IAM Policy Autopilot: Static Analysis for Policy Generation from Application Code

Antonio Filieri✉
Amazon Web Services
USA
afilieri@amazon.com

Luke Kennedy
Amazon Web Services
USA
lukekenn@amazon.com

Kevin Luo
Amazon Web Services
USA
kevinluo@amazon.com

Matt Luttrell
Amazon Web Services
USA
matluttr@amazon.com

Adrián Palacios
Amazon Web Services
USA
accorell@amazon.com

Akash Panda
Amazon Web Services
USA
pandakas@amazon.com

Neha Rungta
Amazon Web Services
USA
rungta@amazon.com

Matthias Schlaipfer
Amazon Web Services
Germany
schlaipf@amazon.com

Karan Jit Singh
Amazon Web Services
USA
kjsing@amazon.com

Anvesh Tanuku
Amazon Web Services
USA
tanuka@amazon.com

Nick Winans
Amazon Web Services
USA
nwinans@amazon.com

Diana Yin
Amazon Web Services
USA
dianayin@amazon.com

Weiben Zhang
Amazon Web Services
USA
weibenz@amazon.com

Abstract

Every application deployed on Amazon Web Services (AWS) requires Identity and Access Management (IAM) policies that specify which API actions the application is authorized to perform. AWS offers managed policies as a convenience to simplify policy authoring. However, these policies are designed to cover common scenarios which can lead to unnecessarily broad permissions beyond what a specific use case requires. AI coding assistants offer an alternative, but can be unreliable: they hallucinate invalid policy content, and lag behind newly launched services and APIs. Moreover, some organizations distrust nondeterministic processes for security-critical artifacts such as access policies.

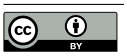
We present IAM Policy Autopilot (IPA), a deterministic static-analysis tool that generates IAM policies by examining API usage in Python, Java, Go, and TypeScript/JavaScript code. The tool occupies a middle ground between hand-crafted least-privilege policies, which require deep IAM expertise, and potentially overpermissive managed policy selection. It implements a three-phase pipeline that parses source files to identify AWS SDK calls via AST pattern matching; maps SDK operations to the required permissions by consulting authoritative, up-to-date service metadata; and synthesizes policy documents with provenance, tracing each permission to its origin in the source code. We evaluate IPA in two settings: (1) on 10 synthetic benchmark applications across four languages, comparing generated policies against minimal baseline policies, AI-generated policies, and an optimal selection of managed policies. IPA-generated policies prove sufficient to run 9 of the 10 applications; and (2) on a production-style multi-service chatbot application in Python, where IPA-generated policies prove sufficient for executing 11 of 12 handlers. Our findings show that policies generated by our tool are sufficient to run applications on a level similar to, or outperforming, AI-generated policies, while being generated deterministically. On average, the policies permit about an order of magnitude fewer permissions than the optimal selection of managed policies and about 60% fewer permissions than an expert developer-authored set of policies, indicating that IAM Policy Autopilot provides a good starting point when writing application policies.

CCS Concepts

• **Security and privacy** → *Access control*; *Usability in security and privacy*; • **Software and its engineering** → **Automated static analysis**.

Keywords

IAM policies, static analysis, access control, AWS, policy generation



ASE '26, Munich, Germany

ACM ISBN 979-8-4007-2882-2/2026/10
https://doi.org/10.1145/3832783.3834512

ACM Reference Format:
Antonio Filieri, Luke Kennedy, Kevin Luo, Matt Luttrell, Adrián Palacios, Akash Panda, Neha Rungta, Matthias Schlaipfer, Karan Jit Singh, Anvesh Tanuku, Nick Winans, Diana Yin, and Weiben Zhang. 2026. IAM Policy Autopilot: Static Analysis for Policy Generation from Application Code. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26), October 12–16, 2026, Munich, Germany*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3832783.3834512

1 Introduction

Every application deployed on Amazon Web Services (AWS) requires Identity and Access Management (IAM) policies that specify which API actions the application is authorized to perform. Every API request is evaluated against the applicable set of policies, and IAM denies any request for which no policy grants explicit permission. This deny-by-default, fine-grained model is a deliberate security design: it enables customers to build applications that follow the principle of least privilege [35], granting each workload only the access it needs and nothing more.

Realizing this benefit in practice, however, is challenging. Writing a correct policy requires knowing the exact IAM action name for each SDK call, cross-service dependencies, the Amazon Resource Name (ARN) format of every resource the action applies to, and any conditions that scope the permission. AWS currently exposes over 20 000 distinct IAM actions across more than 400 services, and the catalog grows weekly as new services and operations are added.

AWS provides *managed policies* (1473 at time of writing) to help developers start out. These are pre-built policy documents with high-level descriptions such as "read-only access to AWS Simple Storage Service (S3)" intended to provide permissions for many common use cases. AWS's guidance is to start with managed policies and gradually move toward least privilege [11], acknowledging that managed policies are typically overpermissive. Selecting the optimal set of managed policies in itself causes friction, which developers short-circuit by attaching overly permissive policies—e.g., `AdministratorAccess`, or statements with `Action: "*"` and `Resource: "*"`—deferring refinement indefinitely. This creates long-lived security debt that undermines the very guarantees IAM was designed to provide. Measurement studies of real-world applications find such overprivilege to be systemic [33, 38]. The problem compounds at organizational scale: large companies may operate thousands of application roles, each requiring a distinct policy tailored to its code's SDK calls.

AI coding assistants can generate application code effectively, but can be unreliable to produce IAM policies when asked to do so. We observe three recurring failure modes leading to denied requests. First, LLMs fail to include permissions that are required for an application to execute. Second, LLMs generate overly specific content, such as conditions or precise resource names, that is too restrictive. Third, LLMs *hallucinate* policy content—for example, nonexistent condition keys like `redshift-data:DbName`. Moreover, LLMs suffer from *stale knowledge*: AWS added 24 new services and 1760 new API operations in 2025 alone. Even if its reasoning were perfect, an LLM cannot know about services and APIs released after its training data cutoff. Beyond these issues, LLM-generated policies are nondeterministic: the same prompt can yield different policies across runs, with no guarantee that any individual output is correct or complete. Some organizations categorically refuse to let a probabilistic model author their access policies, viewing determinism and auditability as prerequisites for security artifacts.

Existing tools address different parts of the problem but leave a gap at the pre-deployment stage. IAM Access Analyzer [8, 23] refines policies using application logs *after* deployment, based on observed API calls—this requires deploying with permissive policies first and then narrowing them. Policy linters such as Parliament [25] validate existing policies for syntax errors and known anti-patterns, but they do not generate policies from code. Infrastructure-as-Code (IaC) scanners such as cfn-lint [2] and Checkov [19] check deployment templates for misconfigurations, but they analyze infrastructure definitions, not application source code. The closest prior work is `iamfast` [29], which also performs deterministic static analysis of application source code to synthesize least-privilege IAM policies before deployment, with support for Python, Java, Go, and JavaScript. `iamfast` extracts resource identifiers from source code and maps SDK calls to IAM actions; however, its scope is limited to direct API-to-action mappings and does not extend to cross-service dependencies. As we show in Sec. 5, these limitations lead to incomplete policies for applications that rely on such dependencies.

This paper presents *IAM Policy Autopilot* (IPA) [17], a static analysis tool that generates baseline IAM identity-based policies from application source code. Hand-crafted least-privilege policies remain the gold standard, but writing them demands deep IAM expertise and is impractical as a starting point for most developers. IPA bridges this gap: it generates a baseline policy that is more precise than the overpermissive managed policies developers typically fall back on, and more complete and trustworthy than LLM-generated alternatives. The generated policy is a practical starting point toward a true least-privilege policy, making the refinement process less labor-intensive than authoring one from scratch.

The tool implements a three-phase pipeline:

(1) **Extract.** Language-specific frontends parse Python, Java, Go, and TypeScript/JavaScript source files using `ast-grep`-based AST pattern matching. They identify AWS SDK calls, extract metadata, and match method names against AWS service API models to produce a unified intermediate representation.
(2) **Enrich.** A language-agnostic enrichment engine maps SDK operations to IAM actions with their resources and ARN patterns through the AWS Service Authorization Reference [12], and expands cross-service dependencies via a fixed-point algorithm.
(3) **Generate.** A policy synthesis engine constructs IAM policy JSON documents from the enriched action set, performing statement merging with resource subsumption analysis and size-aware splitting to stay within the policy size limit.

The entire pipeline is deterministic: the same source code always produces the same policy. Every action in the generated policy carries a provenance explanation tracing it back to a specific SDK call in the source code or to a cross-service dependency chain, supporting auditability and policy review.

