A Fast, Reliable, and Secure Programming Language for LLM Agents with Code Actions

Stephen Mell
University of Pennsylvania
sm1@cis.upenn.edu

Botong Zhang
University of Pennsylvania
bzhang16@seas.upenn.edu

David Mell
Unaffiliated
zraexy@gmail.com

Shuo Li
University of Pennsylvania
lishuo1@seas.upenn.edu

Ramya Ramalingam
University of Pennsylvania
ramya23@seas.upenn.edu

Nathan Yu
Unaffiliated
nathany@alumni.cmu.edu

Steve Zdancewic
University of Pennsylvania
stevez@cis.upenn.edu

Osbert Bastani
University of Pennsylvania
obastani@seas.upenn.edu

Abstract

Modern large language models (LLMs) are often deployed as *agents*, calling external tools adaptively to solve tasks. Rather than directly calling tools, it can be more effective for LLMs to write code to perform the tool calls, enabling them to automatically generate complex control flow such as conditionals and loops. Such *code actions* are typically provided as Python code, since LLMs are quite proficient at it; however, Python may not be the ideal language due to limited built-in support for performance, security, and reliability. We propose a novel programming language for code actions, called QUASAR, which has several benefits: (1) automated parallelization to improve performance, (2) uncertainty quantification to improve reliability and mitigate hallucinations, and (3) security features enabling the user to validate actions. LLMs can write code in a subset of Python, which is automatically transpiled to QUASAR. We evaluate our approach on the ViperGPT visual question answering agent, applied to the GQA dataset, demonstrating that LLMs with QUASAR actions instead of Python actions retain strong performance, while reducing execution time when possible by 42%, improving security by reducing user approval interactions when possible by 52%, and improving reliability by applying conformal prediction to achieve a desired target coverage level.[1]

1 Introduction

Large language models (LLMs) have recently demonstrated remarkable general reasoning capabilities. To leverage these capabilities to solve practical tasks, there has been significant interest in *LLM agents*, where the LLM is given access to *tools* that can be used to interact with an external system. The LLM can autonomously choose when to use different tools to help complete a task given by the user. Tools include functions to read and edit files [28], access to knowledge sources such as databases [10], external memory to store information across different interactions [14, 15], and access to user input/output devices such as mouse, keyboard, and screen [3].

An effective strategy in practice is to provide these tools in the form of software APIs, and then let the LLM write code that includes these APIs [27, 23, 24]; we refer to these systems as LLM

[1] Our implementation is available at `https://github.com/stephenmell/quasar`.

Preprint. Under review.

agents with *code actions*. This strategy enables the LLM to write code that includes control flow to facilitate more complex interactions, such as automating iterative tasks by writing loops. For example, ViperGPT gives the LLM access to external tools such as object detectors to perform image question answering [23], and AppWorld gives the LLM access to a rich variety of smartphone app APIs to enable it to help the user automatically configure their device (in a simulation) [24].

A natural question is what the ideal programming language is for code actions. Python has become the standard choice due to the presence of a large existing ecosystem of software libraries; furthermore, due to the large amount of Python code in most LLM pretraining corpora, LLMs have been shown to be proficient at writing Python code [32, 17, 22, 21].

However, there are also a number of drawbacks of using Python. It is a highly dynamic language, making it difficult to provide assurances that the generated code is safe to execute. It is also challenging to optimize, when many agent workflows exhibit significant potential for parallelism; for instance, programs generated by ViperGPT often call multiple APIs that can in principle be executed in parallel. In addition, agents may call other models, which are themselves prone to hallucination. While conformal prediction [25] can mitigate this for an individual model call by returning a set, the rest of the agent's program must then be executed with a set of values rather than a single concrete value. Python cannot do this kind of set-based execution. As a consequence, there is a unique opportunity to rethink the programming language that forms the basis of code actions.

We propose a novel agent language, QUASAR (for QUick And Secure And Reliable) that combines several promising recent ideas from the programming languages literature. The key idea is to separate *internal computation* from *external actions*. Specifically, QUASAR has a pure, functional "core language" based on lambda calculus, with side effects isolated in "external calls". Internal computations are things like executing the "then" branch of an "if" statement when the condition is true. External actions are things like executing shell programs or making requests to remote APIs. This separation provides several benefits: (1) it enables QUASAR to make use of recently proposed techniques for automatically executing external calls in parallel when possible [16], (2) it can enforce whitelists on external calls to ensure that undesirable APIs are not executed without user permission, and it can efficiently ask the user for approval in batches, and (3) it can incorporate recent techniques for uncertainty quantification in neurosymbolic programs [18].

A key challenge is that unlike Python, LLMs have never seen QUASAR code and therefore do not know how to write code in this language. Rather than directly teach them QUASAR, we propose an alternative strategy where we first implement a transpiler from a subset of Python to QUASAR, and then have the LLM generate Python code in this subset. Then, whenever the LLM writes code to be executed, we translate it to QUASAR and execute it using the QUASAR interpreter instead.

Contributions. (1) We introduce QUASAR, a novel programming language for LLM agent actions. (2) We propose a generation strategy for QUASAR code by first asking the LLM to generate code in a subset of Python, and then transpiling that to QUASAR. (3) We experimentally demonstrate that our generation strategy achieves task performance comparable to standard Python generation, while producing $6.9\times$ and $7.6\times$ fewer erroneous programs than two baselines. (4) We experimentally demonstrate the utility of QUASAR, reducing execution time when possible by 42%, improving security by reducing user approvals when possible by 52%, and improving reliability by applying conformal prediction to achieve a target error rate.

2 Related Work

With the promising capabilities of large language models (LLMs), numerous studies have explored their use as autonomous agents [26, 6, 28]. Early efforts, such as Chain-of-Thought prompting [8], demonstrated that providing in-context reasoning examples can significantly enhance LLM reasoning abilities. Recognizing the tendency of LLMs to produce hallucinations, subsequent work like Retrieval-Augmented Generation (RAG) [10] and Dense Passage Retrieval (DPR) [9] introduced mechanisms to incorporate external knowledge bases, using retrieved information to improve model accuracy and reliability.

