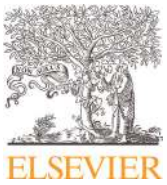
Future Generation Computer Systems 175 (2026) 108067



Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

On-device derivation of IoT usage control policies: Automating U-XACML policy generation from natural language with LLMs in smart homes environments ⋆

Loay Alajramy [a,b,*], Marco Simoni [a,c], Marco Rasori [a], Andrea Saracino [a,b], Paolo Mori [a]

[a] Institute of Informatics and Telematics, National Research Council of Italy, Via G. Moruzzi 1, Pisa, 56124, Italy
[b] Department of Excellence in Robotics and AI, TeCIP, Scuola Superiore Sant'Anna, Piazza Martiri della Libertà 33, 56127, Pisa, Italy
[c] Sapienza Università di Roma, Piazza Aldo Moro 5, 00185, Roma, Italy

ARTICLE INFO

Keywords:
Internet of Things
Usage control
Access control
LLM
Smart home
On-device AI

ABSTRACT

In this paper, we present a framework that integrates AI-based derivation of Access and Usage Control policies for IoT devices, using Large Language Models (LLMs) to automate the generation of policies from unstructured natural language commands. The framework employs a hybrid approach, combining LLMs with dedicated libraries to ensure efficient on-device execution. Our approach is based on a two-step process: first, a fine-tuned LLM converts user commands into structured JSON policy representations; then, a transformation module translates the JSON policies into fully compliant U-XACML policies. To ensure generality across different domains, we introduce a taxonomy-driven dataset creation, which enables policy creation for different environments such as smart homes, smart offices, and healthcare settings. Our evaluation demonstrates that the system achieves 93 % accuracy in policy generation and 91 % accuracy when handling ambiguous or noisy inputs. It also reaches 98 % agreement with expert-defined policies in real-world scenarios. Finally, on-device performance evaluations confirm the feasibility of running the model in practical settings, demonstrating reliable inference under constrained hardware conditions.

1. Introduction

In recent decades, the rapid adoption of Internet of Things (IoT) technologies has transformed modern lifestyles, enabling smarter and more connected living environments. From smart homes to intelligent offices, devices such as smart locks, surveillance cameras, sensors, and actuators have become integral to daily life [1]. However, the widespread use of these devices has also raised significant security and privacy concerns, particularly regarding authorization management. In smart home environments, the integration of devices from different manufacturers and applications capable of interacting with them is becoming increasingly common. Defining and managing authorization policies programmatically in such a dynamic and complex environment poses significant challenges, making it difficult to ensure efficiency, consistency, and adaptability. Yet, ensuring data privacy, safeguarding physical assets integrity, and even protecting user health are critical concerns in such an environment [2]. Some solutions based on remote monitoring and control of the smart home environment already exist. However, these introduce privacy concerns, as private data must be processed outside of the home premises, and they also create a dependency on cloud infrastructure and Internet connectivity.

On-device and cloud-independent solutions have been proposed in frameworks such as the SIFIS-Home framework [3], which manages authorization through the *Usage Control (UCON)* paradigm [4]. Derived from the well-known *Attribute Based Access Control (ABAC)* paradigm [5], UCON enables the definition and enforcement of policies that are both expressive and capable of handling dynamic conditions.

⋆ This work has been partially supported by the Horizon Europe project MEDIATE (grant agreement 101168465), by the European Union – NextGenerationEU within the framework of PNRR Mission 4 – Component 2 – Investment 1.1 under the Italian Ministry of University and Research (MUR) programme "PRIN 2022 PNRR" through the projects AsCoT-SCE (Assessing Compliance of IoT API for Security Critical Environments) – grant number 2022598LMZ – CUP: H53D23003430006, and ASSISTANTS (Assessing Software Security, prIvacy, and SusTAiNability through Testing techniqueS) – grant number P2022WEAH7 – CUP: B53D23026270001.

* Corresponding author.
E-mail addresses: loay.alajramy@santannapisa.it (L. Alajramy), marco.simoni@iit.cnr.it (M. Simoni), marco.rasori@iit.cnr.it (M. Rasori), andrea.saracino@santannapisa.it (A. Saracino), paolo.mori@iit.cnr.it (P. Mori).

https://doi.org/10.1016/j.future.2025.108067
Received 17 March 2025; Received in revised form 21 July 2025; Accepted 5 August 2025
Available online 9 August 2025

One of the key challenges in IoT access control is the complexity of policy generation [6]. The well-known ABAC language, XACML (eXtensible Access Control Markup Language), is expressive but not user-friendly. It is extremely verbose, error prone, and requires deep technical expertise, making it difficult for non-technical users to define policies. To address these challenges, traditional solutions such as *visual policy editors* [7,8] and template-based [9] approaches have provided more user-friendly interfaces, although they still require manual rule configuration and lack in flexibility.

With the increasing popularity of Artificial Intelligence (AI) and the advent of Natural Language Processing (NLP), a number of solutions have been proposed to formulate access control policies based on these technologies [10–13]. However, these approaches still face issues related to scalability, flexibility, and their ability to generalize across different domains. A solution in terms of scalability, flexibility, and generalization can be represented by LLMs, such as ChatGPT [14]. Nevertheless, while LLMs offer significant advantages, their high computational requirements often necessitate an API-based approach, where computation is offloaded to cloud services, thereby reintroducing privacy concerns.

These challenges underscore the need for a fully automated system capable of translating unstructured natural language inputs into precise, adaptable, and machine-readable policies with minimal to no human intervention. This has to be paired with the need to be executed on a general-purpose device, which can be part of a smart home environment.

In this paper, we propose a framework that leverages small-size Large Language Models (LLMs) to automatically generate machine-readable usage control policies from unstructured natural language commands, designed for a smart home environment, being able to run on device and that can be easily generalized to other environments. To the best of our knowledge, we are the first to propose a framework that leverages LLMs to automatically generate usage control policies from unstructured natural language commands. Our approach follows a structured two-step process. First, our fine-tuned LLM, *Text2Policy*, interprets the user's command and converts it into a structured JSON policy, where key elements such as targets, predicates, and conditions are explicitly defined. Next, the structured JSON representation is deterministically converted into a valid U-XACML policy, ensuring full compatibility with access control enforcement mechanisms. The intermediate JSON policy representation reduces the number of tokens the model needs to generate compared to the more verbose XACML format, which contains redundant text. This reduction lowers computational cost and speeds up inference.

To ensure generalization, we introduce a taxonomy-driven pipeline to create datasets. This process first defines the key components—users, actions, resources, and environment attributes—and then systematically generates a dataset of $\langle command, JSON\ policy \rangle$ pairs leveraging predefined templates, forming the training set for the *Text2Policy* model and the test set for evaluation. The test set provides a baseline for assessing the model's ability to accurately translate unstructured natural language commands into valid JSON policies. This approach corresponds to a direct evaluation of U-XACML, since deterministically derived from the JSON policies.

As *Text2Policy* model, we tested two models, namely LLaMA8b [15] and Mistral7b [16], using 4-bit quantization [17] and `unsloth` [18] library to optimize memory usage and enable efficient inference even on resource-constrained devices. LLaMA8b achieved 93 % accuracy in policy generation, while Mistral7b reached 84 %.

To evaluate robustness, we tested *Text2Policy* against ambiguous, noisy, and incomplete inputs. The results show that the generated policies maintained 91 % accuracy for LLaMA8b and 82 % for Mistral7b.

Our taxonomy-driven dataset creation methodology enables seamless adaptation to diverse environments. To assess the ability of the *Text2Policy* model to generalize beyond the smart home domain (used during training) without additional fine-tuning, we tested *Text2Policy* against two distinct test sets built using *smart office* and *healthcare residence for the elderly* taxonomies. These test sets were generated following the same pipeline used for the smart home case. Both models maintained strong performance across these domains, with higher accuracy observed in the smart office scenario and a slight performance drop in the healthcare setting. These results confirm the models' ability to generalize to previously unseen environments without additional fine-tuning.

To further validate our approach, we compared LLM-generated policies with expert-defined ones. A Policy Decision Point (PDP) evaluation demonstrated that policies generated by LLaMA8b achieved a 98 % agreement with expert-defined policies, showcasing the expert-level precision of our method.

Finally, to assess the feasibility of deploying *Text2Policy* in real-word, resource-constrained environments, we conducted a series of on-device performance evaluations, measuring memory usage and inference time under varying policy complexities. Our results show that, even under reduced hardware configurations, the 4-bit quantized model maintained reliable inference performance, demonstrating the practicality of on-device policy generation and reinforcing the impact of our contribution.

Paper structure. The paper is organized as follows. Section 2 introduces Attribute-Based Access Control and the usage control framework. Section 3 describes the reference scenario, providing context for our approach within a smart home environment. Section 4 outlines our methodology for translating natural language commands into U-XACML policies. Section 5 details the dataset construction pipeline for training and testing. Section 6 presents five evaluation test suites assessing *Text2Policy*, including on-device performance testing under resource constraints. Section 7 reviews related research on policy generation and dataset construction. Finally, Section 8 summarizes key findings and future directions.

2. Background

This section provides essential background and introduces key concepts foundational to this work.

2.1. Attribute-based access control

ABAC is an evolution of Access Control List (ACL) that combines the flexibility of Role Based Access Control (RBAC) with fine-grained management of ACL [19]. In ABAC, access is governed by policies that algorithmically combine access control rules. Each rule applies to a target consisting of attributes related to subjects, objects, actions, and the environment. The rule also contains a condition that must be satisfied for it to be applied to the target [20]. ABAC supports attributes from diverse systems, enabling flexible, expressive rules that apply to heterogeneous resources. The evaluation of applicable rules yields to an authorization decision—typically *permit* or *deny*)—or *indeterminate* in case of errors. An advanced extension of ABAC is the Usage Control (UCON) model.

2.2. The usage control framework

Building on ABAC principles, the *UCON framework* regulates the exercise of rights on resources by subjects following the UCON model [5]. It extends the XACML reference architecture and language [21], incorporating dynamic evaluation of policies, continuous monitoring and revocation of ongoing usages, and mutable attribute management.

Unlike traditional access control, which checks permissions only at request time, UCON continuously checks whether access rights remain valid throughout the whole duration of an access to a resource (*usage*). Access grants may be revoked if the value of some subject's, resource's, or environmental attributes change. Attributes whose value can change

over time are called *mutable attributes*. For example, a subject may be allowed to perform an operation in a room only if and as long as the room temperature is between 10 and 35 degrees. Traditional access control only checks conditions at the time of request, allowing the operation if the room temperature is within range. In contrast, UCON not only performs this initial check but also continuously monitors the condition throughout the access period, revoking the grant if the temperature goes out of range.

Usage control policies (*UCPs*) define access strategies using the U-XACML language [22], which extends XACML with time-based conditions. Like standard XACML policies, a UCP consists of multiple *rules*, each returning one of four possible decisions: `Permit`, `Deny`, `NotApplicable`, or `Indeterminate`. The final authorization decision of a policy evaluation is determined by a *rule-combining algorithm*. Additionally, when multiple policies are grouped into a *policy set*, a *policy-combining algorithm* is used to determine a final authorization decision.

Each policy and rule include a *target*, which acts as a filter to determine whether it applies to a given access request. The target can specify a combination of subjects, resources, actions, and environmental attributes. If a request does not match the target, the policy or rule is considered `NotApplicable`, and no further evaluation occurs.

Besides the target, in XACML, a rule consists of a single *effect*—either `Permit` or `Deny`—and a *condition*. The condition, if present, defines the constraints under which the rule applies. It contains a Boolean expression composed of *predicates*, that evaluate contextual attributes of subjects, resources, and environment, such as user roles, resource state, time, and so on. If the condition evaluates to true, the rule contributes to the decision-making process by enforcing its defined effect.

Differently from XACML, a U-XACML rule is structured into three distinct sections, namely, *pre-*, *ongoing-*, and *post-*, each containing its own condition and evaluated at different stages of the usage control workflow, as explained in the following.

The UCON workflow begins when a subject s requests access to perform an action a on a resource r. The *Policy Enforcement Point* (*PEP*) initiates this process by issuing a *tryAccess* message, which includes a *UCON request*, encoded in XACML. A typical UCON request contains the attributes `subject-id`, which identifies the subject, the attribute `resource-id`, which identifies the resource, and the attribute `action-id`, which specifies the action that such a subject wants to perform on the resource.

The UCS processes the request and the Policy Decision Point (PDP) evaluates the *pre-condition* of the selected UCP to determine whether access should be granted. In XACML format, the pre-condition is identified by the tag `DecisionTime="pre"` within the `Condition` element. If the result is `Deny`, the UCS responds to the PEP with a *denyAccess* message, enforcing the decision by denying access to the resource. If `Permit` is returned, the UCS sends a *permitAccess* message to the PEP, allowing access to proceed. Once access begins, the PEP sends a *startAccess* message to the UCS. The UCS then evaluates the *ongoing-condition* to determine whether access can continue. If the result is `Deny`, the UCS sends a *revokeAccess* message to the PEP, which immediately revokes access. If `Permit`, the UCS notifies the PEP via a *permitAccess* message that access can continue. Throughout the access period, if relevant attribute changes occur, the UCS may trigger a re-evaluation of the ongoing-condition, potentially leading to access revocation.

