A Case for Learned Cloud Emulators

Archit Bhatnagar, Yiming Qiu[†], Sarah McClure[‡], Sylvia Ratnasamy[‡], Ang Chen

University of Michigan [†]*The University of Hong Kong* [‡]*University of California, Berkeley*

ABSTRACT

Creating and maintaining cloud infrastructure via "DevOps programs" is essential to using the cloud. However, developing and testing the DevOps programs requires resource provisioning in the cloud, which is time-consuming and costly. *Cloud emulators* seek to enable high velocity development by emulating cloud-level APIs to DevOps programs, enabling frictionless testing locally without going through the cloud. However, developing these emulators today is tedious and error-prone: engineers need to digest extensive documentation, and hand-craft emulation logic for each service and service interactions. We make a case for a fundamentally different approach: to "learn" emulation logic from cloud documentation via automated code synthesis. We observe that this task is particularly amenable to AI automation, and that we can constrain the code generation using principled abstractions for accurate synthesis at scale. We report our preliminary findings and discuss new opportunities that our approach will enable. check

CCS CONCEPTS

• **Networks** → **Cloud computing**; • **Software and its engineering** → **Orchestration languages**.

KEYWORDS

Cloud Management, Emulation, Code Generation, Neural Synthesis, Formal Design

ACM Reference Format:

Archit Bhatnagar, Yiming Qiu[†], Sarah McClure[‡], Sylvia Ratnasamy[‡], Ang Chen . 2025. A Case for Learned Cloud Emulators. In *The 24th ACM Workshop on Hot Topics in Networks (HotNets '25), November 17–18, 2025, College Park, MD, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3772356.3772404

1 INTRODUCTION

As cloud computing gains popularity, DevOps engineering has become an essential task. DevOps engineers create and maintain resources for a cloud infrastructure; and to achieve scalable management, they do this programmatically using cloud development frameworks [4, 9, 10]. These programs eventually invoke low-level APIs [5, 12] exposed by providers to manipulate cloud resources (e.g., creating virtual machines or gateways). Like any other program, DevOps programs need to be tested and debugged before deployment ; however, testing against the cloud is expensive [14], and resource provisioning can be time-consuming. Prices and provisioning time can further increase for resources in high demand [50].

To enable no-risk, no-cost, and high-velocity cloud development, *cloud emulators* [6, 8, 11] are quickly gaining traction. Emulators mimic the cloud by exposing identical API interfaces to DevOps programs and simulating their execution in a mock environment, providing a lightweight backend without going through the real cloud. In order to emulate a resource (e.g., VM), emulator developers sift through cloud documentation, identify target APIs, and handcraft the mockup logic based on their understanding of the expected behavior. Interdependent resources (e.g., VM is associated with Subnet and VPC) further need to be emulated in relation to each other.

While this is a laudable effort, existing practices of emulator development cannot catch up to the complexity and dynamicity of the cloud ecosystem. For instance, AWS alone provides 240 services [2], and a service can expose up to 200 APIs; Azure and GCP exhibit a similar level of complexity. Market competition means that providers often add new services and upgrade existing ones, making the cloud a moving target [13]. Each cloud provider also features a different set of services and APIs, and more players are entering the cloud market (e.g., Oracle, Alibaba). Tenants often construct multi-cloud deployments to use best-of-breed features [20]; as a result, DevOps testing needs to emulate each of the clouds. Hence, manual emulator development—a tedious process that needs to be repeated for each provider—will be increasingly difficult. The stark reality is that even the most advanced emulator [6] today only covers 95 out of over 240 AWS services, and only with partial API coverage for these services; and no mature multi-cloud emulators exist to our knowledge.

In this paper, we make a case for a new approach to building cloud emulators—by "learning" emulation logic from cloud documentation through automated code synthesis. We observe that publicly available cloud documentation provides a treasure trove of information. No matter how complex a

HotNets '25, November 17–18, 2025, College Park, MD, USA

ACM ISBN 979-8-4007-2280-6/25/11

https://doi.org/10.1145/3772356.3772404

cloud's services may be, their usage and behavior is often described in painstaking detail by the cloud provider. This is fundamental to the cloud's business model, as providers need to sufficiently describe their interfaces to the tenants. The incentives are strong for providers to comprehensively document their services and keep the information up to date. Therefore, at least in principle, we should be able to develop "learned emulators" that read cloud documentation using Large Language Models (LLMs) and generate emulation code for the described behavior automatically. This new kind of emulators will potentially not only achieve higher service coverage, but can also easily adapt to service changes, and generalize across cloud providers without multiplying the engineering effort.

We aim to enable this by designing *an AI agent that mimics the workflow of a (human) emulator developer*: digesting swaths of information from documentation, coding emulation logic for each resource and its interactions, detecting incorrect API invocations from buggy DevOps programs, returning useful error logs for DevOps troubleshooting, and finally, testing the emulator against the actual cloud to find and close any gaps. Obviously, this is a challenging task, not least because LLM may hallucinate and generate arbitrarily buggy emulation code. However, we observe that cloud emulation logic follows a highly-structured style—we can view each resource as a *state machine* (hereby referred to as SM), where *transitions* are triggered by API invocations and may further affect the states and transitions of other resources. Through this formal model, we can impose aggressive constraints on code generation for high assurance.

