Skyler: Static Analysis for Predicting API-Driven Costs in Serverless Applications

Bernardo Ribeiro
bernardoribeiro@tecnico.ulisboa.pt
INESC-ID / Instituto Superior Técnico, Universidade de Lisboa
Lisboa, Portugal

Mafalda Ferreira
mafalda.baptista@tecnico.ulisboa.pt
INESC-ID / Instituto Superior Técnico, Universidade de Lisboa
Lisboa, Portugal

José Santos
jose.fragoso@tecnico.ulisboa.pt
INESC-ID / Instituto Superior Técnico, Universidade de Lisboa
Lisboa, Portugal

Rodrigo Bruno
rodrigo.bruno@tecnico.ulisboa.pt
INESC-ID / Instituto Superior Técnico, Universidade de Lisboa
Lisboa, Portugal

Nuno Santos
nuno.m.santos@tecnico.ulisboa.pt
INESC-ID / Instituto Superior Técnico, Universidade de Lisboa
Lisboa, Portugal

Abstract

Unpredictable costs are a growing concern in serverless computing, where applications rely on cloud APIs with complex tiered pricing models. In many deployments, API calls dominate expenses, and a single overlooked design choice can escalate costs by thousands of dollars. Existing tools fall short: provider calculators need unrealistic manual estimates, and dynamic profilers only work post-deployment.

We present Skyler, a static analysis framework for pre-deployment cost estimation of API invocations in serverless workflows. Skyler models control flow behavior and pricing semantics to construct symbolic cost expressions using SMT formulas, exposing *economic sinks*, i.e., code paths where API usage disproportionately impacts cost. This enables developers to identify hotspots and prevent costly architectural errors early. Skyler supports JavaScript-based serverless applications across AWS Lambda, Google Cloud Functions, and Azure Functions, achieving high accuracy (mean absolute percentage error <1% for AWS and Google, 4.5% for Azure).

CCS Concepts: • **Computer systems organization** → **Cloud computing**; • **Theory of computation** → **Program analysis**.

Keywords: Static Analysis, Serverless Computing

ACM Reference Format:
Bernardo Ribeiro, Mafalda Ferreira, José Santos, Rodrigo Bruno, and Nuno Santos. 2026. Skyler: Static Analysis for Predicting API-Driven Costs in Serverless Applications. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '26), March 22–26, 2026, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 20 pages. https://doi.org/10.1145/3779212.3790221

ASPLOS '26, Pittsburgh, PA, USA

ACM ISBN 979-8-4007-2359-9/2026/03
https://doi.org/10.1145/3779212.3790221

1 Introduction

Serverless computing promises effortless scalability and a pay-as-you-go model, making it an increasingly popular choice for cloud applications [57]. However, its pricing model introduces a subtle yet significant challenge: costs can explode unpredictably. In real-world deployments, a single overlooked API call inside a loop or recursive function call can lead to thousands of dollars in charges, even when compute costs remain minimal. The Prime Video case study [30] exemplifies this risk: a surge in Tier-1 S3 operations and AWS Step Function calls pushed costs up by 90%, ultimately forcing a costly architectural redesign. These hidden *economic sinks*, i.e., code paths whose API usage disproportionately drives up costs, are notoriously hard to detect before deployment. Worse, this unpredictability also enables Denial-of-Wallet (DoW) attacks [17, 45, 56], where adversaries exploit public endpoints to trigger expensive workflows.

While provider calculators (e.g., AWS Pricing Calculator [4], Google Cloud Calculator [6]) can account for API usage, they rely on developers to supply accurate invocation counts, a task manageable for simple workflows but nearly impossible for realistic ones. Control flow constructs such as loops, recursion, and conditionals, combined with input-dependent behavior and interconnected functions, make manual estimation highly error-prone. Recent proposals such as Jolteon [61] and Orion [38] adopt a dynamic approach, profiling deployed functions to optimize run-time configuration and reduce compute costs. However, these tools require deployment and rarely account for downstream service charges (e.g., DynamoDB operations, S3 requests, Step Functions) that often dominate the final bill. Achieving accurate, multi-factor cost estimation for serverless workflows, particularly before deployment, remains an open challenge.

This paper tackles this gap by focusing on API invocation costs, often overshadowed by compute-focused approaches yet responsible for a large share of real-world expenses. We present Skyler, the first static analysis framework designed to uncover and quantify *economic sinks* in serverless workflows. Skyler models API pricing semantics, analyzes branching and looping behavior, and produces symbolic cost expressions parameterized by input features. This enables developers to identify cost hotspots, enforce budget guardrails, and prevent catastrophic overruns before code is deployed.

Skyler operates by extracting control and data flow information from serverless functions and workflow specifications, creating a language- and platform-agnostic intermediate representation. Based on this analysis, Skyler derives symbolic cost expressions encoded as SMT formulas capturing input-dependent behavior and control flow constructs such as loops and branching. These symbolic models enable cost simulations that highlight high-risk API paths and worst-case scenarios, helping developers to reason about their design choices before deployment. Skyler currently supports JavaScript, one of the most dynamic and challenging languages to analyze, and handles workflows targeting AWS Lambda, Google Cloud Functions, and Azure Functions. Our evaluation shows that Skyler achieves highly accurate cost estimates, with mean absolute percentage errors under 1% for AWS and Google and 4.5% for Azure.

Contributions: This is the first work to statically model API call costs in serverless workflows. Specifically, we:

- Introduce the concept of *economic sinks* to capture disproportionate cost drivers in serverless API usage;
- Design a static analysis technique that constructs symbolic cost models capturing input-dependent control flow and complex service-level pricing semantics;
- Develop and implement Skyler[1] for JavaScript-based serverless applications, with support for AWS Lambda, Google Cloud Functions, and Azure Functions;
- Create a benchmark suite of workflows and microbenchmarks to evaluate cost estimation accuracy and facilitate future research on API-centric cost analysis.

2 Estimating API Costs in Serverless Apps

Accurately predicting the cost of serverless applications is essential yet notoriously difficult. While provider calculators offer basic estimates, developers must manually specify invocation counts and data sizes—a task complicated by chained function execution, input-dependent behavior, and diverse service pricing rules. This section motivates the need for better solutions through a running example, introduces the concept of economic sinks, and illustrates why existing estimation methods are error-prone and costly in practice.

[1]Skyler is available at https://github.com/arg-inescid/Skyler.git.

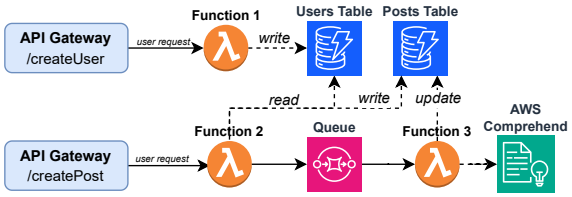

Figure 1. Architecture of the motivating serverless application with two endpoints (`/createUser`, `/createPost`) interacting with AWS services: DynamoDB, SQS, and Comprehend; arrows indicate data flow across functions and services.

2.1 Motivating Serverless Example

To illustrate the challenges of estimating API invocation costs, consider the simplified serverless application shown in Figure 1. While small, this application is representative of patterns found in real-world systems [50, 55]. It implements a basic message posting service with two HTTP endpoints exposed via Amazon API Gateway.

The first endpoint, `/createUser`, registers a new user by invoking a Lambda function that writes user information to a DynamoDB table (*Users table*). The second endpoint, `/createPost`, handles post submissions. It first checks if the author exists in the *Users* table, rejecting the request if not. If valid, the function stores the post in a separate DynamoDB table (*Posts table*) and, with a probability of 1%, forwards the post for moderation by placing a message on an Amazon SQS queue. Moderation is performed asynchronously by a third function triggered by batched SQS messages. This function calls AWS Comprehend, a natural language processing service, to detect toxicity in the post content. Posts flagged as toxic are updated in the *Posts* table with an additional attribute. Sampling is needed because AWS Comprehend is costly, making it financially infeasible to analyze every post.

2.2 Economic Sinks in Serverless Applications

The application introduced above illustrates how a seemingly simple serverless design interacts with multiple cloud services, each incurring API-based charges: DynamoDB for reads and writes, Amazon SQS for message queuing, and AWS Comprehend for text analysis. We refer to these API invocations as *economic sinks*, that is, points in the code where invoking a cloud API directly contributes to the application's cost according to the provider's pricing model.

Figure 2 shows the JavaScript code of the three main endpoints, highlighting the use of economic sinks: `db.putUser()` and `db.putPost()` perform writes to DynamoDB tables; `db.getUser()` performs a read from the *Users* table, and `db.updatePost()` updates a record in the *Posts* table, also interacting with DynamoDB. In contrast, `queue.send()` enqueues a message on Amazon SQS, and `analyzeToxicity()` calls AWS Comprehend for natural language analysis.

Each of these operations is individually priced by the cloud provider. For example, in the AWS `us-east` region, for DynamoDB, write capacity units (WCUs) cost 0.000 000 625 USD each, and read capacity units (RCUs) cost 0.000 000 125 USD each. A write corresponds to 1 KB written per WCU, while a read corresponds to 4 KB per RCU. Thus, `putUser` and `putPost` incur write costs, whereas `getUser` incurs a read cost. In turn, for Amazon SQS, sending a message costs approximately 0.000 000 4 USD per 64 KB chunk of message size, and with AWS Comprehend, text analysis costs 0.0001 USD per 100 characters analyzed, making it significantly more expensive than the other services.

To estimate the cost of this application using AWS's pricing calculator, the developer must provide, for each service, the expected number of API calls and, in some cases, the input size of each request. In practice, however, this task is difficult and error-prone, as we discuss next.

2.3 Challenges of Manual Cost Estimation

Using provider calculators, developers must reason about complex workflows, dynamic input dependencies, and service-specific pricing models, all without specialized tooling support. We illustrate the challenges of doing so using the example shown in Figure 2, which depicts the simplified JavaScript code of the three Lambda functions introduced earlier.

Challenge 1: Workflow complexity. We use the term *workflow* to refer to a sequence of function invocations and API calls triggered by a single request. Even for a simple endpoint like `/createPost`, multiple workflows can arise depending on runtime conditions. For example, the request may invoke only `db.getUser()` followed by `db.putPost()` if moderation is not required, or it may additionally call `queue.send()` to enqueue the post for review. If the post is selected for toxicity analysis, the workflow expands to include `analyzeToxicity()`, which calls AWS Comprehend, and potentially `db.updatePost()` to flag the content as toxic. Each of these workflows incurs different costs because they involve different economic sinks. Tracking all possible workflows manually is error-prone, especially when functions trigger others asynchronously, as seen with SQS in Figure 2. Cloud calculators cannot model these variations without explicit enumeration by the developer.