Finally, when access to the resource ends, the PEP sends an *endAccess* message to the UCS. This triggers the evaluation of the *post-condition*, concluding the workflow.

Conflicts in access and usage control policies—such as contradictory decisions, rule overlaps, or redundant constraints—pose a significant challenge in policy management. In the context of XACML and its extensions, various techniques have been developed to detect and resolve such conflicts at the policy specification level [23–26]. These include static analysis tools and formal methods that analyze rule interactions and enforce consistency prior to deployment.

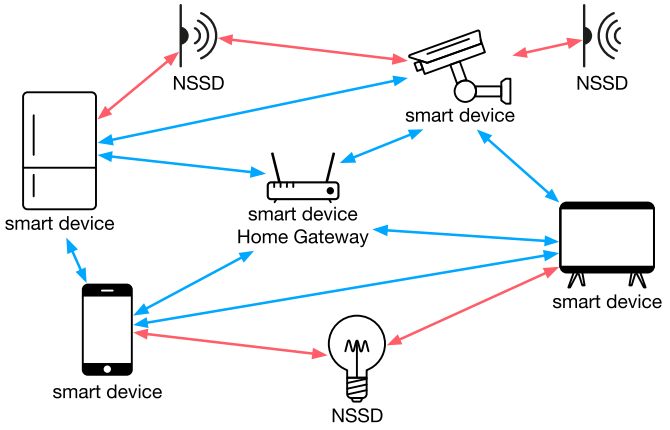


Fig. 1. Smart home's reference architecture.

3. Reference scenario

In this work, we adopt the reference architecture proposed in the EU SIFIS-Home project, as illustrated in Fig. 1 [3]. This architecture represents a distributed smart home environment, which employs a peer-to-peer model, enabling distributed computation, ensuring resiliency by avoiding single point of failure and ensuring privacy as data are stored and analyzed locally in the smart home environments. Core components are the *smart devices*, which possess sufficient computational power to execute complex tasks, such as running AI software for intelligent automation, including text analysis engine, data analysis, and decision-making processes. These devices communicate using mesh-based protocols and Distributed Hash Tables to ensure reliable data exchange, service coordination, and system resilience. Complementing them are *Not So Smart Devices* (NSSD), which are simpler devices such as sensors and actuators that interact with the physical environment. They rely on Smart Devices to process their data and manage control logic. Interested readers can refer to [3] for a complete understanding of the SIFIS-Home architecture and its approach to privacy and security in smart homes.

4. From natural language to U-XACML: policy translation workflow

Policies relevant for smart home environments can be pretty complex as they have to control many users with different access levels, considering attributes for multiple sensors and managing a consistent number of functionalities. Our proposed methodology aims at automating the process of complex policy definition, starting from a voice command. Considering the reference architecture presented in Section 3, the command is collected by a microphone on any smart device and is locally processed. The speech-to-text processing is not in the scope of this work, as it is handled through available speech-to-text libraries. Hence, we assume in the following to have as input the command in natural language in textual form. The process of translating natural language commands into UCON-compliant policies involves a multi-phase workflow, which is illustrated in Fig. 2 and described in the following.

4.1. From natural language command to JSON policy

The process begins with a natural language command from the user, which defines a policy outlining the conditions for executing a specific action on a resource. This command can be provided as text or obtained through transcription of a voice command. In Fig. 2, such a command is: *"Permit to turn on air conditioner if the subject's role is Guest on Saturday between 8 am and 7 pm"*.

In Phase 1 of the workflow, our *Text2Policy* model processes natural language commands in text form, transforming them into a structured

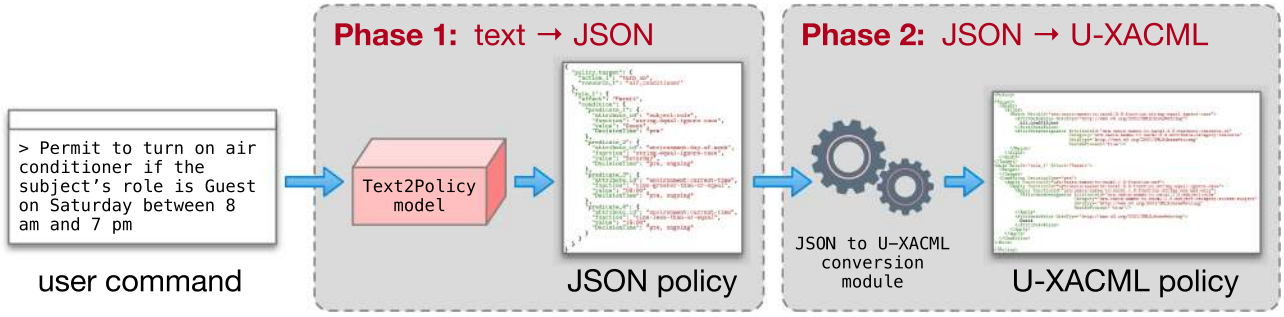


Fig. 2. Policy translation workflow: From natural language command to U-XACML policy.

```
{
  "policy_target": {
    "action_1": "turn_on",
    "resource_1": "air_conditioner"
  },
  "rule_1": {
    "effect": "Permit",
    "condition": {
      "predicate_1": {
        "attribute_id": "subject:role",
        "function": "string-equal-ignore-case",
        "value": "Guest",
        "DecisionTime": "pre"
      },
      "predicate_2": {
        "attribute_id": "environment:day-of-week",
        "function": "string-equal-ignore-case",
        "value": "Saturday",
        "DecisionTime": "pre, ongoing"
      },
      "predicate_3": {
        "attribute_id": "environment:current-time",
        "function": "time-greater-than-or-equal",
        "value": "08:00",
        "DecisionTime": "pre, ongoing"
      },
      "predicate_4": {
        "attribute_id": "environment:current-time",
        "function": "time-less-than-or-equal",
        "value": "19:00",
        "DecisionTime": "pre, ongoing"
      }
    }
  }
}
```

Listing 1. JSON Policy for the user command in Fig. 2.

JSON Policy. This policy represents key elements derived from the command such as actions, resources, subjects, and predicates in a structured JSON format. This representation simplifies the subsequent conversion into U-XACML.

Through the JSON policy, we provide a clear and machine-readable representation of the original command. This approach significantly reduces the number of tokens the LLM needs to generate compared to the more verbose XACML format, which often includes redundant sequences of words. Nonetheless, we structured the JSON policies to mirror the design and logic of UCON policies. Listing 1 shows the JSON policy produced from the user command shown in Fig. 2.

For clarity and convenience, in the following, references to specific lines in the listings use the notation $L{:}N$, where L denotes the listing number and N indicates the corresponding line number. For example, 1:12 refers to line 12 of Listing 1.

Starting from the natural language command, our model first identifies the rules and their corresponding targets. Then, if all the rules share the same target, a policy target is reported in the JSON with the key `policy_target`, positioned at the top of the JSON structure (1:2–1:5). In this way, we reduce the number of generated tokens avoiding redundant repetitions that could lead to undesirable incoherence.

For each identified rule, the model creates a single condition, which is represented as a set of *predicates*. For example, in the JSON policy in Listing 1, there is only one rule, which contains a condition composed of four distinct predicates: `predicate_1` (1:9–1:14) for the subject role, `predicate_2` (1:15–1:20) for the day, `predicate_3` (1:21–1:26) for the starting time, and `predicate_4` (1:27–1:32) for the ending time.

Predicates are organized based on their *decision time* by means of the property name `DecisionTime`, which represents the phase of the UCON decision process when they should be enforced. This allows our tool to map them to their corresponding UCON conditions in the rule of the U-XACML policy, i.e., *pre-conditions* and *ongoing-conditions*.

For example, the `Decision Time` for `predicate_1` (1:13) is set to `pre` because the attribute considered in this predicate is the *subject role*, which is an *immutable* attribute. Therefore, in the corresponding U-XACML policy, the whole predicate will be automatically included within a *pre-condition*, represented by the `<Condition DecisionTime="pre">` element, as illustrated in Listing 2 (2:22).

On the other hand, `predicate_2`, `predicate_3`, and `predicate_4` are related to *mutable* attributes, specifically, the day and time. Therefore, in the JSON policy, the decision times for these predicates are both `pre` and `ongoing` (1:19, 1:25, and 1:31). This means that, in the final U-XACML policy, each of these predicates will be inserted both under the `<Condition DecisionTime="pre">` element and under the `<Condition DecisionTime= "ongoing">` element.

We assume that the predicates in the policies being translated into U-XACML are expressed in *Disjunctive Normal Form (DNF)*. This assumption does not limit the expressivity of our approach, as the Disjunctive Normal Form Theorem states that any logical formula can be transformed into an equivalent one in DNF [27].

This allows us to structure the JSON policy in a straightforward manner. Specifically, predicates within a condition of a rule of a JSON policy are interpreted as a conjunction, meaning that all predicates must be true for the condition to evaluate to true. Conversely, to represent a disjunction of two or more predicates, each predicate must be placed in a distinct rule within the same JSON policy, ensuring that all such rules share the same effect.

4.2. Converting JSON policies to U-XACML

Phase 2 of the workflow involves the *Conversion Module*, which is responsible for transforming a JSON policy into UCON-compliant format. This transformation is deterministic and fully automated, achieved by parsing the JSON structure and systematically mapping attributes, predicates, and other elements into a structured U-XACML policy.

The JSON policy in Listing 1 is complete, as it contains all essential components: the policy target, rules, and conditions. The U-XACML representation in Listing 2 shows the translation of `predicate_1` of `rule_1` into the U-XACML format, representing only a portion of the JSON policy taken as reference. To facilitate comparison, corresponding elements between Listings 1 and 2 are highlighted in both listings. The following explanation shows the general methodology employed by the

```
1  <Policy RuleCombiningAlgId="urn:oasis:names:tc:xacml:3.0:rule-combining-algorithm:deny-unless-permit" ... >
2  ...
3  <Target>
4    ...
5    <AnyOf>
6      <AllOf>
7        <Match MatchId="urn:oasis:names:tc:xacml:3.0:function:string-equal-ignore-case">
8          <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">
9            air_conditioner
10         </AttributeValue>
11         <AttributeDesignator AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-id"
12                              Category="urn:oasis:names:tc:xacml:3.0:attribute-category:resource"
13                              DataType="http://www.w3.org/2001/XMLSchema#string"
14                              MustBePresent="true"/>
15       </Match>
16     </AllOf>
17   </AnyOf>
18 </Target>
19 <Rule RuleId="rule_1" Effect="Permit">
20   <Target>
21   </Target>
22   <Condition DecisionTime="pre">
23     <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:and">
24       <Apply FunctionId="urn:oasis:names:tc:xacml:3.0:function:string-equal-ignore-case">
25         <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:string-one-and-only">
26           <AttributeDesignator AttributeId="urn:oasis:names:tc:xacml:1.0:subject:role"
27                                Category="urn:oasis:names:tc:xacml:1.0:subject-category:access-subject"
28                                DataType="http://www.w3.org/2001/XMLSchema#string"
29                                MustBePresent="true"/>
30         </Apply>
31         <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">
32           Guest
33         </AttributeValue>
34       </Apply>
35       ...
36     </Apply>
37   </Condition>
38 </Rule>
39 ...
40 </Policy>
```

Listing 2. Portion of U-XACML policy generated from the JSON policy shown in Listing 1.

conversion module through a specific example, demonstrating how a JSON policy is translated into a U-XACML policy.

Policy and rule targets. In the JSON policy shown in Listing 1, the `policy_target` object contains the attributes `action_1` (*turn on*) and `resource_1` (*air conditioner*). In the U-XACML format, these attributes must be placed within the policy target, specifically inside the XACML `<Target>` element, which defines the scope of applicability for the policy. For simplicity, Listing 2 only shows the attribute `air_conditioner` (2:9, 2:11) in the `<Target>` section (2:3–2:18). The complete version of the target section would also include the action-id *turn on*, logically ANDed with the resource-id. Regarding rule targets, we note that the JSON policy contains only one rule without a specified target. Consequently, the U-XACML representation includes an empty `Target` element (2:20, 2:21), meaning that the rule applies to all requests that reach the policy, with its applicability determined solely by the rule's condition.

In cases where we have multiple targets within a rule or policy that belong to the same category (subject, action or resource), they are put in logical OR using the tag `<AnyOf>`. The tag `<AnyOf>` combines several `<Match>`, whereby it is sufficient if at least one target is fulfilled (logical OR).

Rules definition. The rules in the JSON policy, such as `rule_1` in Listing 1 (1:6–1:34), are transformed into U-XACML `<Rule>` elements (2:19–2:38). Each rule of the JSON policy consists of the following elements:

- **Rule name**: The rule is represented as a key-value pair, where the key (e.g., `"rule_1"`) serves as the rule's unique identifier (1:6). This key is directly mapped to the `RuleId` attribute of the `<Rule>` element in the U-XACML policy (1:19).
- **Target**: Within the rule, the value of the `target` key determines its applicability. In the provided JSON policy, the rule's target is not present, which results in an empty target in the U-XACML policy (2:20, 2:21). However, if a target were present, it would be represented in the U-XACML policy as discussed above for the policy target.
- **Effect**: Within the rule, the value of the `effect` key specifies whether the rule permits or denies the access (e.g., *Permit* in our example). This maps directly to the `Effect` attribute of the `<Rule>` element in XACML (2:19).
- **Condition**: The `condition` key holds a set of predicates that define additional constraints that must be satisfied for the rule to apply. Note that if a single condition in the JSON policy includes multiple distinct `DecisionTime` values within its predicates, it will be translated into multiple conditions in the U-XACML policy, ensuring proper enforcement at different evaluation stages.