Building on this idea, ReAct [30] extends the role of external resources by providing LLMs with access to executable APIs and external tools, enabling them to perform simple tasks through API calls. While these approaches primarily guide agentic behavior via natural language, recent works

Question: Is there an alcoholic drink in this image?

Figure 1: Illustrative example of an image and a natural language question about that image. We show predictions of both the original object detector (left) and the conformal detector (right). For the latter, the green boxes are identifed as being definitely in the image whereas the yellow boxes may or may not be in the image. The program P_1 in Figure 2 answers this question for its input image.

such as CodeAct [27], ViperGPT [23], and AppWorld [24] take a step further by instructing LLMs to generate executable Python code as agent actions. This transition from natural language to code-based actions has demonstrated improved task performance and greater flexibility.

However, despite these advancements, several challenges remain for LLM agents. These include security and privacy risks [5, 1], persistent hallucination issues [13, 11], and concerns over computational efficiency [29]. Recent work has proposed addressing the security vulnerabilities by analyzing dataflows in agent-generated code, focusing on a restricted subset of Python [4]. However, they do not offer performance or reliability improvements, and their approach does not support asking for batch user approval. Other work has addressed conformal prediction of functional programs with neural components [18] and automatic parallelization of functional programs [16]. Though we draw on insights from this work, neither considers programs generated by LLM agents or the imperative features of languages like Python.

3 QUASAR Programming Language

We first describe the syntax and semantics for QUASAR programs; then, we provide details on how QUASAR improves security, performance, and reliability (summarized in Algorithm 1). We show a running example in Figure 2 for the problem in Figure 1.

3.1 Syntax and Semantics of QUASAR

A QUASAR program $P \in \mathcal{P}$ consists of standard syntatic constructs such as conditionals, loops, and function calls. The execution of a QUASAR program $P \in \mathcal{P}$ is expressed as a set of *rewrite rules* $\mathcal{R}$. If a rule $R \in \mathcal{R}$ is applicable to P, then it transforms P into a new program P', which we denote by $P \xrightarrow{R} P'$. In general, there may be multiple possible programs P' satisfying $P \xrightarrow{R} P'$, for instance, if the rule R is applicable to different parts of P. If there is any rewrite R mapping P to P', then we simply write $P \to P'$. We give the full set of rewrite rules in Figure 6 in Appendix A.

There are two kinds of rewrite rules: internal rules $\mathcal{R}_{\text{int}}$ and external rules $\mathcal{R}_{\text{ext}}$. Internal rules do not have *effects*, meaning they do not have consequences external to the program, including network calls, system calls, calls to external APIs, or even printing. Internal rules perform transformations such as substituting variables, unrolling loops, and resolving conditionals; these rules are applicable if the necessary values are constants (e.g., a conditional where the predicate is `True` or `False`).

For example, in program P_2 in Figure 2, the list in the for loop is a constant value `[patch1, patch2]`, so QUASAR applies a rule to unrolls the for loop, resulting in P_3. Similarly, in P_4, it can apply rewrite rules to rewrite the predicate `"no" == "yes"` to `False` and the predicate `"yes" == "yes"` to `True`, after which it can rewrite the conditionals to obtain P_5.

There is only one external rule $\mathcal{R}_{\text{ext}} = \{R_{\text{ext}}\}$. This rule is designed to enable calls to *external functions* $f \in \mathcal{F}_{\text{ext}}$. Unlike a typical function, which is implemented as QUASAR code, an external

$P_1 =$
```
drink_patches = image_patch.find("drink")
found = False
for drink_patch in drink_patches:
    if drink_patch.simple_query("Does this have alcohol?"):
        found = True
return found
```

$E_1 = \{(\texttt{image\_patch.find("drink")}, \varnothing)\} \rightsquigarrow E_2 = \{(\texttt{image\_patch.find("drink")}, \texttt{[patch1, patch2]})$

$P_2 =$
```
found = False
for drink_patch in [patch1, patch2]:
    if drink_patch.simple_query("Does this have alcohol?"):
        found = True
return found
```

$P_3 =$
```
found = False
if patch1.simple_query("Does this have alcohol?") == "yes":
    found = True
if patch2.simple_query("Does this have alcohol?") == "yes":
    found = True
return found
```

$E_3 = \{(\texttt{patch1.simple\_query(...)}, \varnothing), (\texttt{patch2.simple\_query(...)}, \varnothing)\} \quad \rightsquigarrow$
$E_4 = \{(\texttt{patch1.simple\_query(...)}, \texttt{"no"}), (\texttt{patch2.simple\_query(...)}, \texttt{"yes"})\}$

$P_4 =$
```
found = False
if "no" == "yes"
    found = True
if "yes" == "yes"
    found = True
return found
```

$P_5 =$ `return True`

Figure 2: Given program P_1 for the question in Figure 1, QUASAR may execute it as follows. First, is immediately dispatches `image_patch.find("drink")`, resulting in execution set E_1. This external call finishes running and returns `[patch1, patch2]`, resulting in execution set E_2, after which QUASAR applies R_{ext} to substitute this value into P_1 to obtain P_2. Then, QUASAR applies an internal rule to unroll the for loop in P_2 to obtain P_3. It immediately dispatches both `patch1.simple_query(...)` and `patch2.simple_query(...)` resulting in execution set E_3. As before, these external calls finish running and return `"no"` and `"yes"`, respectively, yielding E_4, so QUASAR applies R_{ext} twice (once for each external call) to substitute these values into P_3 to obtain P_4. Finally, QUASAR applies additional internal rools to simplify the conditionals in P_4, resulting in terminal program P_5.

function is implemented in Python; thus, external functions can perform desirable effects such as printing a value or calling an LLM to obtain its output. An *external call* in program P is a statement $S = y \leftarrow f(x_1, ..., x_k)$ that calls an external function $f \in \mathcal{F}_{\text{ext}}$.

