AutoPAC: Exploring LLMs for Automating Policy to Code Conversion in Business Organizations

Neha Chowdhary
IIT Kharagpur, India
nehachow.cse@kgpian.iitkgp.ac.in

Tanmoy Dutta
IIT Kharagpur, India
duttatanmoy834@gmail.com

Subhrendu Chattopadhyay
IDRBT, Hyderabad, India
subhrendu@idrbt.ac.in

Sandip Chakraborty
IIT Kharagpur, India
sandipc@cse.iitkgp.ac.in

***Abstract*—Managing systems and network policies for large-scale organizations is challenging for business process automation. Although Policy-as-code (PAC) platforms can ease the task of policy management by defining and executing systems and network policies in the form of programmable codes, converting existing organizational policies into PAC-compliant code is not straightforward due to the need for complex dependency resolutions across platforms and applications. On the other hand, policymakers/top management of a business prefer natural language (NL)-based policies that are easy to comprehend. This paper explores large language models (LLMs) to facilitate the automated conversion of NL-based policies to PAC-complaint code. We observe that public LLMs like ChatGPT need thorough multi-round prompt engineering to generate PAC policies. This concerns privacy and security as the organizational policies are sensitive business information. Consequently, we explore using a private and personalized setup, like private LLMs. Notably, we observe that existing personalized LLMs like PrivateGPT fail to understand the system-specific policy semantics. Consequently, we develop a framework called *AutoPAC*, which uses a micro-service architecture coupled with fine-tuned models to generate and validate PAC-complaint policies over a personalized LLM framework. An evaluation with more than 100 test cases indicates that the proposed framework effectively generates and validates PAC policies on the fly.**

***Index Terms*—Policy Management, LLM, PrivateGPT, OPA**

I. INTRODUCTION

Large-scale business organizations often need to comply with the policies of regulatory authorities. For example, information technology (IT) operations in banking are often controlled and managed by the policies defined by the central banking authorities of the corresponding country. As an example, the *Reserve Bank of India* (RBI) provides various policies for user authentication for regulatory compliance with cloud-based services[1]. However, in practice, these policies are modified from time to time based on technological upgrades, changes in the organizational frameworks, regulatory compliances, etc. Therefore, managing such policies in a large-scale organization becomes a challenging task, particularly when the IT operations are distributed and scaled across various administrative bodies. Notably, many business organizations have adopted "policy-as-code" (PAC)[2] [1]-based policy management and compliance systems where the policies are stored in the form of programs/code snippets. Among many, PAC provides ambiguity-free interpretation of policies, consistency across multiple systems, low-effort integration in heterogeneous systems, effective version management, and ease of producing compliance reports for internal and external audits.

However, till today, the adoption of PAC is majorly restricted to the IT industries[3], and business organizations face several challenges in migrating from existing systems to PAC[4]. First, existing PAC frameworks use special-purpose languages to represent policies. For example, the *Open Policy Agent* (OPA) [2] framework uses a query language called `REGO` that extends *datalog* (a declarative logic programming language) with policy-specific semantics to represent system-level policies. To utilize the full potential of the PAC frameworks, such PAC-specific languages have a learning curve that the policymakers (often senior management) need to understand and learn. Further, manually translating a policy schema to a PAC-specific language semantics is tedious and often encounters compliance issues, particularly when the policy spans multiple applications or infrastructure. Second, being one of the new technologies, there is a shortage of trained personnel for PAC systems; therefore, it is often a challenge to adopt PAC for enterprise-scale usage. In addition, such PAC frameworks usually do not have a central system to distribute and manage policies across various applications and infrastructures within an organization; consequently, it becomes difficult to figure out what parts of the environment are affected by the policies. Third, policy databases are vital artifacts often kept on secure premises to prevent data and information leakage. Therefore, outsourcing policy enforcement is usually not advisable [3].

To address the above challenges, in this paper, we propose an automation framework to ease the adoption of PAC frameworks for business organizations. The proposed framework, called *AutoPAC*, aims to enable the users to generate PAC-compatible policy (PAC-policy) without any programming experience, thus avoiding the learning curve for PAC-specific languages. Moreover, the framework can avoid the need for trained personnel. For this purpose, we use Generative Pre-trained (GPT) Large Language Models (LLM) to convert the natural language-specific policies into PAC-compatible policies. However, designing such a framework is non-trivial due to the following challenges.

[1] https://learn.microsoft.com/en-us/azure/governance/policy/samples/rbi-itf-banks-2016
[2] https://www.paloaltonetworks.com/cyberpedia/what-is-policy-as-code
[3] https://github.com/open-policy-agent/opa/blob/main/ADOPTERS.md
[4] https://discovery.hgdata.com/product/open-policy-agent

(1) **Organizational Security and Privacy:** The easiest pick to auto-generate PAC-complaint policies would be to use public LLMs, like ChatGPT, Gemini, etc., which have widely been explored for automated code generation [4], [5], [6]. However, such platforms have also been criticized heavily for leaking sensitive information [7], [8]. Further, such public models must be retrained for language and platform-specific semantics, mainly to generate codes from natural language descriptions [8]. Notably, Leaking information about organizational policies may lead to serious security vulnerabilities; therefore, retraining a public model might lead to privacy concerns. Moreover, GPT models require fine training with domain-specific annotated datasets, which are difficult to gather due to the confidentiality of organizational policies.

(2) **Lack of Existing Pre-trained Models:** Existing PAC[5][6][7] platforms often utilize domain-specific descriptive languages to define the policy semantics. Popular GPT platforms such as ChatGPT [9] or Gemini [10] do not perform well for PAC-policy generation tasks as identification of keywords and variables from the set of tokens in a declarative programming language is difficult. Moreover, learning programming semantics is another challenge the existing models face. On the other hand, training models from scratch is a time-consuming operation that may not be affordable for many organizations.

(3) **Verification of Generated PAC-policies:** Although LLMs may be used to generate PAC policies, the generated policies also need to be validated before their deployment. As discussed earlier, validating complex policies across applications and infrastructure is not straightforward. Therefore, policy generation and validation should come hand-in-hand to enable organizations to use the framework reliably and with minimal manual effort.