Conditions mapping. All the predicates within a `condition` object of a JSON policy, such as `predicate_1` in our example, are mapped to XACML `<Condition>` elements. Each predicate in the JSON policy specifies attribute identifiers, functions, and values, which are transformed into U-XACML format as follows:

- `function` (1:11): mapped to the appropriate XACML `FunctionId` field of the XACML `<Apply>` element (2:24).
- `attribute_id` (1:10): Mapped to the `AttributeId` field within the `<AttributeDesignator>` element in XACML (2:26).
- `value` (1:12): represented in XACML exploiting the `<AttributeValue>` element (1:31–1:33).
- `DecisionTime` (1:13): It is used to chose whether to copy the predicate within the condition of the rule having `DecisionTime="pre"` or within the condition having `DecisionTime="ongoing"` or both.
- `data_type` (2:28): we automatically detect the XACML-compliant data type for the `<AttributeValue>` element by examining the `predicate value` in the JSON policy. Additionally, the algorithm employs a rule-based approach to sequentially evaluate the input against various data types.

Policy generation. Once the U-XACML targets, rules, and conditions are defined, they are incorporated into a complete U-XACML policy using the `<Policy>` element (2:1–2:40).

There could be cases where the user command includes also a default rule, such as: *"Permit to turn on the air conditioner if the subject's role is Guest from Monday to Saturday between 8 am and 7 pm.* ***Deny otherwise****"*. Here, *"Deny otherwise"* represents a default rule (*default-deny* rule), i.e., a rule that is applied when no other rules are applicable. In this example, the JSON policy generated by the *Text2Policy* model would represent the default rule as *Rule 2* (since *"Permit to turn on the air conditioner if the subject's role is Guest from Monday to Saturday between 8 am and 7 pm."* is the same *User Command* of Fig. 2 and can be represented in one rule), with effect *Deny* and no predicates.

To set the appropriate *rule-combining algorithm*, if the generated policy contains a default-deny rule, we use the deny-unless-permit algorithm and remove the default rule. Conversely, if it contains a default-permit rule, we use the permit-unless-deny algorithm and remove the default rule. If all the rules within the policy have a permit effect, we use the deny-unless-permit algorithm. Conversely, if all the rules have a deny effect, we use the permit-unless-deny algorithm. If the rules contain a combination of permit and deny effects, we adopt the deny-unless-permit algorithm. This approach ensures that all possible cases in the user prompt are covered without compromising security.

In Listing 2, the rule-combining algorithm for this policy is `deny-unless-permit` (2:1), as there is only one rule with effect permit and no default rule.

After generating the U-XACML policy, formal verification tools can be employed to validate its correctness and consistency. These tools enable automated reasoning to identify logical conflicts, redundancies, and policy misconfigurations, which is particularly valuable in security-critical environments. For example, several XACML conflict detection tools (e.g., [23–25]) can be readily adapted to our setting, as U-XACML is an extension of XACML.

4.2.1. Flattening process

Although the previous explanations present the conversion as a direct transformation from a JSON policy to a U-XACML policy, the conversion module first applies a preprocessing step: the *flattening process*. This intermediate step simplifies the JSON structure before conversion. It consists of restructuring nested keys into a flat format by concatenating the hierarchical levels with a separator. As an example, Fig. 3 shows how the action `action_1`, which is part of the policy target, is transformed into a flattened structure.

In this case, the nested key `policy_target.action_1` is simplified into the single key `policy_target_action_1`. The conversion module operates on this flattened version, ensuring a structured and deterministic transformation process. However, for clarity, we present the conversion as if applied directly to the original JSON policy. The flattened version serves as an intermediate representation that simplifies attribute resolution and processing within the module.

Since the transformation is deterministic, evaluating the flattened JSON policy is equivalent to evaluating the final U-XACML policy. This equivalence allows us to use the flattened version to assess the performance of the *Text2Policy* model, as detailed in Section 6.

```
// JSON policy from the Text2Policy model
{"policy_target": { "action_1": "turn_on" }}

// Flattened version
{"policy_target_action_1": "turn_on"}
```

Fig. 3. Flattening process of a portion of a JSON policy, where the nested key `policy_target.action_1` is transformed into the flattened key `policy_target_action_1`.

`{effect}`[1] `{action}`[2] `{resource}`[3] *if the subject's role is* `{home-role}`[4], *on* `{Day}`[5] *between* `{F-Time}`[6] *and* `{S-Time}`[7], *or on* `{Day-2}`[8] *between* `{F-Time-2}`[9] *and* `{S-Time-2}`[10].

Command Template 1
Example of command template with annotations.

5. Taxonomy-driven dataset creation

In order to train and test the model for *Text2Policy* purpose, we built a dataset composed of $\langle command, JSON\ policy \rangle$ pairs by applying a domain-specific taxonomy to a set of pre-defined templates.

The development of a domain-specific taxonomy enables a structured representation of smart home environments through the organization of entities, attributes and relationships. Our taxonomy categorizes key elements, including users, devices, actions and environment parameters. To support access management, the taxonomy introduces *Access Levels* (e.g., *Resident*, *Guest*) and *Parental Advisory Levels* (e.g., *red*, *green*, *yellow*), which enable fine-grained control over smart home operations. Resource classification is based on device functionality: *Temperature Devices* (thermostats, heaters, air conditioners), *Brightness Devices* (ceiling lights, desk lamps, smart bulbs), *Security Devices* (door locks, security cameras, garage doors), and *Audio Devices* (smart speakers, televisions, soundbars). General-purpose devices such as motion sensors, smoke alarms, and CO_2 sensors are also included. Actions are structured by their operations. *Temperature control* includes setting, raising and lowering the temperature, while *Brightness control* adjusts the light intensity. *Security operations* manages locking mechanisms and *Audio control* allows volume adjustments. General commands include switching devices on and off as well as document-related actions such as reading and writing. This structured approach supports smart home interactions, security policies, and access controls. The methodology can be extended to other domains, such as smart offices, healthcare, and industrial IoT, ensuring a flexible and consistent workflow.

In the remainder of this section, we explain in detail the pipeline for creating the dataset, which follows the definition of the taxonomy.

5.1. Dataset creation pipeline

Fig. 4 illustrates the dataset generation process once the taxonomy has been established, consisting of the following key steps:

Manual templates creation. Domain experts manually design m initial templates [28–30], including command templates (CT) and their corresponding JSON policy templates (PT). A **Command Template (CT)** defines executable instructions in natural language, such as turning on a device or adjusting temperature, possibly under given constraints. An example of a command template is shown in Command Template 1. A **JSON Policy Template (PT)** specifies the key components for building the policy in JSON format, extracted from a textual command. It includes execution constraints and permissions. The corresponding JSON policy template for Command Template 1 is presented in Listing 3.

We manually built a total of 256 *template pairs*, each consisting of a command template and a JSON policy template. Please note that this task is agnostic of the specific domain, and the produced template pairs can be reused for creating datasets for virtually any smart environment since the domain's taxonomy is applied in a subsequent step, as described later in this section. The process of generating these template pairs is detailed in Section 5.1.1.

Paraphrasing of templates. In **Step 1a** of Fig. 4, the initial command templates are paraphrased by an LLM, generating k paraphrases for each command template. This process improves variability and ensures the coverage of different phrasings for the same command and JSON policy template pair.

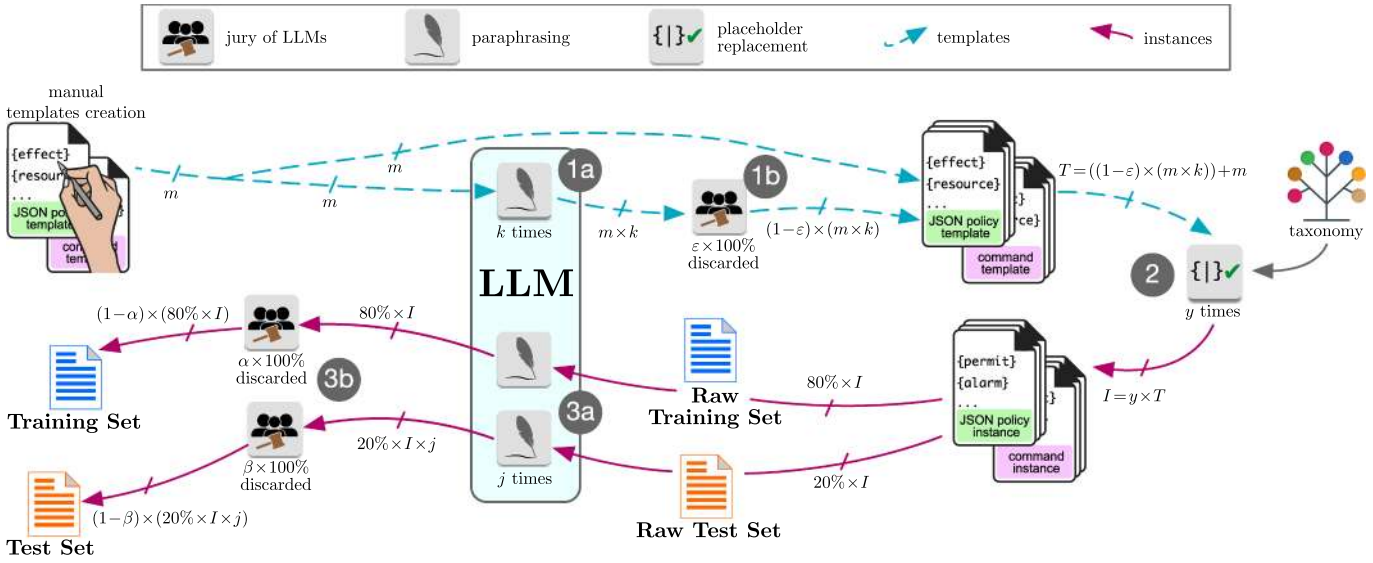


Fig. 4. Dataset creation workflow after taxonomy definition.

`Permit`[1] `turn on`[2] `air conditioner`[3] *if the subject's role is* `"Guest"`[4], *on* `Saturday`[5] *between* `8 am`[6] *and* `7 pm`[7], *or on* `Sunday`[8] *between* `9 am`[9] *and* `5 pm`[10].

Command Instance 1
Example of command instance with annotations.

In **Step 1b**, a **jury of 3 LLMs** (see Section 5.1.1 for more details) evaluates the quality of the paraphrases generated in Step 1a, ensuring that only accurate and meaningful paraphrases are retained. During this process, a fraction ε of template pairs is discarded.

The manually created template pairs and the retained template pairs compose the *final set of template pairs* $T = ((1-\varepsilon) \times (m \times k) + m)$.

Filling placeholders. In **Step 2**, the placeholders within each template pair in T are replaced dynamically with *values* derived from the taxonomy. This step creates realistic and context-aware command-JSON policy instances (*instance pairs*) by exploring y value combinations, resulting in $I = y \times T$ unique instance pairs.

The replacement process follows the order in which placeholders appear in the paraphrased command template. First, the type of each placeholder (e.g., `{action}`, `{resource}`, or `{role}`) is identified. The first placeholder is selected entirely at random from its corresponding domain. For all subsequent placeholders, values are chosen randomly but constrained to those that are compatible with values already assigned to previous placeholders. Compatibility is determined by consulting an *adjacency matrix* A, which ensures logical consistency in the generated instances. Once all placeholders are filled following this process, the final instance pair is added to the set of instance pairs I. Further details on these operations are provided in Section 5.1.2.

Raw dataset generation. The complete set of instance pairs I is divided into two subsets to ensure a structured evaluation. Approximately 80 % of the instances ($\approx 80\,\% \times I$) are allocated to the training set, forming the *Raw Training Set*, while the remaining 20 % ($\approx 20\,\% \times I$) constitute the *Raw Test Set*, used for evaluation.

Final dataset refinement. In **Step 3a**, both the raw training set and the raw test set are provided as input to the LLM, which generates paraphrased versions. Specifically, the entire raw training set is paraphrased once, while the raw test set is paraphrased j times. This ensures robustness and improves the generalization capabilities of the dataset. The result is then used as input for the next step. In **Step 3b**, the jury of 3 LLMs (as in Step 1b) assesses the correctness of the paraphrased training and test sets, ensuring that the meaning of the commands remains consistent. Such a filtering process excludes any paraphrased commands that differ in meaning from their original, non-paraphrased counterparts. This step enhances the dataset's reliability and consistency. After this refinement, the dataset is finalized and ready to be used for training and evaluation of the *Text2Policy* model.

In the following, we detail the step-by-step workflow necessary to build the dataset, discussing **Step 1a** and **Step 1b** in Section 5.1.1, **Step 2** in Section 5.1.2, and **Step 3a** and **3b** in Section 5.1.3.