QUASAR executes external calls as soon as all of their arguments are available. In more detail, an external call $S = y \leftarrow f(x_1, ..., x_k)$ in a program P is *dispatchable* if all of its $x_1, ..., x_k$ are values (such as `0`, `True`, or `"foo"`; recall that variables become values as the program is incrementally rewritten). As QUASAR performs rewrites, it keeps track of the currently executing external calls $(S, B) \in E$, where S is a pointer to the external call in the current program P (preserved by rewrites) and B is a pointer to a value that is initially $\varnothing$ but is eventually set to the output of the external function. After a rewrite $P \rightarrow P'$, QUASAR identifies all the dispatchable external calls S in P' that are not yet in E; for each $S = y \leftarrow f(x_1, ..., x_k)$, it executes f on $x_1, ..., x_k$ in a separate thread T, and adds the pending calls (S, B) to E. The thread T is also given B; once it finishes executing the

Algorithm 1 Pseudocode for the QUASAR interpreter. At each iteration, it validates the current set of external calls with the user, and then executes them. It then rewrites P as much as possible (including waiting for pending external calls to finish running), until it is stuck. Then, it repeats the process until P cannot be rewritten any further, at which point it returns the result.

function RUNQUASAR(P)
  while P has dispatchable external calls **or** P is not terminal **do**
    Identify dispatchable external calls $\{S\}$ in P
    Query user to validate $\{S\}$, and terminate execution if rejected
    Dispatch all external calls in P and add to a set E
    $P \leftarrow$ RUNINTERNAL(P, E)
  return P
function RUNINTERNAL(P, E)
  while $E \neq \varnothing$ **or** P is not terminal **do**
    if there exists $(y \leftarrow f(x_1, ..., x_k), B) \in E$ such that $B \neq \varnothing$ **then**
      apply R_{ext} to P to substitute B in for y
    else if there exists a rule $R \in \mathcal{R}_{\text{int}}$ that is applicable to P **then**
      apply R to P
  return P

external function f, it writes the output of f to B and terminates. Then, QUASAR applies the rewrite rule R_{ext} to the current program (which may no longer be P') to substitute the value in B into the program.

For example, in Figure 2, given the initial program P_1, QUASAR immediately dispatches the external call `image_patch.find("drink")` in a separate thread, leading to execution set E_1. When this thread finishes, it will write the result `[patch1, patch2]` to $\varnothing$, resulting in E_2. This allows R_{ext} to be applied to P_1, obtaining P_2. Similarly, as soon as QUASAR rewrites P_2 to P_3, it dispatches two external calls `patch1.simple_query(...)` and `patch2.simple_query(...)`, resulting in E_3; these execute and return `"no"` and `"yes"`, respectively, resulting in E_4. Finally, QUASAR applies R_{ext} twice to substitute these values into P_3, resulting in P_4. The general approach is given in Algorithm 1.

A program P is *terminal* if no rules are applicable to P, and there are no pending external calls. Assuming each external call only depends on its inputs, then it can be shown that any sequence of rule applications results in the same set of external calls, and therefore the same effects. The order in which the effects happen may be different depending on the sequence of rules applied; dependencies can be enforced by inserting arguments and return values into the relevant external calls, similar to how a pseudorandom number generator can be added to code for deterministic execution.

Because of this property (and assuming external calls depend only on their inputs), the QUASAR interpreter can apply rules to P in any order. The specific strategy it employs is to first minimize the amount of interaction with the user required to validate the external calls it makes, while maximizing performance. These details are discussed in Sections 3.2 & 3.3, respectively.

There are two key benefits of this design of QUASAR. First, it decouples side effects (external) from the pure computation (internal). For instance, any internal rewrite rules cannot pose security issues by construction, since they do not have any effects on the world (other than consuming computational resources to run); thus, we only need to worry about external calls when considering potential security issues. Further, this separation makes it much easier to implement conformal semantics for QUASAR than it would be for Python. Second, because the rules can be applied in any order, execution can continue while waiting for time-consuming external calls to finish running. This is useful both for parallelizability and for reducing the number of user interaction required to validate external calls. We describe these benefits in more detail below.

3.2 Security via Dynamic Access Control

We consider a standard security model based on access control [19, 20], where the user must approve the execution of effects. Because effects are isolated in external calls, we only need to ensure that external calls are consistent with the user's desired security policy. For instance, a smartphone user

might give an app access to a subset of resources such as the user's location and the ability to send emails, in which case the app would only be allowed to access these resources.

In QUASAR, access to certain external functions can be granted ahead of time; alternatively, the user can dynamically approve each external call made by the program. A key challenge with dynamic access control is minimizing the number of rounds of interaction with the user; frequent interruptions can lead to poor usability. Thus, QUASAR is designed to "collect" as many external calls as possible and then query the user to confirm all of them. If rejected, execution terminates; otherwise, the external calls are all dispatched in parallel and execution proceeds. This algorithm is summarized in the RUNINTERNAL subroutine in Algorithm 1, which performs as many rewrites of the current program P as possible (including both applying internal rules as well as handling previously-dispatched external calls). It returns once P cannot be rewritten any further, in which case the main routine RUNQUASAR queries the user to validate all the external calls in P, and then dispatches all of these calls in parallel. This loop continues until P is terminal. For example, in Figure 2, QUASAR asks the user for permission to make the external call `image_patch.find("drink")` in P_1, but then is able to batch the permission requests for `patch1.simple_query(...)` and `path2.simple_query(...)` in P_3.

3.3 Performance via Parallel Evaluation

The strategy QUASAR uses to minimize the number of rounds of interaction for security automatically parallelizes external calls, since all external calls in P are dispatched simultaneously in the RUN-QUASAR routine. The actual ability to expose parallelism comes from the design of the QUASAR language and its internal rewrite rules. Intuitively, because QUASAR programs are interpreted using rewrite rules, a statement can be "executed" as soon as the relevant program variables are substituted with constants. This property enables QUASAR to execute statements out-of-order. For example, in program P_3 in Figure 2, the statement `patch2.simple_query(...)` can be evaluated even though previous statements have not yet been evaluated, since all of the arguments in this external call (the image patch `patch2` and the string `"Does this have alcohol?"`) are constants. As a consequence, this external call can be dispatched in parallel with `patch1.simple_query(...)`, which significantly improves performance compared to ordinary sequential execution in Python.

