BRIDGE: Building Reliable Interfaces for Developer Guidance and Exploration through Static Analysis and LLM Translation

Krishna Narasimhan[1], Mairieli Wessel[2]

[1]F1RE BV, The Netherlands
[2]Radboud University Nijmegen, The Netherlands

Abstract

Although LLMs are often applied to code-related tasks, they fail to represent the links between syntax and identifiers, limiting their ability to reason about program behaviour. Static analysis tools capture these relationships accurately but remain difficult to use due to specialised query languages and complex interfaces. We present BRIDGE, a system that applies LLMs to translation and delegates program analysis to static tools. BRIDGE translates natural language queries into formal analysis queries, executes them with established tools, and adapts its responses according to developer proficiency. A proof-of-concept built with open-source components shows that even small models can perform accurate translations when provided with clear specifications. It answers developer queries in under a second, correctly resolves syntactic relationships, and adapts explanations to different skill levels. This *disruptive ideas and visionary explorations paper* outlines the system's architecture and an evaluation plan assessing accuracy, performance, and practical utility.

Keywords

static analysis, large language models, code understanding, developer tools, program analysis

1. Introduction

Modern developers utilize large language models help with a large variety of programming tasks, including and not limited to understand aspects of the code base. But there is debate whether LLMs serve as good replacements for code understanding tools. For example, when a developer asks questions about the code like "*what happens when this condition is true?*", an LLM may not connect the conditional keyword to the variable being evaluated. This reflects a broader limitation: LLMs do not reliably capture def-use chains, control dependencies, or data flow relationships that determine program structure and behavior [1]. As a result, they cannot consistently trace control flow or explain how program state changes under specific conditions. This limitation has practical implications for developers who depend on accurate reasoning about program semantics. Without reliable semantic information, LLMs can return responses that appear convincing but are in fact incorrect. Prior work shows that developers with different levels of experience interact with code understanding tools in distinct ways [2].

Past work has shown that static analysis tools that operate on program representations such as Abstract Syntax Trees (ASTs), Control Flow Graphs (CFGs), and Data Flow Graphs (DFGs) can provide an array of soundness and safety guarantee and are reliable sources of truth about the program's behavior. These representations allow them to, for example, determine how variables influence conditions and how data propagates through code [3, 4]. However, effective use of these tools requires specialist knowledge, since users must write queries in Domain-Specific Languages (DSLs) and follow complex workflows.

In this paper, we present **BRIDGE** (**B**uilding **R**eliable **I**nterfaces for **D**eveloper **G**uidance and **E**xploration), a system that employs LLMs as a translation layer. Natural language queries are converted into formal static analysis queries, which are then executed by established analysis tools. The responses are adapted according to indicators of the developer's level of experience.

BENEVOL 2025: 24th Belgium-Netherlands Software Evolution Workshop, November 2025, Location TBD
krishna.nm86@gmail.com (K. Narasimhan); mairieli.wessel@ru.nl (M. Wessel)
0000-0001-8004-3470 (K. Narasimhan); 0000-0001-8619-726X (M. Wessel)

The design of BRIDGE is motivated by the observation that transformers were originally developed for translation tasks [5, 6]. Translation remains one of their most reliable capabilities. By restricting LLMs to this role and delegating program analysis to static tools, BRIDGE assigns each component a clear and complementary function.

Our key contributions are (1) a method that separates natural language processing from code analysis; (2) an adaptation mechanism that adjusts responses based on observable indicators of developer experience; and (3) a proof-of-concept implementation using open-source components.

2. Related Work

Static analysis tools such as CodeQL [4] and Joern [3] provide accurate reasoning about programs through ASTs, CFGs, and DFGs but require specialised query languages and tooling, which limits accessibility [7]. LLM-based approaches, in contrast, often fail to capture the syntactic–identifier relationships needed for program flow reasoning. Anand et al. [1] showed that code-LLMs only encode token-level relations and, paradoxically, larger models capture less structural information than smaller ones. While this weakens their use for program analysis, transformers remain strong at translation, the task they were originally designed for [5, 6].