5.1.1. Template pairs creation and paraphrasis

The starting point for the dataset creation pipeline is the generation of command templates and their corresponding JSON policy templates, resulting in the creation of template pairs.

Command template definition. Formally, a *command template* (CT) can be defined as a structured string comprising fixed components and placeholders.

Let $P_{\text{CT}} = \{p_1, p_2, \ldots, p_n\}$ be the set of placeholders for a command template, where each p_i represents a variable slot that can be instantiated with a value from a predefined domain D_i.

A command template CT is defined as:

$$CT = S_0 p_1 S_1 p_2 \ldots p_n S_n,$$

where $\{S_i \mid i = 0, \ldots, n\}$ is a set of fixed strings representing the static parts of the command.

In the example reported in command template 1, the set of placeholders is defined as:

$$P_{\text{CT}} = \{p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8, p_9, p_{10}\},$$

where:

$p_1 =$ `{effect}`, $p_2 =$ `{action}`, $p_3 =$ `{resource}`,
$p_4 =$ `{home-role}`, $p_5 =$ `{Day}`, $p_6 =$ `{F-Time}`,
$p_7 =$ `{S-Time}`, $p_8 =$ `{Day-2}`, $p_9 =$ `{F-Time-2}`,
$p_{10} =$ `{S-Time-2}`.

The fixed components in the command template, denoted as S_{CT}, correspond to the static parts of the command that remain unchanged.

```
{
  "policy_target": {
    "action_1": "{action}",     // Replace with: "turn_on"
    "resource_1": "{resource}" // Replace with: "air_conditioner"
  },
  "rule_1": {
    "effect": "{effect}",       // Replace with: "Permit"
    "condition": {
      "predicate_1": {
        "attribute_id": "subject:role",
        "function": "string-equal-ignore-case",
        "value": "{home-role}", // Replace with: "Guest"
        "DecisionTime": "pre"
      },
      "predicate_2": {
        "attribute_id": "environment:day-of-week",
        "function": "string-equal-ignore-case",
        "value": "{Day}",       // Replace with: "Saturday"
        "DecisionTime": "pre, ongoing"
      },
      "predicate_3": {
        "attribute_id": "environment:current-time",
        "function": "time-greater-than-or-equal",
        "value": "{F-Time}",    // Replace with: "08:00"
        "DecisionTime": "pre, ongoing"
      },
      "predicate_4": {
        "attribute_id": "environment:current-time",
        "function": "time-less-than-or-equal",
        "value": "{S-Time}",    // Replace with: "19:00"
        "DecisionTime": "pre, ongoing"
      }
    }
  },
  "rule_2": {
    "effect": "{effect}",       // Replace with: "Permit"
    "condition": {
      "predicate_1": {
        "attribute_id": "subject:role",
        "function": "string-equal-ignore-case",
        "value": "{home-role}", // Replace with: "Guest"
        "DecisionTime": "pre"
      },
      "predicate_2": {
        "attribute_id": "environment:day-of-week",
        "function": "string-equal-ignore-case",
        "value": "{Day-2}",     // Replace with: "Sunday"
        "DecisionTime": "pre, ongoing"
      },
      "predicate_3": {
        "attribute_id": "environment:current-time",
        "function": "time-greater-than-or-equal",
        "value": "{F-Time-2}", // Replace with: "09:00"
        "DecisionTime": "pre, ongoing"
      },
      "predicate_4": {
        "attribute_id": "environment:current-time",
        "function": "time-less-than-or-equal",
        "value": "{S-Time-2}", // Replace with: "17:00"
        "DecisionTime": "pre, ongoing"
      }
    }
  }
}
```

Listing 3. JSON policy template PT for command template 1.

Formally, S_{CT} is defined as the difference between the full command template CT and the set of placeholders P_{CT}: $S_{\text{CT}} = CT - P_{\text{CT}}$. In the example of command template 1, S_{CT} consists of the following fixed components:

$$s_1 = if the subject's role is, \quad s_2 = on, \quad s_3 = between,$$
$$s_4 = and, \quad s_5 = or on, \quad s_6 = between, \quad s_7 = and.$$

JSON policy template definition. A *JSON Policy Template* (PT) is a structured representation of the key components to build a usage control policy expressed in JSON format. It defines the logical rules, conditions, and predicates that govern the execution of a command. Let $P_{\text{PT}} = P_{\text{CT}} = \{p_1, p_2, \ldots, p_n\}$ be the set of placeholders in the JSON policy template, where each p_i corresponds to the same placeholder in the associated command template CT. A JSON policy template PT is formally expressed as:

$$PT = S_{\text{PT}}(p_1, p_2, \ldots, p_n),$$

where S_{PT} defines the static structure of the JSON object, including its fields (e.g., `policy_target`, `rules`, etc.). Hence, the fixed components S_{PT} represent the structure and logical constraints of the policy, such as predicates and rules.

The JSON policy template shown in Listing 3 corresponds to Command Template 1. Together, they form a template pair.

Template paraphrasing. To expand the pool of command templates available to us, we used a Large Language Model (LLaMA8b [15]) to generate paraphrases or variations of command templates (**Step 1a**). By diversifying the natural language expressions of CT, we enhance the dataset's robustness to linguistic variability, allowing for better generalization in real-world scenarios. Additionally, this process increases the dataset's size, providing additional training examples to improve model performance.

Given a template pair $\langle CT, PT \rangle$ composed of an original command template CT and its corresponding JSON policy template PT, the LLM generates a set of paraphrased template pairs defined as:

$$\text{Var}(\langle CT, PT \rangle) = \{\langle CT, PT \rangle, \langle CT_i, PT \rangle \mid i = 1, \ldots, k\}.$$

In this work, we set $k = 5$, meaning that each original template pair is expanded into five paraphrased variations.

This set $\text{Var}(\langle CT, PT \rangle)$ includes the original template pair as well as its paraphrased versions, where only the command template is modified while the JSON policy template remains unchanged across all template pairs.

Unlike the manually created original CT, which is guaranteed to be semantically correct, automatically generated paraphrases may not always preserve the original meaning. Thus, verifying the correctness of these paraphrases is essential. However, manual validation is impractical due to the large volume of generated paraphrases. To address this challenge, we adopted the *LLM-as-a-Judge* paradigm introduced by Zheng et al. [31] and employed a jury-based approach, as proposed by Verga et al. [32], since it demonstrated a higher agreement rate with human evaluations compared to the single-LLM approach.

In **Step 1b**, we employed three LLMs judges—LLaMA3.3 70b [15], Qwen2.5 70b [33,34], and Phi4 [35]—to form *PoLL* (Π, a Panel of LLM Evaluators). To ensure semantic equivalence, each LLM independently evaluates whether a paraphrased command template CT_i retains the same meaning as the original command template CT. Each judge votes either `true` (if the paraphrase preserves the original intent) or `false` (if the meaning is altered). The final decision, determined using a *majority vote* aggregation strategy, is formally defined as:

$$\Pi(CT, CT_i) = \begin{cases} \texttt{true}, & \text{if the majority of judges vote } \texttt{true} \\ \texttt{false}, & \text{otherwise.} \end{cases}$$

For each template pair, the validation process, which ensures that only semantically accurate paraphrases are retained, is defined as:

$$\text{Validated}(\text{Var}(\langle CT, PT \rangle)) = \{\langle CT_i, PT \rangle \mid CT_i \in \text{Var}(\langle CT, PT \rangle), \Pi(CT, CT_i) = \text{true}\}.$$

As a result of this filtering, a fraction ε of template pairs—those that do not preserve the intended meaning—is discarded. The resulting $\text{Validated}(\text{Var}(\langle CI, PI \rangle))$ contains only those template pairs where the paraphrased was judged to be semantically valid.

After this step, we retained a total of 1604 template pairs. These validated template pairs, together with the m original template pairs, form the final set of template pairs I.

5.1.2. Logical consistency in placeholders replacement

In **Step 2** of Fig. 4, given a set of replacement values $V = \{v_1, v_2, \ldots, v_n\}$, where $v_i \in D_i$ (the domain of placeholder p_i), the generation of command Instance $CI(V)$ and policy instance $PI(V)$ consists in substituting each placeholder p_i in both CT and PT with its corresponding value v_i for all i:

$$CI(V) = S_0\ v_1\ S_1\ v_2\ S_2\ \ldots\ v_n\ S_n,$$
$$PI(V) = S_{\text{PT}}(v_1, \ldots, v_n).$$

This ensures dataset diversity, as each placeholder p_i can take values from its associated domain D_i. For example, considering our taxonomy, the domain D_{effect} includes `Permit` or `Deny`, while D_{resource} contains values like `thermostat`, `ceiling light`, or `door lock`. Command Instance 1 illustrates a command instance generated from command template 1, where each placeholder has been replaced with a value from

its corresponding domain. The specific values used to replace the placeholders are as follows:

$v_1 = \texttt{Permit}$, $v_2 = \texttt{turn on}$, $v_3 = \texttt{air conditioner}$,
$v_4 = \texttt{Guest}$, $v_5 = \texttt{Saturday}$, $v_6 = \texttt{8:00}$,
$v_7 = \texttt{19:00}$, $v_8 = \texttt{Sunday}$, $v_9 = \texttt{9:00}$,
$v_{10} = \texttt{17:00}$.

Since multiple combinations of values can fill the placeholders of Command Template 1, Command Instance 1 represents just one possible instantiation of this template.

When automatically replacing placeholders in CT and PT with real values, we must ensure that the resulting instance pair is *logically consistent*.

To maintain logical consistency, we use an *adjacency matrix* A to map placeholders to compatible values. Formally, let A be a matrix of size $n \times n$, where n represents the total number of values across all domains in the taxonomy. Each entry $A[i, j]$ takes a value from $\{0, 1\}$, where $A[i, j] = 1$ indicates that a value from the domain D_i is compatible with a value from the domain D_j, while $A[i, j] = 0$ indicates incompatibility.

Given a command template CT with placeholders $P_{\text{CT}} = \{p_1, p_2, \ldots, p_n\}$, the adjacency matrix A defines a compatibility graph $G = (V, E)$. In this graph, V represents the set of values that can fill the placeholders in CT (and PT), and E consists of edges (v_i, v_j) such that $A[i, j] = 1$, meaning the values v_i and v_j are compatible.

The adjacency matrix A ensures that when instantiating CT and PT, a value $v_i \in D_i$ for p_i is only selected if it is compatible with the previously instantiated values $V_{\text{prev}} = \{v_1, v_2, \ldots, v_{i-1}\}$, as dictated by A. For example, if CT includes placeholders $P_{\text{CT}} = \{p_1, p_2, p_3\}$, where: $p_1 = \texttt{\{action\}}$, $p_2 = \texttt{\{resource\}}$, and $p_3 = \texttt{\{effect\}}$, the adjacency matrix A enforces logical consistency in value selection. Specifically, actions like `set temperature` (p_1) can only be paired with resources like `thermostat` or `heater` (p_2), while actions like `lock` or `unlock` (p_1) are only compatible with resources like `door lock` or `garage door` (p_2). Furthermore, the effect (p_3) must align with the logical constraints imposed by the chosen action and resource.

Once the command instance CI is constructed by assigning values v_i to each placeholder p_i in CT, the corresponding policy instance PI can be generated. This process involves substituting each placeholder p_i in the policy template PT with the same value v_i that was selected during the construction of CI.

By leveraging A, we ensure that both CI and PI are logically consistent and adhere to the constraints of the environment. For example, `set temperature` paired with `door lock`, or `unlock` paired with `thermostat`, would be prevented by A. This structured approach ensures semantic integrity while allowing for scalable and robust dataset generation. The complete replacement process is represented in Algorithm 1. During this step, each template pair was instantiated five times ($y = 5$) by applying Algorithm 1 iteratively. As a result, a total of 5312 instance pairs were generated.

Algorithm 1: Generation of an instance pair through sequential placeholder replacement.

Input: – *Template pair* $\langle CT, PT \rangle$ with placeholders $P = \{p_1, p_2, \ldots, p_n\}$
– *Domain* D: `Smart Home Environment`;
– *Adjacency matrix* A.
Output: Logically consistent *instance pair* $\langle CI, PI \rangle$.
Initialize: Set of selected values $V \leftarrow \emptyset$;
foreach *placeholder* $p_i \in P$ **do** ▷ ordered, $1 \rightarrow n$
  if $i = 1$ **then** ▷ first placeholder
    Select v_1 randomly from its corresponding category in D;
  else
    Select v_i randomly from its category in D such that it maintains logical consistency:
    $\forall v_j \in V, \quad A[i, j] = 1$ (i.e., v_i is compatible with all previously chosen values);
  end
  Add v_i to V;
end
Build final instances:
  $CI \leftarrow S_0\ v_1\ S_1\ v_2\ S_2\ \ldots\ v_n\ S_n$;
  $PI \leftarrow S_{\text{PT}}(v_1, v_2, \ldots, v_n)$;
return $\langle CI, PI \rangle$;

5.1.3. Training and test set construction

Once all the instance pairs have been generated, the dataset has been split into two subset, namely the training set and the test set.