3.4 Reliability via Conformal Semantics

We also implement *conformal semantics* in QUASAR for uncertainty quantification. Conformal prediction is a popular technique for quantifying the uncertainty of individual blackbox machine learning models by modifying a given model to output a set of labels instead of a single label. For example, an image classification model might output a set of plausible class labels instead of just the most likely one. When QUASAR makes external calls to other machine learning models, we may want to quantify the uncertainty of these models, and then keep track of how this uncertainty propagates through the program. Specifically, program variables are assigned to sets of values instead of individual values.

The key challenge is modifying the program execution to handle sets of values. For example, if a Boolean variable x is bound to the set of values $x \mapsto \{\texttt{True}, \texttt{False}\}$, and a conditional statement `if` x `then` p_{true} `else` p_{false} that branches on x, then we effectively execute both branches p_{true} and p_{false} of the conditional; then, for each variable y defined in these branches, we take the union of the values v_{true} bound to y in p_{true} and v_{false} in p_{false}, i.e., $y \mapsto v_{\text{true}} \cup v_{\text{false}}$. QUASAR includes a modified set of *conformal* rewrite rules that handle variables bound to sets of values in this way.

Because external functions are opaque to QUASAR, abstract versions of them must be provided. In the case of calls to neural models, such as `find`, the abstract version is provided by applying some conformal technique, such as returning the set of labels whose probability is above some threshold. For example, the object detector shown in the left of Figure 1 misses two objects (though in this case, it does not affect the final answer in Figure 2); the output of the conformal detector is shown on the right. In this case, the external call `image_patch.find("drink")` indicates whether each detection is definitely (green) or possibly (yellow) in the image; it represents the set of lists of patches

$$\{[\texttt{patch1}, \texttt{patch2}, \texttt{patch3}, \texttt{patch4}], [\texttt{patch2}, \texttt{patch3}, \texttt{patch4}], \\ [\texttt{patch1}, \texttt{patch2}, \texttt{patch4}], [\texttt{patch2}, \texttt{patch4}]\},$$

```
drink_patches = image_patch.find("drink")
found = False
for drink_patch in drink_patches:
    if drink_patch.simple_query("Does this have alcohol?"):
        found = True
return found
```

(a)

```
({77: '.find', 83: '.simple_query'},
 ('def',
  75,
  ((76,),
   ((('prim', 78, 'drink'),
     ('call', (79,), 77, (76, 78)),
     ('prim', 80, False),
     ('def',
      81,
      ((89, 82),
       ((('prim', 84, 'Does this have alcohol?'),
         ('call', (85,), 83, (82, 84)),
         ('def', 86, (), ((('prim', 87, True),), (87,)))),
         ('def', 88, (), (), (89,)))),
         ('call', (91,), 0, (85,)),
         ('call', (92,), 91, (86, 88)),
         ('call', (90,), 92, ())),
        (90,)))),
     ('call', (93,), 79, (80, 81))),
    (93,)))))
```

(b)

Figure 3: An example of the same agent code, in both Python (a) and raw QUASAR (b) forms.

where the patches are ordered from left to right. Similarly, for each patch, the external call `patch.simple_query("Does this drink have alcohol?")` returns a prediction set that is a subset of $\{$`"yes"`, `"no"`$\}$. QUASAR overapproximates the true output; in this case, the program output is $\{$`"yes"`$\}$, i.e., there is definitely an alcoholic drink in the image.

The conformal guarantee says that, for some target fraction of the test dataset ("coverage"), the ground truth label will be contained in the predicted set of labels. While this can be trivially obtained by outputting the set of all labels, the sizes of sets should be kept as small as possible while satisfying the target coverage. To satisfy the desired coverage guarantee, we use a standard conformal prediction strategy. First, we optimize the thresholds for each individual model on a optimization set [12]. Then, using a held-out calibration set, we jointly rescale these thresholds using a single scaling parameter $\tau \in \mathbb{R}$ chosen using conformal prediction to satisfy a desired coverage guarantee [2, 31].

3.5 Generating QUASAR Code

For purposes of illustration, we have written example code with a syntax similar to Python. However, as shown in Figure 3, raw QUASAR code looks very different. A key challenge is that LLMs have never seen QUASAR code before, and in our experiments, we find that they struggle to generate it directly. Instead, our strategy is to have the LLM generate Python and then *transpile* this Python code to QUASAR. That is, the LLM generates the code in Figure 3a, we transpile it to the code in 3b, and then the QUASAR interpreter executes it. It is very challenging to transpile unrestricted Python to QUASAR, since this strategy would inherit all the challenges of making Python more performant, secure, and reliable. Furthermore, many practical agents do not use the unsupported language features of Python (e.g., classes and inheritance); intuitively, agents are trying to perform actions, not write complex software. Thus, our transpiler supports a restricted subset of Python carefully chosen to balance expressiveness and ease of transpilation. To generate code in our restricted subset of Python, we simply instruct the LLM to do so in the system prompt; more advanced prompting strategies can also be used, but we found this approach to be sufficient for our experiments. We provide details on the supported subset of Python, as well as the transpilation strategy, in Appendix B.

4 Evaluation

We evaluate two aspects of our approach. First, we show that generating QUASAR code via transpilation is more effective than baseline approaches, while retaining task performance comparable to the use of Python (Section 4.1). Second, we show that QUASAR is useful, offering improvements in several diverse regards: performance, with significant reductions in execution time (Section 4.2); security, with significant reductions in the number of user interactions required (Section 4.3); and reliability, with the conformal semantics achieving a target coverage rate (Section 4.4).

We evaluate on ViperGPT [23], a visual question answering agent approach. Given a natural language query about an image, ViperGPT first uses an LLM agent to generate a Python program that would

Approach	Successful Execution	VQA Accuracy
Python	99.7%	70.6%
Transpiled	90.6%	70.6%
Translated	35.4%	61.3%
Direct	28.3%	60.8%

Table 1: Comparison of different code generation approaches on 1000 tasks. "Successful Execution" is the fraction of generated programs that execute successfully (i.e. no syntax or runtime errors). "VQA Accuracy" is the fraction of successful programs that correctly output the ground truth label.

answer that query when provided with an image. The Python program itself has access to various neural modules, including an object detector, a vision-language model, and an LLM. We apply the ViperGPT approach on 1000 tasks randomly sampled from GQA [7], a dataset of questions about various day-to-day images.

