

SHACR: A Graph-Augmented Semi-Autonomous Framework for Multi-Class Conflict Resolution in Smart Home IoT Automation

Leena Marghalani, Walid Aljoby, *Member, IEEE*, Suayb S. Arslan, *Senior Member, IEEE*

Abstract—Smart home automation increasingly relies on user-defined rules across heterogeneous IoT devices. While these rules appear harmless in isolation, their concurrent execution creates hidden, cross-rule interactions via shared devices, environmental variables, and physical topology. These interactions result in unsafe, wasteful, or privacy-threatening behaviors that are completely invisible to text-only analysis. Existing conflict detectors remain siloed, catching either static syntactic conflicts or specific environment-mediated interactions without unifying the two or providing actionable repairs for non-expert users. We present SHACR, a smart home conflict resolution framework that anchors Large Language Model (LLM) unpredictability by grounding its reasoning in a formal, directed knowledge graph. SHACR encodes devices, capabilities, physical states, and Trigger-Condition-Action rules as typed, traversable entities. By elevating physical cause-effect relationships to first-class graph edges, SHACR transforms conflict detection from fragile text inference into deterministic multi-hop graph traversal, unifying logical, semantic, and physical conflict classes. It drives a closed-loop Scan-Explain-Repair-Validate workflow that uses the graph to bound the LLM’s action space. We evaluated SHACR on a testbed of 203 rules deployed across 70 apartments within a smart building. By holding the underlying LLM fixed and introducing SHACR’s knowledge graph, classification errors drop by 36.7%, F1 rises from 0.59 to 0.79, and few-shot calibration further lifts F1 to 0.95, whereas the same calibration barely helps a graph-free LLM. Ultimately, this work challenges the current AI paradigm, establishing that structured knowledge representation is a far more critical factor for dependable IoT automation management than prompt engineering or underlying model architecture.

Index Terms—Smart homes, Internet of Things, knowledge graphs, graph retrieval-augmented generation, large language models, trigger-condition-action rules, conflict detection, conflict resolution, IoT automation, autonomous systems, agentic RAG.

I. INTRODUCTION

SMART homes have become one of the most pervasive deployments of AI-enabled IoT, with residents routinely combining sensors, appliances, and cloud services to automate lighting, climate control, access control, and security monitoring [13]. Unlike enterprise IoT environments, smart homes are configured and maintained by end users—not trained administrators—often across multiple applications and

vendors [1], [5], [28], [32], [34]. As automation rule sets grow, routines that appear benign in isolation can interact in unsafe, wasteful, or privacy-threatening ways once they share rooms, devices, environmental variables, or cross-platform triggers [9], [13]–[15].

Consider a concrete failure scenario. An energy-saving rule turns off a living-room smart plug at night. A surveillance rule activates a security camera when the home enters *away* mode. If the camera draws power through that plug, the first rule silently disables the second. No rule text states “*turn off the camera*,” yet the home loses surveillance precisely when monitoring matters most. This failure is invisible to any analysis confined to rule syntax, because the dependency lives in the physical wiring, not in the logic of either rule.

The same failure mechanism recurs across many everyday configurations. A robot vacuum passing through the entrance hallway triggers the same motion sensor used to detect human presence, satisfying a precondition that unlocks the front door while the apartment is unoccupied. A heater rule raises room temperature past the threshold that opens a motorized window, which then cools the room and reactivates the heater—an energy-wasting feedback loop that neither rule individually encodes. A scheduled oven rule forces the appliance on while a temperature-safety rule attempts to shut it off, producing a thermal overload hazard through conflicting actuation. In each case, the conflict is not a programming error within any single rule; it is an emergent interaction across rules that were each authored independently and each appear reasonable alone.

The fundamental question this work addresses is whether, given a set of heterogeneous smart home automation rules, device placements, and environmental dependencies, it is possible to determine prior to deployment whether any subset of rules can jointly produce unsafe or unexpected behavior. It further asks whether such conflicts can be explained in terms that non-expert users can understand and act on. Existing approaches only partially address this issue. Formal rule-centric analyzers [9], [11] detect syntactic contradictions—two rules issuing incompatible commands to the same actuator—but miss conflicts mediated by device topology or environmental propagation. Semantic frameworks [6], [13], [15] model how device actions propagate through shared physical variables, but typically require manually specified device behavior models and provide neither cross-platform unification nor user-facing repair guidance. Recent LLM-based approaches [16]–[18] improve natural-language interaction but remain unreliable when critical topology and dependency information is absent from the prompt context.

Conflicts arise because reasoning, confined to rule text, is structurally blind to three qualitatively different failure

L. Marghalani and W. Aljoby are with the Department of Information and Computer Science, King Fahd University of Petroleum and Minerals (KFUPM), Dhahran 31261, Saudi Arabia (e-mail: g202303130@kfupm.edu.sa; waleed.algobi@kfupm.edu.sa;).

S. S. Arslan is a Professor in the Department of Computer Engineering and Director of the Institute for Data Science and AI, Boğaziçi University, Istanbul 34342, Turkey, and a Research Affiliate in the Department of Brain and Cognitive Sciences, Massachusetts Institute of Technology, Cambridge, MA, USA (e-mail: suayb.arslan@bogazici.edu.tr, sarslan@mit.edu).

Manuscript received XX Month XXXX; revised XX Month XXXX; accepted XX Month XXXX. This work was supported in part by [funding information]. (Corresponding author: Walid Aljoby.)

classes (§III), and these failures can be avoided if rules are reasoned over a structured representation that makes physical dependencies explicit and traversable (§IV). We therefore present SHACR (Smart Home Semi-Autonomous Conflict Resolution), a semi-autonomous framework built on *graph-grounded conflict detection*: the principle that detecting, explaining, and repairing automation conflicts requires anchoring LLM reasoning in a formal knowledge graph that encodes device placement, capability effects, environmental variables, and Trigger–Condition–Action (TCA) rule structure as typed, directed entities. By making physical cause-effect relationships first-class graph edges, SHACR enables detection of all three conflict classes through structured multi-hop traversal rather than text inference, and enables explanation by tracing the traversal path back to the specific devices and rules that caused the failure.

SHACR achieves this through a tightly integrated architecture that goes well beyond GraphRAG retrieval alone. It comprises a structured data ingestion pipeline that normalizes heterogeneous YAML rule specifications into the knowledge graph, a Neo4j-hosted knowledge graph store, a two-tier MCP server layer that exposes five purpose-built domain tools to the reasoning engine, a Scan–Explain–Repair–Validate workflow that orchestrates multi-step LLM inference over retrieved subgraphs, and a Streamlit dashboard that enforces human approval before any graph modification executes. Each component addresses a specific failure mode in the broader conflict management lifecycle.

We prototype SHACR on Neo4j AuraDB with a custom MCP server layer and a Streamlit user dashboard and evaluate it on a testbed consisting of 203 automation rules across 70 apartments, demonstrating an F1-score of 0.79 in a zero-shot configuration. This represents a 36.7% reduction in classification errors over an LLM-only baseline using the same model, which increases to 0.95 F1 with balanced few-shot prompting.

The principal contributions of this work are as follows.

1. We characterize three qualitatively distinct smart home conflict classes (logical, semantic, and physical) and show that each arises from a structurally different failure mechanism in IoT automation. We further demonstrate that existing approaches detect at most one of these classes, establishing the need for a unified, graph-grounded detection framework (§III).
2. We formalize the three conflict classes over a directed knowledge graph G and show that each maps to a specific traversal pattern over the AFFECTS edge—the physical cause-effect substrate that makes device-mediated interactions graph-detectable before deployment (§IV).
3. We design and prototype SHACR, a semi-autonomous conflict management system comprising a structured data ingestion pipeline, a typed knowledge graph schema, a GraphRAG pipeline with a five-tool MCP server layer, and a Scan–Explain–Repair–Validate workflow that proposes structured repairs within a constrained action space and verifies resolution against the live graph (§V–§VI).
4. We evaluate SHACR on a controlled testbed of 203 automation rules across 70 apartments and show a 36.7% reduction in total classification errors over an LLM-only

baseline using the same underlying model, an F1-score of 0.95 with balanced few-shot prompting, and present a cross-model comparison of Claude Sonnet 4 and Google Gemini under matched prompting conditions. To support reproducibility, the full 203-rule, 70-apartment testbed and ingestion code are released publicly (§VII).

The paper is organized as follows. Section II surveys related work. Section III presents background and motivation. Section IV formalizes the conflict taxonomy. Section V describes the system design. Section VI covers the prototype implementation. Section VII presents the evaluation. Section VIII discusses findings and future direction. Section IX concludes.

II. RELATED WORK

Logical conflict detection. IoTC² [11] applies formal methods to identify rule violations against predefined safety properties. TapChecker [9] uses SMT-based analysis for trigger-action conflicts. Fine-Grained Conflict Detection [7] and IoT-SAFE [13] extend this with service-level and policy-level formulations. These methods reliably surface direct contradictions but are structurally limited to syntactic rule-level analysis; conflicts mediated by device topology or environmental propagation remain invisible.

Semantic and hybrid detection. The IoTMon [10] introduces physical interaction channels. The IoT-Praetor [6] models the environmental influence. SeIoT [8] uses knowledge graphs for semantic anomaly detection, and contextual semantics of behavior-graphs have also been applied to anomaly detection in smart IoT systems [4]. The IoTMediator [15] and SAFE-TAP [3] address cross-platform threats and semantic embeddings, respectively. VISCR [12] and IoTIE [14] combine rule-based and contextual reasoning for cross-platform detection. AutoIoT [2] couples LLMs with formal verification for rule generation. Although these systems demonstrate that IoT devices interact through both cyber and physical channels, they rely on predefined behavior models, lack automated repair mechanisms, and provide limited user-facing explanation.

LLM-based and retrieval-augmented approaches. ChatIoT [19] integrates LLM reasoning with threat intelligence. Rey et al. [17] and Jahanbakhsh et al. [18] use retrieval for personalization and workflow composition. LLM-HA [16] provides explanation for HomeAssistant automations. These systems improve contextual reasoning but rely on textual descriptions and cannot reliably detect multi-hop or physically-mediated conflicts without an explicit relational model.

GraphRAG and agentic RAG. SHACR is a knowledge-based GraphRAG system [39] in which the knowledge graph is the primary carrier of domain semantics, enabling evidence aggregation along explicit relational paths rather than surface-level lexical similarity. Beyond static GraphRAG, SHACR corresponds to the *corrective single-agent* agentic RAG architecture [40]: a centralized reasoning agent iteratively refines its outputs through a structured repair-and-validate loop, grounded in a formal typed graph rather than unstructured document corpora, and gated by explicit user approval before any knowledge graph modification executes.

Positioning. SHACR differs from all prior work in encoding physical cause-effect relationships as first-class, traversable

TABLE I
CAPABILITY COMPARISON ACROSS REPRESENTATIVE SMART HOME
CONFLICT DETECTION FRAMEWORKS.