From cloud documentation, we insist that the LLM articulate its knowledge using the abstraction of *a hierarchy of state machines*. Each SM can only perform transitions encoded in the documentation (e.g., creation, deletion, and updates). The hierarchy enables capturing resource containment relations (e.g., VM and Subnet are contained by their parent VPC), which scope the impact of SM operations; these can be likewise extracted from the documentation, as correctness checks: e.g., resource creation APIs should not be allowed to delete their parent resources; resource deletion must ensure that all children have been reclaimed. By targeting this narrow abstraction, we can drastically narrow the range of errors in an otherwise unfettered generation. Additional AI techniques [25, 43] can further ensure that generated outputs comply to a specified grammar. The SMs serve as an "executable specification," whose instructions are interpreted by an emulator framework that we will build as a one-time effort. Cloud changes can be captured by re-executing this process periodically against the latest documentation versions.

Aligning the behavior of this synthesized emulator to the actual cloud presents another interesting challenge. Thanks to the above abstraction, we can symbolically execute the SMs to produce high-coverage traces that exercise all possible behaviors. Alignment means that permissible behaviors should produce the same effects in the emulator and the cloud, and forbidden behaviors should fail in both; ideally, failures should also result in identical error codes and useful error messages to assist with DevOps debugging. By testing the effect of these traces in both environments, we can detect divergence, track down the source of errors, e.g., to a specific SM implementation, a specific interaction, or even back to the cloud documentation, and fix them. This alignment process also enables us to learn how the cloud produces error logs for each situation, so that our emulator can decode errors in a similar (or even richer!) style, e.g., by passing the specific failure contexts to LLMs to generate an informative response, further helping with the user with debugging.

We report preliminary results from our prototype, and describe new avenues of research. For instance, counting the number of state transitions could quantify cloud complexity, understanding the hierarchy of state machines could measure and improve interoperability, and identifying "anti-patterns" in state machines would help improve documentation quality. In short, we believe that a full realization of "learned emulators" will shed much new light on cloud manageability.

Table 1: The coverage of existing emulator (Moto) is low, even for critical resources like Network Firewall.

Services	APIs	Emulated	Coverage
Compute (ec2)	571	177	31%
DB (dynamodb)	57	39	68%
Network Firewall	45	5	11%
Kubernetes (eks)	58	15	26%
Overall (subset)	**731**	**236**	**~32%**

2 MOTIVATION

Emulation: Why and how. To enable high velocity cloud development, emulators such as LocalStack [6] (with Moto as the backend) have garnered immense popularity (60k+ Github stars). They mock up cloud APIs—e.g., responding to CreateVM() calls in a DevOps program by adding a mock VM name, state, and location to the internal state, but without actually creating anything else. Obviously, this mock cloud infrastructure cannot host real workloads, but it enables developing and testing in a frictionless environment without cloud provisioning. This requires extensive work, however, and cloud providers themselves prioritize building actual cloud features to stay competitive in the market, rather than emulating them. Providers also have stronger incentives for DevOps testing to occur in the actual cloud, since this would generate revenue. Hence, emulators are primarily developed by third-party developers based on the cloud's documentation.

Limitations of existing work. These emulators are Sisyphean efforts, struggling to keep pace with the ever-expanding cloud ecosystem. Analysis of LocalStack, the most mature emulator, shows over 800 contributors making 10k+ commits over 10 years. Missing features and behavioral discrepancies are commonplace [7], requiring continuous development. This leads to two significant limitations for cloud emulation.

Coverage: The cloud is simply too vast. As Table 1 shows, even popular resources like Amazon EKS (Elastic Kubernetes Service) have incomplete support, and some key resources (e.g., Network Firewall) have even lower coverage, e.g., only CreateFirewall() but not DeleteFirewall(). This forces developers to maintain complex and brittle testing setups that use the emulator for some resources and the real cloud for others.

Correctness: Manual implementation is not infallible, and subtle behavioral differences between the emulator and the real cloud are common. For example, a known issue of this emulator is that it allows the DeleteVpc() call to succeed even if it contained an Internet Gateway, while the real AWS API would reject this API with a "DependencyViolation" error. Such errors/inconsistencies can seriously undermine the reliability of the emulator as a testing tool, allowing incorrect code to pass through the emulator.

Need for a new approach. Hence, it is time to rethink the design of cloud emulators using a new approach, leveraging the power of LLMs and opting for automated code synthesis.

3 AN ILLUSTRATIVE EXAMPLE

We provide an illustrative example to establish intuition. At the heart of our design is the abstraction of *a hierarchy of state machines* that capture cloud operations. By orienting our design this way, we reap the benefits of good abstractions [35]: encapsulating info within modules to separate concerns, and composing modules for programming at scale. Each cloud resource is viewed as a SM, where a collection of state variables represents its attributes, and transitions are caused by API invocations which may modify internal (i.e., within the SM) and external state. Consider a Public IP address that can be used by a Network Interface (NIC).

