Abstract

Amazon Web Services (AWS) is a comprehensive and broadly adopted cloud provider. AWS SDKs provide access to AWS services through API endpoints. However, incorrect use of these APIs can lead to code defects, crashes, performance issues, and other problems. AWS best practices are a set of guidelines for correct and secure use of these APIs to access cloud services, allowing conformant clients to fully reap the benefits of cloud computing.

We present static analyses, developed in the context of a commercial service for detection of code defects and security vulnerabilities, to identify deviations from AWS best practices. We focus on applications that use the AWS SDK for Python, called Boto3. Precise static analysis of Python cloud applications requires robust type inference for inferring the types of cloud service clients. However, Boto3's “Pythonic” APIs pose unique challenges for type resolution, as does the interprocedural style in which service clients are used. We offer a layered approach that combines multiple type-resolution and tracking strategies in a staged manner: (i) general-purpose type inference augmented by type annotations, (ii) interprocedural dataflow analysis expressed in a domain-specific language, and (iii) name-based resolution as a low-confidence fallback. Across >3,000 popular Python GitHub repos that make use of the AWS SDK, our layered type inference system achieves 85% precision and 100% recall in inferring Boto3 clients in Python client code.

Additionally, we use real-world developer feedback to assess a representative sample of eight AWS best-practice rules. These rules detect a wide range of issues including pagination, polling, and batch operations. Developers have accepted more than 85% of the recommendations made by five out of eight Python rules, and almost 83% of all recommendations.

2012 ACM Subject Classification Theory of computation → Program analysis; Computer systems organization → Cloud computing

Keywords and phrases Python, Type inference, AWS, Cloud, Boto3, Best practices, Static analysis

Introduction

Amazon Web Services (AWS) is a comprehensive and broadly adopted cloud provider. AWS best practices are a set of guidelines for correct, secure, and performant usage of AWS cloud SDKs. Python is used extensively to build applications on top of the AWS cloud, using the AWS SDK for Python, called Boto3. We report on our experience developing static analyses to enforce AWS best practices in Boto3-based Python applications. These rules are evaluated as part of a commercial cloud service, Amazon CodeGuru Reviewer (henceforth, CodeGuru) [9], that runs static analysis on customer code to detect security vulnerabilities, optimization opportunities, and other defects. Figure 1 shows the CodeGuru architecture.
14:2 Static Analysis for AWS Best Practices in Python Code

![Figure 1 High-level overview of CodeGuru](image)

CodeGuru supports Java and Python, and integrates with different code hosting platforms including GitHub and BitBucket. CodeGuru supports three code scanning modes:

- **Incremental**: A code review is created automatically when a pull request is raised.
- **Full**: The entire codebase is analyzed.
- **CI/CD**: The entire codebase is analyzed as part of CI/CD workflows.

### 1.1 Importance of AWS Best Practices

Deviation from AWS best practices can lead to large-scale operational failures. Consequences include race conditions leading to service outage or auto-ticketing errors; authorization and authentication errors; broken throttling mechanisms that impose unexpected loads on services, thereby leading to high latency or timeouts; missing or incorrect error handling leading to billing errors; and many other severe problems. Risks are commonly discovered by manual inspection or testing. However, many such cases can be detected, and prevented, by applying static analysis to clients of the AWS SDK. The AWS best practices rules that we have developed alert developers to such defects during code review, before customer impact.

We provide lower-bound metrics to give an idea of CodeGuru’s throughput. In an average week, CodeGuru analyzes $\gg 10,000$ pull requests (PRs) containing $\gg 1,000,000$ lines of code across $\gg 100,000$ files, and provides $\gg 1,000$ AWS best practices recommendations due to $\gg 100$ different static analysis detectors.

### 1.2 Scope

CodeGuru supports AWS best practices for both Java and Python. We focus on Python given its dynamic nature and lack of strict static typing. For precise enforcement of AWS best practices, it is essential to identify function calls into the AWS SDK, and which service in particular is used. Java reveals this information through static types, but in Python this information is not available by default: a challenging start for our analyses.

We describe several on-demand type resolution strategies and combinations thereof. We consider three core approaches: (1) Boto3 type stubs, in combination with general-purpose type inference, to resolve types when processing the Python AST; (2) on-demand interprocedural dataflow tracking, in both the forward and the backward directions, to check whether the receiver of a function call corresponds to a given AWS service; and (3) a lightweight over-approximation that simply checks whether the called function’s name is compatible with a given AWS service’s API. We present the approaches themselves and more
advanced algorithms that combine these approaches. We also provide technical details on
the underlying infrastructure that enabled us to implement these approaches: CodeGuru's
code representation and language for rule specification.

1.3 Main Contributions

This principal contributions of this paper are as follows. (1) We offer an on-demand type
resolution strategy, which we demonstrate as effective in the case of Python clients of the
AWS SDK. (2) In support of the above-mentioned strategy, we present the intermediate
representation (IR) and query language used by the CodeGuru service. (3) We describe a
representative sample of the AWS best practices rule suite running as part of the CodeGuru
service. (4) We share our evaluation on 3,027 GitHub repositories, and real-world feedback
we received from developers, to validate our approach.

1.4 Paper Structure

The rest of the paper is organized as follows. Section 2 discusses related work. In Section 3,
we present background about the Boto3 SDK. Section 4 shows several examples that motivate
the need for advanced type inference. Sections 5 and 6 lay down the technical infrastructure
for our approach in describing, respectively, the code representation and query language
that we use to express Python AWS best practices rules. Section 7 describes the different
type inference capabilities we have developed, based both on Boto3 type stubs and data-flow
tracking. In Section 8 we examine eight representative Python AWS best practices rules.
Section 9 states our research hypotheses and reports on experiments to assess the efficacy of
our type inference strategies. Section 10 concludes and outlines future research.

2 Related Work

Different approaches have been taken to infer Python type annotations, and formalize Python
semantics more generally. We review approaches based on program analysis as well as
machine learning, and compare these approaches with CodeGuru.

2.1 Classical Program Analysis

Widely used Python type checkers include mypy [25], Pyre [15], pytype [20], and Pyright [26].
These tools rely on manual type annotations provided by developers, augmented with
varying forms of type inference. However, retrofitting type annotations onto large libraries or
applications can be tedious and error-prone. Other prior work places more emphasis on static
analysis [10,16,17,22,27,32] or dynamic analysis [34] to reduce reliance on human-authored
annotations. Our initial search for supporting infrastructure found that many published tools
have failed to keep up with recent Python releases, or omit support for key Python features
such as exceptions [16] or recursion [27]. We opted to use Pyright as our baseline, as Pyright
is both actively maintained and has a rather advanced inference engine (see Section 7.1).
In spite of these advantages, Pyright alone proved unsatisfactory for our cloud application
domain. The details, as conveyed in Section 9, may serve to highlight challenges for other
developers of general-purpose type inference engines.

When writing type annotations, Python developers often focus on function signatures:
arguments and return values. Some research tools mirror this bias, such as TypeWriter [30].
xu et al. [36] present a probabilistic type inference system, but the accuracy of probabilistically
inferred types for Python variables is limited. Our work requires accurate types for variables, making these two approaches unsuitable.

Any attempt to statically analyze Python code must contend with the intricacies of the Python language. Notable efforts to formalize Python semantics include those by Smeding [33], Politz et al. [29] and Köhl [24]. Smeding’s work predates Python type annotations, while neither Politz et al. nor Köhl mention them in any way. These omissions are not surprising, as type annotations have only limited effects on runtime behavior. Thus, these codifications of Python semantics offer little insight regarding the type-inference challenges addressed here. Our approach is neither sound nor complete (see Section 5.4), so a standard type-soundness theorem relating static types to runtime semantics does not apply.

