Prompt Less, Infer More: Static Analysis-Guided LLM Type Annotation for Python

Wonseok Oh
Korea University
Seoul, Republic of Korea
wonseok_oh@korea.ac.kr

Michael Pradel
CISPA Helmholtz Center for Information Security
Stuttgart, Germany
michael@binaervarianz.de

Hakjoo Oh*
Korea University
Seoul, Republic of Korea
hakjoo_oh@korea.ac.kr

***Abstract*—Type annotations improve the maintainability and reliability of Python programs, but many real-world projects remain under-annotated because manual annotation is time-consuming and error-prone. Recent neural and LLM-based type inference systems can recover missing annotations, yet they often trade accuracy for efficiency. Existing approaches often fail to exploit project-wide type evidence, or do so inefficiently through unnecessary model calls and noisy prompts. We present SALT, an LLM-based type inference system guided by static analysis. SALT integrates static analysis with LLM inference to resolve types statically, prioritize LLM queries, and construct concise type-aware context. To demonstrate the effectiveness of SALT, we evaluated it on 12 real-world Python projects. SALT achieves the best accuracy among the evaluated systems and reduces estimated LLM cost by up to 78.5% compared with a strong coding-assistant baseline. It also shows similar cost savings with the frontier model GPT-5.5, demonstrating that SALT remains effective with strong LLMs.**

***Index Terms*—Python type inference, large language models, static analysis, type annotation**

I. INTRODUCTION

Type annotations have become an important part of modern Python development. Although Python is dynamically typed, type annotations allow developers to catch bugs with static checkers and make large codebases easier to maintain [1]. Unfortunately, many Python projects remain under-annotated [2], and manually annotating them is time-consuming and error-prone [3]. This problem has motivated a growing interest in automated type annotation for Python [4]–[22].

Existing Approaches. Recent work has explored specialized type inference tools based on neural [4]–[13] and LLM-based [14], [15] approaches, which show promise in annotating real-world Python code. More recently, general-purpose LLM coding assistants, such as AIDER [23], CLAUDE CODE [24], CODEX CLI [25], GITHUB COPILOT [26], and CURSOR [27], have become a natural alternative. Although these tools are not designed specifically for type inference, type annotation is also a code-editing task: a tool that can inspect a repository and modify code can be prompted to add missing annotations.

We observed that these assistants can achieve strong accuracy on a benchmark of real-world Python projects. As a preview of our results, the strongest existing specialized tool, TYPET5 [12], reaches 69.30% accuracy, while AIDER, an open-source coding assistant, reaches up to 84.13% accuracy. This result arises because AIDER can retrieve and reason about broad repository context, including cross-file evidence, whereas specialized systems tend to focus only on the target function or fail to make use of diverse contextual information.

However, this accuracy comes at a high cost. For a project with about 14k lines of code, AIDER with GLM-4.7 [28] consumed 24.83M input and 0.65M output tokens, costing approximately $57.67[1]. First, AIDER invokes the LLM for every function, even trivial ones such as `def foo(): return 0`. Second, AIDER constructs context through lexical name matching rather than program analysis, producing verbose prompts that often include irrelevant information.

In summary, specialized tools are limited because they fail to capture relevant context, whereas general-purpose coding assistants are inefficient because they trigger unnecessary LLM invocations and flood the prompt with irrelevant information.

This Work. We present SALT, a system that uses **S**tatic **A**nalysis to guide **L**LM-based **T**ype annotation. SALT retains the broad repository-level reasoning of general-purpose coding assistants, but uses static analysis to reduce unnecessary LLM usage. SALT first performs project-wide forward and backward type inference. Forward inference propagates caller-side evidence, while backward inference recovers use-site evidence that flows from callees. When static evidence is sufficiently precise, SALT directly annotates the code without invoking an LLM. For unresolved functions, SALT uses the inferred type information to prioritize high-leverage LLM queries and to construct concise type-aware context for each target function.

Our evaluation on 12 real-world Python projects from TYPYBENCH [29] shows that this design improves both effectiveness and efficiency. We evaluate SALT with three LLM models of different sizes: Gemma 4 [30], gpt-oss [31], and GLM-4.7 [28]. Across three LLM models, SALT improves accuracy over AIDER by 3.5% on average while reducing LLM calls by 35.7%. In addition, SALT reduces estimated cost by 75.7% on average. SALT also remains effective with the frontier model GPT-5.5, reducing estimated cost by 74.8% while slightly improving accuracy over AIDER (also backed by GPT-5.5). Moreover, our ablation study shows that static analysis,

*Corresponding author

[1]We calculate the cost based on the GLM-4.7 API pricing (https://www.cerebras.ai/pricing). Input: $2.25/M, Output: $2.75/M.

query prioritization, and type-aware context construction each improve accuracy and reduce LLM usage.

Contributions. This paper makes the following contributions:

- We show that general-purpose coding assistants can outperform specialized type inference tools, but at substantial cost from unnecessary LLM calls and verbose context.
- We present SALT, which uses static analysis to actively guide LLM-based annotation through (i) forward and backward inference, (ii) query prioritization, and (iii) concise type-aware context.
- We evaluate SALT on 12 real-world Python projects with four LLMs, showing improved accuracy over strong baselines with up to 78.5% lower cost, including with the frontier model GPT-5.5.

II. OVERVIEW OF SALT

SALT addresses the problem of annotating untyped or partially typed Python functions. We focus on function-level annotations rather than variable annotations because prior work has shown that developers mostly annotate functions [2]. For example, given an unannotated function, the goal is to infer its missing parameter and return annotations:

Unannotated

```python
from pathlib import Path
def foo(p):
  return Path(p+".py")
```

⇒

Annotated

```python
from pathlib import Path
def foo(p: str) -> Path:
  return Path(p+".py")
```

SALT aims to make this process both effective and efficient: it should infer accurate annotations by using project-wide evidence, while minimizing unnecessary LLM usage. SALT achieves this by combining static type analysis with LLM-based annotation in a synergistic way.

A. Motivating Example

Prior work has recognized that type prediction benefits from program context beyond the target function. Existing approaches therefore use techniques such as forward type inference for pre-annotation, caller-callee context construction, and callee-first query ordering. However, we found that these techniques remain suboptimal in how they use type information. First, forward type inference captures only caller-to-callee evidence and misses useful evidence that flows backward. Second, existing approaches do not sufficiently use type analysis results when deciding what context to include and which function to query next. Figure 1 illustrates these limitations.

Limitation of Forward Type Inference. We found that forward type inference is insufficient to capture all the relevant evidence in real-world code. For example, HITYPER [8] uses forward type inference to pre-annotate types that flow from callers to callees, such as the parameter `msg` of `pack` at line 15, which is passed string literals at lines 7 and 12. Subsequently, their approach propagates this annotation to the callee `print_log` at line 16, yielding the parameter annotation `msg: str` in `print_log`.

However, since `pack` constructs `k = String()` and calls `k.kind(name)`, the second parameter `name` of `pack` should

```python
1  class String: def kind(self, name: str)   -> str: ...
2  class Bytes:  def kind(self, name: bytes) -> bytes: ...
3  def get_paths() -> list[str]: ...
4
5  def internal(name):
6    x = get_paths()
7    paths = pack(x, name, "i-str")
8    return paths
9
10 def external(name):
11   x = external_func()
12   paths = pack(x, name, "e-str")
13   return paths
14
15 def pack(paths, name, msg):
16   print_log(name, msg)
17   k = String()
18   return [(norm(p), k.kind(name)) for p in paths]
19
20 def norm(path):
21   return path.rstrip()
22
23 def print_log(name, msg) -> None:
24   print(f"{name}: {msg}")
```

Fig. 1. A motivating example where existing approaches either miss useful static evidence or spend unnecessary LLM calls.

be `str`. This evidence comes from the resolved callee `String.kind` and can be propagated backward to `pack`, but a forward-only approach cannot leverage this information.

Limitation of LLM Querying. Existing approaches also fail to maximize the synergy between LLM queries and type analysis.

The first issue is query ordering. Prior work such as TYPET5 [12] considers a callee-first order, where the LLM is queried for leaf functions before their callers. In Figure 1, this strategy would query `norm` at line 20 before `pack`. However, annotating `norm` first has limited leverage: even if the model correctly infers `norm: str -> str`, this annotation does not determine the type of `paths` in `pack`, so another LLM call is still needed for `pack`.

