# When is LLM-Based Program Reasoning Correct?

A Completion Semantics for LLM-Based Code Inference

ZHIYUAN LIU, Nanjing University, China
YIHE LI*, National University of Singapore, Singapore
TREVOR E. CARLSON, National University of Singapore, Singapore
HUIYAN WANG, Nanjing University, China
RUIJIE MENG, CISPA Helmholtz Center for Information Security, Germany
GREGORY J. DUCK, National University of Singapore, Singapore

Due to token and cognitive limits, *Large Language Models* (LLMs) typically perform program reasoning over *incomplete* code fragments/prompts rather than complete programs. Such reasoning therefore must rely on *assumptions* about omitted code and context. As a result, the meaning of an inference over a program fragment is not absolute, but depends on an implicit *completion model* describing how the fragment may be refined into a complete program. In this paper, we introduce *completion semantics* for LLM-based program reasoning. We formalize incomplete programs as denoting a space of possible refinements and define the correctness of existential inferences relative to a completion model. Under this view, a reported bug is correct whenever there exists a completion within the model that witnesses the bug. This perspective explains why many LLM-generated reports are neither simply correct nor incorrect, but instead depend on assumptions about omitted context. We have instantiated our approach in the form of a witness-generation workflow that concretizes completions underlying an inference by constructing executable refinements of the original program fragment. Witnesses serve both as evidence for existential claims and as a mechanism for exposing the assumptions required to support them. We evaluate our approach on real-world LLM-generated bug reports and program-analysis tasks. Our results show that witness generation effectively distinguishes inferences supported by plausible completions from those requiring unrealistic assumptions, providing a practical mechanism for validating reasoning over incomplete programs.

## 1 Introduction

*Large Language Models* (LLMs) are increasingly used for *program reasoning tasks*, such as code review [36, 38], bug detection [24, 26, 27], security auditing [16, 29], and program comprehension [20, 30]. In a typical workflow, an LLM is *prompted* with a task description (e.g., vulnerability detection, assertion violations, or safety conditions), together with a *code fragment* (e.g., a function, module, or program slice) for the LLM to reason over. The LLM will proceed to reason over the fragment and instructions to produce an *outcome*, constructing an explanation, finding a bug, or determining that a property appears to hold. Recent systems demonstrate how LLMs can reason effectively over code, and can identify behaviors that are difficult to capture using traditional program analyses. As a result, LLM-based reasoning has become a central component of modern software engineering tools and coding agents, and forms the basis of many systems [5, 24, 40].

Unlike traditional program analyses, LLMs rarely reason over *complete programs*, due to practical limitations—including token budgets, retrieval boundaries, and cognitive constraints [12, 21]. Rather, LLM-based program reasoning must typically operate on a *partial* view of a larger codebase. As such, the prompt presented to the LLM typically only contains a *fragment* of the relevant program, such as an individual function together with a limited amount of surrounding context. Consequently,

*Joint first author

Authors' Contact Information: Zhiyuan Liu, Nanjing University, Nanjing, China, zhiyuanliu@smail.nju.edu.cn; Yihe Li, National University of Singapore, Singapore, Singapore, yihe.li@u.nus.edu; Trevor E. Carlson, National University of Singapore, Singapore, Singapore, tcarlson@nus.edu.sg; Huiyan Wang, Nanjing University, Nanjing, China, why@nju.edu.cn; Ruijie Meng, CISPA Helmholtz Center for Information Security, Saarbrücken, Germany, meng@cispa.de; Gregory J. Duck, National University of Singapore, Singapore, Singapore, gregory@comp.nus.edu.sg.

```c
int f(struct S *s) {
  int *p = s->a;
  memset(s, 0, sizeof(*s));
  return p[0];  // Returns zero?
}
```

Fig. 1. Example of an incomplete program fragment (e.g., a *prompt*). Here, the answer to the query "*returns zero?*" depends on assumptions made about the definition of `struct S`.

any inference made by the LLM necessarily depends on assumptions about any code that has been excluded from the prompt. In traditional program analysis, correctness is defined with respect to a complete program and its execution semantics. In contrast, an incomplete code fragment, as part of an LLM prompt, does not denote a unique program. The meaning of an inference over an incomplete fragment is therefore unclear: *what does it mean for the LLM's conclusion to be correct?*

### 1.1 Illustrative Example

Figure 1 illustrates a simple example that highlights the problem with reasoning over incomplete program fragments. Here, we consider a simple query: "*does the function* `f(...)` *always return zero?*" Classical program reasoning methods can attempt to answer this query using standard *points-to* (*alias*) and *data-flow* static analysis methods. If `p` and `s` *alias* the same object in memory, then `f(...)` will indeed always return `0`; otherwise, a non-zero value may be returned.

However, the fragment in Figure 1 is *incomplete*. Specifically, the definition of (`struct S`) has been omitted from the prompt, meaning that the answer to the query implicitly depends on *assumptions* about missing context. For example, suppose that the `a` field is an *inline array field*, such as follows:

```c
struct S { int a[3]; };
```

In this case, `a` will be a sub-object of `*s`, meaning that the `memset(...)` operation will overwrite the storage referenced by `p`. Under this assumption, the function *always returns zero*. Alternatively, suppose that the `a` field is a *pointer field*, such as follows:

```c
struct S { int *a; };
```

In this case, `memset(...)` will only overwrite the pointer itself, and leaves any memory referenced by `p` unchanged. Under this assumption, the function *does not return zero*. Thus, the fragment in Figure 1 admits *multiple completions*, and each yields a *different* answer to the same query.

This simple example highlights the fundamental challenge regarding reasoning over partial programs: the answer often depends on the omitted code and context. For example, if we consider standard *data-flow* (*def-use*) analysis over the fragment, the answer is context-dependent:

| **Definition of** `struct S` | Def(`memset`$(s, \ldots)$) | Use(`p[0]`) | **Returns zero?** |
|---|---|---|---|
| `int a[3]` | $\{\ell_S, \ell_S.a[0], \ell_S.a[1], \ell_S.a[2]\}$ | $\{\ell_S.a[0]\}$ | *yes* |
| `int *a` | $\{\ell_S, \ell_S.a\}$ | $\{\ell_H[0]\}$ | *not guaranteed* |

Assuming `s->a` points to a location $\ell_H$ for the second case. Modern LLMs routinely make such assumptions when reasoning over incomplete programs, yet the meaning of a conclusion relative to those assumptions remains largely undefined. The central question of this paper is not whether an LLM's answer is correct in isolation, but *when* an LLM's answer should be considered correct.

### 1.2 Completion Semantics

The preceding example illustrates a more general observation: an incomplete program fragment does not denote a unique executable program, but rather a *space* of possible *completions*. Here, a completion is the original fragment (unmodified) augmented with some additional code context to make a "complete" standalone program, including adding any missing functions, types, macros, globals, an entry-point, libraries, and environment. Each completion therefore corresponds to a particular *interpretation* of the omitted code and context, and each completion may induce different answers to the same query (as illustrated in Section 1.1). Consequently, the correctness of a query

may not be wholly defined with respect to any individual fragment alone. Instead, correctness must be interpreted *relative* to a set of admissible completions.

In this paper, we formalize this intuition through *completion semantics*. At a high level, a completion extends an incomplete program fragment into a self-contained program. A *completion model* then characterizes which such completions are considered *admissible*. Different completion models induce different notions of correctness. For example, one model may admit all *syntactically valid completions*, while another may restrict completions that satisfy type constraints, API contracts, or other assumptions about intended program behavior. Queries are subsequently interpreted relative to the given completion model. Informally, an existential property (e.g., assertion failure) will hold if there exists an admissible element of the completion model witnessing the property, while a universal property holds if *all* admissible completions satisfy it.

Completion semantics are relevant for LLM-based program reasoning, since the boundary between provided and omitted context in prompts is generally unavoidable in practice. While additional context can always be retrieved and appended to a prompt (e.g., the type definition in Section 1.1), for real-world code, this usually shifts the boundary elsewhere in the program. Modern software systems are sufficiently large that modern LLMs cannot effectively reason over an entire codebase and its environment simultaneously. As a result, assumptions about omitted context are not an implementation artifact, but an inherent aspect of LLM-based reasoning itself.

A practical challenge is that completion models are often *implicit*: the assumptions underlying an LLM's reasoning are usually not directly observable. To study these assumptions empirically, we instantiate our semantics using a *witness-generation* workflow that attempts to construct concrete completions supporting a reported inference. Under our semantics, such witnesses serve as evidence for existential claims and expose the assumptions required for an inference to hold. Witness generation is therefore one pragmatic realization of completions semantics.

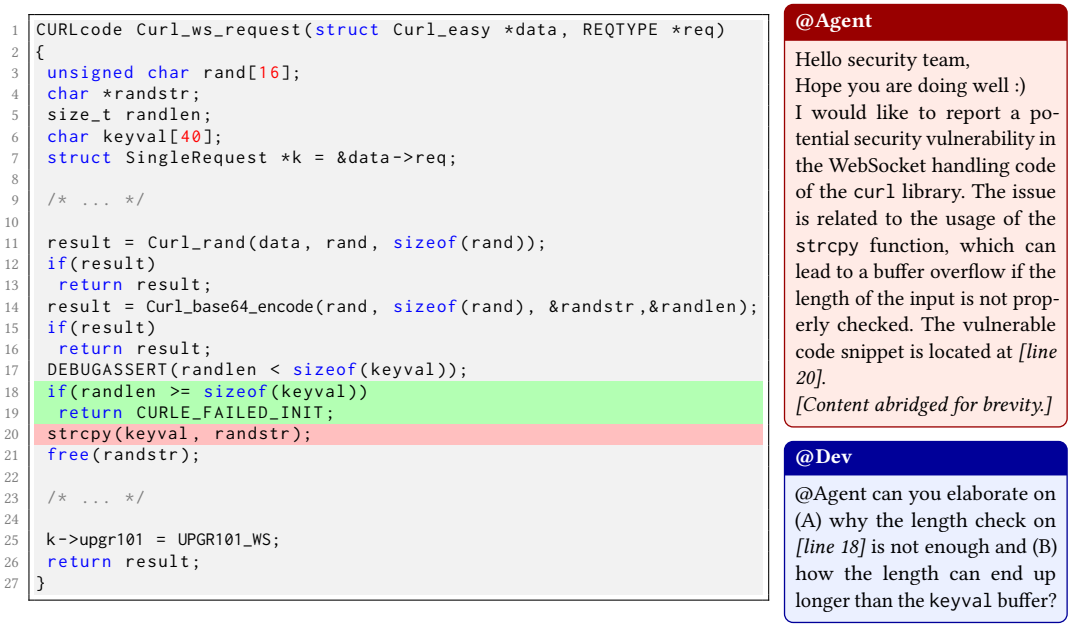
In summary, our contributions are as follows:

- We identify the problem of reasoning over incomplete programs and argue that the correctness of LLM-generated inferences should be defined relative to a *completion model*;
- We introduce *completion semantics*, a formal framework that interprets incomplete programs as spaces of possible refinements and defines query satisfaction relative to completion models;
- We instantiate completion semantics through a *witness-generation workflow* for concretizing completions for code fragments in LLM prompts, and exposing assumptions underlying existential inferences; and
- We evaluate the approach on real-world program-analysis tasks including bug reports, and demonstrate its utility for validating reasoning over incomplete code fragments.

Our completion semantics formalism is intentionally modest. Our objective is not to introduce some deep new semantic machinery, but rather to identify the correct abstraction for reasoning over incomplete programs. Our abstraction has surprising explanatory power: it accounts for a range of empirical phenomena previously studied in LLM-based program reasoning [12, 19, 23, 32, 43]—including context sensitivity, disagreement between analyses, and several distinct classes of reasoning failures—while remaining independent of any particular reasoning engine or prompting strategy. Accordingly, the witness-generation workflow should be viewed as an experimental vehicle for evaluating the semantics rather than as the primary contribution of the paper.