In the specialized domain of machine learning, where Python is perhaps the most popular language, WALA Ariadne [13] analyzes Python specifically to infer the dimensions and types of tensors. Like Ariadne, our work is motivated by a specific application domain, and even a specific framework: Ariadne focuses on machine learning using TensorFlow [1]; CodeGuru focuses on cloud computing using Boto3. Ariadne’s solution entails both a custom type system and an analysis to infer it. Our approach builds upon standard Python types and type annotations. While we crafted our analysis strategy to match idiomatic Boto3 use, these idioms are not exclusive to Boto3 client code. Therefore our layered approach may be more broadly applicable.

2.2 Machine Learning

PYInfer [12] uses deep learning to generate type annotations for Python. PYInfer fuses deep learning with static analysis such as PySonar2 to infer types for variables as well as function-level types in Python. All of these techniques either require labelled type annotations or employ a static analyzer to generate the initial annotations from Python repositories in order to train the deep neural network. However, type resolution for Boto3 service clients is non-trivial due to the reasons mentioned above.

JSNice [31], DeepTyper [23], and LambdaNet [35] use deep learning to generate type annotations for JavaScript and/or TypeScript. LambdaNet’s authors note that TypeScript is an inviting target because “plenty of training data is available in terms of type-annotated programs.” In principle, similar strategies may be applicable to Python. However, it is unclear whether the available corpus of type-annotated Python Boto3 client programs is large enough for effective training in practice.

3 Background on Boto3: the AWS SDK for Python

This section describes the AWS service clients in the AWS SDK for Python, also called “Boto3”. [4]

3.1 Clients and Resources: Low- and High-Level APIs

Boto3 has two distinct levels of APIs:

Client (or “low-level”) APIs provide one-to-one mappings to the underlying HTTP API operations.

Resource APIs hide explicit network calls but instead provide resource objects and collections to access attributes and perform actions. Resources represent an object-oriented interface to AWS. They provide a higher-level abstraction than the raw, low-level calls made by service clients.
A low-level service client can be created by passing the name of service as an argument to the `boto3.client` method. For example, the Python statement,

```python
s3_client = boto3.client('s3')
```

creates a low-level client for the Amazon Simple Storage Service (S3). Conversely, a service resource can be created by passing the name of service as an argument to the SDK `boto3.resource` method. For example, the Python statement,

```python
s3_resource = boto3.resource('s3')
```

creates an Amazon S3 service resource. It is also possible to access the low-level client from an existing resource, as in:

```python
s3_client = s3_resource.meta.client
```

Alternatively, to use service resources, one can invoke the `resource()` method of a `Session` and pass in a service name. For example, one can create an Amazon S3 service resource using:

```python
session = boto3.session.Session()
s3_resource = session.resource('s3')
```

Service clients give access to service operations by calling methods on a client. For example, suppose `s3_client` is an S3 client. Then one can create an S3 bucket, with the bucket name passed via an argument, using:

```python
response = s3_client.create_bucket(Bucket=bucket_name)
```

### 3.2 Boto3 Type Stubs

Boto3-stubs provides full type annotations for Boto3. In particular, Boto3-stubs provides annotations for a `Client` type, `ServiceResource`, and `Resource` type for each AWS service. It also provides annotations for a `Waiter` type, and a `Paginator` type for each service. With help from Boto3-stubs, several Python type-checking tools can discover types for multiple flavors of client construction calls such as `boto3.client`, `boto3.session`, `session.client`, and `session.session`.

### 3.3 API Specifications From Boto3

Some of the AWS best practice rules that are presented in this paper use an external configuration that provides a specification of some service-specific fragment of the complete Boto3 API. This specification includes an API name, type, the service name the API belongs to, and few other attributes that are relevant for the rule. We refer to these external configurations as *API specifications*. One such example is presented in Section 9. API specifications are automatically extracted from Boto3 API models. [6] These API models have specific traits, such as, *Pagination*, *Batch*, *Deprecated*, *Waiters*, or *mutual-exclusion*, which help determine the characteristics of the API. We extract relevant API traits from API models across Boto3 services to construct the complete API specification to enforce. These API specifications are then used by the best practice rules for analyzing client code.

### 4 Motivating Examples

This section presents an example that motivate the need for sophisticated type inference to recover the types of AWS service clients in real-world Python applications. The type annotations in Figure 2 are obtained from Pyright with Boto3 type stubs, which are on lines with the prefix “#→”. 

---

**Figure 2:** Type annotations obtained from Pyright with Boto3 type stubs.
import boto3

class Example(object):
    def get_sns_client():
        return boto3.resource("sns")
    
def M1():
        sns_arn = os.environ["PUBLISH"]
        client = get_sns_client()
        #→ client: SNSServiceResource
        M2(client, topic, subscription)
        return client.Topic(sns_arn)
    
def M2(client, topic, subscription):
        topic = client.Topic(topic)
        #→ (variable) client: Any

Figure 2 Example of a Python application code using Boto3

Example 1: Consider the Python code snippet in Figure 2. Here, the Boto3 client is returned by get_sns_client(). Its type is SNSServiceResource, marked in bold in method M1. This type correctly identifies client as a client for the Amazon Simple Notification Service (SNS). Figure 2 creates client using the boto3.resource() API which gives an object-oriented interface to SNS. [8]. The client flows into M2 via a function parameter. M2 uses client to make API call, topic(). Unfortunately, Pyright was unable to assign client a precise type, leaving it typed simply as the generic Any inside M2. Inference falls short here because Pyright cannot guarantee that client must always be an SNSServiceResource in every possible call to M2. This is safe but, for our purposes, unfortunate: an untyped client cascades into untyped topic and subscription, leaving us with nothing useful to analyze for any of the API calls in M2.

Type resolution of the variable client requires sophisticated type inference coupled with a domain-aware preference for finding Boto3 clients wherever they might arise and be used for API interaction. In this paper, we present a technique that combines Pyright’s type inference with a custom interprocedural dataflow analysis to infer types in such cases.

Furthermore, these API names are exactly the same in Google’s Pub/Sub cloud service [19] and AWS’s SNS service. Our study shows that the names of some cloud service APIs are exactly the same for cloud services from different commercial cloud vendors (AWS, Google, Tencent, etc.). Thus, precise resolution of service clients’ types is extremely important for static analysis of Python applications that use these cloud SDKs.

5 Program Representation

Our analysis represents each program as a collection of per-function graphs called MU graphs.¹ A MU graph roughly corresponds to a data-dependence graph overlaid with a control-flow (not control-dependence) graph (CFG). As in prior work that used similar representations [2,3], we find this representation useful for finding API misuse defects where both the data flowing into an operation and the order of operations are important.

¹ “MU” originally stood for “misuse”, and is pronounced as the name of the Greek letter μ.
5.1 MU-Graph Nodes

MU graphs contain five kinds of nodes. **Entry nodes** represent the start of a function’s execution: one per MU graph. **Exit nodes** represent the end of a function’s execution: one per MU graph. **Control nodes** represent branched control flow, such as a conditional statement or loop. **Action nodes** represent individual execution steps, such as multiplying two values or calling a function. **Data nodes** represent local variables or synthetic temporary values within compound expressions.

Per-node metadata identifies specific uses of these general categories. For example, we distinguish a multiplication action from a function-call action, or an if-statement control node from a while-statement control node.

Multiple assignments to the same local variable use multiple data nodes, as in static single assignment (SSA) form. ϕ action nodes are added as needed to represent converging data flows, such as when both branches of an if statement modify the same variable.

5.2 MU-Graph Edges

**Control edges** order execution among entry, exit, control, and action nodes. No data node is ever the source or target of a control edge. Thus, discarding all data nodes and non-control edges would reduce a MU graph to a traditional CFG. **Data edges** represent movement of data among control and action nodes, and are further categorized as follows:

- **Condition edges** flow from a data node into a control node, representing the information used to decide how execution continues. For example, a condition edge flows from the value of an if statement’s predicate to the control node for the statement itself.
- **Definition edges** flow from an action to a data node defined by that action. For example, a definition edge from a multiplication action to a data node indicates that the action node's operand receives the result of that multiplication.
- **Parameter edges** flow from a data node into an action node. For example, a binary multiplication action is the target of two parameter edges, one for each operand. A function call action is the target of one parameter edge for each actual argument.
- **Receiver edges** flow from a data node into a method-calling action node. These highlight the special role of implicit self or this arguments.
- **Callee edges** flow from a data node into a call action node, identifying the function to be called. For example, in handlers[event](), an indexing action to fetch handlers[event] would define some temporary data node holding the function to call. A callee edge would then flow from that data node to the call action.