We make the following contributions:

- The three-phase static analysis pipeline described above covering four languages and handling SDK-call matching, cross-service dependency expansion, and policy synthesis (Sec. 4).
- An open-source tool [17] with both a command-line interface and a Model Context Protocol (MCP) [14] server, enabling standalone

```
{ "Version": "2012-10-17", "Statement": [
  { "Effect": "Allow",
    "Action": ["s3-object-lambda:PutObject", "s3:PutObject", "s3:PutObjectAcl",
      "s3:PutObjectLegalHold", "s3:PutObjectRetention", "s3:PutObjectTagging"],
    "Resource": ["arn:aws:s3:*:*:accesspoint/*/object/*", "arn:aws:s3:::*/*"]
  },
  { "Effect": "Allow",
    "Action": ["kms:GenerateDataKey"],
    "Resource": "arn:aws:kms:*:*:key/*",
    "Condition": {"StringLike": {"kms:ViaService": "s3.*.amazonaws.com"}}
  }
]}
```

Listing 1: Example IAM identity-based policy for applications uploading objects to S3 with KMS server-side encryption.

use as well as integration with AI coding assistants that can invoke it as a tool and incorporate its deterministic output into their workflows (Sec. 3).

- An evaluation on 10 synthetic benchmark applications across four languages, comparing generated policies against minimal policies, LLM-generated policies, `iamfast`, and an optimal selection of AWS managed policies; and on a production-grade chatbot application in Python (Sec. 5).

Paper organization. Sec. 2 provides background. Sec. 3 presents architecture and design principles, and walks through a running example. Sec. 4 describes the static analysis pipeline. Sec. 5 evaluates the tool. Sec. 6 discusses related work, and Sec. 7 concludes.

2 Background

This section introduces the concepts needed to follow the rest of the paper: IAM policies, AWS SDK calling conventions across languages, and Forward Access Sessions.

2.1 IAM Authorization and Policies

IAM [9] controls authorization for every API call to AWS services. Permissions are expressed through *policies*—JSON documents containing *statements*, each of which specifies an `Effect` (`Allow` or `Deny`), a set of *actions*, a set of *resources*, and optional *conditions*. AWS supports several policy types: *identity-based policies* attached to principals (users, roles, or groups), *resource-based policies* attached to resources such as S3 buckets, and organization-level controls such as *Service Control Policies* (SCPs) and *Resource Control Policies* (RCPs). In this paper we focus on identity-based policies, the primary mechanism for granting application-level permissions.

When a principal makes an API request, IAM evaluates all applicable policies and either denies or allows the request. The default posture is deny: before an application can perform any AWS operation, a developer *must* grant the required permissions in a policy.

The resources in each statement are identified by Amazon Resource Names (ARNs), which follow a structured format: `arn:${Partition}:${Service}:${Region}:${Account}:${Resource}`. Wildcards (`*`) may appear in any field. For example, `arn:aws:s3:::*/*` matches any object in any S3 bucket in the `aws` partition. IAM policies are size-limited (the size depends on policy type) which constrains a policy's verbosity. However, multiple policies can be attached to an entity. Listing 1 shows a concrete identity-based policy that allows an application to upload objects to S3 with AWS Key Management Service (KMS) server-side encryption. It contains two statements: one granting the actions required by the S3 `PutObject` operation, and one granting the KMS action required because S3 invokes KMS on the caller's behalf, scoped by a `kms:ViaService` condition.

The principle of least privilege [35] prescribes that each principal should receive only the permissions it needs. In practice, this means each application role should have a custom policy tailored to the specific SDK calls in its code.

2.2 AWS Managed policies

AWS provides pre-built *managed policies* that bundle sets of permissions under descriptive names like `AdministratorAccess`, `AmazonS3ReadOnlyAccess`, or `AWSLambda_FullAccess`. At time of writing, AWS offers 1473 managed policies, most of them service-specific. As the AWS documentation notes, "managed policies make it convenient [...] to assign appropriate permissions to users, groups, and roles. It is faster than writing the policies yourself, and includes permissions for many common use cases" [10]. However, this convenience comes at the cost of granularity: managed policies may include actions the application never uses. Users do not always downscope managed policies, and even when they do, the wildcards these policies use—for size and forward-compatibility—do not enumerate the individual action names needed to construct a least-privilege policy.

2.3 AWS SDKs

AWS provides official SDKs for Python (boto3), Java (AWS SDK v2), Go (AWS SDK v2), and JavaScript (AWS SDK v3), among others. All SDKs are generated from machine-readable *service models*. The Python SDK is generated from *botocore* models [1]—JSON files that describe every API operation a service exposes, including the operation name, its input and output shapes, as well as paginator and waiter definitions. The Java, Go, and JavaScript SDKs are generated from *Smithy* models [4, 36]—an interface definition language that captures the same information. Both model families are authoritative: they are the single source of truth from which SDK code, documentation, and CLI commands are derived. IPA ingests these models at build time and uses them for service matching and for resolving higher-level abstractions (paginators, waiters, resource APIs) to their underlying operations.

Each SDK exposes the same underlying API operations, but with different naming conventions and calling patterns. Fig. 1 illustrates these differences for the S3 `PutObject` operation. Despite the surface-level differences, every SDK call encodes the same two pieces of information that a static analyzer needs: the *target service* and the *operation name*. Recovering them reliably—including handling operation-name clashes across services and language-specific naming translations—is the subject of Sec. 4.1.

2.4 Forward Access Sessions

Some AWS operations trigger additional API calls to other services on the caller's behalf. For example, when an application calls `s3:PutObject` on a bucket configured with KMS server-side encryption, S3 invokes `kms:GenerateDataKey` to generate a data encryption key before storing the object. The caller's IAM policy must permit this KMS operation, even though the application code contains no KMS SDK calls.

SDK	Example

Python	`import boto3`
	`client = boto3.client("s3")`
	`client.put_object(Bucket="b", Key="k", Body=data)`

Java v2	`import software.amazon.awssdk.services.s3.*;`
	`import software.amazon.awssdk.services.s3.model.*;`
	`S3Client client = S3Client.builder().build();`
	`client.putObject(PutObjectRequest.builder()..., data);`

Go v2	`import "github.com/aws/aws-sdk-go-v2/service/s3"`
	`client := s3.NewFromConfig(cfg) // cfg from config.LoadDefaultConfig`
	`client.PutObject(ctx, &s3.PutObjectInput{Bucket:&b, Key:&k, Body:data})`

JS v3	`import { S3Client, PutObjectCommand } from "@aws-sdk/client-s3"`
	`const client = new S3Client({})`
	`client.send(new PutObjectCommand({Bucket:"b", Key:"k", Body:data}))`

Figure 1: Calling conventions across the four supported SDKs.

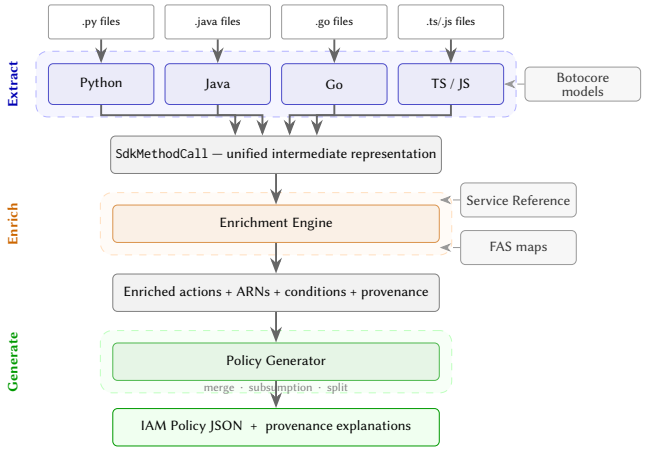


Figure 2: Architecture of IAM Policy Autopilot. Language-specific frontends converge on a shared intermediate representation; enrichment and generation are language-agnostic. Dashed arrows indicate data sources consulted.

These cross-service invocations are enabled by a technology called *Forward Access Sessions* (FAS) [13]. In order to only allow the intended call chain, a policy should use *conditions* that scope the permission to the calling service. For instance, `"StringEquals": {"kms:ViaService": "s3.us-east-1.amazonaws.com"}` ensures that a call is only allowed when it is initiated by S3.

FAS dependencies can be transitive: the caller needs permissions for all calls along a chain. Discovering the complete set of required permissions therefore requires a fixed-point computation, which we describe in Sec. 4.2.

3 Architecture Overview

IAM Policy Autopilot implements a three-phase pipeline: *Extract*, *Enrich*, and *Generate*. Fig. 2 shows the architecture. Language-specific frontends parse source code and produce a unified intermediate representation; a language-agnostic enrichment engine maps SDK operations to IAM actions and expands cross-service dependencies; and a policy generator synthesizes the final IAM policy documents. We first describe the design principles that guided the architecture, then walk through the running example end to end.