System	Year	Conflict Analysis				Capabilities			
		Logical	Semantic	Env. Effects	Multi-Plat.	Mitigation	Explain-able	LLM-based	Graph-Grounded
IoT ² [11]	2019	✓	✗	✗	✗	✗	✗	✗	✗
VISCR [12]	2019	✓	✓	✗	✓	✓	✗	✗	✗
IoTIE [14]	2019	✓	✓	✗	✓	✗	✗	✗	✗
IoT-Praetor [6]	2020	✗	✓	✓	✗	✗	✗	✗	✗
IoTSAFE [13]	2021	✗	✓	✓	✗	✗	✓	✗	✗
IoTMediator [15]	2023	✗	✓	✓	✓	✗	✓	✗	✗
AutoIoT [2]	2024	✓	✓	✗	✓	✓	✗	✓	✗
TapChecker [9]	2024	✓	✗	✗	✗	✗	✗	✗	✗
SAFE-TAP [3]	2025	✓	✓	✓	✗	✗	✗	✓	✗
LLM-HA [16]	2025	✗	✗	✗	✗	✗	✓	✓	✗
SHACR (ours)	2026	✓	✓	✓	✓	✓	✓	✓	✓

graph edges, and in combining structured retrieval, multi-class conflict detection, executable repair, and a validated semi-autonomous pipeline within a single integrated system. Table I maps capabilities across representative frameworks. No prior system simultaneously handles all three conflict classes across heterogeneous platforms while providing automated mitigation, grounded explanation, and LLM-based reasoning within a user-approved repair loop; all existing systems reason over rule text or predefined device models rather than a persistent typed knowledge graph that makes physical dependencies traversable.

III. BACKGROUND AND MOTIVATION

A. What Smart Home Automation Rules Do

A smart home automation system can be understood as a collection of reactive control policies, each expressed as a TCA rule. The trigger specifies an observable event or state transition—motion detected, temperature falling below a threshold, time reaching a scheduled point—that causes the rule to become eligible for evaluation. The condition constrains when the rule should actually fire, filtering eligibility by additional predicates such as the home mode (*Away*, *Night*) or a device’s current state. The action specifies the command issued once the rule fires: turning a device on or off, adjusting a set-point, sending a notification, or locking a physical actuator.

Individual rules are typically authored for specific, narrow purposes: an energy-saving schedule turns off idle devices at night; an occupancy rule enables lights when motion is detected; a safety rule locks the front door when no presence is reported [36]. Considered in isolation, each of these policies is self-consistent and straightforwardly correct. The difficulty arises when multiple rules, each authored independently by a non-expert user, begin to share devices, rooms, environmental variables, or derived contextual predicates. At that point, the interaction space grows in ways that are not visible from any individual rule’s text.

To make this concrete, consider the scenario illustrated in Figure 1. A smart apartment entrance has three rules in deployment. The first, *Robot Vacuum Daily Run*, schedules the vacuum to clean the apartment on a recurring basis. The second, *Entrance Presence*, sets a virtual presence flag whenever the entrance motion sensor detects movement. The third, *Unlock on Presence*, unlocks the front door whenever

that presence flag becomes active during daytime hours. Reading each rule individually, a resident sees three sensible automations: a cleaning schedule, a presence monitor, and a door-unlock convenience. Nothing in any single rule’s text indicates a problem.

The conflict, however, is real. During its scheduled cleaning cycle, the vacuum passes through the entrance hallway. As it does, it triggers the entrance motion sensor—the same sensor that the presence rule monitors. The presence rule activates the virtual flag. The unlock rule finds its daytime condition satisfied and issues an unlock command to the front door. The apartment is now unsecured, and no single rule bears the immediate cause: the failure exists only in the *composition* of the three rules across the shared sensor.

B. How the Knowledge Graph Makes Conflicts Visible

This failure trace illustrates why representation matters. A text-only analysis of the three rules sees three independent logical constructs that share no direct syntactic dependency. The vacuum rule targets the vacuum’s run capability; the presence rule targets the motion sensor state; the unlock rule targets the lock capability conditioned on the presence flag. Without an explicit model of the physical world, no analysis can determine that the vacuum’s movement activates the motion sensor, which corrupts the semantic precondition of the unlock rule.

Figure 1 shows how this failure becomes tractable once the rules are embedded in a knowledge graph. The *AFFECTS* edge from the vacuum’s *RUN* capability to the *MOTION_DETECTED* state node encodes the physical fact that the vacuum’s movement produces sensor activations. This single edge—absent from any rule’s text but present in any accurate model of the deployed environment—closes the causal chain: it makes the path from *vacuum running* → *motion sensor fires* → *presence flag set* → *front door unlocked* a traversable graph structure. Once that structure is explicit, the conflict is graph-detectable before deployment.

The knowledge graph serves three complementary roles in SHACR. As a *dependency map*, the *AFFECTS* edge explicitly encodes physical cause-effect relationships among device capabilities and environmental or state variables that are otherwise invisible to text-based analysis. As a *retrieval index*, apartment-level subgraph extraction gives the LLM a focused, contextually complete view of the relevant environment rather than a flat list of rule texts. As an *explanation structure*, detected conflicts correspond to traversal paths in the graph, so the system can report the specific nodes and edges responsible for the failure, producing explanations that users can verify against their own home configuration and use to decide on a repair.

C. How Conflicts Arise: Three Failure Mechanisms

The vacuum-door scenario is an instance of a broader taxonomy. Conflicts in smart home automation arise through three structurally distinct mechanisms, each requiring different reasoning to detect.

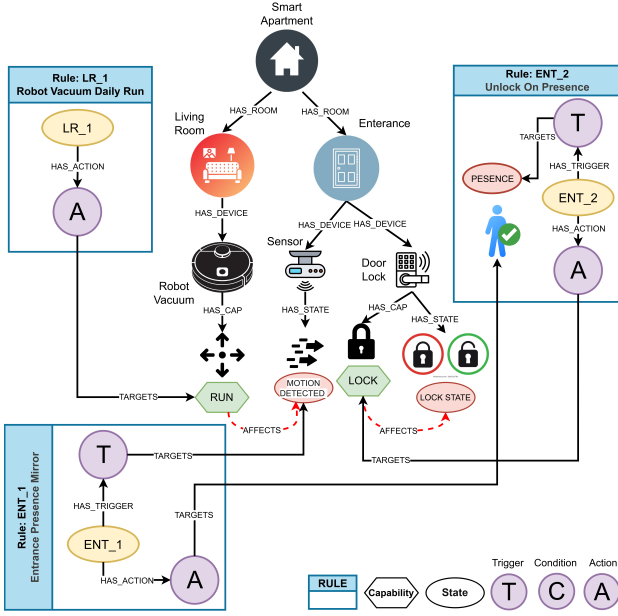


Fig. 1. Knowledge graph subgraph of the running example. Three rules—Robot Vacuum Daily Run, Entrance Presence, and Unlock on Presence—appear independent in rule text but form a traceable causal chain when embedded as graph entities. The AFFECTS edge from the vacuum’s RUN capability to MOTION_DETECTED is the dependency absent from all rule texts yet sufficient to produce an unintended door unlock. Encoding this edge makes pre-deployment detection possible; tracing it back produces a grounded explanation the user can act on.

Direct contradictions (logical conflicts). Two rules issue mutually exclusive commands to the same actuator under overlapping conditions. An air-conditioning cooling rule and an energy-saving shutdown rule that both target the same AC unit are a canonical example. These conflicts are detectable by syntactic rule comparison alone: no model of physical propagation is required because both rules reference the same device state node in their action clauses.

Environment-mediated conflicts (semantic conflicts). One rule alters a physical variable—temperature, illuminance, air-flow, or smoke concentration—that triggers a second rule, forming an implicit causal chain not encoded in either rule’s text. A heater rule that raises room temperature above the threshold that opens a motorized window, which then cools the room and reactivates the heater, creates an energy-wasting feedback loop that is invisible to syntactic analysis. Neither rule is incorrect; the conflict emerges through the shared environmental variable that mediates between them.

Topology-mediated conflicts (physical conflicts). One rule’s action changes a device state that serves as a precondition for a different rule through a multi-hop chain of physical dependencies spanning heterogeneous entity types. The vacuum-door scenario is precisely this class, as is the camera-plug scenario: an energy-saving rule disables a smart plug, which cuts power to a security camera, which defeats a surveillance rule operating on a separate platform. No direct syntactic dependency connects the energy-saving rule to the surveillance rule; the connection exists only through the physical wiring between the plug and the camera.

These three mechanisms define the conflict classes that

SHACR is designed to detect, explain, and repair, formalized in §IV.

D. Why Existing Approaches Fall Short

Each existing approach addresses part of the problem. Formal rule-centric analyzers [9], [11] handle direct contradictions precisely and deterministically, but are structurally limited to syntactic rule-level analysis. They have no model of how a vacuum’s movement can trigger a motion sensor, or how a heater’s action affects a room temperature variable that feeds another rule’s trigger.

Semantic frameworks [6], [10], [15] extend analysis to model physical propagation, but rely on predefined device behavior models and manually specified environmental influence functions. They do not generalize to new device types or cross-platform rule engines without significant manual effort, and they provide limited support for user-facing explanation or automated repair. Recent LLM-based approaches [16]–[18] improve natural-language interaction but remain unreliable when critical topology and dependency information is absent from the prompt: without an explicit model of how devices connect through power, sensor signals, and shared environments, a language model cannot reliably infer that a vacuum’s cleaning route could unlock the front door.

The limitation shared by all existing approaches is that they reason over rule text. Even language models that can interpret natural language cannot reliably infer physical device dependencies from rule descriptions alone, because those dependencies exist in the deployment environment—not in the rule logic. A structured representation is needed that makes three properties explicit: (1) *where* devices are deployed in the home, (2) *what* physical effects their capabilities have on states and environmental variables, and (3) *how* automation rules reference those entities through their TCA structure. With such a representation, detecting a conflict becomes a graph traversal problem rather than an inference problem, and explaining it becomes a matter of returning the traversal path to the user.

IV. CONFLICT TAXONOMY AND FORMAL DEFINITIONS

A. Notation

The following notation is used uniformly throughout the formal definitions in this section. The smart home is represented as a directed, typed knowledge graph $G = (V, E, \tau_V, \tau_E, A_V, A_E)$, where V is the finite set of nodes and $E \subseteq V \times V$ is the set of directed edges. The function $\tau_V : V \rightarrow \mathcal{T}_V$ assigns each node a type drawn from the node type vocabulary $\mathcal{T}_V$ (e.g., Rule, Capability, State, EVar); analogously, $\tau_E : E \rightarrow \mathcal{T}_E$ assigns each edge a type drawn from the edge type vocabulary $\mathcal{T}_E$ (e.g., AFFECTS, TARGETS, HAS_ACTION). The attribute functions $A_V : V \rightarrow \mathcal{A}$ and $A_E : E \rightarrow \mathcal{A}$ map nodes and edges to their associated attribute dictionaries; attribute access is written $A_V(v).attr$ for a specific attribute field.