Considering the above challenges, the proposed framework, *AutoPAC*, utilizes a personalized and private GPT platform to generate PAC-policies from the natural language in a robust, scalable, and privacy-preserved manner. *AutoPAC* is micro-service based; thus, it can achieve rapid deployment using a CI/CD (continuous integration and continuous deployment) pipeline. We first create a dataset of publicly available policies, which has been used to train the model. The framework also requires various pre and post-data cleaning operations to fine-tune the results. We develop a Proof-of-Concept (PoC) implementation, which can be deployed on-premise of an organization to avoid data leakage and requires minimal resource footprints during training and deployment. We use the PoC implementation to perform thorough experiments on fine-tuning, hyper-parameter tuning & model selection. In addition, we develop a unit and integration testing pipeline for comprehensive testing ascertaining the sanity of the generated PAC-policies. Our experimental observations reveal that *AutoPAC* requires less than 2 seconds to generate individual policy. Upon further testing with 116 different test cases, we observed that *AutoPAC* provides almost 94% accuracy in generating the PAC-policies.

The rest of the paper is organized as follows. We first discuss the related literature to highlight existing code generation approaches through LLMs and their limitations in Section II. Section III describes some of our initial experimentation with the existing tools to explain the need for the proposed framework. Section IV and Section V describe the architectural considerations and implementation challenges of the proposed system. In Section VI, we describe the salient features of *AutoPAC* by experimental evaluation. Finally, Section VII concludes the paper.

[5]https://www.openpolicyagent.org/
[6]https://learn.microsoft.com/en-us/azure/governance/policy/overview
[7]https://www.pulumi.com/docs/using-pulumi/crossguard/get-started/

II. Related Work

This section discusses the related literature that has explored the use of LLMs for code generation and analysis with a highlight on their limitations for PAC-complaint code generation. In this context, we also summarize the works that target to automate policy generation and compliance.

A. Code Generation and Understanding with LLMs

Recent advancements in publicly available LLM tools and models are popular in various use cases, including automated code generation. Existing works [4], [5], [6] have described approaches to generate source codes from prompts. However, automated source-code generation scenarios are beneficial where end users are interested in generating programs that can save time and manual effort. The difficulty in understanding the generated code led to the use of LLMs for code explanation [11], [12] and code understanding [13], [14]. These works proposed using LLMs to develop web browser extensions or plugins to help users understand code snippets. Apart from these, LLMs were also utilized in scenarios where codes were incomplete or incomprehensible for code repair [15]. Notably, these works have primarily analyzed how LLMs can effectively assist code-writing and code-analysis tasks, although concerns about their accuracy in generating platform-specific codes exist. A case study [4] showcases various LLMs having maximum accuracy in terms of functional correctness of the programs as 76.2% with *GPT-4*, 67.1% with *Phind-CodeLlama*, and 64.6% with *WizardCoder-CodeLlama*. Moreover, the existing works have proposed approaches to fine-tune LLMs, particularly for functional languages like C, C++, Python, Java, etc., while to the best of our knowledge, no works consider policy-specific languages like `REGO`.

B. Domain-specific Code Generation

Specialized LLM-based solutions have emerged across various industries as business organizations realized the importance of LLMs in increasing efficiency by avoiding several manual efforts. For instance, LLMs are being used in

healthcare industries [16], [17], [18] ranging from day-to-day documentation, sorting feedback of patients, providing an interface for patients to fetch widely available medicinal data to create multiple-choice questions for exams [19] or as self-learning tools [20]. However, we observe that the creation of industry-specific solutions integrating LLMs faces a major hurdle of domain adaptation. Integration can be best utilized when LLMs are trained on the data specific to the domain, thus improving adaptability and accuracy. In contrast to using general-purpose models, we observe from recent literature [21], [22] that domain adaptation is a much more efficient approach for target organizations.

C. LLMs for Policy Generation

Building on the pretext that LLMs are being rapidly adapted for code generation, there emerged the idea to use them for policy generation [23] as well. Available literature emphasizes solutions that have tried to convert user intent into application-specific policies [24]. Robotics at Google [25] also presented using LLM models to produce *robot code*, i.e., planning a sequence of steps from natural language instructions. Among the existing works, a proprietary platform called ARMO [26] is capable of generating REGO-compatible PAC-policies to ensure Kubernetes security. However, we aimed to design a more generalized and useful tool for different business scenarios. Notably, ARMO relies on unit-test-based validation, which may not be feasible for large-scale automation. Moreover, privacy concern significantly hinders adoption of ARMO. Although there exist a few LLM-based natural language to program generators (like *NL2Code* [27], *CodeX* [28], *Code Llama* [29], *StarCoder* [30], *Tabnine*[8], etc.), training domain-specific languages like REGO requires significant resources. Furthermore, LLMs tend to suffer from *hallucinations* [31], [32], [33], [34] where they provide code snippets that are not complete or are incorrect. Notably, code completion tools are available for integration in various IDEs. *GitHub Copilot* [35], powered by OpenAI's *CodeX*, can be used for fetching code suggestions in IDEs as well as seeking chatbot inputs in code. *IntelliCode Compose* [36], a multilingual code completion tool, utilizes *GPT for Code* (*GPT-C*), which is a variant of the GPT-2 model trained from scratch on various source codes written in programming languages like Python, C#, JavaScript and TypeScript. While powerful and robust, this model is mainly based on the code structure and context already available in the code, not on natural languages. Hence, even if fine-tuned for policy-specific coding languages like REGO, its purpose is less suited for PAC-compliant code generation that needs specific semantics based on target infrastructure or applications.

