

KGIoT: Grounding LLM Agents in Actionable Spaces for Reliable Smart Home Automation

LINYA FU, The Hong Kong Polytechnic University, China

LEMING SHEN, The Hong Kong Polytechnic University, China

YUANQING ZHENG*, The Hong Kong Polytechnic University, China

Smart-home automation is widely deployed yet remains difficult for ordinary users to author and maintain. Recent LLM-based approaches translate natural language into trigger-action programs (TAPs), but often produce invalid or non-executable automations due to missing explicit modeling of device capabilities and execution constraints. We argue that reliable TAP synthesis should be formulated as constrained planning over a well-defined action space, rather than free-form text generation. To this end, we present KGIoT, a framework that realizes a capability-centric abstraction as a structured knowledge graph, grounding LLM-based automation in executable device capabilities. KGIoT realizes this abstraction by constructing a KG from platform metadata, organizing device operations into normalized capability primitives, and deriving a task-specific action space with explicit dependency and interlock constraints. Instead of generating TAPs directly, KGIoT decomposes the process into structured constraint parsing and KG-based grounding, enabling generation within a closed and executable action space. Extensive evaluations show that KGIoT achieves substantial improvements in executable correctness (up to 25.0 percentage points over the strongest baseline), while reducing token consumption by over 4.7 $\times$. A real-world case study demonstrates that it can reliably generate executable automations in a practical smart-home setting. A comprehensive user study further shows improved usability, interpretability, and user trust compared to baseline approaches. These results highlight the importance of capability-centric representations, structured grounding, and explicit action spaces for bridging natural language with dependable, human-centered smart-home systems.

CCS Concepts: • **Human-centered computing** → **Ubiquitous and mobile computing**.

Additional Key Words and Phrases: Trigger-action programming, Knowledge graph, LLMs, Internet of Things

ACM Reference Format:

Linya Fu, Leming Shen, and Yuanqing Zheng. 2026. KGIoT: Grounding LLM Agents in Actionable Spaces for Reliable Smart Home Automation. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 10, 4, Article 214 (December 2026), 36 pages. <https://doi.org/10.1145/3857975>

1 Introduction

Equipped with Internet of Things (IoT) devices, smart homes enable users to automate everyday routines [14, 24, 27, 37, 38, 64]. Most mainstream smart home platforms, such as Apple HomeKit [2], Google Home [4], Samsung SmartThings [10], and Home Assistant [6], adopt Trigger-Action Programming (TAP) [56] as the automation paradigm, where users define rules in the form of “IF *triggers* THEN *actions*”. For example, a user

*Corresponding author.

Authors’ Contact Information: Linya Fu, Department of Computing, The Hong Kong Polytechnic University, Hong Kong, China, linyafu@connect.polyu.hk; Leming Shen, Department of Computing, The Hong Kong Polytechnic University, Hong Kong, China, leming.shen@connect.polyu.hk; Yuanqing Zheng, Department of Computing, The Hong Kong Polytechnic University, Hong Kong, China, csyqzheng@comp.polyu.edu.hk.



This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2026 Copyright held by the owner/author(s).

ACM 2474-9567/2026/12-ART214

<https://doi.org/10.1145/3857975>

Table 1. Comparison of LLM-based smart-home automation generation methods.

System	Knowledge Source	Grounding Strategy	Generation Constraint	Validity Enforcement
Sasha [35]	Home platform metadata	LLM filtering	None	None
ChatIoT [26]	Home platform metadata	Semantic filtering	Implicit (filtered context)	Post-hoc validation
AutoIoT [19]	User-uploaded images + device manuals	None	None	Post-hoc validation
HomeGenii [71]	Home platform metadata + Expert rulebase	Retrieval (RAG)	Implicit (retrieved rules)	Post-hoc validation
KGIoT (Ours)	Home platform metadata	KG-based	Explicit (action space)	Pre-constrained

request such as: “Turn on the bathroom light when someone enters the bathroom” can be translated into a TAP rule as “*trigger: (bathroom motion detected), action: (turn on bathroom light)*”.

However, synthesizing precise and functionally correct TAP rules remains challenging in practice [17, 30, 70]. Even with no-code tools such as IFTTT [7, 57] and Zapier [11], users must manually compose rules across heterogeneous devices and interfaces [53]. To alleviate this burden, prior work has explored approaches like interaction support [39–41], behavior-driven inference [23, 69], formal verification [68], and learning-based approaches [52, 66, 67]. These methods, however, are often tailored to specific tasks or rely on predefined structures, limiting their generalizability to diverse smart-home environments.

Large language models (LLMs) [12, 18, 20, 54] have recently shown strong potential for smart-home automation by translating natural-language requests into executable TAP rules [19, 22, 26, 35, 71]. Existing systems explore diverse strategies, including multi-agent prompting, retrieval-augmented generation, and rule retrieval, substantially lowering the barrier to automation authoring. Representative LLM-based smart-home automation systems are summarized in Table 1. Despite their encouraging progress, existing approaches still exhibit several fundamental limitations that hinder reliable TAP generation.

L1: Incorrect grounding of devices and capabilities. Existing approaches (Table 1) often fail to identify the correct devices and capabilities because grounding relies heavily on semantic similarity or retrieval quality, making it sensitive to wording variations and retrieval errors. As a representative example shown in Fig. 1, ChatIoT selects candidate devices by computing embedding similarities between device descriptions and user requests and uses predefined thresholds for filtering. Nevertheless, the bedroom window sensor (labeled as “*magnet_sensor*”), which serves as the trigger for detecting window opening, can be missed due to the low semantic similarity.

L2: Unreliable TAP generation. Even when relevant devices are identified, generating valid TAP rules still requires reasoning over structured relationships among device capabilities that determine their valid compositions. However, existing approaches (Table 1) provide only implicit generation constraints, typically through mechanisms such as context filtering, retrieval, or candidate selection, without explicitly representing such structural relationships. As a result, LLMs lack a well-defined space of valid compositions and have to rely on semantic similarity, often leading to structurally invalid combinations. For example, for a request such as “when the room is dark”, the system should identify capabilities related to environmental illumination sensing. However, although device capabilities are present in the context, their associations with specific devices are not explicitly enforced. The LLM may therefore misassociate properties, incorrectly using *brightness* (a controllable property of a light) instead of *illuminance* (a sensing property of an illumination sensor) and generating *illumination_sensor.brightness* instead of the valid *illumination_sensor.illuminance*. Worse still, when relevant devices are excluded due to imperfect grounding (L1), LLMs may hallucinate non-existent devices or capabilities to satisfy the request, as illustrated in Fig. 1. Such outputs violate the underlying device capability structure, resulting in invalid non-executable TAPs.

L3: Ineffective post-hoc validation. Existing post-hoc validation mechanisms often fail to recover from grounding and composition errors once invalid TAPs have already been generated. Many existing approaches

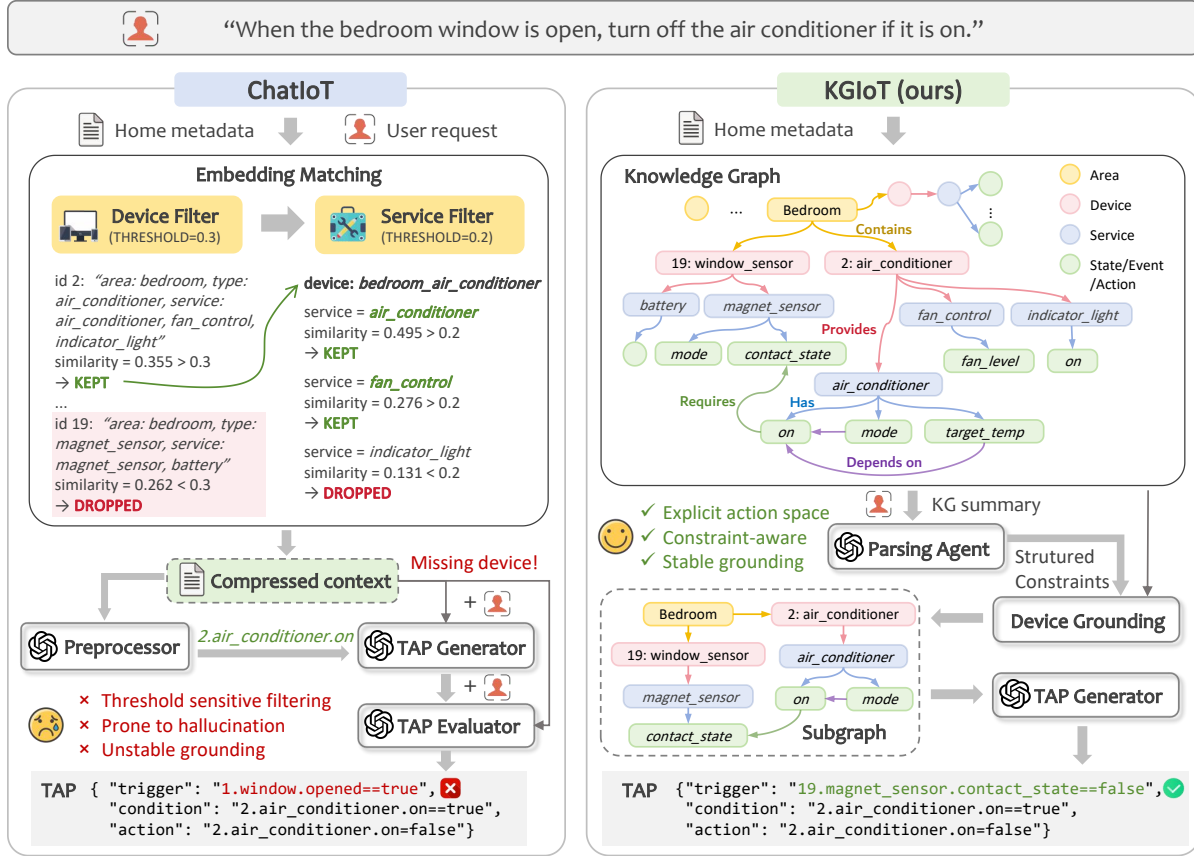


Fig. 1. Comparison between ChatIoT [26] and KGIoT. ChatIoT relies on threshold-based filtering, which may exclude relevant devices (e.g., the window sensor), leading to incomplete context, hallucinated triggers, and ultimately invalid TAP rules that fail during deployment. In contrast, KGIoT constructs a structured knowledge graph and derives a grounded subgraph that explicitly defines the valid action space, enabling constraint-aware generation and producing correct and executable TAPs.

(Table 1) therefore rely on iterative refinement after TAP generation. Nonetheless, such post-hoc validation is often ineffective because it operates on already incorrect or incomplete TAP outputs without access to the underlying device semantics, making it difficult to recover from errors. For instance, in Fig. 1, the model assigns an incorrect device identifier (e.g., using 1 instead of the correct 19) to a hallucinated window sensor in order to conform to the required output format. This shows that post-hoc validation cannot reliably correct invalid outputs and may even propagate errors under incorrect context.

Taken together, these limitations suggest that the key challenge lies in providing structured and precise constraints for TAP generation. We therefore argue that TAP generation should be formulated not as a purely language generation problem, but as a *constrained planning* problem grounded in targeted smart-home environments.

Such constrained planning, however, requires an appropriate representation of the planning space. Existing smart-home abstractions are primarily device- or entity-centric, where functionalities remain tightly coupled to individual hardware. While suitable for device management, these representations do not directly capture the

executable capabilities that planners reason about. We therefore argue that the planning space should instead be organized around a capability-centric abstraction, in which executable capabilities become the primary semantic units for connecting user intents, heterogeneous devices, operational constraints, and executable actions.

To this end, we propose KGIoT, a capability-centric knowledge graph (KG) framework for reliable TAP generation in smart homes. KGIoT organizes knowledge around executable capabilities rather than individual devices, explicitly modeling capability-device associations and composition constraints for grounded planning. Unlike existing smart-home KGs that primarily support semantic modeling or inference [23, 33, 36, 42, 59], KGIoT uses this representation to support automation synthesis. As illustrated in Fig. 1, KGIoT constructs a capability-centric KG of the smart-home environment, uses it to parse user intent into structured constraints, and grounds these constraints to construct a task-specific executable subgraph. Through this design, KGIoT shifts LLM-based TAP generation from free-form text generation to planning over a structured action space, enabling consistent grounding, valid rule synthesis, and reliable executability of generated TAPs.

Realizing this capability-centric abstraction for constrained planning raises three key technical challenges. First, how to construct a capability-centric KG that explicitly represents executable device capabilities and their composition constraints. Second, how to leverage this representation to reliably ground user intent and derive a well-defined action space for TAP generation. Third, how to enable the LLM to perform planning within this constrained action space, rather than relying on unconstrained generation. These challenges remain largely unexplored in existing smart-home TAP generation systems.

To address these challenges, we propose three key technical modules (§ 3.2–3.4). We implement KGIoT and evaluate it on a benchmark of 60 natural-language automation requests constructed for this study across three smart-home environments. We compare against several representative LLM-based baselines, including recent state-of-the-art systems [19, 26, 35, 71]. Results show that KGIoT achieves superior executable correctness, reaching 100.0%, 100.0%, and 85.0% across the three settings, respectively, outperforming the strongest baseline by up to 25.0 percentage points. Moreover, KGIoT significantly reduces token consumption compared to full-context prompting, using over 4.7× fewer tokens while maintaining higher executable correctness. We also conduct a real-world case study demonstrating the end-to-end workflow of KGIoT in a real smart home testbed, showing that it can reliably generate and deploy executable automations in practice. The user study shows that KGIoT significantly improves both user experience and system reliability compared to a vanilla LLM-based baseline that directly generates TAP rules from full context, achieving substantially higher acceptance and validity rates (94.64% vs. 50.00% and 100.00% vs. 76.79%), and consistently higher ratings across all questionnaire dimensions.

In summary, our contributions are as follows:

- We formulate smart-home TAP generation as a *constrained planning* problem over a capability-centric abstraction, realized as a KG that derives an explicit and executable action space for LLM planning.
- We develop KGIoT, a structured framework that grounds LLM-based planning through constraint parsing and subgraph-based construction of a closed action space, enabling accurate and efficient mapping from user intent to executable TAP rules.
- We demonstrate that KGIoT significantly improves the reliability and efficiency of TAP generation compared to prior LLM-based approaches, by preventing invalid compositions during generation and reducing reliance on post-hoc validation.

2 Related Work

2.1 LLM-based Agents for Smart-Home Automation

LLMs have recently been explored across a broad range of application domains, including ubiquitous computing and human-computer interaction [13, 25, 32, 61, 63], IoT program synthesis [48–50], and intelligent assistant

Table 2. Comparison of KG-based approaches for smart-home systems.

Method	Task	KG Role	Mechanism
Corno et al. [23]	Rule recommendation	Rule semantics	Semantic reasoning
Wu et al. [59]	Rule recommendation	User-rule interaction	Feature-based ranking
A3ID [60]	Conflict detection	External semantics	Text-based reasoning
Huang et al. [29]	Conflict detection	Environment modeling	Relation extraction
Huang et al. [33]	Service personalization	Context representation	Scenario matching
Li et al. [36]	Anomaly detection	State modeling	Traffic inference
Mulayim et al. [42]	KGQA	Query support	Semantic parsing
Wu et al. [58]	Rule recommendation	User-rule semantics	LLM-based parsing
KGloT (Ours)	TAP generation	Capability model	Constraint-aware grounding

systems and user-centric applications [31, 45, 55, 62, 65]. With their strong semantic understanding and zero-shot reasoning capabilities, LLMs enable new ways of mapping natural language to smart-home automation.

