Example Query 3 */
where not f.isFinal() and not LLMQuery(f.getName(), "name
  starts with a lowercase letter and the first letters of
  subsequent words are capitalized"/* c2 */)
select f, "field violates naming convention"
```

The `LLMQuery` construct is fully integrated into the grammar of SemQL selection predicates. Consequently, the two queries above can be simply combined into a single query, either by taking the disjunction of their `where` clauses or by using an equivalent `if-then-else` form, as illustrated below:

```
from Field f    /* Example Query 4 */
where if f.isFinal() then not LLMQuery(f.getName(), c1)
      else not LLMQuery(f.getName(), c2)
select f, "field violates naming convention"
```

Identifying statements with side effects. Another example arises when certain optimization settings remove `assert` statements during compilation. The code within these assertions must therefore be free of side effects. A programmer may wish to determine whether there are assertions whose execution alters the program state. Notice that a precise analysis in this case can be costly, especially when statements involve method calls, where a call graph is required alongside data flow tracking. One may instead write the query below to identify assertions with *potential* side effects:

```
from AssertStmt a, Expr e    /* Example Query 5 */
where e = a.getExpr() and
      LLMQuery(e.toString(), "expression has side effect")
select e, "operation may introduce side effects"
```

In this case, the LLM is provided with the text of an expression `e` and is asked to judge whether the expression *might* have side effects. As an example, this query flags a statement with the expression `list.remove(null)` inside. This is because a purely textual reading suggests that this function call is likely to change the state of the variable `list`. On the other hand, for example, the query does not raise a warning about the statement `assert(intVal instanceof Integer)`. We also emphasize that this is *not intended to be a sound analysis*, but rather, to be a *highly scalable proxy* for the kinds of judgments made by programmers as they read the code.

Query evaluation. One approach to evaluating Query 4 would involve starting with a list of *all* fields declared in a given program P. We can forward this list of fields to an LLM and ask it to judge whether each field satisfies criterion `c1` and whether it satisfies `c2`. One can then *materialize* the results of these queries in the form of CodeQL native *predicates*:

```
predicate oracle_p1(string arg) {
      arg in [ /* the list of cases satisfying c1 */ ] }
predicate oracle_p2(string arg) {
      arg in [ /* the list of cases satisfying c2 */ ] }
```

Given these two predicates, one may now write the following standard CodeQL query:

```
from Field f    /* Example Query 6 */
where if f.isFinal() then not oracle_p1(f.getName())
      else not oracle_p2(f.getName())
select f, "field violates naming convention"
```

Of course, Query 6 is *not globally equivalent* to Query 4, as an exhaustive listing of variable names that satisfy criterion `c1` or satisfy criterion `c2` is necessarily infinite. However, since the given program P has finitely many fields, Queries 6 and 4 are indistinguishable in the context of this program.

While this approach streamlines interaction with external oracles, it remains inefficient in terms of oracle usage: An oracle query is required for each unique pair of arguments to `LLMQuery` to determine whether the tuples bound to the first argument satisfy the condition specified by the second. For a program with n fields, this would require asking $2n$ questions of the LLM, which motivates the following more efficient strategy for SemQL query evaluation.

Observe that one can rewrite Query 4 as a DNF query and decompose it into two conjunctive Queries 2 and 3. Next, for each subquery, we can construct a weaker query by eliminating the `LLMQuery` literal. The evaluation of this weaker query will produce a superset of the tuples that the original subquery outputs. We can then submit just the resulting tuples to the LLM oracle to determine the satisfying cases and define the corresponding predicates accordingly.

Here, the first subquery checks `final` fields against `c1`, and the second subquery evaluates only non-`final` fields against `c2`. Since every field is either `final` or non-`final` (but not both), this reduces the number of LLM queries by a factor of 2. Note that the above process yields the minimum number of oracle queries in this example, as the two subqueries produce disjoint sets of tuples. In general, however, this approach can be further optimized, as we now explain.