4.1 Generation of QUASAR Code

To evaluate our strategy for generating QUASAR code, we compare our approach ("transpiled") to two baselines: "translated" (generating Python from the compileable subset, but then using an LLM to translate to QUASAR rather than the transpiler), and "direct" (prompting the LLM to generate QUASAR code directly). We also compare it to "Python" (directly generate and execute unrestricted Python code instead of using QUASAR, as originally proposed in ViperGPT).

For each approach, we consider the evaluation accuracy on the GQA dataset—i.e. for what fraction of tasks does the generated program both execute without error ("Successful Execution") and produce the correct result for the visual question answering task ("VQA Accuracy"). Errors in our approach ("transpiled") are due to the LLM failing to adhere to the allowed subset of Python. Results are shown in Table 1. The accuracy of QUASAR programs is comparable to that of Python programs, but our approach makes $6.9\times$ fewer errors than LLM translation, and $7.6\times$ fewer errors than direct LLM generation.

4.2 Performance

To evaluate the performance improvements of QUASAR, we consider pairs of QUASAR programs and the Python programs that they were transpiled from, ensuring that the programs have the same input-output behavior. We also control for the time that each external call takes to execute by recording every external call that a program makes and what its result and running time are. Then, we replay this recording on both the Python and QUASAR versions of the program and record the total execution time of each. The running times of each program pair are shown in Figure 4b. Across the entire dataset, QUASAR reduces running time by $16\% \pm 25$ (mean $\pm$ stddev). This large variance is because only 41% of tasks are parallelizable. Among those, the running time is cut almost in half, by $42\% \pm 22$.

4.3 Security

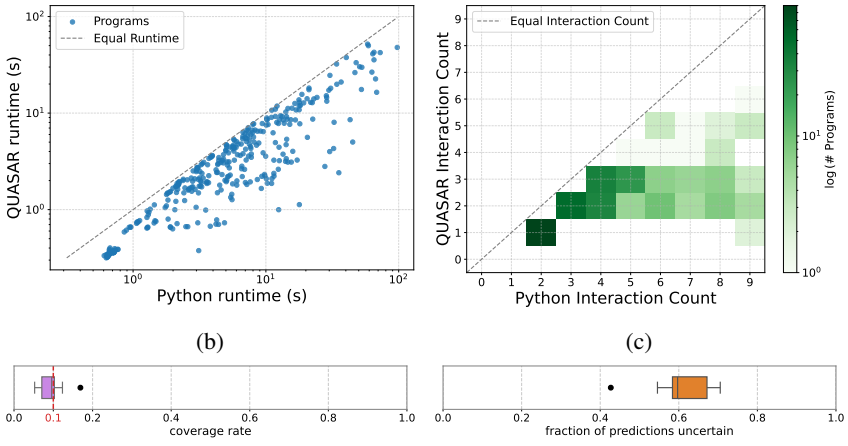
We evaluate the security improvements of QUASAR in terms of the reduction in the number of user interactions required to approve all external calls made by the program. As in Section 4.2, we consider pairs of equivalent Python and QUASAR programs, i.e. that make exactly the same external calls. We compare the number of user approvals required if the external calls are approved one at a time versus if they are approved in batches (i.e. QUASAR executes as much internally as possible before asking the user to approve). Results are shown in Figure 4c. Across the entire dataset, QUASAR reduces the number of user interactions by $22\% \pm 28$. This large variance is because only 42% of tasks offer batching of approvals. Among those, the interaction count is more than cut in half, by $52\% \pm 19$.

4.4 Reliability

We evaluate the reliability improvements of QUASAR by showing how the conformal semantics can achieve a target coverage rate of 0.1 on a test set. Using the same dataset of QUASAR programs, we

Performance	Security	Reliability
reducing running time	reducing interaction count	providing conformal guarantee
$16\% \pm 25$ across dataset	$22\% \pm 28$ across dataset	10% error target
41% of tasks improvable	42% of tasks improvable	$9.1\% \pm 1.9$ empirical error
$42\% \pm 22$ on improvable	$52\% \pm 19$ on improvable	$61.4\% \pm 4.9$ predictions uncertain

(a)



(b) (c)

(d) (e)

Figure 4: An overview of the improvements (mean $\pm$ stddev) provided by QUASAR (a). Python vs QUASAR running time for the improvable tasks (b). Python vs QUASAR user interactions required for the improvable tasks (c). Using the conformal semantics and targeting 0.1 coverage, the distribution of coverage (d) and the distribution of fraction of "uncertain" predictions (e) for 100 different validation/test splits.

evaluated using the conformal semantics with 4 different threshold values, which produce progressively larger output sets for each program. We divided the dataset 100 times into validation/test splits. For each split, we chose the largest threshold (and thus smallest prediction sets) where the validation error was less than 0.1, and then we computed the test error with that threshold. The distribution of these test errors (coverage) is shown in Figure 4d, with mean coverage $9.1\% \pm 1.9$. Because the domain of labels varies based on task (e.g. yes/no, color, object, etc), instead of measuring the size of prediction sets we measure certainty—i.e. the model is certain if the prediction set is size 1, and otherwise it is uncertain. We consider the fraction of tasks on which the model is uncertain. The distribution of such uncertainty rates in shown in Figure 4e, with mean $61.4\% \pm 4.9$.

5 Conclusion and Limitations

There are, of course, limitations of QUASAR, which we leave for future work. Querying the user for approval of external calls requires that such calls be understandable to the user, which they may not always be. The transpilation pipeline is currently specific to Python, but could be generalized to other languages. Currently QUASAR transpiles a single unit of code—ViperGPT only has one round of code-generation—though agents for other domains may generate and execute code in a loop, preserving variable state between executions; QUASAR could be extended to transpile multiple units and preserve variable state. LLM agents may be expecting sequential Python execution, and so they may not prioritize a parallelizable approach over a sequential one even when it is possible, hindering the speedups offered by QUASAR.

In this paper, we presented QUASAR, a language for code actions by LLM agents. Leveraging LLMs proficiency with Python, we transpile from a subset of Python into QUASAR. QUASAR offers several benefits in terms of performance, via automatic parallelization, security, by dynamically asking the user for approval of batches of external calls, and reliability, by supporting offering conformal execution semantics for programs.