Challenge 2: Input-dependent behavior. API costs often scale with input size or external data state. DynamoDB operations charge per KB written or per 4 KB read. Thus, the cost of `db.putPost()` grows with the post's size, and `db.getUser()` depends on user record size. Similarly, AWS Comprehend charges $0.0001 per 100 characters analyzed, making long posts significantly more expensive than short ones. Control flow also depends on external data: if the user does not exist, the workflow ends early; otherwise, it may enqueue messages and trigger toxicity analysis. Figure 2 shows these data dependencies through arrows across functions,

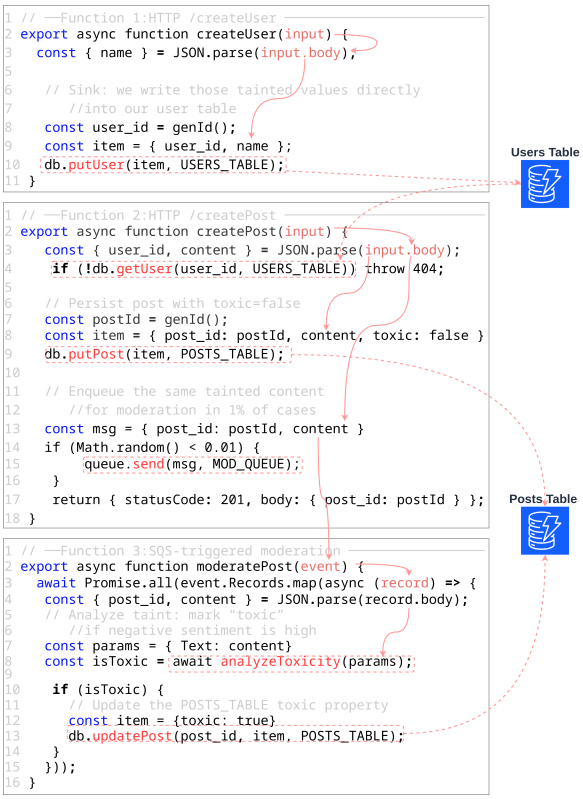

Figure 2. Lambda functions showing economic sinks to DynamoDB, SQS, and Comprehend. Solid arrows indicate data and control dependencies across functions. Dashed arrows indicate data flows between cloud services.

highlighting why developers struggle to predict actual costs based on static pricing tables alone.

Challenge 3: Service-specific pricing complexity. Each cloud service applies unique billing rules, often with non-obvious details. For example, Amazon SQS charges approximately 0.000 000 4 USD per 64 KB chunk of message size, so a 256 KB message is billed as four requests. DynamoDB uses distinct read and write capacity units, each priced differently per unit size, while AWS Comprehend charges per 100-character block. These rules make pricing highly fragmented. Omitting a single detail, such as SQS message fragmentation, can underestimate queue costs by 4×. This complexity is not unique to our example. A recent study of 145 real-world AWS serverless projects found that applications commonly use multiple services and hundreds of API calls, many billed by payload size [50]. This diversity underscores the impracticality of manually aggregating cost rules across services.

2.4 Cost Sensitivity to Workflow and Input Size

To understand how API costs compare to compute costs in serverless applications, we analyze our example under realistic conditions. We consider three workflows: `createUser`, `createPost#path-3` (enqueue without moderation), and

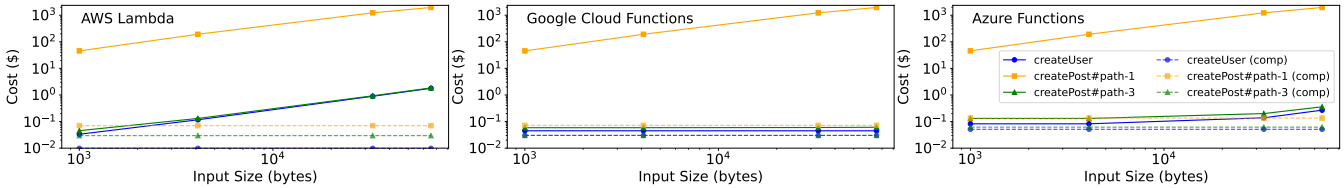

Figure 3. Simulated costs for the motivating application across AWS, Google Cloud, and Azure, assuming two executions per minute over 14 days. Solid lines include API costs; dashed lines show compute costs.

`createPost#path-1` (full moderation path). For each case, we simulate a steady rate of two executions per minute over 14 days, using serverless functions configured with 128 MB of memory, varying input sizes from 1 KB to 64 KB across three major cloud providers. API costs are computed using official pricing formulas [4, 6, 7], while compute costs are derived from measured execution times on each platform.

Figure 3 reports the total estimated cost for each configuration. Three key findings emerge. First, API invocations dominate overall cost in API-heavy workflows: on AWS and Google, compute represents less than 1% of the bill for `createPost#path-1`, despite spanning millions of invocations. Even for simpler endpoints like `createUser`, compute rarely exceeds 0.5%, except when API rates are flat and very low (as in Google), where compute can approach 40%.

Second, input size and workflow choice drive orders-of-magnitude variability. A single post processed via the full moderation path can be up to 1368× costlier than a minimal path, and increasing the payload from 1 KB to 200 KB amplifies costs by over 50× for some endpoints. This sensitivity means small mispredictions in input size or control flow can cascade into substantial cost overruns.

Finally, cost differences across providers are significant and non-linear. AWS is cheapest for small payloads but becomes the most expensive once inputs exceed ~7 KB, due to DynamoDB's tiered per-KB billing, while Google Cloud maintains a flatter per-operation rate. These results show that, for applications dominated by API interactions, precise pre-deployment cost estimation is essential: manual reasoning or static calculators cannot reliably capture these effects.

3 Skyler Overview

Skyler is an offline analysis tool that helps developers estimate and understand the API invocation costs of serverless applications in a fine-grained manner *before deployment*. Its goal is to identify and analyze economic sinks, i.e., API calls that dominate billing, and provide actionable insights to avoid costly design pitfalls. Skyler takes as input the application's source code and Infrastructure-as-Code (IaC) templates, which describe deployed resources, serverless functions, and event triggers. From these inputs, it constructs a symbolic cost model that developers can explore through a suite of queries to answer questions such as:

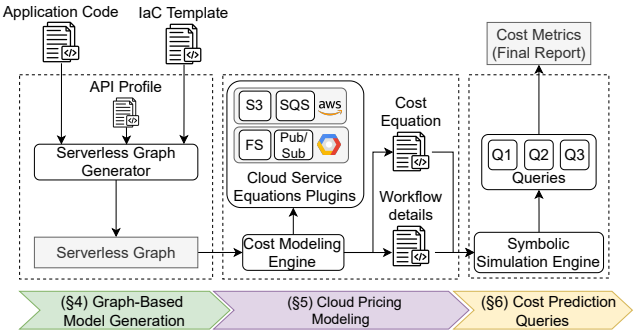

Figure 4. Skyler architecture and analysis pipeline. The system takes source code and IaC templates as input, builds a serverless graph, attaches provider-specific cost models, and executes symbolic queries to predict API invocation costs.

- *Which workflows account for the largest share of the application's cost?*
- *How do input characteristics, such as payload size, impact overall spend?*
- *What would the cost be if this workload ran on AWS vs. Google Cloud vs. Azure?*

Figure 4 shows Skyler's modular architecture, which consists of three main pipeline stages. First, a *serverless graph generator* parses IaC templates and code to build a unified representation of the application's control and data flows. Second, a *cost modeling engine* traverses this graph and instantiates symbolic cost equations using cloud-specific pricing plugins. Finally, a *symbolic simulation engine* runs analysis queries over the cost model to explore provider-specific costs, path-level cost concentration, or input sensitivity.

These stages reflect three design principles that guide Skyler's architecture: (1) a *graph-based model* for representing workflows and dependencies, enabling accurate cost attribution; (2) a *pluggable cost modeling framework* that captures provider-specific billing semantics; and (3) a *query-driven symbolic engine* that supports "what-if" analyses. While conceptually distinct, these components work in concert as a pipeline to deliver accurate, extensible, and platform-agnostic cost estimation. The following sections describe each stage in detail: graph generation (§4), cloud pricing modeling (§5), and cost prediction calculation (§6).

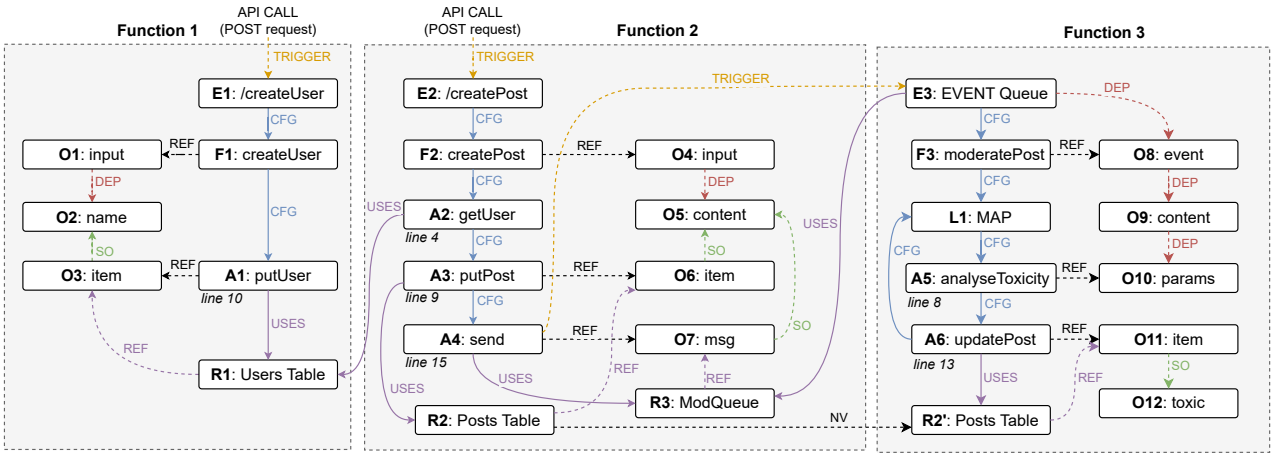

Figure 5. Simplified representation of the serverless economic graph for our toy blog-post application (see Figure 1 and Figure 2). `CFG` and `TRIGGER` edges represent control dependencies, while the remaining edges encode data dependencies.

4 Graph-Based Model Generation

Skyler's program analysis pipeline starts with the construction of an abstract data structure that captures the economic behavior of a serverless application. This data structure, which we call the *serverless economic graph* (SEG), serves as a unified representation that encodes all relevant execution paths, resource interactions, and object dependencies that may influence the application's operational cost. The SEG provides the foundation for later stages of symbolic simulation and cost estimation by exposing how cloud APIs are invoked, which resources they operate on, and how data flows between them. Next, we present the structure and semantics of the serverless economic graph, explain how it is automatically constructed, and discuss cloud-specific details.

4.1 Anatomy of the Serverless Economic Graph

The SEG captures the control flow and data dependencies that enable automated reasoning about the execution and cost impact of *economic sinks*, i.e., billable cloud API calls. To introduce the structure of the SEG, we use the example shown in Figure 5, which presents a simplified view of the SEG generated for the motivating application shown previously in Figure 2. We explain the graph by highlighting the following two core types of SEG dependencies:

Intra-function dependencies: These describe data and control dependencies that exist entirely within the boundary of a single serverless function. For example, in Function 1 of Figure 5, the API call `putUser`—an economic sink—is triggered by an HTTP `POST` request to the `/createUser` endpoint. The input object received via the request body is propagated through the function and eventually passed as an argument to `putUser`. This implies that the function's execution both reaches the sink and is influenced by external input, making the execution cost of the API call input-dependent.