To introduce diversity, the original template pairs were initially paraphrased $k = 5$ times (Step 1a), and then, for each template pair, placeholders were replaced $y = 5$ times (Step 2), thus obtaining the set of instance pairs I. Notably, the y instance pairs derived from the same template pair differ only in the values used for substituting the placeholders. Therefore, the resulting y command instances share the same static part—the fixed string components in S. This limited diversity in phrasing can result in the *Text2Policy* model learning to rely on the static structure of the commands rather than understanding the underlying semantics. As a result, the model may perform well on commands that closely match those seen during training but fail when presented with commands conveying the same meaning but phrased differently.

To mitigate this, we introduced a second layer of paraphrasing at the CI level. This ensured that command instances with the same base structure but different placeholder values also varied in their phrasing, introducing additional linguistic diversity and reducing the model's reliance on static structures.

Similarly to Step 1a, in **Step 3a**, we enhanced the dataset by introducing paraphrased variations of command instances to improve linguistic diversity. For the training set ($80\,\% \times I$), each instance pair $\langle CI, PI \rangle$ was paraphrased once. In contrast, for the test set ($20\,\% \times I$), each instance pair was paraphrased five times (j in Fig. 4), ensuring a more diverse evaluation set. Formally, the paraphrased instance pairs are defined as:

$$\text{Var}(\langle CI, PI \rangle) = \{\langle CI, PI \rangle, \langle CI_i, PI \rangle \mid i = 1, \ldots, j\},$$

where $j = 1$ for the training set and $j = 5$ for the test set.

This process resulted in 4250 instance pairs for training and 5310 for testing.

In **Step 3b**, to ensure semantic equivalence between the original and paraphrased instance pairs, we applied the same validation process as in Step 1b. Specifically, all paraphrases in both the training and test sets were evaluated using the jury-based PoLL approach, as detailed in Section 5.1.1. Through this process, only paraphrases deemed semantically valid were retained, formally captured by $\text{Validated}(\text{Var}(\langle CI, PI \rangle))$, and applied to each instance pair in I. After filtering, a fraction α (β) of instance pairs is discarded from the training (test) set. The final dataset is composed of 4096 training instance pairs and 4681 test instance pairs.

6. Experimental analysis

This section evaluates the performance of the *Text2Policy* model by assessing its *accuracy*, *robustness*, *ability to generalize*, and *practical usefulness*. As the *Text2Policy* model, we trained two LLMs, LLaMA8b [15] and Mistral7b [16], for three epochs using Supervised Fine-Tuning (SFT) [36]. Training was performed on an A100 GPU with 40GB of memory. For training and inference, we used the `unsloth` [18] library, capping memory usage at 70 % for LLaMA8b and 75 % for Mistral7b. Both models were 4-bit quantized [17], ensuring efficient inference even on resource-constrained devices. Since our goal is to support deployment on resource-constrained devices, inference time is not a primary concern. Whether the models take a few seconds or a few minutes [37] to

generate policies is inconsequential, as we prioritize the quality of the generated JSON policies. Therefore, inference can also be performed on a CPU without significant drawbacks. To ensure a comprehensive evaluation, we conducted four key experiments, each designed to analyze a specific aspect of converting natural language commands into U-XACML policies:

1. **Baseline performance evaluation:** We tested both LLaMA8b and Mistral7b on the *test dataset*, generated as described in Section 5.
2. **Robustness analysis:** The models' ability to handle noisy input was assessed using a *mistaken dataset* [38], created by introducing typographical errors into the test dataset.
3. **Generalization test:** The models' adaptability was evaluated by applying them to a different domain, transitioning from a smart home environment to a *smart office* setting. This required constructing a new test dataset (the *smart office dataset*) with a custom taxonomy.
4. **Real-world validation:** Model-generated policies were compared against expert-defined policies using a UCS to assess practical applicability.

The evaluation of the first three test suites relies on four standard metrics: *Accuracy*, *Precision*, *Recall*, and *F1 Score* [39,40]. These metrics assess the correctness of JSON policies generated by our model (*model-generated policies*) by comparing them against JSON policies from the testing dataset (*reference policies*). For these tests, we use the *flattened version* of the JSON policies (see Section 4.2.1) and define the following terms:

- **True positives (TP):** number of key-value pairs that are an *exact match* between the model-generated policy and the reference policy.
- **False positives (FP):** number of key-value pairs missing from the reference policy but present in the model-generated policy (i.e., incorrectly added by the model).
- **False negatives (FN):** number of key-value pairs missing from the model-generated policy but present in the reference policy (i.e., incorrectly omitted by the model).

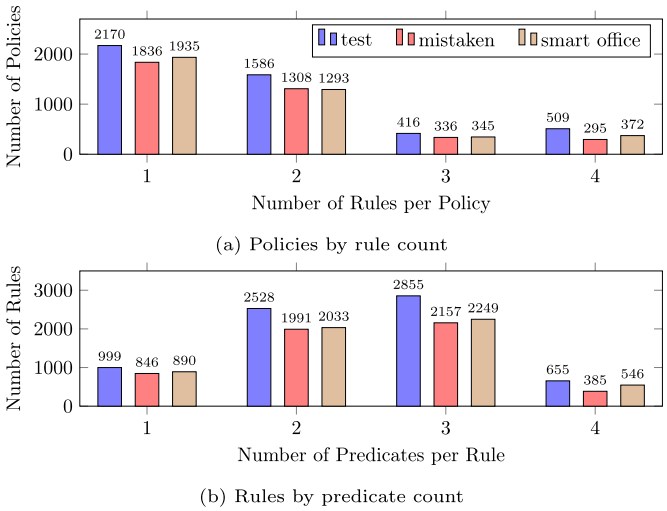
Since all keys are predefined in structured key-value extraction, True Negatives (TN) are not meaningful [41]. Including TN would inflate accuracy due to the large number of unused keys, providing a misleading evaluation of model performance. A key-value pair is considered *false* if it is not an exact match between the model-generated policy and the reference policy. This includes cases where the key is present but assigned a different value. Using these definitions, the evaluation metrics are formally defined as follows:

$$\text{Accuracy} = \frac{TP}{TP+FP+FN}, \qquad \text{Precision} = \frac{TP}{TP+FP},$$

$$\text{Recall} = \frac{TP}{TP+FN}, \qquad \text{F1 Score} = \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}},$$

where *Accuracy* measures the overall correctness of model-generated policies, *Precision* indicates the proportion of correct key-value pairs among those included in the model-generated policies, *Recall* assesses the model's ability to capture all correct key-value pairs present in the reference policies, *F1 Score* provides a balanced measure of precision and recall.

Fig. 5 provides an overview of the policies distribution based on the number of rules (Fig. 5a) and the rules distribution based on the number of predicates (Fig. 5b) across all three datasets (test, mistaken, and smart office). Fig. 5a illustrates the distribution of policies based on the number of rules they contain in each dataset. For instance, in the *test dataset*, 2170 policies (46.4 %) contain only one rule, 1586 (33.9 %) have two rules, 416 (8.9 %) have three rules, and 509 (10.8 %) have four rules. Fig. 5b presents the distribution of rules based on the number of predicates they contain. Most rules contain multiple predicates rather than a single one. In the *test dataset*, 999 rules contain only one predicate, while 2528 have two, 2855 have three, and 655 have four.



(a) Policies by rule count

(b) Rules by predicate count

Fig. 5. Comparison of policies and rules across datasets.

6.1. Evaluating the translation of natural language commands into U-XACML policies

The first evaluation is performed with the *test dataset*, obtained through the dataset creation pipeline described in Section 5 for the smart home domain. The aim of this test is to establish a baseline for the model's ability to accurately translate unstructured natural language commands into valid U-XACML policies. As discussed, the evaluation metrics in the first three test suites are computed using the flattened version of JSON policies. However, we argue that this approach is equivalent to evaluating U-XACML directly. Since U-XACML policies are deterministically derived from the JSON policies, and the flattened version is an intermediate product of the conversion module (see Section 4.2), the results remain consistent across both representations.

Table 1a summarizes the results for both LLaMA8b and Mistral7b models in terms of accuracy, precision, recall, and F1 score across different rule counts. For reference policies with one or two rules, LLaMA8b consistently outperforms Mistral7b in all metrics, although both models exhibit strong results. LLaMA8b achieves nearly perfect accuracy and F1 scores when handling three or four rules. Among the factors influencing performance, the number of rules within a policy consistently emerges as the parameter with the greatest impact on results, with a general trend of improved performance metrics as the number of rules increases, particularly for LLaMA8b.

Overall, LLaMA8b maintains a higher performance across all rule counts, with an overall accuracy of 92.86 % compared to the overall accuracy of Mistral7b of 83.77 %. This highlights LLaMA8b's better generalization and consistency in policy translation, whereas Mistral7b, despite performing well, shows more stability in accuracy and precision as complexity increases.

6.2. Testing the performance and robustness of Text2Policy on noisy commands

The objective of this experiment is to evaluate the model's performance when processing commands that contain errors, such as typos and formatting inconsistencies. Additionally, we aim to assess the model's *robustness*—its ability to generate correct U-XACML policies despite input perturbations. To this end, we introduce the *mistaken dataset*, which includes natural language commands intentionally altered with typographical and minor syntactic deviations. This setup simulates real-world scenarios, where users may unintentionally introduce mistakes. The robustness is measured by comparing the model's performance on the *mistaken dataset* with its performance on the original, correctly

Table 1
Performance comparison of LLaMA8b and Mistral7b models across different rule counts and scenarios. Evaluation metrics (accuracy, precision, recall, F1 score) are presented for both models on the *test dataset*, *mistaken dataset* (with noisy inputs), the *smart office dataset*, and the *HRE dataset*.

# Rules	Total	Accuracy		Precision		Recall		F1 Score	
	Elements	LLaMA8b	Mistral7b	LLaMA8b	Mistral7b	LLaMA8b	Mistral7b	LLaMA8b	Mistral7b
1	2170	0.9120	0.8665	0.9418	0.9082	0.9500	0.9124	0.9471	0.9106
2	1586	0.9123	0.8398	0.9365	0.9141	0.9487	0.9012	0.9471	0.8960
3	416	0.9865	0.9607	0.9872	0.9852	0.9901	0.9634	0.9931	0.9736
4	509	0.9966	0.9350	0.9988	0.9841	0.9976	0.9481	0.9982	0.9615
Overall	4681	0.9286	0.8377	0.9508	0.8982	0.9591	0.9074	0.9572	0.8915

(a) Performance of *Text2Policy* on translating commands into JSON policies with the *test dataset*.

# Rules	Total	Accuracy		Precision		Recall		F1 Score	
	Elements	LLaMA8b	Mistral7b	LLaMA8b	Mistral7b	LLaMA8b	Mistral7b	LLaMA8b	Mistral7b
1	1836	0.8974	0.8604	0.9291	0.9022	0.9403	0.9125	0.9377	0.9051
2	1308	0.9080	0.8365	0.9342	0.9103	0.9470	0.8956	0.9439	0.8932
3	336	0.9631	0.9512	0.9700	0.9763	0.9810	0.9654	0.9791	0.9677
4	295	0.9872	0.9371	0.9950	0.9920	0.9935	0.9537	0.9925	0.9656
Overall	3775	0.9139	0.8169	0.9397	0.8835	0.9501	0.9023	0.9478	0.8773
R_s^M	3775	0.9842	0.9752	0.9883	0.9836	0.9906	0.9944	0.9902	0.9841

(b) Performance of *Text2Policy* on translating commands into JSON policies with the *mistaken dataset*.
The last row presents the overall robustness score (R_s^M) for all evaluation metrics.

# Rules	Total	Accuracy		Precision		Recall		F1 Score	
	Elements	LLaMA8b	Mistral7b	LLaMA8b	Mistral7b	LLaMA8b	Mistral7b	LLaMA8b	Mistral7b
1	1935	0.8445	0.7900	0.8885	0.8306	0.9004	0.8325	0.8969	0.8494
2	1293	0.8367	0.7971	0.8875	0.8754	0.8923	0.8542	0.8861	0.8650
3	345	0.9624	0.9381	0.9656	0.9615	0.9704	0.9541	0.9722	0.9562
4	372	0.9822	0.9193	0.9937	0.9662	0.9920	0.9342	0.9897	0.9463
Overall	3945	0.8657	0.7292	0.9057	0.8064	0.9202	0.8421	0.9096	0.8101

(c) Performance of *Text2Policy* on translating commands into JSON policies with the *smart office dataset*.

# Rules	Total	Accuracy		Precision		Recall		F1 Score	
	Elements	LLaMA8b	Mistral7b	LLaMA8b	Mistral7b	LLaMA8b	Mistral7b	LLaMA8b	Mistral7b
1	1912	0.7855	0.7304	0.8563	0.7945	0.8687	0.8411	0.8612	0.8105
2	1173	0.8043	0.7584	0.8764	0.8319	0.8890	0.8620	0.8804	0.8422
3	227	0.8694	0.8668	0.9166	0.8988	0.9318	0.9329	0.9231	0.9115
4	531	0.8883	0.8585	0.9428	0.9184	0.9328	0.9060	0.9361	0.9108
Overall	3843	0.8104	0.7647	0.8780	0.8292	0.8875	0.8619	0.8811	0.8400

(d) Performance of *Text2Policy* on translating commands into JSON policies with the *HRE dataset*.

formatted commands in the *test dataset*. A higher robustness score indicates that the model's performance remains stable despite the presence of input errors.