References

[1] M. Andriushchenko, A. Souly, M. Dziemian, D. Duenas, M. Lin, J. Wang, D. Hendrycks, A. Zou, Z. Kolter, M. Fredrikson, E. Winsor, J. Wynne, Y. Gal, and X. Davies. Agentharm: A benchmark for measuring harmfulness of llm agents, 2025.

[2] A. N. Angelopoulos, S. Bates, E. J. Candès, M. I. Jordan, and L. Lei. Learn then test: Calibrating predictive algorithms to achieve risk control, 2022.

[3] Anthropic. Claude's extended thinking, 2025.

[4] E. Debenedetti, I. Shumailov, T. Fan, J. Hayes, N. Carlini, D. Fabian, C. Kern, C. Shi, A. Terzis, and F. Tramèr. Defeating prompt injections by design, 2025.

[5] F. He, T. Zhu, D. Ye, B. Liu, W. Zhou, and P. S. Yu. The emerged security and privacy of llm agent: A survey with case studies, 2024.

[6] D. Huang, J. M. Zhang, M. Luck, Q. Bu, Y. Qing, and H. Cui. Agentcoder: Multi-agent-based code generation with iterative testing and optimisation, 2024.

[7] D. A. Hudson and C. D. Manning. Gqa: A new dataset for real-world visual reasoning and compositional question answering. *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019.

[8] e. a. Jason Wei. Chain-of-thought prompting elicits reasoning in large language models, 2023.

[9] V. Karpukhin, B. Oğuz, S. Min, P. Lewis, L. Wu, S. Edunov, D. Chen, and W. tau Yih. Dense passage retrieval for open-domain question answering, 2020.

[10] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W. tau Yih, T. Rocktäschel, S. Riedel, and D. Kiela. Retrieval-augmented generation for knowledge-intensive nlp tasks, 2021.

[11] S. Li, A. Kan, L. Callot, B. Bhasker, M. S. Rashid, and T. B. Esler. Redo: Execution-free runtime error detection for coding agents, 2024.

[12] S. Li, S. Park, I. Lee, and O. Bastani. Traq: Trustworthy retrieval augmented question answering via conformal prediction, 2024.

[13] L. Liu, Y. Pan, X. Li, and G. Chen. Uncertainty estimation and quantification for llms: A simple supervised approach, 2024.

[14] N. Liu, L. Chen, X. Tian, W. Zou, K. Chen, and M. Cui. From llm to conversational agent: A memory enhanced architecture with fine-tuning of large language models, 2024.

[15] A. Maharana, D.-H. Lee, S. Tulyakov, M. Bansal, F. Barbieri, and Y. Fang. Evaluating very long-term conversational memory of llm agents, 2024.

[16] S. Mell, K. Kallas, S. Zdancewic, and O. Bastani. Opportunistically parallel lambda calculus. or, lambda: The ultimate llm scripting language, 2025.

[17] R. Puri, D. S. Kung, G. Janssen, W. Zhang, G. Domeniconi, V. Zolotov, J. Dolby, J. Chen, M. Choudhury, L. Decker, V. Thost, L. Buratti, S. Pujar, S. Ramji, U. Finkler, S. Malaika, and F. Reiss. Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks, 2021.

[18] R. Ramalingam, S. Park, and O. Bastani. Uncertainty quantification for neurosymbolic programs via compositional conformal prediction, 2024.

[19] R. Sandhu and P. Samarati. Access control: principle and practice. *IEEE Communications Magazine*, 32(9):40–48, 1994.

[20] R. S. Sandhu. Role-based access control11portions of this chapter have been published earlier in sandhu et al. (1996), sandhu (1996), sandhu and bhamidipati (1997), sandhu et al. (1997) and sandhu and feinstein (1994). volume 46 of *Advances in Computers*, pages 237–286. Elsevier, 1998.

[21] A. Shypula, S. Li, B. Zhang, V. Padmakumar, K. Yin, and O. Bastani. Evaluating the diversity and quality of llm generated content, 2025.

[22] A. Shypula, A. Madaan, Y. Zeng, U. Alon, J. Gardner, M. Hashemi, G. Neubig, P. Ranganathan, O. Bastani, and A. Yazdanbakhsh. Learning performance-improving code edits, 2024.

[23] D. Surís, S. Menon, and C. Vondrick. Vipergpt: Visual inference via python execution for reasoning, 2023.

[24] H. Trivedi, T. Khot, M. Hartmann, R. Manku, V. Dong, E. Li, S. Gupta, A. Sabharwal, and N. Balasubramanian. Appworld: A controllable world of apps and people for benchmarking interactive coding agents, 2024.

[25] V. Vovk, A. Gammerman, and G. Shafer. *Algorithmic learning in a random world*, volume 29. Springer, 2005.

[26] L. Wang, C. Ma, X. Feng, Z. Zhang, H. Yang, J. Zhang, Z. Chen, J. Tang, X. Chen, Y. Lin, W. X. Zhao, Z. Wei, and J. Wen. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18(6), Mar. 2024.

[27] X. Wang, Y. Chen, L. Yuan, Y. Zhang, Y. Li, H. Peng, and H. Ji. Executable code actions elicit better llm agents, 2024.

[28] J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, and O. Press. Swe-agent: Agent-computer interfaces enable automated software engineering, 2024.

[29] S. Yao, D. Yu, J. Zhao, I. Shafran, T. L. Griffiths, Y. Cao, and K. Narasimhan. Tree of thoughts: Deliberate problem solving with large language models, 2023.

[30] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao. React: Synergizing reasoning and acting in language models, 2023.

[31] B. Zhang, S. Li, and O. Bastani. Conformal structured prediction, 2025.