In the definitions that follow, r_i and r_j denote distinct Rule nodes; a_i and a_j denote their respective Action sub-nodes; s denotes a State node that serves as a shared

actuation target; e denotes an EVar node representing a room-scoped environmental variable; c denotes a Capability node whose actuation propagates effects downstream; and t_j denotes the Trigger or Condition sub-node of r_j that observes the resulting state or variable. The safety predicate $\Phi(G)$ is a system-specific logical formula over the graph state (e.g., “the front door lock is engaged whenever no authorized presence is detected”); a physical conflict is defined as a composition of effects along a multi-hop path that causes $\Phi(G)$ to be violated.

B. Knowledge Graph Schema

The schema is organized into three semantic layers, summarized in Table II. The layered design reflects the natural structure of the smart home: a physical topology layer describes where things are and what they are; a device capability layer describes what those things can do and what they observe; and an automation layer describes how rules reference and act upon all of the above. Each layer exists not only for representational completeness but because the conflict classes defined below require reasoning that cuts across all three layers simultaneously.

Topological layer. A Building node is the root entity containing one or more Apartment nodes. Each apartment is subdivided into Room nodes representing the spatial units of the deployment (living room, bedroom, kitchen, entrance, etc.). Rooms serve as retrieval anchors for apartment-level subgraph extraction and are the primary unit of analysis for conflict localization: a conflict reported in the context of a specific room can be communicated to the user in terms they recognize from their daily experience of the space.

Device layer. A Device node represents a physical IoT sensor or actuator installed in a room. Each device exposes one or more Capability nodes that encode the abstract operations it can perform (e.g., power, locking, heating, recording, cooling, dimming). Capabilities are the primary targets of automation actions: when a rule fires, it actuates a capability, not a device directly. This distinction matters because multiple rules may target the same capability through different actuation paths. Each device also exposes one or more State nodes encoding observable properties such as lock_status, cam_power, temperature_reading, and motion_detected. States are the primary targets of triggers and conditions, connecting the automation layer back to the physical world.

An EVar (Environmental Variable) node captures room-scoped physical quantities (temperature, humidity, illuminance, airflow, smoke concentration) that are not directly tied to a single device but are influenced by device actions and may trigger safety-relevant rule responses. EVars are the mediating layer through which environment-mediated semantic conflicts propagate.

Automation layer. A Context node encodes a derived home predicate (Night, Away, NoMotion10m) inferred from device states or environmental variables. Contexts allow multiple rules to share a high-level mode description without duplicating the underlying state logic, and they introduce

TABLE II
NODE TYPES IN THE SHACR KNOWLEDGE GRAPH.

Node Type	Semantic Role
Building	Root entity containing one or more apartments.
Apartment	Residential unit; boundary for automation scope and deployment.
Room	Spatial subdivision; localizes devices, EVars, and contextual conditions.
Device	Physical IoT sensor or actuator installed in a room.
Capability	Abstract operation a device can perform (e.g., power, lock, heat, record).
State	Observable device attribute that may change over time (e.g., lock_status, cam_power, motion_detected).
EVar	Room-scoped ambient quantity (temperature, humidity, illuminance, airflow, smoke concentration).
Context	Derived predicate computed from states or EVars (e.g., Night, Away, EnergySavingOn).
Rule	Complete TCA automation rule encoding when and how actions execute.
Trigger	Event or state transition that initiates rule evaluation.
Condition	Logical predicate that must hold for the rule to execute.
Action	Effect executed when the rule fires, actuating a capability or influencing a state.

another class of indirect dependency: two rules may interact through a shared context even if they reference entirely different physical devices. A Rule node encodes a complete TCA automation, decomposed into three structural sub-nodes: Trigger, Condition, and Action. Exposing these sub-nodes explicitly—rather than embedding TCA structure in a monolithic rule node—allows conflict-detection traversal to reason about partial overlaps, such as two rules that share a trigger but diverge in conditions or actions, or share a target but assign it different values.

The graph uses 12 directed, typed edges summarized in Table III. The most analytically significant is AFFECTS: (Capability) $\rightarrow$ (State | EVar). This edge encodes physical cause-effect relationships among device capabilities and environmental or state variables—relationships that are invisible to any text-based analysis. For example, the edge power $\xrightarrow{\text{AFFECTS}}$ cam_power encodes the physical fact that switching a smart plug’s power capability changes the camera’s operational state; this dependency cannot be inferred from either rule’s text. Similarly, run $\xrightarrow{\text{AFFECTS}}$ motion_detected encodes that the vacuum’s movement produces sensor activations in rooms it traverses. The AFFECTS edges collectively form the physical interaction substrate over which all three conflict classes are detected: a logical conflict requires no AFFECTS traversal; a semantic conflict traverses exactly one AFFECTS edge to an EVar; a physical conflict traverses a heterogeneous multi-hop path that may include several AFFECTS edges and cross multiple node types.

C. Formal Conflict Definitions

The following definitions formalize the three conflict classes over the directed knowledge graph $G = (V, E, \tau_V, \tau_E, A_V, A_E)$ introduced above. Figure 2 illustrates their structural distinction.

TABLE III
DIRECTED EDGE TYPES IN THE SHACR KNOWLEDGE GRAPH.

Edge	Direction	Interpretation
HAS_APARTMENT	(Building) → (Apartment)	Associates apartments with a building.
HAS_ROOM	(Apartment) → (Room)	Defines rooms within an apartment.
HAS_DEVICE	(Room) → (Device)	Places devices in their physical rooms.
HAS_CAP	(Device) → (Capability)	Specifies the operational capabilities of a device.
HAS_STATE	(Device) → (State)	Captures observable device attributes.
HAS_EVAR	(Room) → (EVar)	Associates ambient environmental variables with their rooms.
AFFECTS	(Capability) → (State EVar)	Encodes the physical or logical effect of actuating a capability on states or environmental variables.
HAS_TRIGGER	(Rule) → (Trigger)	Links a rule to its triggering event or state transition.
HAS_CONDITION	(Rule) → (Condition)	Links a rule to its logical pre-conditions.
HAS_ACTION	(Rule) → (Action)	Links a rule to the action it executes when fired.
TARGETS	(Trigger Cond. Action) → (State EVar Cap. Context)	Identifies the entity observed, checked, or actuated by a TCA component.
DERIVED_FROM	(Context) → (State EVar)	Records that a contextual predicate is inferred from underlying states or EVars.

Definition 1 (Logical Conflict). A logical conflict exists when two automation rules assign mutually exclusive values to the same device state under overlapping execution contexts:

$$\begin{aligned}
 &\exists r_i, r_j, s, a_i, a_j \in V : \\
 &\tau_V(r_i) = \tau_V(r_j) = \text{Rule}, \quad \tau_V(s) = \text{State}, \\
 &\tau_V(a_i) = \tau_V(a_j) = \text{Action}, \\
 &(r_i \xrightarrow{\text{HAS_ACTION}} a_i) \wedge (r_j \xrightarrow{\text{HAS_ACTION}} a_j), \\
 &(a_i \xrightarrow{\text{TARGETS}} s) \wedge (a_j \xrightarrow{\text{TARGETS}} s), \\
 &A_V(a_i).value \neq A_V(a_j).value.
 \end{aligned} \tag{1}$$

Rules r_i and r_j each own an Action sub-node, a_i and a_j respectively, both of which target the same State node s via TARGETS; the conflict condition is that the attribute values assigned by a_i and a_j to s are mutually exclusive. Logical conflicts are direct rule-level contradictions detectable through static constraint analysis without modeling any environmental propagation, because the shared target s is referenced explicitly in both rules' action clauses.

Definition 2 (Semantic Conflict). A semantic conflict exists when the execution of one rule indirectly activates another through shared environmental variables, forming an implicit causal chain absent from the automation logic text:

$$\begin{aligned}
 &\exists r_i, r_j, e, c, a_i, t_j \in V : \\
 &\tau_V(r_i) = \tau_V(r_j) = \text{Rule}, \quad \tau_V(e) = \text{EVar}, \\
 &\tau_V(c) = \text{Capability}, \quad \tau_V(a_i) = \text{Action}, \\
 &\tau_V(t_j) \in \{\text{Trigger}, \text{Condition}\}, \\
 &(r_i \xrightarrow{\text{HAS_ACTION}} a_i) \wedge (a_i \xrightarrow{\text{TARGETS}} c) \wedge (c \xrightarrow{\text{AFFECTS}} e), \\
 &(t_j \xrightarrow{\text{TARGETS}} e) \wedge (r_j \xrightarrow{\text{HAS_TRIGGER} \vee \text{HAS_CONDITION}} t_j).
 \end{aligned} \tag{2}$$

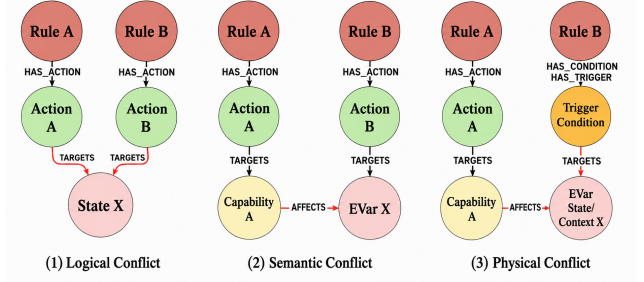


Fig. 2. Structural characterization of the three conflict classes. (1) *Logical*: two actions target the same State node with incompatible values. (2) *Semantic*: an action propagates through a Capability→AFFECTS→EVar chain to activate a second rule's trigger or condition. (3) *Physical*: a multi-hop path through heterogeneous entity types links two rules without any direct dependency edge, composing into an unsafe global state.

where the consequent execution of r_j produces an undesired or unsafe system state. Rule r_i fires an action a_i that targets capability c ; actuating c propagates through the AFFECTS edge to modify environmental variable e ; the trigger or condition sub-node t_j of rule r_j observes e via TARGETS; and r_j is linked to t_j through HAS_TRIGGER or HAS_CONDITION. Semantic conflicts capture environment-mediated couplings between rules that appear syntactically independent, because neither rule's text encodes the environmental variable e that mediates between them.

Definition 3 (Physical Conflict). A physical conflict exists when multi-hop interactions through heterogeneous entity types produce unsafe behavior without any direct rule-to-rule dependency. Formally, a physical conflict exists if there is a path $P = \langle r_i, v_1, \dots, v_k, r_j \rangle$ in G such that:

$$\begin{aligned}
 &\exists r_i, r_j \in V, \exists v_1, \dots, v_k \in V, \exists P = \langle r_i, v_1, \dots, v_k, r_j \rangle : \\
 &(i) \tau_V(r_i) = \tau_V(r_j) = \text{Rule}; \\
 &(ii) \{v_1, \dots, v_k\} \text{ is a heterogeneous mix of } \{\text{Device}, \\
 &\quad \text{Capability}, \text{EVar}, \text{State}, \text{Context}\}; \\
 &(iii) \nexists \text{ direct dependency edge between } r_i \text{ and } r_j; \\
 &(iv) \text{ the composed effects along } P \text{ violate } \Phi(G).
 \end{aligned} \tag{3}$$

Two Rule nodes r_i and r_j are connected by a path P through an intermediate sequence of nodes $v_1, \dots, v_k$ drawn from a heterogeneous mix of entity types; no direct dependency edge exists between r_i and r_j in E , so the interaction is entirely mediated by the intermediate path; and the composed physical effects along P violate the system safety predicate $\Phi(G)$. Note that c does not appear in this definition because physical conflicts generalize beyond single-capability propagation to arbitrary multi-hop chains that may traverse several AFFECTS edges and multiple node types. Physical conflicts arise from latent cyber-physical interactions that are visible only when reasoning over the combined structure of the deployment environment, not from any individual rule's text.