The SEG models relevant entities in the function as graph nodes: event sources (`E#`), function entry points (`F#`), API calls (`A#`), loop nodes (`L#`), cloud resources (`R#`), and objects (`O#`). These nodes are connected using a combination of edges: control flow edges (`CFG`) encoding execution-order relations between instructions; dependency edges (`DEP`) encoding data dependencies between program objects; reference edges (`REF`) expressing that the source object is passed as a parameter to the target function; sub-object edges (`SO`) indicating that the target object is a sub-object of the source object; and new version edges (`NV`) linking successive versions of the same object. By following the tainted input sources, SEG traces paths to the economic sinks, such as `putUser`. The same intra-function analysis applies to Functions 2 and 3 in Figure 5, allowing SEG to pinpoint the origin of sink arguments, enabling fine-grained cost estimation.

Inter-function dependencies: These capture control and data dependencies that span across multiple serverless functions, typically mediated through asynchronous triggers or shared cloud-managed resources. For instance, in Figure 5, Function 2 issues a call to `send`—an economic sink that dispatches a message via a queueing service (e.g., AWS SQS). This message is consumed by Function 3, which receives it as an input and continues the processing.

SEG represents such interactions using special nodes for cloud-managed resources (`R#`) and edges to encode event-based invocation chains (`TRIGGER`) and represent data flow across resource boundaries (`USES`). These dependencies are inferred by combining static analysis of the application source code with information extracted from the IaC template, which defines event bindings, trigger policies, and service configuration. For example, SEG uses this information to identify that messages enqueued by Function 2 will asynchronously trigger Function 3. SEG also models inter-function interactions via shared state. In our example application, two databases

are shared among functions: the `Posts` table, accessed by Function 2 and Function 3, and the `Users` table, which is updated by Function 1 and later read by Function 2. Even when these functions belong to different workflows, SEG captures their interactions as a general form of inter-function dependency. Shared resources like databases are modeled as versionable entities, allowing SEG to reason about workflows' state evolution when estimating cost-impacting behavior.

4.2 Graph Construction

To build the SEG, Skyler extends the *multi-dependency graph* (MDG) introduced by Ferreira et al. [22]. Originally developed for vulnerability analysis in JavaScript programs, MDG provides a fine-grained representation of data dependencies between objects and their properties, along with a versioning-based model that tracks when writes occur. It offers a concise, yet expressive representation of data flows for JavaScript functions, particularly the nodes representing objects and function calls, and the dependency edges such as `DEP`, `SO`, and `REF`. However, for our usage scenario, SEG needs to extend this baseline to support both intra- and inter-function reasoning, capturing interactions with cloud services and enabling cost-oriented analysis of API usage.

To support this specific scope, we introduced several key extensions to MDG. First, we augmented the graph with *control flow edges* and dedicated *event nodes* (`E#`) to mark function entry points, whether triggered by HTTP requests or cloud events. We also introduced explicit *API call nodes* (`A#`), enriched with semantic annotations about the cloud services and operations being performed, as well as *loop nodes* (`L#`) to reason about calls to economic sinks inside loops. For inter-function analysis, we added *resource nodes* (`R#`) to represent cloud-managed entities such as queues, buckets, and databases, annotated with access modes and versioning information, which is particularly important for mutable resources like databases. Access relationships between API calls and resources are encoded using `USES` edges, which capture data dependencies across workflows and shared state. In addition, SEG models event-driven invocation paths across functions (e.g., how the invocation of the `send` API call in node `A4` triggers event node `E4`, which in turn leads to the execution of Function 3) by analyzing event patterns declared in the cloud configuration and adding `TRIGGER` edges. These augmentations generalize MDG to support SEG's specific goal: modeling how serverless applications interact with cloud resources in ways that affect cost.

To build the SEG, Skyler proceeds in two stages. It first parses Infrastructure-as-Code (IaC) templates, such as AWS SAM or Terraform, to extract the application's structural and behavioral specification, namely: provisioned services, serverless functions, event sources, conditional filters, and deployment regions, which influence costs due to regional pricing variation. Skyler also parses a provider-specific *API profile* file (detailed in the next section), which enumerates the platform's economic sinks and maps each API call to the corresponding cloud resource and operation (e.g., read or write on `DynamoDB`). Using this information, Skyler statically analyzes each function, combining inferred code dependencies with semantic annotations to construct partial graphs that capture intra-function control and data flows. In the second stage, Skyler integrates these per-function graphs into a unified SEG. It identifies linkage points by resolving event bindings, shared resource references, and inter-function control flow. Finally, it adds the necessary nodes and edges to link these components, producing a complete SEG instance.

4.3 Cloud-Specific Considerations

While the graph construction process is largely provider-agnostic, Skyler needs to accommodate cloud-specific differences across AWS, Azure, and Google Cloud.

First, to support platform-specific APIs and economic sinks, Skyler includes an *API profile* file tailored to each provider. This file, written by us in YAML, enumerates all billable cloud API calls, and maps each one to a corresponding resource type (e.g., `S3`, `DynamoDB`), its operational mode (e.g., `read`, `write`, `delete`), and any relevant constraints (e.g., payload limits). It may also include event mappings, such as linking `PutObject` operations to corresponding trigger events (e.g., `s3:ObjectCreated:*`). This modular approach isolates cloud-specific semantics from Skyler's core, making the system easily extensible to new APIs or platforms.

Second, the ease of analysis varies by platform. In AWS, SDK calls typically embed all relevant resource and payload information directly in the API arguments—e.g., a call to `putObject` will include the target bucket name and payload data. This makes it straightforward to link the invocation to its cost implications. In contrast, SDK usage in Azure and Google Cloud often follows a more layered structure. For example, Azure APIs commonly use a three-step pattern consisting of (1) client initialization, (2) binding to a resource, and (3) execution of the actual operation. To support these APIs, Skyler includes specific static analysis modules to detect these patterns, allowing Skyler to resolve which resource is targeted by the final API call.

Finally, while Skyler currently supports JavaScript-based applications, this choice stems from practical considerations as MDG-based static analysis tools [22] are readily available for JavaScript. However, the design of the SEG and Skyler's architecture are fundamentally language-agnostic. Porting the system to support other serverless programming languages, such as Python, is feasible and left as future work.

5 Cloud Pricing Modeling

With the SEG constructed, the next step in Skyler's pipeline is to generate a *symbolic cost model*, i.e., a parametric representation of the application's cost that incorporates both the

ID	Rule family	Canonical SMT-LIB snippet	Reads…	Writes…	Purpose & example
R-1	Declare	`(declare-fun id () Real)`	–	r, b, n, s, c, ℓ	Add symbols for request count r, batch size b, API calls n, payload size s, cost c, and loop counter ℓ.
R-2	Control (flow)	`(assert (= r_dst <expr(r,n,b,l)>) )`	Control flow edges	r_{dst}	Propagate invocation counts. Example for queue batches of 10: $rF3 = \frac{rA4}{10}$.
R-3	Data (flow)	`(assert (= s_out <expr(s_in)>) )`	Data flow edges	s_{out}	Track object size across functions, workflows. Example concatenation: `s_03 = s_name_02 + s_user_id_0x`.
R-4	Guard	`(assert (and <= s >=))`	Provider Info	–	Set provider limits (e.g., DynamoDB single-Item size cap), applying restriction to s symbolic vars.
R-5	Price	`(assert (= c <price(n,s)>) )`	Provider Info	c	Convert cloud service usage to dollars.

Table 1. Skyler's rule palette: the five orthogonal constraint families that together form the symbolic cost model.

structure captured in the SEG and billing information specific to the targeted cloud platform. The model is expressed in the SMT-LIB constraint language, a standard format for encoding logical formulas that can be analyzed using satisfiability modulo theories (SMT) solvers such as Z3. The formula's free variables represent factors that influence cost, including request counts, input sizes, loop iteration bounds, and unit pricing. Importantly, the model remains general: these variables are left symbolic so that different usage scenarios can later be explored by posing queries that instantiate them with specific values. The queries themselves, which we cover in the next section, are compiled to concrete solver invocations against this general formula, allowing Skyler to reason about cost without requiring re-analysis of the code.

The symbolic cost model is constructed by walking the SEG and applying five families of constraint-generation rules, summarized in Table 1. For each function, API call, resource, and loop node, Skyler generates symbolic variables representing request counts (`r`), payload sizes (`s`), loop counters (`l`), and costs (`c`) using `declare-fun` statements (*Rule R-1*). Control flow relationships are encoded using *Rule R-2*, which propagates invocation counts through the graph; this rule also accounts for batching (e.g., queue consumers) and loops by scaling counts accordingly. *Rule R-3* handles data-flow, tracking how object sizes are transformed or preserved as they pass through APIs. Guard constraints (*Rule R-4*) enforce hard bounds based on provider documentation (e.g., maximum item size or batch limits), pruning infeasible scenarios. Finally, *Rule R-5* encodes the cost formulas using unit pricing data, mapping usage patterns to dollar amounts. These rules provide a uniform and extensible method that translates SEG structure into an analyzable symbolic representation. To illustrate the structure of the symbolic cost model, we present in Appendix (§11) a simplified SMT-LIB fragment generated for Function 1 of the running example shown in Figure 5.

While the symbolic construction rules are cloud-agnostic, accurate cost modeling depends on cloud-specific pricing parameters and billing semantics. To address this, Skyler uses provider-specific *plugin modules* that encapsulate cost equations and constants for supported APIs. Each plugin specifies the algebraic formulas for calculating cost based on request count, object size, and other inputs, as well as the corresponding SMT variable mappings. Skyler currently includes plugins for multiple services across AWS (e.g., DynamoDB, S3, SNS), Google Cloud (e.g., Firestore, PubSub), and Azure (e.g., Blob Storage, CosmosDB). Adding support for a new service requires implementing a simple pricing function and registering the SDK call in a YAML manifest. During model generation, Skyler conditionally branches to the appropriate formula based on the targeted provider and selected APIs, enabling cross-provider simulations and comparative cost reasoning. This architecture also facilitates future extensions to support new services and evolving billing models.

6 Cost Prediction Queries

This section describes the final stage of Skyler's pipeline: executing *cost prediction queries*, a suite of programmable analyizes that provide actionable insights about cost behavior in serverless applications. These queries operate over the symbolic cost model produced in the previous stage, allowing developers to explore hypothetical scenarios, identify cost bottlenecks, and compare pricing across cloud providers.

Cost prediction queries are written in Python and invoke Skyler's symbolic simulation engine, which loads the SMT-LIB cost model into Z3 [16]. Queries may instantiate symbolic variables, such as input sizes or request rates, to simulate specific conditions, and then extract relevant results by solving the corresponding constraints. This symbolic execution is enriched with metadata derived from the SEG, such as the taint set of input objects and the execution paths associated with each function, enabling precise and customizable simulations. Developers can write new queries with reduced effort by specifying which input objects to vary, which execution paths of the serverless application to activate, and which provider-specific pricing model to use.

To showcase these capabilities, we implemented four built-in queries, each addressing a practical developer concern:

- **Q1: Which execution paths account for at least $X\%$ of total application cost?** Skyler computes a Pareto-optimal subset of workflows, i.e., potential application

execution paths, whose combined cost reaches the specified threshold $X\%$, assuming all workflows are triggered in a worst-case scenario. This highlights the most expensive paths in the application.

- **Q2: Which API calls within a single execution path contribute to $X\%$ of its cost?** On a specific workflow, Skyler identifies the API calls responsible for the majority of its cost using a Pareto-optimal ranking. This helps pinpoint expensive services used within a given path.
- **Q3: How does each input object influence cost?** Skyler varies the size of each tainted input object, while holding other parameters constant, and observes the effect on total cost. This reveals which inputs are most cost-sensitive and could benefit from constraints or validation.
- **Q4: How does cost compare across cloud providers?** Given a workload specification and symbolic model enriched with provider-specific cost annotations, simulate the application under different pricing schemes to identify the most cost-effective deployment platform.