Consider the following SemQL query, which aims to identify strings that either represent untrustworthy URLs within a codebase or expose private information and credentials:

```
from StringLiteral s    /* Example Query 7 */
where (s.getValue().regexpMatch("(https?)://.*"/* reg1 */)
    and LLMQuery(s.getValue(), "is an untrustworthy url"))
    or LLMQuery(s.getValue(), "exposes private information
    or credentials"/* c3 */)
select s.getValue(), "string literal may break security"
```

Similar to the previous example, Query 7 can be split into two subqueries, which identify string literals representing an untrustworthy URL, and those that expose private information or credentials, respectively. Notice also that the union of the results of these subqueries yields the result set of the original query. Considering this, if a tuple has already been included in the result of a subquery, including it in the output of other subqueries is effectively irrelevant. We can therefore safely exclude such tuples from further oracle consideration, thereby reducing the number of oracle queries.

Here, one can evaluate the LLM over the string tuples that match the regex `reg1` and define the predicate `p1` corresponding to the first `LLMQuery`, given the oracle response. Next, to define the predicate `p2` for `LLMQuery(s.getValue(),c3)`, it suffices to evaluate the LLM oracle over the tuples in the result set of the query below, which reasonably excludes strings that are already identified as untrustworthy URLs:

```
from StringLiteral s    /* Example Query 8 */
where ~~LLMQuery(s.getValue(), c3) and~~
not (s.getValue().regexpMatch(reg1) and p1(s.getValue()))
select s.getValue(), "string literal may break security"
```

Given `p1` and `p2`, one can finally rewrite Query 7 as a CodeQL query by replacing `LLMQuery` calls with these two predicates.

Queries	q	::=	(from $T\,x$ (, $T\,x$)*)? (where ϕ)? select e (, e)*
Formulas	ϕ	::=	$e_1 \gtrless e_2$ | e_1 in $[e_2..e_3]$ | e instanceof T
		|	$e_1.p($ (e_2 (, e_3)*)? $)$ | $p($ (e_1 (, e_2)*)? $)$
		|	forall $(T\,x \mid \phi)$ | exists $(T\,x \mid \phi)$
		|	ϕ_1 and ϕ_2 | ϕ_1 or ϕ_2 | not ϕ | ϕ_1 implies ϕ_2
		|	if ϕ_1 then ϕ_2 else ϕ_3 | LLMQuery (e, STRING)
Types	T	::=	int | boolean | string | $\cdots$
Expressions	e	::=	true | false | INT | STRING | x | $e_1 \pm e_2$ | $\cdots$

Figure 1. The syntax of SemQL queries. Note that LLM queries cannot appear nested under quantifiers.

3 Language and Evaluation Algorithm

Syntax. From a syntactic perspective, SemQL queries are a relatively simple extension of classical CodeQL queries, as shown in Figure 1. The `LLMQuery(e, s)` is the only construct which is added to the standard CodeQL language, where e denotes an expression and s is a string literal. The formula `LLMQuery(m.getName(),"name starts with verbs")` is an example use case of this new construct, where `m` is a variable of type `Method` ranging over all methods in the program.

Semantics, well-typedness, and validity. We only consider well-typed SemQL queries, where all variables, expressions, and formulas can be associated with a type, and whose occurrences are consistent with these typing constraints. In subformulas of the form `LLMQuery(e, s)`, we require e to be an expression of type `string`. In addition, every CodeQL query, upon evaluation, must necessarily evaluate to a finite set of tuples. To statically provide this guarantee, the CodeQL compiler (and therefore SemQL) imposes a system of variable bindings and validity checks. Intuitively, every variable with a non-primitive (user-defined) type is *bound* (i.e., has finite range), and formulas of the form $x \in [e_1..e_2]$ bind the variable x when e_1 and e_2 are themselves both bound. Bindings naturally propagate through the boolean connectives: For example, the formula ϕ_1 `and` ϕ_2 binds all variables that are bound either by ϕ_1 or ϕ_2. Similarly, ϕ_1 `or` ϕ_2 binds all variables that are bound both by ϕ_1 and by ϕ_2. We call a query *valid* when all of its variables are bound by its `where` clause.