Edges carry additional role-specific metadata. For example, the two control edges that depart from an if statement’s control node are marked to distinguish the true and false branches. Multiple parameter edges leading to the same action node are ordered, thereby distinguishing an action’s first parameter from its second, third, and so on.

5.3 Overall Properties

In the MU representation, data can only flow from data nodes to control/action nodes, and vice versa. Data edges never connect pairs of data nodes directly. Informally, each action node receives zero or more data nodes as inputs, and may provide an output that flows across a definition edge into some other data node. In \( x + y \times z \), the multiplication action defines an anonymous data node, which in turn flows into the addition action as a parameter.

Figure 3 illustrates several MU-graph features in the representation of \( x \times x \times x - 1 \), or equivalently \( x = x \times (x - 1) \). Solid control edges establish evaluation order as in a CFG: subtraction before multiplication, each represented as a rectangular action node. Elliptic data nodes represent two versions of \( x \): \( x_1 \) before the assignment and \( x_2 \) after. Additional
Figure 3 The MU-graph representation of $x \ast= x - 1$. Entry and exit nodes are trapezoidal; action nodes are rectangular; data nodes are elliptic. Control edges are solid; data edges are dashed.

data nodes represent the literal 1 and a temporary value. The initial value $x_1$ is a parameter to both mathematical operations, and is distinct from the final value $x_2$. The “temp” data node is defined by the subtraction and is also a parameter to the multiplication. Notice that data and non-data nodes strictly alternate along data paths: data nodes provide inputs to action or control nodes, and action nodes’ outputs define data nodes.

5.4 Using Pyright for Best-Effort Graph Construction

Pyright is “a fast type checker meant for large Python source bases.” [26] Pyright is primarily used behind-the-scenes by Python IDEs, or as a command-line linter/checker. However, Pyright’s sophisticated type inference and robust handling of incomplete or incorrect programs make it ideally suited for our purposes as well. MU graph construction begins with a parsed abstract syntax tree (AST) provided by Pyright. We traverse the AST, synthesizing and combining MU graph fragments in a roughly bottom-up manner.

For data nodes, we rely on Pyright to provide static type information and name resolution. Given Python’s dynamic nature, these are both best-effort. Inferred static types can be imprecise, absent, or wrong; names can be aliased or accessed covertly via reflection. We attempt no alias analysis or points-to analysis beyond that implicitly performed by Pyright itself. Pyright’s best-effort types are available on data nodes that represent named variables as well as those that represent intermediate values, such as the “temp” node in Figure 3.

We flatten data node types to their string representations, such as "int" or "MyClass" or "(int, str) -> tuple[int, str]". Stringification discards internal structure, but allows MU graphs to accommodate essentially any type grammar, even from non-Python languages. Types as strings are also forgiving of incomplete programs: we might know that a piece of data is an instance of MyClass even if we know nothing about MyClass’s internal structure or provenance.

The entire process of building MU graphs proceeds, best-effort, even when confronted with imports of missing modules, calls to unknown functions, etc. We represent each questionable operation as some reasonable fallback (e.g., as an empty statement), and move on. Python also contains syntactically ambiguous constructs, such as overloaded operators or the myriad uses of ".". We disambiguate these using types whenever possible, or heuristics when necessary. These approximations mean that we are neither sound nor complete in general. However, these same approximations allow us to provide a representation that is useful in practice when absolute guarantees are not required.
CustomRule rule = new CustomRule.Builder()
    .withName("MathExp")
    .withComment("For small floats `x`, the subtraction in `exp(x) - 1` can result in a loss of precision.")
    .withAllOf(
        b -> b.withMethodCallFilter(".*math盘点.exp").withDefinitionTransform().as("MathExpResult"),
        b -> b.withConstantDataFilter("1").as("ConstantOne")
    )
    .check()
    .withActionFilter("\=\")
    .withDirectDataFromIdFilter("MathExpResult")
    .withDirectDataFromIdFilter("ConstantOne")
    .build();

Figure 4 GQL rule for identifying suboptimal use of the `math.exp` function.

5.5 From Functions to Programs

The construction process described in Section 5.4 yields one MU graph for each named (def) or anonymous (lambda) function. For each script, also create MU graph that represents execution of that script’s top-level statements.

We aggregate these per-function MU graphs to reflect static program structure. Each Python class contains a dictionary of named methods; each script contains a dictionary of named top-level classes and functions; and so on. We do not build a static call graph, since not all downstream consumers of MU graphs require one. However, we organize and manage the MU graph collection in such a way as to facilitate callee resolution later, if needed.

6 Query Language

Working directly atop the MU representation in authoring analysis rules misses important reuse opportunities. We have therefore designed and implemented an API, dubbed the Guru Query Language (GQL), to enable encapsulation, optimization and reuse of a wide variety of analysis constructs. GQL is implemented as a Java library whose main interface with the analysis builder is the CustomRule class. CustomRule instances are created using the fluent builder pattern [18], where builder calls correspond to reasoning steps in the rule. A rule object can be evaluated at different scopes, from entire code bases to single functions. This is an important source of flexibility, which owes to the MU representation and its support for partial programs. (See Section 5.4.) Rule evaluation yields a RuleEvaluationResult for every type and function that the rule visits, which includes rich information on whether, and if relevant where and how, rule evaluation has failed.

As an illustration of GQL syntax, we refer the reader to Figure 4, where a rule that identifies suboptimal use of the `math.exp` function is shown. Here is a simple example of what the rule checks for:

```python
def foo():
    import math
    return math.exp(1e-10) - 1
```

Rule definition begins by setting the rule’s name and user-facing comment text. The following steps, up to the check statement, are preconditions that the rule checks for. Specifically, the withAllOf statement ensures that all the subrules nested within it evaluate successfully, where these check for `math.exp` calls as well as the presence of the constant value 1. The matches are stored into variables (or IDs), to enable downstream reuse thereof, using the as operation. The actual check, or postcondition, is the rule section after the check step.
It establishes whether there is a subtraction operation that the node defined by `math.exp`, along with the constant 1, flow into directly (that is, without the mediation of any other action).

### 6.1 Rule Evaluation

In what follows, we use standard notation, $G = (V, E)$, when referring to MU graphs. Unless stated otherwise, the graphs we mention are specifically MU graphs.

As illustrated above, a GQL rule is an implication relation, $\text{pre} \implies \text{post}$. As such, rule evaluation is satisfied either when $\text{pre}$ is not satisfied or when both $\text{pre}$ and $\text{post}$ are satisfied. $\text{pre}$ and $\text{post}$ are both sequences $[\text{op}]$ of operations.

An operation $\text{op}: \mathbb{P}(V) \rightarrow \mathbb{P}(V)$ is a function whose domain and codomain are both node sets: $V = \{ n : \exists G = (V, E), n \in V \}$. As an example, a filter operation that matches against calls to a function named “foo” evaluates to foo call nodes within the incoming node set, if any, or else $\emptyset$.

Given node $n$, let $G_n$ denote the graph containing $n$, and $G_n.V$ the complete set of nodes that $G_n$ contains. Operations $\text{op}$ satisfy the following two invariants:

1. $\forall N \subseteq V. \text{op}(N) \subseteq \bigcup_{n \in N} G_n.V$. That is, application of an operation to a node set $N$ cannot “exceed” the set of nodes due to the graphs containing the nodes in $N$.

2. $\text{op}(\emptyset) = \emptyset$. That is, application of an operation to the empty node set yields the empty node set.