The Toy Doc

A Public IP address allows Internet resources to communicate inbound to resources in our cloud. Also, Public IP addresses enable our resources to communicate to the Internet and public-facing cloud services. It exposes APIs for creation, deletion, and it can be associated with a Network Interface resource for network connectivity. In the latter case, the PublicIp, and the associated NIC must be located in the same cloud region. API signatures follow:

- **CreatePublicIP(arg)**: arg is a string and specifies the cloud region: 'us-east' or 'us-west'.
- **AssociateNIC(arg)**: arg is an NIC identifier and refers to an already-created NIC.
- **DestroyPublicIP()**: PublicIPs cannot be deleted if they are still attached to their NICs.

For information about creating a NIC and associated APIs, refer to page 2500.

$s \in$ State, $t \in$ StateType, $v \in$ Value, $pred$: Predicates

$prog$	::=	SM $states$ $transitions$	*state machine*
$states$	::=	$s_1 : t_1, \ldots, s_n : t_n$	*typed state*
$transitions$	::=	$expr$	*transitions*
	|	if $pred$ then $expr$ else $expr$	*write expr*
$expr$	::=	$primitive$ | $primitive$, $expr$	
$primitive$	::=	read(s,v)	*read*
	|	write(s,v)	*write*
	|	assert($pred$)	*assertion*
	|	call($transition$)	*call*

Figure 1: The grammar for specifying an emulator.

PublicIP's state would include its status (assigned/idle), cloud zone, IP version and it could further include another state parameter representing the attached resource (NIC), defined elsewhere in the documentation. Transitions occur through well-defined APIs that eventually modify this state.

```
1  /* An abstract state machine */
2  SM sm {
3      States S; //A collection of state vars
4      Transitions T; //Transitions modify state
5  }
```

Further, both state and transitions can be sourced from the documentation, whether by an emulator engineer or by an LLM. Although this example describes the *state variables* implicitly, cloud documentation often contains explicit definitions of resource states alongside their types (e.g., string/-boolean/enum). Likewise, *transitions* correspond to the APIs exposed by the resource, and they may be regulated by certain constraints, e.g., a PublicIP and attached NIC must have the same zone. Generally, these APIs fall into four categories: create(), which initiates a resource, destroy(), which does the opposite, describe(), which reads resource attribute, and modify(), which changes an existing state. The modify() class has the most variety, since resources have tens or hundreds of state variables and may further affect other resources. Hence, a general way to model them is to treat the SM as *symbolic*. A symbolic transition, e.g., modifyX(), will change the state variable named 'X', which is a resource attribute. Resource states can also be captured using a symbolic form.

For a given resource, we can refine the generic SM model using information extracted from the documentation. Figure 1 shows the grammar, and an example spec for the PublicIP is shown below. The `read` & `write` primitives operate on state variables within a SM (e.g., Lines 9-10 modify PublicIP status and zone); the `call` primitive triggers a state transition on an external NIC SM (e.g., Line 14 bidirectionally associates the NIC with the PublicIP); an `assert` primitive encodes constraints (e.g., Lines 13 and 18).

```
SM public_ip {
  States status:enum, zone:str, NIC: SM
    Transitions {
      CreatePublicIP(arg); //Creates PublicIP
      AssociateNIC(arg); //attach with a NIC
      DestroyPublicIP(); } //unassign
    CreatePublicIP(region:str) {
      write(status, ASSIGNED);
      write(loc, region); }
    AssociateNIC(nic_ref:SM) {
      assert(loc == nic_ref.loc);
      call(nic_ref.AttachPublicIP(self));
      write(NIC, nic_ref); }
    DestroyPublicIP() {
      assert(!NIC);
      write(status, IDLE); } }
```

4 RESEARCH AGENDA

We now describe our technical workflow (Fig. 2), a novel neuro-symbolic approach to automatically generate high-fidelity emulation code based on the SM abstraction. We also discuss new use cases our approach will enable.

4.1 Documentation wrangling

The first step of our workflow is to identify or curate authoritative cloud documentation. We call this preprocessing step *documentation wrangling*. Cloud providers have varying practices in maintaining and structuring their documentation. As an example, AWS documents its cloud usage in a set of PDFs, spanning hundreds to thousands of pages; just the EC2 compute instance alone has over 4000 pages [1]. These PDFs also have clear pagination with marked sections indexed on resource names (mapping each API to a resource) with API signatures and error codes. On the other hand, for Azure and GCP, relevant information is scattered across websites, and no consolidated PDF files exist. The pagination styles are also markedly different. Hence, ideally, we need an automated preprocessing step, aided by LLMs, to comprehensively identify the set of documentation for each cloud.