D. Dependability Implications of the Conflict Taxonomy

The three conflict classes carry distinct consequences for system dependability. Logical conflicts are primarily *reliability failures*: contradictory commands leave actuators in undefined or oscillating states, degrading system correctness without necessarily producing immediate physical harm. Semantic conflicts manifest as *safety failures*: an implicit environmental chain produces a hazardous system state—an overheated room, a locked exit during a fire alarm—that the user never authorized and that cannot be diagnosed from rule text alone. Physical conflicts are the most consequential because a single AFFECTS path can simultaneously constitute a safety hazard, a security vulnerability, and a reliability degradation through the same causal chain. The knowledge graph substrate makes all three consequence dimensions inspectable over the same traversal structure, which is why building that structure correctly is the precondition for all subsequent analysis.

V. SYSTEM DESIGN

A. Design Rationale

The core insight driving SHACR’s design is that smart home conflicts are routinely undetected not because they are rare, but because existing representations are insufficient to expose them. A directed knowledge graph restores that visibility by making device relationships, environmental effects, and rule interactions explicit, structured, and inspectable. Because dependencies are encoded as typed graph edges, detecting conflicts becomes a matter of traversing the structure rather than inferring it from natural language, enabling analysis that rule-text processing cannot achieve regardless of the reasoning method applied on top.

Within this framework, automation rules are treated not as isolated text but as interconnected components embedded in the graph through their TCA structure. This enables the system to reason about both direct rule interactions and indirect multi-hop effects propagated through shared states, environmental variables, and capability-induced changes. Building on this representation, SHACR adopts a GraphRAG pipeline in which apartment-level subgraphs are retrieved as the focused reasoning context for a given query. These subgraphs provide a complete structural overview of the apartment’s rules, devices, rooms, and inter-entity relationships, which the LLM uses to generate precise conflict detection, classification, and repair suggestions.

A central design decision is the choice to anchor conflict reasoning in an LLM rather than a purely formal or rule-based verification engine. Non-LLM approaches—SMT-based solvers, finite state machine verifiers, formal conflict checkers—offer deterministic, auditable outputs with no inference variability. They are computationally efficient and produce results that can be formally proven correct within their specification. For strictly logical conflicts, where two rules produce directly contradictory actions on a shared device state, such approaches are well-suited. However, the conflict landscape in smart home automation extends well beyond logical contradictions. Physical conflicts arise from indirect environmental interactions across heterogeneous device types,

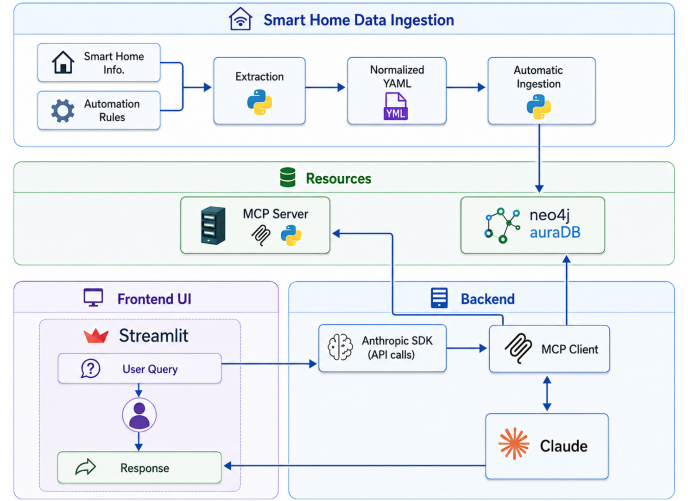


Fig. 3. SHACR system architecture. Automation rules are ingested from YAML specifications and normalized into a Neo4j knowledge graph. The backend reasoning engine accesses the graph through two MCP server tiers. All knowledge graph write operations are gated behind explicit user approval in the Streamlit dashboard.

and semantic conflicts emerge from implicit user-intent mismatches that have no formal encoding in rule syntax. These conflict types cannot be captured by pattern matching or formal verification alone, as they require reasoning over natural-language rule descriptions, contextual knowledge about device capabilities, and causal chains that span multiple graph hops.

The GraphRAG-LLM design addresses this by grounding LLM inference in structured graph relationships, reducing hallucination risk while preserving the semantic reasoning flexibility that formal methods lack [20], [21], [23]. Relevant evidence in this domain is often distributed across entities connected through semantic relations rather than surface-level lexical similarity; graph-based retrieval enables evidence aggregation along explicit relational paths, supporting multi-hop reasoning and improving coherence in structured tasks [22]. In practice, smart home knowledge is initially distributed across heterogeneous or semi-structured sources, and rules are authored in natural language by non-expert users without formal specification. A graph-augmented LLM pipeline therefore facilitates the transformation of this unstructured knowledge into structured representations that support relational reasoning while retaining the flexibility to handle incomplete or ambiguous rule descriptions [24]. The overall design is additionally informed by empirical findings on the challenges non-expert smart home users face in forming accurate mental models of rule interactions and recovering from automation failures [25]–[27], [29]–[31], [33], [35].

B. Architecture Overview

SHACR comprises three functional layers illustrated in Figure 3: (1) a *data ingestion layer* that normalizes heterogeneous rule specifications into the knowledge graph; (2) a *backend reasoning layer* comprising the Neo4j graph store, two-tier MCP server, and LLM orchestrator; and (3) a *user interface layer* that enforces human approval over all graph modifications.

1) *Data Ingestion*: Smart home configurations and automation rules are provided as raw YAML specifications authored on heterogeneous platforms such as Home Assistant and SmartThings. The ingestion pipeline begins with an extraction and normalization module that parses these specifications and maps them into a common TCA representation, resolving platform-specific syntax variations so that the subsequent analysis is independent of the originating platform. The normalization step also resolves device naming conventions, matches capability descriptors to the standard node taxonomy defined in §IV-B, and identifies implicit dependencies—such as a device’s downstream power dependency on a smart plug—that must be made explicit as AFFECTS edges.

Once normalized, the structured representations are ingested into the Neo4j knowledge graph by an automated ingestion script. The script creates typed nodes for all entities in the home (Building, Apartment, Room, Device, Capability, State, EVar, Context, Rule, and the TCA sub-nodes) and instantiates the directed edge relationships defined in Table III. The AFFECTS edges between capability nodes and their downstream state or environmental variable nodes are populated during ingestion using a combination of device documentation lookup and rule-text analysis, ensuring that the physical cause-effect substrate is complete before any conflict reasoning begins.

2) *Backend Reasoning Engine*: The backend comprises three interacting components: the Neo4j AuraDB knowledge graph store, two MCP server tiers, and the LLM-based reasoning module.

Knowledge graph store. The Neo4j AuraDB instance maintains the structured smart home representation constructed during ingestion. It is accessed through an encrypted connection using the `neo4j+s://` protocol, with credentials stored as environment variables. The database serves both as the persistent knowledge repository and as the retrieval source for the GraphRAG pipeline. Any repair approved by the user is written back to this store through the sole write tool in the MCP layer, ensuring that the graph reflects the current configuration state at all times.

MCP server layer. The Model Context Protocol (MCP) is an open protocol developed by Anthropic that provides a standardized interface through which the LLM can invoke external tools and data sources at inference time [41]. Two MCP servers operate concurrently in this system. The first, `mcp-neo4j-cypher`, is a general-purpose open-source server that exposes three primitive database operations (`read_cypher`, `write_cypher`, and `get_schema`), launched automatically via the `uvx` package runner using the `stdio` transport protocol.

The second is a custom domain-specific MCP server that exposes five purpose-built tools (Figure 4) to the LLM over a standardized protocol interface. These tools collectively define the action space available to the reasoning agent, enforce the system’s safety constraints, and mediate all interactions with the knowledge graph. Four tools are strictly read-only; only one tool performs write operations, and it is gated behind explicit user approval. This design prevents the LLM from autonomously modifying the knowledge graph and ensures

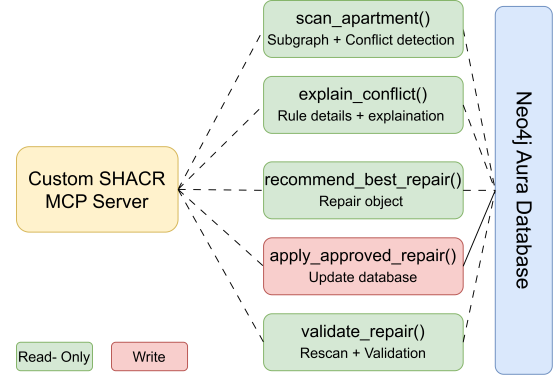


Fig. 4. The five custom MCP tools forming the operational backbone of SHACR. Read-only tools are invoked autonomously by the reasoning engine; the single write tool requires explicit user approval before execution.

that all graph state changes are traceable to a specific user decision.

a) *Tool 1: scan_apartment*: This tool is the entry point for every SHACR session. It executes two lightweight Cypher queries against the knowledge graph and returns a merged raw subgraph covering all rules associated with the target apartment. The first query retrieves each rule’s actions, action targets (State, Capability, or EVar nodes), and the devices those actions apply to. The second query retrieves each rule’s triggers, conditions, and the full capability-to-EVar chain accessible from the rule’s associated devices. The two result sets are merged in Python by `ruleId` into a unified `rawSubgraph` before being passed to the LLM for reasoning. The tool also returns an `apartmentDashboard` payload containing room, device, and rule counts for display in the Streamlit interface.

b) *Tool 2: explain_conflict*: This tool accepts a conflict identifier of the form `ruleAId_ruleBId` and constructs a structured explanation object that exposes the conflicting trigger events, contradicting action sequences, and plain-language descriptions of both rules. The LLM uses this object to produce a human-readable explanation suitable for presentation to the user. This tool is particularly important for semantic and physical conflicts, where the causal chain spans multiple graph hops and cannot be communicated clearly without tracing the specific nodes and edges involved. Rather than asking the user to interpret raw graph data, the system surfaces a causally grounded narrative of why the conflict exists, directly traceable to the home’s physical configuration.

c) *Tool 3: recommend_best_repair*: This is the system’s planning tool. It retrieves the full TCA details of both conflicting rules alongside the current device states of the affected devices and proposes the best-fit repair from within a constrained set of allowed operations. To regulate LLM behavior and minimize the risk of unsafe or semantically inappropriate repair suggestions, the model is restricted to a bounded action space while a corresponding set of forbidden operations is explicitly enforced (Table IV). This ensures that all repair proposals remain within a safe, auditable operational boundary.

TABLE IV
PERMITTED AND FORBIDDEN REPAIR OPERATIONS IN SHACR.