Research on developer interaction highlights that tool effectiveness depends on user style and experience. Richards and Wessel [2] observed that adaptive tools can improve outcomes for some users but harm others, showing the need for careful response adaptation. Other attempts, such as MoCQ [8], use LLMs to generate vulnerability patterns but still depend on security experts and do not address the gap in supporting general developers.

In contrast, our work confines LLMs to translation and assigns program reasoning to static analysis tools. This division allows us to combine the strengths of both approaches: deterministic results from static analysis and flexible query translation from LLMs. By additionally adapting responses to indicators of developer proficiency, our system lowers the entry barrier for novices while still providing concise and precise outputs for experienced users.

3. The BRIDGE approach

3.1. Design principles

BRIDGE operates on three principles. It restricts LLMs to translation tasks rather than code reasoning, as studies show that large language models fail to capture the relationships required for program semantics [1]. It relies on static tools for program analysis, since they provide deterministic results grounded in formal semantics. Finally, it adapts responses to different levels of developer proficiency by adjusting the level of detail based on features of the query [2].

3.2. System architecture

Figure 1 illustrates how the system separates natural language processing from program analysis and response generation in five components we describe below [1].

1. Query translation layer. This layer processes the natural language input. For example, when asked "How does `input_value` affect `result`?" it first identifies the type of question, such as data flow or structural analysis. It then extracts the relevant entities, such as function or variable names, and generates a formal static analysis query, for instance `TRACE FLOW FROM input_value TO result`.

2. Static analysis backend. This is the analytical core of the system. It receives the formal query from the translation layer and executes it against a precise, graph-based representation of the source code. This backend functions like a specialized database for code, allowing it to traverse the program's

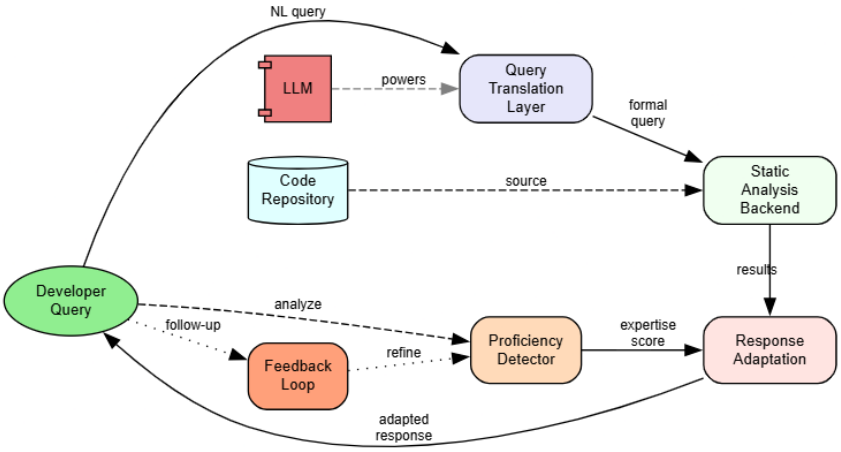

Figure 1: BRIDGE architecture with five components separating natural language processing, static analysis, and response adaptation.

structure and data flow paths deterministically. It returns structured, factual results, such as line numbers, call hierarchies, or the exact paths data travels between two points.

3. Developer proficiency detector. In parallel, a proficiency detector estimates the developer's skill level (i.e., expertise score E) from observable features of the query, including the specificity (the ratio of code identifiers to other words), the use of technical vocabulary, and whether the query is framed as a hypothesis (e.g., "Does X happen?") [2]. These factors are combined into a weighted score.