Now, it is relatively straightforward to define the semantics of a valid, well-typed SemQL query: Consider all possible tuples using variable valuations within their statically defined ranges, filter those tuples which satisfy the `where` clause, and print the fields chosen by the `select` clause. The truth values of formulas of the form `LLMQuery(e, s)` are obtained by asking the LLM whether the values of the expression e satisfy the condition s. We write $[\![Q]\!](P)$ for the set of output tuples obtained upon evaluating Q over some codebase P.

3.1 Query Evaluation

Given a SemQL query Q, we first convert its formula ϕ into DNF form, $\phi = C_1$ `or` C_2 `or` $\cdots$ `or` C_n, while treating all of its quantified subformulas, `forall` $(T\,x \mid \phi)$ and `exists` $(T\,x \mid \phi)$, as atomic constraints. Here, each term C_i is a conjunction of atomic conditions that must simultaneously hold for a tuple to satisfy that term. Naturally, the result of evaluating the original query, $[\![Q]\!](P)$, satisfies

$$[\![Q]\!](P) = [\![Q_1]\!](P) \cup [\![Q_2]\!](P) \cup \cdots \cup [\![Q_n]\!](P),$$

where Q_i is obtained by replacing the entire formula ϕ of the `where` clause of Q with the individual term C_i. Notice that if the entire query Q is valid and well-typed, then each Q_i is also valid and well-typed.

The procedure of Algorithm 1 builds upon the fact that $a \vee b$ is logically equivalent to $a \vee \overline{a}b$. More generally,

$$C_1 \vee C_2 \vee \cdots \vee C_n \equiv C_1' \vee C_2' \vee \cdots \vee C_n',$$

where $C_i' = (\neg C_1 \wedge \neg C_2 \wedge \cdots \wedge \neg C_{i-1}) \wedge C_i$. We therefore construct a sequence of intermediate queries, Q_i', each of which produces the set of tuples satisfying C_i'. Once again, we have $[\![Q]\!](P) = [\![Q_1']\!](P) \cup [\![Q_2']\!](P) \cup \cdots \cup [\![Q_n']\!](P)$. The principal remaining problem is identifying which LLM queries to issue, and when to issue them.

Progressive materialization of LLM queries. Our goal is to replace every oracle query $q =$ `LLMQuery(e, s)` with a materialized predicate p_q. In both CodeQL and SemQL, a *predicate* is a *finite* relation. For instance, after examining a program P with the query

$$q = \texttt{LLMQuery(m.getName(),"name starts with verbs")},$$

one might discover the following methods in the codebase:

```
predicate p_q(string s) {
  s in ["addEntry", "setValue", "replaceToken", "print"]}
```

Algorithm 1 SemQLEval(Q, P). $Q =$ `from` $v_1, v_2, \ldots, v_k$ `where` ϕ `select` $x_1, x_2, \ldots, x_m$ is a valid, well-typed SemQL query and P is the program to which it is being applied.

1. Assume $\phi = C_1$ `or` C_2 `or` $\cdots$ `or` C_n in DNF form, arranged in increasing order of the number of `LLMQuery` literals.