[32] T. Y. Zhuo, M. C. Vu, J. Chim, H. Hu, W. Yu, R. Widyasari, I. N. B. Yusuf, H. Zhan, J. He, I. Paul, S. Brunner, C. Gong, T. Hoang, A. R. Zebaze, X. Hong, W.-D. Li, J. Kaddour, M. Xu, Z. Zhang, P. Yadav, N. Jain, A. Gu, Z. Cheng, J. Liu, Q. Liu, Z. Wang, B. Hui, N. Muennighoff, D. Lo, D. Fried, X. Du, H. de Vries, and L. V. Werra. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions, 2025.

$$
\begin{aligned}
P &::= stmt_1; \ldots; stmt_n; \text{return } x \\
stmt &::= x \leftarrow op \\
op &::= \text{prim } c \\
&\mid x \\
&\mid (x_1, \ldots, x_n) \\
&\mid f\ x \\
&\mid \text{proj } i\ x \\
&\mid \text{fold } w\ x\ block \\
&\mid \text{if } x\ block_1\ block_2 \\
&\mid ?S \\
block &::= \{x \Rightarrow P\} \\
value &::= c \mid (value_1, \ldots, value_n)
\end{aligned}
$$

Figure 5: The grammar defining programs in QUASAR.

A Full QUASAR Language

As described in Section 3, QUASAR executes programs by transforming them with rewrite rules until they reach a result. The syntax of programs is given in Figure 5. A program consists of a sequence of statements, where each statement defines some variable ($x, y, \ldots$) to be the result of some operation (op). Variables are assumed to be defined exactly once (i.e. they are unique and do not shadow each other). An operation can be, in order: a primitive value c (where c ranges over Python values, such as `True`, `5`, or `"foo"`); another variable x; a tuple of variables x_i, the result of calling an external function f (from the set $\mathcal{F}_{ext}$) with argument x; the result of projecting out the i-th component from a tuple x; the result of folding over a list w with initial accumulator x and fold body $block$; an if expression on condition x with then-case $block_1$ and else-case $block_2$; or the result of some pending external call S. A block is a program, but which may additionally have some parameter x (in particular, so that the body of a fold can take the previous accumulator and the current list item as arguments). A value is either a Python object or a (possibly nested) tuple of Python objects—it does not directly occur in programs, but is used in the semantics.

The interpreter state at any time is simply a program P and a set E of dispatched external calls. The semantics consist of rewrite rules $R \in \mathcal{R}$, which transform one execution state to another, written $P, E \xrightarrow{R} P', E'$. Many rules do not affect E, and so are simply written as $P \xrightarrow{R} P'$.

The rewrite rules are given in Figure 6. The rule "alias" removes a statement $y \leftarrow x$, replacing it with nothing, but renaming all occurences of y in the program to x; "proj" replaces a projection operator, if the variable x is known to be a tuple $(x_1, \ldots, x_n)$, with the i-th element; for if statements, when the condition x is the primitive `True` ("if-t"), then the statement is replaced by a copy of $block_1$ (copying ensures that variables are unique; since blocks in if statements do not require parameters, w is bound to an empty tuple); if the condition is `False` ("if-f"), then the same is done for $block_2$; "fold" applies $block$ to each element of the list w, with x being the initial accumulator and y being the final one, and z_i being the i-th intermediate accumulator; "disp" replaces an external call to a function f when the argument x has a value $value$ (i.e. it is a primitive or a tuple of primitives, which $\text{value}(T, x)$ computes) with a placeholder S, begins executing the function f, and updates the execution set E; "ext" applies when an external function has finished executing—and so the execution set E contains a result in place of $\varnothing$—and replaces the placeholder with the result. In Section 3, we simplified S in the execution set to just be the external call statement itself, whereas here it is an identifier for the spawned task.

B Transpilation

QUASAR is functional, while Python supports imperative programming. Thus in QUASAR, variables cannot be changed once they have been defined. Being functional makes supporting parallel, partial, and conformal execution possible, but agents generate Python code with imperative variable updates

$$\frac{}{T[y \leftarrow x] \to \text{rename}(T[[\varnothing]], y, x)}\text{(alias)} \qquad \frac{(x \leftarrow (x_1, \ldots, x_n)) \in T}{T[y \leftarrow \text{proj } i\ x] \to T[[y \leftarrow x_i]]}\text{(proj)}$$

$$\frac{(x \leftarrow \text{prim } \texttt{True}) \in T \qquad \{w \Rightarrow stmts; \text{return } z\} = \text{copy}(block_1)}{T[y \leftarrow \text{if } x\ block_1\ block_2] \to T[[w \leftarrow (); stmts; y \leftarrow z]]}\text{(if-t)}$$

$$\frac{(x \leftarrow \text{prim } \texttt{False}) \in T \qquad \{w \Rightarrow stmts; \text{return } z\} = \text{copy}(block_2)}{T[y \leftarrow \text{if } x\ block_1\ block_2] \to T[[w \leftarrow (); stmts; y \leftarrow z]]}\text{(if-f)}$$

$$\frac{\begin{array}{c}(w \leftarrow \text{prim } [c_1, \ldots, c_n]) \in T \\ \forall i. \{y_i \Rightarrow stmts_i; \text{return } z_i\} = \text{copy}(block) \qquad stmts'_i = (w_i \leftarrow \text{prim } c_i; y_i \leftarrow (z_{i-1}, w_i); stmts_i)\end{array}}{T[y \leftarrow \text{fold } w\ x\ block] \to T[[z_0 \leftarrow x; stmts'_1; \ldots; stmts'_n; y \leftarrow z_n]]}\text{(fold)}$$

$$\frac{\text{value}(T, x) = value \qquad S = \text{spawn}(f, value)}{T[y \leftarrow f\ x], E \to T[[y \leftarrow ?S]], E \cup \{(S, \varnothing)\}}\text{(disp)}$$

$$\frac{term = (stmts; \text{return } x)}{T[y \leftarrow ?S], E \cup \{(S, term)\} \to T[[stmts; y \leftarrow x]], E}\text{(ext)}$$

(a)

$$\frac{(x \leftarrow \text{prim } c) \in T}{\text{value}(T, x) = c} \qquad \frac{(x \leftarrow (x_1, \ldots, x_n)) \in T \qquad \forall i. \text{value}(T, x_i) = value_i}{\text{value}(T, x) = (value_1, \ldots, value_n)}$$

(b)