III. Background and Pilot Study

Several NLP-based generative tools like ChatGPT [9], Gemini [10], etc., can generate source codes from the prompt expressing the objectives in natural language. Therefore, we first conduct a pilot study to try the prompt engineering approaches to generate PAC policies for a given scenario. Prompt

[8]https://www.tabnine.com/

(a) Query used with the expected outcome

```
1  User: Write REGO rules for Global admins and maintainers
       can run any query
2  Expected: allow {
3      object.type == "targeted_query"
4      subject.global_role == admin
5      action = run
6  }
7  allow {
8      object.type == "targeted_query"
9      subject.global_role == maintainer
10     action = run
11 }
12 allow {
13     object.type == "query"
14     subject.global_role == admin
15     action = run_new
16 }
17 allow {
18     object.type == "query"
19     subject.global_role == maintainer
20     action = run_new
21 }
```

(b) Response from ChatGPT which has partial similarity with the expected outcome

```
1  ChatGPT: allow_run_query {
2      action == "run_query"
3      user.role == "global_admin"
4  }
5          allow_run_query {
6      action == "run_query"
7      user.role == "maintainer"
8  }
```

(c) Subsequent prompts used to force ChatGPT to generate a compatible PAC for the target query

```
1  User (Prompt2): Use object.type==targeted_query or query
       and regenerate
2  User (Prompt3): Use subject.global_role=admin and action
      == run/run_new and regenerate
```

Fig. 1: Example Query and Responses from *ChatGPT-4*

engineering is essential in such cases as the public LLMs are trained with general-purpose texts. Therefore, they may need platform-specific guidelines through prompt engineering to generate accurate PAC policies. For this pilot study, we consider Role-based Access Control (RBAC) for Amazon-AWS Multi-tenant scenario as a use case[9]. This particular use case contains 2 tenants, each tenant having 5 users. The users access services from one micro-service, which provides "*read*", "*write*," and "*run*" services depending on the tenant category and user role. For the experimental purpose, we have used Open Policy Agent (OPA) [2] to represent the PAC policies. As mentioned earlier, OPA uses REGO [37] as the policy declarative language.

In the first pilot study, we used the ChatGPT V4-based conversational tool to generate a set of equivalent PAC policies. Although ChatGPT or similar public LLMs have privacy concerns, as we discussed before, our objective here is to check (a) how easy it is to generate a PAC policy from

[9]motivated by https://docs.aws.amazon.com/prescriptive-guidance/latest/saas-multitenant-api-access-authorization/opa-abac-rbac-examples.html

natural language descriptions and (b) how good the generated policies are in terms of platform compatibility. Notably, a compatible policy satisfies all the manually generated test cases during OPA testing under various platform constraints. The major challenge, in this case, is the construction of appropriate queries to obtain the compatible PAC policy as a response from the ChatGPT. We observed that, although the ChatGPT-generated responses (see Figure 1b) closely match with the expected outcome (see Figure 1a), it requires fine-tuning to obtain a compatible PAC. Using standard prompt engineering techniques[10], on average, it takes ≈ 4 prompts to get the compatible rules if the queries are invoked under the same topic/chat thread [11]. For example, the ChatGPT response generates a compatible PAC after using the 2 more queries subsequently in the same thread (Listed in Figure 1c). However, one primary concern in using ChatGPT for these types of tasks is related to privacy (as discussed in **Section I:**(2)), which leads to information leakage during this prompt engineering. For example, in the given use case (Figure 1), the system admin needs to disclose the exact fields and variables (e.g., `subject.global_role, action`) used in their system.

To avoid such issues with data privacy, we next explore whether private LLMs like PrivateGPT [38] can work as an alternative. Consequently, in our second pilot study, we use PrivateGPT which utilizes a light-weight pre-trained model (GGML [39]) and can be custom-trained (*known as "ingestion"*) on particular topics. We have deployed the model using a local server with 8 CPU cores and 32GB of RAM as recommended. We have used the AWS documentation (aforementioned) to ingest the model. However, we observed that the results were not always suitable. For example, Figure 2c presents one generated rule for a particular query string as given in Figure 2a. The PrivateGPT-generated PAC is too generic and requires significant training samples to fine-tune it to obtain the desired results. Another observation is that PrivateGPT generates different outputs while querying with the exact prompt multiple times, and the generated outputs are sometimes incoherent[12] as shown in Figure 2c. Moreover, the output generation often takes minutes to respond, and the underlying micro-services crash too frequently, which motivated us to come up with a stable architecture that can provide better responses.

Take Away and Design Goals. The pilot study shows that while public LLMs like ChatGPT need prompt engineering and are thus vulnerable to privacy attacks, private LLMs like PrivateGPT does not provide a robust, consistent output. Consequently, we target to develop a lightweight pluggable framework for domain-specific adaptations over PrivateGPT to achieve automated PAC policy generation. Accordingly, we consider the following design goals for the proposed architecture;

G1. The proposed framework should need minimum domain adaptation and retraining for platform-specific PAC policy generation without a requirement of explicit prompt engineering.

G2. The proposed framework should be scalable and easily deployable following the CI/CD pipeline.

G3. The proposed framework should also be able to validate the generated PAC policies to make them deployment-ready with minimal effort.

We next discuss the details of the proposed framework architecture.

[10]https://www.promptingguide.ai/techniques

[11]https://community.openai.com/t/chat-thread-functionality/60852

[12]The root causes are investigated and explained in Section V.

(a) Query used with the expected outcome

```
1 User: Write a REGO code for AWS Role-Based Access
      Control (RBAC) of users.
2 Expected:  allow {
3     input.action == "read" or input.action == "write"
4     input.repo == "aws"
5     input.role == "admin"}
```

(b) Sample Output-1: Output is incorrect but shows some similarity with the expected outcome

```
1 PrivateGPT:
2 allow {
3     resource : "aws_s3"
4     }if (resource.bucket == 'admin-users' and action in
       ['read', 'write'])
```

(c) Sample Output-2: Output consisting of meaningless strings like `o<l,vl,vr,ekr,3>=`

```
1  PrivateGPT:
2  allow if { o<l, vl 3>= aws, vr 3>= ekr }
```

Fig. 2: Example Responses from *PrivateGPT*.