Early works focus on interpreting high-level user intentions and executing tasks through planning or interaction. For example, Sasha [35] generates action plans or automation routines from loosely specified goals (e.g., “make it cozy”), while SAGE [47] adopts an autonomous agent paradigm based on iterative planning and tool invocation.

Building on these capabilities, subsequent work has explored directly translating natural language into executable TAP rules. ChatIoT [26] achieves this via context-aware compressive prompting and a multi-stage pipeline. HomeGenii [71] formulates TAP generation as a retrieval-augmented process, leveraging expert-written rules and environmental context to guide generation toward more complex automation behaviors.

Other approaches enhance TAP generation through data-driven or post-processing techniques, including embedding-based retrieval and ranking [22], fine-tuned autoregressive models [21], and graph-based recommendation [58]. More recent efforts further incorporate environment grounding and execution constraints into the generation process, such as formal verification and repair [19], along with multimodal context integration [51].

Despite these advances, existing LLM-based approaches largely rely on textual representations and heuristic pipelines without explicitly modeling structured relationships and constraints in smart-home environments. This limits their ability to consistently generate reliable TAPs.

2.2 Knowledge Graphs in Smart Homes

KGs have been used in smart-home systems to model heterogeneous entities and their relationships, supporting applications such as recommendation, analysis, and context modeling, as summarized in Table 2. Some approaches capture relationships between users, rules, and device functionalities for rule recommendation, such as RecRules [23], rtar [59], and LLM4TAP [58]. Others model device semantics and environmental interactions for reliability and security analysis, including conflict detection [29, 60] and anomaly detection [36]. KGs have also been explored as structured context representations to support scenario matching [33] and LLM-based query and retrieval [42].

Overall, existing KG-based approaches primarily serve as semantic support for matching, recommendation, and analysis. However, they are not explicitly designed to support the composition of executable automation rules. This limits their ability to guide LLMs in generating valid and executable TAPs from user intent, leaving a gap between existing KG designs and the requirements of reliable automation in smart-home environments. In this work, we address this gap by leveraging a capability-centric KG to explicitly model device functionalities and their compositional constraints, enabling structured grounding and constrained TAP generation with LLMs.

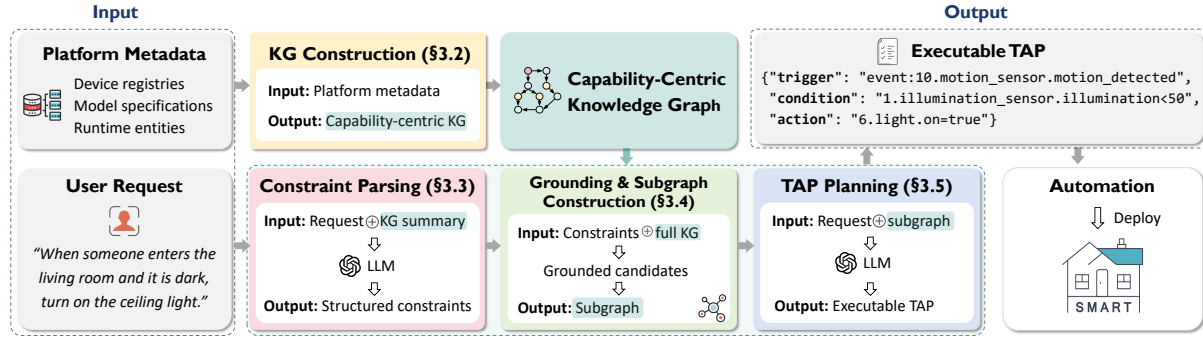


Fig. 2. Overview of KGIoT. The framework takes a natural-language user request and platform metadata as inputs and generates an executable TAP, via KG construction, constraint parsing, grounding, and constrained planning over a subgraph-based closed action space. TAPs specify device-level operations. Numerical prefixes (e.g., 10, 1, 6) denote device identifiers.

3 The KGIoT Framework

3.1 System Overview

Fig. 2 presents the overall framework of KGIoT, which transforms a natural-language user request into an executable TAP by grounding it in a capability-centric KG and performing constrained planning over the resulting action space.

As illustrated in Fig. 2, KGIoT takes two inputs: a natural-language user request specifying the desired automation and the smart-home platform metadata. Such metadata are exposed by smart-home platforms (e.g., Home Assistant [6]) in advance, including device registries, model specifications, and runtime entities (detailed in § 3.2.1). The output of KGIoT is an executable TAP. For the example in Fig. 2, given the request “When someone enters the living room and it is dark, turn on the ceiling light”, the resulting TAP is: `{\"trigger\": \"event:10.motion_sensor.motion_detected\", \"condition\": \"1.illumination_sensor.illumination<50\", \"action\": \"6.light.on=True\"}`, where the condition corresponds to low illumination (i.e., below 50 lux). The TAP is then translated into the platform-specific automation format for deployment.

3.1.1 Workflow. KGIoT decomposes TAP generation into two stages: (1) an offline stage realizing the capability-centric abstraction as a KG that defines the executable action space and (2) an online stage performing intent-guided grounding and constrained TAP planning within this space. The overall workflow proceeds as follows:

- (1) **KG construction (§ 3.2).** KGIoT constructs a capability-centric KG that organizes devices and their capabilities into a structured representation, while preserving their spatial relationships within the environment. The KG is initialized at platform startup and is automatically updated as devices change.
- (2) **Constraint parsing (§ 3.3).** Given a user request and a lightweight KG summary derived from the full KG, a parsing module extracts structured constraints over triggers, conditions, and actions, mapping user intents into a preliminary TAP representation aligned with the KG schema.
- (3) **Grounding and subgraph construction (§ 3.4).** The parsed constraints are grounded in the KG to retrieve candidate devices and capabilities, which are further pruned to construct a task-specific executable subgraph.
- (4) **TAP planning (§ 3.5).** Finally, a TAP planner performs constrained planning over the subgraph-defined action space to construct an executable TAP. The resulting TAP is then translated into a platform-specific automation format for deployment.

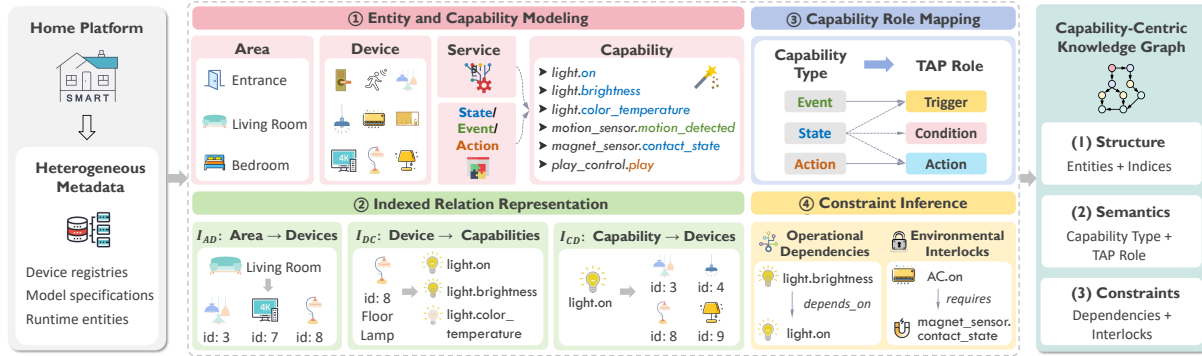


Fig. 3. Construction pipeline of the capability-centric KG. Starting from heterogeneous platform metadata, the pipeline performs: (1) entity and capability modeling, (2) indexed relation representation, (3) capability role mapping, and (4) constraint inference. The resulting KG is organized into three complementary layers: a structural layer, a semantic layer, and a constraint layer, forming a structured representation that supports the construction of an executable action space for TAP generation.

3.2 Capability-Centric Knowledge Graph

This module realizes the capability-centric abstraction as a structured KG. Rather than organizing the environment around individual devices, the proposed representation organizes smart-home knowledge around executable capabilities, while preserving their associations with device instances, spatial organization, and operational constraints.

Challenge C1. Conventional smart-home representations are typically organized hierarchically (e.g., area $\rightarrow$ device $\rightarrow$ capability), where executable functionalities remain tightly coupled to individual hardware. Although suitable for device management, such representations do not naturally support capability selection and composition required for TAP planning. As a result, it remains challenging to reliably generate valid TAP rules, especially when multiple devices with diverse capabilities are involved.

Our Solution. To address this challenge, we realize the proposed capability-centric abstraction as a structured KG, in which executable capabilities are treated as first-class entities. This representation provides a clear structure to facilitate the selection and combination of smart devices with diverse capabilities. As illustrated in Fig. 3, the KG is constructed from platform metadata through four main stages: (1) device entity and capability modeling, (2) indexed relation representation, (3) capability role mapping, and (4) constraint inference.

3.2.1 Knowledge Source. Smart-home platforms (e.g., Home Assistant [6]) expose heterogeneous metadata describing the deployed environment. When smart devices are installed, they are manually registered by users with the platform. As a result, key information about the deployed devices (e.g., their locations, types, and supported capabilities) is recorded and readily available within the platform. KGIoT fetches such information and constructs the KG by integrating three kinds of platform metadata: device registries, model specifications, and runtime entities. Specifically, the device registry describes which devices are present in the environment, along with their unique identifiers and associated areas (e.g., a light with ID 6 located in the living room). Model specifications define the supported capabilities of each device type (e.g., a light can be turned on/off or have its brightness adjusted). Runtime entities further specify how these capabilities are instantiated within the platform (e.g., a light entity currently being “on” with brightness 70%).

3.2.2 Entity and Capability Modeling. The KG models the smart-home environment as a hierarchical graph of entities and their relationships, with device capabilities explicitly represented as first-class elements.

Entities. We define three types of entities:

- *Areas* ($\mathcal{A}$) represent spatial regions in the home (e.g., living room).
- *Devices* ($\mathcal{D}$) correspond to physical instances deployed within these areas (e.g., a light).
- *Services* ($\mathcal{S}$) denote functional modules provided by devices (e.g., a light control service).

A single area may encompass multiple devices, each of which concurrently exposes a distinct set of services. In addition, each service is associated with a set of interaction primitives, including *states*, *events*, and *actions*. States represent observable properties or controllable parameters (e.g., on/off or brightness), events correspond to discrete signals that can trigger automations (e.g., motion detected), and actions denote executable operations of the device (e.g., playing media on a TV).

Capabilities. TAP rules operate on device capabilities rather than devices themselves. For example, a rule may turn on a light, set its brightness, or react to a motion signal, each corresponding to a specific capability exposed by a device. Accordingly, KGIoT adopts a capability-centric abstraction that models device operations at the level of these capabilities.

Therefore, we define each capability as a service-level interaction primitive:

$$c = (s, x), \quad x \in \{\text{state, event, action}\}, \quad (1)$$

where $s \in \mathcal{S}$ denotes a service, and x corresponds to one of its associated interaction types (i.e., state, event, or action). In practice, each capability is serialized as *service.x* (e.g., *light.on* denotes the on/off state of a light control service). Under this formulation, capabilities are defined in relation to services, rather than being modeled as independent nodes.

Fig. 4 illustrates a concrete example of the constructed KG, showing how entities are hierarchically organized, with capability primitives (states, events, and actions) visualized as leaf-level nodes attached to each service.

Capability Catalog. In practice, multiple devices may expose the same capability type (e.g., brightness control for different lights), leading to redundant and inconsistent representations across devices. As a result, it becomes difficult to reason about capabilities in a unified manner, since semantically identical operations are treated as separate, device-specific elements. To address this issue, KGIoT maintains a global *capability catalog* that abstracts redundant device-specific capability instances into a unified set, defined as:

$$C = \{c = (s, x) \mid s \in \mathcal{S}, x \in \{\text{state, event, action}\}, \exists d \in \mathcal{D} \text{ s.t. } d \text{ exposes } (s, x)\}, \quad (2)$$

where each element c corresponds to a unique capability that may be exposed by multiple devices. This design allows capability semantics to be defined once and reused across multiple devices, establishing a consistent abstraction for downstream reasoning (§ 3.2.4).

3.2.3 Indexed Relation Representation. During TAP generation, the system needs to efficiently retrieve devices and capabilities that satisfy structured constraints derived from user requests (e.g., location, device type, and capability). Conventional KG representations typically store relations as edges (e.g., triples or adjacency lists) and retrieve entities through multi-hop traversal [46] or query execution (e.g., SPARQL [43]). However, such mechanisms can be inefficient in our setting, as these general-purpose methods typically require multiple iterations of traversal or formulate complex queries, imposing additional overhead for repeated constraint-based filtering in our tasks. To mitigate this limitation, KGIoT adopts an index-based representation that encodes key relationships in the smart-home environment—such as which devices are located in each area and which capabilities are provided by each device—as direct lookup mappings, while retaining the logical relationships of the original graph. Specifically, we define three indexed mappings:

- *Area-to-device* (I_{AD}) maps areas to their associated devices.
- *Device-to-capability* (I_{DC}) maps devices to the capabilities they expose.
- *Capability-to-device* (I_{CD}) maps capabilities to the devices that provide them.

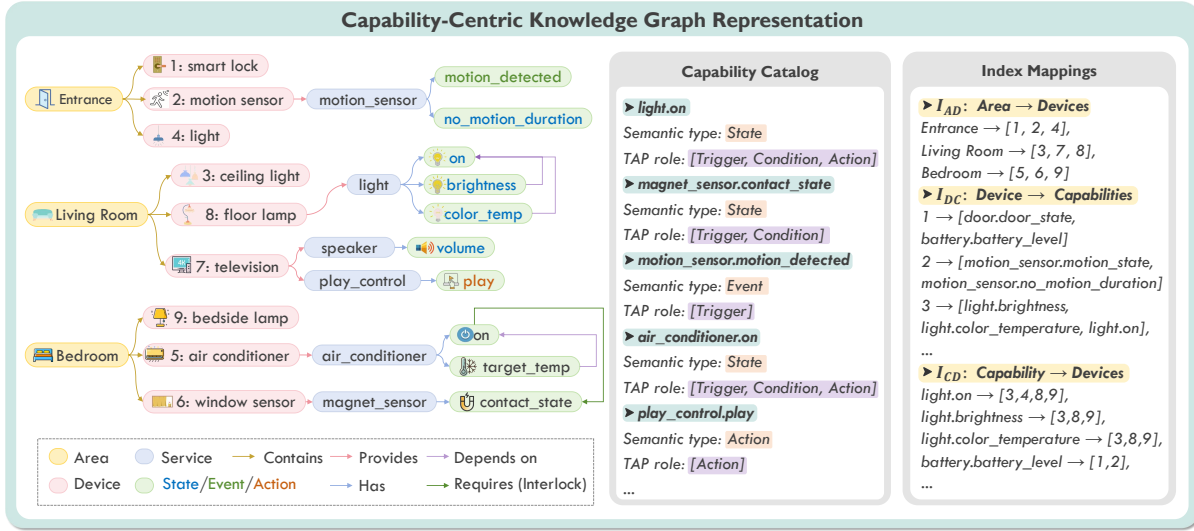


Fig. 4. Example of the constructed capability-centric KG for a smart-home environment. The KG is the structural graph with dependency and interlock relations. The capability catalog stores semantic types and admissible TAP roles, while index mappings support efficient retrieval (area-to-device, device-to-capability, and capability-to-device).