3.1 Design Principles

Deterministic analysis. IPA is deterministic, up to changes in the Service Reference model. Given the same source files and configuration, the tool always produces the same policy. This property is essential for security-conscious users. It enables integration into workflows requiring reproducibility and security auditing, where nondeterministic output would undermine trust.

Ease of use. IPA uses static analysis rather than dynamic analysis of execution traces. A dynamic approach either needs a deployed application (IAM Access Analyzer [8, 23] analyzes CloudTrail logs) or a test suite that exercises all code paths invoking the AWS SDK. Static analysis sidesteps both requirements: IPA needs only the source files, making it easy to run.

Freshness. AWS adds new services and actions regularly. IPA embeds authoritative sources such as the botocore SDK model at build time and queries an endpoint for the latest Service Authorization Reference data at runtime (Sec. 4.2)

Permission over-approximation. IPA aims to produce a *conservative baseline*: a policy that includes every IAM action the application could require at runtime. Resource scopes are over-approximated with wildcards; resolving resource identifiers (e.g., bucket names) statically is future work. The resulting policy is a starting point towards a least-privilege policy, less permissive than a selection of managed policies. The generated policies can be narrowed—by removing actions and replacing wildcards with concrete resource ARNs—but should not need actions added.

Explainability. Every action in the generated policy carries a provenance explanation. For actions derived directly from SDK calls, the explanation identifies the source file, line number, and method name. For operations invoked via FAS, it traces the dependency chain. The goal is to help refine the over-permissive baseline policy into a least-privilege policy. An auditor can verify *why* each permission is present rather than treating the policy as opaque.

3.2 Deployment Interfaces

IPA is packaged as a command-line tool and a Model Context Protocol (MCP) [14] server. The CLI accepts one or more source files and writes IAM policy JSON to stdout. The MCP server exposes a `generate_application_policies` tool that delegates to the same deterministic pipeline. This paper focuses on the static analysis pipeline itself; evaluating MCP integration within different agentic harnesses is left to future work.

3.3 Example Walkthrough

We use a running example throughout the paper: an AWS Lambda function that uploads files to S3 with KMS server-side encryption using a customer-managed key. Listing 2 shows the TypeScript source code; Listing 1 (Sec. 2) shows the IAM policy that IPA generates from it. The generated policy contains two statements: one granting `s3:PutObject` and related actions for the upload, and one granting `kms:GenerateDataKey`—required because S3 invokes KMS on the caller's behalf when server-side encryption is configured—scoped by a `kms:ViaService` condition. Nothing in the application code alludes to the KMS permission being required; IPA discovers it automatically through FAS expansion (Sec. 4.2). The following section describes each pipeline phase in detail.

```
// AWS Cloud Development Kit (CDK)
const bucket = new s3.Bucket(this, 'BucketName', {
  encryption: s3.BucketEncryption.KMS,
  encryptionKey: kmsKeyName, });
```

```
// Application code
const { S3Client, PutObjectCommand }
  = require('@aws-sdk/client-s3');
const s3 = new S3Client({ region: 'us-east-1' });

exports.handler = async (event) => {
  const { fileName, fileContent }
    = JSON.parse(event.body);
  await s3.send(new PutObjectCommand({
    Bucket: process.env.BUCKET_NAME,
    Key: fileName,
    Body: fileContent
  }));
  return { statusCode: 200,
    body: '{"message":"Upload successful"}' }; };
```

Listing 2: Running example: an AWS Lambda function that uploads files to S3. The bucket's KMS encryption is configured in CDK, but not visible in the application code.

4 Policy Generation Pipeline

The policy generation pipeline has three phases. Extraction (Sec. 4.1) is language-specific: each frontend parses source code and produces a unified intermediate representation. Enrichment (Sec. 4.2) and policy generation (Sec. 4.3) are language-agnostic.

4.1 Extraction

4.1.1 SDK Call Extraction. All language frontends use `ast-grep` [24], a structural code search library built on top of `tree-sitter` [20], for AST pattern matching. Each frontend defines AST patterns tailored to its language's SDK calling conventions. For example, the Python frontend matches the general pattern `$OBJ.$METHOD($$$ARGS)` and captures the receiver name, method name, and keyword arguments. The Go frontend matches method calls on service client objects and resolves the client's service from its import path. The Java frontend uses a single-pass design: all extraction patterns (imports, method calls, waiters, paginators) are combined into one ast-grep rule so the AST is scanned only once, with each match routed to the appropriate sub-extractor via a discriminator label. The TypeScript/JavaScript frontend matches the command pattern `new XxxCommand({...})` and extracts the command class name and its argument object.

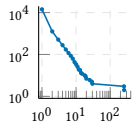
In some languages, the SDK call is syntactically separated from the object that describes it. For instance, in the AWS SDK for JavaScript v3, the developer first constructs a `Command` object (e.g., `new PutObjectCommand({...})`) and then passes it to `client.send()`. In this case, IPA triggers on the `Command` object creation rather than on the `send()` call, and assumes the command may be sent. This design choice may produce false positives if a `Command` object is constructed but never sent.

All frontends produce the same intermediate representation: an *SDK method call* record containing

- the method or command name as it appears in source code,
- the set of candidate (service, operation) pairs (populated during operation to service matching), and
- metadata: source file location, parameter names and values extracted from the call site, and receiver information.

This shared representation is the interface between language-specific extraction and the language-agnostic enrichment phase.

4.1.2 Service Matching. SDK method names are not globally unique: of the 14 283 unique operation names across all AWS services, 1257 (8.8%) are shared by two or more services. The plot shows the number of operation names (y-axis) shared by at least N services (x-axis) on a log–log scale; the most extreme cases—`TagResource` and `UntagResource`—each appear in 256 services. A naïve analysis that maps method names to services without disambiguation would produce policies with spurious permissions for unrelated services.



IPA builds a *service model index* at startup from the botocore SDK model (Sec. 2.3). This index maps each SDK method name to the list of (service, operation) pairs that share that name, and provides the input shape (expected parameter names and types) for each operation. The code generators also translate the canonical PascalCase operation names from the service model into language-idiomatic forms: snake_case for Python (`put_object`), camelCase for Java (`putObject`), and PascalCase with a `Command` suffix for TypeScript (`PutObjectCommand`). These translations are not purely mechanical—e.g., `ModifyDBCluster` becomes `modify_db_cluster` in Python (not `modify_d_b_cluster`).

The disambiguation strategy depends on the language:

- **Python: parameter validation.** For each candidate service, the disambiguator checks whether the keyword arguments at the call site are consistent with that service's input shape. A candidate is rejected if the call provides a keyword argument that does not exist in the operation's input shape, or if the call omits a required parameter (unless dictionary unpacking `**kwargs` makes the argument list open-ended). For example, `client.put_object(Bucket=...,Key=...)`: `PutObject` is defined by both S3 and AWS Elemental MediaStore, but the parameters `Bucket` and `Key` exist only in S3's input shape (MediaStore expects `Path` instead), so the disambiguator narrows the candidates to S3.
- **Java: receiver type resolution.** The Java SDK uses typed client objects (e.g., `S3Client`). The Java frontend resolves the declared type of the receiver variable by walking the AST scope hierarchy—checking local variables, method parameters, `try-with-resources` bindings, and class fields—and uses the type name to identify the service. When the frontend cannot resolve the type (e.g., `var` declarations, since we do not perform type inference), it falls back to import matching: it checks which service client classes are imported and intersects them with the candidate set.
- **Go: import path resolution and field name validation.** The Go SDK places each service in a separate package (e.g., `github.com/aws/aws-sdk-go-v2/service/s3`), so the disambiguator resolves the service by matching the import declarations in the source file against the candidate set, with import aliases resolved to their original package paths. Additionally, when struct literals are passed inline as arguments, IPA validates the provided field names against the expected input shape from the service model to filter out false positives.
- **TypeScript/JavaScript: command class name.** The JS SDK organizes each service into a separate package (e.g., `@aws-sdk/client-s3`), and each operation has a dedicated command class (e.g., `PutObjectCommand`). Disambiguation is immediate given the import; aliased imports are resolved if necessary.

As a fallback for cases where false positives remain, users can provide *service hints* (e.g., `--service-hints s3`) to restrict the candidate set. After disambiguation, any method call with no remaining valid service is discarded—it was a non-AWS call that happened to share a name with an SDK operation.