Internally, queries Q1 and Q2 operate by setting all tainted object sizes to their upper bounds (based on known API constraints), activating one workflow at a time by setting its request count to one, and analyzing the resulting cost breakdown in Z3's model. Q3 simulates a series of workloads by sweeping each input size within its allowed bounds, tracking the change in total cost to identify dominant contributors. Q4 sets request counts to match a representative workload and performs a multi-provider simulation by toggling symbolic pricing constants tied to each cloud's API model. Skyler's symbolic engine handles these transformations automatically, enabling developers to reason about costs with fine granularity and without modifying application code.

7 Implementation

We implemented Skyler as a Python 3.12 prototype, totaling approximately 5,800 lines of code. The system is structured into three main modules. The serverless graph generator parses IaC templates and function code using a custom extractor (~1,800 LoC) and constructs a unified dependency graph (~1,500 LoC), stored in Neo4j (v5.26) and queried via Cypher. Skyler currently supports AWS SAM and Terraform for all constructs used in our benchmarks, with the simplification that the parser handles only the most semantically important IaC arguments (functions, triggers, resources, regions). Static analysis is aided by the use of `graph.js` [22], which facilitates MDG generation and traversal.

The second module, the cost equation generator, walks the SEG and instantiates symbolic cost formulas using a plugin-based system. Skyler currently supports 21 service-specific plugins (avg. 90 LoC each), covering major cloud APIs. These rely on a pricing library (~800 LoC) that fetches up-to-date pricing from provider APIs using `cachetools` v5.5.2 for caching. Final equations are emitted in SMT-LIB format.

Finally, the symbolic simulation engine (~550 LoC), executes cost prediction queries (~250 LoC) over the symbolic model using Z3 [16]. Queries specify simulation parameters such as input sizes, provider choice, and request profiles, and the engine resolves the symbolic model accordingly.

Extensibility and engineering effort. New queries operate solely on the symbolic cost model and require no changes to the SEG or pricing rules; implementing the existing queries (Q1–Q4) required between 20 and 60 lines of code per query, primarily for solver configuration and result extraction. Adding support for a new cloud service requires (i) a small YAML specification (5–6 lines) identifying the SDK call, service name, operation type, and payload constraints, and (ii) a Python function of approximately 70–120 lines encoding the provider's algebraic pricing rules (e.g., chunk sizes and tier thresholds); supporting AWS, Google Cloud, and Azure required no modifications to the SEG or query logic. The JavaScript front-end comprises approximately ~1,800 LoC. Extending support to other languages such as Python or Go is possible by leveraging existing static-analysis frameworks (e.g., CodeQL [2]) that already expose suitable dependency information and can be integrated without changes to the remainder of the system.

8 Evaluation

In this section, we address four key questions: (*i*) How accurate are our cost predictions when compared to ground-truth billing data from major cloud providers? (§8.2) (*ii*) Can Skyler extract actionable cost metrics that help developers optimize application deployment costs? (§8.3) (*iii*) How does Skyler compare with existing cost estimation approaches? (§8.4) (*iv*) What is the performance of Skyler's pipeline? (§8.5)

8.1 Serverless Benchmark Suite

To evaluate Skyler, we require a set of serverless applications that reflect realistic deployment scenarios, exhibit diverse control and data-flow behaviors, and exercise rich cloud API interactions. Unfortunately, there are no consolidated benchmarks suitable for this purpose. Existing suites such as FaaSdom [39] and SeBS [15] focus on cold-start latency, throughput, and memory usage, but do not include applications with complex API usage that drives real-world billing. Others, such as the Serverless-Security benchmark [51], are tailored for security evaluation and limited to Python.

To address these limitations, we curated a benchmark suite tailored to Skyler's goals. It comprises 16 benchmarks written in JavaScript, covering a broad range of architectural patterns and API usage scenarios. All benchmarks were initially developed for AWS, and Table 2 summarizes this baseline version.[2] Our suite includes both *microbenchmarks*, which isolate specific data flow and API invocation patterns,

[2]Additional details are provided in Table 7 and Table 8.

Microbechmark Category	Apps	WFs	Functions	Services	APIs	Triggers	LOC [Min-Max]
Intra-function (IntraF)	3	4	4	3	6	HTTP	[28, 138]
Inter-function (InterF)	9	13	19	15	24	HTTP, Storage, Database, Queue	[48, 204]
End-to-End Application		**WFs**	**Functions**	**Services**	**APIs**	**Triggers**	**LOC**
Claim Processing (CP)		5	7	1	9	HTTP, Invoke,Database	346
Booking (BK)		4	6	4	13	HTTP, DB, Queue	380
Image Processing (IP)		2	4	2	9	Storage, StepFunctions	247
Frame Analysis (FA)		1	2	4	5	Storage, Queue	129

Table 2. Overview of Skyler's serverless benchmark suite (AWS version), listing the total number of applications (Apps), workflows (WFs), functions, distinct cloud services (Services), API calls, event triggers (Triggers), and lines of code (LOC).

and *end-to-end applications*, which emulate full workloads with event triggers, loops, and economic sinks.

To support cost comparisons across providers, we ported most benchmarks to Google Cloud and Azure using equivalent APIs and services. Specifically, all but one inter-function microbenchmarks and one end-to-end application have been successfully ported. This cross-cloud mapping effort ensures that Skyler can evaluate cost behavior under consistent workloads across multiple platforms.

Microbenchmarks. We developed 12 microbenchmarks in total: five adapted from the Serverless-Security suite [51], and seven developed from scratch to stress additional dimensions of cost analysis. They are divided into: (1) *Intra-function*, where data flows from source to sink within a single function; and (2) *Inter-function*, where multiple functions communicate through shared cloud services such as S3 (object storage), DynamoDB (NoSQL databases), and SQS (message queues).

End-to-end applications. We implemented four realistic JavaScript applications adapted from Gupta et al. [24] and AWS reference use cases. Each application features elaborated workflows, multiple event sources, and extensive API interactions. *ClaimProcessing* supports five HTTP endpoints and invokes nine APIs to manage insurance claims submitted by customers and processed by adjusters, interacting with four DynamoDB tables. *Booking*, based on real-world airline reservation patterns [18, 55], stresses DynamoDB, SQS, and SNS across multiple workflows. *ImageProcessing* orchestrates nine API calls across two S3-triggered workflows, using Step Functions to coordinate retrieval, processing, and storage of images. *FrameAnalysis*, adapted from an AWS use case [10], was fully re-implemented in JavaScript. This benchmark suite enables us to evaluate Skyler on realistic workloads derived from AWS reference architectures and industry deployments, matching common triggers, services, and workflow depths reported in empirical Azure/AWS studies [50, 53].

Microbenchmark Application	AWS	Google	Azure
(IntraF) `StorageUsage`	0.31	1.53	2.22
(IntraF) `WebApp`	0	0.76	0.66
(IntraF) `WebAppWithBranchCondition`	0.10	0.71	0.64
(InterF) `StorageUsage`	0.03	1.08	6.73
(InterF) `StorageUsagePutWrongBucket`	0.02	0.71	8.29
(InterF) `StorageUsageWithQueue`	0	0.05	3.94
(InterF) `StorageUsage#2`	0.37	0.63	0.39
(InterF) `WebApp#2`	0.74	0.72	0.51
(InterF) `NosqlWriteNGetNotTaintedItem`	1.10	1.73	3.94
(InterF) `NosqlWriteNGetTaintedItem`	4.23	3.43	2.10
(InterF) `WebAppWithBranchCondition`	0	0.72	0.02
(InterF) `SeqWorkflows`	0	0.71	-
Ent-to-End Applications Workflow	**AWS**	**Google**	**Azure**
(BK) `registerBooking`	2.62	2.75	1.49
(BK) `registerPropertyFunction`	0.13	0	2.93
(BK) `registerUserFunction`	0.0	0	1.17
(BK) `reviewBooking`	0.01	0.71	8.72
(CP) `addAdjuster`	0.0	0	4.58
(CP) `addClaim`	2.25	0.09	0.61
(CP) `addUserPlan`	0.0	0	14.44
(CP) `getClaim`	0.03	0	12.54
(CP) `updateClaim`	0.0	0	7.68
(IP) `UserImagesBucket2`	0	5.74	-
(IP) `AdvertImagesBucket2`	0.20	0.19	-
(FA) `InputFrameBucket`	0.18	1.31	7.03

Table 3. MAPE (%) per microbenchmark application and end-to-end application workflow across different cloud providers.

8.2 Estimation Accuracy Against Ground Truth

We begin by evaluating Skyler's ability to predict real-world cloud costs. Our goal is to quantify how accurately it estimates API-specific charges when compared to billing data from AWS, Google Cloud, and Azure. To this end, we deploy each application on the respective cloud platform and measure the actual costs incurred. We ensure that API invocation costs are isolated from other billed components, such as compute time, storage, and logging, by filtering provider reports to focus solely on the relevant API service line items. On AWS, we enable hourly cost granularity through Cost and Usage Reports [3]; for Google Cloud and Azure, we rely on daily billing exports [5, 8]. Each serverless function is configured with 128 MB of memory. We issue a fixed number of requests per endpoint, ranging from hundreds to thousands, with controlled payload sizes. After a 12-hour delay to allow billing data to settle, we extract ground-truth costs and

Q1: Application	# of Workflows	P-90 Workflows
Booking	4	reviewBooking
ClaimProcessing	5	addClaim, addAdjuster, addUserPlan, updateClaim
ImageProcessing	2	UserImagesBucket2, AdvertImagesBucket2
FrameAnalysis	1	InputFrameBucket
Q2: Workflow	**Total APIs**	**P-85 APIs**
bookings (BK)	8	update, put
reviewBooking (BK)	3	detectSentiment
addClaim (CP)	5	put
UserImagesBucket2 (IP)	9	step func. transition
AdvertImagesBucket2 (IP)	7	step func. transition
FrameAnalysis (FA)	5	detectLabels
Q3: Workflow	**Object**	**Growth coeff. / byte**
reviewbooking (BK)	reviewComment	0.001025 (linear)
registerproperty (BK)	description	1.5e-06 (linear)
addClaim (CP)	ClaimType	1.5e-06 (linear)
addClaim (CP)	Name	1.5e-06 (linear)
addAdjuster (CP)	adjuster	8.79e-07 (linear)
updateClaim (CP)	status	8.79e-07 (linear)

Table 4. Skyler cost metrics for end-to-end applications.

run Skyler locally on our machine using identical invocation profiles to generate symbolic predictions.

Table 3 shows the Mean Absolute Percentage Error (MAPE) between Skyler's predictions and provider-reported costs. Intuitively, MAPE quantifies how much Skyler's estimated costs deviate from the actual billing values observed on the cloud; the lower the MAPE, the more accurate the prediction. For AWS and Google Cloud, Skyler achieves average MAPEs of 0.5% and 0.98%, respectively. Minor discrepancies arise from rounding effects: e.g., a 200 KB DynamoDB read consumes 49 Read Units (RUs), whereas Skyler's symbolic model yields 48.82. For Azure, the average MAPE is slightly higher at 4.5%. This stems from auxiliary operations (e.g., `RenewBlobLease`, `GetServiceProperties`) being billed independently but not yet modeled in Skyler. While these operations vary non-deterministically across runs, their cost impact is modest. The largest deviation (14.4%) occurs with CosmosDB due to opaque pricing dependencies on indexing policies and internal storage structures, which are not yet captured in our model; supporting such auxiliary and provider-specific behaviors is left for future work.