The second issue is context construction. Existing approaches often limit the prompt to directly related caller and callee functions due to prompt size constraints. Suppose an existing approach queries `norm` first. The caller-callee context includes `pack` as the caller and no callees. However, this context still does not reveal the type of parameter `paths` precisely, because `pack` receives `paths` from its callers, such as `internal` and `external`. Without type analysis results that summarize these caller-side facts, the model has limited evidence for inferring the parameter type of `norm`.

Even when querying `pack`, existing context construction is inefficient. TYPET5 [12] and AIDER [23] construct caller-callee context through lexical name matching instead of type analysis. For the call `k.kind(name)` in `pack`, they retrieve all functions named `kind`, including both `String.kind` and the irrelevant `Bytes.kind`. However, only `String.kind` is relevant because `k` is constructed as `String` at line 17, and including `Bytes.kind` introduces noise into the prompt.

B. Illustration of Key Ideas

To address these limitations, SALT combines static analysis with LLM querying in three ways (Figure 2): **(1)** SALT first extends pre-annotation beyond forward propagation by exploiting backward evidence from callees. **(2)** It then uses

Fig. 2. Workflow of SALT. The system repeatedly alternates between static type analysis and LLM annotation until the remaining annotations are resolved.

```
1  class String: def kind(self, name: str)   -> str: ...
2  class Bytes:  def kind(self, name: bytes) -> bytes: ...
3  def get_paths() -> list[str]: ...
4
5  def internal(name: str):
6    x = get_paths()
7    paths = pack(x, name, "i-str")
8    return paths
9
10 def external(name: str):
11   x = external_func()
12   paths = pack(x, name, "e-str")
13   return paths
14
15 def pack(paths, name: str, msg: str):
16   print_log(name, msg)
17   k = String()
18   return [(norm(p), k.kind(name)) for p in paths]
19
20 def norm(path):
21   return path.rstrip()
22
23 def print_log(name: str, msg: str) -> None:
24   print(f"{name}: {msg}")
```

Fig. 3. Type propagation over the motivating example. ⟶ denotes forward propagation, and ⇢ denotes backward propagation.

```
1  def internal(name: str) -> list[tuple[str, str]]:
2    ...
3    paths = pack(x, name, "i-str")
4    return paths
5
6  def external(name: str) -> list[tuple[str, str]]:
7    ...
8    paths = pack(x, name, "e-str")
9    return paths
10
11 def pack(paths: list[str], name: str, msg: str)
12                           -> list[tuple[str, str]]:
13   ...
14   return [(norm(p), k.kind(name)) for p in paths]
15
16 def norm(path: str) -> str: ...
```

Fig. 4. Static propagation after querying `pack`.

the inferred type information to select high-leverage target functions that are worth querying the LLM for. **(3)** Finally, SALT uses the inferred type information to construct type-aware context before querying the model. We illustrate these ideas using the motivating example below.

(1) Combining Forward and Backward Inference. By incorporating backward analysis, SALT can recover evidence that forward propagation alone misses in Figure 1. Figure 3 summarizes how these annotations are propagated.

Forward analysis infers `msg: str` for `pack` and `print_log` from string literals passed by `internal` and `external`. Backward analysis then uses the resolved callee `k.kind(name)` at line 18 to infer `name: str` for `pack` and propagates it back to `internal` and `external`. Moreover, a subsequent round of forward propagation uses the inferred `name: str` to annotate parameter `name` of `print_log` at line 23. Thus, `print_log` becomes fully

```
1  # Type hints from static analysis:
2  # paths <- list[str]   # from internal
3  # paths <- Unknown     # from external
4
5  class String:
6    def kind(self, name: str) -> str: ...
7
8  def internal(name: str):
9    x = get_paths()
10   paths = pack(x, name, "i-str")
11   return paths
12
13 def external(name: str):
14   x = external_func()
15   paths = pack(x, name, "e-str")
16   return paths
17
18 def pack(paths, name: str, msg: str):
19   print_log(name, msg)
20   k = String()
21   return [(norm(p), k.kind(name)) for p in paths]
22
23 def norm(path):
24   return path.rstrip()
25
26 def print_log(name, msg) -> None: ...
```

Fig. 5. Type-guided context construction for querying `pack`. SALT includes the type-resolved callee `String.kind` and concise static hints, without adding irrelevant callees or extra caller-side function bodies.

annotated without an additional LLM query.

A key observation is that backward information should be interpreted together with forward information. For example, consider the following code:

Listing 1: Combining Forward and Backward Evidence

```
1 def foo(path, encoding):
2   if isinstance(encoding, bytes):
3     encoding = encoding.decode()
4   open(path, encoding=encoding)
5
6 foo("src/a.py", "utf-8")
```

The function `open` at line 4 takes a path-like object (`str`, `bytes`, or `PathLike`) for its first argument. If we propagate the type of parameter `path` backward only from `open`, we infer a broad type such as `str | bytes | PathLike`. However, such a broadly inferred type is not useful in practice. Combining this backward evidence with caller-side evidence avoids this over-approximation. Since `foo` is called with the string literal `"src/a.py"`, the type of `path` can be narrowed more precisely to `str`. Conversely, caller-side evidence alone suggests `encoding: str` because `foo` is called with `"utf-8"` at line 6. However, the function body shows that `encoding` can also be `bytes` through the `isinstance(encoding, bytes)` check. Combining these two sources of evidence allows SALT to infer the more appropriate type `str | bytes` for `encoding`.

(2) Prioritizing LLM Calls to Maximize Static Inference. To maximize the leverage of static analysis, we determine the

order of LLM calls based on how many other functions each function can influence.

When selecting a function for an LLM query, SALT considers how many other functions can benefit from its annotation. Suppose that SALT selects the next LLM target after the static propagation shown in Figure 3. In this state, `pack` is a high-impact target because its annotation affects both its callers, `internal` and `external`, and its callee, `norm`. SALT therefore queries the LLM for `pack` first, asking the model to infer the type of `paths` and the return type of `pack`. If this query succeeds (`paths: list[str]` and `return: list[tuple[str, str]]`), subsequent static propagation uses `paths: list[str]` to infer that the first argument of `norm` at line 20 is `str` from the list comprehension at line 18. It also propagates the return type of `pack` to lines 7 and 12, thereby statically annotating the return types of `internal` and `external`. Figure 4 illustrates this propagation.

Thus, by prioritizing LLM calls for functions that can unlock more subsequent static inference, SALT can annotate all functions in the example with a single LLM call.

(3) Type-Guided Context Construction. By leveraging the type analysis, we can construct context that is both effective and efficient. The key idea is twofold: (1) precisely identifying relevant functions through type analysis and (2) providing type hints about incoming values.

Suppose that SALT selects the function `pack` at line 15 as the next LLM target. For the call `k.kind(name)` in `pack` at line 18, type analysis determines that `k` has type `String`, so SALT includes `String.kind` while omitting irrelevant functions with the same lexical name, such as `Bytes.kind`.

Next, prompt size constraints often prevent a tool from including all related caller and callee bodies. To compensate, SALT provides type hints summarized from static analysis. For example, we consider the caller context of `pack`, which includes `internal` and `external`. However, to know that `paths` has type `list[str]`, a context construction method needs to include the function `get_paths` used by `internal`. Instead of adding this extra function body to the prompt, SALT summarizes the types of values that flow into `paths`. Using type analysis, SALT can determine that the call at line 7 passes a `list[str]` value to `paths`, while the call at line 12 passes an unknown value. Providing this summary conveys the useful evidence about `paths` without including additional function bodies in the prompt. Figure 5 shows this context construction for function `pack`.