The scale and complexity of the cloud ensures that documentation will always be extensive. As a practical challenge, present-day LLMs have context window limitations. To address this challenge, the classic ML solution is to use Retrieval Augmented Generation (RAG) [33], but the semi-structured nature of cloud docs provides a unique opportunity: we should be able to create a symbolic parser, based on documentation structure, to preprocess information. The documentation follows a set template indexed by resource type and has ordered information (request, response types) for each API. This preprocessing can build resource-specific information, reducing amount of context that the LLMs have to process and improving the generation accuracy.

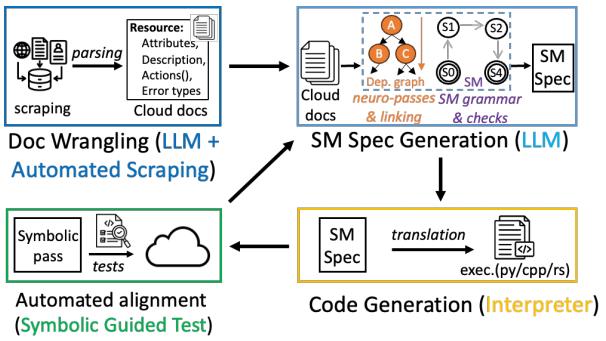

Figure 2: Our envisioned workflow.

4.2 Specification extraction

We envision that the resource SMs would serve as an "execution specification." They not only capture the intended cloud behavior but due to the constrained nature, can also be executed in an interpreter to emulate cloud behavior. This interpreter is the emulator framework that we need to manually engineer, representing a one-time effort. It maps the spec rules to code blocks, leveraging the grammar.

Ensuring *structured generation* comprises our next challenge. After all, LLMs are not guaranteed to generate outputs that conform to any grammar. While we can use iterative prompting by supplying the grammar in the prompt to ensure that the output is a legal spec, a more principled approach is to use *constrained decoding* [43], to constrain the next-token prediction process so that the token will only be generated if it does not violate predefined structures.

Due to the large number of cloud resources, we envision an *incremental extraction* to generate the SM specs. We first symbolically extract a resource-level dependency graph from API input/output dependencies. The LLM iterates over resources in a structured manner in multiple passes: generating individual SMs while leaving stubs corresponding to certain dependent resources that have not been generated—e.g., while generating CreateDefaultVPC() for VPC, it may call CreateSubnet(), which wouldn't have been processed yet. Finally, these incrementally-generated "modules" are spliced together, with a generation pass mimicking a *specification linking* process, patching unfinished stubs with the actual information, while also mapping failed assertions to error codes.

Next, before executing these specifications, we perform consistency checks with the goal of achieving *completeness* on resource type coverage and *soundness* against arbitrary errors. Our definition of completeness here is that all resource-types that we wish to generate code for are captured in the specification, e.g., if a resource A depends on resource B, then both are present in the specification. This again is enabled by the resource dependency structure by computing a

transitive closure. Soundness against arbitrary invalid specification is another consideration: although our grammar constrains the representation of the SMs, AI could still produce syntactically-valid but semantically-incorrect states and transitions. Therefore, we check the resource-level specification against behavioral requirements in the documentation—e.g., a describe() API will be flagged if it inadvertently modifies some state, or a transition attempts to make a call to SMs unreachable in its dependency graph hierarchy. We currently manually capture these template-based checks since it's a limited set, and envision that if needed, the check generation can be automated using LLMs with manual verification. Erroneous components will trigger another round of targeted correction until the spec passes our checks.

4.3 Automated alignment

Thus far, we have used the cloud documentation as the single source of truth. However, it is possible that documentation may contain slight errors or does not stay perfectly in sync with the actual cloud behavior [41]. Hence, the fidelity of the generated emulation program is predicated upon the comprehensiveness and correctness of the documentation. Hence, we need an automated alignment step for refining the emulation behavior. Whereas prior work has found emulator discrepancy using API fuzzing [26, 53], randomly fuzzing the entire emulator is inefficient and can make check mining inefficient.

Our approach involves performing symbolic passes [19, 31] over the SMs to divide the search space into symbolically equivalent classes, based on the check/assert conditions for each state transition. This enables guided searching and testing for each symbolic class. For validating existing checks, the SM ensures that there is a singular check violation in the generated test traces, which helps us pinpoint the failure root cause. Further, for mining missing semantic checks (by testing programs and tracing differences), we leverage the SM abstraction to find the minimal API traces that could trigger the discrepancies. If any discrepancy is identified by the above step, we feed the LLM with the delta to diagnose the error: are the differences attributed to the extracted spec, or the cloud documentation? Eventually, based on the diagnoses, the LLM updates the emulator to align with the cloud behavior. This phase closes the loop, allowing the emulator to continuously and autonomously improve its fidelity over time.

Ideally, we want the emulator to produce similar or identical error responses as the cloud, to further aid DevOps debugging. We hypothesize that "error codes" and "error messages" need to be treated differently. While the former needs to be identically aligned with the cloud response, the messages are for developer consumption and can deviate in its exact wording. Further, we may be able to provide even more informative responses than the cloud, by "decoding" the API call sequences using LLMs to suggest root causes and repairs.