2. Initialize the predicate $p_q := \emptyset$ for each unique subformula q of the form `LLMQuery(e, s)`.
3. For each C_i, in order:
   a. Let $C_i = D$ `and` F, where D denotes its subset of non-`LLMQuery` literals and $F = \bigwedge_{1 \le j \le h} q_j$ `and` $\bigwedge_{h+1 \le j \le l} \overline{q_j}$ is the subset of `LLMQuery` literals, with $q_j =$ `LLMQuery` (e_j, s_j) for all $1 \le j \le l$.
   b. Define $\Delta = \bigwedge_{j<i} \overline{C_j}$ and where all `LLMQuery` atoms have been replaced with their corresponding materializations.
   c. Print the set of "*relevant*" queries as $[\![Q_i^p]\!](P)$, where:
   $$Q_i^p = \texttt{from}\ v_1, v_2, \ldots, v_k\ \texttt{where}\ D\ \texttt{and}\ \Delta\ \texttt{select}\ e_1, e_2, \ldots, e_l.$$
   d. For each $q_j =$ `LLMQuery` (e_j, s_j) in F:
      i. Construct the predicate $p^i_{q_j}$ by asking the LLM whether each value in the j-th column of $[\![Q_i^p]\!](P)$ satisfies the condition s_j.
      ii. Update $p_{q_j} := p_{q_j} \cup p^i_{q_j}$.
   e. Define $R_i = D$ `and` $\bigwedge_{1 \le j \le h} p_{q_j}(e_j)$ `and` $\bigwedge_{h+1 \le j \le l} \overline{p_{q_j}(e_j)}$.
4. Define the classical CodeQL query:
   $$Q_{\text{fin}} = \texttt{from}\ v_1, v_2, \ldots, v_k\ \texttt{where}\ \bigvee_{1 \le i \le n} R_i\ \texttt{select}\ x_1, x_2, \ldots, x_m.$$
5. Return $[\![Q_{\text{fin}}]\!](P)$.

Here, p_q is a (necessarily finite) listing of method names *in the present program P* that start with a verb. If an exhaustive listing of p_q is available, we would have $[\![Q]\!](P) = [\![Q_{\text{fin}}]\!](P)$, where Q_{fin} is a standard CodeQL query obtained by replacing every occurrence of $q = \texttt{LLMQuery}(e, s)$ in Q with the formula $p_q(e)$. Because the truth of q is a function of the string-valued expression e, the materialization predicate p_q is always a unary (i.e., arity-1) relation over `string`-s. By abuse of notation, we also use p_q synonymously with the set of strings that it matches (in this case, {`"addEntry"`, `"print"`, ...}).

Algorithm 1 starts by initializing each of these materialization predicates with an empty set, $p_q = \emptyset$, and scans the conjunctive terms, $C_1, C_2, \ldots, C_n$ of ϕ in increasing order of the number of `LLMQuery` literals. After scanning each term C_i, it issues the relevant LLM queries and updates p_q while maintaining the invariant that $[\![Q_1]\!](P) \cup [\![Q_2]\!](P) \cup \cdots \cup [\![Q_i]\!](P) = [\![Q_{1,\text{fin}}]\!](P) \cup [\![Q_{2,\text{fin}}]\!](P) \cup \cdots \cup [\![Q_{i,\text{fin}}]\!](P)$.

The algorithm exploits knowledge of the previous conjunctive terms C_j, for $j < i$, to prune the set of LLM queries that need to be issued while processing C_i. This optimization, involving the incremental formula Δ is broadly similar to the semi-naive evaluation algorithm for Datalog [11].

The set of LLM queries that need to be discharged is progressively computed using the classical CodeQL queries Q_i^p and is resolved using a prompt similar to the following:

> **Task:** Identify if each given case satisfies the following condition:
> **Condition:** Name starts with verbs.
> **Cases:** addEntry, setValue, dataProcessing, print, replaceToken, ...
> **Output format:** Only output all the cases that satisfy the condition. Each satisfying case must be printed on its own line. Do not include extra text.

4 Experimental Evaluation

Our evaluation sought to answer the following questions:

RQ1. How fast can SemQL queries be evaluated?
RQ2. How heavily does the algorithm use the oracle?

Benchmarks. By consulting the bug descriptions in SpotBugs documentation [8] and CodeQueries dataset [26], we identified 7 bad coding practices in Java which may lead to confusion or introduce bugs to software systems. We then crafted SemQL queries which enable automatic detection of such cases and evaluated these queries on the SpotBug's test codebase [9] involving 74.5K lines of Java code. Table 1 provides detailed information about our benchmarks.