III. ALGORITHM

Preliminaries. We define a program $P \in \mathit{Program}$ as a set of function declarations. A function declaration M is defined as a tuple $(m, \overrightarrow{p})$, where m is the function name and $\overrightarrow{p} = (p_1, \ldots, p_n)$ is a sequence of parameters. We write $\mathsf{Names}(P)$ for the set of function names declared in P. We define a type domain $\mathcal{T}$ as follows:

$$\mathcal{T} = \mathit{BuiltInTypes} \cup \mathit{CustomTypes} \cup \{\mathsf{Unknown}\}$$

Algorithm 1 Overall algorithm of SALT

Require: Program P, the number of queries k
Ensure: Annotation map A

1: $A \leftarrow [s \mapsto \{\mathsf{Unknown}\} \mid s \in \mathsf{slots}(P)]$
2: $H \leftarrow ([\,], [\,], [\,])$ (empty evidence maps)
3: **while** $\mathsf{Unresolved}(P, A) \neq \emptyset$ **do**
4: **repeat**
5: $A_{old}, H_{old} \leftarrow A, H$
6: $H \leftarrow \mathsf{TypeAnalysis}(P, A)$ (Sec III-A)
7: $A \leftarrow \mathsf{Update}(P, A, H)$
8: **until** $(A, H) = (A_{old}, H_{old})$
9: **if** $\mathsf{Unresolved}(P, A) = \emptyset$ **then**
10: **break**
11: **end if**
12: $M \leftarrow \mathsf{TopFuncs}(P, A, H, k)$ (Sec III-B)
13: **for all** $m \in M$ **do**
14: $C \leftarrow \mathsf{Ctx}(P, A, H, m)$ (Sec III-C)
15: $A \leftarrow \mathsf{QueryLLM}(C, A, m)$
16: **end for**
17: **end while**
18: **return** A

where *BuiltInTypes* is the set of built-in types (e.g., `int`, `str`, `list`), *CustomTypes* is the set of user-defined types (e.g., `MyClass`), and Unknown is a special element representing an unresolved type. We write $t \in \mathcal{T}$ for a single type and $T \subseteq \mathcal{T}$ for a set of types.

Problem. Given an untyped program P, our goal is to compute an annotation map A:

$$A \in \mathit{AnnotMap} \qquad A : \mathsf{slots}(P) \to 2^{\mathcal{T}}$$

where $\mathsf{slots}(P)$ is the set of parameter type slots:

$$\mathsf{slots}(P) = \{(m, i) \mid (m, \overrightarrow{p}) \in P \text{ and } 1 \leq i \leq |\overrightarrow{p}|\}.$$

A slot $s = (m, i)$ denotes the i-th parameter of function m; return slots are handled analogously and omitted for simplicity.

Overall Algorithm. Algorithm 1 presents the overall workflow of SALT. SALT first initializes every slot as Unknown (line 1). As described in Section II, we first perform static type analysis to collect forward and backward evidence (line 6), select high-leverage functions for LLM queries (line 12), and construct type-aware context for each selected function (line 14). We repeat this process until all slots are resolved (line 3–17). We define $\mathsf{Unresolved}(P, A)$ as the set of unresolved methods:

$$\mathsf{Unresolved}(P, A) = \{\, m \mid \exists i.\ (m, i) \in \mathsf{slots}(P) \wedge A((m, i)) = \{\mathsf{Unknown}\} \,\}$$

The following subsections detail each step.

A. Forward and Backward Type Analysis

To infer annotations for the slots in A, SALT follows a two-step process. First, forward and backward analyses collect type candidates for each slot from caller-side information and use-site evidence, respectively (line 6). The result is an

evidence tuple $H = (F, B_s, B_w)$, where F is produced by forward analysis and B_s and B_w are produced by backward analysis. As illustrated in Listing 1, backward evidence is split into strong evidence B_s, which can be used directly, and weak evidence B_w, which is informative but must be used conservatively.

Second, SALT combines these evidence maps to determine which annotations can be added directly to A (line 7).

Forward Analysis. Forward analysis collects caller-side type candidates in a map F:

$$F : \mathsf{slots}(P) \to 2^{\mathcal{T}}.$$

For a slot $s = (m, i)$, $F(s)$ summarizes the types of values that callers pass to the i-th parameter of m.

SALT analyzes each function independently. For a function $(m, \overrightarrow{p})$ with $\overrightarrow{p} = (p_1, \ldots, p_n)$, the analysis initializes a local type environment $\Gamma_m : \mathit{Var} \to 2^{\mathcal{T}}$ from the annotation map:

$$\Gamma_m(p_i) = A(s_i) \quad \text{where } s_i = (m, i) \text{ and } 1 \le i \le n.$$

It then propagates types through standard typing rules, updating Γ_m as it encounters assignments and expressions.

The key update happens at call sites. When the analysis reaches a call site ℓ, it uses the current local type environment to resolve the callee when possible. If the callee is resolved to a function m_ℓ and there is an argument a_i passed to the i-th parameter of m_ℓ, SALT records this caller-side evidence by updating:

$$F(s_i^\ell) \leftarrow F(s_i^\ell) \cup \Gamma_m(a_i) \quad \text{where } s_i^\ell = (m_\ell, i).$$

This update is applied only when the analysis can identify the target function precisely enough. The forward analysis continues until all functions have been analyzed, producing a complete map of caller-side type candidates.

Backward Analysis. Backward analysis collects use-site evidence in two maps:

$$B_s, B_w : \mathsf{slots}(P) \to 2^{\mathcal{T}}.$$

B_s stores explicit evidence from direct type checks in the function body, such as `isinstance(x, t)`. B_w stores indirect evidence from callee parameter types, which describe accepted inputs rather than directly constraining the caller's argument type.

For each function $(m, \overrightarrow{p})$, SALT traverses paths from return statements backward to the function entry. It maintains two local evidence maps, Γ_m^s and Γ_m^w, which track strong and weak evidence for local variables, respectively:

$$\Gamma_m^s, \Gamma_m^w : \mathit{Var} \to 2^{\mathcal{T}}.$$

After the traversal finishes, SALT transfers the evidence accumulated for each parameter p_i to the corresponding global slot $s_i = (m, i)$ for $1 \le i \le |\overrightarrow{p}|$:

$$B_s(s_i) \leftarrow B_s(s_i) \cup \Gamma_m^s(p_i) \quad B_w(s_i) \leftarrow B_w(s_i) \cup \Gamma_m^w(p_i).$$

Example 1. Consider the following program:

```
1 def g(z: int | str): ...
2 def f(x, y):
3     z = x
4     if isinstance(z, int):
5         return z
6     else:
7         return g(y)
```

The guard `isinstance(z, int)` yields strong evidence for `z`, assigning int to $\Gamma_f^s(z)$. Following the assignment from `z` back to the first parameter `x`, the analysis records this evidence as $B_s(s_1^f) = \{\mathsf{int}\}$, where $s_1^f = (f, 1)$. In contrast, `g(y)` yields weak evidence for `y` because `g` expects an `int` or `str` value as its first argument. The analysis therefore adds $\{\mathsf{int}, \mathsf{str}\}$ to $\Gamma_f^w(y)$ and records it as $B_w(s_2^f) = \{\mathsf{int}, \mathsf{str}\}$, where $s_2^f = (f, 2)$.

Type Annotation from Evidence. After collecting evidence, SALT updates A with $\mathsf{Update}(P, A, H)$ (line 7):

$$\mathsf{Update}(P, A, H) = A[s \mapsto \mathsf{Concrete}(s, A, H) \mid s \in \mathsf{slots}(P)]$$

Here, Concrete replaces $A(s)$ only when the evidence yields a fully concrete type set; otherwise, it preserves $A(s)$:

$$\mathsf{Concrete}(s, A, H) = \begin{cases} \mathbb{T}(s, H) & \text{if } \mathsf{Unknown} \notin \mathbb{T}(s, H) \\ A(s) & \text{otherwise} \end{cases}$$

$\mathbb{T}(s, H)$ combines evidence to determine the final set of candidate types for slot s:

$$\mathbb{T}(s, (F, B_s, B_w)) = \begin{cases} B_s(s) \cup \{t\} & \text{if } B_w(s) = \{t\} \text{ and } F(s) = \{\mathsf{Unknown}\}, \\ B_s(s) \cup F(s) & \text{otherwise} \end{cases}$$

For direct annotation, we use forward and strong backward evidence. Weak backward evidence is used only when it yields a single candidate type and no forward evidence is available.

We repeat the type analysis and annotation update until no further changes occur (lines 4–8).

B. Prioritizing LLM Calls

SALT invokes the LLM for functions that still contain unresolved slots. The order of these calls matters because an LLM-generated annotation can restart static propagation and enable further static inference without additional LLM calls.