4.1.3 Higher-Level SDK Abstractions. Beyond direct API calls, AWS SDKs provide higher-level abstractions—paginators, waiters, resource APIs, and other utilities—that wrap underlying operations. The extraction phase normalizes these abstractions so that the enrichment phase sees only (service, operation) pairs, regardless of how the developer invoked the SDK.

Paginators wrap API operations that return results in successive pages, iterating the underlying call until all pages are retrieved. For example, in Python, `client.get_paginator('list_objects_v2')` creates a paginator for `ListObjectsV2`; the extractor parses the paginator name argument and resolves it to the underlying operation using the botocore paginator model. Dedicated sub-extractors handle Go and Java paginators by matching paginator type constructors.

Waiters poll an API operation until a condition is met. For example, in Python, `client.get_waiter('bucket_exists')` wraps a `HeadBucket` call; the extractor resolves waiter names to their underlying operations via the botocore waiter model. In Java, the frontend detects `waiter().waitUntilXxx()` call chains and maps the waiter method name to the polled operation.

Higher-level resource APIs (Python-specific) provide object-oriented access to AWS resources. For example, `s3.Object('bucket','key').get()` is resolved to the underlying `GetObject` client call using an embedded boto3 resource model.

Each SDK also ships *higher-level utility libraries* that map a convenience method to one or more underlying operations. For example, an S3 `upload` call may fan out to `PutObject`, `CreateMultipartUpload`, `UploadPart`, and `CompleteMultipartUpload`. Each frontend ships a *utility model*, currently written by developers, that maps these convenience methods to operations. Using IPA itself to generate them is future work.

4.2 Enrichment

The enrichment phase is language-agnostic: it takes the list of SDK method call records produced by any frontend and maps them to IAM actions, resource types, and resolves potential cross-service dependencies. The mapping is one-to-many: a single SDK operation can require multiple actions (for example, some operations require both a primary action and a `Describe*` action).

We use the AWS Service Authorization Reference [12] as authoritative mapping. For each (service, operation) pair, the enrichment engine retrieves the list of required actions, associated resource types, and ARN templates for those resources.

Algorithm 1 summarizes the two-phase enrichment procedure. In Phase 1, the worklist W is initialized with all (service, operation) pairs extracted from the source code; the dependency map D, which doubles as a visited set and provenance record, is seeded with the same pairs. For each pair, the engine consults the FAS map (line 5). Each FAS entry yields a target service, operation, and a set of conditions C—e.g., `"kms:ViaService":"s3.${region}.amazonaws.com"`—that scope the permission to the calling service. The

Algorithm 1 FAS expansion and operation to action mapping

Input: Extracted operations $O = \{(s_1, op_1), \ldots, (s_n, op_n)\}$
Output: Enriched actions A; dependency map D (provenance)

▷ Phase 1: FAS expansion
1: $W \leftarrow O$ ▷ worklist
2: $D \leftarrow \{(o, \emptyset) \mapsto \emptyset \mid o \in O\}$ ▷ dependency map
3: **while** $W \neq \emptyset$ **do**
4: $(s, op) \leftarrow \textsc{Pop}(W)$
5: **for all** $(s', op', C) \in \textsc{FasLookup}(s, op)$ **do**
6: **if** $(s', op', C) \in D$ **then**
7: $D(s', op', C) \leftarrow D(s', op', C) \cup \{(s, op)\}$
8: **else**
9: $D(s', op', C) \leftarrow \{(s, op)\}$
10: $W \leftarrow W \cup \{(s', op')\}$
11: **end if**
12: **end for**
13: **end while**
14: $A \leftarrow \emptyset$ ▷ Phase 2: action mapping
15: **for all** $(s, op, C) \in D$ **do**
16: $A \leftarrow A \cup \textsc{MapToActions}(s, op, C)$
17: **end for**
18: **return** A, D

conditions are stored as part of the key in D, so the same operation with different conditions is tracked separately. Operations already in D only gain an additional parent edge (line 7); newly discovered operations are added to both D and the worklist (lines 9–10). This ensures termination while recording all derivation paths.

In Phase 2, the engine maps every discovered operation to actions by querying the Service Authorization Reference [12] endpoint, passing along the associated conditions C (line 16). The returned dependency map D serves as provenance: for each action, D records the parent operations that led to its discovery. During policy generation, the FAS conditions may be further combined with action-level conditions from the Service Authorization Reference.

In the running example, Phase 1 starts with `(s3, PutObject)` in the worklist. The FAS map yields one dependency: `(kms,GenerateDataKey)` with a `ViaService` condition. This is added to the worklist; the next iteration finds no further FAS dependencies for the KMS operation, and Phase 1 terminates. Phase 2 then maps both operations to their respective actions, attaching the `ViaService` condition to the KMS action.

The FAS maps are currently compiled manually by experts, but collection is amenable to automation. They currently cover the most common cross-service dependency patterns but are not exhaustive; missing entries are a known limitation discussed in Sec. 5.3.

4.2.1 Resource and ARN Resolution. Each action is associated with resource types from the Service Authorization Reference. Each resource type has an ARN template with placeholders—for example, `arn:${Partition}:s3:::${BucketName}/${ObjectName}` for S3 objects. The enrichment engine resolves these templates (Sec. 2.1) as follows. Infrastructure-level placeholders (`Partition`, `Region`, `Account`) are filled from user-provided configuration; `Partition` is derived automatically from the region using botocore's partition data. Resource-specific placeholders (`BucketName`, `TableName`, `KeyId`) are replaced with wildcards (`*`), since these values are typically determined at runtime. Improving static resolution of resource-specific placeholders is future work.

4.3 Policy Generation

The policy generation phase transforms the enriched action set into one or more IAM policy JSON documents.

Statement Construction and Merging. For each enriched action, the engine creates an initial `Allow` statement with the action name, the resolved ARN pattern, and any conditions. These individual statements are then merged to produce compact, readable policies. The merging strategy groups statements by *resource compatibility*. Statements with identical resource sets are merged by combining their action lists into a single statement. Statements whose resources are *incomparable*—for instance, an S3 object ARN and a DynamoDB table ARN—are kept separate by default (cross-service merging is disabled to preserve readability). When one statement's resources subsume another's, the statements are *not* merged: doing so would grant the more-specific statement's actions to the broader resource set, violating least privilege. Resource relationships are determined by treating them as regular expressions and checking language inclusion. For example, `"arn:aws:s3:::*/*"`, which matches any S3 object, would not be merged with `"arn:aws:s3:::*"`, which matches more broadly on the S3 bucket-level. Statements are merged only if they have identical condition sets.

Size-Aware Splitting. IAM customer-managed policies are limited to 6144 characters of non-whitespace content. The engine monitors the running size of the policy during merging and starts a new policy document when the limit is exceeded.

4.3.1 Provenance Explanations. Every action in the generated policy carries a provenance explanation: *direct* actions are traced to a specific SDK call (source file, line number, and method name); *FAS* actions are traced to the dependency chain that led to their inclusion. If the same action is required by multiple SDK calls, the explanations are merged. These explanations serve two purposes: first, they help developers understand the one-to-many mapping from operations to actions; second, they provide a means to identify operations that were erroneously included during static analysis.

5 Evaluation

We evaluate IAM Policy Autopilot in two complementary settings. Sec. 5.1 uses synthetic benchmarks—small, single-file scripts across four languages—to compare generated policies against minimal baselines, LLM-generated policies, and AWS managed policies. Sec. 5.2 evaluates IPA on an open-source production-style application deployed to AWS, comparing IPA-generated, LLM-generated, and developer-authored policies, and validating all three via live execution.

5.1 Synthetic Benchmarks

Experimental Setup. We generate policies for 10 benchmarks. Each benchmark is a small, AI-generated project that deploys infrastructure to AWS and then executes the application using that infrastructure—we create an execution role and grant it permissions by attaching the policy under evaluation. Each application is translated in the four languages supported by IPA, with equivalent API calls across all versions. We ensured that the applications fail on errors so that access denial errors are not hidden.

Baseline minimal policy. As a ground-truth baseline we derive a *minimal policy* for each application: a policy whose action set cannot be reduced further without preventing the application from executing successfully. We implement delta-debugging [39] to find this minimal set of actions. If IPA produces a working policy, we use it as the starting point for delta-debugging; otherwise, we construct a working policy and use that as the starting point for minimization. A candidate *passes* if the application completes without errors, and *fails* otherwise; actions whose removal causes the test to fail are retained in the minimal policy.

5.1.1 Policy Generation. We generate policies using the following approaches: (1) IAM Policy Autopilot, (2) a set of managed policies covering the actions required by the minimal policy, (3) `iamfast` [29], and (4) AI-generated policies using single-shot experiments.