Figure 6: The rewrite rules of the semantics of QUASAR (a), and the formal definition of the value function used by the "disp" rule (b). $T[stmt]$ means a program P with some statement $stmt$ in it; $T[[stmts]]$ means that the statement $stmt$ was replaced by the list of statements $stmts$. For each rule, the $P, E \to P', E'$ below the line is an allowed rewrite, subject to all of the conditions above the line. If E is not modified by a rule, we omit it for concision. $stmt \in T$ means that $stmt$ is in the list of statements of T.

```
found = False
cond = drink_patch.simple_query("...")
if cond:
    found = True
```

$\rightsquigarrow$

```
found0 = False
cond = drink_patch.simple_query("...")
def then_case():
    found1 = True
    return found1
def else_case():
    return found0
found2 = then_case() if cond else else_case()
```

Figure 7: An illustration of the translation of "if" statements, shown in Python syntax. Before, with imperative updates in "if" statements (a). After, with no imperative updates and a functional conditional operation (b).

to local variables. Handling updates in "straight-line" code (without control-flow structures like `if` and `for`) is straightforward. However, updates inside of control-flow structures, which thus may or may not happen, are more challenging. In the example from Figure 1, `found` may be updated inside of the loop.

Currently, QUASAR supports the subset of Python that uses function calls, local variable assignments, and `if`, `for`, and `while` control-flow constructs. It does not support early returns from loops (i.e. `break`, `continue`, or `return` inside of a loop). To support imperative control-flow structures in Python, we convert them into a functional form. `if` statements in Python are transformed as shown in Figure 7, where variables that might be updated by a statement-level conditional are instead returned from an expression-level conditional. A similar translation is done from imperative `for` loops to functional fold operations: variables that might be updated by the `for` loop are instead passed as the fold accumulator (in Python, this fold operation is called `reduce`).

$$
\begin{aligned}
op &::= \ldots \\
&\mid \text{absprim } \{c_1, \ldots, c_n\} \\
&\mid \text{abslist } [(c_1, b_1), \ldots, (c_n, b_n)] \\
&\mid \text{join } \{x_1, \ldots, x_n\} \\
absvalue &::= \{c_1, \ldots, c_n\} \mid (absvalue_1, \ldots, absvalue_n)
\end{aligned}
$$

Figure 8: The additional operations in the grammar of QUASAR to support conformal evaluation.

$$
\frac{y \leftarrow \text{join } y_1, \ldots, y_m}{T[x \leftarrow \text{join } x_1, \ldots, x_n, y] \rightarrow T[[x \leftarrow \text{join } x_1, \ldots, x_n, y_1, \ldots, y_m]]} \text{(join-join)}
$$

$$
\frac{(x_i \leftarrow (w_{i,1}, \ldots, w_{i,m})) \in T \qquad stmts_j = (y_j \leftarrow \text{join } w_{1,j}, \ldots, w_{n,j})}{T[x \leftarrow \text{join } x_1, \ldots, x_n] \rightarrow T[[stmts_1; \ldots; stmts_n; x \leftarrow (y_1, \ldots, y_m)]]} \text{(join-tuple)}
$$

$$
\frac{(x_i \leftarrow \text{prim } c_i) \in T}{T[x \leftarrow \text{join } x_1, \ldots, x_n] \rightarrow T[[x \leftarrow \text{absprim } \{c_1, \ldots, c_n\}]]} \text{(join-prim)}
$$

$$
\frac{\begin{array}{c}(x \leftarrow \text{absprim } \{\texttt{True}, \texttt{False}\}) \in T \\ \{w_1 \Rightarrow stmts_1; \text{return } z_1\} = \text{freshen}(block_1) \qquad \{w_2 \Rightarrow stmts_2; \text{return } z_2\} = \text{freshen}(block_2)\end{array}}{T[y \leftarrow \text{if } x\ block_1\ block_2] \rightarrow T[[w_1 \leftarrow (); stmts_1; w_2 \leftarrow (); stmts_2; y \leftarrow \text{join } z_1, z_2]]} \text{(if-tf)}
$$

$$
\frac{\begin{array}{c}(w \leftarrow \text{abslist } [(c_1, b_1), \ldots, (c_n, b_n)]) \in T \\ \forall i. \{y_i \Rightarrow stmts_i; \text{return } z_i\} = \text{freshen}(block) \qquad stmts'_i = (w_i \leftarrow \text{prim } c_i; y_i \leftarrow (z_{i-1}, w_i); stmts_i) \\ stmts''_i = \text{if } b_i \text{ then } stmts'_i \text{ else } (stmts'_i; z_i \leftarrow \text{join } z_i, z_{i-1})\end{array}}{T[y \leftarrow \text{fold } w\ x\ block] \rightarrow T[[z_0 \leftarrow x; stmts''_1; \ldots; stmts''_n; y \leftarrow z_n]]} \text{(fold-abs)}
$$

Figure 9: The additional rewrite rules in the semantics of QUASAR to support conformal evaluation.

C Conformal Semantics for QUASAR

In order to support conformal evaluation, QUASAR must be extended to support sets of values. The syntax has three additional operations, as shown in Figure 8. In order: abstract primitives represent one of a set of Python values, c_i; abstract lists represent a list where some of the elements may be uncertain: if b_i is `False` the element c_i may or may not be in the list, whereas if b_i is `False`, then c_i is definitely in the list; and a join operation, which combines two computations into a set. Join is distinct from an abstract set, since in the latter the values must be known, whereas in the former they may not yet be computed.

The semantics also contains additional rules in order to support these new operations, as shown in Figure 9. The rule "join-join" applies when x is the join of variables, and one of them, y, is itself a join, in which case they can be flattened to a single join; "join-tuple" applies when x is the join of n tuples of identical length m, in which case it becomes the tuple of joins of the respective components; "join-prim" applies when x is the join of n primitives c_i, in which case it becomes an abstract set of those values; "if-tf" applies when the condition of an if statement is the abstract set of both `True` and `False`, in which case both branches are taken, resulting in z_1 and z_2, which are joined to produce y; "fold-abs" applies when folding over an abstract list, in which case a copy of $block$ is made for each element of the list, however if a list element is uncertain (b_i = `False`), then the resulting accumulator z_i is joined with z_{i-1} to capture both the case when c_i is and is not in the list.