Takeaway. Skyler delivers accurate cost predictions, with average MAPEs of 0.5% on AWS, 0.98% on Google Cloud, and 4.31% on Azure, allowing developers to assess expected costs with reasonable confidence, without executing workloads.

8.3 Extracting Actionable Cost Metrics

We now evaluate Skyler's ability to extract actionable cost metrics that can guide developers in optimizing application deployment costs. To this end, we apply the four symbolic queries introduced in §6, each designed to uncover dominant cost contributors at different abstraction levels–workflows, API calls, and input objects–as well as to compare deployment costs across cloud providers. All results reported in Table 4 refer to our AWS benchmarks for queries Q1–Q3,[3] while the evaluation of Q4 is presented in Figure 6.

Q1: Which execution paths account for at least 90% of the total application cost? In *Booking*, the `reviewBooking` workflow alone accounts for over 90% of the application's total cost (assuming one worst-case invocation per entry point). Most of this cost arises from a single expensive API (Comprehend's `detectSentiment`) whose per-character pricing is orders of magnitude higher than DB or queue operations. This highlights a clear optimization opportunity: developers should focus cost-reduction efforts on this workflow. The remaining applications exhibit a more balanced cost distribution, with no single workflow dominating the total spend.

Q2: Which APIs calls in an execution path contribute to 85% of its total cost? This query asks Skyler to identify, for each workflow, the minimal set of API calls that collectively account for at least 85% of its total cost. Table 4 shows the results for selected workflows in the third column under Q2. For example, in the `reviewBooking` workflow, the Amazon Comprehend `detectSentiment` API alone contributes more than 85% of the total cost. In *ImageProcessing*, while individual storage operations are inexpensive, the workflow relies on Step Functions/Workflows transitions, which bill per state transition; this cumulative transitions account for over 85% of per-invocation costs. Lastly, in the *FrameAnalysis* workflow, the `detectLabels` API is responsible for the bulk of the cost; however, since it charges a flat fee per invocation, it offers limited opportunities for cost reduction. These findings suggest practical optimization strategies. For cost-sensitive APIs like `detectSentiment`, developers can consider constraining input sizes to lower per-request charges. For workflows relying on Step Functions, like *ImageProcessing*, switching to S3 event triggers may eliminate orchestration overhead while preserving the intended functionality.

Q3: How does each input object influence cost? This query quantifies the sensitivity of application costs to the size of individual input fields. Skyler symbolically simulates increasing the size of each field in the input payload and computes the marginal cost increase per byte, ranking input fields by their cost impact. Table 4 reports, for each workflow, the input field with the highest cost coefficient. In the `reviewBooking` workflow, the `reviewComment` field dominates, incurring the steepest cost increase as its size grows. In *ClaimProcessing*, the `Name` and `Adjuster` fields emerge as primary cost drivers, as they are passed to the `put` API. These findings suggest that developers should prioritize size constraints for these fields to curb cost growth. In contrast, the input fields in *ImageProcessing* and *FrameAnalysis* do

[3] Full cross-provider results are provided in Table 9.

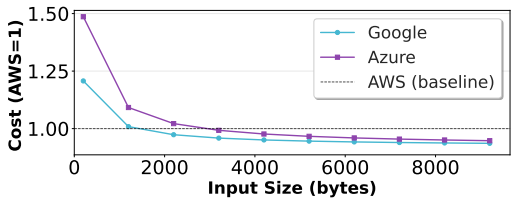

Figure 6. Cloud provider costs normalized to AWS for different input sizes of the *Booking* application.

not significantly influence total cost (excluding storage fees), indicating that payload size is not a dominant cost factor.

Q4: How does the cost compare across cloud providers? This query estimates how the same workload would be billed by different cloud platforms. We analyze the *Booking* application for AWS, Google Cloud, and Azure, fixing the number of requests per endpoint (1 M for `bookings`, 50 k each for `registerUser` and `registerProperty`, and 20 k for `reviewBooking`) while varying input payload sizes from 200 B to 10 KB. As shown in Figure 6, AWS is the most cost-effective choice for small payloads (≤1 KB), with cost savings of up to 32% at 200 B compared to the next-cheapest provider. For larger payloads, Google Cloud becomes more economical, with costs up to 6.4% lower than those of AWS and Azure. We see that provider cost efficiency depends heavily on workload characteristics. Skyler allows developers to quantify such trade-offs before committing to a deployment.

Takeaway. Skyler offers a systematic pre-deployment analysis tool that helps developers identify dominant cost contributors and compare provider pricing. These insights can inform targeted cost optimizations and may also assist in detecting cost-amplifying usage patterns that could signal vulnerabilities to denial-of-wallet attacks.

8.4 Comparison with Existing Approaches

Cloud providers offer two mechanisms, with increasing levels of sophistication, to estimate the cost of serverless applications prior to deployment: cost calculators and cloud emulators. In this section, we empirically compare Skyler against both baselines, focusing on the information they require from developers, the cost-related questions they can answer, and their limitations when applied to multi-function serverless workflows.

Provider cost calculators. Cloud cost calculators estimate application costs by aggregating charges across individual services. These tools are accurate with respect to provider pricing rules and support a wide range of services. However, they rely on a simple model in which developers must manually supply expected request counts, input sizes, and usage patterns for each cloud API call. Consequently, calculators do not reason about application structure, control flow, or data dependencies.

To better understand the effort required to use cost calculators accurately, we decompose manual cost estimation into four incremental steps that reflect increasingly detailed knowledge of application behavior. In *Step S1*, developers identify the economic sinks invoked by application entry points without analyzing control or data flow; in the absence of taint analysis, all API inputs are conservatively assumed to have maximal size. In *Step S2*, developers enumerate execution paths from external inputs to economic sinks by inspecting branch conditions and calls to local functions. *Step S3* extends this analysis to asynchronous event triggers and loops, which introduce additional execution paths and invocation amplification. Finally, *Step S4* performs full data-flow analysis, allowing developers to propagate input sizes and request frequencies accurately across workflows, including across inter-function dependencies.

Application	Experiment	S1	S2	S3	S4
Booking (BK)	Elements (#)	13	20	27	38
	Error (AWS)	531%	531%	525%	0%
	Error (Google)	608%	608%	649%	0%
	Error (Azure)	601%	601%	615%	0%
Claim Processing (CP)	Elements (#)	10	15	19	26
	Error (AWS)	131%	131%	189%	0%
	Error (Google)	27%	27%	127%	0%
	Error (Azure)	153%	153%	220%	0%

Table 5. Evolution of manual cost estimation complexity (# of elements considered) and % error across steps (S1–S4).

We apply this process to the two most complex applications in Skyler's benchmark suite, *Booking* and *ClaimProcessing*, and quantify both the number of elements a developer must consider and the resulting estimation error at each step for AWS, Google, and Azure. We define elements as application entry points, economic sinks, execution paths, event triggers, loops, and data dependencies. For *Booking*, we consider five requests to the `booking` endpoint and one request to each remaining endpoint, assuming average object sizes of 100 KB for `User` items, 400 KB for `Property` items, and 5 KB for `Review` items. For *ClaimProcessing*, we consider one request to each endpoint, assuming average object sizes of 5 KB for `Claim`, 200 KB for `Adjuster`, and 100 KB for `UserPlan`.

As shown in Table 5, manual estimation requires tracking a steadily increasing number of elements, reaching 38 for *Booking* and 26 for *ClaimProcessing*. Estimation error (MAPE) remains high through Steps S1–S3, often exceeding 500% for *Booking* and 100% for *ClaimProcessing*. In Step S1, this error arises because developers conservatively apply the maximum input size to every API invocation, simplifying estimation at the cost of large over-approximations. In Step S2, although developers analyze execution paths, error typically remains high because multiple paths from sources to sinks are identified, while only a subset corresponds to successful executions; failed or short-circuiting paths nonetheless contribute to cost overestimation. In Step S3, developer incorporate additional execution paths by considering loops

Feature	AWS	GCP	Azure
Emulator	`LocalStack`	`GCP Local Emulators`	`Azurite + Cosmos DB`
Services emulated (#)	51	6	4
Fully supported apps	2/4 (IP, CP)	1/4 (CP)	1/4 (CP)
Missing services	NLP, Rekognition, Translation	NLP, Workflows, Vision, Translation	NLP, Vision, Translation
Cost estimation limitations	No native cost estimation (all) No input size visibility (all) Limited API usage visibility (AWS) Manual event trigger configuration (GCP)		

Table 6. Comparison of local cloud emulators' limitations.

and event triggers; however, without precise data-flow information, the newly discovered sinks are assigned inaccurate input sizes, which amplifies the error. Accurate estimation is achieved only at Step S4, once developers perform a complete taint analysis that captures both intra- and inter-function data dependencies. In contrast, Skyler achieves high accuracy without requiring the developer to specify or analyze any elements manually.

Local cloud emulators. Cloud emulators allow developers to execute serverless functions and managed cloud services locally, and therefore represent a more advanced baseline than provider cost calculators. We assess their suitability for cost estimation through a survey and experimental evaluation of representative emulators for AWS, Google Cloud (GCP), and Azure, focusing on service coverage, setup complexity, and the availability of cost-relevant information.

Table 6 summarizes the capabilities of the evaluated emulators and their ability to support end-to-end applications, which is primarily determined by the set of cloud services they emulate. For AWS, the reference emulator is LocalStack [36], with which supports for local deployment and execution of unmodified serverless applications and emulates more than 51 AWS services. However, LocalStack supports only two of the four applications (IP and CP), as it does not emulate services such as Comprehend and Rekognition. For GPC, we use the official local emulators [23], which support only five services, together with a community-maintained storage emulator [44]. Emulating Azure applications requires Azurite [40] and the Cosmos DB Emulator [41], which together cover four services. In sum, GCP and Azure emulators support only one application (CP).

Even if all necessary services are supported, using local emulators for cost estimation is far from trivial as input sizes for API invocations are not exposed by any emulator, and only the AWS emulator provides limited visibility into API invocation counts. In particular, LocalStack reports per-service invocation counts via its dashboard but does not expose resource-level information or input sizes. Inspecting storage services is also insufficient, as only the final object state is available and overwritten or deleted objects cannot be inspected retrospectively. In GCP, event triggers are supported but not automatically configured, requiring developers to manually trigger events (e.g., via a monitoring program). Due to these limitations, developers must employ additional mechanisms to estimate costs. In our evaluation, this required developing custom scripts to configure local services and event triggers, instrumenting application code to log API invocation counts and input sizes, and manually aggregating the collected logs using provider cost calculators. For example, in the *ClaimProcessing* application, this involved instrumenting nine distinct APIs with both item size and invocation count. Even with these extensions, emulators fundamentally operate on concrete executions, requiring developers to explicitly select which execution paths to run.