Managed policy cover computation. We select a covering set of AWS managed policies whose union contains every action in the minimal policy. The selection is formulated as a weighted set-cover integer linear program and solved with HiGHS [28]: let $x_i \in \{0, 1\}$ indicate whether managed policy i is included, with cost c_i equal to the number of actions it grants; minimise $\sum_i c_i x_i$ subject to $\sum_{\{i \mid a \in coverage(i)\}} x_i \geq 1$ for each required action a. We only count the actions each managed policy unconditionally grants: an `Allow` statement contributes to coverage only if its action patterns cover, and its resource patterns subsume, those of the minimal policy and it contains no `Condition` block. Candidate policies are excluded if any `Deny` statement matches a required action. Policies with conditions could work in practice, potentially yielding a larger candidate set, but we conservatively do not count these conditional actions towards the overpermissioning factor.

LLM policy generation. We generate policies for the scripts using Claude Opus 4.6 [15] and repeat each experiment 5 times to account for nondeterminism. Each prompt begins with the following instruction, where we ask for a working policy—and not least-privilege, in order not to penalize the LLM by giving it a harder task:

> *Generate an identity-based AWS IAM Policy which allows me to execute this application. {resource_instruction} Application: {source_code}*

where *{resource_instruction}* is one of three strategies for guiding the model on how to handle resource ARNs in the generated policy:

- *bare*: omitted entirely (no guidance). This prompt resembles using the LLM off-the-shelf, and it leads to the LLM in some cases generating policies with placeholder variables leading to failure, e.g., `"arn:aws:s3:::${BucketName}/data/*"`,
- *wild*: the placeholders occasionally introduced in *bare* led us to refine the prompt: *"Fill in all placeholder variables; if you don't know what to put, use the wildcard *."*. This prompt sometimes leads to failure due to incorrect resource or condition blocks. The LLM sometimes ignores the suggestion to use `*` or makes other mistakes like choosing wrong context keys, even though the action was chosen correctly: e.g., `"StringEquals":{"kms:RequestAlias":"alias/repo-monitor-run003-3cbeaff7"}`.
- *Res**: the recurrent mistakes we observed with *wild* led us to focusing on the actions chosen by prompting the LLM more strictly to *"Use `"Resource": "*"`."*. Note that such a policy is more permissive than using an ARN as generated by IPA—even with `*` used in places—as it allows access across regions and

Table 1: Per-run results. Each cell is a 2×2 grid encoding four languages: Python Go / Java TS. LLM experiments are repeated 5 times. Color indicates success class: 5 = 5/5 (success), 1–4 = 1–4/5 (partial), 0 = 0/5 (fail). IPA and iamfast are deterministic, experiments are run once: ✓ = pass, ✗ = fail, ▢ = N/A.

		LLM			LLM$_{CTX}$			
#	IPA	bare	wild	Res*	bare	wild	Res*	iamf.
01	✗ ✗ / ✗ ✗	0 4 / 2 0	5 5 / 5 5	5 5 / 5 5	0 1 / 3 0	5 5 / 5 3	2 5 / 3 5	▢ ✗ / ✗ ▢
02	✓ ✓ / ✓ ✓	0 0 / 5 0	5 5 / 5 5	5 5 / 5 5	0 0 / 0 0	0 5 / 5 5	5 5 / 5 5	▢ ✗ / ✗ ▢
03	✓ ✓ / ✓ ✓	0 0 / 0 0	5 5 / 5 5	5 5 / 5 5	0 0 / 0 0	5 3 / 5 5	5 5 / 5 5	▢ ✗ / ✗ ▢
04	✓ ✓ / ✓ ✓	5 5 / 5 5	5 5 / 5 5	5 5 / 5 5	5 5 / 5 5	5 5 / 5 5	5 5 / 5 5	▢ ✓ / ✓ ▢
05	✓ ✓ / ✓ ✓	1 0 / 0 0	0 0 / 0 0	0 0 / 0 0	0 0 / 0 0	0 0 / 0 0	0 0 / 0 0	▢ ✗ / ✗ ▢
06	✓ ✓ / ✓ ✓	0 0 / 0 0	5 5 / 5 5	5 5 / 5 5	0 0 / 0 0	4 3 / 4 4	5 5 / 5 5	▢ ✗ / ✗ ▢
07	✓ ✓ / ✓ ✓	0 0 / 0 0	5 5 / 5 5	5 5 / 5 5	0 0 / 0 0	5 5 / 5 5	5 5 / 5 5	▢ ✗ / ✗ ▢
08	✓ ✓ / ✓ ✓	5 1 / 1 3	5 5 / 5 5	5 5 / 5 5	0 2 / 5 0	5 5 / 5 5	5 5 / 5 5	▢ ✗ / ✗ ▢
09	✓ ✓ / ✓ ✓	0 0 / 0 0	5 5 / 5 5	5 5 / 5 5	0 0 / 0 0	5 5 / 5 5	5 5 / 5 5	▢ ✗ / ✗ ▢
10	✓ ✓ / ✓ ✓	0 3 / 4 1	5 5 / 5 5	5 5 / 5 5	0 0 / 1 5	5 5 / 5 5	5 5 / 5 5	▢ ✗ / ✗ ▢
Pass	**90%**	**25%**	**90%**	**90%**	**19%**	**83%**	**88%**	**10%**

accounts. It is also harder to manually refine the policy into a more precise one as there is no resource block to start from.

We run these prompts in two settings which differ in the context supplied to the model:

- *LLM*: the prompt with only the application source code.
- LLM$_{CTX}$: same prompt, but the LLM's context window is additionally populated with surrounding project files (other source files, configuration, and documentation) not informative for the sake of policy generation, simulating an AI-agent scenario in which the agent includes files open in the IDE or retrieved via tool calls. For example, the project context for a Java benchmark comprises 613 lines: 454 `.java`, 94 `.ts`, 51 `.md`, and 14 `.json`. Each language has a context that we reuse across all benchmarks.

On average, a single LLM policy-generation call consumed 7129 input tokens (±4632) and 651 output tokens (±202). At list pricing, this corresponds to $0.0519 per call, at time of writing.

5.1.2 *Microbenchmark Results.*

Sufficiency and nondeterminism. Table 1 reports on the sufficiency of generated policies for each of the 10 benchmarks—a policy is sufficient if it allows the deployment and execution of the application without permission errors. IPA produced a sufficient policy for 9 out of 10 benchmarks, where the failure on `01` is due to a missing FAS mapping, which we will analyze in detail at the end of the section.

For `iamfast`, we report on the generated policies for the Go and Java versions of the applications. `iamfast` supports Javascript (not Typescript); for Python it returned an empty policy for all the benchmarks, so we excluded it. For Go and Java, `iamfast` produced a sufficient policy for only one of the 10 benchmarks. We also noticed that the generated policies are often empty for the other benchmarks; our conjecture is that its aim at generating least-privilege policies drives the tool to include only permissions it is certain are needed (Fig. 3 shows the high precision). Since its source code is not available, we cannot verify this. Furthermore, manually inspecting the generated policies, the tool does not include actions that may be required indirectly via FAS.

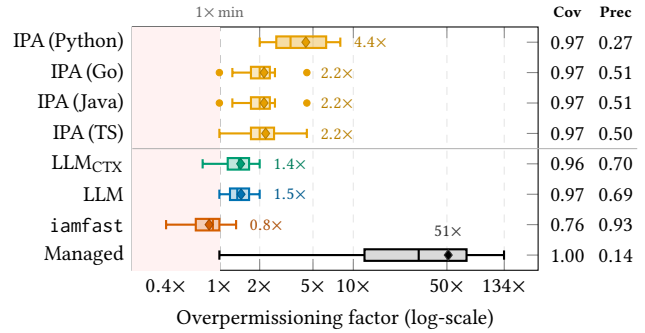


Figure 3: Overpermissioning factor (number of IAM actions in the generated policy divided by the number in the minimal policy) for 10 benchmarks, aggregated across prompts and settings. IPA results are shown per language. LLM-based methods and `iamfast` are aggregated across languages. The dashed line marks the minimal-policy baseline; values to the left indicate missing permissions. The vertical line inside each box is the median; the diamond (♦) marks the mean, whose value is annotated next to each box. The right panel shows coverage (fraction of required actions present) and precision (fraction of generated actions that are required).

The performance of the LLM experiments shows some variance across prompts, settings, and programming languages: Cumulatively, out of 6 configurations, 2 perform on-par and 4 perform worse than IPA. The variance highlights that LLMs can be hard to predict: for example, LLM$_{wild}$ succeeds, but LLM/Res∗ fails for `#01` in some cases, even though the difference in prompt should not affect the actions. Benchmark `#05` consistently fails due to a missing indirect permission (`kms:GenerateDataKey`), only one run adds it. We observe that adding irrelevant context makes results worse (LLM → LLM$_{CTX}$). The results confirm that while capable given the right prompt and context, LLM-based methods exhibit nondeterministic behavior, which may render them less reliable and unsuitable for workflows where deterministic generation is required. We also point out that the benchmarks do not use services or APIs launched after training cutoff (implied by the benchmarks being AI-generated themselves), which would penalize the LLM.