These indexed relation representations facilitate direct lookup of entities relevant to user requests, enabling efficient grounding (§ 3.4), and better aligning the KG representation with the constraint-driven retrieval required by our task.

3.2.4 Capability Role Mapping. Structural relations alone may not determine how a capability can be used in automation rules. Each capability has a constrained set of admissible roles in a TAP rule, determined by its semantics. For example, an event capability such as *motion detected* can serve as a trigger, but cannot be used as a condition or an action. To address this, KGIoT defines a role-mapping strategy that assigns each capability a set of admissible roles in a TAP rule, *i.e.*, *trigger*, *condition*, and *action*, based on its underlying type (*i.e.*, *state*, *event*, or *action*). As illustrated in Fig. 3, we adopt a type-driven mapping aligned with TAP semantics: event capabilities serve as triggers; state capabilities may serve as triggers or conditions, and as actions when their synchronized platform metadata indicate write access; and action capabilities are restricted to actions. This mapping constrains each capability to semantically valid roles, thereby reducing the feasible action space during TAP generation. The mapping is incorporated into the KG summary provided to the parsing agent (§ 3.3), enabling role-aware interpretation of capabilities during constraint parsing.

3.2.5 Operational Constraint Inference. While the above KG design captures available devices and their capabilities, it may still be insufficient for TAP generation, as user requests often omit necessary operational prerequisites. For example, a user may request “*if the bedroom is too hot, set the air conditioner to 25 °C*”, without specifying necessary prerequisites, such as turning on the air conditioner before adjusting its temperature or ensuring that the window is closed. Without such constraints, a generated TAP may be syntactically valid but fail to execute as intended or lead to undesirable outcomes.

To address this, KGIoT incorporates operational constraints into the KG as explicit relations during construction. These constraints are automatically instantiated by applying a small set of reusable domain-level constraint templates to the synchronized platform metadata. As a result, home-specific constraint relations are generated

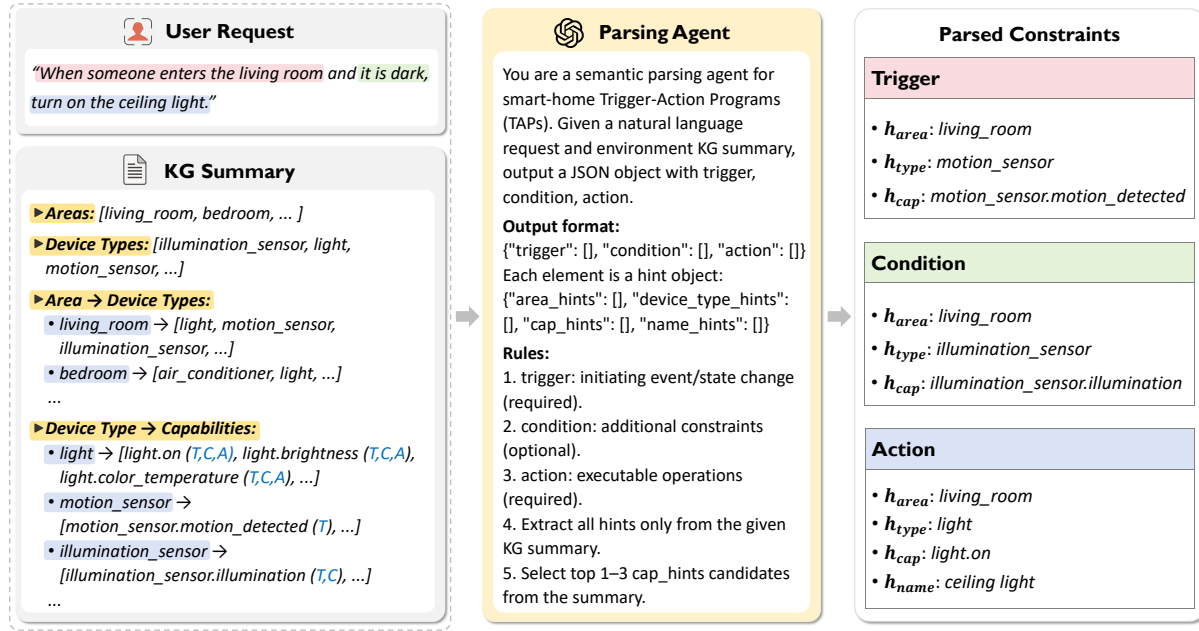


Fig. 5. Example workflow of constraint parsing. Given a natural language user request and a lightweight KG summary, the parsing agent produces structured constraint blocks for the trigger, condition, and action fields. In the KG summary, capability annotations (T, C, A) denote admissible TAP roles, corresponding to *trigger*, *condition*, and *action*, respectively.

automatically during KG construction without per-device or per-request manual annotation. Specifically, we formalize two representative classes of constraints to capture common operational dependencies in smart-home environments.

(1) Operational Dependencies. Certain capabilities depend on prerequisite device states. For example, as illustrated in Fig. 4, the capability *light.brightness* of a floor lamp depends on *light.on*, as brightness adjustment is only effective when the light is powered on. We represent such dependencies as

$$c_i \xrightarrow{\text{depends on}} c_j, \quad (3)$$

where c_i and c_j are capabilities, and c_j must hold for c_i to be effective. These dependencies are inferred from capability metadata using general patterns of device operation. For example, when a service exposes a controllable power state, other adjustable properties within the same service are treated as dependent on that state. This enables KGIoT to automatically materialize intra-device dependency relations without device-specific annotations. Whenever new devices are synchronized from the smart-home platform, the corresponding dependency edges are regenerated automatically.

(2) Environmental Interlocks. Some capabilities depend on environmental states involving other devices. For example, as illustrated in Fig. 4, the capability *air_conditioner.on* of an AC is associated with the capability *magnet_sensor.contact_state* of a window sensor in the same area. We represent such constraints as

$$c_i \xrightarrow{\text{requires}} (c_j = v), \quad (4)$$

where c_i and c_j are capabilities, and c_j must satisfy the condition $c_j = v$ for c_i to be effective. These interlocks are instantiated using domain-level templates that link capabilities to relevant environmental states within the same area, allowing KGIoT to capture cross-device dependencies in a structured and reusable manner. These templates are defined at the capability level rather than for individual devices. During KG synchronization, they are automatically instantiated into device-specific interlock relations by matching synchronized capability metadata and area information.

Overall, these constraint relations augment the KG with execution semantics and are used during capability pruning (§ 3.4.2) and subgraph construction (§ 3.4.3). As a result, the task-specific subgraph is expanded with dependency and interlock relations, supporting the construction of an execution-aware action space for TAP planning.

3.2.6 Automatic Synchronization. The KG is automatically constructed from synchronized platform metadata together with a set of reusable capability-level constraint templates. Platform metadata (e.g., devices, entities, capabilities, and locations) are refreshed during KG construction and reconstructed when device additions or removals are detected. During reconstruction, newly discovered devices are automatically associated with their corresponding capabilities, and applicable dependency and interlock relations are instantiated by matching the synchronized metadata against the predefined templates. Consequently, no manual modification of the KG is required for individual device additions or removals. Only the reusable constraint templates need to be updated when introducing new categories of operational constraints.

Overall, the proposed KG serves as a capability-centric representation of the smart-home environment. The resulting KG integrates three complementary types of information: (1) structural information that captures device entities and their relations for efficient retrieval, (2) semantic information that defines capability abstractions and admissible TAP roles, supported by a global capability catalog that standardizes capability types across devices, and (3) inferred constraint relations that capture operational dependencies and environmental interlocks between capabilities. Figure 4 illustrates the resulting graph representation.

3.3 Constraint Parsing

This module bridges natural language requests and the capability-centric KG by extracting structured constraints that guide subsequent grounding.

Challenge C2. User requests are expressed in natural language, which often does not naturally align with the structured representation of devices and capabilities in the KG. For example, a request such as “when the room is dark” does not explicitly specify the underlying capability (e.g., illumination sensing) or the corresponding threshold. As a result, directly grounding relevant devices and capabilities may require searching over a large portion of the KG, making it difficult to efficiently and reliably identify the relevant entities.

Our Solution. To address this limitation, rather than grounding over the full KG, KGIoT first extracts a lightweight KG summary that preserves structural topology and capability-level semantics while abstracting away instance-level details. Based on this summary and the user request, we introduce a constraint parsing paradigm that transforms natural language into structured constraints, serving as a preliminary TAP specification for subsequent grounding. As illustrated in Fig. 5, the parsing agent produces structured constraint blocks for the trigger, condition, and action fields.

3.3.1 KG Summary. As shown in Fig. 5, the summary provides a schema-level abstraction of the environment, including: (1) available areas, (2) device types, (3) area-to-type mappings, and (4) type-level capability mappings. In addition, each capability is annotated with its admissible TAP roles (*trigger*, *condition*, *action*) based on the capability role mapping defined in § 3.2.4. For example, *light.on* can serve as a trigger, condition, or action, while *motion_sensor.motion_detected* is restricted to trigger roles.

Given a request, this structured summary constrains the parser to consider only valid device types, capabilities, and their admissible TAP roles. Instead of exposing the full KG, KGIoT uses this compact summary to preserve structural constraints while abstracting away instance-level details, enabling efficient parsing with significantly lower token consumption.

3.3.2 Parsing agent. The structured constraints are produced by an LLM-based parsing agent that maps the user request and the KG summary into trigger, condition, and action components, as illustrated in Fig. 5. The agent extracts structured constraint blocks that capture the semantic intent of each TAP component, such as area (h_{area}), device type (h_{type}), capability (h_{cap}), and optionally device name (h_{name}).

To account for uncertainty in language interpretation, the parser is allowed to produce a small set of candidate capability hints (typically top-1 to top-3) for each block. When the user intent is unambiguous, a single hint is produced; otherwise, multiple candidates are retained to preserve plausible alternatives for subsequent grounding. These candidates are ordered by the LLM according to their estimated semantic relevance to the user request. Subsequent grounding and sanitization remove candidates that are incompatible with the admissible capability roles or unavailable in the current environment.

Importantly, parsing is constrained by the KG summary, which defines a closed and structured capability space for the LLM. Rather than freely generating entities, the parser must select areas, device types, and capabilities from those defined in the summary, so that extracted hints correspond to valid elements in the environment. Moreover, capability-level role annotations further guide the parsing process, restricting how each capability can be used in trigger, condition, and action components. Together, these constraints reduce spurious or invalid compositions and improve the reliability of the parsed constraints.

3.3.3 Structured Constraint Representation. The output of the parsing agent is a set of constraint blocks

$$\mathcal{B} = \{b_1, b_2, \dots, b_m\}, \quad (5)$$

where each block corresponds to a trigger, condition, or action component. Each block $b_i \in \mathcal{B}$ is represented by a subset of semantic hints $\mathcal{H}_{b_i} \subseteq \mathcal{H}$, where

$$\mathcal{H} = \{h_{\text{area}}, h_{\text{type}}, h_{\text{cap}}, h_{\text{name}}\}. \quad (6)$$

As illustrated in Fig. 5, a trigger block may include hints such as $h_{\text{area}} = \text{living_room}$, $h_{\text{type}} = \text{motion_sensor}$, $h_{\text{cap}} = \text{motion_sensor.motion_detected}$, indicating that the event occurs in the living room and is associated with a motion sensor. Condition and action blocks capture corresponding semantic constraints in a similar structured form. The name hint h_{name} is used when the request refers to a specific device (e.g., “ceiling light”).

These hints represent symbolic constraints over spatial scope, device category, and capability identifiers. Importantly, this representation is intentionally underspecified: it captures semantic intent while leaving concrete device resolution to the grounding stage. This separation allows the parser to focus on intent extraction, improving robustness and flexibility.

3.4 Grounding and Subgraph Construction

This module translates abstract constraint representations into a concrete and executable action space derived from the KG. Given the structured parsing output, KGIoT first performs grounding to map semantic hints to candidate devices and capabilities, and then constructs a task-specific executable subgraph, as illustrated in Fig. 6.

3.4.1 Block-wise device grounding. Grounding is performed independently for each constraint block. For a given block b_i (Eq. 5), each semantic hint retrieves a set of candidate devices via the corresponding index mapping

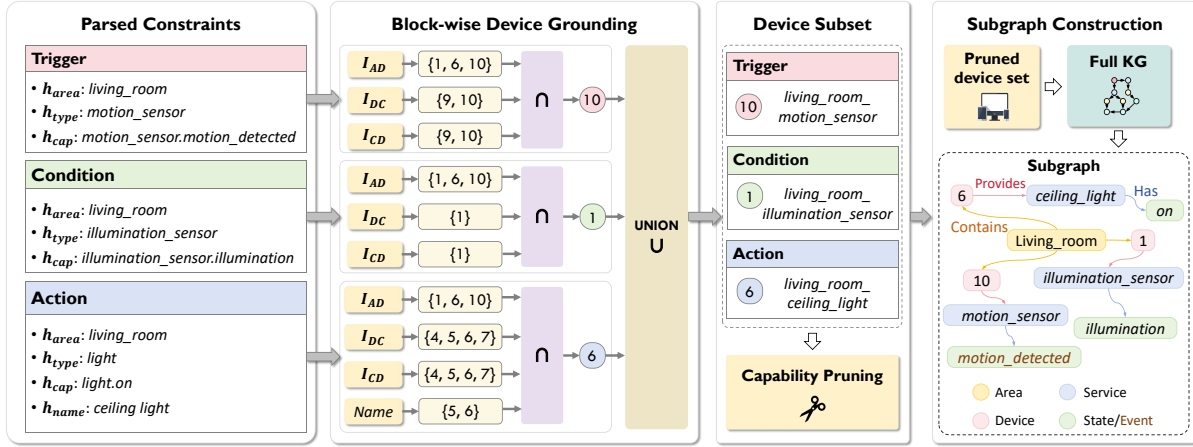


Fig. 6. Pipeline of device grounding, capability pruning, and subgraph construction. Constraint blocks are resolved via index-based intersection and aggregated through union to obtain the grounded device set, followed by capability pruning and subgraph construction.

(§ 3.2.3), and the final result is obtained by intersecting these sets:

$$\mathcal{D}_{b_i} = \bigcap_{h \in \mathcal{H}_{b_i}} I_h(h), \quad (7)$$

where $I_h(h)$ denotes the device set associated with hint h , and $\mathcal{H}_{b_i}$ is the set of hints for block b_i (Eq. 6). The resulting $\mathcal{D}_b$ contains devices that satisfy all constraints within the block.

For instance, as illustrated in Fig. 6, consider a trigger block with hints ($h_{area} = \text{living_room}$, $h_{type} = \text{motion_sensor}$, $h_{cap} = \text{motion_sensor.motion_detected}$), the *area-to-device* (I_{AD}) index retrieves all devices in the living room, while the *device-to-capability* (I_{DC}) and *capability-to-device* (I_{CD}) indices return devices matching the motion sensor and its sensing interface. Intersecting these sets yields the devices consistent with all constraints; in this case, a single device corresponding to the living room motion sensor.

After grounding each block independently, the final device set is obtained by aggregating all block-level results:

$$\mathcal{D}^* = \bigcup_{b_i \in \mathcal{B}} \mathcal{D}_{b_i}. \quad (8)$$

This aggregation combines devices involved in different roles to form the complete set required for TAP generation. In Fig. 6, this results in three devices in the living room: the motion sensor, the illumination sensor, and the main light. Each grounded device is assigned a functional role (trigger, condition, or action) based on the constraint blocks from which it is derived.