Given rule $r = [\text{op}_1, \ldots, \text{op}_k] \implies [\text{op}_{k+1}, \ldots, \text{op}_n]$ and input graph $G = (V, E)$, we denote the node set flowing into $\text{op}_j$ as $\sigma_{j-1}$. The node sets are defined as follows:

$$
\sigma_i = \begin{cases} 
V & \text{if } i = 0 \\
\emptyset & \text{if } i = k \land \text{op}_k(\sigma_{k-1}) = \emptyset \\
\text{op}_i(\sigma_{i-1}) & \text{otherwise}
\end{cases}
$$

Per the first case, precondition evaluation starts from the complete set of graph nodes ($V$). Per the second and third cases, the transition from precondition to postcondition is either trivial if the precondition has not been satisfied (second case), or — analogously to precondition evaluation — postcondition evaluation starts from $V$ (third case). Any other transition along the operation sequence is simply an application of the operation to its incoming node set.

Rule evaluation is successful if and only if (i) a prefix of $\text{pre}$ evaluates to $\emptyset$ (in which case the precondition is not satisfied); or (ii) both $\text{pre}$ and $\text{post}$ evaluate to non-empty node sets (in which case the precondition and postcondition are both satisfied).

To add color to the formal description so far, rule evaluation is essentially a process of matching against a pattern, or semantic property, where a non-empty node set is a match frontier that feeds into the next reasoning step. Failure to maintain a non-empty match frontier means that the given (pre or post) condition is not satisfied by the input function.

### 6.2 Rule Structure

While our formal presentation above of GQL rules is as logical implication relationships, in practice a rule object has additional information and structure. A GQL rule consists of four sections, as follows: (i) setup: the rule’s name, and the comment (or description) associated with the rule; (ii) function matcher: a rule can optionally define criteria when to be evaluated, for example based on function name, attributes, annotations, containing type,
parameter types, and so on; (iii) **precondition**: the sequence of operations up to the check builder step; and (iv) **postcondition**: the sequence of operations following the check builder step.

Since GQL rules follow the fluent builder pattern, there is risk that users would miss, misuse, or reorder rule constructs or sections. For example, the user might build a rule lacking a check step; forget to set the rule’s name; or try to apply incompatible filters in succession. To ensure rule integrity, we employ a hybrid solution that combines metadata contributed by operations with runtime checking. Operations expose a “signature”, as explained in Section 6.4, such that improper compositions can be detected and localized ahead of rule evaluation.

### 6.3 Language- and Domain-specific Rule Constructs

Beyond the core GQL constructs, which are applicable across different programming languages and problem domains, there are reusable albeit language- or domain-specific constructs. As an example, constructs like `withNamedArgumentsTransform` or `withUnpackedArgumentsTransform` are useful for Python rules, but do not apply to Java. GQL enables such constructs to be organized into subclasses of `CustomRule`, such as `PythonCustomRule`, while containing `CustomRule` to the core analysis constructs.

This approach has several important advantages. First, we avoid API bloat by distributing analysis constructs across more than just `CustomRule`. Second, we avoid misuse errors due to a construct being used outside its intended context, for example a Python analysis construct used in a rule that targets Java programs. Finally, GQL extensions sometimes introduce dependencies. We have implemented, for example, a `CustomRule` extension in the domain of data leaks, where some of the analysis constructs rely on an ML model to predict whether a given data access is retrieving sensitive information. These dependencies should not be forced on GQL users outside the given domain.

### 6.4 GQL Operations

We now take a closer look at the different operations that comprise GQL rules. These divide into 4 categories, discussed below in turn. Beyond the information in this section, we refer the reader to the accompanying technical report for a more detailed description of the operation categories as well as examples from each category [28].

For safety and fault localization, GQL requires that operations be annotated with their **signature**, which states the types of nodes that they accept as input and yield as output. (See Section 5.) The `withReceiverTransform` operation, for example, accepts as input action (and more specifically, call) nodes, and outputs data nodes. If a user attempts to compose operations incorrectly, for example by routing the output of a `withDataByNameFilter` operation to `withReceiverTransform`, then GQL identifies the violation at runtime and generates a meaningful failure message that localizes it and explains why rule evaluation has been terminated. We are currently in the process of shifting the failure left to rule building time, and as a longer-term objective, compile time.

#### 6.4.1 Core Operations

Core operations apply to all rules, regardless of their scope and logic. Some of the core operations, in particular `check` and `as`, have already been explained in the context of Figure 4. Additional core operations include the ability to reset the match frontier,
interleave instrumentation (for example, for debugging or profiling purposes), read and write mutable auxiliary state, and so on.

6.4.2 Filter Operations

A filter operation $f$ satisfies the invariant: $\forall V \in \mathcal{V}. f(V) \subseteq V$. That is, a filter operation selects a subset of the input node set. Its result cannot exceed the incoming set.

GQL offers a wide selection of built-in filters. Beyond `withActionFilter`, `withMethodCallFilter`, `withConstantDataFilter` and `withDirectDataFromIdFilter` that are used in Figure 4, there are filters for matching against control structures, constants, actions with specific arguments (like constants or `null/None`), and so on.

The GQL filter operations — almost without exception — are defined using a unary predicate ranging over nodes, and as such, filtering is done point-wise. As an example, `withMethodCallFilter` is instantiated through a predicate that accepts action (and specifically, call) nodes where the callee matches the provided regex specification. A common practice with filter operations is to compose them, which enables refinement in pattern matching. An example of that is the consecutive `withDirectDataFromIdFilter` operations in the rule in Figure 4.

6.4.3 Transform Operations

Transform operations enable the transition from a given match frontier to another frontier that derives from it. For example, a frontier that consists of function calls can be transformed to the respective arguments or receivers, or the values defined by the calls, as illustrated with `withDefinitionTransform` in Figure 4.

GQL offers many built-in transform operations. Examples include `withArgumentsTransform`, which transforms an action node to its respective arguments; `withControlDependenciesTransform`, which transforms a node to its set of control dependencies; `withDataDependenciesTransform` (resp. `withDataDependentsTransform`), which transforms a node to its set of (transitive) data dependencies (resp. dependents); and `withReceiverTransform`, which transforms a call node to the receiver (if available).

6.4.4 Second-order Operations

Logical structures and operators are necessary to express certain rule logic in a precise and concise manner. As a simple example, the user may wish to check if a given function call "zoo" has a receiver of type either `Foo` or `Bar`. Another use case, illustrated in Figure 4, is the need to check that several conditions are all met through `withAllOf`.

To enable such control and logical structures, GQL exposes second-order operations. These are operations that are themselves parameterized by one or more rules, which we refer to as subrules.

As an illustration, here is the GQL syntax for the above example:

```gql
.withMethodCallFilter("zoo")
 .withOneOf(
     b -> b.withReceiverByTypeFilter("Foo"),
     b -> b.withReceiverByTypeFilter("Bar"))
```

The `withOneOf` construct evaluates to the first subrule that yields a non-empty result, or else it evaluates to $\emptyset$. 
6.5 Interprocedural Analysis

As noted above, GQL provides the ability to perform interprocedural analysis through the \texttt{withInterproceduralMatch} construct and several specializations thereof. The underlying call-graph representation resolves call sites on demand, per the CHA call-graph construction algorithm [21], based on the (i) name of the callee, (ii) number of arguments, and (iii) argument types. Though the CHA algorithm is known to be imprecise [11], we have rarely seen cases where that was the cause of imprecision in GQL rule evaluation. We hypothesize that this is because (i) interprocedural analysis is run at file or package scope, but not beyond, so there is less room for error, plus (ii) imprecision in interprocedural analysis is potentially mitigated by other rule steps.

At a high level, the interprocedural tracking algorithm performs a fixpoint computation starting from the seeding function graph and matched nodes therein. At each step, the argument rule is applied to match against additional nodes. The algorithm is parametric, enabling the user to decide the scope (intra-class, intra-file, or entire codebase) and direction (forward or backward) for tracking. In the forward direction, the algorithm transitions from a call site to the callees and from a function’s exit to callers. In the backward direction, the algorithm transitions from a function’s entry to call sites and from call-site definitions (for example, \texttt{x = foo()}, where \texttt{x} is tracked) to callee exits.