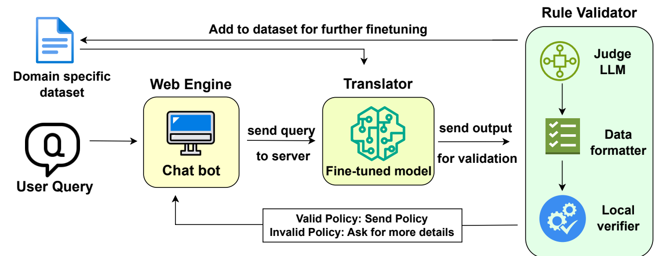

Fig. 3: *AutoPAC*: Proposed Architecture.

IV. Platform Architecture

The *AutoPAC* architecture is sub-divided into 3 major micro-services (as shown in Figure 3): (i) *Web Engine*, (ii) *Translator*, and (iii) *Rule Validator*. The micro-service architecture helps the business administration deploy the platform seamlessly with optimized resource footprints with load-balancing capabilities, thus lowering the response time. The *Web Engine* micro-service is a web-based conversational front-end (chatbot) that provides the user an interface through which they put forth the query in the Natural Language (NL). This component complements the challenge of requiring a complicated learning curve by providing a simple and easily

understandable platform for the user to query even in non-technical language, thereby reducing the need for expertise in policy language to create organizational policies. The query submitted by the user is processed by the *Translator* back-end micro-service, which is connected with the Web Engine over `REST` API. The *Translator* uses a fine-tuned LLM trained explicitly in the PAC-policy language. Upon receiving the user queries in the natural language, the *Translator* generates the corresponding PAC-policies as the output. Since we use LLMs to generate PAC-policies, the output may not always be correct. Therefore, *AutoPAC* uses a separate *Rule Validator* micro-service which validates the generated PAC-policies. The *Rule Validator* is sub-divided into 3 modules as follows: (a) Judge LLM, (b) Data Formatter, and (c) Local Verifier and the working principles of the system are described as follows.

For a system that is supposed to encounter a large number of queries, curating unit test cases for each of them and maintaining test coverage for each generated policy may not be a scalable task. Therefore, the generated PAC-policies are coarsely filtered using an auxiliary LLM termed as *Judge LLM* [40]. This module can verify the output policies depending on various factors such as completeness, legibility, etc. Post coarse filtration, we use fine-grained filtration using subsequent modules. During fine-grained filtration, we use *Data Formatter* module, which formats the *valid* outputs of Judge-LLM using a PAC-policy language-specific "linter"[13]. A linter is a tool used to identify and correct structural errors and stylistic constructs in code. Here, the primary objective of this module is to perform syntactic analysis of PAC-policies, and therefore, it improves the code readability by refactoring them. The formatted PAC-policies are further verified by the user-provided test cases inside a *Local Verifier* module, which contains a set of unit test cases for the given query provided by the user.

The overall architectural design also helps us to future-proof the system. It allows future adaptors to employ a CI/CD pipeline to fine-tune it further. To show the capabilities of this framework, we develop a Proof of Concept (PoC) implementation as discussed next.

V. *AutoPAC*-OPA: PoC Platform Implementation

In order to develop the PoC implementation, we have to custom-train the *AutoPAC* with a particular PAC-policy-specific language. In our implementation, we have identified the popular PAC platform called OPA [2] that uses `REGO` [37] as the policy declarative language. Although OPA does not provide policy enforcement services, it is compatible with multiple open-source projects [41] that can enforce the policies, such as *Kubernetes, Envoy, Express, Terraform, Linux-PAM,* etc. We have created a dataset using publicly available OPA policies to custom-train our framework, as discussed next.

A. Domain-specific Dataset

The created dataset contains `REGO` policies for the OPA framework and their annotations, which constitute a brief description of the rules. To obtain the `REGO` policies, we have selected 19 publicly available repositories (listed in Table I) from `GitHub`. The collected samples mainly contain different types of RBAC and Attribute-based Access Control (ABAC) policies. We have used our customized Python programs to clean the dataset, which includes (a) segregation of comments from the policy code and (b) automated annotations based on the available comments in the program. We used human annotation to label the dataset in a few situations where the comments were unavailable. Finally, we have extracted 1100 such labeled PAC-policies repositories consisting of ≈ 50 rules per repository. In total, the dataset contains approx $79,206$ word tokens. During our customized model training phase (described next), we have sub-divided the dataset into 3 divisions (80%, 10%, and 10%) and used them for training, validation, and testing datasets, respectively.

TABLE I: Training Dataset Sourced from Github Repositories

Repository	Link
18F/fleet	https://github.com/18F/fleet
CptOfEvilMinions/fleet	https://github.com/CptOfEvilMinions/fleet
DominusKelvin/fleet	https://github.com/DominusKelvin/fleet
KarlatIwoca/fleet	https://github.com/KarlatIwoca/fleet
TheDyingYAK/splunk_siem	https://github.com/TheDyingYAK/splunk_siem
blazman/fleet	https://github.com/blazman/fleet
empayre/fleet	https://github.com/empayre/fleet
erikng/fleet	https://github.com/erikng/fleet
fleetdm/fleet	https://github.com/fleetdm/fleet
groob/fleetdm-fleet	https://github.com/groob/fleetdm-fleet
kapawit/fleet	https://github.com/kapawit/fleet
kolbeface/fleet	https://github.com/kolbeface/fleet
kyle-humane/fleet	https://github.com/kyle-humane/fleet
lizthegrey/fleet	https://github.com/lizthegrey/fleet
noahtalerman/fleet-1	https://github.com/noahtalerman/fleet-1
stephanmiehe/fleet-1	https://github.com/stephanmiehe/fleet-1
weswhet/fleet	https://github.com/weswhet/fleet
y0zg/fleet	https://github.com/y0zg/fleet
yonnym/fleet	https://github.com/yonnym/fleet

B. Training System

We used a workstation with Ubuntu $20.04.6$ and Linux kernel $5.15.0-83-$generic for custom training purposes. The system is equipped with 20 GB RAM, 16 CPU cores, and $12GB$ GPU memory with CUDA 12.2 support. We have used Python-3.10.14 with Conda-24.1.2 for the LLM model training. In this paper, we have adopted the transfer learning approach where we have used a pre-trained model and fine-trained it further with a supervised `REGO`-specific dataset. The functional validation and accuracy of the output is measured by the *Rule Validator* which tests the rules on their syntactical validity and legibility as well as their functional efficiency and provides a score based on the number of test cases passed.