3.4.2 Capability Pruning. Given the grounded device set $\mathcal{D}^*$, retaining all capabilities would unnecessarily enlarge the planning context. Therefore, KGIoT prunes capabilities to retain only those relevant to the task. For each constraint block b_i (Eq. 5), capabilities matching the parsed capability hints h_{cap} (§ 3.3.3) are first retained. The resulting set is then expanded with prerequisite capabilities required by dependency and interlock relations in the KG (§ 3.2.5), ensuring executability while keeping the planning space compact.

3.4.3 Executable subgraph construction. After grounding and pruning, the remaining devices and capabilities define a set of candidate operations but do not yet form a structured action space, as the relational structure of

the original KG is no longer preserved. To recover this structure, KGIoT constructs a task-specific subgraph that restores the relations among these elements, enabling valid composition for subsequent TAP generation.

This subgraph is obtained by projecting and expanding the global KG around the grounded devices and retained capabilities, preserving only entities and relations relevant to the current request. As illustrated in Fig. 6, the subgraph is instantiated over the selected devices and their associated capabilities. For retained devices in the living room, capabilities such as *light.on*, *illumination_sensor.illumination*, and *motion_sensor.motion_detected* are preserved. In contrast to pruning, which operates within the grounded device set, this step restores relational structure from the full KG and introduces additional devices and capabilities as required by dependency and interlock relations (§ 3.2.5).

This reconstruction restores the compositional structure required for valid execution, enabling the constructed subgraph to serve as the basis for a finite, structured action space over grounded operations.

3.5 TAP Planning and Deployment

This module generates executable TAPs via constrained planning over the subgraph-defined action space and translates them into platform-specific automations for deployment.

Challenge C3. Even with a structured action space, LLMs may generate TAPs in a free-form manner, which may ignore the provided constraints and produce invalid or inconsistent outputs, such as incorrect device references or unsupported capability paths. For example, under complex requests involving multiple actions, the LLM may fail to adhere to the structured action space and instead produce free-form natural language outputs. This is because LLMs perform probabilistic token-level generation rather than explicit constraint satisfaction, and therefore do not explicitly enforce structured constraints during decoding. As task complexity increases (e.g., with multiple coordinated actions), the model may prioritize fluent natural language generation over strict adherence to the predefined action space.

Our Solution. We formulate TAP generation as constrained planning over a closed, subgraph-defined action space, where the LLM selects and composes only valid operations. This transforms generation from free-form text synthesis into structured decision-making over grounded primitives.

As illustrated in Fig. 7, the action space is instantiated as a set of indexed stems derived from the subgraph, each uniquely corresponding to a valid executable capability primitive with lightweight semantic annotations. For example, the living room illumination sensor, ceiling light, and motion sensor correspond to stems such as *[S0] 1.illumination_sensor.illumination*, *[S1] 6.light.on*, and *[S2] 10.motion_sensor.motion_detected*, with type-level cues such as *state* or *event*. These indexed stems form a closed set of valid operations, restricting the LLM to selecting and composing only admissible primitives, thereby reducing invalid capability combinations and improving the reliability of generated TAPs.

Given the user request, the LLM acts as a *TAP planner* that selects and composes these stems into a complete TAP following a predefined format. Operators and parameter values are inferred from the user request and capability semantics. In Fig. 7, for the request “when someone enters the living room and it is dark, turn on the ceiling light,” the planner maps each component to the corresponding stems in the action space. It selects *[S2]* as the trigger for “someone enters,” applies a condition on *[S0]* to capture “it is dark” (i.e., *illumination < 50 lux*, inferred by the LLM from the user request and capability semantics), and chooses *[S1]* as the action for “turn on the light,” forming a complete TAP.

In this way, invalid devices, unsupported interfaces, and inconsistent compositions are largely avoided by construction, as all primitives are derived from the subgraph together with their associated dependency and interlock relations. The selected stems are then mapped into executable TAP rules with only lightweight post-processing for structural consistency. This design fundamentally differs from existing LLM-driven TAP generation

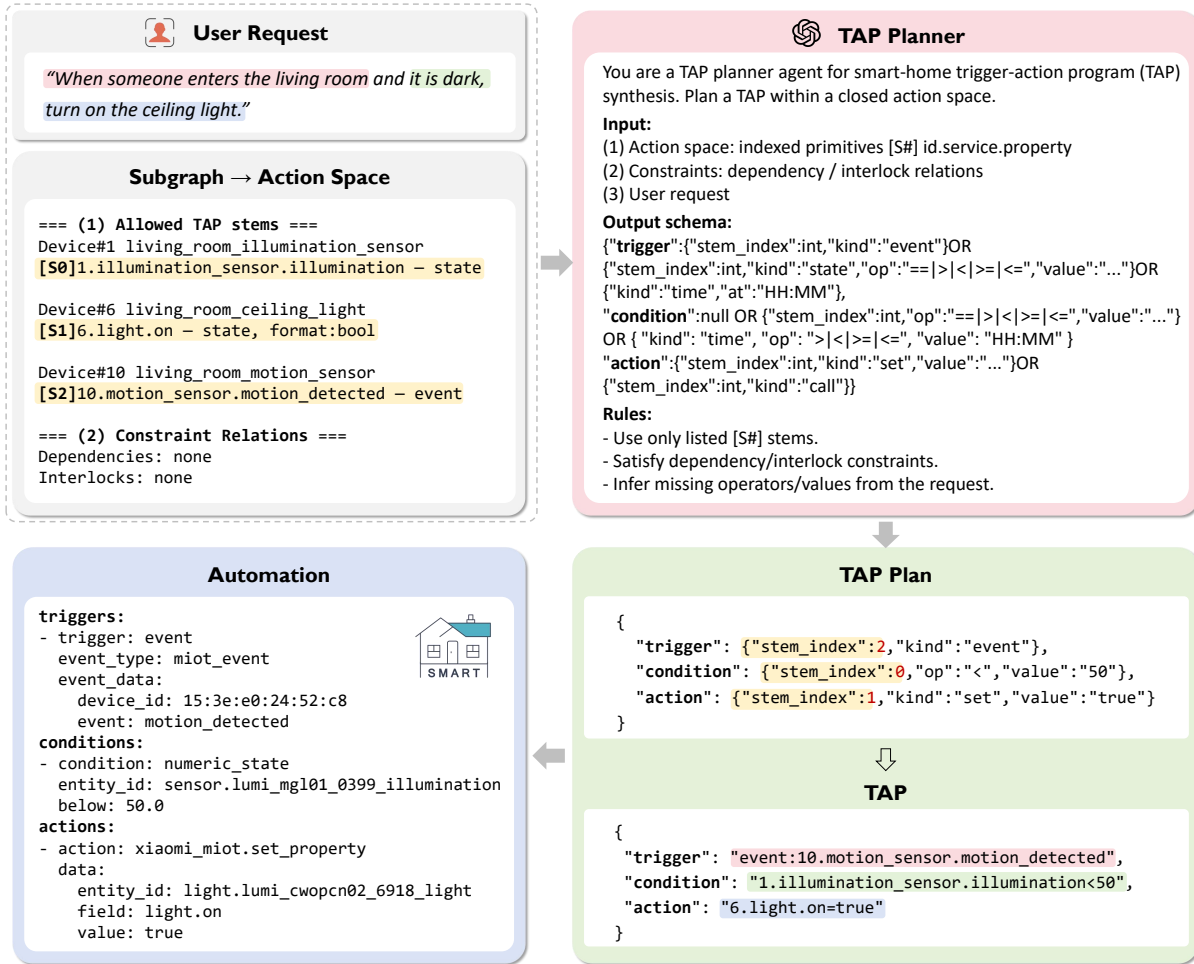


Fig. 7. TAP generation as constrained planning over a closed action space. A task-specific subgraph is transformed into a set of indexed stems that define the action space. The TAP planner then selects and composes these stems into a structured TAP, which is translated into an executable automation for deployment.

methods [19, 26, 71] that rely on post-hoc evaluation or iterative repair of generated outputs. Instead, our method promotes validity by construction through a constrained action space, reducing reliance on post-hoc validation.

This design choice is particularly important in smart-home automation, where an invalid TAP may originate from incorrect grounding, missing dependency constraints, or planning decisions. Although post-hoc validation can detect that a TAP is invalid, it cannot reliably identify which upstream step caused the error. Consequently, repairing the workflow often requires revisiting multiple upstream decisions rather than applying a local correction.

Table 3. Device configurations of the simulated smart-home environments.

Area	Devices
Home-S (10 devices)	
Entrance (2)	door sensor, light
Living room (3)	light, motion sensor, illumination sensor
Bedroom (3)	light, air conditioner, temperature/humidity sensor
Bathroom (2)	light, motion sensor
Home-M (20 devices)	
Entrance (2)	smart lock, light
Living room (5)	ceiling light, motion sensor, illumination sensor, floor lamp, TV
Bedroom (5)	ceiling light, air conditioner, temperature/humidity sensor, bedside lamp, window sensor
Bathroom (5)	light, motion sensor, bathroom heater, temperature/humidity sensor, door sensor
Kitchen (3)	light, range hood, cooker
Home-L (40 devices)	
Entrance (3)	smart lock, light, door sensor
Living room (8)	ceiling light, motion sensor, illumination sensor, floor lamp, TV, window sensor, air purifier, curtain
Bedroom (8+7)	Master bedroom: ceiling light, air conditioner, temperature/humidity sensor, bedside lamp, window sensor, curtain, humidifier, motion sensor; Second bedroom: ceiling light, air conditioner, temperature/humidity sensor, bedside lamp, window sensor, curtain, motion sensor
Bathroom (5)	light, motion sensor, bathroom heater, temperature/humidity sensor, door sensor
Kitchen (4)	light, range hood, cooker, motion sensor
Study (5)	ceiling light, desk lamp, motion sensor, air conditioner, temperature/humidity sensor

4 Evaluation

We evaluate KGIoT in a controlled smart-home simulation environment built on Home Assistant [6], enabling systematic comparison across baselines, home scales, and request types. We also conduct a real-world case study demonstrating the end-to-end workflow in a smart-home testbed with physical devices. Finally, we conduct a comprehensive user study comparing KGIoT with a baseline method, evaluating usability, perceived reliability, and overall user experience in interactive smart-home automation tasks.

4.1 Experimental Setup

4.1.1 Implementation Details. Local computing tasks are mainly performed on a server-grade workstation equipped with an Intel CPU, 96 GB RAM, and an NVIDIA GeForce RTX 5070 GPU. We use GPT-4o [34] as the base LLM with temperature set to 0. We also evaluate several recent LLMs (*e.g.*, DeepSeek [16] and Qwen [15]); detailed quantitative results and representative failure cases are reported in Appendix B. We use Home Assistant [6] as the smart-home platform and deploy it in Docker [3]. The current implementation is built on Home Assistant using MIoT [44] device specifications. Accordingly, all experiments evaluate this implementation. The core pipeline introduces minimal computational overhead and does not rely on GPU acceleration for local processing. Therefore, both the real-world case study and the user study are conducted on a consumer-grade laptop with modest hardware specifications.

4.1.2 Home Settings. We construct three simulated smart home environments with increasing scale: Home-S, Home-M, and Home-L, containing 10, 20, and 40 devices, respectively. Devices are instantiated based on MIoT specifications [44] using the `python-miio` library [1], and integrated into Home Assistant via the `hass-xiaomi-miot` integration [5]. The detailed device configurations of each environment are summarized in Table 3. Home-S represents a small apartment, Home-M a typical mid-sized home, and Home-L a larger multi-room environment. The three settings vary in both device density and environmental complexity, enabling evaluation under progressively more challenging conditions.

4.1.3 Evaluation Benchmark. Due to the lack of a publicly available benchmark for natural-language smart-home TAP generation, we construct an evaluation benchmark¹ consisting of 60 instances across three home settings. Each instance contains a natural-language automation request, an executable ground-truth TAP, and explicit/implicit annotations for fine-grained evaluation. Requests were initially drafted with LLM assistance (GPT-5-class) based on the corresponding home configurations and manually curated to ensure realistic user expressions, category coverage, and executable semantics. The LLM was used only for drafting candidate requests and was not involved in constructing ground-truth TAPs or annotations.

The benchmark contains 20 requests for each home setting, covering representative trigger-action patterns with increasing planning complexity as the home scale grows.

Specifically, the requests are grouped into four categories, with five requests per category:

- **Basic TAP (RC1):** single-trigger, single-action rules, representing the simplest automation form.
- **Conditional TAP (RC2):** rules that combine a trigger with additional conditions.
- **Multi-Action TAP (RC3):** rules where a single trigger activates multiple coordinated actions.
- **Constraint-Aware TAP (RC4):** rules that require implicit operational dependencies or environmental interlocks for correct and executable behavior, as detailed in Section 3.2.5.

The benchmark categories follow representative trigger-action structures observed in real-world TAP workloads [8, 57], ranging from simple trigger-action rules to conditional and multi-action workflows. Unlike existing TAP datasets, our benchmark additionally evaluates operational dependencies and environmental interlocks required for executable smart-home automation.

Ground-truth TAPs were manually constructed and verified for executability and semantic correctness. For RC4, required operational dependencies and environmental interlocks are intentionally omitted from natural-language requests and provided as explicit annotations, enabling separate evaluation of explicit-intent satisfaction and implicit-constraint completion. To avoid construction bias, requests were finalized before constructing ground-truth TAPs and annotations, and benchmark categories were defined based on general TAP structures rather than KGIoT’s internal knowledge graph organization.

4.1.4 Baselines. We compare KGIoT against five representative LLM-based smart-home automation approaches:

- **Vanilla.** A zero-shot LLM baseline with full-context prompting. Given a user request and the complete device schema of the target home, the model directly generates a TAP in a single interaction without explicit grounding, retrieval, or iterative refinement.
- **Sasha** [35]. A three-stage goal-oriented reasoning pipeline consisting of *Clarifying*, *Filtering*, and *Planning*. The system progressively verifies user goals, filters task-relevant devices, and generates TAPs from the filtered context. We preserve the original reasoning pipeline and adapt only the *Planning* output to the unified TAP representation.
- **ChatIoT** [26]. A multi-agent pipeline consisting of similarity-based context filtering, TAP generation, and post-generation evaluation. To avoid underestimating ChatIoT due to threshold sensitivity, we additionally

¹ <https://github.com/linyafu/KGIOT-artifact>

perform an offline threshold sensitivity analysis (see Appendix A), from which a single post-hoc global threshold configuration is selected for the main evaluation.

- **HomeGenii** [71]. A retrieval-augmented smart-home automation framework based on two-stage retrieval of Enhanced TAP rules and home context for TAP generation. We implement a task-adapted version that preserves its retrieval-augmented paradigm while adapting the output to the unified TAP representation.
- **AutoIoT** [19]. A reliability-oriented smart-home automation framework following a generate–verify–repair pipeline. A generated TAP is verified before deployment and iteratively repaired when verification fails. We implement a task-adapted version that preserves AutoIoT’s verification-guided refinement mechanism while omitting its original multi-rule conflict detection, because our benchmark evaluates one TAP per request rather than interactions among coexisting rules.
- **Knowledge-parity variants (+DK)**. To isolate the contribution of structured domain knowledge (Section 3.2.5), we additionally evaluate knowledge-parity variants of the baselines, denoted as **+DK**. These variants receive the same domain knowledge as plain-text prompts encoding operational dependencies and environmental interlocks. Because ChatIoT performs heuristic context compression before LLM generation, plain-text domain knowledge cannot be reliably preserved through the compression stage. We therefore evaluate the knowledge-parity setting using ChatIoT-NoComp+DK, and additionally report ChatIoT-NoComp to isolate the effect of disabling context compression.