4.4 New opportunities of this approach

We now reflect on the broader implications of taking a new approach to cloud emulation. By formalizing cloud API behavior into a machine-readable model, our approach enables novel ways to analyze and improve cloud services themselves.

Quantifying cloud complexity: The extracted specification comprises a graph of interacting state machines. This provides objective metrics (e.g., number of nodes, edge density) for a quantitative analysis of cloud service complexity. This allows for comparisons, for example, between the complexity of AWS Lambda and Azure Functions, and could assist cloud providers to modularize their resources.

Cloud gym: This emulation framework can also act as a playground for learning and testing cloud services for AI agents. There has been a recent line of work on building AI agents for cloud management [49], with the goal of automating DevOps engineering. To train these agents, we need a high-fidelity gym with a no-cost and zero-risk environment.

Documentation engineering: By analyzing the specifications, we can detect potential design flaws and anti-patterns. For instance, a modify() call that requires a long and complex chain of actions updating multiple dependencies across resources may indicate a poorly designed API; or, documentation that consistently leads the AI to generate incorrect logic may be flagged as ambiguous and in need of refinement. This will improve API and documentation engineering [3].

Multi-cloud emulation: Our approach is provider-agnostic and can generalize to any cloud backend. By consuming different cloud providers' documentation, we can generate a standardized formal model for all of them, and generate a "universal emulator" for testing multi-cloud DevOps programs. Our approach also enables formal, automated comparisons of equivalent services—e.g., whether Azure's CreateVM() requires the same dependency checks as AWS's RunInstance() in AWS—and can help improve cross-cloud portability.

5 PRELIMINARY EVALUATION

We have built a preliminary prototype based on our methodology, and present initial results benchmarking it against a) the state-of-the-art emulator LocalStack, which is manually engineered, and b) an approach that directly prompts an LLM to read cloud documentation and generate emulator code.

Prototype setup. For a fair comparison with direct code generation, we evaluate the emulator version using direct prompting (i.e., without RAG) to generate SM specs using the grammar. We currently don't employ constrained decoding but enforce syntactic checks in the interpreter and re-prompt in case of issues. The interpreter is a shim layer that maps SM blocks to Python code with some neural-assisted refactoring and helper code generation. We also showcase results

for the workflow without any alignment, but see significant improvements with alignment.

Basic functionality. We wrote an AWS DevOps program that creates a VPC, attaches it with a subnet, and then modifies the subnet to enable the MapPublicIpOnLaunch attribute, which assigns a PublicIP to all instances on launch. Our emulator was able to successfully maintain the required state (e.g., vpc_id, subnet_id) and process the API calls (e.g., emulate ModifySubnetAttribute()). The code synthesis only took a couple of minutes; moreover, our emulator's responses aligned with the actual cloud responses for this case.

Versus manual engineering. As discussed before, despite extensive efforts, manually-engineered emulators struggle to cover the diverse range of resources and APIs. Whereas Moto only covers 11% APIs for Network Firewall (and Localstack doesn't emulate that service at all), our preliminary prototype captures all 45 API calls through automated generation. Our prototype also captures all EC2 and DynamoDB API calls.

Versus direct-to-code. We then tested the direct-to-code (D2C) baseline, where the same LLM (i.e., Gemini 2.5 Pro) is prompted to generate the emulation logic directly from cloud docs. While D2C has achieved similar API coverage, the generated code is prone to critical logic and state manipulation errors that our system prevents by design. To evaluate accuracy, we compare the response alignment against the cloud for 4 traces across 3 scenarios: provisioning, state updates, and edge cases that target subtle underspecified checks. Overall, the D2C emulator aligned in only 3 out of 12 traces (as shown in Fig. 3), primarily due to two categories of issues:

(i) State errors: The D2C emulator fails to capture the important state variables, such as the *InstanceTenancy* or CreditSpecification attributes, rendering it incapable of testing certain resource update scenarios that could happen in production scenarios. It also missed state checks, like ensuring that no gateways/endpoints exist in a VPC before processing a DeleteVPC() call. Other errors include a lack of resource context, like allowing DNS hostnames to be enabled on a VPC where DNS support is disabled.

(ii) Transition errors. For instance, when tested with a StartInstances() call on an already-running instance, the D2C emulator failed silently; instead of returning the expected "IncorrectInstanceState" error, it returned a success code. This creates a dangerous state inconsistency that the DevOps program cannot detect. Furthermore, its check validation logic is shallow; while it can check for simple CIDR conflicts, it incorrectly allows the creation of a subnet with an invalid prefix size (e.g., /29). Other notable issues include failure to return the specific error codes required by client-side tooling.