We also evaluate the feasibility of answering Skyler's cost-related queries with emulators using the *ClaimProcessing* application. Queries Q1 and Q2 can be answered accurately, but only by manually identifying and executing the cost-maximizing execution path for each endpoint with inputs chosen to maximize cost. This process is error-prone, as conditional branches and inter-function dependencies significantly increase the number of possible execution paths. Query Q3 is also possible but particularly challenging: answering a single sensitivity query requires repeated executions for different input sizes, resulting in 120 distinct executions for *ClaimProcessing*. Inter-endpoint data dependencies further complicate this process, as certain execution paths can only be exercised after executing other endpoints in specific orders.

Takeaway. Cost calculators and local cloud emulators constitute the primary mechanisms available to developers for estimating serverless application costs today. Cost calculators accurately aggregate provider pricing rules but require developers to manually derive the quantities that drive cost, while emulators enable concrete execution but demand extensive instrumentation and exhaustive path exploration to analyze cost behavior. Both approaches operate on fully instantiated executions and do not scale to reasoning about the space of possible workflows. Skyler addresses this gap by elevating cost to a first-class, queryable property of serverless applications, enabling systematic reasoning across execution paths, inputs, and workflows without manual enumeration.

8.5 Quantifying Skyler's Analysis Overhead

We now evaluate the performance of Skyler's analysis pipeline by measuring the execution time of each stage described in §3. All experiments were run on a system with 32 GB RAM and an 8-core AMD Ryzen 7 PRO 7840U CPU at 3.3 GHz.

Figure 7 reports the end-to-end runtime for each analysis stage and query across the four benchmark end-to-end applications. The first stage, serverless graph generation (SG Gen), dominates overall runtime and scales linearly with the

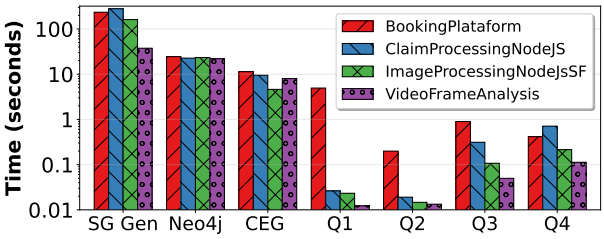

Figure 7. Execution time for each analysis stage and query.

number of functions. For example, *FrameAnalysis*, with only two functions, completes in 37.5 seconds, whereas *ClaimProcessing*, with seven functions, takes 280.4 seconds. This linear growth stems from running static analysis independently on each function, with each invocation taking 19-40 seconds on average. Given that real-world serverless applications typically contain only a small number of functions [53], this cost is acceptable in practice. The next stages, namely Neo4j import and Cost Equation Generation (CEG), are influenced by graph size but exhibit relatively stable execution times, remaining under 18 and 9 seconds respectively across all applications. Finally, query execution (Q1–Q4) is highly efficient, completing in under one second in most cases.

Takeaway. Skyler's full analysis runs once per code version and completes in under 312 seconds even for the largest applications in our benchmark. Moreover, queries can be executed repeatedly–and new ones introduced–without requiring code re-analysis, Skyler supports fast, iterative cost debugging and is well suited for CI/CD workflows.

8.6 Limitations

We identify two limitations in our prototype, which we defer to future work. First, when the number of API calls cannot be calculated statically (e.g. call inside a loop whose iteration count cannot be determined), Skyler requires developers to specify the expected number of API calls. Second, Skyler does not model complex database operations (e.g. AWS DynamoDB and Azure Cosmos DB `Query`) whose cost depends on internal factors (number of read entries, indexes, etc).

9 Related Work

Serverless cost analysis. Prior work has used dynamic profiling and modeling to improve cost-efficiency in serverless applications [19, 20, 27, 38, 61]. Orion [38] and Jolteon [61] focus on *computation* costs, modeling the impact of VM size and concurrency to meet SLOs at minimal cost. SONIC [37], Locus [49], and Pocket [29] address *data communication* costs across functions. In contrast, Skyler uses static analysis to estimate the overlooked costs of *cloud API usage*. As such, it provides a complementary lens to these runtime-oriented tools, helping developers reason about API costs before deployment or execution traces are available.

Serverless pricing models. A separate line of work rethinks how serverless platforms charge users. Pei et al. [46] propose pricing discounts when functions experience slowdowns. Lin et al. [35] break down infrastructure components to explain their contribution to billing. Cao et al. [14] design a pay-for-use platform with billing-aware scheduling. While Skyler does not change the pricing model, it provides visibility into how current models translate into costs at the level of APIs, enabling developers to make more cost-effective design decisions under existing billing schemes.

Program analysis. Code Property Graphs (CPGs) have been used extensively for vulnerability detection across C/C++ [59, 60], PHP [11, 47], JavaScript [2, 28, 32, 33], and WebAssembly [13], as well as for compliance [12, 21, 54] and privacy analysis [31]. ODGen [33] builds Object Dependency Graphs atop CPGs, while MDG [22] further captures object state evolution. Skyler extends MDG by introducing the SEG, which models event-driven, cross-service workflows via explicit event, trigger, and resource relationships. This allows Skyler to propagate data and cost across serverless functions and integrate provider-specific pricing semantics, which are capabilities beyond MDG's single-function scope. Within the serverless domain, GRASP [48] builds a reachability graph from IAM policies and IaC templates to identify exposed resources. Obetz et al. [42, 43] construct call graphs enriched with cloud-specific nodes and event-trigger edges. Growlithe [24] applies static data-dependency analysis to enforce permission boundaries. CtxTainter [9] uses dynamic taint tracking to detect inter-function leaks. To our knowledge, Skyler is the first framework to statically analyze fine-grained data flows for cost estimation in serverless systems.

Performance interfaces. PIX [25] proposes *performance interfaces* for network functions (NFs) to model performance metrics as a function of input and workload. It identifies performance-critical variables and uses taint-guided symbolic execution with hardware models to derive performance bounds. Applying this approach directly to serverless cost estimation is not possible since: i) serverless pricing is governed by provider-specific billing rules and is driven by API invocations and data volume across managed services, rather than by execution time or instruction counts alone; ii) serverless applications are event-driven and span multiple functions connected via triggers, cloud services, and infrastructure-as-code specifications, requiring end-to-end modeling of distributed control flow and cross-service data dependencies that PIX does not capture; iii) PIX assumes statically compiled programs with stable instruction semantics, whereas serverless functions are typically written in dynamic languages (e.g., JavaScript or Python), where runtime effects preclude stable performance-to-cost mappings. In sum, while both systems employ symbolic reasoning, PIX targets performance of single, self-contained programs, whereas Skyler reasons about monetary cost across distributed, event-driven serverless workflows.

Performance upper bounds and WCET analysis. Prior work on performance upper bounds and worst-case execution time (WCET) focus on deriving conservative latency guarantees under fixed hardware assumptions, using static control-flow and data-flow analysis together with microarchitectural models [34, 52, 58]. Recent systems extend these ideas to network functions by bounding performance despite input-dependent behavior through symbolic analysis and domain-specific abstractions [26]. While these approaches share methodological similarities with Skyler, they target performance metrics such as latency or throughput for single-program executions. In contrast, cost analysis in serverless systems depends on non-linear, provider-specific pricing rules and on event-driven, cross-service workflows defined via infrastructure-as-code triggers and managed services. These semantics do not arise in timing-based models and cannot be captured by attaching price tables to WCET-style abstractions. Skyler addresses this gap by integrating cloud pricing with cross-function dependencies through the SEG and symbolic cost equations.

Canary and testing environments. Canary deployments and testing environments operate post-deployment, observing application behavior under concrete workloads [1]. While effective for detecting functional regressions, they are poorly suited for cost analysis: cost overruns often arise from rare, input-dependent, or multi-service execution paths that limited test traffic is unlikely to exercise. Because these approaches rely on concrete executions, they cannot provide worst-case reasoning, input-sensitivity analysis, or closed-form cost models. Skyler instead targets pre-deployment analysis, enabling symbolic reasoning over all feasible workflows and early identification of cost risks before deployment.

10 Conclusions

Skyler statically estimates cloud API costs with high accuracy (<1% MAPE for AWS and Google, and <4.5% Azure). Empowered with Skyler, developers benefit from cost-aware development, receiving early cost estimates, including workflows, APIs, and providers, before deploying applications.

Acknowledgements

We thank the anonymous reviewers for their insightful comments and suggestions. This work was supported by IAPMEI under grant C6632206063-00466847 (SmartRetail), and by Fundação para a Ciência e a Tecnologia I.P. (FCT), under grants 2025.06573.BD and 2021.06134.BD, and projects UID/50021/2025 (DOI: https://doi.org/10.54499/UID/50021/-2025), UID/PRR/50021/2025 (DOI: https://doi.org/10.54499/UID/PRR/50021/2025), LISBOA2030-FEDER-00748300 (DOI: https://doi.org/10.54499/2023.16994.ICDT), and WebCAP (ref. 2024.07393.IACDC, DOI: https://doi.org/10.54499/2024.07393.IACDC).

References

[1] 2018. *Canarying Releases.* Accessed 20-08-2025.
[2] 2021. *CodeQL.* Accessed 20-08-2025.
[3] 2025. *AWS Cost and Usage Reports.* [Accessed 20-08-2025].
[4] 2025. AWS Pricing Calculator — calculator.aws. [Accessed 20-08-2025].
[5] 2025. Cloud Billing Reports | Google Cloud — cloud.google.com. https://cloud.google.com/billing/docs/reports. [Accessed 20-08-2025].
[6] 2025. Google Cloud Pricing Calculator — cloud.google.com. [Accessed 20-08-2025].
[7] 2025. Pricing Calculator | Microsoft Azure — azure.microsoft.com. https://azure.microsoft.com/en-us/pricing/calculator/. [Accessed 20-08-2025].
[8] 2025. Quickstart - Start using Cost analysis - Microsoft Cost Management — learn.microsoft.com. https://learn.microsoft.com/en-us/azure/cost-management-billing/costs/quick-acm-cost-analysis. [Accessed 20-08-2025].
[9] Mohamed Wael Alzayat. 2023. Efficient request isolation in Function-as-a-Service. *Ph.D. dissertation, Universität des Saarländes* (2023).
[10] Moataz Anany. 28-07-2017. Create a Serverless Solution for Video Frame Analysis and Alerting. https://aws.amazon.com/blogs/machine-learning/create-a-serverless-solution-for-video-frame-analysis-and-alerting/. [Accessed 20-08-2025].
[11] Michael Backes, Konrad Rieck, Malte Skoruppa, Ben Stock, and Fabian Yamaguchi. 2017. Efficient and flexible discovery of php application vulnerabilities. In *2017 IEEE european symposium on security and privacy (EuroS&P)*. IEEE, 334–349.
[12] Christian Banse, Immanuel Kunz, Angelika Schneider, and Konrad Weiss. 2021. Cloud property graph: Connecting cloud security assessments with static code analysis. In *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*. IEEE, 13–19.
[13] Tiago Brito, Pedro Lopes, Nuno Santos, and José Fragoso Santos. 2022. Wasmati: An efficient static vulnerability scanner for WebAssembly. *Computers & Security* 118 (2022), 102745.
[14] Tingjia Cao, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, and Tyler Caraza-Harter. 2025. Making Serverless {Pay-For-Use} a Reality with Leopard. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*. 189–204.
[15] Marcin Copik, Grzegorz Kwasniewski, Maciej Besta, Michal Podstawski, and Torsten Hoefler. 2021. SeBS: A Serverless Benchmark Suite for Function-as-a-Service Computing. In *Proceedings of the 22nd International Middleware Conference* (Québec city, Canada) *(Middleware '21)*. Association for Computing Machinery, 64–78.
[16] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: an efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340.
[17] Hieu Do. 2020. We Burnt $72K testing Firebase — Cloud Run and almost went Bankrupt. https://hotlemon99.medium.com/we-burnt-72k-testing-firebase-cloud-run-and-almost-went-bankrupt-part-2-8092a3bb773d/. [Accessed 20-08-2025].
[18] Hieu Do. 2023. A year of running a hotel booking application on AWS Serverless services for $0.8/month. https://hieudd.substack.com/p/a-year-of-running-a-hotel-booking/. [Accessed 20-08-2025].
[19] Simon Eismann, Long Bui, Johannes Grohmann, Cristina Abad, Nikolas Herbst, and Samuel Kounev. 2021. Sizeless: Predicting the optimal size of serverless functions. In *Proceedings of the 22nd International Middleware Conference*. 248–259.
[20] Simon Eismann, Johannes Grohmann, Erwin Van Eyk, Nikolas Herbst, and Samuel Kounev. 2020. Predicting the costs of serverless workflows. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering*. 265–276.
[21] Mafalda Ferreira, Tiago Brito, José Fragoso Santos, and Nuno Santos. 2023. RuleKeeper: GDPR-aware personal data compliance for web

frameworks. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2817–2834.