All methods produce the same structured TAP representation and are evaluated using the same deployment pipeline and evaluation metrics. Unless otherwise specified, all baselines use their original configurations, and the corresponding **+DK** variants are reported only in the knowledge-parity evaluation. Detailed implementation of the adapted baselines, including prompts, hyperparameters, and reproduction details, is provided in Appendix A and released with the artifact repository.

4.1.5 Evaluation Metrics. KGIoT is evaluated along three complementary dimensions: (1) TAP generation quality, (2) grounding performance during context construction, and (3) system efficiency.

- (1) **TAP Generation Quality.** We report one primary end-to-end metric and three diagnostic metrics: executable correctness, platform validity, explicit-intent score, and implicit-constraint completion. Together, these metrics distinguish failures caused by invalid outputs, incorrect interpretation of explicit user intent, and missing implicit operational constraints.
 - **Executable Correctness.** This is the primary metric. A generated TAP is considered executable only if it matches an accepted ground-truth TAP under semantic atom matching, including both explicit user intent and annotated operational dependencies/interlocks. Let $e_i \in \{0, 1\}$ indicate whether request i is executable:

$$\text{Executable Correctness} = \frac{1}{N} \sum_{i=1}^N e_i \times 100\%.$$

- **Platform Validity.** This metric measures whether a generated TAP is syntactically and operationally admissible before semantic correctness is considered. A TAP is valid if it can be parsed, references available devices/capabilities with valid roles, and passes Home Assistant deployment validation. A platform-valid TAP may still express unintended behaviors or omit implicit constraints.
- **Explicit-intent Score.** This metric evaluates whether a generated TAP satisfies only the trigger, condition, and action constraints explicitly stated in the user request. Annotated implicit dependencies and interlocks do not affect this score, allowing explicit intent understanding to be evaluated independently.
- **Implicit-constraint Completion.** This metric measures whether a generated TAP recovers the operational dependencies and environmental interlocks required for executable behavior but omitted from

the request. For each request with annotated implicit constraints, we report the fraction of correctly recovered implicit atoms. Requests without implicit constraints are excluded.

- (2) **Grounding Performance.** Grounding quality is evaluated by comparing the generated grounding context with the annotated ground truth.

- **Device-level grounding** measures whether the correct set of relevant devices is retrieved. We report precision and recall of the retrieved device set compared to the ground truth.
- **Capability-level grounding** evaluates whether the correct capabilities are selected. We report precision and recall of the selected capability set compared to the ground truth.

For KGIoT, grounding is evaluated on the constructed subgraph after device selection and capability pruning. ChatIoT reports device-level grounding on the compressed context and capability-level grounding on the property list produced by its preprocessor agent. When context compression is disabled (ChatIoT-NoComp), only capability-level grounding is reported. Sasha and HomeGenii report only device-level grounding because they do not explicitly select capabilities, whereas Vanilla and AutoIoT perform no explicit grounding stage and are therefore excluded from grounding evaluation.

- (3) **Efficiency.** We report two metrics:

- **Token consumption:** the average prompt and completion tokens per request.
- **Latency:** the end-to-end time from receiving a request to generating the final TAP.

4.2 TAP Generation Performance

Table 4 summarizes the end-to-end and diagnostic results across the three home settings. For RC1–RC3, which contain no annotated implicit constraints, executable correctness is equivalent to the explicit-intent score. RC4 is separately evaluated using the explicit-intent score, implicit-constraint completion, and executable correctness. We next analyze TAP generation quality, grounding performance, efficiency, and failure patterns.

4.2.1 TAP Generation Quality. As shown in Table 4, KGIoT achieves the highest executable correctness across all three home settings, reaching 100.0%, 100.0%, and 85.0% on Home-S, Home-M, and Home-L, respectively. The performance advantage becomes more evident as home scale and request complexity increase: while most baselines perform competitively on RC1, their correctness degrades on RC3 and RC4 due to more challenging grounding and implicit constraint requirements.

For RC4, most baselines achieve higher explicit-intent scores than executable correctness, indicating that they can capture user-stated requirements but often miss implicit operational dependencies and environmental interlocks. For example, Vanilla correctly generates the requested brightness-setting action but omits the prerequisite power-on action, resulting in a semantically correct yet non-executable TAP.

Providing equivalent domain knowledge improves implicit-constraint completion for several baselines, confirming that missing domain knowledge contributes to RC4 failures. However, the remaining performance gap shows that knowledge alone is insufficient: KGIoT benefits from structured grounding, which explicitly maps reusable constraints to executable device capabilities through the KG.

4.2.2 Grounding Performance. Grounding quality provides further insight into the end-to-end performance differences. At the device level, KGIoT achieves the most balanced precision and recall across the three home settings, reaching 100.0%/100.0%, 93.75%/100.0%, and 92.09%/94.59% on Home-S, Home-M, and Home-L, respectively.

Sasha and HomeGenii achieve relatively high device recall by reducing the search space before TAP generation. However, both operate primarily at the device level and leave capability selection to the downstream generator. ChatIoT performs finer-grained capability selection and maintains high capability precision, but its recall remains below that of KGIoT. Its embedding-based compressor also exhibits an increasingly pronounced precision–recall

Table 4. Performance by request category and overall diagnostic metrics. Corr. denotes executable correctness, and Valid. denotes platform validity. P^D/R^D and P^C/R^C denote device-level and capability-level precision/recall, respectively. Lat. denotes latency. RC1, RC2, RC3, and RC4 correspond to Basic TAP, Conditional TAP, Multi-Action TAP, and Constraint-Aware TAP. Expl. and Impl. denote explicit-intent score and implicit-constraint completion, respectively. All correctness, validity, precision, and recall metrics are reported as percentages.

Method	Overall								Request Category					
	Corr.	Valid.	P^D	R^D	P^C	R^C	Tokens	Lat. (s)	RC1	RC2	RC3	RC4-Expl.	RC4-Impl.	RC4-Corr.
Home-S														
Vanilla	70.0	95.0	–	–	–	–	4835	1.25	100.0	80.0	100.0	100.0	0.0	0.0
Vanilla+DK	70.0	95.0	–	–	–	–	5139	1.38	80.0	80.0	60.0	60.0	100.0	60.0
Sasha [35]	75.0	100.0	98.34	100.0	–	–	4224	2.83	100.0	100.0	100.0	100.0	0.0	0.0
Sasha+DK	85.0	100.0	98.34	100.0	–	–	4521	2.72	80.0	100.0	100.0	60.0	60.0	60.0
ChatIoT [26]	45.0	70.0	49.12	85.84	100.0	79.59	2819	3.52	60.0	40.0	80.0	80.0	0.0	0.0
ChatIoT-NoComp	70.0	95.0	–	–	100.0	92.92	4543	3.32	100.0	80.0	100.0	100.0	0.0	0.0
ChatIoT-NoComp+DK	80.0	100.0	–	–	89.92	100.0	5607	3.31	100.0	100.0	60.0	60.0	100.0	60.0
HomeGenii [71]	70.0	100.0	41.0	100.0	–	–	3817	2.39	80.0	100.0	100.0	100.0	0.0	0.0
HomeGenii+DK	75.0	100.0	41.0	100.0	–	–	4128	2.24	80.0	60.0	80.0	80.0	100.0	80.0
AutoIoT	70.0	80.0	–	–	–	–	4302	1.32	40.0	100.0	80.0	80.0	60.0	60.0
AutoIoT+DK	60.0	95.0	–	–	–	–	5298	1.48	80.0	60.0	60.0	60.0	80.0	40.0
KGIoT	100.0	100.0	100.0	100.0	94.67	100.0	3656	2.58	100.0	100.0	100.0	100.0	100.0	100.0
Home-M														
Vanilla	50.0	85.0	–	–	–	–	12304	1.51	100.0	40.0	60.0	80.0	0.0	0.0
Vanilla+DK	75.0	85.0	–	–	–	–	12607	1.45	100.0	60.0	80.0	60.0	100.0	60.0
Sasha [35]	30.0	85.0	81.68	90.0	–	–	7103	2.71	80.0	20.0	20.0	80.0	0.0	0.0
Sasha+DK	40.0	80.0	81.68	90.0	–	–	7371	2.28	80.0	40.0	20.0	80.0	50.0	20.0
ChatIoT [26]	35.0	75.0	30.47	95.84	92.50	81.01	3944	3.34	80.0	20.0	40.0	80.0	0.0	0.0
ChatIoT-NoComp	40.0	95.0	–	–	97.50	92.01	9385	3.26	80.0	40.0	40.0	80.0	0.0	0.0
ChatIoT-NoComp+DK	60.0	85.0	–	–	95.0	100.0	10396	3.37	60.0	60.0	60.0	80.0	90.0	60.0
HomeGenii [71]	45.0	90.0	44.34	100.0	–	–	4464	2.41	80.0	60.0	40.0	100.0	0.0	0.0
HomeGenii+DK	60.0	80.0	44.34	100.0	–	–	4774	1.97	60.0	40.0	80.0	60.0	80.0	60.0
AutoIoT	30.0	90.0	–	–	–	–	13175	1.61	60.0	40.0	20.0	80.0	10.0	0.0
AutoIoT+DK	45.0	85.0	–	–	–	–	12410	1.69	40.0	40.0	60.0	40.0	80.0	40.0
KGIoT	100.0	100.0	93.75	100.0	85.75	100.0	5232	3.30	100.0	100.0	100.0	100.0	100.0	100.0
Home-L														
Vanilla	40.0	65.0	–	–	–	–	50861	3.02	100.0	60.0	0.0	80.0	0.0	0.0
Vanilla+DK	45.0	65.0	–	–	–	–	51165	2.72	80.0	60.0	0.0	40.0	80.0	40.0
Sasha [35]	45.0	80.0	83.34	85.0	–	–	18899	3.19	100.0	80.0	0	80.0	0.0	0.0
Sasha+DK	60.0	90.0	88.34	90.0	–	–	20051	2.60	100.0	80.0	40.0	40.0	20.0	20.0
ChatIoT [26]	35.0	75.0	19.42	86.00	85.22	73.50	8145	4.37	80.0	60.0	0.0	60.0	0.0	0.0
ChatIoT-NoComp	45.0	95.0	–	–	98.34	89.05	35615	4.21	100.0	80.0	0.0	60.0	0.0	0.0
ChatIoT-NoComp+DK	55.0	90.0	–	–	91.88	90.22	36671	4.85	100.0	60.0	20.0	40.0	40.0	40.0
HomeGenii [71]	30.0	60.0	42.0	83.25	–	–	8333	2.62	80.0	40.0	0.0	60.0	0.0	0.0
HomeGenii+DK	40.0	65.0	42.0	83.25	–	–	8644	2.14	60.0	60.0	0.0	40.0	80.0	40.0
AutoIoT	40.0	60.0	–	–	–	–	50243	2.78	80.0	60.0	20.0	60.0	10.0	0.0
AutoIoT+DK	40.0	80.0	–	–	–	–	65073	4.12	60.0	80.0	20.0	0.0	60.0	0.0
KGIoT	85.0	95.0	92.09	94.59	83.74	93.96	10750	3.32	100.0	80.0	60.0	100.0	100.0	100.0

trade-off as the home scale grows: device recall remains high, while precision drops from 49.12% on Home-S to 19.42% on Home-L.

In contrast, KGIoT combines task-aware semantic parsing with KG-guided grounding to construct an executable subgraph containing relevant devices, capabilities, and constraints. This design avoids both the context loss

Table 5. Primary error categories of failed TAP generations across Home-S, Home-M, and Home-L. Each failed request is assigned to a single primary error category according to the earliest error preventing executable deployment. For the knowledge-parity variants, we additionally report failures caused by over-injected domain knowledge.

Error Category	Vanilla	Sasha	ChatIoT	HomeGenii	AutoIoT	KGIoT
Missing implicit constraints	15	13	13	13	9	0
Grounding error	1	0	10	0	0	0
Incorrect constraint application	3	7	3	5	5	0
Incomplete multi-action planning	0	1	2	1	0	2
Invalid TAP	9	7	6	10	14	1
Over-injected domain knowledge (+DK)	8	1	9	8	11	–

introduced by aggressive semantic filtering and the capability ambiguity of device-only retrieval, providing a more reliable basis for downstream TAP planning.

4.2.3 Efficiency. Table 4 compares the efficiency of different methods. As home scale increases, full-context prompting becomes increasingly costly: Vanilla requires 50,861 tokens on Home-L compared with 4,835 tokens on Home-S. In contrast, KGIoT increases from 3,656 to 10,750 tokens by constructing task-relevant executable subgraphs, using approximately $4.7\times$ fewer tokens than Vanilla on Home-L while achieving higher executable correctness (85.0% vs. 40.0%).

Existing context reduction strategies achieve efficiency at different costs. ChatIoT obtains the lowest token consumption (2,819–8,145 tokens) through semantic compression, but the reduced context coverage limits TAP generation quality. Sasha and HomeGenii reduce prompt size through device-level filtering or retrieval, and AutoIoT incurs substantially higher token usage in larger homes due to its generate–verify–repair iterations.

In terms of latency, KGIoT requires 2.58–3.32 s per request. Although single-pass methods such as Vanilla are faster, KGIoT remains comparable to other multi-stage approaches. Overall, KGIoT achieves a better efficiency–performance trade-off by controlling context growth through structured grounding rather than aggressive context removal.

4.2.4 Failure Analysis. To better understand the remaining performance gap, Table 5 reports the primary error category for each failed TAP generation across the three home settings. Each failure is assigned to the earliest error that prevents executable deployment.

Missing implicit constraints are the dominant failure mode for all no-DK baselines, accounting for 9–15 failures per method. These systems often capture the explicitly requested automation but omit prerequisite actions or environmental interlocks required for executable behavior.

Providing plain-text domain knowledge reduces missing-constraint errors but introduces incorrect rule application, such as applying interlocks to irrelevant requests or grounding generic rules to incorrect devices. This indicates that domain knowledge alone is insufficient without reliable grounding. Other failures reflect differences in pipeline design: ChatIoT is mainly affected by grounding errors from semantic filtering, while AutoIoT produces more invalid TAPs due to imperfect post-hoc repair. KGIoT eliminates grounding errors, missing implicit constraints, and incorrect constraint applications; its remaining failures consist of two incomplete multi-action plans and one parser failure in Home-L.

Representative KGIoT failure cases are provided in Appendix B.2, together with structured-output and capability-selection errors observed under alternative planner backbones.

Table 6. Ablation results (Home-L). Corr. denotes executable correctness, and Valid. denotes platform validity. Expl. and Impl. denote explicit-intent score and implicit-constraint completion, respectively. P^D/R^D and P^C/R^C denote device-level and capability-level precision/recall, respectively. All correctness, validity, precision, and recall metrics are reported as percentages.