Quantifying service complexity. Our approach can be used to quantify the complexity of cloud services by the number of state variables and transitions for a given state machine. We show the distribution for several AWS services in Figure 4: as

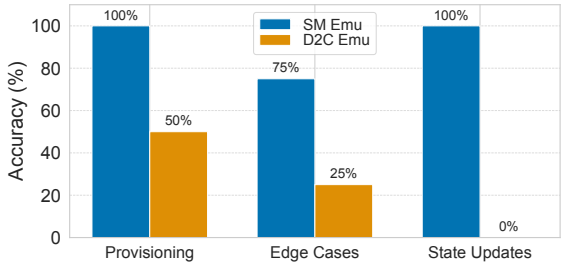

Figure 3: Accuracy of learned emulators across scenarios

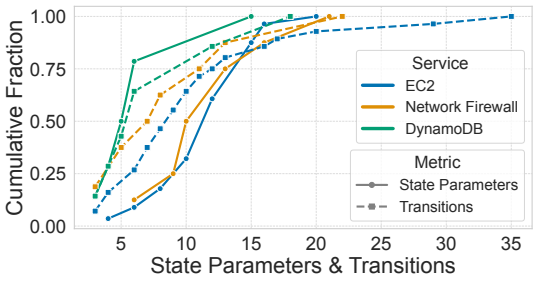

Figure 4: CDF of SM complexity across services

we can observe, the SMs in the EC2 service are more complex than others. Overall, our generated specs included 28 SMs for EC2, 8 for network firewall, and 7 for DynamoDB services. This complexity distribution, given that we can handle more complex services like EC2 well, is a positive sign indicating that our SM generation, for all services scaling across the cloud, is amenable to LLM-based generation.

Multi-cloud We replicated the same workflow on Azure and achieved comparable accuracy. The primary additional effort in generalizing to other cloud providers lies in documentation wrangling: whereas AWS offers a centralized repository of API definitions, Azure distributes definitions across resource-specific web pages. We need to adapt our information gathering workflow according to such provider-specific patterns.

6 CHALLENGES & LIMITATIONS

Underspecified Documentation. For resources with underspecified information (mainly the description of API specific behavior), our emulator relies solely on the alignment phase to gather concrete resource behavior. We're also exploring scraping information from cloud-specific resources (like for SDK) for further information about API behaviors and checks.

Alignment Completeness. The objective of the alignment phase is to harden the frequently executed paths. Given the unbounded number of programs that can be used for alignment, we do not aim to offer completeness guarantees.

7 RELATED WORK

Specification mining. Our work builds on a rich history of techniques for automatically extracting API specifications [32, 37, 42, 44, 54], especially those that leveraged the advent of LLMs to directly translate informal natural language comments and documents into checkable assertions [22, 52], temporal properties [21, 36], and other formal specifications [29, 39, 40]. While we draw heavily from these advances, most of them stop at inferring invariants for existing implementations, rather than generating end-to-end emulation code.

Within the domain of spec mining, our work is most related to active automata learning, or "model learning" [46, 47], where an algorithm interactively queries a black-box system to infer a state machine model of its behavior using traces or documentation. The most related work, Hermes [15] synthesizes FSMs from network protocol documents for security analysis, while other works have focused on attack synthesis [45], automated testing [34], or bug detection in protocols [23, 24]. To the best of our knowledge, our work is the first to use LLMs to translate API documentation into complete, executable SMs to emulate cloud services.

Cloud testing. Most cloud testing works have focused on finding bugs and vulnerabilities in the cloud service implementation itself, using stateful fuzzing [16, 17, 26, 28, 30, 48] (more so with LLMs [53]) or differential testing [27]. The closest work to ours [41] aims to find behavioral gaps between real cloud and emulators, but does not fix them in a principled manner. Our key contribution is to close the loop: use the discrepancies to refine our learned model itself. We take inspiration from the prior work on protocol reverse engineering [18, 38, 51] for efficient automated alignment.

8 SUMMARY

The development of modern cloud infrastructure needs emulation for better testing and validation. Historically, this task requires constant, error-prone manual efforts. We make a case for a new approach, using LLMs to learn the emulation logic from cloud documentation and automatically generating emulation code. Specifically, we argue for modeling cloud resources as state machines, paving the way toward principled code generation. This further opens up new avenues for AI-assisted cloud operations, e.g., for automated cloud testing, or building gym environments for cloud agents.

ACKNOWLEDGMENTS

We thank the anonymous reviewers and our shepherd, Barath Raghavan, for their insightful feedback. This work is partially supported by a VMware Early Career Faculty Grant, a Cisco grant, and NSF grants CNS-1942219, CNS-2106751, CNS-2107147, CNS-2214272.

REFERENCES

[1] Amazon elastic compute cloud - api reference. https://docs.aws.amazon.com/pdfs/AWSEC2/latest/APIReference/ec2-api.pdf. [Accessed 09-07-2025].

[2] Amazon Web Services — aboutamazon.com. https://www.aboutamazon.com/what-we-do/amazon-web-services. [Accessed 29-06-2025].

[3] API Design Guide | Google Cloud — cloud.google.com. https://cloud.google.com/apis/design. [Accessed 09-07-2025].