Allowed Operations	Forbidden Operations
Modify trigger conditions (modify_trigger)	Delete rules, devices, rooms, or apartments
Add mutual-exclusion conditions (add_condition)	Invent new graph entities
Remove redundant or conflicting conditions (remove_condition)	Remove physical topology edges (HAS_DEVICE, HAS_ROOM, HAS_STATE)
Refine existing conditions for specificity (refine_condition)	Modify graph structure beyond rule-level constraints
Modify conflicting actions to resolve direct contradictions (modify_action)	Introduce semantically arbitrary repairs
Adjust rule priority to control execution order (add_priority)	Execute write operations without user approval

d) *Tool 4: apply_approved_repair*: This is the sole write tool in the system and the only point at which the knowledge graph is modified. It accepts the structured repair object produced by `recommend_best_repair` and executes the corresponding Cypher write against Neo4j. The tool validates that the repair object contains both a target rule and a specified operation before executing, and returns a structured success or failure response. Currently implemented operations include `add_condition`, which creates a new `Condition` node with a mutual-exclusion constraint. This tool cannot be invoked autonomously by the LLM: explicit user approval is a hard architectural requirement, not a soft preference, ensuring that all knowledge graph modifications are supervised.

e) *Tool 5: validate_repair*: This tool re-executes a targeted conflict-detection query scoped to the two rules involved in the original conflict, determines whether the logical contradiction between their actions on the shared device still exists, and returns a boolean `isResolved` flag alongside the current condition state of both rules. If the conflict persists, the tool reports the remaining conflict count so that the reasoning engine can generate an alternative repair. Together with `apply_approved_repair`, this tool implements the Scan-Approve-Execute-Verify cycle that forms the operational core of SHACR's conflict resolution pipeline.

Reasoning module. The LLM operates in a semi-autonomous capacity as the system's reasoning orchestrator. It is connected to both MCP servers through the Anthropic SDK and discovers all available tools at inference time. Upon receiving a user query, it selects and invokes the appropriate MCP tool, processes the returned structured subgraph data, performs conflict analysis, and generates a structured response comprising detected conflicts and repair recommendations. This positions the LLM as an active orchestrator rather than a passive text generator, enabling multi-step reasoning pipelines that interleave graph retrieval, conflict analysis, and repair generation within a single inference cycle. To ensure bounded and auditable behavior, the LLM's action space is constrained by the MCP tool definitions and the system prompt, and no write operations may be executed without explicit user

approval.

3) *User Interface*: The Streamlit frontend provides an interactive dashboard through which users initiate and monitor the conflict resolution workflow. A user submits an apartment identifier to trigger a full apartment scan, which invokes the reasoning pipeline with the active MCP configuration. The returned response is rendered as structured summary metrics, a list of detected conflicts with graph-grounded causal explanations, and corresponding repair recommendations. A mandatory approval step is enforced before any repair action is applied to the knowledge graph: the system surfaces the specific repair proposal (what change will be made, which rules are affected, and what the expected outcome is) and requires an explicit confirmation before `apply_approved_repair` is invoked. This human-in-the-loop mechanism ensures that the system operates as a decision-support tool, with final authority over the home configuration remaining with the user.

C. Failure Conditions and Scope

SHACR targets pre-deployment conflict detection in a multi-platform smart building environment and does not assume adversarial occupants. It is, however, designed to be robust to two categories of imperfect input that arise naturally in real deployments.

Imperfect rule descriptions. Automation rules may contain misleading, incomplete, ambiguous, or semantically inconsistent descriptions. These imperfections may arise from user misconfiguration, cross-platform semantic mismatches, automatically generated rule templates, or inadequately described device capabilities. When rule descriptions do not accurately reflect the intended semantics of the deployment, the constructed knowledge graph may become inconsistent or incomplete. This can result in incorrect reasoning about device interactions, failure to detect unsafe physical conflicts, and unsafe repair suggestions. The framework addresses this risk through the formal graph schema: the normalization step during ingestion maps rule descriptions against the typed node and edge taxonomy, and mismatches are surfaced as graph construction anomalies rather than silently propagated into the reasoning pipeline.

LLM-specific reasoning limitations. Because the framework uses an LLM for rule interpretation, conflict explanation, and repair suggestion, it inherits the inference variability of generative models. The LLM may produce outputs that appear plausible yet contradict the underlying knowledge graph, or exhibit overconfidence on borderline classification cases. The framework mitigates these risks through four architectural measures: (1) all LLM reasoning is grounded in Cypher-retrieved subgraphs rather than free-form text; (2) deterministic graph traversal provides an independent post-classification verification step; (3) repairs are restricted to a bounded, semantically validated operation set; and (4) all write operations require explicit user approval before execution.

VI. PROTOTYPE IMPLEMENTATION

A. System Configuration

The SHACR stack integrates four principal components: a Neo4j AuraDB graph database, an MCP server layer, an

TABLE V
SYSTEM COMPONENTS, VERSIONS, AND ROLES.

Component	Version	Role
Neo4j AuraDB	5.26 LTS (cloud)	Knowledge graph store
mcp-neo4j-cypher	0.6.0	General-purpose MCP bridge to Neo4j
Claude Sonnet 4	claude-sonnet-4-20250514	Reasoning orchestrator
Custom MCP Server	Python 3.10+	Domain-specific tool layer
Streamlit	1.45.1	User dashboard interface
Anthropic SDK	0.107.1	Claude client; drives MCP server

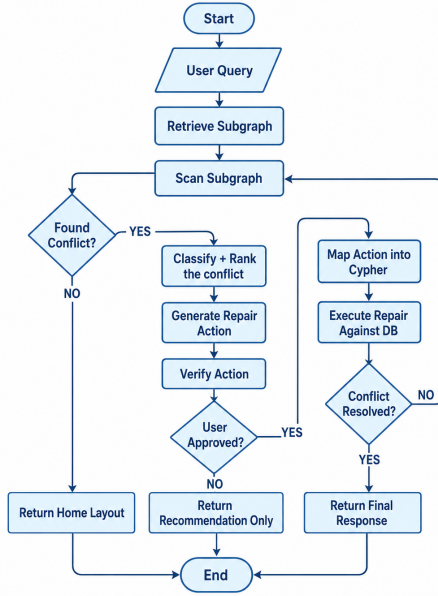


Fig. 5. SHACR runtime workflow. Subgraph retrieval feeds conflict detection and classification. Each detected conflict yields a causal explanation and a constrained repair recommendation presented to the user. If approved, the repair is applied and re-verified; if the conflict persists, an alternative repair is generated. The workflow terminates when all conflicts are resolved or no valid repair can be produced.

LLM reasoning orchestrator, and a Streamlit web interface. Table V lists the specific technologies, versions, and roles of each component.

Both MCP servers are registered in the host configuration file and instantiated as subprocesses at application startup. The reasoning module discovers all available tools at inference time and selects the appropriate tool based on the active user query and system prompt instructions. Connection credentials for Neo4j are stored as environment variables and are never embedded in source code, in accordance with standard security practices.

B. Runtime Workflow

Figure 5 illustrates the end-to-end operational workflow. The reasoning engine interprets the user's apartment-scan query, invokes `scan_apartment` to retrieve the relevant subgraph, and analyzes all rule pairs within the subgraph for conflicts across the three defined classes. For each de-

tected conflict, the engine invokes `explain_conflict` to construct a causally grounded explanation, then invokes `recommend_best_repair` to propose a repair from within the constrained action space. The explanation and repair recommendation are rendered in the Streamlit dashboard, where the user reviews the proposed change and decides whether to approve it.

If approved, the repair is applied through `apply_approved_repair` and immediately re-verified through `validate_repair`. If the validation confirms resolution, the conflict is marked as resolved and the workflow proceeds to the next detected conflict. If the validation finds that the conflict persists despite the applied repair, the system reports the unresolved state and generates an alternative repair recommendation. The workflow terminates when either all detected conflicts have been resolved or no valid repair can be generated within the allowed operation set. This bounded termination condition prevents the system from entering unbounded repair loops and ensures that the user is informed of any conflicts that require manual intervention.

Algorithm 1 formalizes the internal logic of the detection and repair pipeline.

Arriving at the final retrieval design required three successive engineering iterations. Each phase exposed a fundamental limitation that motivated the next, and the progression from a single-query design to a split retrieval architecture reflects the practical constraints of operating on a cloud-hosted graph database at non-trivial scale.

Phase 1: Single combined retrieval query. The initial design attempted to capture the complete apartment subgraph using a single Cypher query that traversed rules, actions, triggers, conditions, devices, capabilities, and environmental variables in a single execution pipeline. This design failed in practice. The large number of chained traversal operations caused combinatorial path expansion and consistently triggered query timeouts on Neo4j AuraDB, particularly for apartments with larger rule sets and richer device relationship graphs. As a result, this approach was unable to return reliable results and was abandoned.

Phase 2: Hardcoded conflict detection in Cypher (V1). The second iteration shifted strategy from raw subgraph retrieval toward embedding conflict detection logic directly in Cypher. Rather than retrieving the full graph context for the LLM to reason over, this approach compared rule actions pairwise on shared devices within the query and returned pre-identified conflicting rule pairs. This phase resolved the timeout issue and produced stable execution performance. However, it introduced a more fundamental architectural limitation: conflict detection was now performed by the Cypher query layer rather than by the LLM. The detection logic was confined to direct action contradictions on the same device state, which corresponds only to logical conflicts. Semantic conflicts requiring causal reasoning through `Capability`→`AFFECTS`→`EVar` chains and physical conflicts involving indirect multi-hop interactions across heterogeneous entity types were structurally undetectable under this design.

Phase 3: Split two-query retrieval (V2, final). The

Algorithm 1 SHACR Conflict Detection and Safe Repair**Require:** Apartment identifier apt_id **Ensure:** Classified conflict reports C , causal explanations, approved repairs, and validation outcomes

```

1: Connect to Neo4j knowledge graph via two-tier MCP
   server
2:  $subgraph \leftarrow \text{SCANAPARTMENT}(apt\_id)$   $\triangleright$  Retrieve
   merged rules, devices, states, EVars, capabilities
3:  $C \leftarrow \text{LLMDETECT}(subgraph)$   $\triangleright$  LLM identifies all
   conflicting rule pairs
4: for each  $cf \in C$  do
5:    $cf.type \leftarrow \text{LLMCLASSIFY}(cf, subgraph)$   $\triangleright$  Classify
   as LOGICAL, SEMANTIC, or PHYSICAL
6: end for
7: for each  $cf \in C$  do
8:    $expl \leftarrow \text{EXPLAINCONFLICT}(cf.id)$   $\triangleright$  Trace causal
   path through graph entities
9:    $repair \leftarrow \text{RECOMMENDBESTREPAIR}(cf.id)$   $\triangleright$  Select
   from constrained action space
10:   Present  $expl$  and  $repair$  to user
11:   if user approves  $repair$  then
12:      $\text{APPLYAPPROVEDREPAIR}(repair)$   $\triangleright$  Updates live
     knowledge graph
13:      $resolved \leftarrow \text{VALIDATEREPAIR}(cf.id)$ 
14:     if  $resolved$  then
15:       Mark  $cf$  as resolved in  $C$ 
16:     else
17:        $repair \leftarrow \text{RECOMMENDBESTREPAIR}(cf.id)$   $\triangleright$ 
       Generate alternative repair
18:       Present alternative to user; repeat approval step
19:     end if
20:   end if
21: end for
22: return  $C$ , explanations, repair outcomes, validation results

```

final design fully decouples retrieval from conflict detection. Rather than embedding detection logic into Cypher, the system retrieves the raw graph structure and delegates all conflict reasoning to the LLM. This decoupling is achieved by splitting the single combined query into two lightweight parallel queries. The first query focuses on rule actions, action targets, and associated devices. The second retrieves triggers, conditions, and the capability-to-EVar relationships reachable through the devices involved in each rule. The two result sets are merged in Python using `ruleId` as the join key, producing a unified `rawSubgraph` structure that is passed to the LLM for reasoning. Each individual query is sufficiently narrow to avoid AuraDB timeout thresholds; together, they expose the complete contextual neighborhood required for reasoning over all three conflict classes.