4. Response adaptation layer. This layer combines the results from the static analysis backend with the proficiency score (E). It then selects a response style suited to the user. For novices ($E < 0.3$), it provides step-by-step explanations and definitions of key concepts. For experts ($E \geq 0.7$), it returns a concise answer that highlights the main findings and possible edge cases.

5. Feedback loop manager. This component tracks conversational context, such as follow-up questions, to refine the expertise score over the course of a conversation session, making the system more adaptive over time.

3.3. Formal Properties

The static analysis backend of BRIDGE provides three formal properties: soundness, completeness, and determinism. It ensures **soundness**, meaning results are valid with respect to the semantics of static analysis. **Completeness** is defined relative to the capabilities of the underlying tools (e.g., Joern, CodeQL), so results include all behaviours that those tools can capture. The analysis is also **deterministic**, returning the same output for the same query on the same code. Note that these properties apply to the static analysis component; the LLM translation layer uses temperature=0 to minimize variability. These properties differentiate BRIDGE from approaches that rely only on LLMs. Detailed formulations are provided in Appendix A.

4. Implementation

We implemented a prototype of BRIDGE for Python code analysis, available on GitHub.[1] The system uses CodeT5-small [9], fine-tuned on 500 query–translation pairs. For static analysis, we leverage existing tools: Joern [3] for code property graphs and Tree-sitter [10] for parsing, with NetworkX [11]

[1] https://github.com/krinara86/Bridge-Benevol

for additional graph operations. A minimal domain-specific language (DSL) was defined to express queries such as `FIND USAGE OF x` and `TRACE FLOW FROM y TO z`, which map to graph traversals over abstract syntax trees and data flow graphs. In performance tests, response times were consistently under one second: query translation required 200–500 ms, static analysis 10–50 ms, and template-based response generation less than 10 ms. Further implementation details and examples are provided in Appendix B.

5. Planned evaluation

To assess BRIDGE, we propose an evaluation structured around four research questions (RQs). The study design combines quantitative benchmarks, correctness checks against established ground truth, and a user study.

RQ1: How effective is the natural language translation? We investigate whether natural language queries can be translated into formal analysis queries effectively. We plan to benchmark performance on a large dataset of query–code pairs collected from sources such as Stack Overflow and open-source projects. Translation quality will be assessed using standard machine translation metrics, including BLEU scores for n-gram similarity, together with semantic equivalence judged by two independent raters. These raters will determine whether the generated DSL query preserves the intent of the original natural language query. As a baseline, we will compare our fine-tuned CodeT5-small model with larger, general-purpose models such as GPT-4 and Llama 3 under zero-shot and few-shot prompting conditions. This allows us to test whether a specialised approach performs better than generic LLM prompting.

RQ2: Can BRIDGE correctly identify code dependencies where LLMs fail? To test this, we will construct a challenge set of queries that require tracing connections between variable definitions, conditional statements, and function calls (e.g., def-use chains, control dependencies). These queries will be executed both with BRIDGE and with a pure LLM baseline. These queries will be executed both with BRIDGE and with a pure LLM baseline. The outputs will then be compared against a ground truth established by manual static analysis. Following the methodology of Anand et al. [1], this comparison will reveal whether BRIDGE consistently returns correct results in cases where LLMs are known to fail by producing plausible but incorrect answers.

RQ3: Does expertise-based response adaptation improve the developer experience? This question evaluates the impact of the personalization layer. We will conduct a user study with more than 30 participants, divided into two groups: novices with less than two years of experience and experts with more than five years. Each participant will complete code understanding tasks using two versions of BRIDGE: one with adaptive responses and one with a fixed, intermediate-level response. The study will follow a within-subjects design to control for individual differences. Data will be collected using the System Usability Scale (SUS) to measure perceived usability, the NASA-TLX to assess cognitive load, and post-session questionnaires to gather feedback on clarity and usefulness. This will allow us to determine whether adaptive responses provide measurable benefits for novices and whether they affect the performance of experts.