[4] AWS CloudFormation. https://aws.amazon.com/cloudformation/.

[5] cloudcontrolapi — aws.amazon.com. https://aws.amazon.com/cloudcontrolapi/. [Accessed 05-07-2025].

[6] LocalStack for AWS — localstack.cloud. https://www.localstack.cloud/localstack-for-aws. [Accessed 29-06-2025].

[7] localstack/localstack — github.com. https://github.com/localstack/localstack/issues. [Accessed 05-07-2025].

[8] Moto: Mock AWS Services &x2014; Moto 5.1.7.dev documentation — docs.getmoto.org. https://docs.getmoto.org/en/latest/. [Accessed 02-07-2025].

[9] Pulumi: Infrastructure as code in any programming language. https://www.pulumi.com/.

[10] Terraform by Hashicorp. https://www.terraform.io/.

[11] Use Azurite emulator for local Azure Storage development — learn.microsoft.com. https://learn.microsoft.com/en-us/azure/storage/common/storage-use-azurite?tabs=visual-studio%2Cblob-storage. [Accessed 02-07-2025].

[12] What is Terraform | Terraform | HashiCorp Developer — developer.hashicorp.com. https://developer.hashicorp.com/terraform/intro. [Accessed 05-07-2025].

[13] What's New at AWS – Cloud Innovation & News — aws.amazon.com. https://aws.amazon.com/new/. [Accessed 05-07-2025].

[14] How can i experiment with cloud (azure, aws, google, etc) without going broke? https://devops.stackexchange.com/questions/1002/how-can-i-experiment-with-cloud-azure-aws-google-etc-without-going-broke?noredirect=1&lq=1, 2021. [Accessed 09-07-2025].

[15] A. Al Ishtiaq, S. S. S. Das, S. M. M. Rashid, A. Ranjbar, K. Tu, T. Wu, Z. Song, W. Wang, M. Akon, R. Zhang, et al. Hermes: unlocking security analysis of cellular network protocols by synthesizing finite state machines from natural language specifications. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 4445–4462, 2024.

[16] V. Atlidakis, P. Godefroid, and M. Polishchuk. Restler: Stateful rest api fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 748–758. IEEE, 2019.

[17] V. Atlidakis, P. Godefroid, and M. Polishchuk. Checking security properties of cloud service rest apis. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 387–397. IEEE, 2020.

[18] G. Bossert, F. Guihéry, and G. Hiet. Towards automated protocol reverse engineering using semantic information. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*, pages 51–62, 2014.

[19] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. volume 8, pages 209–224, 01 2008.

[20] S. Chasins, A. Cheung, N. Crooks, A. Ghodsi, K. Goldberg, J. E. Gonzalez, J. M. Hellerstein, M. I. Jordan, A. D. Joseph, M. W. Mahoney, A. Parameswaran, D. Patterson, R. A. Popa, K. Sen, S. Shenker, D. Song, and I. Stoica. The sky above the clouds, 2022.

[21] M. Cosler, C. Hahn, D. Mendoza, F. Schmitt, and C. Trippel. nl2spec: Interactively translating unstructured natural language to temporal logics with large language models. In *International Conference on Computer Aided Verification*, pages 383–396. Springer, 2023.

[22] M. Endres, S. Fakhoury, S. Chakraborty, and S. K. Lahiri. Can large language models transform natural language intent into formal method postconditions? *Proceedings of the ACM on Software Engineering*, 1(FSE):1889–1912, 2024.

[23] P. Fiterau-Brostean, B. Jonsson, K. Sagonas, and F. Tåquist. Automata-based automated detection of state machine bugs in protocol implementations. In *NDSS*, 2023.

[24] P. Fiterău-Broştean, B. Jonsson, K. Sagonas, and F. Tåquist. Smbugfinder: An automated framework for testing protocol implementations for state machine bugs. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1866–1870, 2024.

[25] Y. Fu, E. Baker, Y. Ding, and Y. Chen. Constrained decoding for secure code generation, 2024.

[26] P. Godefroid, B.-Y. Huang, and M. Polishchuk. Intelligent rest api data fuzzing. In *Proceedings of the 28th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*, pages 725–736, 2020.

[27] P. Godefroid, D. Lehmann, and M. Polishchuk. Differential regression testing for rest apis. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 312–323, 2020.

[28] P. Godefroid, H. Peleg, and R. Singh. Learn&fuzz: Machine learning for input fuzzing. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 50–59. IEEE, 2017.

[29] N. Jain, S. Vaidyanath, A. Iyer, N. Natarajan, S. Parthasarathy, S. Rajamani, and R. Sharma. Jigsaw: Large language models meet program synthesis. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1219–1231, 2022.

[30] M. Kim, Q. Xin, S. Sinha, and A. Orso. Automated test generation for rest apis: No time to rest yet. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 289–301, 2022.

[31] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.

[32] I. T. Leong and R. Barbosa. Generation of oracles using natural language processing. In *2021 28th Asia-Pacific Software Engineering Conference Workshops (APSEC Workshops)*, pages 25–31. IEEE, 2021.