Overpermissioning. The controlled microbenchmarks also allow us to quantify by how much the generated policies were overpermissive in comparison to the reference minimal policies. We measure this by comparing the number of actions (with wildcards expanded) in statements with effect `Allow` (no policy contained `Deny`). We do not measure along the resource and condition dimensions, since both of these dimensions are unbounded, unlike the number of actions across AWS. Fig. 3 summarizes the overpermissioning factors. LLM-based methods generally produce smaller policies than IPA, with mean overpermissioning factors of approximately 1.45× compared to 2.2–4.4× for IPA. AWS managed policies overpermission

by a larger margin: the mean factor is 51×, roughly an order of magnitude more than the other approaches. Note that we construct the managed policy baseline by computing a minimal set cover from the reference minimal policy, which is the ideal solution; in reality a manual choice of managed policies is likely more permissive.

Analysis of benchmark `01`. In this benchmark IPA failed across all four languages (Table 1, first row). The root cause was a missing entry in our FAS dependency model for the AWS Redshift service: the benchmark calls the `(redshift-data,ExecuteStatement)` operation, which transitively calls `(redshift,GetClusterCredentials)`. Because the FAS model lacked this entry, IPA did not include the required permission.

We have since added the missing entry to our models, and the tool now generates a sufficient policy for `01`. However, this failure highlights a current limitation: the completeness of IPA's output depends on the completeness of its FAS maps. We are actively working towards automatically generating the FAS maps which will remove the reliance on manual curation by experts and ensure coverage is up-to-date as AWS adds new cross-service dependencies.

5.2 Real-World Application Evaluation

In this section we evaluate IPA on *aws-genai-llm-chatbot* [3], an open-source production-style chatbot in Python with retrieval-augmented generation (RAG) that uses a wide range of AWS services including Bedrock, DynamoDB, S3, OpenSearch, and others.

Experimental setup. The chatbot application comprises 18 Python entry points (Lambda handlers and Batch jobs) that share a common library (`genai_core`). Five handlers are excluded: two require SageMaker GPU instances for which we could not obtain capacity at this time, and three are schedule-triggered RSS ingestors that cannot be exercised on demand. Of the remaining 13, we evaluate 12; the last handler (`bedrock-agents-handler`) requires a containerized AgentCore runtime that we did not configure. The evaluated handlers range from 91 to 12 299 lines of code, span 1 to 109 source files, and take 0.4–24 seconds for IPA to analyze.

Baseline policy. For each handler, we extract the IAM policies from the deployed execution role as reference policies: inline policies authored by the application developers in AWS CDK with specific actions and resource ARNs, plus attached AWS managed policies expanded into concrete actions. We call these *authored policies*.

Import-aware source file selection. Handlers that use the shared `genai_core` layer require import-aware file selection to avoid analyzing library code that a given handler never reaches. Passing the entire shared library to IPA would generate permissions for every SDK call in `genai_core`, regardless of whether the handler invokes them. We implement a regex-based breadth-first import tracer that follows Python imports from the handler entry point through the shared layer, passing only reachable files to all policy generators.

Policy validation. Comparing action sets statically can measure overpermissioning but cannot determine whether a policy is *sufficient*, since runtime-only dependencies may be invisible in the source code. We therefore perform a policy-swap experiment on the live deployment. For each handler, we replace the authored policies (the developer-written CDK policies described above) with only the IPA-generated policy, execute targeted scenarios, and check for `AccessDenied` errors in CloudWatch logs. We then restore the original policies. Only one handler's role is swapped at a time; all other handlers retain their original policies, isolating the effect. The chatbot exposes its functionality through an AppSync GraphQL API, which is the entry point for all scenarios. We derived the scenarios from the application's event routing, tracing which AppSync resolvers invoke which Lambdas and Step Functions. The scenarios and validation scripts are included in the artifact. For `api-handler`, which serves as the GraphQL resolver for all queries and mutations, we run all scenario groups that route through it—including workspace lifecycle, document operations, file upload, and web crawling—providing comprehensive coverage of its code paths.

Table 2: Evaluation on 12 handlers of the *aws-genai-llm-chatbot* application. Action counts show the number of concrete IAM actions in each policy. Dev. = developer-authored; LLM_b = bare prompt; LLM_w = wildcards prompt. LLM action counts are medians over 5 trials. Live columns show whether the handler operates correctly under each policy. IPA is deterministic: ✓ = pass, × = fail. LLM experiments use 5 trials: 5 = 5/5 (success), 1–4 = 1–4/5 (partial), 0 = 0/5 (fail).

	Action counts				Live validation		
Handler	***IPA***	***Dev.***	***LLM_b***	***LLM_w***	***IPA***	***LLM_b***	***LLM_w***
add-user-to-group	3	6	6	3	✓	2	5
api-handler	83	104	34	40	×	0	1
create-aurora-ws	60	22	12	186	✓	1	5
create-opensearch-ws	60	23	9	34	✓	3	5
delete-document	60	79	21	29	✓	1	5
delete-workspace	60	79	19	33	✓	1	5
file-import-batch	62	84	32	35	✓	0	5
langchain-req-handler	70	339	212	211	✓	3	5
pg-setup	2	11	6	8	✓	4	5
send-query-resolver	5	18	9	11	✓	0	5
upload-handler	60	88	43	36	✓	5	5
web-crawler-batch	61	84	17	188	✓	0	5
Agg. factor (vs. IPA)	1.0×	1.6×	0.7×	1.4×			

LLM-generated policies. We also compare against LLM-generated policies (using Claude Opus 4.6 [15]). We evaluate two prompt strategies described in Sec. 5.1: *LLM_b* (bare prompt) and *LLM_w* (wildcards prompt), which achieved the lowest and highest success rates in the synthetic benchmarks, respectively. Both receive the same concatenated source files as IPA (handler entry point plus import-traced shared layer files). Each experiment is repeated 5 times; we apply the same policy-swap live validation to LLM-generated policies as to IPA.

5.2.1 Results. Table 2 presents the number of concrete IAM actions in each generated policy alongside the authored policies, and whether each handler operates correctly under the generated policy. Unlike the microbenchmarks, where IPA policies were compared against delta-debugged minimal baselines, here we compare against manual practice, which are not necessarily minimal policies. Delta-debugging is impractical in this setting because each iteration requires deploying and exercising a live multi-service stack. We note that the authored policies may include permissions for code paths that our scenarios do not exercise.

Table 2 shows that the authored policies, on average, contain more actions than the IPA-generated ones (by 1.6×). Two handlers (`create-aurora-workspace` and `create-opensearch-workspace`) are exceptions: IPA generates more actions because the import tracer includes shared-layer code paths not exercised at runtime.

Eleven of twelve handlers pass all scenarios with IPA-generated policies. The only partial failure is `api-handler`, where 12 of 16 scenarios fail due to a missing `ssm:GetParameter` permission. The root cause is that the handler calls `genai_core.parameters.get_config()`, which delegates to `aws_lambda_powertools.utilities.parameters`, an AWS utility library that wraps the SDK call. IPA detected SSM as a service from other code paths but missed this specific operation because it is not a direct SDK call. This is a known limitation of the tool's SDK-call-only analysis (Sec. 5.3). An additional missing permission (`secretsmanager:GetSecretValue`) affects one scenario (`list_models`), but is masked by the SSM failure, so even with the SSM permission added, `api-handler` would not pass all scenarios. The four passing scenarios use DynamoDB directly without traversing either wrapper.

LLM comparison. LLM_b performs poorly: only `upload-handler` passes all 5 trials. As seen already in the synthetic benchmarks, without resource instructions the LLM generates resource ARNs containing unresolved placeholder variables (e.g., `${AWS::Region}`, `${WORKSPACES_TABLE_NAME}`) that IAM rejects as syntactically invalid. LLM_w performs substantially better: 11 of 12 handlers pass all 5 trials. LLM_w correctly identifies the actions that IPA misses (including `ssm:GetParameter`), but `api-handler` still fails in 4 of 5 trials because the LLM generates resource ARN patterns that do not match the CDK-generated resource names. For example, the LLM restricts `ssm:GetParameter` to resources matching `parameter/*ConfigParameter*`, which does not match the actual CDK-generated parameter name (`CFN-SharedConfig358B4A20-6LBjNlmyRxpN`).