To create the *mistaken dataset*, we used LLaMA8b to insert typographical errors and minor syntactic changes into the test set of command instances described in Section 6.1. These changes were carefully designed to alter the textual representation while preserving the original semantics of the commands. The modifications do not affect the taxonomy values present in the commands. Instead, they are applied only to the remaining parts of the command, including auxiliary words, grammatical structures, and minor typographical variations.

After we introduced typographical errors in the *test dataset*, we applied the *PoLL* validation process (also utilized in Section 5), which eliminates any altered commands (along with their associated directives) that, with the inserted mistakes, no longer align with their intended meaning, according to the jury of LLMs. This process yielded a final dataset, the *mistaken dataset*, comprising 3775 instance pairs.

In our threat model, the entities responsible for defining security policies are trusted users, such as the homeowner in a residential setting or a department safety & cybersecurity officer of a company. These users are not expected to input adversarial or malicious commands. As such, our performance evaluation focuses on realistic, unintentional input variations—such as typos and ambiguous phrasing—rather than on defending against deliberate harmful inputs.

Table 1b presents the performance metrics for both models when evaluated on the *mistaken dataset*. LLaMA8b achieves an overall accuracy of 91.39 % , while Mistral7b reaches an overall accuracy of 81.69 %. For overall precision, LLaMA8b attains 93.97 % compared to 88.35 % for Mistral7b, whereas overall recall values are 95.01 % and 90.23 %, respectively.

Both models show better performance as the number of rules increases. For policies with three or more rules, LLaMA8b maintains high precision and recall, reflecting its ability to generate accurate U-XACML policies despite input perturbations. Mistral7b is still effective as well, this suggests that Mistral7b also can handle the noise in the input text.

To quantify the *robustness*, we compute the relative variation in all evaluation metrics (accuracy, precision, recall, and F1 score) between the *test dataset* and the *mistaken dataset*. We define the *robustness score* R_s^M for a given metric M as follows:

$$R_s^M = 1 - \frac{|M_{test} - M_{mistaken}|}{M_{test}},$$

where M_{test} is the metric value obtained on the *test dataset*, and $M_{mistaken}$ is the corresponding value on the *mistaken dataset*. A robustness score closer to 1 indicates minimal degradation in performance due to noise.

The last row of Table 1b presents the results for the robustness score. LLaMA8b achieves robustness scores of 0.9842 for accuracy, 0.9883 for precision, 0.9906 for recall, and 0.9902 for F1 score. In comparison, Mistral7b attains 0.9752 for accuracy, 0.9836 for precision, 0.9944 for recall, and 0.9841 for F1 score. These values confirm that LLaMA8b consistently retains higher performance across all metrics, showing its better robustness to noisy inputs.

Performance across error impact intervals. Table 1b presents the overall performance results of the LLaMA8b and Mistral7b models on the *mistaken dataset*, which consists of command instances exhibiting varying degrees of errors—ranging from minor deviations to significant input corruption. This section explores and analyzes the models' performance across these different levels of input errors within the *mistaken dataset*. To quantify the impact of these errors, we introduce the concept of *Error Impact*, which is defined using the *Levenshtein Distance* [42], denoted by d_L, as described in by the following equation:

$$\text{Error Impact} = 100 \cdot \frac{d_L(\texttt{original}, \texttt{mistaken})}{\max\left(|\texttt{original}|, |\texttt{mistaken}|\right)}.$$

In this equation, the `original` input refers to the correctly formatted commands from the *test dataset* (introduced in Section 6.1). The `mistaken` input represents the same command after intentional modifications to introduce errors.

The Levenshtein Distance measures how different the mistaken command is from its original counterpart by counting the minimum number of single-character edits (insertions, deletions, or substitutions) required to transform one into the other. The formula then normalizes this value by dividing it by the length of the longer command, scaling it to a percentage.

While Table 1b aggregates performance across all error levels, we further examine how accuracy, precision, recall, and F1 score vary with different degrees of input errors. Specifically, we divide the dataset into error impact intervals, such as 10–20, 20–30, up to 70+, and analyze the models' performance within these intervals. Fig. 6 illustrates this breakdown, with the x-axis showing the error impact intervals, while the y-axis displays the evaluated metric values.

LLaMA8b consistently outperforms Mistral7b in accuracy and precision across all error intervals, with the gap widening as errors increase. In the 10–20 error interval, LLaMA8b achieves 96.2 % accuracy vs. Mistral7b's 90.6 %, while in the 70+ range, it maintains 85.5 % vs. 77.3 %. For recall and F1 score, a similar trend emerges. In the 10–20 interval, LLaMA8b attains 98.7 % recall and 97.9 % F1 score, surpassing Mistral7b's 92.9 % and 94.2 %. This highlights LLaMA8b's robustness, particularly under high-error conditions. These results demonstrate that the proposed method achieves high accuracy and robustness and maintains them even under increasing input corruption, highlighting its reliability in processing noisy real-world user commands. Overall, both models perform well with minimal errors, but LLaMA8b maintains stronger robustness as error impact increases, reinforcing the importance of robustness in real-world noisy data handling.

6.3. Testing the generalization capabilities of Text2Policy in smart office and healthcare residence for the elderly environments

To evaluate the model's ability to generalize across different domains, we conduct a similar domain adaptation experiment in which the test set is generated using the *smart office taxonomy* instead of the smart home taxonomy. Although the smart office domain introduces different entities and attributes, it remains conceptually similar. To further assess the model's generalization, we conduct an additional domain adaptation experiment using the *Healthcare Residence for the Elderly (HRE) taxonomy*. Unlike the smart home or smart office domains, the HRE setting introduces a diverse set of attribute values specific to institutional healthcare environments. This domain is characterized by subject profiles, medical devices, and related actions significantly diverging from the those of the original smart home dataset, which the model was trained on.

This setup allows us to assess the model's capability to transfer learned knowledge from one domain to another without additional fine-tuning. For each of the two domains, we perform a dataset creation workflow that follows a pipeline similar to that of the smart home dataset presented in Section 5.1. Specifically, it shares Steps 1a and 1b of Fig. 4, leading directly to Step 2, where the same set of template pairs T from the smart home scenario is reused. In this step, placeholders within these templates are replaced with values from the specific taxonomy under consideration (i.e., either the smart office or the HRE taxonomy), generating instance pairs specific to the respective domain.

After placeholders are replaced—unlike in the smart home scenario—all generated instance pairs are allocated to the *Raw test set* (see Fig. 4), since no training set is needed to train or re-train our *Text2Policy* model.

Finally, to enhance variability and ensure semantic correctness of all instance pairs, Steps 3a and 3b are performed. The resulting two test sets—the *smart office dataset*, consisting of 3945 instance pairs, and *the*

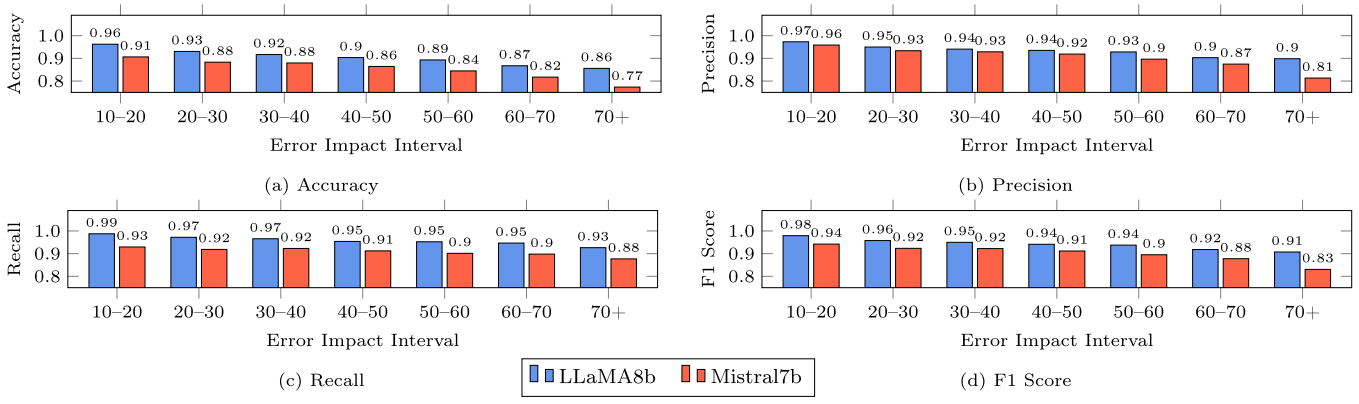


Fig. 6. Performance metrics for LLaMA8b and Mistral7b evaluated across different error impact intervals in the *mistaken dataset*.

HRE dataset, consisting of 3843 instance pairs—serve as the basis for evaluating the model's domain adaptation capability.

Table 1c highlights the performance of LLaMA8b and Mistral7b in the smart office domain across various metrics, showcasing LLaMA8b's superior ability to generalize and adapt. In this domain, LLaMA8b achieves an overall accuracy of 86.57 %, precision of 90.57 %, recall of 92.02 %, and F1 score of 90.96 %, consistently outperforming Mistral7b, which records 72.92 %, 80.64 %, 84.21 %, and 81.01 % for the same metrics.

Table 1d presents the models' performance in the HRE domain across various metrics, where LLaMA8b similarly excels, achieving an overall accuracy of 81.04 %, precision of 87.80 %, recall of 88.75 %, and F1 score of 88.11 %, compared to Mistral7b's 76.47 %, 82.92 %, 86.19 %, and 84.00 %. This evaluation confirms the method's accuracy and its ability to generalize across related domains with different taxonomies.

By comparing results for the models, we observe that the performance gap between them widens as the policy complexity increases. For example, in the smart office scenario, for policies with one or two rules, LLaMA8b demonstrates strong performance with F1 scores of 89.69 % and 88.61 %, while Mistral7b achieves 84.94 % and 86.50 %, respectively. In more complex cases, such as policies with four rules, LLaMA8b achieves high metrics, including an F1 score of 98.97 %, compared to Mistral7b's 94.63 %. These results underscore LLaMA8b's robustness in handling complex domain-specific rules. The good performance across both simple and complex policies in new domains further confirms the method's robustness and its capacity for zero-shot generalization. However, the performance of testing the model on the HRE domain is noticeably lower than in the smart home and office domains. This degradation occurs because HRE introduces distinct device types, contextual attributes, and user roles that differ significantly from those seen during training.

6.4. Validation of model-generated policies against expert-defined policies

Ensuring that a policy generated by our model accurately reflects the intended authorization rules specified in a natural language command is essential for its reliability. To assess the correctness of a model-generated U-XACML policy, we evaluate whether it produces the same authorization decisions as an expert-written U-XACML policy for the same command.

Our evaluation process involves systematically comparing policy pairs—one generated by the model and one written by an expert—by executing a predefined set of UCON requests and analyzing the authorization decisions returned by the UCS[1], specifically the PDP. A model-generated policy P_m and an expert-defined policy P_e are considered *equivalent* within R if they produce identical authorization decisions for every UCON request in the given set R. Formally:

$$P_m \equiv P_e \quad \Leftrightarrow \quad \forall r \in R,\ \ \mathrm{UCS}(P_m, r) = \mathrm{UCS}(P_e, r).$$

If P_m is not fully equivalent to P_e, we assess their degree of agreement using the *decision agreement rate* (DAR). The DAR represents the percentage of UCON requests for which the model-generated policy produces the same authorization decision as the expert-defined policy when evaluated by the UCS. It is calculated as:

$$\mathrm{DAR}(P_m, P_e, R) = \frac{|\{r \in R \mid \mathrm{UCS}(P_m, r) = \mathrm{UCS}(P_e, r)\}|}{|R|}.$$

This metric provides a clear method for evaluating the model's alignment with expert-defined reference policies, which act as the authoritative standard or "*ground truth*" for authorization decisions.

Experimental setup and UCON request generation. This set of experiments involves 31 policies defined by experts in two forms: a natural language description and its corresponding U-XACML representation (denoted as P_e). These policies were carefully selected to cover a broad range of logical structures, including simple conditions as well as nested expressions combining AND and OR operators. Each expert-defined policy is evaluated against a set of systematically generated UCON requests.

The generation of UCON requests is based on a two-step process: attribute extraction and systematic value expansion. This process establishes the foundation for generating UCON requests by identifying relevant attributes and defining a set of possible values for each. Essentially, it determines the domain within which the UCON requests are constructed.

Attribute extraction is an automated process that, given a policy P_e, identifies and extracts all `attributeIds`. For each attribute $\texttt{attributeId}_i$, it retrieves the set V_i of its associated values, i.e., the values used to define the constraints on that attribute in the policy P_e. We refer to this the set of pairs $\langle \texttt{attributeId}_i, V_i \rangle$ as the *base attribute-value set* S. Since this step only retrieves the specific values present in the policy, a *systematic value expansion* step is applied to improve coverage. For each attribute $\texttt{attributeId}_i$ identified in the previous step, the set of attribute values V_i is manually expanded by including additional, meaningful alternatives, thus resulting in the expanded set V_i^x. We call this expanded set of pairs $\langle \texttt{attributeId}_i, V_i^x \rangle$ the *extended attribute-value set* S^x.