Experimental setup and baseline. We conducted our experiments on a workstation machine with an AMD Ryzen 9 5950X CPU and 128 GB of memory running Ubuntu 22.04. We also used GPT-4o-mini [12] as the LLM oracle to evaluate queries. We compare our DNF-based query evaluation algorithm to the naive strategy discussed in Section 2. For every expression occurring as the first argument in an oracle call, the naive algorithm simply collects all tuples bound to that expression and evaluates the LLM oracle over these tuples.

4.1 RQ1: Temporal Efficiency of Query Evaluation

To measure the efficiency of our query evaluation algorithm, we compared its processing time and that of the baseline for each SemQL query and present the results in Table 1.

The query processing time mainly depends on two factors: (*a*) The time needed for interaction with the oracle, which itself depends on the number of tuples requiring oracle evaluation, and (*b*) the number of intermediate queries to extract these tuples. The baseline algorithm produces only one intermediate query that includes expression arguments of oracle invocations in the `select` clause. In contrast, our query evaluation algorithm operates on a DNF formula, where the number of intermediate queries corresponds to the number of DNF terms with oracle literals (see Line 3c of Algorithm 1).

Observe that in most cases, our query evaluation algorithm requires significantly less time to process the given query and generate its result set compared to the baseline. As expected, the processing time increases with the number of tuples passed to the oracle and the number of intermediate queries evaluated.

In the case of Query 7, which includes a single oracle literal, our algorithm behaves identically to the baseline. Therefore, the elapsed time is expected to be the same in this case.

4.2 RQ2: Economic Efficiency of Query Evaluation

We also measured how heavily the two algorithms used the oracle in terms of input tuples submitted, financial cost, and the time spent within the oracle, as reported in Table 1.

Observe that our query evaluation algorithm significantly reduces the oracle usage both in terms of time needed for the oracle evaluation and the number of tokens which are submitted to the oracle. Overall, our query evaluator submits on average 81% fewer tuples to the LLM oracle than the baseline. This causes the baseline algorithm to spend 3.8× more time inside the oracle.

Given the total number of output and input tokens (input tuples + prompt tokens), evaluating all benchmark queries on the SpotBugs codebase costs only 0.74¢ using our DNF-based algorithm, compared to 3.72¢ (≈ 5× higher) with the baseline. For this set of queries, we used GPT-4o-mini, a cost-effective model that performs adequately for our LLM-based analysis. The cost gap would likely widen if a more advanced model were required to handle more complex queries.

Note that LLM oracles are inherently non-deterministic. This may affect both the oracle evaluation time and the number of output tokens which are identified as satisfying cases. The latter may also influence the number of input tokens that need oracle evaluation in the subsequent steps. Although we observed that this variation across different runs is minor and negligible, we adopted two strategies to alleviate this issue and reduce its threat to our evaluation results: First, we sorted the resulting tuples of intermediate queries lexicographically. This ensures that the sequence of cases which

Table 1. Our SemQL query benchmarks and evaluation results. The best performing approach in each case is marked in bold. All reported times are in seconds, and the column labeled "Query Size" indicates the number of unique atoms in ϕ.

ID	Detection Target	Query Size, $\lvert\phi\rvert$	DNF Terms	Baseline			DNF-based Algorithm			
				Oracle Time	Input Tuples	Total Time	Oracle Time	Input Tuples	Total Time	Inter. Queries
1	Methods violating naming convention	4	3	1,189	28,180	1,208	**146**	**4,604**	**172**	2
2	Classes violating naming convention	6	4	863	16,766	881	**176**	**4,445**	**213**	3
3	Fields violating naming or access modifier convention	8	4	865	17,397	891	**528**	**3,797**	**561**	2
4	Invalid regular expressions	8	3	87	2,721	101	**2**	**22**	**23**	3
5	Untrustworthy URLs	2	1	58	2,721	71	**1**	**6**	**7**	1
6	Assert statements with side effect	2	1	406	11,574	421	**1**	**18**	**12**	1
7	String literals exposing private information	1	1	**70**	**2,721**	**84**	**70**	**2,721**	**84**	1

are submitted to the oracle remains nearly identical across different runs. Second, we processed each query 3 times using the two algorithms and reported, for each algorithm, the average number of input tokens, the average processing time, and the average oracle time across these 3 runs.