Method	Corr.	Valid.	Expl. (%)	Impl. (%)	$P^D/$	R^D	P^C	R^C	Token	Latency (s)
w/o Intent-Guided Grounding	95.0	100.0	95.0	100.0	88.84	100.0	86.20	98.34	49819	5.76
w/o Subgraph Construction	40.0	95.0	60.0	0.0	91.50	90.01	83.72	89.00	10170	3.66
w/o Closed Action Space	65.0	80.0	65.0	100.0	93.67	93.34	83.01	91.67	10405	3.50
w/o Grounding + Subgraph	65.0	100.0	90.0	0.0	89.84	100.0	87.86	98.34	49362	5.26
Full KGIoT	85.0	95.0	85.0	100.0	83.74	93.96	83.74	93.96	10750	3.32

4.3 Ablation Study

To examine the contribution of individual components in KGIoT, we conduct ablation studies on Home-L. We evaluate three single-component variants that remove intent-guided grounding, executable subgraph construction, or the closed action space, respectively. Each ablation tests a specific design hypothesis: whether intent-guided grounding primarily improves efficiency by reducing the candidate space, whether executable subgraph construction contributes relational organization and propagation of dependency/interlock knowledge, and whether the closed action space improves the validity and executability of generated platform operations. We further include a combined variant that jointly removes intent-guided grounding and executable subgraph construction, as these two upstream modules are functionally coupled. All variants retain the remaining components whenever they can operate without the removed module, and no additional dependency or interlock knowledge is supplied.

4.3.1 w/o Intent-Guided Grounding. To evaluate the role of intent-guided grounding, which combines constraint parsing (§ 3.3) with block-wise device grounding (§ 3.4.1), we remove both components and instead ask the LLM to select relevant capabilities directly from the full KG. The selected capabilities are then passed to the same executable subgraph construction and closed action-space planner used by the full KGIoT pipeline.

As shown in Table 6, this variant achieves 95.0% executable correctness while maintaining 100.0% platform validity and implicit-constraint completion. This indicates that, in the evaluated Home-L setting, a sufficiently strong LLM can directly identify relevant capabilities from the full KG without reducing planning quality. However, token consumption increases from 10,750 to 49,819 (4.63×) and latency rises from 3.32 s to 5.76 s. These results suggest that intent-guided grounding primarily improves efficiency by reducing the candidate space while preserving high planning performance.

4.3.2 w/o Subgraph Construction. To evaluate the role of executable subgraph construction (§ 3.4.3), we retain intent-guided grounding and the closed action space but remove the stage that organizes grounded capabilities into a task-specific relational structure. The planner instead receives a flat list of indexed capability stems, without dependency or interlock information.

Removing executable subgraph construction reduces executable correctness from 85.0% to 40.0%, while the explicit-intent score decreases from 85.0% to 60.0% and implicit-constraint completion drops from 100.0% to 0.0%. Although platform validity remains high (95.0%), the planner no longer receives explicit device–capability relations or dependency/interlock knowledge, making it difficult to compose correct multi-action workflows. These results indicate that executable subgraph construction primarily contributes relational organization and propagation of constraint knowledge.

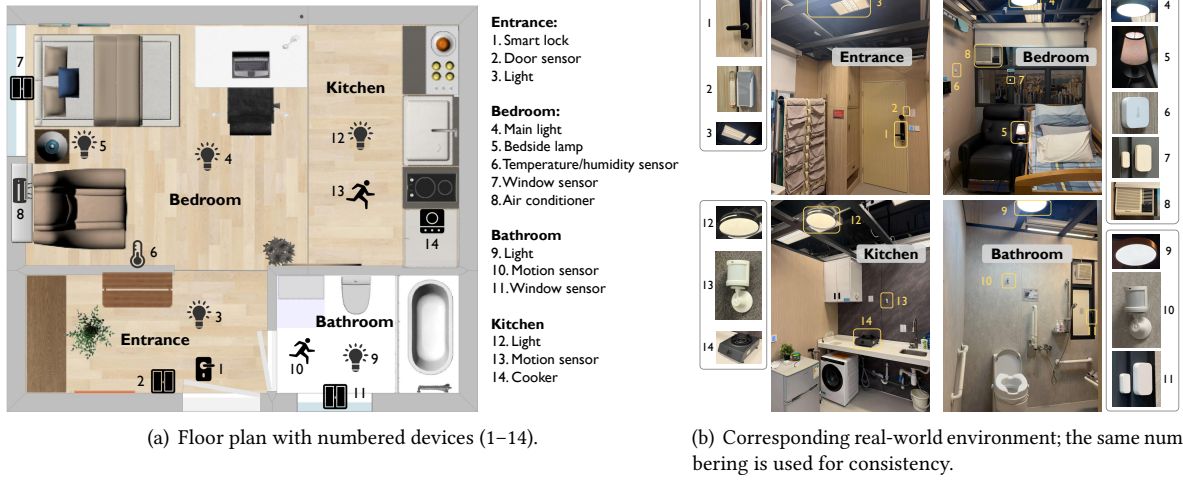


Fig. 8. Real-world smart-home setup.

4.3.3 w/o Closed Action Space. To evaluate the role of constrained action generation, we retain intent-guided grounding and the executable subgraph but remove the closed action space. The planner therefore receives the same grounded relational context but is no longer restricted to selecting valid indexed capability stems.

Removing the closed action space reduces executable correctness from 85.0% to 65.0% and platform validity from 95.0% to 80.0%, while implicit-constraint completion remains unchanged. Although the planner still receives complete dependency/interlock knowledge, it more frequently generates malformed operations or unsupported device–capability combinations. These results show that the closed action space mainly improves structural validity and executable command generation.

4.3.4 w/o Grounding + Subgraph. To study the interaction between the two upstream modules, we jointly remove intent-guided grounding and executable subgraph construction. The LLM directly selects capabilities from the full KG, which are passed to the planner as a flat list of indexed capability stems without relational organization or dependency/interlock knowledge.

This variant achieves 65.0% executable correctness, higher than removing only executable subgraph construction (40.0%) but lower than the full pipeline (85.0%). Platform validity remains at 100.0%, and the explicit-intent score reaches 90.0%, indicating that direct LLM-based capability selection provides a more focused candidate set for explicit planning. However, implicit-constraint completion remains 0.0%, confirming that direct capability selection cannot replace executable subgraph construction for propagating dependency and interlock knowledge. Moreover, token consumption (49,362) remains comparable to the individual grounding ablation, showing that this strategy substantially increases computational cost.

Overall, the ablation results show that the three components play complementary rather than interchangeable roles. Intent-guided grounding primarily reduces the candidate space and improves efficiency; executable subgraph construction organizes grounded capabilities and propagates dependency/interlock knowledge; and the closed action space constrains planning to executable platform operations. The combined ablation further confirms that closed action-space encoding alone cannot compensate for the absence of executable subgraph construction.

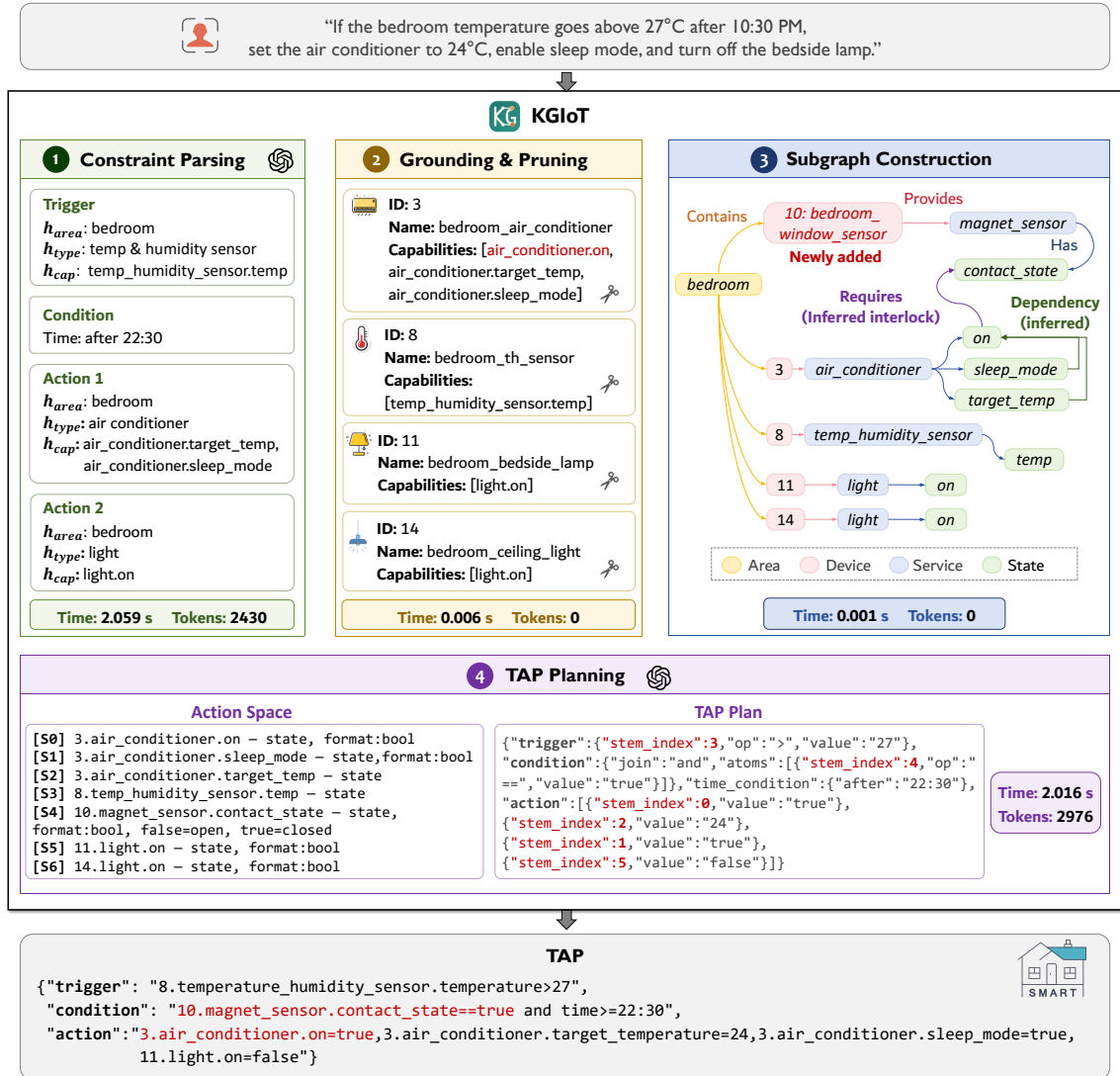


Fig. 9. End-to-end workflow of KGIoT in the real-world case study. Given a user request, KGIoT first extracts structured constraints, grounds them to concrete devices and capabilities in the KG, constructs a task-specific subgraph, and finally generates the TAP rule. Notably, the subgraph enables automatic completion of implicit operational constraints, including the window-closed interlock for air-conditioning and the power-on dependency before setting the AC mode and temperature.

4.4 Real-World Case Study

To demonstrate the practical applicability of KGIoT, we conduct a real-world case study in a physical smart-home environment integrated with Home Assistant. We present both the real-world setup and an end-to-end example to illustrate the complete workflow of TAP generation and deployment. The case study uses the same LLM setting as the main experiments.

4.4.1 Home Layout. Fig. 8 presents the real-world smart-home setup. The floor plan with device annotations is shown in Fig. 8 (a), while the corresponding real-world environment is shown in Fig. 8 (b). Both views share a consistent device indexing (1–14). The setup covers four functional areas, namely the entrance, bedroom, bathroom, and kitchen, each equipped with multiple devices integrated into the Home Assistant platform. Upon system initialization, the KG is constructed once, taking 0.007 seconds.

4.4.2 End-to-end TAP Generation Example. To illustrate the operation of KGIoT in practice, we present one representative end-to-end example with the request: “If the bedroom temperature goes above 27°C after 10:30 PM, set the air conditioner to 24°C, enable sleep mode, and turn off the bedside lamp.” Fig. 9 illustrates the complete workflow, including constraint parsing, device and capability grounding, subgraph construction, and TAP planning.

Constraint Parsing. Given the user request and a lightweight KG summary, the system first extracts constraints through a parsing agent. As shown in Fig. 9, it identifies the trigger as the bedroom temperature exceeding 27°C, the time condition as after 10:30 PM, and multiple actions including setting the air conditioner to 24°C, enabling sleep mode, and turning off the bedside lamp. This stage takes 2.059 seconds and consumes 2430 tokens.

Grounding and Pruning. The parsed constraints are then grounded to concrete devices and capabilities in the KG. Specifically, the system identifies the bedroom temperature sensor, air conditioner, and two lights, along with preserved capabilities after pruning. This step is completed in 0.006 seconds without any token consumption.

Subgraph Construction. Based on the grounded devices and capabilities, KGIoT constructs a task-specific executable subgraph from the full KG. In particular, it automatically introduces the window-closed condition as an interlock for air-conditioning and the power-on requirement as a dependency before setting the AC mode and temperature. This step is lightweight and takes 0.001 seconds without token consumption.

TAP Planning. Finally, KGIoT performs constrained TAP planning over the constructed subgraph. The system selects valid capability stems from the closed action space and composes them into a structured TAP rule. In the intermediate TAP plan, temporal constraints are represented separately as the `time_condition` field for planning clarity, and are subsequently merged into the final executable TAP condition. This stage takes 2.016 seconds and consumes 2976 tokens.

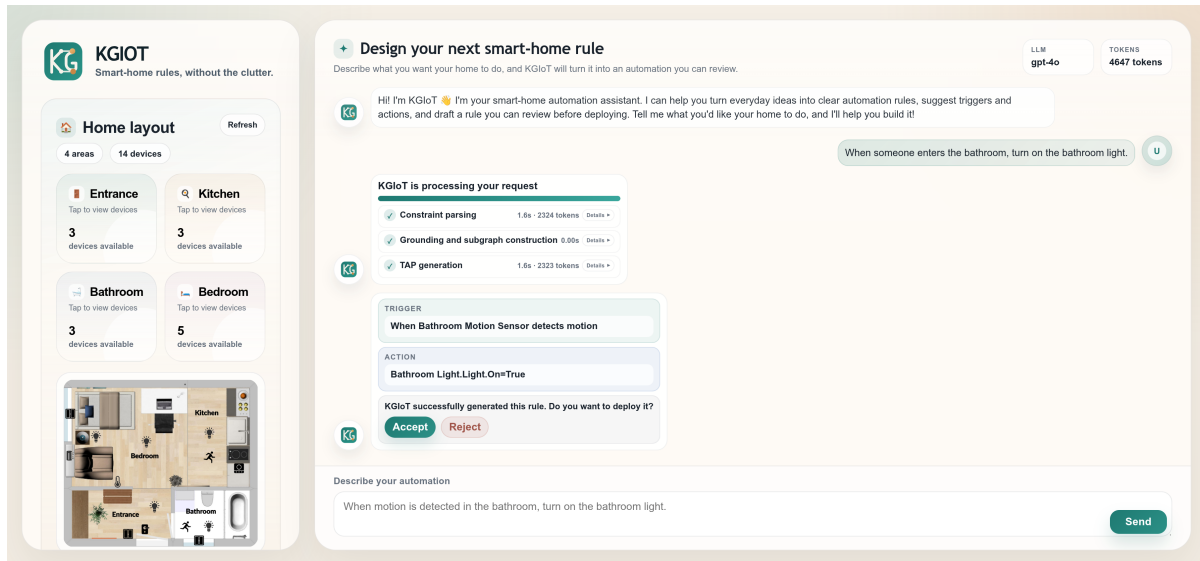
Overall, the entire pipeline is completed in 4.082 seconds with a total token consumption of 5406. The generated TAP accurately captures the trigger, time condition, and multiple actions, while automatically incorporating implicit constraints inferred from the KG. This TAP is then translated into an automation and successfully deployed to the Home Assistant platform.