The objective is to ensure that the set of UCON requests we generate largely explores the complexity of each policy, covering a diverse set of realistic access scenarios. During systematic value expansion, the additional values are carefully selected to be meaningful within the context of the expert-defined policy. For instance, if a policy includes a predicate involving the attribute *num* such as "*num greater than or equal to 1*", the automated extraction initially retrieves only the value *1*. To improve coverage, the set is manually expanded with values such as *0* (to test boundary conditions), *2* (to verify standard compliance), and `MAX_VALUE / MIN_VALUE` (to examine domain limits). Similarly, if a policy contains a boolean predicate involving the attribute *isInternal* like "*isInternal equal to true*", the complementary value, *false*, is added to ensure both possible states are tested. Once the attributes and their expanded values have been determined, the set of UCON requests R^x is generated by systematically combining these values in all possible ways. As a result, the generated UCON requests effectively test the policy's decision logic, thereby enhancing the reliability of the evaluation compared to having the set of UCON requests built from the base attribute-value set S.

Each expert-defined policy P_e is then tested against the set of UCON requests R^x generated for that policy during the previous step, having the UCS as policy enforcement engine. The testing follows a complete UCON workflow (see Section 2.2), which involves sending a *tryAccess* message to initiate the access evaluation. If the response from the UCS is `Permit`, a *startAccess* message is issued, and if the response to this second request is also `Permit`, an *endAccess* message is sent. If at any point a `Deny`, `Indeterminate`, or `NotApplicable` response is received, the workflow is halted. All responses from the UCS are recorded for subsequent analysis.

After evaluating the expert-defined policies, which serve as the ground truth, we assess the accuracy of our model in translating the natural language policy descriptions into U-XACML policies. The same set of 31 policy descriptions is provided as input to the model, which generates corresponding policies, denoted as P_m, and referred to as model-generated policies. While the model's output may differ syntactically from the expert-defined policies, the key evaluation criterion is whether they produce the same authorization decisions. To verify this, the model-generated policies undergo the same test process as the expert-defined ones, using the same set of UCON requests R^x and following the identical UCON workflow.

Finally, the responses obtained with the expert-defined policies and the model-generated policies are compared.

[1] https://sssg-dev.iit.cnr.it/marco-rasori/new-ucs

Table 2
Decision agreement rate for LLaMA8b and Mistral7b.

Pol.#	$\lvert R^x \rvert$	DAR(P_m, P_e, R^X) LLaMA	Mistral
1	64	1.0000	1.0000
2	128	1.0000	1.0000
3	640	1.0000	1.0000
4	72	1.0000	1.0000
5	24	1.0000	1.0000
6	12	1.0000	1.0000
7	20	1.0000	1.0000
8	8	1.0000	0.8750
9	8	1.0000	0.8750
10	60	1.0000	0.9166
11	60	1.0000	1.0000
12	16	1.0000	1.0000
13	16	1.0000	1.0000
14	12	1.0000	1.0000
15	120	1.0000	0.8583
⋮	⋮	⋮	⋮
16	36	1.0000	0.9444
17	4	1.0000	0.2500
18	6	1.0000	0.3333
19	42	1.0000	0.8333
20	32	1.0000	0.9062
21	16	1.0000	1.0000
22	16	1.0000	1.0000
23	10	1.0000	1.0000
24	28	1.0000	1.0000
25	4	1.0000	0.5000
26	24	1.0000	0.7083
27	16	0.8125	0.2500
28	60	1.0000	0.6833
29	20	0.0000	0.4500
30	120	1.0000	0.7500
31	140	0.9500	0.8714
Total	1848	0.9838	0.9232

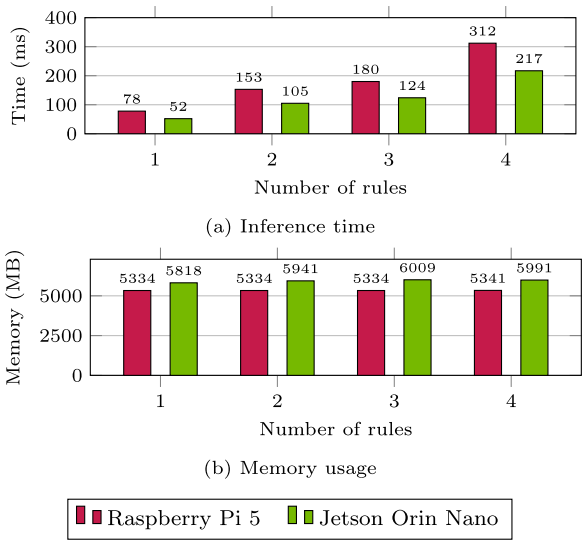


Fig. 7. Inference time and memory usage of the LLaMA8b-based *Text2Policy* model on Raspberry Pi 5 and Jetson Orin Nano emulator, measured for policies containing one to four rules.

Evaluation results. Table 2 shows the decision agreement rate DAR(P_m, P_e, R^x), obtained for each policy by both LLaMA8b and Mistral7b models. For LLaMA8b, nearly all policies (28 out of 31) achieve a perfect DAR of 1.0000, meaning that the model-generated policy P_m is equivalent to the expert-defined policy P_e, i.e., all $|R^x|$ evaluations produced the same authorization decision for both P_m and P_e. This high consistency indicates that LLaMA8b is particularly effective at capturing the intended authorization logic across a broad range of policies. Its ability to produce semantically equivalent policies across varied inputs further supports the accuracy and the robustness of the proposed method in faithfully interpreting natural language commands. In contrast, Mistral7b achieves equivalence for fewer policies (less than half), with several policies exhibiting high DAR, albeit below 1.0000. This indicates that, while there are some discrepancies, the model-generated mimics the expert-defined policy's logic well for most cases. As expected, LLaMA8b outperforms Mistral7b, achieving equivalence for more policies. The larger number of parameters of LLaMA8b likely contributes to better generalization and a stronger ability to capture complex patterns in policy translation [43,44]. Additionally, LLaMA8b's pre-training data or tokenizer may be better aligned with the syntactic structures of the JSON format, further enhancing its performance. The overall DAR values (0.9838 for LLaMA8b and 0.9232 for Mistral7b) highlight the strong performance of both models, with LLaMA8b outperforming Mistral7b in terms of equivalence.

6.5. Testing on-device performance

To assess the practical feasibility of running the *Text2Policy* model on resource-constrained hardware, we evaluated its inference performance on two representative devices: a Raspberry Pi 5 and a Jetson Orin AGX development kit configured to emulate a Jetson Orin Nano. In the latter case, both the available RAM and number of CPU cores were limited to match the Nano's specifications (8GB RAM, 6 CPU cores), and both GPU and CPU frequencies were capped to reflect the Nano's lower performance profile.

While earlier sections evaluated both LLaMA8b and Mistral8b, we focus here on LLaMA8b due to its superior accuracy and generalization performance. As previously described, the model was quantized to 4-bit precision to reduce its memory requirements and enable efficient execution on constrained hardware.

Inference on Raspberry Pi 5, which does not feature a GPU, was made possible using the Llama.cpp[2] framework. The model was first converted to the GGUF format and quantized to the q4_K_M version to reduce memory and computational requirements, then run entirely on CPU.

The same quantized GGUF model was also deployed on the Jetson Orin Nano emulator, which features a constrained GPU. In this case, the entire model was loaded into the GPU memory, leveraging Llama.cpp's GPU backend for faster inference while still staying within the tight hardware limits of the device. This setup enabled inference within the constraints of the device's available memory and processing power.

Fig. 7 shows the inference time and the memory usage across varying number of rules in the policy. Each bar corresponds to the average inference result obtained from evaluating 30 distinct policies. Specifically, we tested 30 policies with one rule, 30 policies with two rules, and so on, up to four rules, in order to assess how performance scales with policy complexity.

Fig. 7a reports the inference time and shows that the Jetson Orin Nano emulator consistently achieves lower inference times than the Raspberry Pi 5, ranging from 52 ms to 217 ms compared to 78 ms to 311 ms, respectively. Although inference time on the Raspberry Pi is higher, it remains acceptable given that the policy generation is not a real-time task.

Memory usage, shown in Fig. 7b, remains largely constant regardless of the rule count. The Raspberry Pi maintains a usage of around 5.3GB, while the Jetson Orin Nano emulator shows slightly higher usage, reaching just over 6GB. Memory consumption is therefore dominated by model loading rather than policy complexity.

We also measured the energy consumption of the Raspberry Pi 5 during inference. Results indicate a modest increase in energy use as the number of rules in the policy grows. Specifically, the average instantaneous current remains relatively stable during the inference process (around 1670 mA), while average energy consumption required for executing the inference process of one policy increases from 38 mAh for one rule to 153 mAh for four rules. This indicates that the rise in energy consumption with more complex policies is primarily due to longer inference durations, as the current draw remains nearly constant.

Among various factors influencing performance, the number of rules within a policy consistently emerges as the parameter with the greatest impact on results, highlighting its critical role in performance

[2] https://github.com/ggml-org/llama.cpp

variation, especially in the time required to generate the policy (inference time), and hence in the energy consumption. These results indicate that the *Text2Policy* model can operate within the constraints of limited hardware resources, while still maintaining reasonable inference performance.

7. Related work

Due to the complexity of XACML, automating policy generation has become an area of active research. Several approaches have been proposed to facilitate this process. One of these approaches, proposed in [7,8], uses user-friendly graphic policy editors. These editors replace the complex text of the policy with a block-based visual interface, which helps policy creators by offering visual feedback, organizing compatible elements, and guiding users through the policy-building process. This approach retains XACML's core logic and structure while making it more user-friendly [7]. Fatemian et al. [8] proposed Dual-XACML a Domain-Specific Modeling Language designed to generate ABAC and RBAC policies. Their tool assists users in creating XACML policies using graphical notations. Raschke and Zickau [9] proposed a Template-Based Policy Generation approach. This approach focuses on defining reusable policy patterns to simplify the requirements of large and complex attribute-based policies, improving the efficiency and consistency of policy creation. However, utilizing these methods still requires a certain level of technical knowledge. Current natural language policy-generating methods primarily assist administrators in accurately extracting access control policy components such as resources, subjects, actions, and environments from high-level requirement specification documents, thereby reducing human errors [45,46]. Shallow parsing techniques are used to extract policy components based on a predefined controlled grammar for structuring policies in a clear format. Xiao et al. [10] proposed a text-to-policy approach that uses syntactic pattern matching to identify Natural Language Algorithmic Compliance Policies. The developed approach used shallow parsing based on four semantic patterns to extract policy components. This method leverages predefined grammar rules, ensuring reliable identification of policy components [47,48]. However, unstructured natural language text challenges shallow parsers, because their variability makes it difficult to apply fixed grammar rules for accurate entity extraction [49]. Narouei et al. [11] propose approaches to attribute-based access control that automate the extraction of machine-readable security policies from high-level requirements specification documents using the RNN algorithm. The framework utilizes pre-trained word embeddings to detect sentences containing access control policy content from a dataset of real-world policy documents. The proposed method outperformed some state-of-the-art models, such as SVM [11]. The authors of [50] proposed RAGent, a framework that leverages LLaMA to extract access control components from high-level specifications and translates them into structured policies with a verification-refinement method, ensuring that the generated policies are reliable; Administrators can easily convert these structured policies into XACML format. Shan et al. [51] proposed a deep learning-based approach to automatically generate ABAC policies from natural language documents. This approach used ChatGLM to extract access control statements, and then the ID-CNN-CRF model was used to annotate the attributes of the subject, object, and action of these statements. Another approach involves converting policy texts into different formats, such as dependency graphs and trees, to extract policy components based on these alternative presentations [12,13]. Alohaly et al. [13] proposed a method for converting a natural language policy (in sentence form) into a dependency tree based on grammatical structure, then using a deep learning algorithm to predict whether a word serves as an attribute, allowing them to extract subject and object attributes for ABAC. Due to the lack of available datasets, they created their own by artificially injecting attribute information into RBAC policies following the grammatical structure of sentences.

8. Conclusions and future work

In this paper, we presented a novel framework leveraging Large Language Models to automate the generation of U-XACML policies from natural language commands. Our approach bridges the gap between human-readable policy definitions and machine-readable access control rules, enabling non-expert users to efficiently define precise and usage control policies.

To evaluate the effectiveness of our *Text2Policy* model, we conducted extensive experiments across multiple dimensions, including accuracy, robustness to noisy inputs, generalization across domains, and real-world applicability. Our results demonstrate that the best model we used as *Text2Policy* (LLaMA8b), achieves 93 % accuracy and an F1 score of 96 % on test data, confirming its ability to generate policies with high precision. Furthermore, we assessed the model's robustness using a dataset containing typographical and syntactic variations, revealing that *Text2Policy* maintains strong performance even under input perturbations. Additionally, we validated the framework's adaptability by transitioning from a smart home to a smart office environment, demonstrating its capacity to generalize to new domains without the need for additional fine-tuning. We further validated the quality of generated policies by comparing authorization decisions with those derived from expert defined policies, observing a 98 % agreement rate. Finally, we assessed the feasibility of running *Text2Policy* on constrained hardware through on-device performance tests.