Table VI summarizes the progression across the three engineering phases.

VII. EVALUATION

We evaluate SHACR to answer four questions: (i) how accurately it detects and classifies the three conflict classes;

TABLE VI
RETRIEVAL DESIGN EVOLUTION ACROSS ENGINEERING PHASES.

Phase	Approach	Outcome
P1	Single combined sub-graph query	Repeated AuraDB timeouts due to combinatorial path expansion; unreliable execution.
P2 (V1)	Hardcoded pairwise conflict detection in Cypher	Execution stable; detection limited to logical conflicts; semantic and physical classes structurally undetectable.
P3 (V2)	Split two-query retrieval with Python merge; full LLM reasoning	Reliable within AuraDB timeout thresholds; complete graph context exposed for all three conflict classes.

(ii) whether lightweight few-shot calibration improves detection, and whether that improvement is specific to the graph-grounded system; (iii) how much of the performance is attributable to the knowledge graph rather than to the underlying LLM; and (iv) how SHACR compares against established prior-work detectors.

A. Testbed and Setup

To our knowledge, no publicly available benchmark exists for large-scale smart building automation conflict detection. We therefore constructed a controlled testbed of 70 apartments in a simulated smart building that spanned family, bachelor, and elderly household types, capturing diverse device inventories, lifestyle patterns, and automation complexity. Automation rules were drawn from publicly available repositories and official platform documentation for SmartThings and Home Assistant, providing realistic user-authored patterns across heterogeneous IoT ecosystems [42]–[44]. The rules collected were normalized into a unified YAML schema, mapped to the building layout, and ingested following the schema in §IV-B; where necessary, the rules were adapted to ensure structural consistency with the knowledge graph model while preserving their original semantics.¹

Table VII summarizes the testbed statistics and the distribution of ground-truth conflict, and Table VIII lists the device inventory. The distribution is deliberately sparse and representative of real deployments: 39 of 70 apartments contain at least one conflicted rule and 31 are conflict-free, and among the conflicted apartments 27 contain only one or two conflicts, while 12 contain three or more, reflecting the predominance of isolated rather than systemic conflicts at household scale. At the rule level, 88 of 203 rules (43%) participate in a conflict, partitioned into 38 logical, 31 semantic, and 19 physical conflicts. Logical conflicts are the most common and physical conflicts—those mediated by multi-hop device topology—the rarest, consistent with deep dependency chains being less frequent but more consequential. This sparsity makes precision as important as recall because over-flagging imposes a real burden on users within a predominantly clean rule set.

We compare **full SHACR** (the complete framework comprising the knowledge graph, GraphRAG retrieval, LLM-

¹The complete testbed—the 70 apartment YAML specifications, the knowledge-graph ingestion scripts, and the ground-truth conflict labels—is publicly available at <https://github.com/Aljoby/SHACR>.

TABLE VII
SMART BUILDING TESTBED STATISTICS AND GROUND-TRUTH CONFLICT DISTRIBUTION ($n = 203$ RULES).

Metric	Value
Total apartments	70
Conflicted apartments	39
Clean apartments	31
Apartments with 1–2 conflicts	27
Apartments with ≥ 3 conflicts	12
Total rules	203
Conflicted rules	88
Clean rules	115
Conflict scenarios	63
<i>Conflict distribution by type</i>	
Logical conflicts	38
Semantic conflicts	31
Physical conflicts	19

TABLE VIII
IoT DEVICE INVENTORY ACROSS THE TESTBED.

Device Category	Count
Motion sensors	28
Smart lighting	24
Smart cameras (indoor/entry/security)	14
Robot vacuum cleaners	12
Door/entry sensors	12
Heaters	10
Smart door locks	10
Temperature sensors	9
Safety and alarm systems	9
Nursery and elderly care devices	8
HVAC and air conditioning units	8
Smart windows and actuators	7
Smart lamps	7
Smoke sensors	7
Air quality sensors	6
Exhaust and ventilation fans	6
Leak detection sensors	5
Water control valves	5
Humidity sensors	3

based reasoning, and executable repair) against two complementary families of baselines. The first is a graph-free *ablation* we call the **YAML-only baseline**, in which the identical LLM reasoning step runs on raw YAML rule text *without* the knowledge graph, evaluated with both Claude Sonnet 4 (claude-sonnet-4-20250514) and Google Gemini (gemini-2.5-flash) so that the contribution of graph grounding is separated from raw model capability. Full decoding parameters appear in the supplementary material. Because full SHACR and the YAML-only Claude baseline use the identical underlying model, their difference isolates the effect of the knowledge graph. The second family comprises two established rule-centric detectors from the prior literature, **IoTC**² [11] and **VISCR** [12], which bound what is achievable from rule text alone without a structured physical model (§VII-E). Detection is scored at the rule level using precision,

TABLE IX
RULE-LEVEL CONFUSION MATRIX, FULL SHACR ($n = 203$).

	Pred. Conflict	Pred. Clean
Actual Conflict	70 (TP)	18 (FN)
Actual Clean	20 (FP)	95 (TN)

TABLE X
OVERALL CONFLICT DETECTION METRICS, FULL SHACR (ZERO-SHOT).

Metric	Value
Precision	0.78
Recall	0.80
F1-score	0.79
Accuracy	0.81
Micro-F1 (multi-class)	0.78
Specificity	0.83

recall, F1, and accuracy, following prior IoT conflict-detection work [3], [8]–[10].

TABLE XI
PER-CONFLICT-TYPE DETECTION BREAKDOWN, FULL SHACR (CONFLICT-INSTANCE LEVEL).

Type	TP	FP	FN	Prec.	Rec.	F1
Logical	34	18	4	0.65	0.89	0.75
Semantic	16	8	15	0.67	0.52	0.58
Physical	20	6	6	0.77	0.77	0.77
Micro Avg.	70	32	25	0.69	0.74	0.71

B. Conflict Detection Across All Three Classes

We first characterize full SHACR in its zero-shot configuration. Tables IX and X report overall rule-level performance: precision 0.78, recall 0.80, F1 0.79, and accuracy 0.81, with 70 true positives, 18 false negatives, 20 false positives, and 95 true negatives. The low false-negative count matters in safety-critical settings, where a missed conflict is far costlier than a conservative over-flag, and the specificity of 0.83 confirms that the system separates conflicted from clean rules well despite the sparse class balance. The micro-averaged F1 of 0.78 across the three classes indicates that aggregate per-class performance is balanced, even though the classes differ markedly in difficulty.

Table XI breaks down detection performance by conflict class. Unlike the rule-level metrics in Table IX, these counts are aggregated at the conflict-instance level. Because a single rule can participate in multiple distinct conflict pairs, it is counted once in the rule-level matrix but enumerated per pair here; consequently, the total false positive (FP) and false negative (FN) counts differ between the two tables. The asymmetry across classes is structurally meaningful. Logical conflicts are detected most reliably (recall 0.89) because a direct contradiction between two actions on a shared state node is surfaced in a single graph traversal and needs no multi-hop reasoning. Physical conflicts—entirely invisible to

TABLE XII

FEW-SHOT PROMPT CALIBRATION ACROSS THREE CONFIGURATIONS, COMPARING FULL SHACR (CLAUDE SONNET 4 + KNOWLEDGE GRAPH) AGAINST THE YAML-ONLY CLAUDE SONNET 4 AND GOOGLE GEMINI BASELINES ($n = 203$, $TP+FN = 88$, $TN+FP = 115$). SHACR (HIGHLIGHTED) LEADS ON F1 IN EVERY CONFIGURATION AND REACHES 0.95 WITH BALANCED EXAMPLES, WHILE FEW-SHOT CALIBRATION BARELY MOVES THE GRAPH-FREE BASELINES.

Method	Acc.	Prec.	Rec.	F1	TP	TN	FP	FN
<i>Exp. 1: Zero-shot</i>								
SHACR (Claude + KG)	0.81	0.78	0.80	0.79	70	95	20	18
Claude Sonnet 4 (YAML-only)	0.70	0.75	0.48	0.59	42	101	14	46
Google Gemini (YAML-only)	0.72	0.70	0.64	0.67	56	91	24	32
<i>Exp. 2: Few-shot (3 conflict examples)</i>								
SHACR (Claude + KG)	0.86	0.80	0.90	0.84	79	95	20	9
Claude Sonnet 4 (YAML-only)	0.76	0.74	0.70	0.72	62	93	22	26
Google Gemini (YAML-only)	0.72	0.70	0.64	0.67	56	91	24	32
<i>Exp. 3: Few-shot (7 balanced examples)</i>								
SHACR (Claude + KG)	0.96	1.00	0.90	0.95	79	115	0	9
Claude Sonnet 4 (YAML-only)	0.73	0.74	0.60	0.66	53	96	19	35
Google Gemini (YAML-only)	0.75	0.74	0.66	0.70	58	95	20	30

text-based methods—are detected with balanced precision and recall (both 0.77), confirming that the explicit multi-hop AFFECTS paths encoded in the graph make topology-mediated interactions tractable. Semantic conflicts remain the hardest (recall 0.52, F1 0.58): detecting them requires the model to follow chained AFFECTS edges through EVar nodes and judge that the resulting environmental change constitutes an undesired interaction. The semantic false negatives, which account for the majority of all missed detections and concentrate in apartments with longer environmental dependency chains, point to EVar modeling completeness as the main lever for closing this gap. Crucially, even zero-shot SHACR detects all three classes—including the physical class that rule-centric and text-only methods cannot reach at all (§VII-E).

C. Effect of Few-Shot Prompt Calibration

The zero-shot configuration exhibits two systematic error modes: over-detection of structurally similar but benign rule pairs (false positives) and missed multi-hop semantic chains (false negatives). We ask whether lightweight few-shot calibration—annotated examples injected into the system prompt, with no change to the graph or retrieval pipeline—can address both. Table XII reports three configurations for full SHACR alongside the two YAML-only (graph-free) LLM models.