## 2 Motivation

In this section, we provide motivating examples, a problem statement, and summarize our approach.



```
1  CURLcode Curl_ws_request(struct Curl_easy *data, REQTYPE *req)
2  {
3   unsigned char rand[16];
4   char *randstr;
5   size_t randlen;
6   char keyval[40];
7   struct SingleRequest *k = &data->req;
8
9   /* ... */
10
11  result = Curl_rand(data, rand, sizeof(rand));
12  if(result)
13   return result;
14  result = Curl_base64_encode(rand, sizeof(rand), &randstr,&randlen);
15  if(result)
16   return result;
17  DEBUGASSERT(randlen < sizeof(keyval));
18  if(randlen >= sizeof(keyval))
19   return CURLE_FAILED_INIT;
20  strcpy(keyval, randstr);
21  free(randstr);
22
23  /* ... */
24
25  k->upgr101 = UPGR101_WS;
26  return result;
27 }
```



Fig. 2. Code fragment from cURL WebSocket handling reported as a buffer overflow vulnerability due to use of `strcpy`. The correctness of the inference made by the LLM depends on assumptions made over the complete program.

## 2.1 Example

Figure 2 shows an AI-generated bug report submitted to the bug bounty program of the cURL project.[1] Here, the LLM flags a "*potential security vulnerability in the WebSocket handling code of the curl library*" related to the use of the `strcpy` function in the highlighted line (line 20). The LLM has inferred that the call to `strcpy` "*can lead to a buffer overflow if the length of the input is not properly checked.*" Based on this inference, the report goes on to describe reproduction steps involving a "*base64-encoded nonce value that exceeds the buffer size*" and recommends replacing `strcpy` with `strncpy` as a safer alternative. Although the original prompt was never made public, we can induce a similar bug report for line 20 using local LLMs (e.g., `llama3`) with the function and a prompt asking the existential query "*does the following function have a buffer overflow?*". In this paper, our question is not whether LLMs can make mistakes (they can), but rather *what it means for a given LLM inference to be correct or not*?

The answer to this question depends on what *assumptions* can be made over the omitted code. As illustrated in Figure 1 in Section 1.1, a program fragment can be *completed* in different ways with different definitions, and the choice of the completion can change the result of the query—and by extension, the correctness of any inference made by the LLM. For the bug report in Figure 2, we consider two possible completions that result in different interpretations.

*2.1.1 Interpretation A: Contract-Preserving Completions.* The most natural interpretation of the example in Figure 2 is to assume that omitted code behaves according to its *intended contracts*. Specifically, we can identify two main contracts over the return values of the `curl_base64_decode` call, specifically:

(1) *Valid string*: The `randstr` string is a valid object and correctly null-terminated; and

[1] See https://hackerone.com/reports/2298307 (retrieved July 2026).

(2) *Length*: `randlen == strlen(randstr)`

Under this interpretation, the safety check on lines 18–19 guarantees that the `strcpy` destination buffer on line 20 is sufficiently large for the copied string. Consequently, no buffer overflow can occur, meaning that the original LLM inference is **incorrect**.

The actual cURL implementation satisfies this interpretation, but the argument does not depend on the concrete codebase itself in general. Rather, it relies on the broader assumption that omitted functions respect their intended behavior and maintain consistent length and string invariants. For example, contract (1) is a common assumption for all strings passed on over function boundaries in C programs. Contract (2) can be inferred by a natural language interpretation over the (`randstr`, `randlen`) variable pair. Specifically, that `randstr` returns a string, the common `rand`-prefix shared between variables, and the `len`-suffix are all heuristic clues that `randlen` is intended to return the length of `randstr`. Both humans and LLMs alike will make such assumptions.

*2.1.2 Interpretation B: Contract-Agnostic Completions.* Instead of relying on heuristics, we could allow completions that ignore implied contracts. Under this relaxed model, we consider any completion to be valid provided it is *syntactically correct* and does not redefine any symbol. However, under this model, it turns out that the *call to* `strcpy` *on line 20 can actually overflow*, despite the safety check on lines 18–19. Under this interpretation, the original LLM inference must be deemed **correct**.

To see why, we consider a completion of `Curl_base64_encode` that breaks the implied contracts (1) and (2). For example, we can consider a completion with the following function definition:

```
CURLcode curl_base64_decode(const char *, size_t, char **outstr, size_t *outlen) {
  *outptr = malloc(40);
  memset(*outptr, 'A', 40);      // Violation since *outlen is not nul-terminated
  *outlen = 20;                  // Violation since *outlen != strlen(*outstr)
  return CURL_OK;
}
```

This function is syntactically correct and compiles without error. If this definition were allowable, then the overflow on line 20 can be induced as follows:

(1) The call to `Curl_base64_encode` returns a non-null-terminated string of length 40;
(2) The safety check on line 18 passes since `randlen` is 20 and `sizeof(keyval)` is 40; and
(3) The `strcpy` operation on line 20 *overflows* given the non-null terminated `randstr`.

This example further highlights how inference over incomplete program fragments implicitly depends on assumptions about the missing code, and how even questionable LLM inference could actually be valid under some interpretations. The question arises, when should we consider a completion to be valid, and when it is not?

## 2.2 Problem Statement

The example in Figure 2 demonstrates that correctness cannot be defined relative to arbitrary completions alone. If every syntactically valid completion is admitted, then many properties become trivially realizable by constructing sufficiently pathological implementations of omitted code. In the present example, the reported overflow can be induced by a completion that violates natural assumptions regarding string termination, length consistency, and API behavior. Under such a model, the bug report must be considered correct, despite relying on assumptions that most developers would regard as unreasonable.

At the opposite extreme, we may attempt to restrict completions using heuristically inferred contracts and domain knowledge. For example, C strings are overwhelmingly null-terminated when

passed across function boundaries, and paired variables such as (`randstr`, `randlen`) strongly suggest a length-string relationship. Both humans and LLMs routinely rely on such assumptions when reasoning about code. Under these assumptions, the report in Figure 2 is incorrect. However, such assumptions are also not universally valid. Familiar library interfaces such as `strncpy`, `strxfrm`, and `readlink` violate common string-handling conventions, while real-world software frequently contains undocumented behavior, incomplete specifications, and implementation-specific invariants. Consequently, correctness cannot be reduced to heuristic assumptions alone.

The challenge is therefore to characterize which completions should be considered *admissible* when reasoning over an incomplete program fragment. On the one hand, purely syntactic notions of completion are too permissive and admit unrealistic interpretations. On the other hand, informal notions of intended behavior are difficult to define precisely and may vary across developers, domains, and programming environments. The problem is further complicated by the fact that LLMs themselves are often treated as *black boxes*: while an inference may implicitly depend on assumptions about omitted context, those assumptions are generally not made explicit.

This paper takes the position that reasoning over incomplete programs should be understood in terms of a *completion model*: a set of admissible completions against which queries are interpreted. Rather than attempting to define correctness directly in terms of LLM outputs, we instead define correctness relative to a completion model and treat the LLM as a mechanism for exploring that model. This perspective separates the semantic question—*what completions are considered valid?*—from the algorithmic question of how a particular model, human, or analysis procedure reasons about those completions.

### 2.3 Our Approach

We interpret an incomplete program fragment as denoting a space of possible completions rather than a single program. A completion model specifies which of these completions are admissible, thereby determining the assumptions under which omitted code is interpreted. Program properties are then evaluated relative to the completion model: existential queries hold whenever some admissible completion witnesses the property, whereas universal queries must hold for every admissible completion.

A key feature of our approach is that we *separate* three different concerns that can be conflated when using LLMs for program reasoning:

(1) Firstly, the underlying *program semantics* determines *when* a property should hold, but generally only over a complete program;
(2) Secondly, the *completion model* determines *how* program fragments should be interpreted, and which assumptions about omitted context are considered admissible; and
(3) Finally, a *reasoning procedure* determines *whether* properties can be established within those completions.

This separation allows reasoning over incomplete programs to be studied independently of any particular analysis technique or language model: a program *fragment* implies a *completion model*, which defines the *semantics*, which can be decided via a *reasoning procedure*. Our approach is illustrated in Figure 3.

Our framework is designed for LLM-based program reasoning. When presented with an incomplete code fragment, an LLM must implicitly make assumptions about context that is not present in the prompt. These assumptions may concern missing implementations, programmer intent, coding conventions, undocumented contracts, or other latent information. Completion models provide a formal abstraction of such assumptions. Under this view, an LLM-generated inference may be understood as a claim about the existence or absence of completions supporting

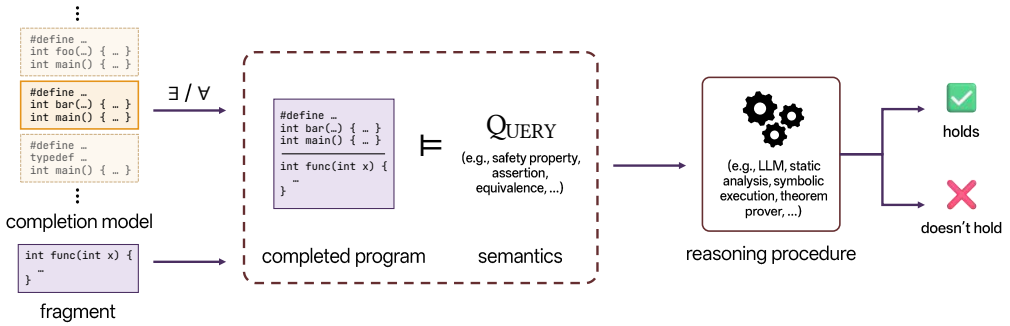



Fig. 3. Illustration of completion semantics for reasoning over incomplete programs.

the reported property. Failure of this chain is not as simple as "*hallucination*", but could arise from query defects/misinterpretation, the completion model being too restricted or permissive, or a failure of the reasoning procedure itself.

A final practical challenge is that completion models are often *implicit*. For example, LLM-based reasoning is often treated as a *black-box*, meaning that the underlying assumptions that an LLM uses for inference are typically not directly observable. However, it may be possible to use the same LLM to generate a completion that *supports* (or *refutes*) a specific inference, and to also concretize any assumption used by the LLM in the inference. This is a form of *witness generation*: rather than treating the LLM as an opaque oracle, we generate a concrete completion (a witness) to *expose* the assumptions necessary for an inference to hold over the original fragment.

## 3 Completion Semantics

In this section, we formalize *completions* and *completion models* for reasoning over incomplete program fragments.

### 3.1 Fragments and Completions

An incomplete program fragment does not denote a single executable program, but rather admits many possible *completions*—each corresponding to a different interpretation of the omitted context. For example, the fragment in Figure 1 may be completed with different definitions of `struct S`, leading to different answers to the same query.

To capture this intuition, we model both *fragments* and *completions* as *strings* of program text.

**Definition 3.1** (Completion)**.** Let $f_r$ be a *program fragment* represented as a *string*. A *completion* of $f_r$ is any string $C$ that extends $f_r$ by supplying additional program context while preserving the contents of $f_r$. We write $C \sqsupseteq f_r$ to denote that $C$ is a completion of $f_r$.

Our definition is intentionally permissive. A completion need not compile, execute, or satisfy any semantic property. It needs only to extend the original fragment. More restrictive notions of completion, such as *syntactically* valid programs, *compilable* programs, or *contract-respecting* programs, will be introduced formally as *completion models*.

Furthermore, a precise definition of the $\sqsupseteq$ operation is also intentionally left open. One definition could be the *substring* relation, i.e., $C = \mathit{prefix} +\!\!+ f_r +\!\!+ \mathit{postfix}$, which is our assumed default. In general, more complicated definitions are possible.