With real-world applications we found that LLM-generated policies can contain more actions than IPA (1.4×), different from our findings in Fig. 3. This is due to the LLM using service-level wildcards (e.g., `bedrock-agentcore:*` which expands to 148 actions) where IPA uses concrete required actions.

LLM_w achieves comparable success but exhibits nondeterministic failures across trials, while LLM_b fails for most handlers. The results show that IPA-generated policies are practical starting points for real-world applications: they are sufficient for 11 out of 12 handlers and less permissive than the authored policies and LLM_w.

5.3 Limitations

Gap to least-privilege policies. A policy generated by IPA may be overpermissive along three dimensions. On the *action* dimension the over-approximation is conservative and bounded: IPA aims to include every action the application may require, so a policy can contain actions a given execution does not exercise. On the *resource* dimension, IPA cannot currently resolve bucket names, table names, or other resource identifiers that come from variables, environment variables, or configuration files, resorting to `*` wildcards where resolution is not possible. We are extending our data-flow analysis capabilities to propagate values within the codebase and integrations with IaC templates that declare resource names. On the *condition* dimension, IPA emits conditions only where they arise from FAS dependencies. Conditions are otherwise highly dependent on the particular application—e.g., permissions scoped to a time window or an IP range—so their fully automatic generation is not feasible. There is a risk that developers or agents use the generated policies as-is, without narrowing to reach least privilege. We plan to mitigate this by emitting warnings. Measuring the work required to refine a generated policy to a least-privilege one, relative to authoring from scratch or narrowing a managed policy, is future work.

Third-party SDK wrappers. Libraries that wrap AWS SDK calls (e.g., custom client wrappers, `aws_lambda_powertools`) are not recognized by the AST patterns. The tool only detects direct SDK usage. The real-world evaluation (Sec. 5.2) confirms this limitation: the single handler failure was caused by an SDK call routed through a third-party utility library. While in principle the source code of the wrappers could itself be analyzed statically alongside the application code, it is likely to produce high false positive rates. We are investigating interprocedural analysis to selectively analyze only wrapper code used by the application.

Identity-based policies only. The tool generates identity-based policies. Other policy types (resource-based, etc.) are out of scope.

Nondeterminism from changes in Service Reference. IPA queries the Service Authorization Reference at runtime (Sec. 4.2), so a generated policy is reproducible only against the version served at analysis time. Making previous versions of the Service Reference model available for auditability, and recording its modification date in IPA's output, is future work.

FAS map completeness. The embedded FAS maps cover the most common cross-service dependencies but are not exhaustive. A missing entry leads to underpermissioning, and a stale entry to overpermissioning. The current FAS model is curated by experts at AWS. To guarantee accuracy, we are automating its construction by building a system that attaches tracing metadata to FAS calls throughout the AWS network. This data can be surfaced through our existing authorization context sampling system [5] to reconstruct how services call each other on behalf of the customer.

Dynamic dispatch and reflection. SDK calls invoked via reflection, dynamic dispatch (e.g., Python's `getattr(client,method_name)`), or decorators are not handled by static extraction.

6 Related Work

IAM policy analysis and generation. The most closely related work is `iamfast` [29], which performs deterministic static analysis of application source code to generate IAM policies, supporting Python, Java, Go, and JavaScript. `iamfast` focuses on direct SDK call to action mappings. Besides extracting direct calls, IPA can additionally model cross-service dependencies, performs SDK call disambiguation, and models higher-level SDK abstractions (e.g., waiters, paginators, or utility methods). IAM Access Analyzer [8, 23] generates policies from CloudTrail logs after deployment, recommending only the permissions that were observed in use. This requires deploying the application with permissive policies first and observing its runtime behavior to identify opportunities for reducing permissions. IPA and Access Analyzer can thus complement each other:

IPA generates an initial policy that allows the application deployment, while Access Analyzer helps the user identify and eliminate possible overpermissions.

Zelkova [18] uses SMT-based reasoning to verify properties of existing IAM policies (e.g., whether one policy is more permissive than another). Policy linters such as Parliament [25] check existing policies for syntax errors and anti-patterns but do not generate policies from code. Cloud security auditing tools—Prowler [34], ScoutSuite [31], and AWS Config rules—scan deployed infrastructure for misconfigurations but do not analyze application source code to generate the policies. All these tools are complementary to IPA, which generates policies from an existing codebase that can then be analyzed and refined using these analysis tools. Yeboah-Duako and Datta [38] statically extract function-to-resource interactions from serverless application code and compare them against declared policies to quantify overprivilege, a dual to the generation problem addressed here. Their measurement of real-world AWS Lambda applications corroborates our experimental findings, reporting overprivilege to be systemic and most pronounced for policies that rely on wildcards.

Static analysis for cloud security. Several tools apply static analysis to cloud infrastructure, but at the IaC layer rather than the application layer. `cfn-lint` [2] validates CloudFormation templates; `Checkov` [19] checks Terraform, CloudFormation, and Kubernetes configurations for security misconfigurations. These tools are complementary: they validate the infrastructure templates into which IAM Policy Autopilot's generated policies can be inserted. IaC frameworks such as CDK [7] partially address the policy-authoring problem through declarative permission grants (e.g., `bucket.grantRead(lambdaFn)`) that automatically derive IAM actions from resource configuration. However, these grants are defined only for selected permissions, not all IaC frameworks support them, and many applications are deployed without IaC entirely. Cauli et al. [21] formalized CloudFormation semantics for pre-deployment security reasoning, and Xu et al. [37] developed ontology-based modeling for cross-resource security analysis—both operate on infrastructure definitions rather than application code. CodeGuru [6, 30] performs static analysis of application code for quality and security best practices, including several AWS-specific checks, but does not generate IAM policies. Obetz et al. [32] apply static analysis to AWS Lambda serverless applications, employing summary functions to approximate the behavior of third-party libraries without analyzing their implementation, and statically resolving the identifiers of accessed resources. Both techniques address limitations of our approach, namely the handling of SDK calls wrapped by third-party libraries and the over-approximation of resource ARNs.

Permission inference in other domains. The problem of inferring required permissions from API calls has been studied in the Android ecosystem. Stowaway [26] and PScout [16] map Android API calls to the permissions they require, using static analysis structurally analogous to our extraction phase. However, the Android permission model is simpler than IAM: it has no resource-scoped permissions, no conditions, and no cross-service dependencies analogous to FAS. Our FAS expansion algorithm (Algorithm 1) has no counterpart in the Android domain. RBAC mining techniques [22] discover access control policies from existing permission assignments rather than source code—analogous in spirit to IAM Access Analyzer rather than to our approach.

Tool-augmented AI. AI coding assistants can generate IAM policies when prompted, but they exhibit three failure modes: missing required permissions, hallucinating invalid policy content, and stale knowledge of recently launched services. Our evaluation (Sec. 5) confirms these issues and additionally shows that LLM-generated policies are nondeterministic, showing also that pass rates may drop when additional project context is included in the prompt. MCP [14] and similar protocols enable LLMs to delegate domain-specific tasks to external tools. While its evaluation is beyond the scope of this paper, IPA includes an MCP server interface [17] aligned with this pattern: the LLM handles natural-language interaction and code context, while the static analyzer provides deterministic, auditable IAM analysis grounded in authoritative AWS metadata.

7 Conclusion

Writing correct IAM policies that follow the principle of least privilege remains a persistent challenge for cloud developers. IAM Policy Autopilot addresses this through deterministic static analysis of application source code, generating baseline policies that are grounded in authoritative AWS metadata, traceable to individual SDK calls, and available before the application is ever deployed.

Our evaluation shows that IPA-generated policies are sufficient to run 9 of 10 synthetic benchmarks across four languages and 11 of 12 handlers in a production-style chatbot application. The generated policies are an order of magnitude less permissive than the optimal selection of managed policies and about 60% less permissive than developer-authored policies. LLM-generated policies are similarly sufficient but exhibit nondeterministic failures.

The main avenues for improving completeness—inter-procedural analysis, automated FAS map construction, and support for third-party SDK wrappers—are discussed in Sec. 5.3. A natural next step is combining IPA's deterministic output with LLM-based refinement: the static analyzer provides a verified action skeleton that an LLM narrows with resource-specific ARNs and conditions, a workflow the MCP integration already supports. Integrating with IaC templates to resolve resource placeholders would further tighten the generated policies.

Data Availability Statement

IAM Policy Autopilot is available as open-source software at https://github.com/awslabs/iam-policy-autopilot under the Apache-2.0 license. A replication package archiving the synthetic benchmark applications, their ground-truth policies, the real-world application [3] and its evaluation scenarios, and the LLM context files used in our experiments is available on Zenodo [27] (DOI: https://doi.org/10.5281/zenodo.21706325).