C. Model Selection

In the case of any standard LLM-based transfer learning approach, the choice of the initial LLM model is crucial. In this work, we have tested with 3 existing models as discussed next.

1) ***gpt4all:*** As mentioned in Section III, our initial testing with `gpt4all-ggml` [39] was not satisfactory, and the results generated were not stable most of the time. Upon further investigation, we found that this discrepancy is attributed to the

[13] https://www.openpolicyagent.org/integrations/regal/

inefficient embedding vector representation, which represents the relationship between the word and its contextual meaning of the model. During the training phase, the *vector store*, which refers to a mechanism used for efficiently storing and retrieving vector representations of the data (typically embeddings), is getting updated with the metadata used for training. After the training we observed that various program-specific identifiers were misrecognized such as special symbols like three "==", identifiers like "$vlan$" were incorrectly identified as "3 >=" and "vl" respectively which leads us to the conclusion that the embedding was improper and sometimes arbitrary. Our analysis reveals that the primary reason lies in the Byte-Pair encoding (BPE) scheme used by the model, which often breaks the identifiers used in the source code into sub-strings, which leads to a loss of contextual information that is important for PAC policies. The generated response lacks sanity even after ingesting documents containing multiple `REGO` rules. Based on the intuition that `gpt4all-ggml` belongs to the GPT family, which uses the decoder-only transformer architecture, it lacks deep comprehension of the entire input sequence.

*2) **t5-base**:* Next, we focused on the models that use decoder-encoder transformer architecture to overcome the above issue during model training. Loosely categorizing the PAC-policies as text data, we used *Text-to-Text Transfer Transformer* (`t5-base`) [42] as the base model for the `Translator`. We observed that even though this model uses BPE as used in the previous case, it could extract better contextual information from the submitted dataset of `REGO` codes. However, as the `t5-base` model uses Masked Span Prediction (MSP)[14], it is more sensitive towards declarative languages. Given that the `t5-base` has been predominantly trained on natural language corpus, it becomes less sensitive to the keywords used in programming languages. This can be attributed to the fact that `t5-base`'s pre-training involves *span corruption*, where the tokens of arbitrary lengths are masked, and the model is tasked with predicting them, thus improving context awareness. In the case of source codes, language-specific syntax, data types, control structures, or other structural features of codes need to be considered, which the `t5-base` does consider effectively. However, a few programming language-specific vital tokens (such as colons, parenthesis, identifiers, etc.) were incorrectly identified due to the partial masking of tokens. As an effect, this model disrupts the code structure during training, resulting in the generation of non-compatible `REGO` rules observed in Figure 4.

*3) **codet5-small**:* To overcome the above issue, we finally selected `CodeT5-small`[15] which has been pre-trained in programming languages like `Python`, `C++`, `JAVA`, etc., and was able to learn the program structure of `REGO` including identifiers, keywords, colons, parenthesis, etc. `CodeT5-small` employs a three-step pre-training [43] as follows; (a) Identifier-aware MSP with whole word masking, (b) Identifier Tagging, and (c) Masked Identifier Prediction

[14]Tokens are masked arbitrarily over their lengths, often partially, and is used to generate the contextual information

[15]https://huggingface.co/Salesforce/codet5-small

```
1 User: Write a REGO rule to provide admin access to read and
      write if data.repo==aws
2 Expected:  allow {
3     input.action == "read" or input.action == "write"
4     input.repo == "aws"
5     input.role == "admin"}
6 PrivateGPT: tm09in  if dr==aws, c=c and ixrnsx6k. kr.
```

Fig. 4: Sample output using `t5-base` for the given query generates meaningless characters like `tm09in, ixrnsx6k, kr` due to incompatible encoding mechanism.

(MIP). *Identifier-aware MSP* ensures that complete words (especially identifiers) and significant tokens in the code are adequately masked. The approach to treating identifiers as whole units rather than partial tokens prevents disruption of the code structure and allows the model to better understand and predict identifiers in context. On the other hand, *Identifier Tagging* enhances the model's understanding of identifiers by marking them separately from the used variables in an approach similar to syntax highlighting. *Masked Identifier Prediction* instructs the model to precisely mask the identifiers (e.g., variable names, function names) and it is trained to predict these masked identifiers. Specialized masking helps improve the model's ability to anticipate and understand the roles of different identifiers in the code. Our experimental observation reveals that, with the fine-tuned `CodeT5-small` model, the generated PAC-policies codes become acceptably sane most of the time. For example, in Figure 5, the generated `REGO` codes are complete with parenthesis, semicolons, etc. Therefore, for the rest of the PoC implementation, we have continued to use `CodeT5-small` as the selected model.

```
1 User: Write a REGO rule to provide admin access to read and
      write.
2 Expected:  allow {
3     action == "read" or action == "write"
4     repo == "aws"
5     role == "admin"}
6 PrivateGPT: allow {
7     global_role == admin
8     action == write/read
9     repo == aws}
```

Fig. 5: Sample output generated using `CodeT5-small` which closely matches with the expected outcome.

D. Model Fine-Tuning

With the selected model, we started the domain-specific fine-tuning process. During the fine-tuning procedure using standard transfer learning approaches, we have tokenized our dataset (as described in Section V-A) into a maximum of 128 input and output tokens, which were decided based on the highest length of the PAC-policy available in the dataset. We have used `cross-entropy` as the loss function and `decoupled weight-decay Adam optimizer`. The training loss function is defined as the cross-entropy loss over the model's output from the training dataset. In contrast, the validation loss function corresponds to

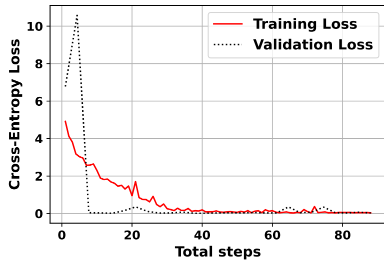

Fig. 6: Cross Entropy Loss during training.