### 3.2 Completion Models

The unrestricted notion of a completion is intentionally permissive. In practice, not all completions are equally meaningful. For example, since a completion can be an arbitrary string, almost all

completions will fail to parse, will violate language semantics, or will rely on assumptions that would be considered unreasonable by a developer (see Section 2.1.2). To address this issue, we introduce the notion of a *completion model*. A completion model characterizes which completions are considered *admissible* when reasoning about an incomplete program fragment.

**Definition 3.2** (Completion Model)**.** Let $f_r$ be a program fragment. A *completion model* $\mathcal{M}(f_r)$ is any set of completions of $f_r$.

A completion model may be viewed as a filter over the unrestricted completion space. Different models admit different completions and therefore induce different interpretations of the same fragment. The examples in Section 2.1 illustrate this phenomenon: a reported bug may be realizable under one completion model (e.g., all *compilable* completions) while impossible under another (assumed *developer* contracts).

The simplest completion model is the *unrestricted model*, which admits every completion of a fragment: $\mathcal{M}_{\text{ALL}}(f_r) = \{C \mid C \sqsupseteq f_r\}$. This model is usually far too permissive, as it admits completions containing arbitrary syntax errors, inconsistent type definitions, or implementations that deliberately violate the intent of the surrounding code. More restrictive models are constructed by imposing additional constraints on admissible completions. For example:

- *Syntactic models* only admit completions that parse successfully under the language grammar;
- *Compilation models* admit only completions that satisfy the language's static semantics (e.g., type checking, linking, and having an entry-point);
- *Contract models* that restrict completions according to programmer-supplied specifications, inferred contracts, API documentation, or other behavioral assumptions; and
- *Domain-specific models* that incorporate any other additional assumption about a particular programming environment, framework, or application domain.

These examples illustrate an important property of completion semantics: correctness is always relative to a completion model. A query that is true under one model may be false under another (see Section 2.1). The choice of completion model therefore determines which assumptions about omitted context are considered admissible.

In the context of LLM-based program reasoning, the most interesting completion model is the one implicitly assumed by the model itself. Intuitively, an LLM does not reason over all possible completions of a fragment, but rather over some subset that it considers *plausible* given the prompt, training data, and inferred intent. While such a completion model is not directly observable, it provides a useful abstraction for understanding the assumptions underlying LLM-generated inferences. We return to this idea in Section 4.

### 3.3 Query Satisfaction

The preceding sections define how incomplete program fragments induce spaces of possible completions. We now define how queries are interpreted over such spaces.

Completion semantics does not replace existing notions of program semantics. Rather, it builds upon them. We assume some underlying (possibly partial) semantic relation:

$$\mathit{Prog} \models \textsc{Query} \qquad \text{(Semantic-Relation)}$$

between complete programs *Prog* and properties Query. The precise interpretation of $\models$ and the Query language depends on the analysis/semantics being applied. For example, Query may denote an *execution* property (e.g., assertion failure or memory safety), and a *static* property (e.g., type correctness). Furthermore, the relation is typically only defined for completions within the *semantic domain* of the query, written dom(Query). For example, if Query is an execution property, then dom(Query) will typically contain all *compilable* and *runnable* programs. Likewise, if Query is a

type-correctness property, then dom(Query) will typically contain all *parseable* programs. The Query itself can be expressed in any language of choice, including formal languages, and even natural language (common for LLM prompting).

The role of completion semantics is to *lift* the semantic relation from complete programs to incomplete program fragments. Given a fragment $f_r$ and a completion model $\mathcal{M}(f_r)$, queries are interpreted relative to the completions admitted by the model:

**Definition 3.3** (Existential Satisfaction)**.** A completion model $\mathcal{M}(f_r)$ *existentially* satisfies a Query if *there exists* a completion $C \in \mathcal{M}(f_r)$ such that $C \models$ Query, i.e.:

$$\mathcal{M}(f_r) \models_{\exists} \text{Query} \qquad \text{iff} \qquad \exists C \in \mathcal{M}(f_r) \cap \text{dom}(\text{Query}) : C \models \text{Query} \qquad \text{(Existential)}$$

Intuitively, an existential query holds whenever at least one admissible completion witnesses the property. Many program-analysis tasks naturally take this form. For example, a *bug report* can be interpreted as the claim that there exists a completion under which an assertion can fail or a vulnerability can be triggered under the program's execution semantics.

Likewise, we can define a *universal query*:

**Definition 3.4** (Universal Satisfaction)**.** A completion model $\mathcal{M}(f_r)$ *universally* satisfies a Query if *for all* completion $C \in \mathcal{M}(f_r)$ we have that $C \models q$, i.e.:

$$\mathcal{M}(f_r) \models_{\forall} \text{Query} \qquad \text{iff} \qquad \forall C \in \mathcal{M}(f_r) \cap \text{dom}(\text{Query}) : C \models \text{Query} \qquad \text{(Universal)}$$

Universal queries correspond to properties that must hold regardless of how the omitted context is completed. For example, for Figure 1, the query "*does* `f(...)` *always return zero?*" is an example of a universal property. However, the answer depends on the completion model: if the model admits both pointer-based and array-based definitions of `struct S`, then the property does *not* hold universally. The distinction gives rise to three useful notions:

- A query is *realizable* if it holds for at least one completion: $\mathcal{M}(f_r) \models_{\exists}$ Query
- A query is *locally true* if it holds for every completion: $\mathcal{M}(f_r) \models_{\forall}$ Query
- A query is *locally false* if it holds for no completion: $\mathcal{M}(f_r) \not\models_{\exists}$ Query

An example of a *locally true* or *false* property could be "*is the function correctly formatted?*" under a supplied coding standard represented as ($\models$). Such a query depends only on the fragment $f_r$, and is independent from any context.

### 3.4 Discussion

Completion semantics provides a very general mechanism for *lifting* existing reasoning techniques from complete programs to incomplete program fragments. Any analysis capable of evaluating a property over a complete program may be used as the underlying semantic relation. Examples include *execution-based* techniques such as *testing*, *symbolic execution*, and *model checking*, as well as static analyses such as *type checking*, *data-flow analysis*, and *abstract interpretation*, or even syntactic properties such as *code formatting standards*. Completion semantics remains agnostic to the particular analysis being employed; its role is simply to interpret the resulting property relative to a completion model.

Viewed through this lens, the examples from Figure 1 and Figure 2 illustrate two distinct sources of ambiguity that arise when reasoning about incomplete programs. In Figure 1, the truth of the query depends on omitted type information, and different completions of `struct S` induce different program behaviors, leading to different answers to the same question. In Figure 2, the ambiguity is more subtle. While a vulnerability may be realizable under some completions, many such completions rely on assumptions that violate the apparent *contracts* and *conventions* of the

surrounding code. The challenge is therefore not merely to determine whether a witness exists, but to determine which completions should be considered *admissible*.

These examples illustrate an important separation of concerns. Traditional program semantics determines whether *a property holds for a complete program*. Completion models determine which assumptions about omitted context are considered *admissible*. Reasoning procedures determine how properties are established within those completions. Completion semantics isolates these concerns and allows them to be studied independently. As a consequence, disagreements about the validity of an inference can often be traced to differences in completion models rather than differences in the underlying program semantics.

**Example 3.1** (cURL Bug Report Revisited)**.** The distinction can be illustrated using the cURL example from Figure 2. Here, the underlying *semantics* is simply the execution semantics of the completed program: a bug exists if an execution reaches the reported overflow. The *completion model* determines which implementations of the omitted context are considered admissible. For example, a reasonable completion model may only contain definitions for `Curl_base64_encode(...)` that produce a nul-terminated string whose reported length is consistent with the generated output. Finally, the *reasoning procedure* determines how the property is established. One analysis may search for a violating execution using testing or fuzzing, while another may use symbolic execution to prove that no such execution exists. In fact, under this completion model and symbolic execution, all paths to the `strcpy` operation must satisfy:

$$\mathit{strlen}(\texttt{randstr}) < \mathit{sizeof}(\texttt{keyval}) \qquad \text{(Path-Constraint)}$$

meaning that no `strcpy` overflow is possible. These choices are independent: the same completion model may be analyzed using different reasoning procedures, while the same reasoning procedure may yield different conclusions under different completion models.

## 4 Completion Models for LLM-Based Program Reasoning

Applying completion semantics to LLMs is straightforward: the prompt defines an incomplete fragment, and the LLM implicitly reasons relative to an (unknown) completion model.

### 4.1 LLM Completion Models

Completion semantics can be applied to any setting in which program reasoning is performed over incomplete program fragments. LLM-based program reasoning is a particularly important instance of this problem because the LLM is rarely prompted with a complete program (due to token limits and cognitive resources). Instead, the prompt necessarily contains only a fragment of a larger codebase together with a task description and a limited surrounding context. Consequently, any inference produced by the model necessarily depends on assumptions regarding code that is not present in the prompt fragment. This observation suggests a natural interpretation of LLM-based reasoning in terms of completion models. Given a *prompt*, comprising a program fragment $f_r$ and instruction $i$, we associate the LLM with an *implicit completion model* $\mathcal{M}^{i}_{\text{LLM}}(f_r)$. This model represents the set of completions that the LLM considers *plausible* when reasoning over fragment $f_r$ under instruction $i$.

*Completion Model Definition.* Unlike some of the example completion models introduced in Section 3, $\mathcal{M}^{i}_{\text{LLM}}(f_r)$ will generally not have a self-contained definition. For example, a compiler-defined completion model can be characterized by the language specification. Likewise, a contract-based completion model can be characterized by a formal description of explicit behavioral constraints. In contrast, the assumptions underlying an LLM's reasoning emerge from training data, instruction ($i$) interpretation, and inferred context. As a result, the corresponding completion model may generally

lack an explicit descriptive definition. That said, for our purposes, completion semantics does not require $\mathcal{M}^i_{\text{LLM}}(f_r)$ to have an explicit, self-contained, definition. Rather, the purpose of the model is not to describe the internal operation of the LLM, but rather to provide a semantic interpretation of the inferences it produces. In particular, the completion model serves as an abstraction of the assumptions that must hold for a reported conclusion to be valid.

*Completion Model Membership.* Even though $\mathcal{M}^i_{\text{LLM}}(f_r)$ may generally lack a descriptive definition, it is nevertheless possible to construct individual completions that are consistent with the model's assumptions. This can be achieved by constructing candidate completions and instructing the LLM to *test* for validity ($\mathcal{M}^i_{\text{LLM}}(f_r)$ membership) under the LLM's internal assumptions. Alternatively, we may also instruct the LLM to *generate* candidate completions. This observation will form the basis of self-validation via witness generation that we shall explore further in Section 5.

Unlike compiler- or contract-defined completion models, there is no guarantee that elements of $\mathcal{M}^i_{\text{LLM}}(f_r)$ correspond to valid or executable programs. An LLM may reason using assumptions that are incomplete, inconsistent, or otherwise unrealizable. Likewise, the LLM may also make mistakes when testing for, or generating, element membership of the completion set. This is a possible failure mode, and we shall elaborate on this issue later in Section 4.3.

### 4.2 Reasoning over Completion Models

The completion model $\mathcal{M}^i_{\text{LLM}}(f_r)$ provides a semantic interpretation of the assumptions underlying an LLM's reasoning. The remaining question is how inferences produced by the model should be interpreted relative to this completion model. Under completion semantics, an LLM-generated inference is viewed as a claim regarding the satisfaction of a query over $\mathcal{M}^i_{\text{LLM}}(f_r)$. For example, suppose the model reports that a program fragment contains a vulnerability (i.e., is *unsafe*). Such a report is interpreted as an *existential* claim stating that there exists a completion under which the vulnerability is realizable. This is represented as follows:

$$\mathcal{M}^i_{\text{LLM}}(f_r) \models_{\exists} \textsc{Bug} \qquad \text{(Existential-Reasoning)}$$

Likewise, a report asserting that a fragment is *safe* is interpreted as a universal claim stating that this property holds for all admissible completions:

$$\mathcal{M}^i_{\text{LLM}}(f_r) \models_{\forall} \textsc{Safe} \qquad \text{(Universal-Reasoning)}$$

More generally, completion semantics do not prescribe which lifted satisfaction relation should be used for a given query. Rather, the interpretation depends on the intended meaning of the instruction $i$ from the prompt, which is typically expressed in natural language. For example, bug-finding instructions are naturally interpreted as existential queries, whereas safety or verification instructions are naturally interpreted as universal queries. Completion semantics provides a formal meaning once the interpretation is fixed.