SALT selects the k functions with the highest priority for LLM queries (line 12):

$$\mathsf{TopFuncs}(P, A, H, k) = \mathsf{Top}_k(\Pi(P, A, H))$$

where $\Pi(P, A, H)$ associates each unresolved function with its priority scores:

$$\Pi(P, A, H) = \{(m, \mathsf{Impact}(P, A, m), \mathsf{Richness}(P, H, A, m)) \mid m \in \mathsf{Unresolved}(P, A)\}$$

Impact measures how many unresolved call-graph neighbors a function has, while Richness estimates the informativeness of its type hints. Selecting the top-k functions by these scores prioritizes calls likely to unlock static propagation and provide rich evidence.

Impact Score. The impact score of a function m is defined as the number of unresolved functions that are directly connected to m in the call graph. SALT constructs a call graph $\mathcal{G}_A$ using the current annotation map A:

$$\mathcal{G}_A \subseteq \mathit{Methods} \times \mathit{Methods},$$

where $(m, n) \in \mathcal{G}_A$ means that function m calls function n. The call graph can be incomplete if some types are still unresolved. SALT intentionally omits such unresolved edges because speculative edges do not support static propagation, which is the objective of prioritizing LLM calls.

SALT calculates the impact score of a function m as follows:

$$\mathsf{Impact}(P, A, m) = \\ |\{\, n \in \mathsf{Unresolved}(P, A) \mid (n, m) \in \mathcal{G}_A \vee (m, n) \in \mathcal{G}_A \,\}|$$

A high score indicates annotating m is likely to unlock forward propagation to callees or backward propagation to callers.

Richness Score. As a tie breaker, SALT uses a richness score that estimates the informativeness of type hints for the unresolved slots of m. We define $\phi(P, A, m)$ as the set of unresolved slots for function m:

$$\phi(P, A, m) = \{\, (m, i) \mid (m, i) \in \mathsf{slots}(P) \\ \wedge A((m, i)) = \{\mathsf{Unknown}\} \,\}.$$

Then, the richness score is the average hint score over these unresolved slots:

$$\mathsf{Richness}(P, H, A, m) = \frac{\sum_{s \in \phi(P,A,m)} \mathsf{Hint}(H, s)}{|\phi(P, A, m)|}.$$

The function Hint assigns a higher score when both caller-side hints and callee-side evidence contain concrete types, and a lower score when the evidence is partial or unknown:

$$\mathsf{Hint}((F, B_s, B_w), s) = \\ \begin{cases} 1.00 & \mathsf{IsConcrete}(F(s)) \wedge \mathsf{IsConcrete}(B(s)) \\ 0.75 & \mathsf{IsConcrete}(F(s)) \wedge \mathsf{IsPartial}(B(s)) \\ 0.75 & \mathsf{IsPartial}(F(s)) \wedge \mathsf{IsConcrete}(B(s)) \\ 0.50 & \mathsf{IsPartial}(F(s)) \wedge \mathsf{IsPartial}(B(s)) \\ 0.25 & \mathsf{IsPartial}(F(s)) \wedge B(s) = \{\mathsf{Unknown}\} \\ 0.00 & \text{otherwise} \end{cases} \\ \text{where } B(s) = B_s(s) \cup B_w(s)$$

The function $\mathsf{IsConcrete}(T)$ returns true if T contains only concrete types, while $\mathsf{IsPartial}(T)$ returns true if T contains at least one concrete type and at least one Unknown.

The intuition is that clearer type evidence is more useful to the LLM. We give slightly higher priority to slots with concrete forward hints, because weak backward evidence reflects callee-side expectations and may lead the model to choose overly broad types.

C. Type-Aware Context Construction

For each method selected by TopFuncs, SALT builds a prompt from code context and type-hint context (line 14):

$$\mathsf{Ctx}(P, A, H, m) = \\ (\mathsf{CodeCtx}(A, m), \{\mathsf{TypeHint}(s, H) \mid s \in \phi(P, A, m)\})$$

where CodeCtx returns relevant code for m and TypeHint returns evidence for slot s.

Code Context. Let m be the target method. The code context includes caller and callee context resolved using the call graph $\mathcal{G}_A$. We define the caller and callee sets as follows:

$$\mathsf{Callers}(A, m) = \{n \mid (n, m) \in \mathcal{G}_A\} \\ \mathsf{Callees}(A, m) = \{n \mid (m, n) \in \mathcal{G}_A\}$$

If the call graph is incomplete due to unresolved types, SALT falls back to lexical name matching to find additional callers and callees for the code context.

For callees, SALT includes the callee body only when the callee still has an unresolved signature; otherwise, its current signature is sufficient:

$$\mathsf{CalleeCtx}(n) = \begin{cases} \mathsf{Signature}(n) & \text{if } n \text{ is fully annotated} \\ \mathsf{Body}(n) & \text{otherwise} \end{cases}$$

where $\mathsf{Signature}(n)$ returns the signature of method n and $\mathsf{Body}(n)$ returns the body of method n.

Thus, the code context for m is the union of the target method, the caller bodies, and the callee contexts:

$$\mathsf{CodeCtx}(A, m) = \mathsf{Body}(m) \\ \cup \{\mathsf{Body}(n) \mid n \in \mathsf{Callers}(A, m)\} \\ \cup \{\mathsf{CalleeCtx}(n) \mid n \in \mathsf{Callees}(A, m)\}$$

Type-Hint Context. The second part of the prompt summarizes the type evidence collected by static analysis. For each unresolved parameter slot $s \in \phi(P, A, m)$ of the target method m, SALT reports the candidate types suggested by caller-side propagation and callee-side evidence:

$$\mathsf{TypeHint}(s, (F, B_s, B_w)) = \big(s, F(s), B_s(s), B_w(s)\big).$$

Intuitively, $F(s)$ describes types observed at call sites, $B_s(s)$ describes direct constraints from the body of m, and $B_w(s)$ describes indirect constraints obtained from callees.

Querying the LLM. Finally, SALT queries the LLM with the constructed context to obtain type predictions for the unresolved slots of m (line 15). SALT asks the LLM to predict a set of types for each unresolved slot, and it updates the annotation map A with the predictions.

IV. EVALUATION

We evaluate SALT through the following research questions:

- **RQ1. Effectiveness and Efficiency:** How does SALT compare to type inference baselines in terms of annotation accuracy and LLM usage?
- **RQ2. Generalization to a Frontier LLM:** Does SALT remain effective when paired with a frontier LLM?
- **RQ3. Ablation Study:** How do different components of SALT contribute to its performance?
- **RQ4. Case Study:** What are representative examples where SALT succeeds or fails?

[2]pre-commit-hooks

TABLE I: Dataset statistics for the selected TYPYBENCH projects.

Project	Funcs	Slots	Files	Lines
black	607	1,453	56	96,673
fastapi	274	1,717	53	11,076
flake8	203	462	35	3,039
flask	326	786	23	4,008
poetry	865	1,825	144	14,012
pre-commit	328	909	61	5,193
pc-hooks[2]	104	228	34	1,799
private-gpt	183	451	56	4,307
rich	866	2,372	81	19,960
typer	149	698	19	4,043
urllib3	451	1,232	40	7,816
vnpy	410	851	31	4,216
Total	4,766	12,984	633	176,142

TABLE II: LLM models used in our evaluation. Prices are shown as dollars per million input/output tokens.

Model	Params	Source	Serving	Price
Gemma 4 [30]	E4B-IT	Google	Local	–
gpt-oss [31]	120B	OpenAI	Cerebras API	$0.35/$0.75
GLM-4.7 [28]	358B	Z.ai	Cerebras API	$2.25/$2.75
GPT-5.5	–	OpenAI	OpenAI API	$5.00/$30.00

A. Experimental Setup

SALT is implemented in about 15,000 lines of Python 3.13.12 code on top of AIDER (v.0.86.3) [23]. SALT uses JEDI (v.0.20.0) [32] for call resolution and PYREFLY (v.1.0.0) [16] for return type inference. We run all experiments on a Linux machine (Debian 13) with an Intel Xeon Silver 4214 CPU, an NVIDIA RTX A6000 GPU, and 128 GB of memory.

Dataset. We conduct a repository-level evaluation on 12 high-quality projects from TYPYBENCH [29], removing all type annotations and inferring types for all functions in each project. We select projects that satisfy the following criteria: at least 3,000 GitHub stars, at least 95% function-level annotation coverage, and between 100 and 1,000 functions to keep the number of LLM API calls and evaluation cost manageable. Table I summarizes the selected projects, which include diverse domains such as web frameworks, code formatters, and data processing libraries. Next, we remove all type annotations from these projects to create an untyped codebase for evaluation. We compute accuracy over normalized type annotations, since the same Python type can be written in multiple syntactic forms, such as `Union[int, None]` and `Optional[int]`.