[22] Mafalda Ferreira, Miguel Monteiro, Tiago Brito, Miguel E Coimbra, Nuno Santos, Limin Jia, and José Fragoso Santos. 2024. Efficient static vulnerability analysis for javascript with multiversion dependency graphs. *Proceedings of the ACM on Programming Languages* 8, PLDI (2024), 417–441.

[23] Google. 2025. *gcloud alpha emulators*. Google. https://docs.cloud.google.com/sdk/gcloud/reference/alpha/emulators

[24] Praveen Gupta, Arshia Moghimi, Devam Sisodraker, Mohammad Shahrad, and Aastha Mehta. 2025. Growlithe: A Developer-Centric Compliance Tool for Serverless Applications. In *2025 IEEE Symposium on Security and Privacy (SP)*. IEEE, 3161–3179.

[25] Rishabh Iyer, Katerina Argyraki, and George Candea. 2022. Performance interfaces for network functions. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. 567–584.

[26] Rishabh Iyer, Luis Pedrosa, Arseniy Zaostrovnykh, Solal Pirelli, Katerina Argyraki, and George Candea. 2019. Performance contracts for software network functions. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 517–530.

[27] Jananie Jarachanthan, Li Chen, Fei Xu, and Bo Li. 2021. Astra: Autonomous serverless analytics with cost-efficiency and QoS-awareness. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 756–765.

[28] Soheil Khodayari and Giancarlo Pellegrino. 2021. {JAW}: Studying client-side {CSRF} with hybrid property graphs and declarative traversals. In *30th USENIX Security Symposium (USENIX Security 21)*. 2525–2542.

[29] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. 2018. Pocket: Elastic ephemeral storage for serverless analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 427–444.

[30] Marcin Kolny. 2023. Scaling up the Prime Video audio/video monitoring service and reducing costs by 90%. https://archive.is/LFtNg. [Accessed 20-08-2025].

[31] Immanuel Kunz, Konrad Weiss, Angelika Schneider, and Christian Banse. 2023. Privacy property graph: Towards automated privacy threat modeling via static graph-based analysis. *Proceedings on Privacy Enhancing Technologies* (2023).

[32] Song Li, Mingqing Kang, Jianwei Hou, and Yinzhi Cao. 2021. Detecting node. js prototype pollution vulnerabilities via object lookup analysis. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 268–279.

[33] Song Li, Mingqing Kang, Jianwei Hou, and Yinzhi Cao. 2022. Mining node. js vulnerabilities via object dependence graph and query. In *31st USENIX Security Symposium (USENIX Security 22)*. 143–160.

[34] Xianfeng Li, Yun Liang, Tulika Mitra, and Abhik Roychoudhury. 2007. Chronos: A timing analyzer for embedded software. *Science of Computer Programming* 69, 1 (2007), 56–67. doi:10.1016/j.scico.2007.01.014 Special issue on Experimental Software and Toolkits.

[35] Changyuan Lin, Mohammad Shahrad, et al. 2025. Getting to the Bottom of Serverless Billing. *arXiv preprint arXiv:2506.01283* (2025).

[36] Localstack. 2025. *Localstack Website*. Localstack. https://www.localstack.cloud/

[37] Ashraf Mahgoub, Karthick Shankar, Subrata Mitra, Ana Klimovic, Somali Chaterji, and Saurabh Bagchi. 2021. {SONIC}: Application-aware data passing for chained serverless applications. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 285–301.

[38] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Sameh Elnikety, Somali Chaterji, and Saurabh Bagchi. 2022. {ORION} and the three rights: Sizing, bundling, and prewarming for serverless {DAGs}. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 303–320.

[39] Pascal Maissen, Pascal Felber, Peter Kropf, and Valerio Schiavoni. 2020. FaaSdom: a benchmark suite for serverless computing. In *Proceedings of the 14th ACM International Conference on Distributed and Event-Based Systems* (Montreal, Quebec, Canada) *(DEBS '20)*. Association for Computing Machinery, 73–84.

[40] Microsoft. 2025. *Use the Azurite emulator for local Azure Storage development*. Microsoft. https://learn.microsoft.com/en-us/azure/storage/common/storage-use-azurite

[41] Microsoft. 2025. *What is the Azure Cosmos DB emulator?* Microsoft. https://learn.microsoft.com/en-us/azure/cosmos-db/emulator

[42] Matthew Obetz, Anirban Das, Timothy Castiglia, Stacy Patterson, and Ana Milanova. 2020. Formalizing event-driven behavior of serverless applications. In *European Conference on Service-oriented and Cloud Computing*. Springer, 19–29.

[43] Matthew Obetz, Stacy Patterson, and Ana Milanova. 2019. Static call graph construction in {AWS} lambda serverless applications. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*.

[44] oittaa. 2025. *Local Emulator for Google Cloud Storage Repository*. oittaa. https://github.com/oittaa/gcp-storage-emulator

[45] OWASP Foundation. 2018. OWASP Serverless Top 10. https://owasp.org/www-project-serverless-top-10/

[46] Qi Pei, Yipeng Wang, and Seunghee Shin. 2024. Litmus: Fair Pricing for Serverless Computing. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*. 155–169.

[47] Giancarlo Pellegrino, Martin Johns, Simon Koch, Michael Backes, and Christian Rossow. 2017. Deemon: Detecting CSRF with dynamic analysis and property graphs. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 1757–1771.

[48] 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*. 1644–1655.

[49] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. 2019. Shuffling, fast and slow: Scalable analytics on serverless infrastructure. In *16th USENIX symposium on networked systems design and implementation (NSDI 19)*. 193–206.

[50] Giuseppe Raffa, Jorge Blasco Alis, Dan O'Keeffe, and Santanu Kumar Dash. 2023. Awsomepy: A dataset and characterization of serverless applications. In *Proceedings of the 1st Workshop on SErverless Systems, Applications and MEthodologies*. 50–56.

[51] Giuseppe Raffa, Jorge Blasco, Dan O'Keeffe, and Santanu Kumar Dash. 2024. Towards Inter-Service Data Flow Analysis of Serverless Applications. In *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 654–658.

[52] Simon Schliecker and Rolf Ernst. 2011. Real-time performance analysis of multiprocessor systems with shared memory. *ACM Trans. Embed. Comput. Syst.* 10, 2, Article 22 (Jan. 2011), 27 pages. doi:10.1145/1880050.1880058

[53] Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the wild: characterizing and optimizing the serverless workload at a large cloud provider. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC'20)*. USENIX Association, USA, Article 14, 14 pages.

[54] Faysal Hossain Shezan, Zihao Su, Mingqing Kang, Nicholas Phair, Patrick William Thomas, Michelangelo van Dam, Yinzhi Cao, and Yuan Tian. 2023. Chkplug: Checking gdpr compliance of wordpress plugins via cross-language code property graph. In *Network and Distributed System Security (NDSS) Symposium 2023*.

[55] Masoom Tulsiani. [n. d.]. aws-serverless-ticket-booking. https://github.com/masoom/aws-serverless-ticket-booking. [Accessed 20-08-2025].

[56] Masoom Tulsiani. 2025. Serverless-Horrors. https://serverlesshorrors.com/. [Accessed 20-08-2025].

[57] Jinfeng Wen, Zhenpeng Chen, Xin Jin, and Xuanzhe Liu. 2023. Rise of the planet of serverless computing: A systematic review. *ACM Transactions on Software Engineering and Methodology* (2023).

[58] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. 2008. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.* 7, 3, Article 36 (May 2008), 53 pages. doi:10.1145/1347375.1347389

[59] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *2014 IEEE Symposium on Security and Privacy*. 590–604.

[60] Fabian Yamaguchi, Alwin Maier, Hugo Gascon, and Konrad Rieck. 2015. Automatic inference of search patterns for taint-style vulnerabilities. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 797–812.

[61] Zili Zhang, Chao Jin, and Xin Jin. 2024. Jolteon: unleashing the promise of serverless for serverless workflows. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 167–183.

11 Appendix: Symbolic Modeling of Cloud Pricing

To illustrate the structure of the symbolic cost model, we present in Listing 1 a simplified SMT-LIB fragment generated for Function 1 of the running example shown in Figure 5. As reflected in the SEG, Function 1 is triggered by HTTP requests to the `/createUser` endpoint (event node `E1`) and invokes an API call `putUser` (node `A1`), which writes object `O3` to a DynamoDB table. This object is derived from input object `O1`, passed via the HTTP request. Because `A1` is an economic sink, AWS charges for each invocation based on the size of the data written. Specifically, DynamoDB bills writes at a rate of 0.000 000 625 USD per 1024 byte chunk.

Skyler captures this cost structure symbolically by generating a set of SMT-LIB statements. The listing begins with variable declarations: request counters (e.g., `r_F1`, `r_A1`), payload sizes (e.g., `s_O1`, `s_O3`), and cost expressions (e.g., `c_A1`, `Total_COST`). It also includes constants derived from the provider's pricing model, such as the maximum allowed payload size and the cost per write chunk. The subsequent constraints encode the structure of the SEG: for instance, the equalities among request counts reflect direct invocation chains (e.g., `r_F1` equals `r_E1`, and `r_A1` equals `r_F1`); the equalities between sizes model object propagation (`s_O3` is ultimately derived from `s_O1`); and the cost equation for `A1` computes cost as the number of requests times the number of chunks written, times the unit cost. The final assertion computes the total cost for Function 1, and a `check-sat` command verifies model consistency. Skyler applies the same transformation across the entire SEG, producing a unified cost model for the entire serverless application.

```
; --- Declarations ---
(declare-fun r_E1 () Real) ; num. requests Endpoint 1
(declare-fun r_F1 () Real) ; num. requests Function 1
(declare-fun r_A1 () Real) ; num. requests API 1
(declare-fun s_O3 () Real) ; size of Object 3
(declare-fun s_O2 () Real) ; size of Object 2
(declare-fun s_O1 () Real) ; size of Object 1
(declare-fun c_A1 () Real) ; cost of API A1
(declare-fun Total_COST () Real)

(declare-fun AWS_DYNAMODB_WU_COST () Real)
(declare-fun AWS_DYNAMODB_WRITE_CHUNK () Real)
(declare-fun AWS_DYNAMODB_MAX_WRITE_INPUT () Real)

(assert (= AWS_DYNAMODB_WU_COST 0.000000625))
(assert (= AWS_DYNAMODB_WRITE_CHUNK 1024))
(assert (= AWS_DYNAMODB_MAX_WRITE_INPUT 409600))

; --- Equalities between r_* ---
(assert (= r_F1 r_E1))
(assert (= r_A1 r_F1))

; --- Equalities between s_* ---
(assert (= s_O3 s_O2))
(assert (= s_O2 s_O1))

; --- Bounds on s_O1 ---
(assert (>= s_O1 0.0))
(assert (<= s_O1 AWS_DYNAMODB_WRITE_INPUT))

; --- Cost equations ---
(assert (= c_A1 (* r_A1 (/ s_O3 AWS_DYNAMODB_WRITE_CHUNK)
      AWS_DYNAMODB_WU_COST)))
(assert (= Total_COST c_A1))

(check-sat)
```

Listing 1. Simplified SMT-LIB cost model for Function 1.