5 Discussion and Future Work

Alternative formalisms. We modeled `LLMQuery(e, s)` as a binary predicate. One can alternatively consider this construct as a unary predicate taking an arbitrary string expression, which directly encodes the condition alongside the tuples requiring oracle evaluation, for example:

```
LLMQuery("Does " + m.getName() + " start with verbs?").
```

This may increase the expressive power of the query language. However, the batch oracle queries described in the LLM prompt in Section 3 would no longer be applicable. Instead, individual queries would need to be submitted to the LLM for each tuple, significantly impacting performance.

Reasoning accuracy. Although the accuracy of the output tuples generated by SemQL queries for answering analytic questions is important, we believe it is largely influenced by prompt design and by how the desired conditions are specified in the `LLMQuery` construct. For instance, incorporating illustrative examples directly into the condition can help the LLM resolve ambiguities when determining satisfying cases. Therefore, we do not measure tuple-level accuracy in this work and leave this evaluation to future research.

Comparison to coding agents. Our approach can be seen as the conceptual reverse of agentic systems such as Claude Code [1], which are fundamentally LLM-centric pipelines augmented with system commands and tools. In those systems, the LLM drives the overall reasoning process, making the entire analysis workflow inherently probabilistic. In contrast, our architecture is CodeQL-centric: the core analysis is performed by CodeQL, which operates over formal program representations (e.g., ASTs, CFGs) that ensure reproducibility and structural grounding. We delegate only semantic or interpretive judgments to the LLM. As a result, any potential hallucination or probabilistic behavior is localized to a narrow auxiliary layer, while the main analytical pipeline remains deterministic and formally grounded. This significantly reduces fuzziness compared to agentic approaches, preserves stronger guarantees and soundness, and makes the system more suitable for rigorous program analysis tasks.

6 Related Work

A large body of research has explored the use of large language models for program analysis. Prior work leverages LLMs for static analysis [13, 15, 18, 19, 23], symbolic execution [16, 21], and abstract interpretation [22], either as standalone reasoning engines or coupled with traditional analysis tools to refine warnings and reduce false alarms [30].

In contrast, we extend a declarative query language itself with a new construct that allows the analysis framework to use external information beyond the facts derived solely from the codebase. Although we use LLM oracles in this paper, the extended query language and the query evaluation process are independent of the underlying oracle and can be easily adapted to use other types of oracles as well.

By extending SQL with LLM call operators, Glenn et al. introduced BlendSQL [17], a declarative query language which integrates LLM invocations directly into its relational query plan. Although BlendSQL is conceptually similar to SemQL, it notably builds on the open-source SQLite database engine, incorporating LLM-backed operators directly into the optimizer and execution pipeline, and therefore does not face the constraints of a closed-source query evaluation framework.

7 Conclusion

We introduced SemQL, a declarative language that extends CodeQL with oracle invocations to enable richer semantic reasoning over codebases. By integrating LLM queries into the query language and leveraging propositional abstraction and DNF-based query normalization, we provide a principled strategy for evaluating SemQL queries while minimizing oracle usage. Our prototype implementation and the benchmarks are available at github.com/SaraBaradaran/SemQL.

Acknowledgments

This research was supported in part by the U.S. National Science Foundation (NSF) under grant CCF-2146518.