Our perspective also highlights the key distinction between traditional program reasoning and reasoning over incomplete programs. Given a complete program, the primary question is whether a property holds under the underlying program semantics. In contrast, reasoning over an incomplete fragment requires an additional layer of interpretation concerning the omitted context. Consequently, the validity of an LLM-generated inference depends not only on the correctness of the reasoning process itself, but also on the assumptions embodied by the completion model.

*Examples.* Returning to the examples from Figure 1 and Figure 2, the reported conclusions can now be interpreted as claims over completion models. In Figure 1, the query is "*does the function* `f(...)` *always return zero?*". The answer to the query depends on whether the completion model admits pointer-based definitions of `struct S`. Most modern LLMs consider completions with the

pointer-based definition to be *admissible*. This means that the correct inference for the query must be "*false*", "*not guaranteed*", or equivalent.

Likewise, for Figure 2, the reported vulnerability depends on whether the completion model admits implementations of `Curl_base64_encode(...)` that violate the apparent contracts of the surrounding code. As shown in Section 2.1, if we allow for syntactic and compilable completions, then the inference of the vulnerability is **correct**. However, such models tend to be too permissive. Instead, we can test for self-consistency against the LLM's own completion model. To do so, we instruct the LLM to test candidate completions, such as the one from Section 2.1.2, for feasibility. Assuming GPT-5, the LLM explicitly rejects this completion for violating implied contracts. Specifically, the proposed completion:

> "*is inconsistent with how a string-returning base64 routine is ordinarily expected to behave.*"

Similarly, if we ask for GPT-5 to *generate* a completion for a reported vulnerability, it reports:

> "*I do* **not** *see a reasonable implementation of the base64 routine that both (1) satisfies the apparent API contract, and (2) causes the* `strcpy()` *into* `keyval[40]` *to overflow.*" (emphasis original)

Assuming these answers accurately reflect the completion model $\mathcal{M}^{i}_{\text{LLM}}(f_r)$ under GPT-5 and this fragment, then any inference that Figure 2 contains a real vulnerability must be **incorrect**.

*Discussion.* Completion semantics provides a meaning for LLM-generated inferences independently of how those inferences are produced. Whether the underlying reasoning procedure is a human developer, a language model, a symbolic executor, or some combination thereof, the resulting conclusion can be interpreted relative to a completion model. This provides a common semantic framework for reasoning about incomplete programs and the assumptions required to make such reasoning meaningful.

We are also careful to point out that completion semantics themselves are a model for explaining inference, and not how LLMs actually work internally. For example, completion semantics can explain *failure modes* as discussed next.

### 4.3 Failure Modes

A useful consequence of completion semantics is that it makes explicit the distinct ways in which LLM-based program reasoning can fail. Such failures are often grouped together under broad terms such as "*hallucination*". However, once reasoning is interpreted relative to a completion model, it becomes possible to distinguish failures arising from incorrect assumptions about omitted context from failures arising from incorrect reasoning over those assumptions. We explore these ideas in this subsection.

*4.3.1 Completion Model Errors (Prompting Failure).* The first class of failures arises when the completion model itself is *misaligned* with the intended interpretation of the fragment or prompt instructions. Informally, we may view the developer as inducing an idealized completion model $\mathcal{M}^{i}_{\text{DEV}}(f_r)$ that captures the knowledge and assumptions intended for the surrounding code, and $\mathcal{M}^{i}_{\text{LLM}}(f_r)$ is often an *abductive* approximation of the developer's intent guided by training. However, there is no guarantee that $\mathcal{M}^{i}_{\text{LLM}}(f_r)$ will actually coincide with $\mathcal{M}^{i}_{\text{DEV}}(f_r)$ for any given code fragment. As such, an LLM may admit or include completions that violate implicit contracts or other assumptions. For example, consider the simple code fragment:

```c
void my_copy(char *dst, const char *src, size_t n) {
    for (size_t i = 0; i < n; i++) dst[i] = src[i];
}
```

Without additional context, an LLM may abductively assume a `memcpy`-style contract for this function (non-`NULL` arguments for `n` $> 0$, buffers valid for `n` bytes, no buffer overlap), and the completion model $\mathcal{M}^i_{\text{LLM}}(f_r)$ will reflect this. However, it is possible that the developer intended a different implied contract (e.g., overlapping buffers allowed, or `dst` being `NULL` is allowed as a *nop*), meaning that the code is actually buggy. In such cases, the resulting inference may be technically correct with respect to $\mathcal{M}^i_{\text{LLM}}(f_r)$ while nevertheless being unhelpful in practice.

Likewise, LLM-based inference may also fail due to misinterpretation of the instruction ($i$). Completion semantics defines both an *existential* ($\models_\exists$) and a *universal* ($\models_\forall$) definition of satisfaction, and the correct definition to apply is usually implicit in the instruction. For example, "*does this code contain a bug?*" implies existential reasoning, whereas "*does this loop always terminate?*" implies universal reasoning. In principle, an LLM could misunderstand instructions and apply the wrong reasoning. Such errors can be broadly classified as *prompting* or *interpretation* errors, and indicate that the prompt was insufficient to constrain the LLM's interpretation. This is an inherent failure mode of LLM-based reasoning in general.

*4.3.2 Reasoning Errors (Hallucinations).* The second class of failures arises when the completion model is reasonable, aligned with developer expectations, and the instruction is interpreted correctly; however, the reasoning process itself is incorrect. In this case, the LLM fails to correctly determine whether or not a property holds relative to its own completion model. Since LLMs are fundamentally approximate reasoning engines, they are prone to making mistakes. For example, suppose that:

$$\mathcal{M}^i_{\text{LLM}}(f_r) \models_\exists \textsc{Bug}$$

holds, since there is a concrete element $C \in \mathcal{M}^i_{\text{LLM}}(f_r)$ which witnesses the $\textsc{Bug}$ (i.e., the LLM agrees that $C$ is a valid completion), and $C \in \text{dom}(\textsc{Bug})$. However, the LLM may still erroneously report that no such bug exists. Likewise, the converse may occur, where the model reports a bug exists, despite no admissible completion witnessing the property.

Such failures correspond most closely to the classical notion of a *hallucination*: the interpretation and assumptions used by the LLM are reasonable and aligned, but the inference itself is incorrect.

*4.3.3 Recovery Errors.* A third class of failures arises when attempting to recover or reconstruct the LLM's reasoning, e.g., by finding elements of the completion model. As discussed in Section 4.1, LLM completion models generally lack an explicit descriptive definition. Consequently, practical techniques, such as membership testing and completion generation, may themselves be imperfect.

For example, the LLM may incorrectly reject a completion that is consistent with its own assumptions, incorrectly accept an inconsistent completion, or generate a completion that does not lie within the semantic domain of the query (i.e., $C \notin \text{dom}(q)$). Likewise, external validation procedures may fail to recognize a valid witness due to incompleteness of the underlying oracle, resource limits, or timeouts. These failures do not necessarily imply that the completion model or reasoning process is incorrect per se; rather, these arise from the difficulty of recovering explicit elements from an implicit completion model.

## 5 Witness Generation Instantiation

Completion semantics provides a general interpretation of LLM-generated inferences relative to completion models. While the completion model induced by an LLM may lack an explicit descriptive

definition, we can nevertheless use the LLM to *self-validate* under its own completion model using a *witness generation* workflow. By instantiating completion semantics with an execution-based oracle, witness generation provides a method to *test* whether LLM-based program reasoning is correct for certain classes of queries. Witness generation is not part of completion semantics itself. Rather, it is one practical instantiation that makes completion models observable and allows the proposed semantics to be evaluated empirically.

### 5.1 Completion Witnesses

Although we interpret inference as a claim regarding the existence of a completion satisfying some property, the completion itself is typically not directly observable. To address this problem, we introduce the notion of a *completion witness*. A completion witness is a concrete completion that justifies an *existential inference*, defined as follows:

**Definition 5.1** (Completion Witness)**.** Let $f_r$ be a fragment, Query a query, and $\mathcal{M}(f_r)$ a completion model. Then $W$ is a *completion witness* for the existential query $\mathcal{M}(f_r) \models_\exists$ Query iff: (1) $W \in \mathcal{M}(f_r)$, (2) $W \in \mathrm{dom}(\text{Query})$, and (3) $W \models$ Query.

In other words, a *completion witness* is a completion that both satisfies the assumptions of the completion model, is within the semantic domain of the query, and demonstrates the queried property under the underlying semantics, satisfying Definition 3.3. Completion witnesses are evidence for *existential* queries:

- The existence of a completion witness for Query establishes $\mathcal{M}(f_r) \models_\exists$ Query; but
- The failure to construct such a witness does *not* imply $\mathcal{M}(f_r) \not\models_\exists$ Query.

Likewise, a completion witness for the negated query can be used to *refute* universal queries:

- The existence of a completion witness for ¬Query establishes $\mathcal{M}(f_r) \not\models_\forall$ Query; but
- The failure to construct such a witness does *not* imply $\mathcal{M}(f_r) \models_\forall$ Query.

Consequently, completion witness generation can be viewed as a practical validation (or refutation) mechanism rather than a decision procedure.

### 5.2 Completion Witness Generation for Self-Validation

Suppose an LLM reports an existential inference:

$$\mathcal{M}^i_{\text{LLM}}(f_r) \models_\exists \text{Query}.$$

Under completion semantics, this report implicitly claims the existence of a completion witness satisfying the conditions of Definition 5.1. Consequently, a natural form of self-validation is to *prompt* the same model to generate a completion witness explicitly. The key intuition is that: if the LLM can correctly reason that a property holds under some completion, then it should also be capable of constructing a completion exhibiting the property. Completion witness generation therefore converts an implicit inference into an explicit artifact that can be inspected and validated.

More concretely, completion witness generation decomposes the original inference into three separate obligations to address the concerns identified in Section 2.3:

(1) **Completion construction.** Generate a completion that makes explicit the assumptions required for the inference.
(2) **Domain validation.** Establish that the completion is within the semantic domain of the query.
(3) **Property validation.** Establish that the queried property holds for the completion under the underlying semantics and underlying reasoning procedure.

For LLM-induced completion models, the first obligation is naturally performed by the LLM itself as a form of *self-validation*. That is, the model is challenged to construct the omitted program

context that explains or justifies the reported inference, and the resulting completion serves as a concrete realization of assumptions that were previously implicit. The second and third properties can be established using *external oracles*. Depending on the underlying semantics, these oracles may consist of compilers, interpreters, theorem provers, symbolic executors, testing frameworks, or other program-analysis tools. Completion witness generation itself is therefore not tied to a particular semantic domain; rather, it provides a general mechanism for connecting completion semantics with existing validation procedures.

### 5.3 Execution Witnesses

Completion semantics is agnostic to the underlying semantic relation ($\models$) used to evaluate queries, meaning that the witness definition of Section 5.1 therefore supports arbitrary semantic domains in principle. In this section, we focus on *execution-based* queries of the form:

$$\mathit{Prog} \models \exists e : q(e) \qquad \text{(Existential-Execution)}$$

where $q(e)$ is predicate over an *execution* $e$ of program $P$. Here, an *execution* is any *semantic object* that can be used to represent the concept of an execution through program $P$, which is typically defined by the programming language semantics. As with the definition of ($\models$) itself, we also deliberately keep the definition of *execution* abstract, as there are many possible ways to represent an execution through a program (e.g., *traces* and *inputs*). An *execution witness* is therefore a concrete execution $w$ such that $q(w)$ holds.