Functional summaries are utilized to avoid redundant computation. In the forward direction, these document the relationship between a call-site argument and the definition (if exists) plus other arguments. In the backward direction, the summary documents the relationship between the definition and call-site arguments.

A more complete, and technical, explanation of the GQL interprocedural tracking algorithm is available in the accompanying technical report [28]. The description there ties into a pseudocode description of the algorithm.

6.6 Dataflow Analysis

Beyond its interprocedural capabilities, GQL also has built-in support for several flavors of dataflow analysis, including slicing and taint tracking. These build directly on top of the data edges exposed by the MU representation, in conjunction with the interprocedural matching algorithm described above.

The main feature that the GQL dataflow analysis provides beyond a standard fixpoint algorithm over the dataflow relation is the ability to specify matchers on graph edges to tag them with unique roles: \texttt{passthrough} (data flows across the call site), \texttt{blocking} (an edge being either a sanitizer or a validator), \texttt{side effecting} (data flows into the receiver of a call), or \texttt{reading} (data flows from the receiver to the definition). The user-provided specification is then enforced as part of the fixpoint algorithm.

7 Type Inference for Boto3 Clients

As explained in Section 3.1, a Python AWS application creates an AWS service client by passing the name of the service as an argument to one of two distinct levels of APIs. The

\footnote{GQL additionally features finite state machine (FSM) and typestate analysis, though these involve not just dataflow but also control-flow reasoning. These capabilities are not consumed by the rules that we discuss later in the paper, so we suffice by noting them here.}
use of these multiple API flavors, the interactions between them, and the use of strings as service selectors, all pose challenges for type inference.

Regardless of which API is used, AWS service clients are ultimately just data values. Like any other data, service clients can be stored in class variables, assigned into global variables, returned from functions, and so on. Code might use a service client locally within a single function or globally within or even across the files that comprise the complete application. The complexity of these definition-use chains (DU chains) further complicates type inference.

In this section, we present different type inference strategies that can be used in this challenging application domain.

7.1 Pyright’s Type Inference With Boto3-Stubs

Pyright supports type inference for function return values, instance variables, class variables, local variables, and global variables. Pyright’s inference engine uses several advanced type inference techniques, such as a flexible model of “type assignability”, inferred types for self and cls, parameterized generic types, including both polymorphic container types as well as optional types, union types representing arbitrary sets of possible types, overloaded function types as a special case of union types for ad hoc polymorphic functions, literal types, such as Literal["str"] as a subtype of str that represents only the string literal "s3", and few others.

A full discussion of these capabilities is outside of the scope of this paper, and in any case Pyright is not our contribution. We treat Pyright’s type inference engine as a powerful, featureful, but opaque black box.

If Pyright cannot infer the type of some symbol, then that symbol’s type is set to Any. This fallback type is a useful warning marker that lets inference consumers (such as CodeGuru) recognize cases where Pyright type inference fell short.

Type inference can incur significant computation overhead for large code bases. Also, Pyright cannot always infer correct types without some outside help. Hence, type annotations are a practical requirement for building a robust type inference system. We use third-party type stubs, called Boto3-stubs [14], that provide full type annotations for Boto3. Pyright ingests type annotations provided by Boto3-stubs to further enhance and constrain its type inference.

Figure 2 give an examples of Pyright’s Type Inference with Boto3-stubs (denoted by the prefix “#→”). However, in Figure 7, Pyright fails to infer a precise type for s3_client in the method load_df_from_s3, instead giving it the fallback Any type.

7.2 Type Inference Using Custom Dataflow Rules

As an alternative to Pyright, we have used GQL to implement custom inference rules based on dataflow analysis. These rules do not provide universal, generic type inference. Instead, they focus on idiomatic, interprocedural Boto3 usage patterns that Pyright’s general-purpose engine fails to address. There are a total of ten GQL-based custom dataflow rules, among which only one is intraprocedural rule and rest nine are interprocedural rules. For illustration purpose, we select few representative interprocedural GQL rules that have low to medium complexity (in terms of number of operations in the rules) and that performs dataflow analysis at file-scope or package-scope.

7.2.1 Representative Examples of Interprocedural Rules

Each GQL rule in Figures 5–6 implements some form of interprocedural dataflow analysis. Each operates on a function graph and matching API nodes along with the receiver nodes
of calls to the corresponding APIs. For example, in Figure 7, one relevant API node is `get_object`, for which the corresponding receiver node is `s3_client`. Our strategy for resolving call actions to callees is name-based: we match the name of the API entry point (callee) in the code against API specifications that are extracted from Boto3.

Figure 5 shows one such rule. The scope of this rule’s interprocedural match operation is `FILE_FORWARD_REACHABLE`, which directs GQL to track dataflow forward using a “data dependents” transform operation that transforms from incoming nodes to nodes that are data dependent on them, including in other functions. The result of this interprocedural tracking is then checked to determine if it matches one of the known flavors of Boto3 clients (low-level or object-oriented), by calling the utility methods inside the `withOneOf` operation.

The rule in Figure 6 implements interprocedural backward dataflow analysis, complementary to the forward analysis of Figure 5. For the backward version, tracking is specified as `FILE_BACKWARD_REACHABLE`. This scope directs the interprocedural analysis to perform backward dataflow tracking using a “data dependencies” transformer that transforms from incoming nodes to nodes that are data dependent on them, including in other functions. Similar to the previous rule, this rule’s `withOneOf` clause then checks whether the result of backward interprocedural tracking matches one of the known flavors of Boto3 clients.

### 7.2.2 Example of Type Inference Using Custom Dataflow Rules

Figure 7 shows a Python code snippet with variable- and function-level type annotations from Pyright. The type of `s3_client` in the method `write_df_to_s3_location` is correctly inferred as `S3Client`: an Amazon S3 service client. This client is passed via input parameter to the method `load_df_from_s3`. In absence of the type annotation for the input parameter, Pyright could not infer the type of `s3_client` (denoted by `Any`), inside the method `load_df_from_s3`.

However, one of our custom dataflow rules can resolve the type of `s3_client` in method
def write_df_to_s3_location(file_path, bucket_name, metadata, sep=None):
    s3_client = create_s3_client()
    #→ s3_client: S3Client
    load_df_from_s3(s3_client, bucket=bucket_name, path="")
    s3_client.put_object(Body=file_path, Bucket=bucket_name)

def create_s3_client():
    return Boto3.client("s3")
    #→ create_s3_client: () → S3Client

def load_df_from_s3(s3_client, bucket, path):
    raw_data = s3_client.get_object(Bucket=bucket, Key=object_path)
    #→ s3_client: Any

Figure 7 Type annotation for AWS client passed by input parameter

load_df_from_s3. The applicable rule starts from a matching API node, s3_client.get_object, where the type of the receiver node s3_client needs to be determined. Recall that the matching API node is obtained by matching the name of the API against the API specification extracted from Boto3. Starting from a matching API node, the rule uses a “parameter transform” operation that transforms incoming nodes to the parameters of the respective functions. This rule then uses a “backward data dependencies” transform that transforms from incoming nodes to their data dependencies, including in other functions. The rule’s result includes the node s3_client in the method write_df_to_s3_location, whose type is already known to be S3Client. It is worth noting that the type of s3_client could also be inferred by a stand-alone custom dataflow rule (in absence of type annotations from Pyright). However, the rule specification would be more complex. We prefer to augment Pyright’s capabilities rather than replace them.