References

[1] [n. d.]. *Claude Code Documentation.* https://code.claude.com/docs

[2] [n. d.]. *Comby: A Tool for Searching and Changing Code Structure.* https://comby.dev

[3] [n. d.]. *Oracle Java Documentation.* https://docs.oracle.com/javase/tutorial/java/nutsandbolts/variables.html

[4] [n. d.]. *QL Language Specification.* https://codeql.github.com/docs/ql-language-reference/ql-language-specification

[5] [n. d.]. *Repository of the paper CodeQueries: A Dataset of Semantic Queries over Code.* https://github.com/thepurpleowl/codequeries-benchmark/blob/main/resources/codequeries_meta.json

[6] [n. d.]. *Semgrep: Lightweight Static Analysis for Many Languages.* https://semgrep.dev

[7] [n. d.]. *Server Side Request Forgery (SSRF).* https://owasp.org/www-community/attacks/Server_Side_Request_Forgery

[8] [n. d.]. *SpotBugs bug descriptions.* https://spotbugs.readthedocs.io/en/latest/bugDescriptions.html

[9] [n. d.]. *SpotBugs Java Codebase.* https://github.com/spotbugs/spotbugs/tree/master/spotbugsTestCases/src/java

[10] [n. d.]. *Unvalidated Redirects and Forwards.* https://owasp.org/www-community/attacks/open_redirect

[11] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of databases.* Vol. 8. Addison-Wesley Reading.

[12] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. GPT-4 Technical Report. *arXiv preprint arXiv:2303.08774* (2023). doi:10.48550/arXiv.2303.08774

[13] Vaibhav Agrawal and Kiarash Ahi. 2025. LLM-Driven SAST-Genius: A Hybrid Static Analysis Framework for Comprehensive and Actionable Securit. *arXiv preprint arXiv:2509.15433* (2025). doi:10.48550/arXiv.2509.15433

[14] Pavel Avgustinov, Oege de Moor, Michael Peyton Jones, and Max Schäfer. 2016. QL: Object-oriented Queries on Relational Data. In *Proceedings of the 30th European Conference on Object-Oriented Programming (ECOOP '16).* Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2:1–2:25. doi:10.4230/LIPIcs.ECOOP.2016.2

[15] Patrick J. Chapman, Cindy Rubio-González, and Aditya V. Thakur. 2024. Interleaving Static Analysis and LLM Prompting. In *Proceedings of the 13th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis (SOAP '24).* Association for Computing Machinery, New York, NY, USA, 9–17. doi:10.1145/3652588.3663317

[16] Jiachi Chen, Zhenzhe Shao, Shuo Yang, Yiming Shen, Yanlin Wang, Ting Chen, Zhenyu Shan, and Zibin Zheng. 2025. NumScout: Unveiling Numerical Defects in Smart Contracts Using LLM-Pruning Symbolic Execution. *IEEE Trans. Softw. Eng.* 51, 5 (2025), 1538–1553. doi:10.1109/TSE.2025.3555622

[17] Parker Glenn, Parag Dakle, Liang Wang, and Preethi Raghavan. 2024. BlendSQL: A Scalable Dialect for Unifying Hybrid Question Answering in Relational Algebra. In *Findings of the Association for Computational Linguistics (ACL '24).* Association for Computational Linguistics, Bangkok, Thailand, 453–466. doi:10.18653/v1/2024.findings-acl.25

[18] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. 2023. Assisting Static Analysis with Large Language Models: A ChatGPT Experiment. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23).* Association for Computing Machinery, New York, NY, USA, 2107–2111. doi:10.1145/3611643.3613078

[19] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. 2024. Enhancing Static Analysis for Practical Bug Detection: An LLM-Integrated Approach. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 111 (2024), 26 pages. doi:10.1145/3649828