For example, for any valid completion $C$ of Figure 2, an *execution witness* would be any execution that induces an overflow in `strcpy`. The execution witness may be represented as an input to the completion witness $W$ that induces the overflow. The overflow can be detected by a reasoning procedure, which can be a bug detection oracle such as AddressSanitizer [37].

*Universal Queries.* As with completion witnesses, an execution witness for the negated query can be used to *refute* universal queries of the form: $\mathit{Prog} \models \forall e : q(e)$.

### 5.4 Witness Generation Workflow

The witness-generation process combines the completion-witness and execution-witness concepts introduced in the previous sections. Given a fragment $f_r$, instruction $i$, and an existential inference:

$$\mathcal{M}^{i}_{\text{LLM}}(f_r) \models_{\exists} \exists e : q(e) \qquad \text{(existential-query)}$$

the goal is to construct a witness pair $\langle W, w \rangle$ consisting of a *completion witness* $W$ and an *execution witness* $w$. The completion required for validation is typically much smaller than a fully functional program. Rather, witness generation may seek a *minimal completion* sufficient to explain the reported inference. In practice, such completions often resemble a test *harness* that supplies the minimal context required for the property to hold.

Conceptually, the workflow proceeds in two stages. First, the LLM is tasked with *generating* a completion witness $W$ that makes explicit the assumptions underlying the reported inference. The LLM is explicitly instructed to find a minimal completion $W$ such that:

(1) $W \in \mathcal{M}^{i}_{\text{LLM}}(f_r)$;
(2) $W$ compiles and runs (i.e., $W$ lies within the semantic domain of the query); and
(3) $W$ accepts an input that induces an execution $w$ such that $q(w)$ holds.

Since membership in $\mathcal{M}^{i}_{\text{LLM}}(f_r)$ has no independent oracle, completion generation must necessarily rely on the LLM itself (i.e., self-validation). However, both (2) and (3) can be validated with external oracles. As with completion generation, execution-witness generation may be performed iteratively, using feedback from the oracle to refine candidate witnesses.

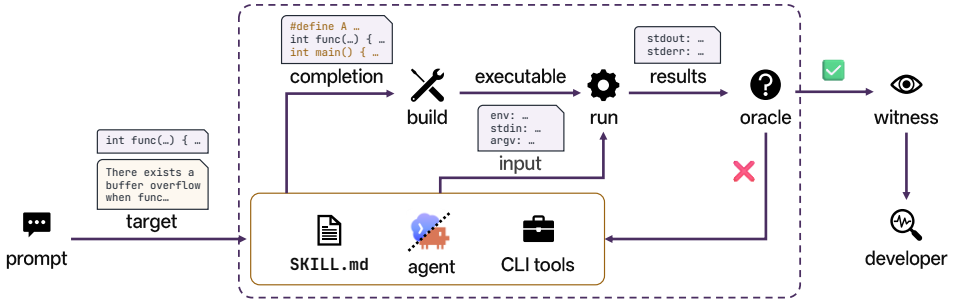



Fig. 4. The WitGen workflow.

*Example.* When applied to the vulnerability report of Figure 2, the workflow attempts to construct both a completion explaining the reported overflow and an execution demonstrating the overflow. Failure to generate such a witness pair provides evidence that the reported vulnerability is unsupported by the completion model explored by the recovery procedure.

### 5.5 Implementation

We have implemented the workflow as an AI coding agent *skill* [3] that is compatible with modern coding agents, such as Codex [33] and Claude Code [2]. Here, an agentic *skill* is a modular bundle of instructions, resources, and executable components that agents can dynamically invoke for specialized tasks. WitGen is our implementation of a witness generation agent skill.

An overview of the WitGen is shown in Figure 4. The core orchestration is handled by the agent, and the core functionality is handled by a `witgen` CLI tool, implemented in Python, that supports candidate completion witness generation, building, and running. Leveraging the CLI tool, WitGen works in a refinement loop: given a *target* consisting of a code fragment and an existential claim, WitGen first completes the code fragment under the assumptions that explain the inference. Next, WitGen attempts to build and run the completion with synthesized inputs and specified environments, such as command-line arguments and environment variables. Then, the execution results are judged by an external oracle other than WitGen itself, like embedded assertions, sanitizers, etc. If a candidate is generated, but the oracle fails to confirm the claimed property (e.g., no buffer overflow detected), the agent will refine the completion or propose a new one in the next iteration; otherwise, if this successfully yields a valid completion and execution witness that proves the claim, or the agent reaches a preset maximal retry count, then the process is over with a reported verdict: supported witness or no witness found.

Additionally, by adjusting the instructions in the *skill*, the agent can be guided to generate completions following different disciplines, such as preserving standard library behavior or rewriting it, based on the LLM's knowledge and understanding from training data and its reasoning capabilities. This unlocks WitGen to generate witnesses according to different completion models, e.g., compilation models and contract models mentioned in Section 3.2, enabling us to reveal witnesses in different contexts during evaluation.

## 6 Evaluation

### 6.1 Experimental Setup

*Dataset Selection.* To evaluate WitGen and completion semantics, we use several datasets, including both *true positives* (TP) where the inference should hold, and *false positives* (FP) where the inference should not hold. The TP/FP judgment is made by an external oracle. The datasets are:

Table 1. Summary of results in WitGen evaluation. Here, *Found* is the number of witnesses found, *Unrealistic* is the number of *unrealistic* witnesses found (as deemed by an external oracle), *Not Found* is the number of subjects where no witness was found, and *Aligned* is how closely the result aligns with expectations (found for TP, not found for FP).

| Dataset | Subset | Total | Found | Unrealistic | Not Found | Aligned |
|---|---|---|---|---|---|---|
| CVEs | TP | 50 | 43 (86.0%) | 0 (0.0%) | 7 (14.0%) | **43 (86.0%)** |
| AI-Slop | FP | 12 | 0 (0.0%) | 4 (33.3%) | 8 (66.7%) | **8 (66.7%)** |
| AutoBug | TP | 15 | 15 (100.0%) | 0 (0.0%) | 0 (0.0%) | **15 (100.0%)** |
| | FP | 15 | 0 (0.0%) | 7 (46.7%) | 8 (53.3%) | **8 (53.3%)** |

- CVEs: 50 real-world CVEs from MegaVul [31]. All subjects are TPs.
- AI-Slop: 12 false bug reports (all FPs) listed under the cURL project's "*AI slop hall of shame*"[2]. We treat the cURL developer's rejection of these bug reports as the external oracle.
- AutoBug: 30 subjects from the Mixed-Manual dataset from AutoBug [24]. Out of the 30 subjects, 15 are reported as TPs, and the other 15 are reported as FPs. We treat the original test harness for these subjects as the external oracle.

The dataset selection provides a balance between valid report, and false reports as judged by an external oracle. For the AI-Slop, we selected the 12 usable reports out of 49 candidates that (1) specifically identified a bug, and (2) specifically identified a source location. The remaining 38 reports were not bugs (e.g., missing documentation), reported alleged buggy behavior but failed to identify a location, or appeared to be entirely hallucinated (reported location does not exist). Both CVEs and AI-Slop focus on existential inference in the form of bug reports, whereas AutoBug is existential inference in the form of post-condition violations represented by an `assert` statement.

*Dataset Configuration.* For each test subject in our dataset, we construct natural-language instructions (e.g., existence of a *bug* or *assertion failure*) with a corresponding code fragment (relevant function). For the CVEs dataset, we extract the instruction from the *Common Vulnerabilities and Exposures* (CVE) description and the original issue report, and the affected function from the corresponding MegaVul metadata. For the AI-Slop dataset, we manually select reports that refer to concrete code within the cURL codebase, retrieve the relevant code fragment, and condense each report into concise one- or two-sentence instructions. Finally, for AutoBug, we used the original subjects explicitly marked pre- and post-conditions. We then formulate a claim that the post-condition assertion fails for some input satisfying the pre-condition. These subjects exhibit a range of behaviors. The CVEs dataset tests real-world vulnerabilities to test the effectiveness of WitGen against *true positives* (TPs) as independently determined by a third party. The AI-Slop tests WitGen against *false positives* (FPs) as determined by a third party (the cURL developers). Finally, the AutoBug dataset tests WitGen against C and Python code subjects obtained from solutions to coding contest problems. Here, a subject is considered to be a TP if there exists an input that satisfies the pre-condition, and yet violates the post-condition. In the evaluation, we use the GPT-5.4 [34] frontier model as the underlying LLM in WitGen's workflow. We use Codex [33] as the underlying agentic architecture. The WitGen loop is run for a maximum of 3 iterations.

*Prompt (Re)Construction.* For some subjects, the original prompt and LLM (if any) used to make the inference were never made public and are deemed lost. Thus, for these experiments, we run WitGen using a *reconstructed prompt* comprising a natural language *description* of the inference (bug type and location) as the instruction $i$, and the function where the bug occurs as the code fragment $f_r$.

[2]See https://gist.github.com/bagder/07f7581f6e3d78ef37dfbfc81fd1d1cd?permalink_comment_id=5956460 (retrieved July 2026).

Table 2. Table Of Results In WitGen Evaluation On AI-Slop and AutoBug false positives (FP). Here "*Not Found*" means that no witness was found (the expected result for a FP) and "*Unrealistic*" means a witness was found, but deemed unrealistic by the external oracle.

| | Id | +Hack | Explanation | Default | Explanation | +Context | Explanation |
|---|---|---|---|---|---|---|---|
| AI-Slop | 2298307 | Unrealistic | Mismatched `payload` | ✓Not Found | | ✓Not Found | = |
| | 2823554 | Unrealistic | Injected `strlen` | ✓Not Found | | ✓Not Found | = |
| | 2887487 | Unrealistic | Mismatched `size` | ✓Not Found | | ✓Not Found | = |
| | 2981245 | Unrealistic | Dangling parameter | ✓Not Found | | ✓Not Found | = |
| | 3117697 | Unrealistic | Circular reference | Unrealistic | = | ✓Not Found | + `Cookie` invariant |
| | 3137657 | ✓Not Found | | ✓Not Found | = | ✓Not Found | = |
| | 3230082 | ✓Not Found | | ✓Not Found | = | ✓Not Found | = |
| | 3392174 | Unrealistic | Mismatched `outlen` | ✓Not Found | | ✓Not Found | = |
| | 3459636 | Unrealistic | Injected `strcpy` | ✓Not Found | | ✓Not Found | = |
| | 3462525 | Unrealistic | Section 6.3.1 | Unrealistic | = | ✓Not Found | + `buf` space pre-cond. |
| | 3516186 | Unrealistic | Section 6.3.3 | Unrealistic | = | ✓Not Found | + `time_t` non-negative |
| | 3516202 | Unrealistic | Section 6.3.2 | Unrealistic | = | ✓Not Found | + Single threaded |
| AutoBug (FP) | q0003 | Unrealistic | Invalid UTF-8 | Unrealistic | = | ✓Not Found | + `s.isascii()` |
| | q0007 | ✓Not Found | | ✓Not Found | = | ✓Not Found | = |
| | q0069 | ✓Not Found | | ✓Not Found | = | ✓Not Found | = |
| | q0080 | ✓Not Found | | ✓Not Found | = | ✓Not Found | = |
| | q0161 | ✓Not Found | | ✓Not Found | = | ✓Not Found | = |
| | q0238 | Unrealistic | Integer overflow | Unrealistic | = | ✓Not Found | + Element range |
| | task23 | ✓Not Found | | ✓Not Found | = | ✓Not Found | = |
| | task36 | ✓Not Found | | ✓Not Found | = | ✓Not Found | = |
| | task51 | Unrealistic | `NaN` parameter | Unrealistic | = | ✓Not Found | + Element range |
| | task54 | Unrealistic | `NaN` parameter | ✓Not Found | | ✓Not Found | = |
| | task58 | Unrealistic | Missing `year` limits | Unrealistic | = | ✓Not Found | + Element range |
| | task61 | Unrealistic | Dynamic typing | Unrealistic | = | ✓Not Found | + Type hint |
| | task69 | Unrealistic | Dynamic typing | Unrealistic | = | ✓Not Found | + Type hint |
| | task72 | Unrealistic | Dynamic typing | ✓Not Found | | ✓Not Found | = |
| | task84 | Unrealistic | UTF-8 `swapcase()` | Unrealistic | Dynamic typing | ✓Not Found | + `s.isascii()` |

We run WitGen under the assumption that the prompt had resulted in the corresponding inference, and WitGen attempts to generate a witness accordingly.