For full SHACR, calibration helps substantially and monotonically. Adding three conflict examples—one per class (Exp. 2)—raises recall from 0.80 to 0.90, dropping false negatives from 18 to 9, by anchoring the model to the AFFECTS-chain patterns that signal genuine multi-hop conflicts; but supplying only positive examples leaves false positives at 20, the expected label-distribution bias of one-sided prompting [37], [38]. Adding four clean examples that target the specific false-positive patterns (Exp. 3) corrects this directly: precision rises to 1.00 as false positives fall to 0 while recall holds at 0.90, lifting F1 from 0.79 to **0.95** and accuracy to

0.96. Graph retrieval supplies the structured evidence; the few-shot examples calibrate how the model reasons over it—complementary mechanisms addressing distinct failure modes. The perfect precision in Exp. 3 reflects the tight alignment between the examples and the controlled testbed’s conflict patterns and is likely optimistic in the wild; the underlying mechanism, however, is general.

The same calibration tells a sharply different story for the graph-free models, and that contrast is itself a result. Few-shot prompting barely moves them: YAML-only Claude improves only from F1 0.59 to a peak of 0.72 and then *regresses* to 0.66 under the balanced examples, while Gemini stays essentially flat (0.67 to 0.70). Notably, the balanced examples that drive SHACR to perfect precision actively *reduce* YAML-only Claude’s recall, from 0.70 to 0.60. The reason is structural. Few-shot examples teach a decision rule, but a rule can be applied only if the features it references are present. Distinguishing a genuine conflict from a benign look-alike requires EVar ranges, AFFECTS semantics, and context-mode exclusivity—information that lives in the graph, not in rule text. SHACR has those features, so a decision criterion such as “a structurally similar pair is clean when its AFFECTS paths do not compose into a hazard” becomes a rule the model can verify. Before flagging a pair, it queries the graph to confirm whether the shared EVar or composing AFFECTS path exists, turning each example into a checkable decision aligned with ground truth, and precision climbs to 1.00. The YAML-only models receive the same decision criterion but cannot verify it; they approximate “shared EVar” from surface similarity in device names and rule text, an unreliable proxy, so the same examples inject noise and the models grow uniformly more cautious, shedding true positives with the false ones. Calibration thus pays off only in the presence of the graph, demonstrating that prompt engineering is not a substitute for the structured substrate but rather a complement to it.

TABLE XIII

CONFLICT DETECTION: SHACR VS. REIMPLEMENTED PRIOR-WORK BASELINES (RULE-LEVEL, $n = 203$). IoTC² AND VISCR OPERATE ON RULE TEXT ALONE, WITHOUT A KNOWLEDGE GRAPH OR LLM, AND ARE REIMPLEMENTED FROM THEIR PUBLISHED DESCRIPTIONS. SHACR (HIGHLIGHTED) DOMINATES BOTH BASELINES ACROSS ALL PROMPTING CONFIGURATIONS, EVEN ZERO-SHOT.

Method	Acc.	Prec.	Rec.	F1	TP	TN	FP	FN
IoT ² [11]	0.68	0.73	0.43	0.54	38	101	14	50
VISCR [12]	0.66	0.70	0.38	0.49	33	101	14	55
SHACR (Exp. 1, zero-shot)	0.81	0.78	0.80	0.79	70	95	20	18
SHACR (Exp. 2, few-shot)	0.86	0.80	0.90	0.84	79	95	20	9
SHACR (Exp. 3, few-shot)	0.96	1.00	0.90	0.95	79	115	0	9

D. Contribution of the Knowledge Graph

The configurations in Table XII also isolate the knowledge graph’s contribution, because full SHACR and the YAML-only Claude baseline share the identical model and differ only in whether the graph is present. In the zero-shot setting, adding the graph lifts F1 from 0.59 to 0.79 and cuts total classification errors (FP + FN) from 60 to 38, a **36.7%** reduction; against YAML-only Gemini (56 errors) the reduction is **32.1%**. Because the model is held fixed in the Claude comparison, this gap is attributable to the graph rather than to model capability.

The error structure explains why. Without the graph, Claude reasons only over rule text. It attains moderate precision (0.75) but poor recall (0.48), missing more than half of all conflicts because the multi-hop dependency paths that connect interacting rules are simply absent from the text it sees. Gemini is more balanced (F1 0.67) but still far short of SHACR, and for the same reason—neither model can follow a causal chain such as *vacuum runs* → *motion sensor fires* → *door unlocks* when no edge in the input encodes it. The graph supplies exactly those edges: the AFFECTS substrate makes physical dependencies traversable, and the hierarchical room/device/apartment structure grounds each conflict in its correct spatial scope rather than treating rules as free-floating text fragments. The knowledge graph, not the choice of LLM, is the decisive architectural factor.

E. Comparison with Prior Work Baselines

The ablation isolates the graph by holding the LLM fixed; we now ask how SHACR compares against established detectors from the prior literature. We select two rule-centric baselines that are knowledge-graph-free and LLM-free and that together bound what is achievable from rule text alone. IoTC² [11] is the canonical formal-method detector of logical contradictions—two rules issuing opposing actions on a shared actuator. VISCR [12] strengthens this with a vendor-independent topology abstraction and a mutual-exclusion filter over context-gated rules, a stronger non-learning baseline. Both are widely cited and target the same pre-deployment task. As neither has a public implementation, we faithfully reimplement the core detection logic of each from its published description and run it on the identical 203-rule testbed and ground truth used throughout §VII; this reimplement is the comparison’s principal fairness caveat and is disclosed explicitly.

Table XIII reports the results. Even zero-shot SHACR (F1 0.79) exceeds the better baseline by 0.25 F1, and balanced few-shot SHACR reaches 0.95. The baselines are bounded on recall by construction: both detect only direct action contradictions, so their ceiling is the fraction of conflicts expressible in that form—38 of 88, or 43%, in our testbed. IoTC²’s recall of 0.43 matches that fraction almost exactly. It recovers essentially all logical conflicts and *none* of the 50 semantic and physical conflicts, which carry no action-level contradiction to find. VISCR’s mutual-exclusion filter additionally over-skip five genuine logical conflicts, lowering its recall to 0.38. The lesson generalizes beyond formal methods. Every rule-text-only approach we evaluate—the formal checkers here (F1 0.49–0.54) and the LLMs reading raw YAML in §VII-D (F1 0.59–0.67)—plateaus well below SHACR. What breaks the ceiling is not a more capable reasoning engine but the traversable graph substrate that exposes the multi-hop and environment-mediated conflicts none of these methods can see.

The baselines’ precision shortfall (0.73 and 0.70, versus 1.00 for SHACR at Exp. 3) is equally structural. Their 14 shared false positives fall into three benign patterns that are indistinguishable from genuine conflicts at the rule-text level. *Complementary control* pairs fire an action and its inverse under disjoint conditions, such as irrigation that turns on when soil is dry and off when rain is detected. *Hysteresis* pairs use non-overlapping threshold triggers that can never fire together, such as a grow-light that turns on below 200 lux and off above 800 lux. *Emergency overrides* let a safety rule deliberately countermand a routine one, such as a smoke-triggered unlock that supersedes a routine away-lock. Telling these apart from real conflicts requires EVar ranges, AFFECTS semantics, and context-mode exclusivity, exactly the information the knowledge graph encodes and rule text does not, which is why SHACR clears all 14. Although the perfect precision of §VII-C is testbed-specific, the underlying mechanism is structural rather than tuned.

VIII. DISCUSSION

A. Why the Knowledge Graph Is Decisive

The evaluation supports a single, sharp conclusion: the structured knowledge graph—not the choice of language model or prompt—is what makes accurate conflict detection possible. The ablation in §VII-D holds the model fixed and varies only the presence of the graph. The gap is large,

adding the graph raises F1 from 0.59 to 0.79 and reduces total classification errors by 36.7% over the YAML-only baseline that uses the same model. Because nothing but the representation changes, the improvement cannot be attributed to model capability; it is the difference between a baseline that misses more than half of all conflicts (recall 0.48) and a system that detects four in five (recall 0.80), and it follows directly from making physical dependencies traversable rather than leaving them implicit in rule text.

Compared with the graph, few-shot calibration plays a minor role, and how it acts shows why the graph is decisive. Balanced examples lift SHACR from F1 0.79 to 0.95 by eliminating the false positives induced by structurally similar but benign rule pairs, yet the same examples barely move the graph-free baselines and, for YAML-only Claude, even reduce recall. This asymmetry is not incidental. A few-shot example can teach a decision rule, but that rule is actionable only where its discriminating features (AFFECTS composition, EVar ranges, and context-mode exclusivity) are present. The graph supplies these features, so calibration takes effect. Without it, the same prompting has no evidence to act on. Graph grounding and prompt calibration are therefore complementary rather than interchangeable, and the latter is conditional on the former.

B. Where Detection Is Hardest, and What It Means for Deployment

Detection is not uniform across classes, and the asymmetry is informative. Logical conflicts are recovered most reliably because their evidence is local—a single contradiction on a shared state node. Physical conflicts, invisible to text-based methods, are detected with balanced precision and recall once the multi-hop AFFECTS paths are explicit. Semantic conflicts remain hardest (recall 0.52): they require the model to judge that an environmental side effect constitutes an undesired interaction, and their misses concentrate in apartments with long dependency chains, pointing to EVar modeling completeness as the primary lever for improvement.

The practical significance is that the classes differ in consequence as well as in difficulty. A physical conflict can simultaneously be a safety hazard, a security hole, and a reliability failure along one AFFECTS path, whereas a logical conflict typically produces only an oscillating actuator. A deployment in a safety-critical setting—elderly or medical care, for instance—should therefore bias the few-shot calibration toward recall on physical and high-consequence semantic conflicts, accepting some additional false positives. The few-shot mechanism makes this trade-off available as a lightweight, prompt-level adjustment, with no retraining and no change to the graph schema or retrieval pipeline.

C. Positioning and Future Directions

SHACR's probabilistic reasoning is deliberately aimed at the conflict classes that formal methods cannot express. For strictly logical contradictions, SMT solvers and finite-state verifiers offer deterministic guarantees that SHACR does not; a natural hybrid would route logical conflicts to

a formal checker and reserve the GraphRAG-LLM pipeline for the semantic and physical classes that require reasoning over natural-language descriptions and implicit environmental dependencies—retaining formal guarantees where they are achievable and LLM flexibility where they are not.

Three engineering directions remain. On privacy, SHACR already transmits only the relevant apartment subgraph per session rather than the full graph; locally hosted models are the principal route for deployments where data residency is binding. On cost, the per-session LLM calls that are acceptable at apartment scale will, at building scale, require incremental scanning that re-analyzes only rules changed since the last cycle rather than full rescans. On portability, production use requires platform-specific ingestion adapters to normalize SmartThings, Apple HomeKit, Google Home, and Home Assistant formats into the unified schema—an engineering prerequisite, not a research barrier, but a necessary step before SHACR is deployed on in-the-wild rule sets.

IX. CONCLUSION

Smart home automation conflicts are fundamentally a representation problem. Individual rules are locally correct, yet the hazards they create emerge only through interactions across shared devices, environmental variables, and physical topology, dependencies that remain invisible to any analysis confined to rule text. SHACR overcomes this by elevating physical cause-effect dependencies to first-class edges in a structured, unified knowledge graph. Anchoring LLM reasoning in this relational substrate turns conflict management from open-ended natural-language inference into deterministic multi-hop graph traversal, executed through a safe, validated, human-in-the-loop Scan-Explain-Repair-Validate workflow.