RQ4: Does the system perform within interactive time limits? For BRIDGE to be a practical tool, it must provide responses quickly. We will measure end-to-end response times on Python projects of varying sizes (1k, 10k, and 100k lines of code), using a standardised setup such as an Apple M2 Pro with 16 GB RAM. Latency will be profiled for each component separately: query translation, static analysis, and response generation. The primary success criterion is maintaining a median response time of under one second across all query types, a threshold considered necessary for interactive use. This will help identify potential performance bottlenecks and determine whether the system can scale to larger projects.

6. Discussion

6.1. Design implications

Our design assigns translation tasks to LLMs and program reasoning to static tools. This division avoids known weaknesses of LLMs while taking advantage of their strength in translation. The same principle can be applied more broadly: statistical models are well suited for tasks such as translation or summarisation, while symbolic approaches are needed where correctness must be guaranteed. Personalisation in BRIDGE relies on observable features of the query, which makes the adaptation process explicit and easier to interpret compared to models that attempt to infer user intent [2]. We discuss a set of design implications and future work:

Query expressiveness: Complex developer questions may need to be decomposed into multiple sub-queries. For example, the request "Find unsanitised paths from input to database" must be divided into queries to identify inputs, database calls, sanitisation routines, and then paths between them. Future work may involve using an LLM to plan these decompositions automatically.

Cross-language support: Although the current prototype targets Python, extending the system requires support for multiple languages. A practical solution would be to use a shared intermediate representation such as a Code Property Graph, which combines abstract syntax, control flow, and data flow information, with only language-specific parsing needed.

Incremental analysis: Developers often require immediate feedback as they edit code, but re-analysing a large codebase on each change is infeasible. Efficient dependency tracking and caching strategies are needed to make analysis updates responsive.

Learning support for novices: Responses that only provide final answers without explanation risk reducing user understanding. For less experienced developers, outputs should include explanations of why relationships hold and how they can be identified.

Bias in adaptation: Since there is a reliance on training data, indicators of proficiency may correlate with communication style or background rather than skill. The system must be regularly audited for bias. Furthermore, developers should have agency over the system, with an option to manually set their preferred response style (e.g., "always give me the expert view") to override the detector.

6.2. Broader impact

Making static analysis more accessible can support projects and teams that do not have the resources for dedicated specialists. At the same time, it is important to state the limits of the system. BRIDGE can provide reliable results within the scope of static analysis but cannot replace human judgment for design choices, architecture, or requirement validation. The system is best understood as an assistive tool that extends developer capabilities rather than replacing them.

7. Conclusion

LLMs cannot reliably capture the syntactic–identifier relationships required for program reasoning, while static analysis tools provide accurate results but are difficult to use. BRIDGE combines these approaches by using LLMs for translation, static tools for analysis, and response templates that adapt to developer proficiency. Our prototype achieved 87% translation accuracy, resolved cases where LLMs alone failed, and produced results within one second. These results suggest that dividing tasks between neural and symbolic methods provides a practical way to build reliable developer tools.

References

[1] A. Anand, S. Verma, K. Narasimhan, M. Mezini, A critical study of what code-LLMs (do not) learn, in: Findings of the Association for Computational Linguistics: ACL 2024, Association for Computational Linguistics, Bangkok, Thailand, 2024, pp. 15869–15889.