*Variants.* We also test three main variants of our setup:

(1) *Default*: Attempts to find a completion consistent with $\mathcal{M}^i_{\text{LLM}}(f_r)$;
(2) *+Hack*: Attempts to find *any* completion that compiles and triggers the claim; and
(3) *+Context*: Same as (1), but manually adds additional context to the prompt to further constrain $\mathcal{M}^i_{\text{LLM}}(f_r)$ to match developer intent.

Here, (1) is our main result. The (2) variant uses a modified version of WitGen that instructs the LLM to generate a witness under $\mathcal{M}_{\text{ALL}}$—including any witness that would generally be considered to be unreasonable. The (3) variant is used to distinguish prompting errors from other hallucinations.

### 6.2 Insights from Witness Generation

The objective of our evaluation is to investigate whether witness generation can expose assumptions predicted by the completion semantics foundation. Consequently, successful witness generation provides evidence that the reported inference is supported under the explored completion model, while failure to recover such a witness may indicate either a recovery limitation or a mismatch between the assumptions required by the inference and those admitted by the completion model. A summary of the overall results is shown in Table 1, and detailed results for the FP subjects are showing in Table 2. The main findings are discussed below.

*True Positives Admit Witnesses.* Across the MegaVul and AutoBug (TP) datasets, WitGen consistently generates executable witnesses for the majority of subjects (Table 1). This result is consistent with the completion semantics interpretation of existential reasoning: if a reported vulnerability is *realizable* under the completion model explored by the LLM, then there should exist a completion

witness together with a corresponding execution witness demonstrating the property. Cases where no witness was recovered tend to correspond to practical limitations of witness recovery (recovery errors) rather than evidence that the underlying vulnerability does not exist.

*False Positives Tend to Lack Admissible Witnesses.* The AI-Slop and AutoBug (FP) datasets may initially appear *plausible* when considered over incomplete program fragments alone. However, when WitGen attempts to explicitly construct a completion witness, the process often fails, or underlying (unreasonable) assumptions tend to become exposed. For the majority of reports (66.7% for AI-Slop and 53.3% for AutoBug (FP)), WitGen fails to generate *any* reasonable completion. In other cases, the workflow is able to construct witnesses, but only by introducing assumptions that were deemed implausible or inconsistent with the surrounding code, especially for Table 2 +Hack mode, where unreasonable witnesses are expressly allowed. These examples support the central claim of this paper: the disagreement in inference results is not simply LLM "*hallucination*". Rather, the reported inference depends on a completion model, which can differ from that implicitly assumed by the developers. We will discuss some case studies in more detail in Section 6.3.

*Additional Context Resolves Completion Model Mismatch.* To further investigate unrealistic witnesses, we performed an additional +Context experiment in which the original prompt was manually extended with additional context or pre-conditions that mirror the external oracle. Recall that we use *systematic* prompt (re)construction for Default: the prompt fragment $f_r$ contains only the relevant function and no other context, which is sometimes insufficient. The +Context experiment tests a prediction made by completion semantics: that unrealistic witnesses can arise from overly permissive completion models that arise from under-constrained prompts. Our results support this hypothesis. For most unrealistic witnesses, additional context can be provided to eliminate unrealistic completions. This suggests that many unrealistic witnesses should not necessarily be interpreted as failures of the LLM's reasoning procedure or of witness generation itself, but rather as consequences of an under-constrained prompt inducing an overly permissive completion model.

*Discussion.* Overall, our evaluation supports the completion semantics perspective developed throughout this paper. Witness generation is valuable not merely because it validates individual LLM-generated inferences, but because it makes explicit the assumptions required for those inferences to hold. When witnesses can be constructed, they provide concrete evidence supporting an existential claim. When witness generation fails, or only succeeds under unrealistic assumptions, the failure itself is informative: it often reveals a mismatch between competing completion models rather than a simple reasoning error. We believe this ability to expose hidden assumptions is at least as valuable as the witnesses themselves, since it provides developers with an explanation of *why* different analyses may legitimately disagree over incomplete programs.

### 6.3 Case Studies

We consider three case studies from the AI-Slop dataset that highlight the challenges with LLM-based program inference. Despite these bug reports being rejected by the cURL developers, many reports nevertheless support interpretations where the inference is *correct*. We discuss below.

*6.3.1 Case Study: Missing Calling Assumptions.* Our first case study[3] illustrates the simplest form of completion ambiguity. The report concerns the following small helper function:

[3]https://hackerone.com/reports/3462525

```
static int storebuffer(unsigned char outc, void *f) {
  char **buffer = f;
  **buffer = (char)outc;       // Store outc into the buffer
  (*buffer)++;                 // Increment the buffer pointer
  return 0;
}
```

A call to `storebuffer(ch, &buf)` essentially just implements the operation `*buf++ = ch`. When presented with this function in isolation, most LLMs will flag a potential buffer overflow hazard, since the function does no explicit bounds checking internally. Furthermore, we can use WitGen to synthesize a witness program that (1) allocates a single-byte buffer, and (2) invokes `storebuffer(...)` twice to cause an overflow in the second call. The overflow is readily detected using AddressSanitizer. The inference is thus **correct** under the LLM's completion model ($\mathcal{M}^{i}_{\text{LLM}}$).

The cURL developers nevertheless rejected the report. Their explanation was not that the synthesized witness is necessarily incorrect, but rather that it violates an implicit assumption of the function's API. Specifically, `storebuffer(...)` is only meant to be called when the *caller* has verified that sufficient buffer space remains. Consequently, the witness generated by the LLM does not reflect any intended or actual usage of the function. The inference is therefore **incorrect** under the developer's implicit completion model.

This case study illustrates an example of *prompting failure*. Specifically, the prompt omitted critical information necessary to constrain the completion model to match that of the developer, allowing the LLM to reason over a substantially larger space of admissible completions than intended. Importantly, the disagreement did not arise due to either party performing incorrect reasoning. Rather, both the LLM and the developers derive valid conclusions with respect to their respective completion models. This simple example also highlights how prompt construction for LLM-based program reasoning is non-trivial in the general case.

*6.3.2 Case Study: Single-Threaded versus Multi-Threaded Execution Model.* Our second case study[4] alleges a use-after-free vulnerability in the `replace_existing(...)` function from the cURL cookie subsystem. The report claims that "*the function modifies a linked list while iterating over it*", creating the potential for memory corruption in concurrent environments. The report assumes that multiple invocations of the function may execute concurrently on the same cookie data structure. Under such a completion model, the inference is **correct**, since one thread may retain a pointer to a linked-list node while another thread concurrently removes and frees the same node. Since the function performs no synchronization, such an execution may result in a use-after-free.

However, the cURL developers promptly rejected the report, explaining that the function is never executed concurrently and that "*each* cURL *invoke runs isolated from the others*" so "*there isn't really a race condition*". Under a completion model that only admits *single-threaded* execution environments (consistent with the actual cURL code base), access to the cookie database is externally serialized. Consequently, the reported execution cannot occur, and the inference is **incorrect**.

This example illustrates how completion models not only describe missing source code, but also constrain the execution environment in which the fragment executes. Here, the relevant distinction is not the implementation of any omitted function, but whether the surrounding context permits concurrent execution. The same program fragment, therefore, admits different conclusions depending solely on which execution model is considered admissible.

[4]https://hackerone.com/reports/3516202

*6.3.3 Case Study: Integer Width Assumptions.* Another report[5] submitted to the cURL bug bounty program alleged an integer clamping bug when processing the Max-Age attribute of HTTP cookies:

```
time_t now = time(NULL);
if (CURL_OFF_T_MAX - now < co->expires)   // Does expires+now exceeds the max value?
  co->expires = CURL_OFF_T_MAX;           // If yes, then clamp
else
  co->expires += now;                     // If no, then accumulate normally
```

This code ensures the sum co->expires+now is representable as a 64-bit signed integer, or else the result is clamped to the max value (CURL_OFF_T_MAX). However, the bug report alleges that when "*the current time (*now*) is large enough*", then the subtraction "*produces unexpected results*", allowing for the else-branch to be taken in cases where the sum can overflow or wrap.

The report was promptly dismissed by the cURL developers, who noted that "*for this to be true, surely* now *would have to be larger than* CURL_OFF_T_MAX?" Under the actual cURL implementation, an implicit representation assumption (CURL_OFF_T_MAX $\geq$ now) will always hold, since CURL_OFF_T_MAX is the maximum possible value (INT64_MAX) and that now/expires are non-negative values. This means the reported bug is impossible in practice, and any inference under such a model will be **incorrect**. The Table 2 +Context result makes these assumptions explicit.

However, this inference assumes that the time_t type is a 64-bit unsigned integer. While this is true for all modern operating systems, the C standard does not technically guarantee this. Instead, the C standard merely requires time_t to be an integer type of some unspecified signedness and width. Under this specification, the C standard would technically allow a *65-bit width* for time_t:

```
typedef unsigned _BitInt(65) time_t;
```

Such a completion would allow for values that directly violate the invariant, for example:

```
now = CURL_OFF_T_MAX + 1
```

Under such a completion model, the if-conditional is *not* sufficient to prevent the else-branch from being taken when wrapping could occur, meaning the inference is **correct**.

This example illustrates an important feature of completion semantics. The disagreement is not fundamentally about arithmetic, but about omitted assumptions. Under the intended cURL completion model with the ubiquitous time_t definition, the report is invalid. However, under a more permissive completion model admitting all C-conforming implementations, the same inference technically becomes valid. The bug report therefore cannot be classified as simply "correct" or "incorrect" independently of the completion model against which it is interpreted.

## 6.4 Empirical Study Comparison

Recent empirical studies [12, 19, 23, 32, 43] have collectively identified a consistent set of phenomena regarding LLM-based program analysis, particularly for *vulnerability detection*, *classification*, and *repair*. Across different models, benchmarks, and prompting strategies, these studies repeatedly report that the quality of LLM-based program reasoning heavily depends on contextual information, assumptions about omitted code, and prior exposure to similar implementations. Collectively, these observations suggest that many apparent failures of LLM-based program analysis arise from reasoning over incomplete programs, rather than from reasoning errors alone. A qualitative analysis mapping these empirical works to completion semantics is shown in Table 3. In this section, we summarize the major findings and relate them back to completion semantics.

*Context is Important.* Many empirical studies [19, 23, 32, 43] identify *insufficient context* as a major factor in the quality of LLM-based code inference (Table 3 (1), (2), and (8)). For example, especially

[5] https://hackerone.com/reports/3516186

with *vulnerability detection*, most studies found that *function-level* analysis is generally insufficient, and advocate providing additional contextual information. Completion semantics reaches a similar conclusion: insufficient context leaves the completion model under-constrained, allowing for the admission of reasonable program completions that disagree with the developer's expectations or the code base. This is reflected in our Table 2 Default and +Context results, where the difference in alignment is explained by missing context.