References

[1] Amazon Web Services. 2013. botocore: Low-Level Interface to Amazon Web Services. https://github.com/boto/botocore. Accessed: 2026-04-13.
[2] Amazon Web Services. 2018. AWS CloudFormation Linter. https://github.com/aws-cloudformation/cfn-lint. Accessed: 2026-04-13.
[3] Amazon Web Services. 2023. AWS GenAI LLM Chatbot. https://github.com/aws-samples/aws-genai-llm-chatbot. Accessed: 2026-04-13.

[4] Amazon Web Services. 2025. API Models for all public AWS Services. https://github.com/aws/api-models-aws. Accessed: 2026-04-13.

[5] Amazon Web Services. 2025. AWS re:Invent 2025 — Innovation in Identity Security: How We Protect the Cloud and Help You Do It Too. https://youtu.be/qi1OStAJ3xk?t=1263. Accessed: 2026-07-31.

[6] Amazon Web Services. 2026. Amazon CodeGuru. https://aws.amazon.com/codeguru/. Accessed: 2026-04-13.

[7] Amazon Web Services. 2026. AWS Cloud Development Kit (CDK). https://aws.amazon.com/cdk/. Accessed: 2026-04-13.

[8] Amazon Web Services. 2026. AWS IAM Access Analyzer. https://docs.aws.amazon.com/IAM/latest/UserGuide/what-is-access-analyzer.html#what-is-access-analyzer-unused-access-analysis. Accessed: 2026-04-13.

[9] Amazon Web Services. 2026. AWS Identity and Access Management. https://aws.amazon.com/iam. Accessed: 2026-04-13.

[10] Amazon Web Services. 2026. AWS Managed Policies. https://docs.aws.amazon.com/IAM/latest/UserGuide/access_policies_managed-vs-inline.html. Accessed: 2026-04-13.

[11] Amazon Web Services. 2026. AWS Managed Policy Reference. https://docs.aws.amazon.com/aws-managed-policy/latest/reference/about-managed-policy-reference.html. Accessed: 2026-04-13.

[12] Amazon Web Services. 2026. AWS Service Authorization Reference. https://docs.aws.amazon.com/service-authorization/latest/reference/. Accessed: 2026-04-14.

[13] Amazon Web Services. 2026. Forward Access Sessions. https://docs.aws.amazon.com/IAM/latest/UserGuide/access_forward_access_sessions.html. Accessed: 2026-04-13.

[14] Anthropic. 2024. Model Context Protocol Specification. https://modelcontextprotocol.io/. Accessed: 2026-04-13.

[15] Anthropic. 2026. The Claude Model Spec — Claude 4.6 Opus System Card. https://www.anthropic.com/claude-opus-4-6-system-card. Accessed: 2026-04-20.

[16] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. 2012. PScout: Analyzing the Android Permission Specification. In *19th ACM Conference on Computer and Communications Security (CCS)*. ACM, 217–228. doi:10.1145/2382196.2382222

[17] AWS Labs. 2025. IAM Policy Autopilot. https://github.com/awslabs/iam-policy-autopilot. Accessed: 2026-04-13.

[18] John Backes, Pauline Bolignano, Byron Cook, Catherine Dodge, Andrew Gacek, Kasper Luckow, Neha Rungta, Oksana Tkachuk, and Carsten Varming. 2018. Semantic-based Automated Reasoning for AWS Access Policies Using SMT. In *2018 Formal Methods in Computer Aided Design (FMCAD)*. IEEE, 1–9. doi:10.23919/FMCAD.2018.8602994

[19] Bridgecrew. 2020. Checkov: Policy-as-Code for Cloud Infrastructure. https://github.com/bridgecrewio/checkov. Accessed: 2026-04-13.

[20] Max Brunsfeld et al. 2018. Tree-sitter: An Incremental Parsing System for Programming Tools. https://tree-sitter.github.io/tree-sitter/. Accessed: 2026-04-13.

[21] Claudia Cauli, Meng Li, Nir Piterman, and Oksana Tkachuk. 2021. Pre-deployment Security Assessment for Cloud Services Through Semantic Reasoning. In *Computer Aided Verification (CAV)*. Springer, 767–780. doi:10.1007/978-3-030-81685-8_36

[22] Alessandro Colantonio, Roberto Di Pietro, Alberto Ocello, and Nino Vincenzo Verde. 2009. A Formal Framework to Elicit Roles with Business Meaning in RBAC Systems. In *14th ACM Symposium on Access Control Models and Technologies (SACMAT)*. ACM, 85–94. doi:10.1145/1542207.1542223

[23] Loris D'Antoni, Shuo Ding, Amit Goel, Mathangi Ramesh, Neha Rungta, and Chungha Sung. 2024. Automatically Reducing Privilege for Access Control Policies. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 298 (Oct. 2024), 28 pages. doi:10.1145/3689738

[24] Herrington Darkholme. 2022. ast-grep: A CLI Tool for Code Structural Search, Lint, and Rewriting. https://ast-grep.github.io/. Accessed: 2026-04-13.

[25] Duo Labs. 2019. Parliament: AWS IAM Linting Library. https://github.com/duo-labs/parliament. Accessed: 2026-04-13.

[26] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. 2011. Android Permissions Demystified. In *18th ACM Conference on Computer and Communications Security (CCS)*. ACM, 627–638. doi:10.1145/2046707.2046779

[27] Antonio Filieri, Luke Kennedy, Kevin Luo, Matt Luttrell, Adrián Palacios, Akash Panda, Neha Rungta, Matthias Schlaipfer, Karan Jit Singh, Anvesh Tanuku, Nick Winans, Diana Yin, and Weiben Zhang. 2026. IAM Policy Autopilot: Static Analysis for Policy Generation from Application Code (artifact). doi:10.5281/zenodo.21706325

[28] Qi Huangfu and Julian A. J. Hall. 2018. Parallelizing the dual revised simplex method. *Mathematical Programming Computation* 10, 1 (2018), 119–142. doi:10.1007/s12532-017-0130-5

[29] Ian Mckay. 2021. iamfast: IAM Policy Generation from Application Code. https://github.com/iann0036/iamfast. Accessed: 2026-04-13.

[30] Rajdeep Mukherjee, Omer Tripp, Ben Liblit, and Michael Wilson. 2022. Static Analysis for AWS Best Practices in Python Code. In *36th European Conference on Object-Oriented Programming (ECOOP)*. Schloss Dagstuhl, 14:1–14:28. doi:10.4230/LIPIcs.ECOOP.2022.14

[31] NCC Group. 2018. ScoutSuite: Multi-Cloud Security Auditing Tool. https://github.com/nccgroup/ScoutSuite. Accessed: 2026-04-13.

[32] Matthew Obetz, Stacy Patterson, and Ana Milanova. 2019. Static Call Graph Construction in AWS Lambda Serverless Applications. https://www.usenix.org/conference/hotcloud19/presentation/obetz. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*. USENIX Association.

[33] Isaac Polinsky, Pubali Datta, Adam Bates, and William Enck. 2024. GRASP: Hardening Serverless Applications through Graph Reachability Analysis of Security Policies. In *Proceedings of the ACM Web Conference 2024* (Singapore, Singapore) *(WWW '24)*. Association for Computing Machinery, New York, NY, USA, 1644–1655. doi:10.1145/3589334.3645436

[34] Prowler. 2016. Prowler: AWS Security Best Practices Assessment. https://github.com/prowler-cloud/prowler. Accessed: 2026-04-13.

[35] Jerome H. Saltzer and Michael D. Schroeder. 1975. The Protection of Information in Computer Systems. *Proc. IEEE* 63, 9 (1975), 1278–1308. doi:10.1109/PROC.1975.9939

[36] Smithy. 2019. Smithy: A Language for Defining Services and SDKs. https://smithy.io/. Accessed: 2026-04-13.

[37] Zhixing Xu, Shengjian Guo, Oksana Tkachuk, Saeed Nejati, Niloofar Razavi, and George Argyros. 2024. Cloud Resource Protection via Automated Security Property Reasoning. In *39th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. ACM, 2170–2175. doi:10.1145/3691620.3695279

[38] Elvis Yeboah-Duako and Pubali Datta. 2026. Overprivilege Analysis of Security Policies in Serverless Cloud Applications. arXiv preprint arXiv:2607.02875. arXiv:2607.02875 [cs.CR] doi:10.48550/arXiv.2607.02875

[39] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and Isolating Failure-Inducing Input. *IEEE Transactions on Software Engineering* 28, 2 (2002), 183–200. doi:10.1109/32.988498

Received 2026-04-30; accepted 2026-07-01