[20] Jiawei Li, Yang Gao, Yizhe Yang, Yu Bai, Xiaofeng Zhou, Yinghao Li, Huashan Sun, Yuhang Liu, Xingpeng Si, Yuhao Ye, Yixiao Wu, Yiguan Lin, Bin Xu, Bowen Ren, Chong Feng, and Heyan Huang. 2025. Fundamental Capabilities and Applications of Large Language Models: A Survey. *ACM Comput. Surv.* 58, 2, Article 38 (2025), 42 pages. doi:10.1145/3735632

[21] Yihe Li, Ruijie Meng, and Gregory J. Duck. 2025. Large Language Model Powered Symbolic Execution. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 385 (2025), 29 pages. doi:10.1145/3763163

[22] Jacqueline Mitchell, Brian Hyeongseok Kim, Chenyu Zhou, and Chao Wang. 2025. Understanding Formal Reasoning Failures in LLMs as Abstract Interpreters. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Language Models and Programming Languages (LMPL '25).* Association for Computing Machinery, New York, NY, USA, 71–83. doi:10.1145/3759425.3763389

[23] Mohammad Mahdi Mohajer, Reem Aleithan, Nima Shiri Harzevili, Moshi Wei, Alvine Boaye Belle, Hung Viet Pham, and Song Wang. 2024. Effectiveness of ChatGPT for Static Analysis: How Far Are We?. In *Proceedings of the 1st ACM International Conference on AI-Powered Software (AIware '24).* Association for Computing Machinery, New York, NY, USA, 151–160. doi:10.1145/3664646.3664777

[24] Oege de Moor, Mathieu Verbaere, Elnar Hajiyev, Pavel Avgustinov, Torbjorn Ekman, Neil Ongkingco, Damien Sereni, and Julian Tibble. 2007. Keynote Address: .QL for Source Code Analysis. In *Proceedings of the 7th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM '07).* 3–16. doi:10.1109/SCAM.2007.31

[25] Serge Lionel Nikiema, Jordan Samhi, Abdoul Kader Kaboré, Jacques Klein, and Tegawendé F Bissyandé. 2025. The Code Barrier: What LLMs Actually Understand? *arXiv preprint arXiv:2504.10557* (2025). doi:10.48550/arXiv.2504.10557

[26] Surya Prakash Sahu, Madhurima Mandal, Shikhar Bharadwaj, Aditya Kanade, Petros Maniatis, and Shirish Shevade. 2024. CodeQueries: A Dataset of Semantic Queries over Code. In *Proceedings of the 17th Innovations in Software Engineering Conference (ISEC '24).* Article 7, 11 pages. doi:10.1145/3641399.3641408

[27] Sijia Shen, Feiyan Jiang, Peiyan Wang, Yubo Feng, Yuchen Jiang, and Chang Liu. 2025. Do LLMs Know and Understand Domain Conceptual Knowledge?. In *Findings of the Association for Computational Linguistics (EMNLP '25).* Association for Computational Linguistics, Suzhou, China, 5967–5976. doi:10.18653/v1/2025.findings-emnlp.319

[28] Chia-Yi Su and Collin McMillan. 2026. Do Code LLMs Do Static Analysis? *Empirical Software Engineering* 31, 5 (2026), 116. doi:10.1007/s10664-026-10853-z

[29] Jiayimei Wang, Tao Ni, Wei-Bin Lee, and Qingchuan Zhao. 2025. A Contemporary Survey of Large Language Model Assisted Program Analysis. *arXiv preprint arXiv:2502.18474* (2025). doi:10.48550/arXiv.2502.18474

[30] Cheng Wen, Yuandao Cai, Bin Zhang, Jie Su, Zhiwu Xu, Dugang Liu, Shengchao Qin, Zhong Ming, and Tian Cong. 2024. Automatically Inspecting Thousands of Static Bug Warnings with Large Language Model: How Far Are We? *ACM Trans. Knowl. Discov. Data* 18, 7, Article 168 (2024), 34 pages. doi:10.1145/3653718

Received 10 March 2026; accepted 15 April 2026