Most existing studies conclude that more context *should* be included into prompts. However, the recent work of [43] also demonstrates that indiscriminately supplying additional context can also *reduce* analysis accuracy due to the introduction of "excessive noise". This indicates that the objective is not to maximize the amount of code presented to the LLM, but to provide minimal information that most effectively constrains the intended completion model to the intended target. Consequently, practical LLM-based analysis involves selecting a context that best approximates the intended semantic completion while remaining within the reasoning capabilities of the model.

*Many Factors Shape LLM Completion Models.* A second recurring observation is that a wide variety of seemingly unrelated factors influence LLM reasoning (Table 3 (3)-(6)), including assumptions about omitted APIs, identifier naming, code obfuscation, and prior exposure to similar code during training. From the perspective of completion semantics, these factors all influence the shape of the induced completion model ($\mathcal{M}^i_{\text{LLM}}$) by changing which program completions are considered admissible. Our framework provides a common semantic interpretation for these otherwise disparate empirical observations. At the same time, completion semantics is largely agnostic as to how strongly each factor influences the induced completion model. This is illustrated by WitGen—which self-validates against the LLM's own completion model—independent of empirical observations.

*Completion Models need not Contain Well-formed Programs.* Finally, it has been observed that LLMs may generate code that does not compile [12]. This motivates one of our design choices: completion models are not restricted to syntactically or semantically well-formed programs. Instead, they represent the space of candidate completions that could be considered during inference, while independent oracles (e.g., compilers and test suites) may subsequently reject invalid completions.

*Discussion.* Our work is complementary to these empirical studies. Rather than evaluating prompting strategies or proposing new prompts, we ask a more fundamental semantic question: *what does it mean for an LLM-generated inference over an incomplete program fragment to be correct?* We argue that incomplete code fragments do not denote a single program, but rather a space of possible completions, and that correctness must therefore be interpreted relative to a completion model. From this perspective, many reported "false positives" are not necessarily incorrect inferences, but instead reflect different assumptions about omitted code, contracts, and execution context. Consequently, our framework complements empirical observations by explaining why these limitations arise as an inherent consequence of reasoning over incomplete programs.

## 7 Related Work

*Classical Program Semantics.* Classical program semantic frameworks [9, 14, 17] generally define reasoning over complete programs by first interpreting programs within some *semantic domain* (e.g., *traces*, *transition systems*, *abstract domains*, or *logical formulae*). Completion semantics instead addresses an orthogonal question: *how should incomplete program fragments be interpreted before such semantics are applied?* Rather than replacing existing semantic frameworks, completion semantics lifts them to incomplete programs by interpreting a fragment as a set of admissible completions. Any underlying program semantics can then be applied unchanged to each completion. Likewise, completion semantics is independent of the reasoning procedure used to establish properties over

Table 3. Examples of prior empirical observations and their interpretation under completion semantics.

| No. | Empirical observation | Completion semantics interpretation |
|---|---|---|
| (1) | LLM conclusions change as additional code context is provided. [19, 23, 32, 43] | Additional context rules out previously admissible completions. |
| (2) | Redundant code context can degrade LLM analysis accuracy [43]. | The representation supplied to the LLM is no longer an efficient encoding of the intended completion model. |
| (3) | LLMs make incorrect assumptions about missing code or APIs. [23, 32, 43] | The LLM reasons under a completion model that differs from the developer's intended completion model or the original code base. |
| (4) | Meaningful identifier names improve analysis accuracy [12]. | Identifier names constrain the completion model by excluding otherwise plausible interpretations. |
| (5) | Code obfuscation reduces analysis accuracy [12, 19]. | Obfuscation removes semantic constraints from the fragment, expanding the completion model and increasing ambiguity. |
| (6) | LLM analyses are influenced by prior exposure to similar code. [12, 19, 32] | Previously observed implementations (during training) can bias the completion model towards particular completions. |
| (7) | LLMs often generate broken code that does not compile [12]. | The completion model does not necessarily only contain compilable-programs. |
| (8) | LLM analyses can produce false positives and false negatives. [19, 23, 32, 43] | These may arise either from reasoning errors under the chosen completion model or from mismatch between the induced completion model and the intended developer model. |

those semantics. This separation is particularly natural for LLM-based program analysis, where the reasoning procedure is an *approximate* language model operating directly over the source text.

*Modular Program Reasoning.* Classical approaches to modular program reasoning [6, 28, 35], including *contract-based verification*, *assume-guarantee reasoning*, and *type systems*, also reason about incomplete programs. Rather than requiring a complete program, these techniques use *interfaces*, *contracts*, or other *specifications* to characterize the behavior of omitted components. Such specifications effectively define a completion model by restricting which implementations of the missing components are considered admissible. Classical modular reasoning typically assumes specifications are explicitly provided and can be reasoned about formally. In contrast, LLM-based reasoning can be understood in terms of a completion model built from the prompt, surrounding code, and prior knowledge encoded by the LLM.

*Specification Inference.* Related work on *specification inference* [11, 13], including LLM-based specification inference [7, 10, 25], attempts to *recover* assumptions about missing program behaviour automatically. Such inferred specifications can be understood as constraints over, or descriptions of, possible completion models. Completion semantics is intentionally more general, as admissible completions need not be expressible as behavioural specifications alone. Missing *type definitions*, *data structure layouts*, *libraries*, or *environmental assumptions* may all influence the meaning of an incomplete fragment without naturally corresponding to traditional specifications. Completion semantics therefore separates the question of which completions are admissible from the particular formalism used to describe them, allowing both explicit specifications and implicitly inferred assumptions to be treated within a common semantic framework.

*Program Slicing.* Program slicing [18, 39] seeks to identify the *subset* of a program relevant to a computation or property. AutoBug [24] and HyllFuzz [26] are examples of slice-based LLM program reasoning systems. In terms of completion semantics, the slice may correspond to an "optimized" fragment whose omitted context ought not alter the semantics. Slicing and completion semantics address different questions: slicing attempts to *minimize context*, whereas completion semantics assigns meaning to the ultimate fragment. Slicing may still recursively introduce dependencies as relevant procedures, types, and global state, which can still be of significant size. Thus, even slice-based systems tend to omit context, meaning that completion semantics remains applicable.

*LLM-based Program Reasoning.* Beyond empirical studies of Section 6.4, several works [4, 8, 15, 22] have investigated the broader problem of *LLM-based program reasoning*. Becker et al. [4] study reasoning over source code as a program-analysis task, characterizing the capabilities and limitations of LLMs when performing semantic reasoning about programs. $\mathcal{R}$Eval [8] extends this direction by introducing benchmarks that evaluate reasoning about program runtime behaviour (execution paths and program state), together with a notion of incremental reasoning consistency. Crucially, these works primarily investigate how well LLMs reason about *complete executable programs* that have a limited size. In contrast, our work addresses a complementary semantic question: because LLMs over large code bases will typically reason over incomplete program fragments, the correctness of an inference necessarily depends on assumptions about omitted context.

*LLMs and Artifact Generation.* Recent work [1, 41, 42, 44] has increasingly coupled LLM reasoning with the generation of *artifacts* that can be independently validated. Examples include executable proof-of-concept exploits and regression tests for validating bug reports (e.g., AnyPoC [44] and Otter [1]), as well as machine-checkable proofs, verification annotations, and counterexamples in LLM-assisted formal verification systems such as AutoVerus [41] and ExVerus [42]. Our work is complementary to this direction. We do not claim artifact generation as a contribution in itself; rather, we provide a semantic foundation explaining why such artifacts are meaningful for reasoning over incomplete programs.

*AI Coding Agents.* Recent coding agents such as Claude Code [2] and Codex [33] mitigate limited context by iteratively retrieving additional files, invoking search tools, and requesting clarifications when insufficient information is available. This changes the practical workflow, but not the underlying semantic problem considered in this paper. At every stage, the agent reasons over a finite context while treating the remainder of the codebase and execution environment as implicit. Even when an agent decides that additional context is required, it must eventually terminate retrieval and perform inference relative to the remaining omitted context. Completion semantics therefore applies equally to agentic workflows: the retrieval strategy simply changes where the boundary between explicit and implicit context is drawn.

## 8 Discussion

The primary contribution of this paper is a new semantic framework for program reasoning over incomplete programs. *Completion semantics* is intentionally straightforward: an incomplete program fragment effectively denotes a space of possible completions, and correctness is defined relative to this space. This formalism leads to several useful observations regarding the nature of LLM-based program reasoning that are difficult to express without first making completion models explicit. We discuss the main findings in this section.

*Correctness is Relative.* The motivating question of this paper is: *when is an LLM-based program reasoning correct?* Completion semantics suggests that this question does not admit an absolute answer. Since LLMs are typically forced to reason over incomplete program fragments, and fragments do not denote a single program, then correctness is necessarily *relative* to the completion model used to interpret the fragment.

This perspective explains why apparently contradictory conclusions may simultaneously be reasonable. The examples from Figure 1 and Figure 2, and the case studies from Section 6.3, all illustrate this phenomenon. The same program fragment will frequently admit both "correct" and "incorrect" interpretations depending on the assumptions made that are encoded in the completion model, specifically regarding omitted *types*, *functions*, *contracts*, *execution environments*, or *platform-specific behavior*. Likewise, the case study in Section 6.3.3 demonstrates how even seemingly objective

notions, such as "*the* C *language semantics*", can lead to surprising inferences. Disagreements therefore do not imply incorrect reasoning; they may instead arise because different parties reason under different completion models. Completion semantics therefore shifts the question from "*is the inference correct?*" to "*under which completion model is the inference correct?*"

*Context Selection Problem.* Completion semantics also highlight a fundamental tension in LLM-based program reasoning. Adding additional context to a prompt generally improves the accuracy, with respect to developer expectations, of the induced completion model. However, larger prompts also consume finite reasoning resources, increase retrieval complexity, and may reduce reasoning quality due to context dilution or LLM cognitive overload. Conversely, aggressively minimizing prompts may improve reasoning efficiency, but it enlarges the completion space, increasing the likelihood that the model reasons under assumptions that differ from those intended.

This trade-off appears to be inherent rather than incidental. Modern software systems are sufficiently large that an LLM cannot reason over an entire codebase and its execution environment simultaneously. Regardless of how much context is supplied, a boundary between observed and omitted context will usually exist. Prompt construction therefore becomes an optimization problem balancing completion-model fidelity against available reasoning resources. Completion semantics describes this trade-off independently of any particular retrieval strategy or language model.

*Reasoning Failures are Not Monolithic.* Current discussions of LLM reliability often describe incorrect outputs simply as "*hallucinations*." Completion semantics suggests that this view is overly coarse. Section 4.3 identifies several distinct failure modes. An inference may fail because the LLM's completion model itself differs from the developer's intended interpretation. Alternatively, the completion model may be appropriate, while the reasoning procedure itself reaches the wrong conclusion. Further failures arise when attempting to recover explicit witnesses from an implicit completion model. These failure modes have different causes and therefore should have different remedies. Completion-model mismatch motivates improved retrieval or prompt construction. Reasoning and recovery failures motivate improved (frontier) models. Reasoning errors also motivate validation methods such as witness generation. Distinguishing these cases provides a more precise understanding of LLM behavior.

*Implications for Evaluation.* Completion semantics also have implications for the evaluation of LLM-based program reasoning systems. Many existing benchmarks classify generated reports simply as correct or incorrect. Completion semantics suggests that such labels implicitly assume a particular completion model, even if that model is never explicitly stated. Consequently, benchmark disagreements may reflect differences in assumed context rather than deficiencies of the underlying reasoning procedure. This observation suggests that future evaluations should distinguish reasoning quality from completion-model quality whenever possible. For example, witness generation allows an inferred completion to be inspected directly, making it possible to determine whether a disagreement arises because the reasoning procedure failed or because the assumptions underlying the inference differ from those intended by the developer.