4.5 User Study

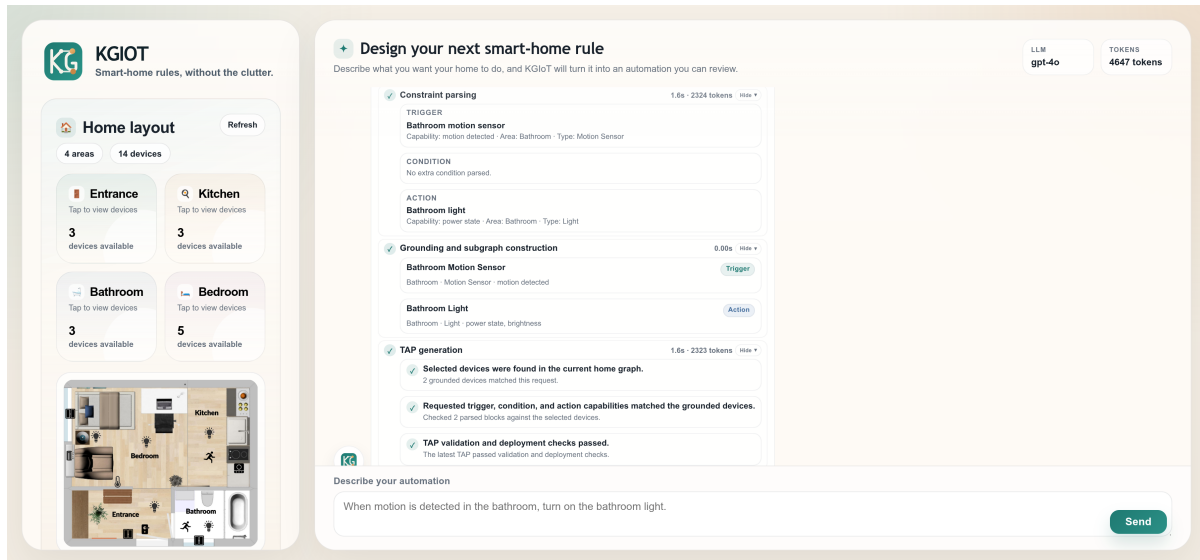
We conduct a user study using the real-world Home Assistant deployment described in Section 4.4 to evaluate the usability, reliability, and user trust of KGIoT compared with a vanilla LLM baseline. We choose Vanilla as the baseline because it provides a clean reference for unconstrained LLM-based automation without introducing additional system components that may confound user experience evaluation.

4.5.1 User Interface. Fig. 10 presents the interface used in the user study. Participants can submit natural-language requests, review generated automations, and inspect intermediate results including constraint parsing, device grounding, and validation outcomes. The interface also provides the home layout and available devices to improve transparency and help users understand how the system derives automation rules.

4.5.2 Participants. We recruited 15 participants (ten male and five female) aged between 23 and 30, all holding a Bachelor’s degree or higher, with diverse academic backgrounds. Fig. 11 shows participants’ prior experience with smart-home devices and programming or automation tools. While most participants had limited experience



(a) Users can enter natural-language requests, review the generated automations, and choose to accept or reject them.



(b) The system also provides detailed intermediate results (e.g., constraint parsing, device grounding, and validation), allowing users to inspect how the automation is generated.

Fig. 10. User interface of KGIoT used in the user study.

with automation programming, they were generally familiar with smart-home devices, representing the target user population of end users seeking easier automation authoring.

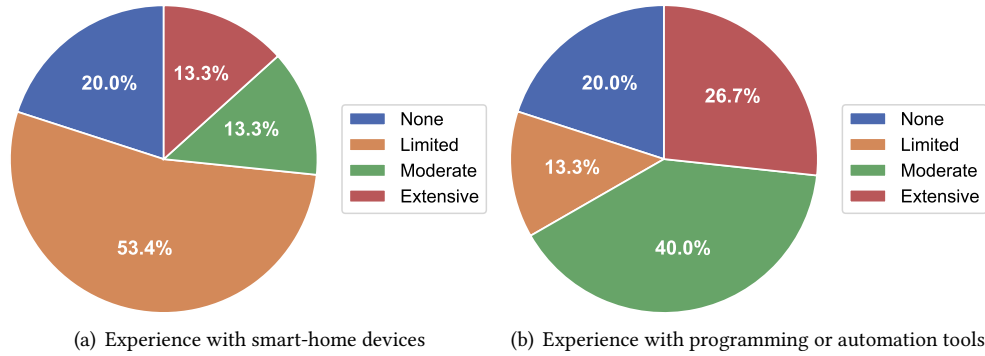


Fig. 11. Distribution of participants' prior experience levels ($N=15$).

4.5.3 Study Design. We compare KGIoT with Vanilla, a zero-shot LLM baseline that directly generates TAPs from user requests given the full home context (the same as in § 4.1.4).

To enable controlled comparison, target automation scenarios were predefined, while participants were free to express each scenario in their own language. This design ensures differences in user ratings primarily reflect the systems' handling of requests rather than variations in task difficulty or automation goals. Participants were divided into two groups and assigned one of two task sets, each containing four automation scenarios covering basic trigger-action rules, conditional rules, multi-action workflows, and constraint-aware cases.

The study was conducted in a meeting room. Participants interacted with the deployed smart-home system through the proposed user interface, which presented the home layout and available devices corresponding to the real-world Home Assistant deployment. Each study session began with a brief tutorial introducing the task setting and system interface. Participants first completed a practice task to familiarize themselves with the interaction.

Each participant interacted with both systems under the same home context. The presentation order was alternated across participants to reduce ordering bias. Participants reviewed the generated automation and decided whether to accept or reject the result. After completing all tasks, participants rated both systems using a 5-point Likert questionnaire. Each session lasted approximately 20–25 minutes.

4.5.4 Questionnaire Design. Following prior work [71], we designed a post-study questionnaire, as shown in Table 7, to evaluate user experience across five dimensions: *usability*, *intent alignment*, *grounding and interpretability*, *trust*, and *adoption*. Each dimension is assessed using two statements rated on a 5-point Likert scale (1: strongly disagree, 5: strongly agree).

The questionnaire captures both system usability (e.g., ease of use and clarity of presented automation) and perceived system quality, including how well the generated automation aligns with user intent, whether system decisions are understandable, and whether users trust and are willing to adopt the system in real-world scenarios.

4.5.5 Result Analysis. Table 7 and Fig. 12 summarize the questionnaire results. For each item, we report the mean Likert score for both systems and perform a Wilcoxon signed-rank test [28].

As shown in Fig. 12, KGIoT achieves higher ratings than the baseline across all ten questions. The improvements are statistically significant for all items ($p < 0.01$), with several questions reaching stronger significance levels ($p < 0.001$), indicating a clear advantage of KGIoT in terms of user experience.

In particular, notable improvements are observed in intent alignment (Q3–Q4), grounding and interpretability (Q5–Q6), and trust (Q7–Q8). These results suggest that capability-aware grounding and structured generation help

Table 7. User study questionnaire and response distributions for KGIoT and Vanilla. Each vertical set of colored squares corresponds to the responses of one participant across all questions.

		Ratings: (strongly disagree) 1 2 3 4 5 (strongly agree)									
Category	Questions	KGIoT					Vanilla				
Usability	Q1: I found the system easy to use.	■	■	■	■	■	■	■	■	■	■
	Q2: The system presented the generated automation in a clear and understandable way.	■	■	■	■	■	■	■	■	■	■
Intent alignment	Q3: The generated automation matched what I wanted.	■	■	■	■	■	■	■	■	■	■
	Q4: The system correctly captured the trigger, condition, and action in my request.	■	■	■	■	■	■	■	■	■	■
Grounding & interpretability	Q5: The system selected appropriate devices and functions for the task.	■	■	■	■	■	■	■	■	■	■
	Q6: I could understand why the system generated this automation.	■	■	■	■	■	■	■	■	■	■
Trust	Q7: I feel comfortable deploying this automation in a real home.	■	■	■	■	■	■	■	■	■	■
	Q8: The generated automation appears reliable and free of obvious mistakes.	■	■	■	■	■	■	■	■	■	■
Adoption	Q9: I would be willing to use this system in real smart-home scenarios.	■	■	■	■	■	■	■	■	■	■
	Q10: The benefits of this system outweigh the effort required to use it.	■	■	■	■	■	■	■	■	■	■

the system better capture user intent, select appropriate device capabilities, and provide more understandable and reliable automation outcomes. Furthermore, KGIoT also shows strong gains in usability (Q1–Q2) and adoption (Q9–Q10), indicating that users find the system easier to use and are more willing to deploy it in real-world scenarios.

In addition to subjective questionnaire responses, we also evaluate objective performance metrics, including the acceptance rate and the validity rate. The acceptance rate is defined as the proportion of generated automations that participants chose to accept. The validity rate measures whether the generated TAP is syntactically correct and executable.

KGIoT achieves a substantially higher acceptance rate (94.64%) compared to the Vanilla baseline (50.00%), indicating that users are far more satisfied with the generated automations. In terms of validity, KGIoT consistently produces correct outputs, achieving a validity rate of 100.00%, while the baseline reaches only 76.79%.

These results suggest that KGIoT not only improves user-perceived quality but also supports reliable and executable automation generation in practice.

5 Further Discussions

This section discusses the practical scope of KGIoT beyond the experimental results and outlines several limitations that motivate future research.

5.0.1 Dynamic Environment Modeling. The current KG primarily models static deployment knowledge, including device capabilities, area assignments, operational dependencies, and environmental interlocks. Runtime states, such as changing device conditions, occupancy, and workflow execution status, are not explicitly represented or synchronized. Extending the capability-centric KG with runtime state representations would enable state-aware planning and replanning in dynamic home environments, providing a promising direction for future work.

5.0.2 Scope of Supported Automation. The current KGIoT framework focuses on automations that can be grounded into executable device capabilities. However, several practical scenarios remain unsupported. First, many real-world TAP rules involve heterogeneous cloud services such as weather APIs, calendars, email services, and push notifications [8], which are not naturally represented by capability-oriented device abstractions and are therefore not currently modeled by the KG. Second, while KGIoT supports clock-time conditions (e.g., “after 9 PM”), it does not explicitly model richer temporal operators such as delayed execution (“after five minutes”) or persistence conditions (“if the condition lasts for five minutes”), which require duration-aware capability

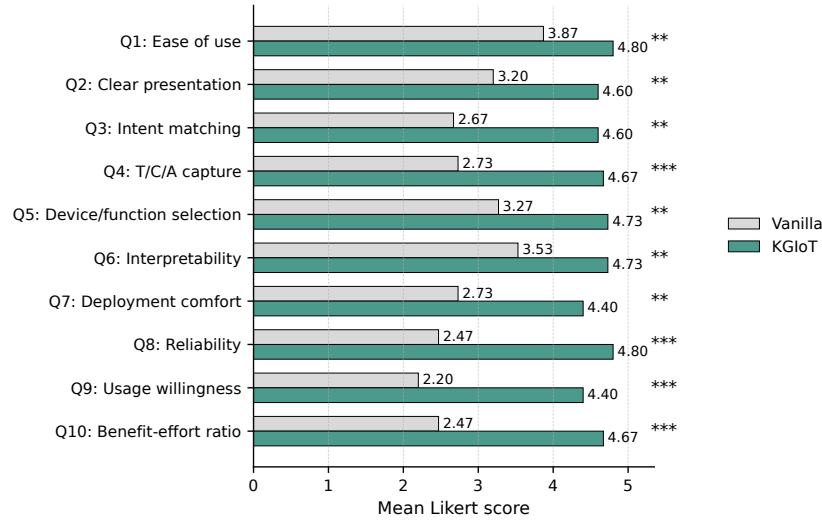


Fig. 12. Questionnaire results comparing KGIoT and the Vanilla baseline. Bars show mean Likert scores. Statistical significance is computed using the Wilcoxon signed-rank test (* $p < 0.05$, ** $p < 0.01$, *** $p < 0.001$).

representations and temporal reasoning. Future work will investigate richer capability abstractions capable of representing external services and advanced temporal semantics within the same planning framework.

5.0.3 Knowledge Acquisition and Personalization. The current implementation relies on manually maintained reusable capability-level templates to represent operational dependencies and environmental interlocks. In addition, qualitative expressions such as “dark” are interpreted by the LLM planner based on capability semantics, but the resulting thresholds are neither explicitly personalized nor learned from user behavior. Future work will investigate automatic acquisition of constraint knowledge from platform documentation or execution traces, together with personalized threshold learning from long-term user interactions.

5.0.4 Platform Generality. The current implementation of KGIoT is developed for Home Assistant using MIoT device specifications [44]. Although the proposed capability-centric KG construction and constrained planning pipeline are conceptually independent of a particular automation syntax, supporting additional smart-home ecosystems (e.g., Matter-enabled [9] platforms or Apple HomeKit [2]) requires access to structured metadata describing device capabilities, entities, and their relationships, together with platform-specific metadata adapters and automation mappings. For platforms that do not expose such metadata, additional metadata extraction mechanisms would be required. Accordingly, the experiments in this paper evaluate only the Home Assistant implementation, while cross-platform deployment remains future work.

6 Conclusion

This paper presents KGIoT for reliable generation of executable TAPs from natural-language requests in smart homes. It represents device capabilities and compositional constraints in a capability-centric KG and reformulates TAP generation as constrained planning over the resulting closed, executable action space. Extensive evaluations show that KGIoT achieves substantial gains in executable correctness, outperforming the strongest baseline by up to 25.0 percentage points while reducing token consumption by over 4.7×, and consistently produces executable TAPs in real-world deployment. In addition, our comprehensive user study demonstrates that KGIoT improves

usability, interpretability, and user trust compared to baseline approaches. Overall, these results highlight the importance of capability-centric representations, structured grounding, and explicit action spaces for bridging natural language and dependable smart-home systems.

Acknowledgments

We thank anonymous reviewers for their constructive feedback and valuable suggestions, which helped improve this work. This work is supported by Hong Kong GRF 15216325 and 15211924, and Hong Kong CRF C5085-25G.