TABLE II: Hyper-parameters used during fine-tuning

Hyper-parameters	Value	Hyper-parameters	Value
Min. Input Tokens	128	Max. Output Tokens	128
Training batch size	8	Validation batch size	2
Epochs	8		

the output from the validation dataset. Our observed training and validation losses per step are shown in Figure 6. We also observed that training over 8 epochs leads to validation loss greater than the training loss. Therefore, we keep 8 epochs for training to avoid over-fitting. From our dataset size of 1100 samples, we choose 80% (880 samples) for training. The batch size 8 indicates that during each iteration, the model processes 8 samples at a time, which amounts to 110 iterations per epoch, totaling 880 iterations over 8 epochs. We choose to use a training batch of size 8 and a validation batch of size 2 based on optimizing training over the available CPU and GPU resources and considering the prevention of overfitting the model to training data. The remaining hyper-parameters were selected based on experimentation and are listed in Table II. Once the training is complete, the generated fine-tuned model is used in our implemented *AutoPAC*.

E. Implementation of Components

We have deployed the individual micro-services using `Docker` to achieve micro-servicification. The details about the components are as follows.

1) Web Engine: The *Web Engine* component uses `ReactJS` to provide the chatbot interfaces to the end users over the web interface. We have observed that this component can behave well with 1 vCPU core and 1GB of RAM. The end-users' queries are forwarded to the Translator, and the response is displayed in the GUI.

2) Translator: The Translator engine has a custom Python-Flask server, allowing other components to access the services via the `REST` API. The incoming user queries to the Flask server are resolved via internal API calls, resulting in the generated `REGO` rules being forwarded to the *Validator* module via the REST API. The post-validated policies are eventually sent as a response to the *Web Engine*. The component is resource-provisioned for optimum performance using 2 CPU cores and 3GB of RAM.

3) Rule Validator: The output is further processed through an on-premise *Rule Validator* to avoid incomplete and irrelevant results generated by the *Translator*. The generated

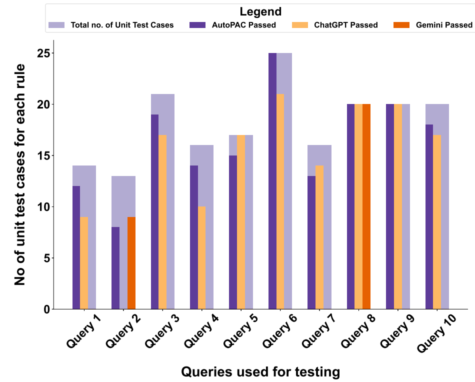

Fig. 7: Unit Testing Comparison of *AutoPAC*-OPA, ChatGPT and Gemini in *Local Verifier*.

`REGO` policies were sent to the publicly available LLMs like ChatGPT [9], which acts as the *Judge LLM*. Although these LLMs themselves can not ensure the guaranteed correctness of the generated PAC-policies, it could still be used to separate out the same PAC-policies from the malformed/incorrect ones. In this case, we emphasize that this testing methodology during fine-tuning does not affect organizational security as the training dataset is publicly available and does not leak sensitive organization-specific policies. Moreover, the user can turn off this verification process and solely rely on the subsequent filtration processes using Linter and unit test cases.

Post Judge-LLM filtration, the generated programs are formatted using the OPA linter module, which checks for the policy syntax and refactors the code into a readable format. In the case of user-provided unit-test cases, the rule is further tested using an OPA unit testing framework, which provides the number of passed test cases along with the generated `REGO` rules.

VI. Experimental Results

We have tested the efficacy of *AutoPAC*-OPA via two mechanisms: (a) Unit test-case validation and (b) using third-party open source LLMs such as ChatGPT-4 and Gemini. For the unit-test-based evaluation, we have shortlisted 10 annotations from our testing dataset (as listed in Table IV) and transformed the annotations into queries. We have manually created $15-25$ unit test cases for each query depending on the complexity of the expected `REGO` rules. We have used the *Local Verifier* module to test *AutoPAC*-OPA and compared its accuracy with the existing ChatGPT and Gemini-generated rules (see Table III). In summary, the results (see Figure 7) reveal that, for 6 queries out of the 10 test queries, *AutoPAC*-OPA provided better performance than ChatGPT and Gemini. On the other hand, Gemini and ChatGPT outperform *AutoPAC*-OPA in one (Query-2) and two (Query-5&7) test queries, respectively. For one query (Query-8), *AutoPAC*-OPA, ChatGPT, and Gemini gave equal accuracy.

As generating exhaustive unit testing test cases for a more extensive test dataset is infeasible, we have used ChatGPT-4 to verify the results. We have used Selenium scripts to

TABLE III: Responses Comparison: For the given query, responses generated by each tool are presented along the passed unit test-cases out of 14 test-cases

User: Any user can read/write their own sessions

AutoPAC

```
allow {
object.type == "session"
subject.global_role == admin
action == [read, write][_]
}
```

Gemini

```
allow {
    input.object.type == "session"
    input.subject.id == input.object.
     user_id
    input.action == "read" ||
    input.action == "write"
}
```

ChatGPT

```
allow {
    input.action == "read"
    input.object.type == "session"
}
allow {
    input.action == "write"
    input.object.type == "session"
}
```

AutoPAC	Gemini	ChatGPT
Test cases passed:(12/14)	Test cases passed:(0/14)	Test cases passed:(9/14)

TABLE IV: List of Queries Used for Unit Testing

Query Number	Query Description
Query 1	Any user can read/write own session.
Query 2	Any logged in user can read global config.
Query 3	Only global admins and maintainers can read/write packs.
Query 4	Team maintainers can read for appropriate teams.
Query 5	Team admin, maintainer, observer_plus and observer running an observers_can_run query that belongs to their team and there are no target teams.
Query 6	Global admins, maintainers, observers and observer_plus can read all software.
Query 7	Global gitops can write MDM Apple settings.
Query 8	Global admins can read and write Apple devices.
Query 9	If role is observer on any team, can read team details.
Query 10	Only global admins and maintainers can read and write labels.