*Broader Perspective.* Completion semantics are intentionally independent of any particular reasoning engine. Throughout this paper, we focused primarily on LLM-based reasoning as a natural application of completion models. However, any reasoning over incomplete programs may be interpreted through the same completion-model perspective. Likewise, witness generation should be viewed as one possible instantiation of the framework (rather than the framework itself). Witnesses provide a practical mechanism for exposing assumptions underlying existential inferences, but other mechanisms for exploring or characterizing completion models are possible.

## Acknowledgements

This research is supported by the National Research Foundation, Singapore, under its National Cybersecurity R&D Programme (Award No. CRPO-GC5-NUS-004).

## References

[1] Toufique Ahmed, Jatin Ganhotra, Rangeet Pan, Avraham Shinnar, Saurabh Sinha, and Martin Hirzel. 2025. Otter: Generating Tests from Issues to Validate SWE Patches. In *Proceedings of the 42nd International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 267)*, Aarti Singh, Maryam Fazel, Daniel Hsu, Simon Lacoste-Julien, Felix Berkenkamp, Tegan Maharaj, Kiri Wagstaff, and Jerry Zhu (Eds.). PMLR, 752–771. https://proceedings.mlr.press/v267/ahmed25b.html

[2] Anthropic. Accessed: 2026-06. Claude Code. https://code.claude.com/docs.

[3] Anthropic. Accessed: 2026-06. Extend Claude with Skills. https://code.claude.com/docs/en/skills.

[4] Norman Becker, Tural Mammadov, and Andreas Zeller. 2026. Can LLMs Really Reason about Code? Studying How Well LLMs Understand the Relation between Input, Code, and Output. In *Proceedings of the 3rd ACM International Conference on AI-Powered Software (AIware '26)*. 21–30. doi:10.1145/3805760.3814888

[5] Dimitrios Stamatios Bouras and Sergey Mechtaev. 2026. Defusing Logic Bombs in Symbolic Execution with LLM-Generated Ghost Code. arXiv:2603.19239 [cs.PL] https://arxiv.org/abs/2603.19239

[6] K.M. Chandy and J. Misra. 1988. *Parallel Program Design: A Foundation*. Addison-Wesley.

[7] 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 2024)*. 9–17. doi:10.1145/3652588.3663317

[8] Junkai Chen, Zhiyuan Pan, Xing Hu, Zhenhao Li, Ge Li, and Xin Xia. 2025. Reasoning Runtime Behavior of a Program with LLM: How Far are We?. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. 1869–1881. doi:10.1109/ICSE55347.2025.00012

[9] Edsger Dijkstra. 1975. Guarded commands, nondeterminancy and formal derivation of programs. *Commun. ACM* 18, 8 (1975), 453–457. doi:10.1145/360933.360975

[10] Madeline Endres, Sarah Fakhoury, Saikat Chakraborty, and Shuvendu K. Lahiri. 2024. Can Large Language Models Transform Natural Language Intent into Formal Method Postconditions? *Proc. ACM Softw. Eng.* 1, FSE, Article 84 (July 2024), 24 pages. doi:10.1145/3660791

[11] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. 2007. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming* 69, 1 (2007), 35–45. doi:10.1016/j.scico.2007.01.015

[12] Chongzhou Fang, Ning Miao, Shaurya Srivastav, Jialin Liu, Ruoyu Zhang, Ruijie Fang, Asmita, Ryan Tsang, Najmeh Nazari, Han Wang, and Houman Homayoun. 2024. Large Language Models for Code Analysis: Do LLMs Really Do Their Job?. In *33rd USENIX Security Symposium (USENIX Security 24)*. 829–846. https://www.usenix.org/conference/usenixsecurity24/presentation/fang

[13] Cormac Flanagan and K. Rustan M. Leino. 2001. Houdini, an Annotation Assistant for ESC/Java. In *Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity (FME)*. 500–517. doi:10.5555/647540.730008

[14] Robert Floyd. 1967. Assigning Meanings to Programs. *Proceedings of Symposium on Applied Mathematics* 19 (1967), 19–32.

[15] Alex Gu, Baptiste Rozière, Hugh Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida I. Wang. 2024. CRUXEval: a benchmark for code reasoning, understanding and execution. In *Proceedings of the 41st International Conference on Machine Learning (ICML'24)*. Article 659, 54 pages. doi:10.5555/3692070.3692729

[16] Jinyao Guo, Chengpeng Wang, Xiangzhe Xu, Zian Su, and Xiangyu Zhang. 2025. Repoaudit: An autonomous llm-agent for repository-level code auditing. In *Proceedings of the 42nd International Conference on Machine Learning*. https://openreview.net/forum?id=TXcifVbFpG

[17] C.A.R. Hoare. 1969. An axiomatic basis for computer programming. *Commun. ACM* 12, 10 (Oct. 1969), 576–580. doi:10.1145/363235.363259

[18] Bogdan Korel and Janusz Laski. 1988. Dynamic program slicing. *Inform. Process. Lett.* 29, 3 (1988), 155–163. doi:10.1016/0020-0190(88)90054-3

[19] Damian Koterba, Maryna Łukaczyk, and Wojciech Książek. 2026. Leveraging Large Language Models for advanced static code analysis: Assessing the feasibility of AI superseding traditional vulnerability detection methods. *Information and Software Technology* 198 (2026), 108213. doi:10.1016/j.infsof.2026.108213

[20] Teemu Lehtinen, Charles Koutcheme, and Arto Hellas. 2024. Let's ask ai about their programs: Exploring chatgpt's answers to program comprehension questions. In *Proceedings of the 46th International Conference on Software Engineering:*

*Software Engineering Education and Training (ICSE-SEET)*. 221–232. doi:10.1145/3639474.3640058

[21] Mosh Levy, Alon Jacoby, and Yoav Goldberg. 2024. Same Task, More Tokens: the Impact of Input Length on the Reasoning Performance of Large Language Models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 15339–15353. doi:10.18653/v1/2024.acl-long.818

[22] Wen-Ding Li and Kevin Ellis. 2024. Is programming by example solved by LLMs?. In *Proceedings of the 38th International Conference on Neural Information Processing Systems (NIPS '24)*. Article 1422, 30 pages. doi:10.5555/3737916.3739338

[23] Yue Li, Xiao Li, Hao Wu, Minghui Xu, Yue Zhang, Xiuzhen Cheng, Fengyuan Xu, and Sheng Zhong. 2025. Everything You Wanted to Know About LLM-based Vulnerability Detection But Were Afraid to Ask. arXiv:2504.13474 [cs.CR] https://arxiv.org/abs/2504.13474

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

[25] Lezhi Ma, Shangqing Liu, Yi Li, Xiaofei Xie, and Lei Bu. 2025. SpecGen: Automated Generation of Formal Program Specifications via Large Language Models. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering (ICSE '25)*. 16–28. doi:10.1109/ICSE55347.2025.00129

[26] Ruijie Meng, Gregory J. Duck, and Abhik Roychoudhury. 2026. Large Language Model assisted Hybrid Fuzzing. *IEEE Transactions on Software Engineering* (2026), 1–16. doi:10.1109/TSE.2026.3694408

[27] Ruijie Meng, Martin Mirchev, Marcel Böhme, and Abhik Roychoudhury. 2024. Large language model guided protocol fuzzing. In *Proceedings of the 31st Annual Network and Distributed System Security Symposium (NDSS)*. doi:10.14722/ndss.2024.24556

[28] Bertrand Meyer. 1997. *Object-Oriented Software Construction* (2nd ed.). Prentice Hall.

[29] Viraaji Mothukuri, Reza M Parizi, and James L Massa. 2024. Llmsmartsec: Smart contract security auditing with llm and annotated control flow graph. In *Proceedings of the 2024 IEEE International Conference on Blockchain (Blockchain)*. 434–441. doi:10.1109/Blockchain62396.2024.00064

[30] Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. 2024. Using an llm to help with code understanding. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. 1–13. doi:10.1145/3597503.3639187

[31] Chao Ni, Liyu Shen, Xiaohu Yang, Yan Zhu, and Shaohua Wang. 2024. MegaVul: A C/C++ Vulnerability Dataset with Comprehensive Code Representations. In *Proceedings of the 21st International Conference on Mining Software Repositories (MSR)*. 738–742. doi:10.1145/3643991.3644886

[32] Yu Nong, Guangbei Yi, Mohammed Aldeen, Long Cheng, Hongxin Hu, and Haipeng Cai. 2026. Assessing and Improving Prompting Large Language Models for Software Vulnerability Analysis. *ACM Trans. Softw. Eng. Methodol.* (June 2026). doi:10.1145/3821416

[33] OpenAI. Accessed: 2026-06. Codex. https://developers.openai.com/codex/.

[34] OpenAI. Accessed: 2026-06. Introducing GPT-5.4. https://openai.com/index/introducing-gpt-5-4/.

[35] Benjamin C. Pierce. 2002. *Types and programming languages*. MIT Press.

[36] Zeeshan Rasheed, Malik Abdul Sami, Muhammad Waseem, Kai-Kristian Kemell, Xiaofeng Wang, Anh Nguyen, Kari Systä, and Pekka Abrahamsson. 2024. Ai-powered code review with llms: Early results. arXiv:2404.18496 [cs.SE] https://arxiv.org/abs/2404.18496

[37] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: a fast address sanity checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference (USENIX ATC)*. 28. doi:10.5555/2342821.2342849

[38] Tao Sun, Jian Xu, Yuanpeng Li, Zhao Yan, Ge Zhang, Lintao Xie, Lu Geng, Zheng Wang, Yueyan Chen, Qin Lin, et al. 2025. Bitsai-cr: Automated code review via llm in practice. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering (FSE Companion)*. 274–285. doi:10.1145/3696630.3728552

[39] Mark Weiser. 1981. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering (ICSE)*. 439–449. doi:10.5555/800078.802557

[40] Yaoxuan Wu, Xiaojie Zhou, Ahmad Humayun, Muhammad Ali Gulzar, and Miryung Kim. 2026. PALM: Path-aware LLM-based Test Generation with Comprehension. arXiv:2506.19287 [cs.SE] https://arxiv.org/abs/2506.19287

[41] Chenyuan Yang, Xuheng Li, Md Rakib Hossain Misu, Jianan Yao, Weidong Cui, Yeyun Gong, Chris Hawblitzel, Shuvendu Lahiri, Jacob R. Lorch, Shuai Lu, Fan Yang, Ziqiao Zhou, and Shan Lu. 2025. AutoVerus: Automated Proof Generation for Rust Code. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 396 (Oct. 2025), 29 pages. doi:10.1145/3763174

[42] Jun Yang, Yuechun Sun, Yi Wu, Rodrigo Caridad, Yongwei Yuan, Jianan Yao, Shan Lu, and Kexin Pei. 2026. ExVerus: Verus Proof Repair via Counterexample Reasoning. In *Proceedings of the 43rd International Conference on Machine Learning*. ICML. https://www.microsoft.com/en-us/research/publication/exverus-verus-proof-repair-via-counterexample-reasoning/

[43] Yixin Yang, Bowen Xu, Xiang Gao, and Hailong Sun. 2025. Context-Enhanced Vulnerability Detection Based on Large Language Models. *ACM Trans. Softw. Eng. Methodol.* (Dec. 2025). doi:10.1145/3779222

[44] Zijie Zhao, Chenyuan Yang, Weidong Wang, Yihan Yang, Ziqi Zhang, and Lingming Zhang. 2026. AnyPoC: Universal Proof-of-Concept Test Generation for Scalable LLM-Based Bug Detection. arXiv:2604.11950 [cs.SE] https://arxiv.org/abs/2604.11950