Application Name	Description	Features Tested	Funcs	Services	APIs	Triggers
Intra-Function Applications						
StorageUsage	Application that performs basic storage operations within a single function flow.	Storage service, intra-function data dependencies	1	2	2	HTTP
WebApp	Application that performs basic NoSQL database operations within a single function.	Database service, intra-function data dependencies	1	1	1	HTTP
WebAppWithBranchCondition	Web application demonstrating conditional branching within a single serverless function. One execution path performs a database write, while the other performs a queue write, both depending on external inputs.	Database and queue services, branch conditions, intra-function data dependencies	2	2	3	HTTP
Inter-Function Applications						
StorageUsage	Storage application demonstrating inter-function communication through event triggers.	Storage event triggers, inter-function data dependencies	2	2	2	HTTP, Storage
StorageUsage#2	Application that writes data to storage and later retrieves it in a separate workflow.	HTTP trigger, storage access, inter-function dependencies across workflows	2	1	2	HTTP
WebApp#2	Application that writes an item to a NoSQL database and retrieves it in a different workflow.	HTTP trigger, databases, inter-function dependencies across workflows	2	1	2	HTTP
StorageUsagePutWrongBucket	Application that tests storage operations with misconfigured bucket settings to verify correct event filter behavior.	Storage event triggers with filters	2	1	2	HTTP, Storage
StorageUsageWithQueue	Application that consumes messages from a queue and stores them in storage, modeling batch processing loops.	Queue events, batch processing loops, inter-function dependencies	2	2	2	HTTP, Queue
NosqlWriteAndGetTaintedItem	Application that writes and retrieves data items using NoSQL across multiple functions within the same workflow.	NoSQL operations, inter-function dependencies	2	2	3	HTTP, Queue
NosqlWriteAndGetNotTaintedItem	Application that writes an item with a large property (e.g., 400KB), updates this property with a small value (e.g., 10B), and retrieves it to evaluate update behavior and resource versioning.	NoSQL operations, inter-function dependencies, resource versioning	2	2	4	HTTP, Queue
WebAppWithBranchCondition	Web application with conditional branching logic across multiple functions. External inputs flow into data-processing services, with queues and databases handling inter-function data transfer.	Branch conditions, inter-function dependencies, cloud data processing services	3	4	5	HTTP, Database, Queue
SeqWorkflows	Sequential workflow implemented using orchestration services (AWS Step Functions and Google Workflows), modeling inter-function coordination and state transitions.	Orchestration services, inter-function dependencies	2	2	2	Storage, Orchestration

Table 7. Overview of Skyler's serverless benchmark suite (micro applications).

Application	Description
(CP) addAdjuster	Workflow responsible for registering new claim adjusters.
(CP) addUserPlan	Workflow responsible for registering new users.
(CP) addClaim	Workflow that adds a new claim, validates it, and assigns an adjuster to unassigned claims. The process consists of three steps: (i) the first function writes the claim to the database, (ii) the second function is triggered by this write to validate the claim, and (iii) the final function assigns an adjuster.
(CP) getClaim	Workflow that retrieves a claim from the database.
(CP) updateClaim	Workflow that updates the state of an existing claim.
(BK) registerPropertyFunction	Workflow that registers property details in the database so that the property can be booked.
(BK) registerUserFunction	Workflow that registers a user in the booking system.
(BK) registerBooking	Workflow with three stages: (i) receives user and property IDs plus booking dates, validates user and property existence, checks for conflicting bookings, and writes booking data; (ii) a second function processes payment, updates the booking state to confirmed and writes the guest and booking IDs to the queue. (iii) a final function consumes batched queue messages, retrieves guest details, and publishes a booking confirmation and guest info to the property owner through a topic.
(BK) reviewBooking	Workflow that allows users to submit reviews with comments and ratings. The system analyzes reviews for toxicity, stores them in the database, and updates the property rating.
ImageProcessing	Workflow that retrieves images from an S3 bucket, pre-processes them through a sequence of three or four functions orchestrated via AWS Step Functions, and stores the intermediate and final results multiple times in S3.
FrameAnalysis	Workflow inspired by a real-world video frame analysis system. It is triggered when a frame is stored in a bucket, processes the frame using an image analysis service to extract labels, stores the extracted metadata in a database, and publishes a notification through a topic when people are detected in the image.

Table 8. Overview of Skyler's realistic applications.

Q1: Application	**# of Workflows**	**P-90 Workflows**
Booking	4	reviewBooking
ClaimProcessing	5	addClaim, addAdjuster, addUserPlan, updateClaim
ImageProcessing	2	UserImagesBucket2, AdvertImagesBucket2
FrameAnalysis	1	InputFrameBucket
Q2: Workflow	**Total APIs**	**P-85 APIs**
registerBooking (BK)	8	update, put
registerUser (BK)	1	put
registerProperty (BK)	1	put
reviewBooking (BK)	3	detectSentiment
addClaim (CP)	5	put
addAdjuster (CP)	1	put
addUserPlan (CP)	1	put
getClaim (CP)	1	get
updateClaim (CP)	1	update
UserImagesBucket2 (IP)	9	step func. transition
AdvertImagesBucket2 (IP)	7	step func. transition
FrameAnalysis (FA)	5	detectLabels
Q3: Workflow	**Object**	**Growth coeff. / byte**
registerBooking (BK)	propertyId	1.504e-06 (linear)
registerBooking (BK)	guestId	1.504e-06 (linear)
registerBooking (BK)	endDate	1.504e-06 (linear)
registerBooking (BK)	startDate	1.504e-06 (linear)
registerUser (BK)	email	8.94e-07 (linear)
registerUser (BK)	userName	9.024e-07 (linear)
registerProperty (BK)	coords	1.504e-06(linear)
registerProperty (BK)	locationName	1.504e-06 (linear)
registerProperty (BK)	minimumGuestRating	1.504e-06 (linear)
registerProperty (BK)	price	1.504e-06 (linear)
registerProperty (BK)	propertName	1.504e-06 (linear)
registerProperty (BK)	description	1.504e-06 (linear)
reviewBooking (BK)	reviewComment	0.001025 (linear)
reviewBooking (BK)	rating	1.5040e-06 (linear)
addClaim (CP)	ClaimType	1.5e-06 (linear)
addClaim (CP)	Name	1.5e-06 (linear)
addClaim (CP)	UserId	1.5e-06 (linear)
addAdjuster (CP)	adjuster	8.79e-07 (linear)
addUserPlan (CP)	name	8.79e-07 (linear)
updateClaim (CP)	status	8.79e-07 (linear)

Table 9. Skyler all cost metrics for AWS end-to-end applications.

12 Artifact Appendix

12.1 Abstract

This artifact provides Skyler, a cost prediction tool for serverless applications that analyzes Infrastructure-as-Code (IaC) templates and application code to generate symbolic cost models. The artifact includes the complete source code, benchmark suite (SkylerBench), and instructions on how to build and run the experiments. These experiments result in the tables and plots presented in the paper, which can be used to validate the results.

12.2 Artifact check-list (meta-information)

- **Data set:** SkylerBench benchmark suite (Micro-Benchmark and MainExamples)
- **Run-time environment:** Docker container (Linux-based)
- **Output:** JSON files (parsed paths, configurations), SMT files, CSV files (query results)
- **Experiments:** Cost prediction evaluation, cost query extraction, performance overhead analysis
- **How much disk space required (approximately)?:** 4.3 GB (Docker image)
- **How much time is needed to prepare workflow (approximately)?:** 3 minutes (load pre-built image)
- **How much time is needed to complete experiments (approximately)?:** 2 hours
- **Publicly available?:** Yes. DOI: 10.5281/zenodo.17943056.

12.3 Description

12.3.1 How to access. The artifact is available as a persistent DOI at 10.5281/zenodo.18078537. The repository contains all source code, benchmarks, and Docker configuration files necessary to reproduce the experimental results.

12.3.2 Hardware dependencies. The evaluation of this artifact does not depend on specific hardware. Any system capable of running Docker is sufficient.

12.3.3 Software dependencies. The software requirements to evaluate this artifact are:

- Linux (tested with Ubuntu 24.04 and Fedora 39, kernel 6.11.9-100.fc39.x86_64)
- Docker (tested with Docker version 27.3.1)

All other dependencies (Python packages, Node.js, Neo4j, cloud provider SDKs) are included in the Docker image.

12.4 Installation

This section describes how to set up the Docker environment required to run Skyler. All scripts in this guide execute Docker commands and should be run from the `scripts/` directory. The output of experiments will be stored at Benchmark directory.

12.4.1 Prerequisites. Before setting up the environment, ensure that:

- Docker is installed and running on your machine
- You have sufficient disk space (the Docker image is approximately 4.3 GB)
- You have read/write permissions in the repository directory

12.4.2 Building the Docker Image. Skyler is distributed as a Docker container to ensure consistent execution across different environments. You can use our pre-built image.

Load Pre-built Image: To use the pre-built image, from the repository's root folder, run:

```
cd scripts
./load_image.sh
```

This approach takes typically 3 minutes as it avoids the compilation and installation steps.

12.4.3 Verifying the Installation. After building or loading the image, verify that it was created successfully. You should see an image named `skyler:latest`.

12.4.4 Important Notes. Credentials: AWS and Google Cloud credentials must be configured prior to running the following scripts. Google Cloud credentials are handled automatically during script execution, whereas for AWS you must manually export the required environment variables before executing each script.

```
export AWS_ACCESS_KEY_ID=your_access_key
export AWS_SECRET_ACCESS_KEY=your_secret_key
```

12.5 Experiment workflow

A description of Skyler, quick test instructions for running the full Skyler pipeline, and instructions for running custom simulations are provided in the README file in the `scripts` folder.

12.6 Evaluation and expected results

The artifact provides instructions to reproduce the experiments that support the claims of the paper in the README file in the `scripts` folder. The expected results from this artifact should reproduce the key results shown in Tables 3 and 4, as well as in Figures 6 and 7.

12.7 Reusability of the Artifact

The artifact includes the complete Skyler code and is organized to support extensibility and reuse. The `Reusability.md` file provides detailed instructions on how developers can extend Skyler to support new cloud providers, APIs, or cost queries.