[33] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in neural information processing systems*, 33:9459–9474, 2020.

[34] H. Li, Z. Dong, S. Wang, H. Zhang, L. Shen, X. Peng, and D. She. Extracting formal specifications from documents using llms for test automation. In *2025 IEEE/ACM 33rd International Conference on Program Comprehension (ICPC)*, pages 1–12. IEEE Computer Society, 2025.

[35] B. Liskov and S. Zilles. Programming with abstract data types. In *Proceedings of the ACM SIGPLAN Symposium on Very High Level Languages*, page 50–59, New York, NY, USA, 1974. Association for Computing Machinery.

[36] J. X. Liu, Z. Yang, B. Schornstein, S. Liang, I. Idrees, S. Tellex, and A. Shah. Lang2ltl: Translating natural language commands to temporal specification with large language models. In *Workshop on Language and Robotics at CoRL 2022*, 2022.

[37] M. Liu, X. Peng, A. Marcus, C. Treude, X. Bai, G. Lyu, J. Xie, and X. Zhang. Learning-based extraction of first-order logic representations of api directives. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 491–502, 2021.

[38] Z. Luo, K. Liang, Y. Zhao, F. Wu, J. Yu, H. Shi, and Y. Jiang. Dynpre: Protocol reverse engineering via dynamic inference. In *Proc. NDSS*, pages 1–18, 2024.

[39] L. Ma, S. Liu, Y. Li, X. Xie, and L. Bu. Specgen: Automated generation of formal program specifications via large language models. *arXiv preprint arXiv:2401.08807*, 2024.

[40] S. Mandal, A. Chethan, V. Janfaza, S. Mahmud, T. A. Anderson, J. Turek, J. J. Tithi, and A. Muzahid. Large language models based automatic synthesis of software specifications. *arXiv preprint arXiv:2304.09181*, 2023.

[41] A. Mazhar, S. S. Alam, W. X. Zheng, Y. Chen, S. Nath, and T. Xu. Fidelity of cloud emulators: The imitation game of testing cloud-based software. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 614–614. IEEE Computer Society, 2025.

[42] M. Motwani and Y. Brun. Automatically generating precise oracles from structured natural language specifications. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 188–199, 2019.

[43] N. Mündler, J. He, H. Wang, K. Sen, D. Song, and M. Vechev. Type-constrained code generation with language models. *Proc. ACM Program. Lang.*, 9(PLDI), June 2025.

[44] H. A. Nguyen, R. Dyer, T. N. Nguyen, and H. Rajan. Mining preconditions of apis in large-scale code corpus. In *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*, pages 166–177, 2014.

[45] M. L. Pacheco, M. von Hippel, B. Weintraub, D. Goldwasser, and C. Nita-Rotaru. Automated attack synthesis by extracting finite state machines from protocol specification documents. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 51–68. IEEE, 2022.

[46] A. Tran Van, O. Levillain, and H. Debar. Mealy verifier: An automated, exhaustive, and explainable methodology for analyzing state machines in protocol implementations. In *Proceedings of the 19th International Conference on Availability, Reliability and Security*, pages 1–10, 2024.

[47] F. Vaandrager. Model learning. *Communications of the ACM*, 60(2):86–95, 2017.

[48] H. Wu, L. Xu, X. Niu, and C. Nie. Combinatorial testing of restful apis. In *Proceedings of the 44th International Conference on Software Engineering*, pages 426–437, 2022.

[49] Z. Yang, A. Bhatnagar, Y. Qiu, T. Miao, P. T. J. Kon, Y. Xiao, Y. Huang, M. Casado, and A. Chen. Cloud infrastructure management in the age of ai agents. *arXiv preprint arXiv:2506.12270*, 2025.

[50] Z. Yang, Z. Wu, M. Luo, W.-L. Chiang, R. Bhardwaj, W. Kwon, S. Zhuang, F. S. Luan, G. Mittal, S. Shenker, and I. Stoica. SkyPilot: An intercloud broker for sky computing. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 437–455, Boston, MA, Apr. 2023. USENIX Association.

[51] Y. Ye, Z. Zhang, F. Wang, X. Zhang, and D. Xu. Netplier: Probabilistic network protocol reverse engineering from message traces. In *NDSS*, 2021.

[52] J. Zhai, Y. Shi, M. Pan, G. Zhou, Y. Liu, C. Fang, S. Ma, L. Tan, and X. Zhang. C2s: translating natural language comments to formal program specifications. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, page 25–37, New York, NY, USA, 2020. Association for Computing Machinery.

[53] T. Zheng, J. Shao, J. Dai, S. Jiang, X. Chen, and C. Shen. Restless: Enhancing state-of-the-art rest api fuzzing with llms in cloud service computing. *IEEE Transactions on Services Computing*, 2024.

[54] H. Zhong, L. Zhang, T. Xie, and H. Mei. Inferring resource specifications from natural language api documentation. In *2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 307–318, 2009.