LLM Models. For RQ1, we evaluate SALT with Gemma 4, gpt-oss, and GLM-4.7; the RQ3 ablations use gpt-oss. Gemma 4 runs on a local machine, while gpt-oss and GLM-4.7 are accessed through the Cerebras API[3]. For RQ2, we use GPT-5.5 through the OpenAI API as a frontier LLM. Table II summarizes the model size, serving environment, and pricing for each model.

We use temperature 0 for all models except GLM-4.7. For GLM-4.7, we set temperature 1 because temperature 0 frequently causes non-terminating responses in our experiments. When an LLM response is not parseable, we query the model again, allowing at most two retries. If the response still cannot be parsed, we skip the function.

Baselines. We compare SALT against the following baselines:

- **TIGER** [13]: a neural type inference system that uses a graph neural network to predict types.
- **TYPET5** [12]: a sequence-to-sequence model that generates type annotations.
- **TYPEGEN** [15]: an LLM-based system that generates type annotations.
- **AIDER** [23]: a general-purpose LLM coding assistant that can be prompted to add type annotations.

For the end-to-end evaluation, we exclude HITYPER [8] and NESTER [14] because we could not obtain runnable prediction pipelines from their released artifacts. In our attempts, the Type4Py-based Docker server [9] used by HITYPER failed during installation, and the artifact for NESTER appeared to require external files that were not included.

Thus, we select TIGER, TYPET5, and TYPEGEN as reproducible state-of-the-art specialized baselines in Python, covering recent approaches that predict individual type slot annotations, infer complete function signatures, and use LLM-based generation. In addition, we use AIDER as the general-purpose coding assistant baseline due to its open-source, model-agnostic, and scriptable design. This allows us to build SALT on the same infrastructure and isolate the effect of our techniques under controlled, multi-model experiments.

For each baseline, we follow the prediction granularity used in its original evaluation. TIGER and TYPEGEN predict one type at a time for each argument or return slot. In contrast, TYPET5 and AIDER infer all missing annotations in a target function in a single prediction step.

For an ablation study, we also compare SALT against two static type inference tools: HITYPER [8] and PYREFLY [16]. Although HITYPER does not provide access to its language model, its static analysis component can be run independently to produce type annotations. In addition, we include PYREFLY as a representative industrial type inference tool.

B. RQ1. Effectiveness and Efficiency

Table III compares the accuracy and LLM usage of SALT against the baselines on the benchmark. AIDER outperforms the specialized type inference tools, making it the strongest baseline in our evaluation. SALT further improves over AIDER, achieving higher accuracy for all evaluated model families. The gain is most pronounced for the smaller local model, where accuracy improves from 72.03% to 76.76%.

The results also show that the accuracy gains do not come from using more LLM calls. Instead, SALT substantially reduces model usage by resolving types statically and querying the LLM for high-leverage targets. With gpt-oss, the number of calls decreases by 33.1% and the estimated price drops from $35.25 to $9.59, a 72.8% reduction. With GLM-4.7, the estimated price decreases from $285.08 to $61.19, a 78.5% reduction. The runtime benefit is largest for Gemma 4, where

[3]https://www.cerebras.ai

TABLE III: The accuracy and LLM usage on TYPYBENCH. **Calls**: the number of LLM queries. **Accuracy**: the percentage of correctly inferred type annotations. **Time**: the total time taken for type inference. **Input**: the total number of input tokens. **Output**: the total number of output tokens. **Price**: the estimated cost of LLM usage.

System	Model	Accuracy	Time (hh:mm)	LLM Usage			
				Calls	Input (M)	Output (M)	Price ($)
TIGER	CodeT5 (Fine-tuned)	49.83%	17:22	–	–	–	–
TYPET5	CodeT5 (Fine-tuned)	69.30%	01:53	–	–	–	–
TYPEGEN	gpt-oss	46.51%	08:00	13,702	21.48	4.37	$10.79
AIDER	Gemma 4	72.03%	26:12	5,507	126.83	1.09	–
SALT	Gemma 4	(↑6.6%) **76.76%**	(↓32.6%) **17:40**	(↓40.1%) **3,300**	(↓79.0%) **26.63**	(↓69.9%) **0.33**	–
AIDER	gpt-oss	82.41%	15:12	4,733	95.33	2.52	$35.25
SALT	gpt-oss	(↑2.0%) **84.06%**	(↓6.0%) **14:17**	(↓33.1%) **3,167**	(↓76.1%) **22.78**	(↓14.2%) **2.16**	(↓72.8%) **$9.59**
AIDER	GLM-4.7	84.13%	16:15	4,752	119.16	3.77	$285.08
SALT	GLM-4.7	(↑2.0%) **85.82%**	(↓4.4%) **15:32**	(↓33.8%) **3,146**	(↓80.9%) **22.77**	(↓41.2%) **2.22**	(↓78.5%) **$61.19**

TABLE IV: Per-project comparison between AIDER and SALT on TYPYBENCH. **Accuracy**: the percentage of correctly inferred type annotations. **Calls**: the number of LLM queries. **Tokens**: the total number of input and output tokens.

Project	Gemma 4						gpt-oss						GLM-4.7					
	Accuracy		Calls		Tokens (M)		Accuracy		Calls		Tokens (M)		Accuracy		Calls		Tokens (M)	
	AIDER	SALT	AIDER	SALT	AIDER	SALT	AIDER	SALT	AIDER	SALT	AIDER	SALT	AIDER	SALT	AIDER	SALT	AIDER	SALT
black	67.79%	**71.16%**	649	**418**	18.98	**3.15**	78.05%	**79.49%**	555	**398**	14.37	**2.81**	**79.97%**	79.90%	558	**396**	15.01	**2.83**
fastapi	53.58%	**54.63%**	336	**238**	8.57	**2.03**	65.06%	**74.61%**	269	**217**	6.06	**1.80**	76.70%	**79.21%**	281	**212**	6.43	**1.76**
flake8	81.39%	**85.28%**	232	**126**	2.84	**0.56**	84.42%	**87.01%**	205	**122**	2.30	**0.56**	83.77%	**87.66%**	204	**122**	2.36	**0.55**
flask	**66.67%**	63.99%	393	**286**	9.26	**1.92**	**78.24%**	76.21%	330	**262**	7.10	**1.74**	79.77%	**80.41%**	328	**259**	7.14	**1.75**
poetry	81.26%	**85.15%**	995	**579**	23.54	**4.54**	**90.19%**	89.81%	871	**557**	18.76	**4.27**	90.14%	**90.58%**	872	**560**	25.49	**4.31**
pre-commit	80.75%	**86.91%**	371	**220**	6.54	**0.83**	90.32%	**90.98%**	332	**205**	5.16	**0.81**	84.82%	**90.87%**	331	**206**	5.25	**0.80**
pc-hooks	90.35%	**91.67%**	105	**45**	0.68	**0.12**	93.86%	**94.74%**	105	**44**	0.63	**0.13**	92.98%	**94.74%**	101	**44**	0.63	**0.13**
private-gpt	80.27%	**87.80%**	219	**151**	3.46	**0.93**	89.14%	**89.58%**	187	**150**	2.65	**0.91**	89.58%	**91.13%**	188	**150**	2.69	**0.89**
rich	74.33%	**84.06%**	1,035	**577**	28.74	**7.70**	87.69%	**88.07%**	871	**566**	21.52	**7.13**	89.04%	**90.56%**	885	**560**	37.92	**7.23**
typer	59.03%	**63.75%**	181	**109**	2.60	**0.74**	78.37%	**78.94%**	156	**99**	1.96	**0.64**	78.94%	**80.09%**	149	**100**	1.90	**0.65**
urllib3	69.97%	**77.52%**	510	**270**	13.79	**2.34**	79.30%	**81.25%**	439	**273**	10.63	**2.20**	79.30%	**82.55%**	441	**263**	11.40	**2.14**
vnpy	85.31%	**88.60%**	481	**281**	8.92	**2.10**	**88.95%**	88.13%	413	**274**	6.70	**1.95**	**89.19%**	88.72%	414	**274**	6.73	**1.94**
Overall	72.03%	**76.76%**	5,507	**3,300**	127.92	**26.95**	82.41%	**84.06%**	4,733	**3,167**	97.85	**24.94**	84.13%	**85.82%**	4,752	**3,146**	122.93	**24.98**

time decreases from 26:12 to 17:40. This suggests that static analysis is especially useful when model serving is relatively slow, such as on a local machine.