TABLE V: Accuracy of *AutoPAC*-OPA using ChatGPT

Valid	Invalid	Incomplete
94%	3.4%	2.6%

TABLE VI: Similarity Score with Gold Standard

Sample ID	1	2	3	4
Score	0.95	0.95	0.36	0.36

automate the rule checking for this purpose and observed that, out of 116 generated REGO rules, 94% were approved by ChatGPT. Among others, 3.4% were identified as wrong, and 2.6% resulted in syntactically correct but incomplete rules without any conditions to check for (see Table V). Upon further investigation, we observe that (see Table VI) the cosine similarity score between the Gold Standard (from the annotated Dataset) and the generated REGO rule is very similar for 2 out of 4 failed queries. The generated rules and the corresponding Gold Standard rule are presented in Table VII, which justifies that, even among the failed queries (as evaluated by Judge-LLM), there are a few cases that require minor token alteration to fix the problem.

During the experimentation, we also observed that the responses for the queries require 1.23 ± 0.27 seconds to generate the rules. Moreover, the average CPU and memory utilization of the Translator micro-service is approximately 0.13% and 2.2GB, which justifies the light-weighted nature of the implementation.

TABLE VII: Output Comparison for Failed Queries

AutoPAC-OPA generated	From Dataset	Score
team_role(subject, subject.teams[_].id) == maintainer action == run }	team_role(subject, subject.teams[_].id) == admin action == run_new }	0.95
team_role(subject, subject.teams[_].id) == maintainer action == run }	team_role(subject, subject.teams[_].id) == admin action == run_new }	0.95
team_role(subject, team_id) == [admin, maintainer][_] }	team_role(subject, subject.teams[_].id) == [admin,maintainer][_] action == run_new }	0.36
team_role(subject, team_id) == [admin, maintainer][_] }	team_role(subject, subject.teams[_].id) == [admin,maintainer][_] action == run_new }	0.36

VII. Conclusion and Future Work

In this work, we have proposed *AutoPAC*, a framework to convert natural language-based policies into PAC-policies with the help of LLM. We have created an annotated dataset based on publicly available PAC-policies to fine-tune the LLM. The proposed architecture provides a lightweight, robust, and practical approach to utilizing private LLMs for infrastructure-specific PAC policy generation by combining microservice-based deployment architecture with organization-driven query format processing at its core. As a future extension of this work, we plan to make the setup more scalable. We also plan to test with the more heavyweight models like *CodeX* and *Code Llama* in the Translator architecture and measure if we can seek more accuracy regarding rule structure while optimizing resource consumption. In scenarios where *AutoPAC* gives slightly inaccurate PAC-policies, we plan to conduct a detailed analysis of the similarity using various metrics in future updates. Nevertheless, to the best of our knowledge, *AutoPAC* provides first-of-its-kind private LLM architecture to automate organization and platform-specific PAC policy generation based on natural language queries, which can help organizational governance and IT services automation.

References

[1] M. S. I. Shamim, F. A. Bhuiyan, and A. Rahman, "XI Commandments of Kubernetes Security: A Systematization of Knowledge Related to Kubernetes Security Practices," *IEEE Secure Development*, pp. 58–64, 2020.

[2] OPA, "Open Policy Agent," 2016, [accessed November 13, 2024]. [Online]. Available: https://www.openpolicyagent.org/

[3] Reserve Bank of India, "Master Direction on Outsourcing of Information Technology Services," https://www.rbi.org.in/Scripts/BS_ViewMasDirections.aspx?id=12486, April 2023, [accessed November 13, 2024].

[4] T. Coignion, C. Quinton, and R. Rouvoy, "A Performance Study of LLM-Generated Code on Leetcode," in *International Conference on Evaluation and Assessment in Software Engineering*, 2024.

[5] L. Zhong and Z. Wang, "Can LLM Replace Stack Overflow? A Study on Robustness and Reliability of Large Language Model Code Generation," in *Association for the Advancement of Artificial Intelligence Conference on Artificial Intelligence*, vol. 38, no. 19, 2024, pp. 21 841–21 849.

[6] B. Idrisov and T. Schlippe, "Program Code Generation with Generative AIs," *Algorithms*, vol. 17, no. 2, p. 62, 2024.

[7] Z. Zhang, M. Jia, H.-P. Lee, B. Yao, S. Das, A. Lerner, D. Wang, and T. Li, ""It's a Fair Game", or Is It? Examining How Users Navigate Disclosure Risks and Benefits When Using LLM-Based Conversational Agents," in *ACM Conference on Human Factors in Computing Systems*, 2024, pp. 1–26.

[8] M. L. Siddiq, L. Roney, J. Zhang, and J. C. D. S. Santos, "Quality Assessment of ChatGPT Generated Code and their Use by Developers," in *International Conference on Mining Software Repositories*, 2024, pp. 152–156.

[9] OpenAI, "ChatGPT," https://chat.openai.com/, November 2022, [accessed November 13, 2024].

[10] Google LLC, "Gemini," https://gemini.google.com/app, March 2023, [accessed November 13, 2024].

[11] J. Leinonen, P. Denny, S. MacNeil, S. Sarsa, S. Bernstein, J. Kim, A. Tran, and A. Hellas, "Comparing Code Explanations Created by Students and Large Language Models," in *ACM Innovation and Technology in Computer Science Education*, 2023, pp. 124–130.

[12] P. Vaithilingam, T. Zhang, and E. L. Glassman, "Expectation vs. Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models," in *Conference on Human Factors in Computing Systems Extended Abstracts*, 2022, pp. 1–7.

[13] D. Nam, A. Macvean, V. Hellendoorn, B. Vasilescu, and B. Myers, "Using an LLM to Help with Code Understanding," in *International Conference on Software Engineering*, 2024, pp. 1–13.

[14] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, "Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation," *Neural Information Processing Systems*, vol. 36, 2024.

[15] M. Jin, S. Shahriar, M. Tufano, X. Shi, S. Lu, N. Sundaresan, and A. Svyatkovskiy, "Inferfix: End-to-end Program Repair with LLMs," in *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 1646–1656.