Future work will focus on extending our dataset generation pipeline by leveraging an ontology-driven approach to support automated policy generation across broader domains with minimal manual effort. Using this approach also enables the construction of different complex nested logical structures (e.g., AND, OR, NOT). We also aim to enhance the model's generalization and reasoning capabilities by integrating *reinforcement learning* [52] and *Chain-of-Thought (CoT) prompting* [53] into the training pipeline. Additionally, future work will focus on enhancing the model's robustness to linguistic variability, including handling inputs from non-native speakers, slang, and grammatically incorrect sentences. We also plan to explore interactive clarification techniques for ambiguous or incomplete commands, as well as support for additional languages.

Another promising direction is the integration of explainability techniques to improve transparency in model behavior. Methods such as attention visualization or saliency maps can help clarify how the model maps natural language commands to structured policy representation. This would support user trust and facilitate validation in security-sensitive domains, though it requires dedicated tooling and methodological design that we leave for future investigation.

CRediT authorship contribution statement

Loay Alajramy: Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Resources, Methodology, Investigation, Formal analysis, Data curation, Conceptualization; **Marco Simoni:** Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Resources, Methodology, Investigation, Formal analysis, Data curation, Conceptualization; **Marco Rasori:** Writing – review & editing, Writing – original draft, Validation, Supervision, Software, Methodology, Investigation, Data curation; **Andrea Saracino:** Writing – review & editing, Writing – original draft, Validation, Supervision, Project administration, Methodology; **Paolo Mori:** Writing – review & editing, Validation, Supervision, Project administration, Methodology, Investigation, Funding acquisition, Conceptualization.

Data Availability

Data will be made available on request.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] P. Colombo, E. Ferrari, Access control technologies for big data management systems: literature review and future trends, Cybersecurity 2 (1) (2019) 3. https://doi.org/10.1186/S42400-018-0020-9

[2] H. Wu, H. Han, X. Wang, S. Sun, Research on artificial intelligence enhancing internet of things security: a survey, IEEE Access 8 (2020) 153826–153848. https://doi.org/10.1109/ACCESS.2020.3018170

[3] L. Ardito, L. Barbato, P. Mori, A. Saracino, Preserving privacy in the globalized smart home: the SIFIS-home project, IEEE Secur. Privacy 20 (1) (2022) 33–44.

[4] A. La Marra, F. Martinelli, P. Mori, A. Saracino, Implementing Usage Control in Internet of Things: a Smart Home Use Case, in: Trustcom/BigDataSE/ICESS 2017, IEEE Computer Society, 2017, pp. 1056–1063.

[5] J. Park, R. Sandhu, The $UCON_{ABC}$ usage control model, TISSEC 7 (1) (2004) 128–174.

[6] K. Ragothaman, Y. Wang, B. Rimal, M. Lawrence, Access control for IoT: a survey of existing research, dynamic policies and future directions, Sensors 23 (4) (2023) 1805. https://doi.org/10.3390/S23041805

[7] H. Nergaard, N. Ulltveit-Moe, T. Gjøsæter, ViSPE: a Graphical Policy Editor for XACML, in: ICISSP 2015, 576 of Communications in Computer and Information Science, Springer, 2015, pp. 107–121.

[8] A. Fatemian, B. Zamani, M. Masoumi, M. Kamranpour, B.T. Ladani, S.K. Rahimi, Automatic Generation of XACML Code Using Model-driven Approach, in: ICCKE 2021, 2021, pp. 206–211.

[9] P. Raschke, S. Zickau, A Template-based Policy Generation Interface for RESTful Web Services, in: OTM 2014 Workshops, 8842 of *Lecture Notes in Computer Science*, Springer, 2014, pp. 137–153.

[10] X. Xiao, A.M. Paradkar, S. Thummalapenta, T. Xie, Automated Extraction of Security Policies from Natural-language Software Documents, in: SIGSOFT/FSE'12, ACM, 2012, p. 12.

[11] M. Narouei, H. Khanpour, H. Takabi, N. Parde, R.D. Nielsen, Towards a Top-down Policy Engineering Framework for Attribute-based Access Control, in: SACMAT 2017, ACM, 2017, pp. 103–114.

[12] J. Slankas, X. Xiao, L.A. Williams, T. Xie, Relation Extraction for Inferring Access Control Rules from Natural Language Artifacts, in: ACSAC 2014, ACM, 2014, pp. 366–375.

[13] M. Alohaly, H. Takabi, E. Blanco, A Deep Learning Approach for Extracting Attributes of ABAC Policies, in: SACMAT 2018, ACM, 2018, pp. 137–148.

[14] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F.L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat, et al., Gpt-4 technical report, arXiv preprint arXiv:2303.08774 (2023).

[15] D. Patterson, J. Gonzalez, U. Hölzle, Q. Le, C. Liang, L.-M. Munguia, D. Rothchild, D.R. So, M. Texier, J. Dean, The carbon footprint of machine learning training will plateau, then shrink, Computer 55 (7) (2022) 18–28.

[16] A.Q. Jiang, A. Sablayrolles, A. Mensch, C. Bamford, D.S. Chaplot, C.D. de las, F. Bressand, G. Lengyel, G. Lample, L. Saulnier, et al., Mistral 7B, arXiv preprint arXiv:2310.06825 (2023).

[17] T. Dettmers, A. Pagnoni, A. Holtzman, L. Zettlemoyer, Qlora: efficient finetuning of quantized LLMs, Adv. Neural Inf. Process. Syst. 36 (2023) 10088–10115.

[18] M.H. Daniel Han, U. team, Unsloth, 2023, http://github.com/unslothai/unsloth.

[19] P. Samarati, S. De Capitani di Vimercati, Access Control: Policies, Models, and Mechanisms, in: FOSAD 2000, 2171 of *Lecture Notes in Computer Science*, Springer, 2000, pp. 137–196. https://doi.org/10.1007/3-540-45608-2_3

[20] V.C. Hu, D.R. Kuhn, D.F. Ferraiolo, Attribute-based access control, Computer 48 (2015) 85–88. https://doi.org/10.1109/MC.2015.33

[21] Zolertia S.L., eXtensible Access Control Markup Language (XACML) Version 3.0 Plus Errata 01, 2017, http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-en.HTML.

[22] E. Carniani, D. D'Arenzo, A. Lazouski, F. Martinelli, P. Mori, Usage control on cloud systems, Future Gen. Comput. Syst. 63 (2016) 37–55.

[23] M. Yu, F. Li, N. Yu, X. Wang, Y. Guo, Detecting conflict of heterogeneous access control policies, Dig. Commun. Netw. 8 (5) (2022) 664–679. https://doi.org/10.1016/J.DCAN.2022.09.002

[24] X. Liang, L. Lv, C. Xia, Y. Luo, Y. Li, A Conflict-related Rules Detection Tool for Access Control Policy, in: Frontiers in Internet Technologies: Second CCF Internet Conference of China, ICoC 2013, Zhangjiajie, China, July 10, 2013, Revised Selected Papers, Springer, 2013, pp. 158–169.

[25] M. St-Martin, A.P. Felty, A Verified Algorithm for Detecting Conflicts in XACML Access Control Rules, in: Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs, 2016, pp. 166–175.

[26] B. Stepien, A. Felty, Resolving XACML Rule Conflicts Using Artificial Intelligence, in: Proceedings of the 3rd International Conference on Information Science and Systems, Association for Computing Machinery, New York, NY, USA, 2020, pp. 121–127. https://doi.org/10.1145/3388176.3388188

[27] H.B. Enderton, A mathematical introduction to logic, Academic Press, 1972.

[28] L. Yu, N.A. Madjid, D.E. Difallah, CrunchQA: a Synthetic Dataset for Question Answering over Crunchbase Knowledge Graph, in: Big Data 2022, 2022, pp. 4635–4641.

[29] D. Seyler, M. Yahya, K. Berberich, Knowledge Questions from Knowledge Graphs, in: SIGIR 2017, 2017, pp. 11–18.

[30] A. Formica, I. Mele, F. Taglino, A template-based approach for question answering over knowledge bases, Knowl. Inf. Syst. 66 (1) (2024) 453–479.

[31] L. Zheng, W.-L. Chiang, Y. Sheng, S. Zhuang, Z. Wu, Y. Zhuang, Z. Lin, Z. Li, D. Li, E. Xing, et al., Judging llm-as-a-judge with mt-bench and chatbot arena, Adv. Neural Inf. Process. Syst. 36 (2023) 46595–46623.

[32] P. Verga, S. Hofstatter, S. Althammer, Y. Su, A. Piktus, A. Arkhangorodsky, M. Xu, N. White, P. Lewis, Replacing judges with juries: evaluating llm generations with a panel of diverse models, arXiv preprint arXiv:2404.18796 (2024).

[33] B. Peng, J. Quesnelle, H. Fan, E. Shippole, Yarn: Efficient context window extension of large language models, arXiv preprint arXiv:2309.00071 (2023).

[34] A. Yang, B. Yang, B. Zhang, B. Hui, B. Zheng, B. Yu, C. Li, D. Liu, F. Huang, H. Wei, et al., Qwen2. 5 technical report, arXiv preprint arXiv:2412.15115 (2024).

[35] M. Abdin, J. Aneja, H. Behl, S. Bubeck, R. Eldan, S. Gunasekar, M. Harrison, R.J. Hewett, M. Javaheripi, P. Kauffmann, et al., Phi-4 technical report, arXiv preprint arXiv:2412.08905 (2024).

[36] L. von Werra, Y. Belkada, L. Tunstall, E. Beeching, T. Thrush, N. Lambert, S. Huang, K. Rasul, Q. GallouÃ©dec, TRL: Transformer Reinforcement Learning, 2020, (https://github.com/huggingface/trl).

[37] M. Saplin, Running Local LLMs: CPU vs GPU - A Quick Speed Test, 2024. https://dev.to/maximsaplin/running-local-llms-cpu-vs-gpu-a-quick-speed-test-2cjn/.

[38] A. Singh, N. Singh, S. Vatsal, Robustness of llms to perturbations in text, arXiv preprint arXiv:2407.08989 (2024).

[39] M.N. Nobi, M. Gupta, L. Praharaj, M. Abdelsalam, R. Krishnan, R. Sandhu, Machine learning in access control: a taxonomy and survey, arXiv preprint arXiv:2207.01739 (2022).

[40] J. Dagdelen, A. Dunn, S. Lee, N. Walker, A.S. Rosen, G. Ceder, K.A. Persson, A. Jain, Structured information extraction from scientific text with large language models, Nat. Commun. 15 (1) (2024) 1418.

[41] C.D. Manning, P. Raghavan, H. SchÃ¼tze, Introduction to information retrieval, Cambridge University Press, 2008.

[42] V.I. Levenshtein, Binary codes capable of correcting deletions, insertions, and reversals, Soviet Phys. Doklady 10 (1965) 707–710.

[43] J. Kaplan, S. McCandlish, T. Henighan, T.B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, D. Amodei, Scaling laws for neural language models, arXiv preprint arXiv:2001.08361 (2020).

[44] T. Henighan, J. Kaplan, M. Katz, M. Chen, C. Hesse, J. Jackson, H. Jun, T.B. Brown, P. Dhariwal, S. Gray, et al., Scaling laws for autoregressive generative modeling, arXiv preprint arXiv:2010.14701 (2020).

[45] Y. Xia, S. Zhai, Q. Wang, H. Hou, Z. Wu, Q. Shen, Automated Extraction of ABAC Policies from Natural-language Documents in Healthcare Systems, in: BIBM 2022, IEEE, 2022, pp. 1289–1296. https://doi.org/10.1109/BIBM55620.2022.9995559

[46] L. Yang, X. Chen, Y. Luo, X. Lan, L. Chen, Purext: automated extraction of the purpose-aware rule from the natural language privacy policy in IoT, Secur. Commun. Netw. 2021 (2021) 5552501:1–5552501:11.

[47] Z. Shen, N. Gao, Z. Liu, M. Li, C. Wang, Using Chinese Natural Language to Configure Authorization Policies in Attribute-Based Access Control System, in: SciSec 2021, 13005 of *Lecture Notes in Computer Science*, Springer, 2021, pp. 110–125.

[48] X. Liu, B. Holden, D. Wu, Automated Synthesis of Access Control Lists, in: ICSSA 2017, IEEE, 2017, pp. 104–109.

[49] S.H. Jayasundara, N.A. Gamagedara Arachchilage, G. Russello, SoK: Access control policy generation from high-level natural language requirements, ACM Comput. Surv. 57 (4) (2024). https://doi.org/10.1145/3706057

[50] S.H. Jayasundara, N.A.G. Arachchilage, G. Russello, Ragent: retrieval-based access control policy generation, arXiv preprint arXiv:2409.07489 (2024).

[51] F. Shan, Z. Wang, M. Liu, M. Zhang, Automatic generation of attribute-based access control policies from natural language documents, Comput. Mater. Continua 80 (3) (2024) 3881–3902.

[52] Z. Shao, P. Wang, Q. Zhu, R. Xu, J. Song, X. Bi, H. Zhang, M. Zhang, Y.K. Li, Y. Wu, et al., Deepseekmath: pushing the limits of mathematical reasoning in open language models, arXiv preprint arXiv:2402.03300 (2024).

[53] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q.V. Le, D. Zhou, et al., Chain-of-thought prompting elicits reasoning in large language models, Adv. Neural Inf. Process. Syst. 35 (2022) 24824–24837.