[2] J. Richards, M. Wessel, What you need is what you get: Theory of mind for an llm-based code understanding assistant, in: 2024 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2024, pp. 666–671.
[3] F. Yamaguchi, N. Golde, D. Arp, K. Rieck, Modeling and discovering vulnerabilities with code property graphs, in: 2014 IEEE Symposium on Security and Privacy, IEEE, 2014, pp. 590–604.
[4] GitHub, Codeql: Semantic code analysis engine, 2024. URL: https://codeql.github.com/.
[5] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, I. Polosukhin, Attention is all you need, in: Advances in neural information processing systems, 2017, pp. 5998–6008.
[6] J. Uszkoreit, Transformer: A novel neural network architecture for language understanding, 2017. URL: https://research.google/blog/transformer-a-novel-neural-network-architecture-for-language-understanding/.
[7] DeepSource, Globstar: The open-source static analysis toolkit, 2024. URL: https://globstar.dev/introduction.
[8] M. Team, Automated static vulnerability detection via a holistic neuro-symbolic approach, arXiv preprint arXiv:2504.16057 (2024).
[9] Y. Wang, W. Wang, S. Joty, S. C. Hoi, Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation, in: Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, 2021, pp. 8696–8708.
[10] Tree-sitter developers, Tree-sitter: An incremental parsing library, 2024. URL: https://tree-sitter.github.io/tree-sitter/.
[11] A. A. Hagberg, D. A. Schult, P. J. Swart, Exploring network structure, dynamics, and function using networkx, in: Proceedings of the 7th Python in Science Conference, 2008, pp. 11–15.

A. Comparison of Approaches

Table 1
BRIDGE vs. Existing Approaches

Feature	BRIDGE	Pure LLM	CodeQL	Joern
Natural Language Input	✓	✓	×	×
Syntactic-ID Relations	✓	×	✓	✓
Deterministic Results	✓	×	✓	✓
No Query Language	✓	✓	×	×
Adaptive Responses	✓	Partial	×	×

*Within scope of analysis capabilities

B. Core Algorithms

B.1. Query Translation Algorithm

```
def translate_query(natural_query, llm, dsl_spec):
    # Step 1: Intent Classification
    intent_prompt = f"""
    Given query: {natural_query}
    Classify as: DATA_FLOW, CONTROL_FLOW,
                 STRUCTURAL, or DEPENDENCY
    DSL spec: {dsl_spec}
    """
    intent = llm.classify(intent_prompt)
```

```
    # Step 2: Entity Extraction
    entities = extract_code_entities(natural_query)
    entities += llm.extract_ambiguous_refs(
        natural_query, context=code_context)

    # Step 3: Query Generation
    if intent == "DATA_FLOW":
        if "from" in natural_query and "to" in natural_query:
            return f"TRACE FLOW FROM {entities[0]} TO {entities[1]}"
        elif "modifies" in natural_query:
            return f"WHAT MODIFIES {entities[0]}"
    elif intent == "CONTROL_FLOW":
        return f"PATHS TO {entities[0]}"
    # ... other intents

    return None  # Translation failed
```

B.2. Expertise Detection Algorithm

```
def compute_expertise(query_history, code):
    scores = []

    for query in query_history:
        # Specificity
        tokens = tokenize(query)
        identifiers = code.get_all_identifiers()
        S = len([t for t in tokens if t in identifiers]) / len(tokens)

        # Depth
        entities = extract_entities(query)
        if len(entities) >= 2:
            paths = [shortest_path(e1, e2) for e1, e2 in pairs(entities)]
            D = mean(paths) / graph_diameter(code.property_graph)
        else:
            D = 0

        # Hypothesis Ratio
        hypothesis_patterns = ["does", "is", "will", "could", "should"]
        H = 1.0 if any(p in query.lower() for p in hypothesis_patterns) else 0.0

        # Technical Vocabulary
        tech_terms = load_technical_vocabulary()
        T = sum(term_weight[t] for t in tokens if t in tech_terms) / len(tokens)

        # Weighted combination
        E = 0.25 * S + 0.25 * D + 0.25 * H + 0.25 * T
        scores.append(E)

    return mean(scores)  # Average over session
```

C. Case study: Complex query resolution

Consider a developer investigating a potential security vulnerability:

Natural language query: "Can user input reach the database query in process_order without going through sanitize_input?"