[16] N. Rane, A. Tawde, S. Choudhary, and J. Rane, "Contribution and Performance of ChatGPT and other Large Language Models (LLM) for Scientific and Research Advancements: A Double-Edged Sword," *International Research Journal of Modernization in Engineering Technology and Science*, vol. 5, no. 10, pp. 875–899, 2023.

[17] H. Ali, J. Qadir, T. Alam, M. Househ, and Z. Shah, "ChatGPT and Large Language Models in Healthcare: Opportunities and Risks," in *IEEE International Conference on Artificial Intelligence, Blockchain, and Internet of Things*, 2023, pp. 1–4.

[18] S. Pal, M. Bhattacharya, S. S. Lee, and C. Chakraborty, "A Domain-Specific Next-Generation Large Language Model (LLM) or ChatGPT is Required for Biomedical Engineering and Research," *Annals of Biomedical Engineering*, vol. 52, pp. 451–454, 2024.

[19] Y. Artsi, V. Sorin, E. Konen, B. S. Glicksberg, G. Nadkarni, and E. Klang, "Large Language Models for Generating Medical Examinations: Systematic Review," *BioMed Central Medical Education*, vol. 24, no. 1, p. 354, 2024.

[20] W. Choi, "Assessment of the Capacity of ChatGPT as a Self-Learning Tool in Medical Pharmacology: A Study Using MCQs," *BioMed Central Medical Education*, vol. 23, no. 1, p. 864, 2023.

[21] N. Zhang, Y. Liu, X. Zhao, W. Cheng, R. Bao, R. Zhang, P. Mitra, and H. Chen, "Pruning as a Domain-Specific LLM Extractor," in *Findings of the Association for Computational Linguistics*, 2024, pp. 1417–1428.

[22] Y. Ge, W. Hua, K. Mei, J. Ji, J. Tan, S. Xu, Z. Li, and Y. Zhang, "OpenAGI: When LLM Meets Domain Experts," in *Neural Information Processing Systems Datasets and Benchmarks Track*, 2023.

[23] D. Ferrin, "How LLMs Can Simplify Writing Company Policies and Procedures," 2023, [accessed November 13, 2024]. [Online]. Available: https://projectinggroup.com/how-llms-can-simplify-writing-company-policies-and-procedures/

[24] K. Dzeparoska, J. Lin, A. Tizghadam, and A. Leon-Garcia, "LLM-Based Policy Generation for Intent-Based Management of Applications," in *International Conference on Network and Service Management*, 2023, pp. 1–7.

[25] J. Liang, W. Huang, F. Xia, P. Xu, K. Hausman, B. Ichter, P. Florence, and A. Zeng, "Code as Policies: Language Model Programs for Embodied Control," in *IEEE International Conference on Robotics and Automation*, 2023, pp. 9493–9500.

[26] ARMO, "ARMO and ChatGPT – Create Custom Controls Faster," 2023, [accessed November 13, 2024]. [Online]. Available: https://www.armosec.io/blog/armo-chatgpt-create-custom-controls-faster/

[27] F. F. Xu, B. Vasilescu, and G. Neubig, "In-IDE Code Generation from Natural Language: Promise and Challenges," in *ACM Transactions on Software Engineering and Methodology*, 2021.

[28] M. Chen and J. Tworek, "Evaluating Large Language Models Trained on Code," *IEEE/ACM International Conference on Software Engineering*, 2021.

[29] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez *et al.*, "Code Llama: Open Foundation Models for Code," *International Workshop on Large Language Models for Code*, 2023.

[30] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim *et al.*, "StarCoder: May the Source Be With You!" *Transactions on Machine Learning Research*, 2023.

[31] A. Eghbali and M. Pradel, "De-Hallucinator: Mitigating LLM Hallucinations in Code Generation Tasks via Iterative Grounding," *arXiv preprint arXiv:2401.01701*, 2024.

[32] F. Leiser, S. Eckhardt, M. Knaeble, A. Maedche, G. Schwabe, and A. Sunyaev, "From ChatGPT to FactGPT: A Participatory Design Study to Mitigate the Effects of Large Language Model Hallucinations on Users," in *ACM Mensch und Computer*, 2023, pp. 81–90.

[33] A. Martino, M. Iannelli, and C. Truong, "Knowledge Injection to Counter Large Language Model (LLM) Hallucination," in *European Semantic Web Conference*. Springer, 2023, pp. 182–185.

[34] Z. Ji, T. YU, Y. Xu, N. Lee, E. Ishii, and P. Fung, "Towards Mitigating LLM Hallucination via Self Reflection," in *Empirical Methods in Natural Language Processing*, 2023, pp. 1827–1843.

[35] "What is GitHub Copilot?" 2023, [accessed November 13, 2024]. [Online]. Available: https://docs.github.com/en/copilot/about-github-copilot/what-is-github-copilot

[36] A. Svyatkovskiy, S. K. Deng, S. Fu, and N. Sundaresan, "IntelliCode Compose: Code Generation Using Transformer," in *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1433–1443.

[37] REGO, "Open Policy Agent — Policy Language," 2016, [accessed November 13, 2024]. [Online]. Available: https://www.openpolicyagent.org/docs/latest/policy-language/

[38] "Private GPT," https://hub.docker.com/r/rattydave/privategpt, [accessed November 13, 2024].

[39] R. Marella, "gpt4all-j-ggml," https://huggingface.co/rustformers/gpt4all-j-ggml, 2023, [accessed November 13, 2024].

[40] L. Zheng and W.-L. C. et. al., "Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena," in *International Conference on Neural Information Processing Systems*, vol. 36, 2023, pp. 46 595–46 623.

[41] Open Policy Agent, "OPA Eco-system REST API Integrations," https://www.openpolicyagent.org/ecosystem/rest-api-integration/, April 2023, [accessed November 13, 2024].

[42] H. W. Chung and L. H. et. al., "Scaling Instruction-Finetuned Language Models," in *Journal of Machine Learning Research*, vol. 25, no. 70, 2022, pp. 1–53.

[43] Y. Wang, W. Wang, S. Joty, and S. Hoi, "CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation," in *Conference on Empirical Methods in Natural Language Processing*, 2021, pp. 8696–8708.