7.3 Layered Type Inference

The example in Figure 7 shows that a hybrid approach for type inference can combine custom dataflow rules with Pyright’s type inference to resolve types that Pyright cannot resolve by itself. Each of these type inference approaches have complementary strengths. This quality suggests a layered approach for type inference that combines these strategies in a staged manner. Our layered approach first uses Pyright’s type inference with Boto3 stubs to infer type annotations for at least some Boto3 clients. Per Section 5.4, data nodes in MU graphs carry type metadata reflecting Pyright’s inference results. If the type of an API call of interest is already known, then that may be sufficient to recognize that the API belongs to Boto3. If the type of the API call of interest is unknown, then our layered approach deploys custom dataflow rules to infer client types. Section 9 presents our empirical evaluation of the strengths and limitations of this layered approach.

8 AWS Best Practices Rules

In this section, we describe a representative sample of eight rules that detect different types of defects related to usage of the Boto3 API. These rules cover approximately 200 public-facing AWS services. All Python AWS best practices rules (as well as most other CodeGuru rules) are implemented atop GQL (see Section 6), and follow the same rule evaluation mechanism that is discussed in Figure 4. Of the eight rules discussed in this section, we focus in particular on two rules — concerning pagination and batchable APIs —
def sync_ddb_table(source_ddb, destination_ddb):
    response = source_ddb.scan(TableName="table1")
    for item in response['Items']:
        destination_ddb.put_item(TableName="table2", Item=item)

Figure 8 Non-compliant Pagination Example

def sync_ddb_table(source_ddb, destination_ddb):
    response = source_ddb.scan(TableName="table1")
    for item in response['Items']:
        destination_ddb.put_item(TableName="table2", Item=item)
    # Keeps scanning until LastEvaluatedKey is null
    while "LastEvaluatedKey" in response:
        response = source_ddb.scan(TableName="table1",
                                   ExclusiveStartKey=response['LastEvaluatedKey'])
        for item in response['Items']:
            destination_ddb.put_item(TableName="table2", Item=item)

Figure 9 Correct Pagination Example

to enable thorough discussion of rule syntax and sample detections.

Worthy of mention is our ability, thanks to the AWS best practices rules and their
detections, to form an effective collaboration between the CodeGuru and AWS SDK teams.
From our side, the collaboration consists of frequent feedback to the SDK team (either
conveying developer feedback or trends that we observe across multiple detections). From the
AWS SDK team's side, our rules and detection technologies pose as a platform to promote
awareness of new features, for example the SDK V2 pagination feature.

8.1 Detecting Misuse of Paginated APIs

The pagination trait is implemented by over 1,000 APIs belonging to >150 AWS services.
This trait is commonly used when the result set due to a query is too large to fit within
a single response. For the complete set of results, a pagination token is used to perform
iterative requests and receive the response in parts. Developers who are not aware of this
trait might mistakenly suffice with a single request/response result, as illustrated in Figure 8.

Here the scan call is used to read items from an Amazon DynamoDB table, where put item
saves those items to another DynamoDB table. The scan API implements the pagination trait.
However, the code neglects to check for additional results beyond the initial batch, which
is clearly wrong. Our pagination rule detects the missing pagination in this example, and
generates a recommendation to iterate on the complete result set through the LastEvaluatedKey
token available through response. A compliant version of the code, consistent with this
recommendation is shown in Figure 9.

8.2 Error Handling for Batch Operations

More than 20 AWS services expose batch APIs, which enable bulk request processing. Batch
operations can succeed without throwing an exception even if processing fails for some items.
Therefore, a recommended best practice is to explicitly check for failures in the response due
to the batch API call. We illustrate incorrect and correct usages of batch APIs in Figures 10
and 11, respectively.

The rule for detection of batch operations where failures are not checked is shown in
def noncompliant():
    sqs = boto3.client('sqs', 'us-west-2')
    sqs.send_message_batch()

Figure 10 Incorrect Error handling for Batch Operation example

def compliant():
    sqs = boto3.client('sqs', 'us-west-2')
    response = sqs.send_message_batch()
    if "Failed" in response:
        raise SendMessageToSQSFailure("Failed")

Figure 11 Correct Error handling for Batch Operation example

Figure 12. Like many other CodeGuru rules, in particular in the AWS best practices category, this rule is parameterized by a configuration. (See Section 9 for an example.) The rule’s precondition searches for batch API calls per the configuration, then transforms from the calls to their respective receivers, which are stored into variable AWS_CLIENT. Backward propagation, in an attempt to relate these receiver nodes to applicable Boto3 services, then takes place through the getBoto3 call.

The postcondition loads the batch API call, stored as variable BATCH_API_CALL, then checks whether the result of the call is ignored through withOutputIgnoredFilter. This filter checks whether the call node(s) flowing into it define(s) a node that has no outgoing edges.

8.3 Other Representative Rules

We now switch to additional rules in the AWS best practices category, and provide an explanation of what they each check for.

Use waiters in place of polling API: Waiters are utility methods that make it easy to wait for a resource to transition into a desired state by abstracting out the polling logic into a simple API call. The waiters interface provides a custom delay strategy to control the sleep time between retries, as well as a custom condition on whether polling of a resource should be retried. Our rules detect code that appears to be waiting for a resource before it runs. In such cases, it recommends using the waiters feature to help improve efficiency.
Detect missing `None` check on cached response metadata: Response metadata represents additional information included with a response from AWS. Response metadata varies by service, but all services return an AWS request ID that can be used in the event a service call isn’t working as expected. If the code attempts to access the response metadata, `ResponseMetadata`, without performing a `None` check on the response object, then this might cause a `NoneType` error. To prevent this, our rule recommends adding a `None` check on the response object before accessing the response metadata.

Detect failed records in Kinesis `PutRecords`: The `put_records` operation in AWS Kinesis service might fail, thereby causing loss of records. This rule detects if the code handles the failed records from the `put_records` operation. In the absence of such handling of failed records, the rule recommends checking the `FailedRecordCount` in the `put_records` response to see if there are failed records in the response. A failed record includes `ErrorCode` and `ErrorMessage` values. If failed records are found, the rule recommends adding them into the next request.

Detect deprecated APIs: This rule detects usage of deprecated APIs in Python application code. A total of 107 deprecated API specifications are extracted from Boto3, identified from the use of `deprecated` trait in the API models. These API specifications are fed into the rule for detecting deprecated APIs in real world Python code.

Detect inefficient/redundant API chains: The rule for inefficient/redundant API chains detects usage of less performant APIs or outdated APIs, an API call chain that could be replaced with a single API call, a manual pagination operation where the SDK provide a Paginator API to automatically perform the pagination, and much more.

Detect expensive client object construction in Lambda handler: This rule detects a Boto3 client that is initialized from a Lambda handler. In order to speed up Boto3 client initialization and minimize the operational cost of the Lambda function, the rule recommends creating the client at the level of the module that contains the handler, and then reusing it between invocations. This is stated in the best practices for the lambda handler. [5]

9 Experimental Results

In this section, we report on experiments to validate our approach for on-demand resolution of Python types. Our experiments are guided by the following research hypotheses:

Hypothesis 1: Skipping type inference, instead relying solely on function names and arguments, is insufficient since that might lead to excessively many false positive detections.

Hypothesis 2: The dataflow-based and Pyright-with-stub-based resolution strategies have complementary strengths.

Hypothesis 3: A staged approach that combines dataflow and stubs with name-based resolution as a low-confidence fallback is effective.

Hypothesis 4: The AWS best practices rules, running atop the staged algorithm, are sufficiently precise, efficient and actionable to provide value during code review.

We note that beyond type inference, once a function call is confirmed to invoke a given AWS service, most of the rules are straightforward and do not require complex and/or interprocedural analysis to detect incorrect or suboptimal use of the AWS API. There are few exceptions, where the actual rule’s logic can be imprecise, but overall the correctness of type inference is a good proxy for the correctness of a rule finding.