Table IV presents a per-project comparison. To assess whether SALT outperforms AIDER, we apply Wilcoxon signed-rank tests [33] to the paired project-level results. SALT reduces LLM calls and token usage in all 36 project–model pairs, with statistically significant reductions for each model ($p < 0.001$ for both metrics). For accuracy, the p-values are 0.001, 0.055, and 0.002 for Gemma 4, gpt-oss, and GLM-4.7, respectively, indicating significant improvements for Gemma 4 and GLM-4.7. Overall, SALT achieves consistent cost savings and generally improved accuracy.

C. RQ2. Generalization to a Frontier LLM

To evaluate whether SALT remains effective with a stronger frontier LLM, we additionally compare SALT and AIDER using GPT-5.5. Due to the high cost of frontier-model evaluation, we sample 10% of files from each project, remove type annotations only from the sampled files, and evaluate both systems on these files.

Table V shows that SALT continues to provide benefits even with a much stronger model. In the sampled-file setting, SALT improves accuracy from 92.71% to 93.09% while reducing LLM calls from 640 to 448. The cost reduction is more substantial: the estimated price decreases from $73.42 to $18.49, a 74.8% reduction, on the 10% sampled files. These results suggest that static analysis remains useful even when the underlying LLM is highly capable, primarily by avoiding unnecessary calls and reducing the amount of context sent to

TABLE V: Per-project comparison between AIDER and SALT with GPT-5.5 on the 10% sampled-file setting. **Sample Slots**: the number of evaluated annotation slots in the sampled files. **Accuracy**: the percentage of correctly inferred type annotations. **Calls**: the number of LLM queries. **Price**: estimated with $5/M input tokens and $30/M output tokens.

Project	Sample Slots	Accuracy		Calls		Price ($)	
		AIDER	SALT	AIDER	SALT	AIDER	SALT
black	325	89.23%	**90.77%**	119	**93**	21.43	**5.38**
fastapi	52	**94.23%**	**94.23%**	12	**11**	1.46	**0.60**
flake8	115	92.17%	**93.04%**	48	**29**	3.24	**0.85**
flask	160	88.12%	**88.75%**	63	**44**	7.91	**1.72**
poetry	160	**98.12%**	96.25%	90	**57**	10.25	**1.96**
pre-commit	130	**96.15%**	**96.15%**	52	**34**	4.53	**0.70**
pc-hooks	30	90.00%	**96.67%**	13	**5**	0.44	**0.08**
private-gpt	38	97.37%	**100.00%**	15	**13**	1.08	**0.29**
rich	82	**100.00%**	97.56%	34	**23**	3.54	**1.48**
typer	133	**90.23%**	87.97%	29	**25**	2.64	**1.34**
urllib3	138	**94.93%**	92.75%	61	**44**	7.49	**1.79**
vnpy	243	92.18%	**95.06%**	104	**70**	9.41	**2.31**
Overall	1,606	92.71%	**93.09%**	640	**448**	73.42	**18.49**

TABLE VI: Ablation results with gpt-oss on the 12 projects. **Accuracy**: the percentage of correctly inferred type annotations. **Tokens**: the total number of input and output tokens. **Calls**: the number of LLM queries. **Price**: the estimated API cost at $0.35/M input tokens and $0.75/M output tokens.

Configuration	Accuracy	Calls	Tokens (M)	Price ($)
ForwardOnly	81.16%	3,666	29.00	11.18
NoStaticFill	83.06%	5,069	36.83	14.05
NoFeedback	83.00%	3,814	31.50	12.07
CalleeFirst	82.45%	3,373	26.79	10.31
NoCompression	83.31%	3,554	74.10	26.89
Salt	**84.06%**	**3,167**	**24.94**	**9.59**

the model.

D. RQ3. Ablation Study

We evaluate five ablated configurations against the full Salt using gpt-oss on the 12 projects.

Contribution of Static Analysis. We examine the roles of backward inference, direct static annotation, and feedback propagation through three configurations:

- **ForwardOnly** uses Pyrefly [16] for forward type analysis without backward inference.
- **NoStaticFill** disables direct annotation by static analysis while retaining static hints and prioritization.
- **NoFeedback** performs the initial static analysis and filling, then queries the remaining functions without propagating subsequent LLM predictions through static analysis.

Compared with the full Salt configuration, ForwardOnly has the largest accuracy drop, from 84.06% to 81.16%. NoStaticFill increases calls by 60.1%, highlighting the benefit of direct static annotation. NoFeedback increases calls by 20.4%, showing that propagating LLM predictions further reduces the query workload.

Contribution of LLM Query Design. We examine query prioritization and context compression through two configurations:

- **CalleeFirst** replaces our query prioritization with a callee-first order.
- **NoCompression** uses the default context of Aider without omitting irrelevant code or replacing supporting code with static hints.

CalleeFirst lowers accuracy from 84.06% to 82.45% and requires 6.5% more LLM calls than Salt. This result suggests that our query prioritization benefits both prediction accuracy and query efficiency. NoCompression uses nearly three times as many tokens as the full Salt configuration, highlighting how context compression using type information reduces query costs.

Static Annotation Quality. To demonstrate the quality of our static analysis component, we evaluate its effectiveness before invoking the LLM. Specifically, we run the static analysis once on the initial untyped codebase and measure both recall and precision.

TABLE VII: Recall and precision of static type inference tools. **Rec.**: the percentage of all annotation slots correctly resolved by static analysis. **Prec.**: the percentage of correctly inferred type annotations among all inferred annotations.

Project	HiTyper		Pyrefly		Salt	
	Rec.	Prec.	Rec.	Prec.	Rec.	Prec.
black	8.74%	60.19%	25.12%	92.64%	**30.35%**	**96.29%**
fastapi	**15.14%**	23.70%	5.77%	83.19%	8.79%	**87.79%**
flake8	17.32%	57.14%	29.22%	84.91%	**39.83%**	**96.84%**
flask	13.23%	44.83%	**22.90%**	83.33%	22.52%	**90.31%**
poetry	20.55%	60.29%	33.42%	93.99%	**34.41%**	**96.47%**
pre-commit	16.50%	49.50%	23.43%	92.61%	**34.43%**	**99.05%**
pc-hooks	32.02%	57.94%	36.84%	92.31%	**64.47%**	**99.32%**
private-gpt	23.50%	55.50%	**30.38%**	92.57%	26.39%	**95.97%**
rich	24.45%	50.88%	**29.09%**	86.57%	28.08%	**94.74%**
typer	11.03%	43.75%	12.75%	81.65%	**15.62%**	**87.90%**
urllib3	24.27%	44.49%	**26.87%**	85.75%	25.97%	**94.96%**
vnpy	37.49%	76.87%	**39.13%**	94.33%	33.14%	**98.26%**
Overall	19.64%	47.89%	25.15%	89.45%	**27.24%**	**95.44%**

Table VII compares the recall and precision of the static type inference in Salt with those of the HiTyper and Pyrefly baselines. Salt achieves the highest recall and precision, correctly resolving 27.24% of all slots with 95.44% precision. This shows that Salt can infer a substantial number of correct type annotations through static analysis. At the same time, the limited recall indicates that static analysis alone cannot resolve most annotation slots, motivating the need for LLM queries.

Type Hint Quality. We investigate the usefulness of static type hints as context for the LLM. We measure the caller and callee code that the LLM would need to inspect to infer types of parameters without static hints. This code averages about 1,830 tokens per parameter, compared with 26 tokens for a static hint. Thus, static type hints reduce the token cost of supplying type information.

Among the hinted slots across the three models in RQ1, 47.6% receive hints that exactly match the reference type. On these slots, Salt achieves 9.5% higher exact-match accuracy than Aider. Incorrect hints account for 44.9% of hinted slots, where Salt's accuracy is 3.0% lower than Aider's on average. Overall, Salt achieves an average relative accuracy gain of 4.1% over Aider on hinted slots, suggesting that static hints generally provide useful context for type prediction.