Our experimental evaluation confirms these design principles. On a 203-rule, 70-apartment testbed, introducing the knowledge graph while holding the underlying LLM fixed cuts total classification errors by 36.7% and raises the zero-shot F_1 -score from 0.59 to 0.79; balanced few-shot calibration lifts it further to 0.95. The same calibration yields negligible gains for graph-free baselines, proving that prompt engineering cannot substitute for an explicit relational model. SHACR thus establishes that structured knowledge representation, not prompt design or model choice, is the decisive factor for dependable IoT automation, intercepting conflicts before deployment and resolving them without burdening end users.

REFERENCES

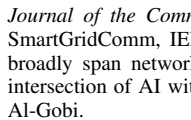
- [1] S. J. Philip, T. J. Luu, and T. Carte, "There's no place like home: Understanding users' intentions toward securing Internet-of-Things (IoT) smart home networks," *Computers in Human Behavior*, vol. 139, p. 107551, 2023.
- [2] Y. Cheng et al., "AutoIoT: Automated IoT platform using large language models," *IEEE Internet of Things Journal*, 2024.
- [3] Z. Kuang et al., "SAFE-TAP: Semantic-aware and fused embedding for TAP rule security detection," *Neurocomputing*, p. 131529, 2025.
- [4] Q. Lin et al., "Anomaly detection in smart IoT systems based on contextual semantics of behavior graphs," *IEEE Internet of Things Journal*, 2024.
- [5] A. J. B. Brush et al., "Home automation in the wild: Challenges and opportunities," in *Proc. SIGCHI Conf. Human Factors in Computing Systems (CHI)*, pp. 2115–2124, 2011.

- [6] J. Wang et al., "IoT-Praetor: Undesired behaviors detection for IoT devices," *IEEE Internet of Things Journal*, vol. 8, no. 2, pp. 927–940, 2020.
- [7] D. Chaki and A. Bouguettaya, "Fine-grained conflict detection of IoT services," in *Proc. IEEE Int. Conf. Services Computing (SCC)*, pp. 321–328, 2020.
- [8] R. Li et al., "SeIoT: Detecting anomalous semantics in smart homes via knowledge graph," *IEEE Trans. Information Forensics and Security*, 2024.
- [9] L. Chen et al., "TapChecker: A lightweight SMT-based conflict analysis for trigger-action programming," *IEEE Internet of Things Journal*, vol. 11, no. 12, pp. 21411–21426, 2024.
- [10] W. Ding and H. Hu, "On the safety of IoT device physical interaction control," in *Proc. ACM SIGSAC Conf. Computer and Communications Security (CCS)*, pp. 832–846, 2018.
- [11] A. Al Farooq et al., "IoT²: A formal method approach for detecting conflicts in large scale IoT systems," in *Proc. IFIP/IEEE Symp. Integrated Network and Service Management (IM)*, pp. 442–447, 2019.
- [12] V. Nagendra et al., "VISCR: Intuitive & conflict-free automation for securing the dynamic consumer IoT infrastructures," *arXiv:1907.13288*, 2019.
- [13] W. Ding, H. Hu, and L. Cheng, "IoTSafe: Enforcing safety and security policy with real IoT physical interaction discovery," in *Proc. Network and Distributed System Security Symp. (NDSS)*, 2021.
- [14] Z. Chen et al., "Multi-platform application interaction extraction for IoT devices," in *Proc. IEEE 25th Int. Conf. Parallel and Distributed Systems (ICPADS)*, pp. 990–995, 2019.
- [15] H. Chi, Q. Zeng, and X. Du, "Detecting and handling IoT interaction threats in multi-platform multi-control-channel smart homes," in *Proc. 32nd USENIX Security Symposium*, pp. 1559–1576, 2023.
- [16] M. Giudici et al., "Generating HomeAssistant automations using an LLM-based chatbot," *arXiv:2505.02802*, 2025.
- [17] J. Rey-Jouanchicot et al., "Leveraging large language models for enhanced personalised user experience in smart homes," *arXiv:2407.12024*, 2024.
- [18] N. Jahanbakhsh et al., "Leveraging retrieval-augmented generation for automated smart home orchestration," *Future Internet*, vol. 17, no. 5, p. 198, 2025.
- [19] Y. Dong et al., "ChatIoT: Large language model-based security assistant for Internet of Things with retrieval-augmented generation," *arXiv:2502.09896*, 2025.
- [20] P. Lewis et al., "Retrieval-augmented generation for knowledge-intensive NLP tasks," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 33, pp. 9459–9474, 2020.
- [21] J. Pan et al., "Knowledge graphs and large language models: A survey," *Machine Learning*, Springer, 2025.
- [22] L. Liang et al., "KAG: Boosting LLMs in professional domains via knowledge augmented generation," in *Companion Proc. ACM Web Conference 2025*, pp. 334–343, 2025.
- [23] K. Guu et al., "Retrieval augmented language model pre-training," in *Proc. Int. Conf. Machine Learning (ICML)*, pp. 3929–3938, 2020.
- [24] Y. Wang et al., "Knowledge graph prompting for multi-document question answering," in *Proc. AAAI Conf. Artificial Intelligence*, vol. 38, pp. 19206–19214, 2024.
- [25] S. Davidoff et al., "Principles of smart home control," in *Proc. 8th Int. Conf. Ubiquitous Computing (UbiComp)*, 2006.
- [26] J. Huang and M. Cakmak, "Supporting mental model accuracy in trigger-action programming," in *Proc. ACM Int. Joint Conf. Pervasive and Ubiquitous Computing (UbiComp)*, 2015.
- [27] D. Caivano et al., "Supporting end users to control their smart home," *Journal of Systems and Software*, 2019.
- [28] W. He et al., "When smart devices are stupid: Negative experiences using home smart devices," in *Proc. IEEE Security and Privacy Workshops (SafeThings)*, 2019.
- [29] W. Brackenbury et al., "How users interpret bugs in trigger-action programming," in *Proc. CHI Conf. Human Factors in Computing Systems*, 2019.
- [30] V. Koshy et al., "“We just use what they give us”: Understanding passenger user perspectives in smart homes," in *Proc. CHI Conf. Human Factors in Computing Systems*, 2021.
- [31] L. Zhang et al., "Helping users debug trigger-action programs," *Proc. ACM on IMWUT*, vol. 6, no. 4, p. 196, 2022.
- [32] E. Becks et al., "Complexity of smart home setups," *Technologies*, vol. 11, no. 1, p. 9, 2023.
- [33] Z. Zaidi et al., "A user-centric evaluation of smart home resolution approaches for conflicts between routines," *Proc. ACM on IMWUT*, vol. 7, no. 1, p. 45, 2023.
- [34] L. Y. Hao et al., "From frustration to function: A study on usability challenges in smart home IoT devices," in *Proc. IEEE Consumer Communications and Networking Conf. (CCNC)*, 2024.
- [35] M. P. Woźniak et al., "Connecting the dots: How users understand and diagnose smart home ecosystems," *Int. Journal of Human-Computer Studies*, vol. 203, p. 103559, 2025.
- [36] H. Hazazi and M. Shehab, "Exploring end users' perceptions of smart lock automation," in *Proc. European Symp. Usable Security (EuroUSEC)*, pp. 112–124, 2024.
- [37] T. B. Brown et al., "Language models are few-shot learners," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 33, pp. 1877–1901, 2020.
- [38] I. Beltagy et al., "Zero- and few-shot NLP with pretrained language models," in *Proc. 60th Annual Meeting of ACL: Tutorial Abstracts*, pp. 1–7, 2022.
- [39] Q. Zhang, S. Chen, Y. Bei, Z. Yuan, H. Zhou, Z. Hong, H. Chen, Y. Xiao, C. Zhou, J. Dong et al., "A survey of graph retrieval-augmented generation for customized large language models," *arXiv preprint arXiv:2501.13958*, 2025.
- [40] A. Singh, A. Ehtesham, S. Kumar, T. T. Khoei, and A. V. Vasilakos, "Agentic retrieval-augmented generation: A survey on agentic RAG," *arXiv preprint arXiv:2501.09136*, 2025.
- [41] Anthropic, "Model Context Protocol: An open standard for connecting AI assistants to tools and data sources," *Anthropic Technical Documentation*, 2024. [Online]. Available: <https://modelcontextprotocol.io>
- [42] T. Wald, "HomeAssistantExamples: Home Assistant automations and examples," GitHub repository, `airtouch4-deprecated` directory. [Online]. Available: <https://github.com/tomwaldnz/HomeAssistantExamples/tree/main/automations/airtouch4-deprecated>. [Accessed: 20-Jun-2026].
- [43] Home Assistant, "Automations in YAML," Home Assistant Documentation. [Online]. Available: <https://www.home-assistant.io/docs/automation/yaml/>. [Accessed: 20-Jun-2026].
- [44] SmartThings Developers, "Sample-RulesAPI," GitHub repository, Samsung SmartThings Developer Programs. [Online]. Available: <https://github.com/SmartThingsDevelopers/Sample-RulesAPI>. [Accessed: 20-Jun-2026].

PLACE
PHOTO
HERE



Leena A. Marghalani received her M.S. degree in Security and Information Assurance from King Fahd University of Petroleum and Minerals (KFUPM), Dhahran, Saudi Arabia, and her B.S. degree in Cybersecurity and Digital Forensics from Imam Abdulrahman Bin Faisal University (IAU), Dammam, Saudi Arabia. Her research interests include IoT security, AI-driven cybersecurity, and usable security.



Walid Aljoby (Member, IEEE) received the Ph.D. degree in computer science from the National University of Singapore (NUS) in 2020, and held post-doctoral positions at NUS and Carnegie Mellon University (CMU). He is currently an Assistant Professor of computer science at the King Fahd University of Petroleum and Minerals (KFUPM). He has authored articles in the *IEEE Journal on Selected Areas in Communications*, *IEEE Transactions on Network and Service Management*, *IEEE Open Journal of the Computer Society*, *IEEE Open Journal of the Communications Society*, *IEEE ICNP*, *IEEE NetSoft*, *IEEE SmartGridComm*, *IEEE ICC*, and *IEEE/IFIP NOMS*. His research interests broadly span networked systems and security, with a current focus on the intersection of AI with networking and security. He is also known as Waleed Al-Gobi.



Suayb S. Arslan (Senior Member, IEEE) received the M.Sc. and Ph.D. degrees in Electrical and Computer Engineering from the University of California at San Diego, La Jolla, CA, USA, in 2009 and 2012. From 2012 to 2016, he was a Senior Researcher at Quantum Corporation, Irvine, CA, and between 2022 and 2024, he served as visiting associate Professor in the SinhaLab at MIT, Cambridge, MA, USA. He is currently a Professor in the Department of Computer Engineering and Director of the Institute for Data Science and AI at Boğaziçi University, Istanbul, Türkiye. His research interests include information theory, digital communications, cloud and quantum computing, and AI for IoT systems. He is an Executive Editor of *IoT Journal* (Elsevier).