We illustrate rule dependence on identification of the Boto3 service being invoked using the JSON snippet below, taken from our service’s production configuration. The “Missing Pagination” rule, whose specification is described in the snippet, searches for paginated
functions like `list_dataset_groups` in the specific context of the `forecast` AWS service. Recall that these API specifications are automatically extracted from the API models in Boto3.

```
{
  "expectedPaginationMethods": ["IsTruncated", "NextToken"],
  "paginatedMethod": "list_dataset_groups",
  "resultKeys": ["DatasetGroups"],
  "serviceId": "forecast"
}
```

We have evaluated the strategies described in Section 7 using a dataset consisting of 3,027 public GitHub repositories. These repositories were selected based on the following criteria: (1) The repository contains Python source files (at least 3, and with a total of at least 100 lines of code). (2) The repository has an MIT or Apache license. (3) The repository has a rating of 3 stars or more. (4) The repository makes use of the AWS SDK.

### 9.1 Performance of Resolution Strategies in Isolation

To examine the first two hypotheses laid out above, we begin by computing precision and recall for the different type resolution strategies in isolation. Precision is measured as the proportion of correct (TP) versus incorrect (FP) type resolutions, and recall is measured as the proportion of correct (TP) versus missed (FN) type resolutions. In what follows, we use the notation $t[s]$ to refer to the type of SDK service client $s$.

#### 9.1.1 Type-Resolution Strategies

We consider 3 different strategies for resolution of $t[s]$:

- **Strategy 1**: Use Pyright’s type inference in conjunction with third-party Boto3 type stubs. This strategy potentially recover types beyond the boundaries of a single function.
- **Strategy 2**: Use interprocedural dataflow analysis, combining backward and forward queries.
- **Strategy 3**: Match against the API name without attempting to resolve the type of the receiver, which is an over-approximate yet cheap approach.

#### 9.1.2 Results

Table 1 shows the number of resolutions due to each of the strategies when applied to the GitHub dataset. To gain qualitative insight into the results, and how many of the type resolutions are accurate, we manually reviewed 50 Boto3 client detections, selected at random, for each of the three strategies for a total of 150 detections. Reviewers consisted of five senior engineers and scientists, all expert users of the Boto3 library.

Our qualitative analysis suggests that strategies 1 and 2 are highly precise, as reported in the “Precision” column of Table 1. All 50 cases sampled for manual review were judged as correct. By contrast, for strategy 3, only 54% of the samples (27 out of 50) were correct. By definition, strategy 3 achieves 100% recall and thus establishes an upper bound on the number of false negatives due to strategies 1 and 2.

The set of detections obtained from strategy 1 and strategy 2 are not exactly the same, and they do not subsume each other: some strategy-1 detections are omitted by strategy 2, and vice versa. Out of 27 true positive detections from strategy 3, 19 detections are also
obtained from strategy 1 and strategy 2 combined. The remaining 8 detections (30%) are exclusive to strategy 3.

9.1.3 Discussion

We consider the pros and cons of the three strategies in light of these results.

Strategy 1 uses third-party Boto3 type stubs, together with Pyright’s type inference to resolve AWS SDK clients. Unlike strategy 2, where type resolution occurs during rule evaluation, strategy 1’s Pyright-derived types are available before rule evaluation, during MU graph construction. This allows type resolution to run once rather than on every application of every rule: a major performance boost.

On the negative side, strategy 1 suffers from low recall, as shown in the “Type Resolution Count” column of Table 1. This is due to the different ways in which AWS SDK clients are obtained, and in particular, the common case of passing them as function parameters. Pyright does not search for callers of the function, thus assigning Any as the type of the parameters unless annotations are explicitly provided.

Moving to strategy 2, the ability to perform backward dataflow tracking addresses the challenge of passing AWS SDK clients as function parameters. Duplication of work on type resolution is mitigated by a staged algorithm that first attempts intraprocedural resolution, then performs tracking at the file level, and finally at the level of the entire codebase. From our experience, and performance measurements, the staged algorithm is quite effective. Like strategy 1, strategy 2 retains full precision, yet has much higher recall as shown in the “Type Resolution Count” column of Table 1.

In spite of its overall effectiveness, strategy 2 — which tracks dataflow through local variables — can miss cases where the client is stored as a field or global variable. These cases are handled by strategy 1.

Our analysis of the gaps between strategies 1 and 2 is confirmed experimentally. In line with hypothesis 2, we have found 60 detections that are exclusive to strategy 1 and 832 detections that are exclusive to strategy 2.

Finally, the low precision of strategy 3 (just over 50%) confirms hypothesis 1. At the same time, the computational cost of strategy 3 is virtually zero, and thanks to its simplicity, it is able to sometimes completely bypass complex tracking scenarios that are beyond the power of strategies 1 and 2. An example is given in Figure 13, where neither strategy 1 nor strategy 2 is able to recognize that self.ec2_client is a Boto3 client in the body of the ec2_client.describe_snapshots(**kwargs) method. Strategy 3 succeeds here simply by recognizing describe_snapshots as the name of an AWS SDK client API method.

To make use of strategy 3 in spite of its approximate nature, we “penalize” detections due to this strategy by assigning a confidence score of 0.5 to those detections compared to 1.0 if the detection is due to strategies 1 or 2, as shown in the “Confidence” column of Table 1. The exact value of 0.5 is arbitrary, but serves to distinguish the lower-confidence detections of strategy 3 from the higher-confidence detections of strategies 1 or 2. This is in

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Confidence</th>
<th>Description</th>
<th>Type Resolution Count</th>
<th>Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.0</td>
<td>Pyright with Boto3 type stubs</td>
<td>2,293</td>
<td>100%</td>
</tr>
<tr>
<td>2</td>
<td>1.0</td>
<td>Dataflow tracking</td>
<td>3,065</td>
<td>100%</td>
</tr>
<tr>
<td>3</td>
<td>0.5</td>
<td>API name based resolution</td>
<td>5,403</td>
<td>54%</td>
</tr>
</tbody>
</table>

Table 1 Number of type resolutions due to each of the resolution strategies.
class AwsClient(object):
    def __init__(self, *args, **kwargs):
        self._boto3client = None
        super(AwsClient, self).__init__(*args, **kwargs)

    def create_ec2_client(self, context=None):
        # → (method) create_ec2_client:
        # (self: Self@AwsClient, context=None) -> Any
        return boto3.client('ec2')

    def get_aws_client(self, context):
        if not self._boto3client:
            ec2_client = self.create_ec2_client(context)
            # → (variable) ec2_client: Any
            return self._boto3client

    def describe_snapshots(self, **kwargs):
        response = self._ec2_client.describe_snapshots(**kwargs)
        # → (variable) _ec2_client: Any

Figure 13 Detections from Strategy 3 that strategies 1 and 2 miss

line with our earlier comment that the correctness of type resolution is a good proxy for the correctness of a detection.

9.2 Performance of Combined Resolution Strategies

The results in Section 9.1.1 suggest that there is benefit in combining the different strategies in light of their complementary strengths. Starting from this motivation, we report here on experiments with “hybrid” resolution strategies, which we refer to as configurations.

9.2.1 Type Resolution Configurations

We consider two configurations: High Confidence (HC) runs strategy 1, then strategy 2 where needed to complement strategy 1. Mixed Confidence (MC) runs strategies 1 and 2 in the same fashion as HC, but rather than giving up if both fail, proceeds to strategy 3 in an attempt to generate a low-confidence detection.

CodeGuru uses the confidence score to rank the detections as per the “Confidence” column in Table 1. Detections from strategy 1 and strategy 2 rank higher than detections from strategy 3 thanks to their higher confidence score. CodeGuru imposes different restrictions and limitations on detectors, in particular with regard to the overall number of detections, which means that in the presence of sufficiently many high-confidence detections, low-confidence detections are suppressed. By implication, low-confidence MC detections are not always reported to the user.

9.2.2 Results

Table 2 reports results for both configurations, running against the dataset of 3,027 GitHub repositories. The total time for running each configuration is close to 5 hours.

In line with hypothesis 2, the HC configuration generates more detections than strategies 1 or 2 in isolation. The total number of detections due to the HC configuration is 60 more than strategy 2: exactly the number of detections that are exclusive to strategy 1.