E. RQ4. Case Study

***Strengths of* Salt.** To understand the accuracy gains of Salt, we inspect the GLM-4.7 cases where Salt correctly predicts 678 slots missed by Aider. Among 100 sampled slots, 54 involve complex generic or class types that often require evidence beyond the target function. We find two recurring failure modes: the evidence is either outside the retrieved context or present but not used correctly by the model.

In Figure 6a, suppose the target function is `get_definitions` at line 7, which takes a parameter `fields`. Salt propagates the return type of `get_fields` at line 1 through `get_openapi` at line 3 to infer `fields: list[ModelField]` in `get_definitions`. By contrast, Aider does not retrieve `get_fields` because

```python
def get_fields() -> list[ModelField]: ...

def get_openapi():
  all_fields = get_fields()
  get_definitions(all_fields)

def get_definitions(fields):
  for field in fields: ...
```

(a) A FastAPI example. The target function is `get_definitions`. While SALT infers `list[ModelField]`, AIDER predicts `Sequence[ModelField]` for parameter `fields`.

```python
def handle():
  publisher = Publisher()
  cert = Path(...) if ... else None
  publisher.publish(cert)

class Publisher:
  def publish(self, cert=None):
    certificates = external_func()
    resolved_cert = cert or certificates.cert
```

(b) A poetry example. The target function is `publish`. While SALT infers `Union[Path, None]`, AIDER predicts `Union[str, None]` for parameter `cert`.

Fig. 6. Examples where SALT succeeds but AIDER fails.

the evidence is one step beyond the direct caller, and predicts the broader `Sequence[ModelField]` from the loop over `fields` at line 8.

In Figure 6b, let the target function be `publish` at line 7, which takes a parameter `cert`. SALT observes that `handle` at line 1 passes either a `Path` object or `None` to `Publisher.publish`, so this caller-side evidence is propagated to the parameter `cert`. Even though AIDER has access to the caller `handle` in its context, its reasoning trace describes `cert` as a "certificate file path" and concludes that it "could be Union[str, None]". This suggests that the LLM is biased toward the common convention that certificate paths are represented as strings, even when the caller provides more specific evidence. In contrast, static analysis preserves that concrete evidence across call boundaries.

***Weaknesses of* SALT.** Because SALT relies on static analysis to guide LLM annotation, incorrect static evidence can limit its effectiveness. Inspecting the static-analysis errors in Table VII, we find that 121 of 169 are over-narrowing errors: 71 infer a return type narrower than the developer-written annotation, and 50 infer a parameter type narrower than intended from observed call arguments.

Figure 7a illustrates over-narrowing in return types. The function body strongly suggests `list[PackageNode]`: the implementation constructs a list, appends `PackageNode` objects, and returns it directly. However, the developer-written annotation is `Sequence[PackageNode]`, intentionally exposing a more general interface. This suggests that static evidence can be too concrete when the desired annotation intentionally abstracts over the implementation.

Figure 7b shows over-narrowing for parameters. Within the observed project code, `border_style` is passed only as a string, so the static analysis infers `str`. The ground truth, however, is `StyleType` because `Table` is part of a public API that accepts both `Style` objects and strings. This limitation is

```python
def reachable(self):
  children = []
  for d in self.all_requires:
    children.append(PackageNode(d))
  return children
```

(a) Return type over-narrowing. The static analysis infers `list[PackageNode]`, while the developer-written annotation is `Sequence[PackageNode]`.

```python
StyleType = Style | str

class Table:
  def __init__(self, border_style=""):
    ...

if __name__ == "__main__":
  table = Table(border_style="not dim")
```

(b) Parameter type over-narrowing. The static analysis infers `str`, while the developer-written annotation is `StyleType`.

Fig. 7. Examples where static analysis over-narrows developer-written types.

inherent to evidence-driven static inference: when all observed uses are narrow, the analysis can miss broader types that reflect external API intent rather than internal usage.

V. DISCUSSION & THREATS TO VALIDITY

A. Type-Checker Results

Our evaluation treats predictions that differ from developer-written types as incorrect, even though alternative types can also be valid. To complement this evaluation, we examine the alarms reported by PYREFLY for annotations generated by SALT and AIDER.

Type-Checker Alarms. We count type-related alarms after excluding those already present in the original code. Compared with AIDER, SALT reduces the number of alarms from 2,049 to 1,865 for Gemma 4. The count also decreases from 925 to 911 for gpt-oss, but increases slightly from 997 to 1,012 for GLM-4.7. To examine these alarm counts in more detail, we also compare the use of standalone `Any`, which can suppress type-checker alarms. Across the three models, AIDER produces 82.3% more standalone `Any` annotations than SALT. Of these annotations, 66.7% do not match the developer-written types for AIDER, compared with 49.0% for SALT. In summary, SALT produces alarm counts comparable to those of AIDER, at substantially lower LLM cost and with fewer standalone `Any` annotations.

Analysis of Mismatches. We also examine whether predictions that differ from developer-written types affect PYREFLY alarms. For each mismatch, we replace the predicted type with `Any` to relax its type constraint and compare alarms before and after replacement. A decrease in alarms indicates that the predicted type triggers type-checker errors. Alarms remain unchanged for 70.7% of AIDER mismatches and 71.4% of SALT mismatches. These results suggest that SALT can offer plausible alternative type annotations at substantially lower LLM cost.

B. Threats to Validity

Construct Validity. Our accuracy metric uses exact match, so it can penalize valid alternative types. Although we complement this metric with type-checker analysis to assess annotation quality (Section V-A), further evaluation is needed to assess developers' preferences for the inferred types.

External Validity. Our evaluation removes type annotations from 12 TYPYBENCH projects to provide a controlled setting for comparison. However, this setting may not fully reflect naturally under-annotated repositories, and further evaluation on such repositories would help assess the broader applicability of our findings. Since reference annotations are unavailable for missing types, such evaluation would rely on type-checker analysis or developer feedback on the inferred types.

Internal Validity. Static filling may insert incorrect types or propagate incorrect LLM predictions. Approximately 2% of slots receiving static type information are correctly annotated by AIDER but not by SALT. A post-processing step that uses type-checker feedback to correct such errors would help improve annotation quality. In addition, LLM non-determinism may affect the reproducibility of results. We release all LLM interaction logs to support reproducibility.

VI. RELATED WORK

Rule-based Type Inference. Rule-based approaches infer types by applying hand-designed rules over program syntax and semantics [17]–[19], [34]–[37]. This line of work has had substantial practical impact, especially in Python static analysis and type checking tools [16], [20]–[22]. However, Python programs often rely on dynamic features, implicit data flows, and project-specific conventions that are difficult to capture completely with fixed rules.

Learning-based Type Inference. Early learning-based systems predict types using supervised models, including graph neural networks and deep similarity models [4]–[10]. However, their predictions remain constrained by training distributions and type vocabularies, limiting their ability to infer complex annotations. Recent work explores context construction for type inference with pre-trained code models and LLMs [11]–[13], [15], [38]. TYPET5 [12] uses caller-callee context and incorporates previous type predictions through iterative decoding. TYPEGEN [15] combines dependency-based code slices, available types, and chain-of-thought examples in its prompts. TYPEPRO [38] uses static analysis to extract interprocedural code slices for its prompts. A common challenge is to construct compact context while preserving relevant type information. SALT supplements its code context with slot-specific hints that summarize type evidence from forward and backward propagation, reducing the need for supporting caller and callee code.

Hybrid Approach to Type Inference. Hybrid techniques combine learned models with static analysis to improve prediction quality [6]–[8], [14], [39]. DLINFER [6] and NESTER [14] extract dataflow graphs from the local context to provide additional information to the model. HITYPER [8] applies forward type inference and type checking to validate model predictions. To improve type correctness, HITYPER selects type slots whose predictions enable forward inference for other unresolved slots. This selection does not directly optimize query cost in SALT's setting, where annotating a function can help resolve callers and callees through forward and backward propagation. To reduce queries, SALT prioritizes functions with more distinct unresolved caller and callee neighbors. It updates these priorities after each propagation step to reflect newly resolved functions. PYTIR [39] iteratively refines types and dependencies across a repository, using type-checker feedback to re-query the LLM for conflicting annotations. SALT reduces the query workload through direct static annotation and propagation. The approaches are complementary and could be combined to reduce query costs while improving annotation consistency through type-checker feedback.