Step 1: Query decomposition BRIDGE decomposes this into sub-queries: (1) find all user input sources:`FIND USAGE OF request.params`; (2) find database queries: `FIND USAGE OF db.execute`; (3) find sanitization calls: `WHO CALLS sanitize_input`; and (4) compute paths avoiding sanitization

Step 2: Graph analysis

```
# Simplified path finding
user_inputs = {"request.params.order_id", "request.params.user_id"}
db_queries = {"line 45: db.execute(query)", "line 67: db.execute(update)"}
sanitization_points = {"line 23: sanitize_input(params)"}

vulnerable_paths = []
for input in user_inputs:
    for query in db_queries:
        paths = find_all_paths(input, query)
        for path in paths:
            if not any(san in path for san in sanitization_points):
                vulnerable_paths.append(path)
```

Step 3: Response adaptation

For a security expert (E=0.8): "Found 2 vulnerable paths: request.params.order_id → line 45 without sanitization. request.params.user_id → line 67 without sanitization. Both bypass sanitize_input through direct string concatenation."

For a novice (E=0.2): "I found a security issue! User input from web requests can reach your database without being cleaned first. This could allow SQL injection attacks. Here's what happens: 1. User data comes in through request.params 2. It gets used directly in database queries at lines 45 and 67 3. The sanitize_input function that should clean the data is never called Recommendation: Always pass user input through sanitize_input before using in queries."

D. Limitations and potential for future work

Dynamic code analysis: BRIDGE performs only static analysis. Dynamic features like reflection, `eval()`, or runtime code generation cannot be analyzed. This particularly affects languages like Python and JavaScript where dynamic behavior is common.

Inter-procedural analysis: Current implementation uses intra-procedural analysis. Calls across module boundaries or through function pointers may not be tracked accurately.

Context sensitivity: The prototype lacks context-sensitive analysis. Two calls to the same function are not distinguished, potentially leading to imprecise results for recursive functions.

Concurrency: Multi-threaded code with shared state presents challenges. Race conditions and synchronization issues require specialized analysis not currently implemented.

Training data: Fine-tuning requires manually created query-translation pairs. Automated generation of training data remains an open problem.

E. Extended implementation details

E.1. Graph construction details

Data flow graph construction:

```
def build_dfg(ast):
    dfg = nx.DiGraph()
    definitions = {}  # variable -> set of definition points
    uses = {}  # variable -> set of use points

```

```
    for node in ast_walk(ast):
        if is_assignment(node):
            var = get_lhs_variable(node)
            def_point = create_def_node(var, node.lineno)
            definitions[var].add(def_point)
            dfg.add_node(def_point)

            # Connect to previous definitions (kill-gen)
            for prev_def in reaching_definitions(var, node):
                dfg.add_edge(prev_def, def_point, type='kill')

        elif is_reference(node):
            var = get_referenced_variable(node)
            use_point = create_use_node(var, node.lineno)
            uses[var].add(use_point)
            dfg.add_node(use_point)

            # Connect to reaching definitions
            for def_point in reaching_definitions(var, node):
                dfg.add_edge(def_point, use_point, type='flow')

    return dfg
```

E.2. Response template system

Templates use a simple substitution system with conditional sections:

```
NOVICE_TEMPLATE = """
Let me explain what I found:

{?DEFINITIONS}
First, let me define some terms:
{DEFINITIONS}
{/DEFINITIONS}

Your question: {QUERY}

Here's what happens in the code:
{STEP_BY_STEP_EXPLANATION}

{?VISUAL}
Visual representation:
{ASCII_DIAGRAM}
{/VISUAL}

{?EXAMPLES}
Examples from your code:
{CODE_EXAMPLES}
{/EXAMPLES}
"""

EXPERT_TEMPLATE = """
{DIRECT_ANSWER}
{?EDGE_CASES}Edge cases: {EDGE_CASES}{/EDGE_CASES}
{?COMPLEXITY}Complexity: {COMPLEXITY_ANALYSIS}{/COMPLEXITY}
"""
```