Moving to the MC configuration, the number of detections that it generates is identical to strategy 3 in isolation, which is expected. The important difference, however, is that most
Table 2 Type Inference Configurations

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Strategies</th>
<th>Description</th>
<th>Number of Detections</th>
</tr>
</thead>
<tbody>
<tr>
<td>HC</td>
<td>1, 2</td>
<td>Pyright with stubs followed by dataflow</td>
<td>3,125</td>
</tr>
<tr>
<td>MC</td>
<td>1, 2, 3</td>
<td>All layers</td>
<td>5,403</td>
</tr>
</tbody>
</table>

(that is, 3,125) of the detections have high confidence, with only 2,278 detections relying on strategy 3.

Projecting from the detections we sampled and triaged, we estimate that the MC configuration has a precision score of 0.85 along with perfect recall, whereas the HC configuration has perfect precision but a recall score of roughly 0.72 (with the assumption that 54% of the findings found by MC but not HC are true positives). This analysis supports hypothesis 3, which favors use of strategy 3 as part of the combined strategy rather than relying only on the high-confidence strategies.

9.3 Real-world Feedback on the Rules

Beyond our offline study, we also report on data from the field driven by comments that CodeGuru has left on code reviews in production. CodeGuru posts comments on code reviews just as a human reviewer would. We have augmented the comment UI with a feedback menu, so that a developer can optionally rate a detection as “Useful”, “Not Useful” or “Not Sure” and/or provide free-form textual feedback. These feedback mechanisms give the CodeGuru team insight into the performance of different detectors and enable detector tuning over time.

For AWS best practices, each CodeGuru comment contains two key fields:

1. One or two paragraphs explain what the issue is, and why fixing it is important. For example, in the case of a batch operation whose output is ignored, the explanation states that even if some items are not processed successfully, the batch operation might still complete successfully without raising an exception.
2. A “Learn More” hyperlink directs the user to the appropriate section in the Boto3 online documentation for complete information on the API in question.

We provide lower-bound metrics to give a sense of the size of CodeGuru’s input funnel. In the studied time period of 10 weeks, CodeGuru analyzed \( \gg 1,000,000 \) lines of code. We applied \( \gg 10 \) detectors, yielding \( \gg 10,000 \) AWS best practice recommendations, which we reported to \( \gg 1,000 \) developers.

We note that by definition, the codebases involved in this study are all live (undergoing code reviews and modifications). These are Python cloud services and applications that make use of Boto3, where the developers are industry practitioners with Python and cloud background. Hence we assign high weight to their feedback on CodeGuru detections.

In CodeGuru, we measure acceptance as an indication of whether or not developers have found a given rule’s review comments useful. Given a set of “Useful” \( U \), “Not Useful” \( NU \) and “Not Sure” \( NS \) ratings, we compute acceptance as the ratio \( \frac{|U|}{|U| + |NU| + |NS|} \), where by \( |U| \) we mean the number of “Useful” feedback points, and analogously for \( NU \) and \( NS \). Note, importantly, that we conservatively treat “Not Sure” the same as “Not Useful”.

Table 3 shows the acceptance data for eight of the Python AWS best practices rules for a time period of 10 weeks. We obtained \( \gg 100 \) feedback points from a population of \( \gg 100 \) developers through the feedback UI described above. As reported in Table 3, developers...
accepted over 85% of the recommendations made by five out of the eight rules, and almost 83% of the overall recommendations.

Only one of the eight rules, “Use Waiters instead of Polling APIs”, has an acceptance rate below 75%. Our analysis of this rule’s performance, including communication with some of the developers who left feedback on its detections, suggests that the gap between acceptance and correctness is important. Developers often acknowledge the detection as correct, but push back for one or more of the following reasons: (1) The intent of the PR is different, and they prefer not to merge multiple unrelated changes into the same PR. (2) The change is applicable, but requires upgrading the codebase to use the latest AWS Python SDK, which again exceeds the scope of the PR. (3) The change is not applicable, since the code in question is test code or there is no concern about polling in the given context. It is worth adding that outside the time period reported here, we have seen multiple weeks where acceptance rate for “Use Waiters instead of Polling APIs” was high.

Overall, acceptance data from the field supports hypothesis 3 in showing that developers mostly find the detections by to the Python AWS best practices rules useful. These are made using the MC configuration, which integrates all three of the resolution strategies described in Section 9.1.1.

From our conversations with developers, the textual feedback they provided, and our own review of some of the detections and their corresponding feedback, we have identified two main factors that contribute to the usefulness of our rules: (1) Missed features: SDK changes across versions, in particular new features, are sometimes missed by developers. Pagination, retry and error handling are examples of such features, where developers not familiar with these built-in capabilities sometimes implement “manual” mechanisms instead. Another example is manual polling versus the recommended use of the waiter utility. (2) Missed expectations: Developers sometimes assume, rather than verify, the functionality of a given API or the role of a given parameter. An example is the QueryResponse::hasItems method, whose
(boolean) return value is sometimes incorrectly interpreted to mean that the response contains a non-empty collection of items, where what is in fact meant is that response defines an Items property. To make sure whether any items are contained in the response, the developer needs to also check Items::isEmpty. Mistakes like this can lead to large-scale operational failures.

Table 4 reports the breakdown, by confidence level (high versus low), for the detections in Table 3. In sharp contrast to the distribution due to strategy 3 from the offline study, where approximately 45% of the detections had a low confidence score, the hybrid inference strategy leans heavily towards high-confidence detections (88% of all detections). This is consistent with the suppression policy described above, in Section 9.2.1, for low-confidence detections. The tradeoff that the hybrid strategy offers in the presence of confidence-based suppression is appealing, in that low-confidence detections are typically shadowed by high-confidence detections, which limits the impact of such detections on precision and allows them to play an important role in pushing coverage upwards when high-confidence detections are absent. Also note, from Table 4, that the proportion of low-confidence detections among “Not Accepted” detections is higher compared to “Accepted” detections (16% versus 7%), which is consistent with the data from the offline study.

Overall, our analysis of detections from the field, and how these map back to the hybrid strategy, are in support of hypothesis 4. Developers tend to view our AWS best practices recommendations as useful. Most of the recommendations build on high-confidence type inference, with some remaining cases benefiting from the low-confidence resolution strategy.

## Conclusion and Future Work

We have presented an industrial-strength framework for precise static analysis of Python applications that use AWS cloud services. In support of this goal, we have developed a novel type inference system for identifying and tracking AWS service clients in real-world Python applications. Our Python MU graph IR is suitable for building a wide range of static analyses or best-practice rules for Python applications. Furthermore, the Guru Query Language provides the right level of abstraction with its encapsulation, optimization and reuse features to develop static analysis rules that can be evaluated at different scopes, from single functions to entire applications.

Experiments on 3,027 open-source Python GitHub repositories show that individual inference strategies have complementary strengths. The most effective solution, then, is a layered approach that combines Pyright with Boto3 stubs, custom dataflow analysis in GQL, and name-based resolution as a low-confidence fallback. Our layered strategy achieves 85% precision and 100% recall in typing relevant Boto3 values in Python client code. The ultimate authorities on the value of our approach are real-world developers, with no ties to the authors. Those developers accepted more than 85% of the recommendations made by five out of eight rules, and roughly 83% of the recommendations on average.

In the future, we plan to extend and generalize our type inference infrastructure to other rule suites and properties that apply to Python programs. We are also examining ways to reuse our work on Python on-demand type inference when adding support for other dynamically typed languages.


4 Amazon Web Services. AWS SDK for Python (Boto3) [online]. URL: https://aws.amazon.com/sdk-for-python/ [cited 2022-05-12].


14 Vlad Emelianov. mypy_boto3_builder: Type annotations builder for boto3 compatible with VSCode, PyCharm, Emacs, Sublime Text, pyright and mypy [online]. URL: https://vemel.github.io/mypy_boto3_builder/ [cited 2021-12-01].