VII. CONCLUSION

This paper presented SALT, a type annotation system that uses static analysis to guide LLM inference for Python programs. Rather than invoking an LLM uniformly for every target function, SALT uses project-wide type evidence to resolve annotations statically, prioritize high-value LLM queries, and construct concise type-aware context. Our evaluation on real-world Python projects shows that this combination improves both accuracy and efficiency over specialized type inference systems and strong coding-assistant baselines. The results suggest that static analysis is a useful complement to LLM-based type inference: it preserves precise value-flow evidence, reduces unnecessary model calls, and provides concise context that improves the quality of LLM predictions. This combination provides a practical way to improve type annotation for large Python codebases.

DATA AVAILABILITY

For open science, we have made our implementation and evaluation data publicly available on GitHub[4] and Zenodo[5].

ACKNOWLEDGMENT

This work was supported by the German Research Foundation (DFG; projects 492507603, 516334526, and 526259073). This work was partly supported by Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korean government (MSIT) (No. 2020-0-01337, (SW STAR LAB) Research on Highly-Practical Automated Software Repair) and by ICT Creative Consilience Program through the Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korean government (MSIT) (IITP-2026-RS-2020-II201819, 5%). This work was also supported by the National Research Foundation of Korea (NRF) grant funded by the Korean government (MSIT) (No. 2021R1A5A1021944 (10%), RS-2026-25470524 (50%)).

[4]https://github.com/kupl/salt-artifact

[5]https://doi.org/10.5281/zenodo.23788803

References

[1] W. Xie, S. Zhang, and Z. Wang, "Do type annotations help students improve Python program development efficiency?" in *2025 25th International Conference on Software Quality, Reliability, and Security Companion (QRS-C)*. IEEE, 2025, pp. 594–600.

[2] L. Di Grazia and M. Pradel, "The evolution of type annotations in Python: an empirical study," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 209–220.

[3] J.-P. Ore, S. Elbaum, C. Detweiler, and L. Karkazis, "Assessing the type annotation burden," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 190–201.

[4] J. Wei, M. Goyal, G. Durrett, and I. Dillig, "LambdaNet: Probabilistic type inference using graph neural networks," in *International Conference on Learning Representations*, 2020.

[5] V. J. Hellendoorn, C. Bird, E. T. Barr, and M. Allamanis, "Deep learning type inference," in *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 152–162.

[6] Y. Yan, Y. Feng, H. Fan, and B. Xu, "DLInfer: Deep learning with static slicing for Python type inference," in *2023 IEEE/ACM 45th International Conference on Software Engineering*, 2023, pp. 2009–2021.

[7] M. Pradel, G. Gousios, J. Liu, and S. Chandra, "TypeWriter: Neural type prediction with search-based validation," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 209–220.

[8] Y. Peng, C. Gao, Z. Li, B. Gao, D. Lo, Q. Zhang, and M. Lyu, "Static inference meets deep learning: a hybrid type inference approach for Python," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 2019–2030.

[9] A. M. Mir, E. Latoškinas, S. Proksch, and G. Gousios, "Type4Py: Practical deep similarity learning-based type inference for Python," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 2241–2252.

[10] M. Allamanis, E. T. Barr, S. Ducousso, and Z. Gao, "Typilus: Neural type hints," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 91–105.

[11] K. Jesse, P. T. Devanbu, and T. Ahmed, "Learning type annotation: is big data enough?" in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 1483–1486.

[12] J. Wei, G. Durrett, and I. Dillig, "TypeT5: Seq2seq type inference using static analysis," in *The Eleventh International Conference on Learning Representations*, 2023.

[13] C. Wang, J. Zhang, Y. Lou, M. Liu, W. Sun, Y. Liu, and X. Peng, "Tiger: A generating-then-ranking framework for practical Python type inference," in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE, 2025, pp. 321–333.

[14] G. Li, Y. Wan, H. Zhang, Z. Zhao, W. Jiang, X. Shi, H. Jin, and Z. Wang, "Dataflow-guided neuro-symbolic language models for type inference," in *International Conference on Machine Learning*. PMLR, 2025, pp. 36 000–36 020.

[15] Y. Peng, C. Wang, W. Wang, C. Gao, and M. R. Lyu, "Generative type inference for Python," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 988–999.

[16] Meta, "Pyrefly: A fast type checker and language server for Python," https://github.com/facebook/pyrefly, accessed: 2026-06-28.

[17] M. Hassan, C. Urban, M. Eilers, and P. Müller, "MaxSMT-based type inference for Python 3," in *International Conference on Computer Aided Verification*. Springer, 2018, pp. 12–19.

[18] M. Salib, "Starkiller: A static type inferencer and compiler for Python," Ph.D. dissertation, Massachusetts Institute of Technology, 2004.

[19] W. Oh and H. Oh, "Towards effective static type-error detection for Python," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 1808–1820.

[20] Microsoft, "Pyright: Static type checker for Python," https://github.com/microsoft/pyright, accessed: 2026-06-28.

[21] Python, "mypy: Optional static typing for Python," https://www.mypy-lang.org/, accessed: 2026-06-28.

[22] Google, "pytype: A static type analyzer for Python code," https://github.com/google/pytype, accessed: 2026-06-28.

[23] Aider-AI, "Aider: AI pair programming in your terminal," https://github.com/Aider-AI/aider, accessed: 2026-06-26.

[24] Anthropics, "Claude Code," https://github.com/anthropics/claude-code, accessed: 2026-06-26.

[25] OpenAI, "Codex CLI," https://github.com/openai/codex, accessed: 2026-06-26.

[26] GitHub, "GitHub Copilot," https://github.com/features/copilot, accessed: 2026-06-26.

[27] Anysphere, "Cursor," https://cursor.com/, accessed: 2026-06-26.

[28] Z.ai, "GLM-4.7," https://z.ai/blog/glm-4.7, accessed: 2026-06-24.

[29] H. Dong, J. Yang, X. Deng, Y. Jiang, G. Pekhimenko, F. Long, and X. Si, "TypyBench: Evaluating LLM type inference for untyped Python repositories," in *Proceedings of the 42nd International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, A. Singh, M. Fazel, D. Hsu, S. Lacoste-Julien, F. Berkenkamp, T. Maharaj, K. Wagstaff, and J. Zhu, Eds., vol. 267. PMLR, 13–19 Jul 2025, pp. 14 203–14 230. [Online]. Available: https://proceedings.mlr.press/v267/dong25l.html

[30] Google DeepMind, "Gemma 4," https://deepmind.google/models/gemma/gemma-4/, accessed: 2026-06-24.

[31] OpenAI, "gpt-oss," https://openai.com/index/introducing-gpt-oss/, accessed: 2026-06-24.

[32] Dave Halter, "Jedi: An awesome autocompletion, static analysis and refactoring library for Python," https://github.com/davidhalter/jedi, accessed: 2026-06-25.

[33] F. Wilcoxon, "Individual comparisons by ranking methods," *Biometrics Bulletin*, vol. 1, no. 6, pp. 80–83, 1945. [Online]. Available: http://www.jstor.org/stable/3001968

[34] M. Gorbovitski, Y. A. Liu, S. D. Stoller, T. Rothamel, and T. K. Tekle, "Alias analysis for optimization of dynamic languages," in *Proceedings of the 6th Symposium on Dynamic Languages*, 2010, pp. 27–42.

[35] A. Rigo and S. Pedroni, "PyPy's approach to virtual machine construction," in *Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications*, 2006, pp. 944–953.

[36] J. Dolby, A. Shinnar, A. Allain, and J. Reinen, "Ariadne: analysis for machine learning programs," in *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, 2018, pp. 1–10.

[37] S. Lagouvardos, J. Dolby, N. Grech, A. Antoniadis, and Y. Smaragdakis, "Static analysis of shape in TensorFlow programs," in *34th European Conference on Object-Oriented Programming*, 2020, pp. 15:1–15:29.

[38] T. Lin, M. Fan, H. Huang, Z. Shen, and R. Wu, "TypePro: Boosting LLM-based type inference via inter-procedural slicing," *Proc. ACM Softw. Eng.*, vol. 3, no. FSE, Jun. 2026. [Online]. Available: https://doi.org/10.1145/3808124

[39] S. Sun, S. Zhang, J. Yan, J. Yan, and J. Zhang, "Co-evolution of types and dependencies: Towards repository-level type inference for Python code," *Proc. ACM Softw. Eng.*, vol. 3, no. FSE, Jun. 2026. [Online]. Available: https://doi.org/10.1145/3797103