References

- [1] 2024. python-miio. <https://github.com/rytilahti/python-miio>.
- [2] 2026. Apple HomeKit. <https://www.apple.com/sg/home-app/>.
- [3] 2026. Docker Documentation. <https://docs.docker.com/>.
- [4] 2026. Google Home. <https://home.google.com/welcome/>.
- [5] 2026. hass-xiaomi-miot: Xiaomi MIoT Integration for Home Assistant. <https://github.com/al-one/hass-xiaomi-miot>.
- [6] 2026. Home Assistant: Open source home automation. <https://www.home-assistant.io/>.
- [7] 2026. IFTTT. <https://ifttt.com/>.
- [8] 2026. IFTTT Recipes. <https://www.kaggle.com/datasets/hrs2kr/ifttt-recipes>.
- [9] 2026. Matter Smart Home Standard. <https://csa-iot.org/all-solutions/matter/>.
- [10] 2026. Samsung SmartThings. <https://www.samsung.com/us/smartthings/>.
- [11] 2026. Zapier. <https://zapier.com/>.
- [12] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. GPT-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).
- [13] Riku Arakawa, Jill Fain Lehman, and Mayank Goel. 2024. PrISM-Q&A: Step-aware voice assistant on a smartwatch enabled by multimodal procedure tracking and large language models. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 8, 4 (2024), 1–26.
- [14] Luca Arrotta, Gabriele Civitarese, and Claudio Bettini. 2022. Dexar: Deep explainable sensor-based activity recognition in smart-home environments. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 6, 1 (2022), 1–30.
- [15] Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, et al. 2023. Qwen technical report. *arXiv preprint arXiv:2309.16609* (2023).
- [16] Xiao Bi, Deli Chen, Guanting Chen, Shanhuang Chen, Damai Dai, Chengqi Deng, Honghui Ding, Kai Dong, Qishi Du, Zhe Fu, et al. 2024. Deepseek LLM: Scaling open-source language models with longtermism. *arXiv preprint arXiv:2401.02954* (2024).
- [17] Will Brackenbury, Abhimanyu Deora, Jillian Ritchey, Jason Vallee, Weijia He, Guan Wang, Michael L Littman, and Blase Ur. 2019. How users interpret bugs in trigger-action programming. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. 1–12.
- [18] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in Neural Information Processing Systems* 33 (2020), 1877–1901.
- [19] Ye Cheng, Minghui Xu, Yue Zhang, Kun Li, Ruoxi Wang, and Lian Yang. 2025. AutoIoT: Automated IoT platform using large language models. *IEEE Internet of Things Journal* 12, 10 (2025), 13644–13656.
- [20] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. 2023. PaLM: Scaling language modeling with pathways. *Journal of Machine Learning Research* 24, 240 (2023), 1–113.
- [21] Gaetano Cimino and Vincenzo Deufemia. 2025. End-user customization of trigger-action rules through fine-tuned LLMs. In *End-User Development: 10th International Symposium*. 22–40.
- [22] Gaetano Cimino, Vincenzo Deufemia, and Mattia Limone. 2025. IoT rule generation with cross-view contrastive learning and perplexity-based ranking. *IEEE Internet of Things Journal* 12, 17 (2025), 35828–35847.
- [23] Fulvio Corno, Luigi De Russis, and Alberto Monge Roffarello. 2019. RecRules: recommending IF-THEN rules for end-user development. *ACM Transactions on Intelligent Systems and Technology (TIST)* 10, 5 (2019), 1–27.
- [24] Yibo Fu, Vivian Shen, Victor Riera Naranjo, Bolei Deng, Alex Adams, and Josiah Hester. 2025. SoundOff: Low-cost passive ultrasound tags for non-invasive and non-intrusive smart home sensing. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 9, 4 (2025), 1–32.
- [25] Weiwei Gao, Kexin Du, Yujia Luo, Weinan Shi, Chun Yu, and Yuanchun Shi. 2024. Easyask: An in-app contextual tutorial search assistant for older adults with voice and touch inputs. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 8, 3

- (2024), 1–27.
- [26] Yi Gao, Kaijie Xiao, Fu Li, Weifeng Xu, Jiaming Huang, and Wei Dong. 2024. ChatIoT: Zero-code generation of trigger-action based IoT programs. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 8, 3 (2024), 1–29.
 - [27] Markus Henkel, Steffen Haesler, Hiba K Al-Najmi, Frank Hessel, and Christian Reuter. 2025. The house that saves me? Assessing the role of smart home automation in warning scenarios. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 9, 1 (2025), 1–32.
 - [28] Myles Hollander, Douglas A Wolfe, and Eric Chicken. 2013. *Nonparametric statistical methods*. John Wiley & Sons.
 - [29] Bing Huang, Hai Dong, and Athman Bouguettaya. 2021. Conflict detection in IoT-based smart homes. In *2021 IEEE International Conference on Web Services (ICWS)*. IEEE, 303–313.
 - [30] Justin Huang and Maya Cakmak. 2015. Supporting mental model accuracy in trigger-action programming. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing*. 215–225.
 - [31] Xinyu Huang, Leming Shen, Zijing Ma, and Yuanqing Zheng. 2026. Towards Privacy-Preserving and Personalized Smart Homes via Tailored Small Language Models. *IEEE Transactions on Mobile Computing* (2026).
 - [32] Yifei Huang, Jilan Xu, Baoqi Pei, Yuping He, Guo Chen, Lijin Yang, Xinyuan Chen, Yaohui Wang, Zheng Nie, Jinyao Liu, et al. 2025. Vinci: A real-time smart assistant based on egocentric vision-language model for portable devices. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 9, 3 (2025), 1–33.
 - [33] Zhiming Huang, Jiawen Chen, Liwei Shen, and Xing Chen. 2022. Fraship: A Framework to support end-user personalization of smart home services with runtime knowledge graph. In *WWW '22: Companion Proceedings of the Web Conference 2022*. Association for Computing Machinery.
 - [34] Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, et al. 2024. GPT-4o system card. *arXiv preprint arXiv:2410.21276* (2024).
 - [35] Evan King, Haoxiang Yu, Sangsu Lee, and Christine Julien. 2024. Sasha: creative goal-oriented reasoning in smart homes with large language models. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 8, 1 (2024), 1–38.
 - [36] Ruoyu Li, Qing Li, Yucheng Huang, Qingsong Zou, Dan Zhao, Zhengxin Zhang, Yong Jiang, Fa Zhu, and Athanasios V Vasilakos. 2024. SeIoT: Detecting anomalous semantics in smart homes via knowledge graph. *IEEE Transactions on Information Forensics and Security* (2024).
 - [37] Haipeng Liu, Yuheng Wang, Anfu Zhou, Hanyue He, Wei Wang, Kunpeng Wang, Peilin Pan, Yixuan Lu, Liang Liu, and Huadong Ma. 2020. Real-time arm gesture recognition in smart home scenarios via millimeter wave sensing. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 4, 4 (2020), 1–28.
 - [38] Liwei Liu, Tao Wang, Wei Chen, Jun Wei, Wei Wang, and Guoquan Wu. 2025. LEGO: Synthesizing IoT device components based on static analysis and large language models. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 9, 2 (2025), 1–26.
 - [39] Xiaoyi Liu, Yingtian Shi, Chun Yu, Cheng Gao, Tianao Yang, Chen Liang, and Yuanchun Shi. 2023. Understanding in-situ programming for smart home automation. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 7, 2 (2023), 1–31.
 - [40] Andrea Mattioli and Fabio Paternò. 2023. A mobile augmented reality app for creating, controlling, recommending automations in smart homes. *Proceedings of the ACM on Human-Computer Interaction* 7, MHCI (2023), 1–22.
 - [41] Andrea Mattioli and Fabio Paternò. 2025. Personalised recommendations for daily automations controlled by mobile augmented reality. *Proceedings of the ACM on Human-Computer Interaction* 9, 4 (2025), 1–22.
 - [42] Ozan Baris Mulayim, Gabe Fierro, Mario Bergés, and Marco Pritoni. 2025. Towards zero-shot question answering in CPS-IoT: large language models and knowledge graphs. In *Proceedings of the 2nd International Workshop on Foundation Models for Cyber-Physical Systems & Internet of Things*. 7–12.
 - [43] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. 2009. Semantics and complexity of SPARQL. *ACM Transactions on Database Systems (TODS)* 34, 3 (2009), 1–45.
 - [44] Xiaomi IoT Developer Platform. 2026. MIoT Specification. <https://iot.mi.com/v2/new/doc/introduction/knowledge/spec>.
 - [45] Pragnya Ramjee, Bhuvan Sachdeva, Satvik Golechha, Shreyas Kulkarni, Geeta Fulari, Kaushik Murali, and Mohit Jain. 2025. CataractBot: an LLM-powered expert-in-the-loop chatbot for cataract patients. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 9, 2 (2025), 1–31.
 - [46] Hongyu Ren, Hanjun Dai, Bo Dai, Xinyun Chen, Michihiro Yasunaga, Haitian Sun, Dale Schuurmans, Jure Leskovec, and Denny Zhou. 2021. LEGO: latent execution-guided reasoning for multi-hop question answering on knowledge graphs. In *International Conference on Machine Learning*. 8959–8970.
 - [47] Dmitriy Rivkin, Francois Hogan, Amal Feriani, Abhisek Konar, Adam Sigal, Xue Liu, and Gregory Dudek. 2025. AIoT smart home via autonomous LLM agents. *IEEE Internet of Things Journal* (2025).
 - [48] Leming Shen, Qiang Yang, Xinyu Huang, Zijing Ma, and Yuanqing Zheng. 2025. GPIoT: Tailoring Small Language Models for IoT Program Synthesis and Development. In *Proceedings of the 23rd ACM Conference on Embedded Networked Sensor Systems*. 199–212.

- [49] Leming Shen, Qiang Yang, Yuanqing Zheng, and Mo Li. 2025. AutoIoT: LLM-Driven Automated Natural Language Programming for AIoT Applications. In *Proceedings of the 31st Annual International Conference on Mobile Computing and Networking*. 468–482.
- [50] Leming Shen and Yuanqing Zheng. 2024. IoTCoder: A Copilot for IoT Application Development. In *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking*. 1647–1649.
- [51] Yingtian Shi, Xiaoyi Liu, Chun Yu, Tianao Yang, Cheng Gao, Chen Liang, and Yuanchun Shi. 2024. Bridging the gap between natural user expression with complex automation programming in smart homes. *arXiv preprint arXiv:2408.12687* (2024).
- [52] Yunpeng Song, Yiheng Bian, Xiaorui Wang, and Zhongmin Cai. 2024. Learning from user-driven events to generate automation sequences. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 7, 4 (2024), 1–22.
- [53] Vijay Srinivasan, Christian Koehler, and Hongxia Jin. 2018. RuleSelector: Selecting conditional action rules from user behavior patterns. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 2, 1 (2018), 1–34.
- [54] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. LLaMA: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971* (2023).
- [55] Shreshth Tuli, Giuliano Casale, and Manuel Roveri. 2025. SELA: Smart edge LLM agent to optimize response trade-offs of AI assistants. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 9, 3 (2025), 1–18.
- [56] Blase Ur, Elyse McManus, Melwyn Pak Yong Ho, and Michael L Littman. 2014. Practical trigger-action programming in the smart home. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 803–812.
- [57] Blase Ur, Melwyn Pak Yong Ho, Stephen Brawner, Jiyun Lee, Sarah Mennicken, Noah Picard, Diane Schulze, and Michael L Littman. 2016. Trigger-action programming in the wild: An analysis of 200,000 IFTTT recipes. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*. 3227–3231.
- [58] Gang Wu, Liang Hu, Yuxiao Hu, Xingbo Xiong, and Feng Wang. 2025. LLM4TAP: LLM-enhanced TAP rule recommendation. *IEEE Internet of Things Journal* 12, 10 (2025), 13157–13169.
- [59] Qinyue Wu, Beijun Shen, and Yuting Chen. 2020. Learning to recommend trigger-action rules for end-user development: A knowledge graph based approach. In *Reuse in Emerging Software Engineering Practices*. Springer, 190–207.
- [60] Ding Xiao, Qianyu Wang, Ming Cai, Zhaohui Zhu, and Weiming Zhao. 2020. A3ID: an automatic and interpretable implicit interference detection method for smart home via knowledge graph. *IEEE Internet of Things Journal* 7, 3 (2020), 2197–2211.
- [61] Kaijie Xiao, Yi Gao, Fu Li, Weifeng Xu, Pengzhi Chen, and Wei Dong. 2024. ChatCam: Embracing LLMs for contextual chatting-to-camera with interest-oriented video summarization. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 8, 4 (2024), 1–34.
- [62] Xuhai Xu, Bingsheng Yao, Yuanzhe Dong, Saadia Gabriel, Hong Yu, James Hendler, Marzyeh Ghassemi, Anind K Dey, and Dakuo Wang. 2024. Mental-LLM: Leveraging large language models for mental health prediction via online text data. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 8, 1 (2024), 1–32.
- [63] Zhenyu Xu, Hailin Xu, Zhouyang Lu, Yingying Zhao, Rui Zhu, Yujiang Wang, Mingzhi Dong, Yuhu Chang, Qin Lv, Robert P Dick, et al. 2024. Can large language models be good companions? An LLM-based eyewear system with conversational common ground. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 8, 2 (2024), 1–41.
- [64] Xiaoying Yang, Jacob Sayono, Jess Xu, Jiahao Nick Li, Josiah Hester, and Yang Zhang. 2022. MiniKers: Interaction-powered smart environment automation. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 6, 3 (2022), 1–22.
- [65] Ziqi Yang, Xuhai Xu, Bingsheng Yao, Ethan Rogers, Shao Zhang, Stephen Intille, Nawar Shara, Guodong Gordon Gao, and Dakuo Wang. 2024. Talk2care: An LLM-based voice assistant for communication between healthcare providers and older adults. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 8, 2 (2024), 1–35.
- [66] Imam Nur Bani Yusuf, Diyanah Binte Abdul Jamal, Lingxiao Jiang, and David Lo. 2022. Recipegen++: an automated trigger action programs generator. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1672–1676.
- [67] Imam Nur Bani Yusuf, Lingxiao Jiang, and David Lo. 2022. Accurate generation of trigger-action programs with domain-adapted sequence-to-sequence learning. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*. 99–110.
- [68] Lefan Zhang, Weijia He, Jesse Martinez, Noah Brackenbury, Shan Lu, and Blase Ur. 2019. AutoTap: Synthesizing and repairing trigger-action programs using LTL properties. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 281–291.
- [69] Lefan Zhang, Weijia He, Olivia Morkved, Valerie Zhao, Michael L Littman, Shan Lu, and Blase Ur. 2020. Trace2TAP: Synthesizing trigger-action programs from traces of behavior. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 4, 3 (2020), 1–26.
- [70] Lefan Zhang, Cyrus Zhou, Michael L Littman, Blase Ur, and Shan Lu. 2023. Helping users debug trigger-action programs. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 6, 4 (2023), 1–32.
- [71] Yuchen Zhao, Lifu Wang, and Kai Dong. 2026. Retrieval-augmented Generation of Enhanced Trigger-action Programming Rules in Smart Home. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 10, 1 (2026), 1–32.

A Baseline Adaptation

This appendix summarizes additional implementation details and task adaptations of reproduced baselines. Complete prompts, hyperparameters, implementation specifications, and evaluation scripts are available at: <https://github.com/linyafu/KGIOT-artifact>.

A.1 Common Configuration

All baselines are evaluated under the same protocol: the same LLM backbone and decoding configuration, Home Assistant deployment environment and MIoT device inventory, unified TAP representation, and deployment/evaluation pipeline. All methods receive a single natural-language request and generate one TAP; interactive clarification is disabled following the single-turn benchmark protocol.

A.2 Sasha

For Sasha [35], we preserve its original Clarifying–Filtering–Planning workflow while adapting the Planning stage to the unified TAP representation. The Filtering stage receives device metadata for task-relevant device selection, and the Planning stage receives the corresponding capability schema.

A.3 ChatIoT

We reproduce ChatIoT [26] with its semantic compression pipeline. The compressor uses all-MiniLM-L6-v2 with cosine similarity and DBSCAN clustering ($\epsilon = 0.05$, $\min_samples = 1$). The original AskUser module is disabled under the single-turn protocol.

To address threshold sensitivity, we perform a benchmark-wide sweep and report both the original thresholds ($\tau_d = 0.45$, $\tau_s = 0.35$) and one global diagnostic configuration ($\tau_d = 0.10$, $\tau_s = 0.20$), selected once for the entire benchmark.

A.4 HomeGenii

We implement a task-adapted HomeGenii [71] baseline while preserving its retrieval-augmented generation pipeline. The released Enhanced TAP rulebase is used for expert-rule retrieval. Task-relevant home devices are retrieved before generation, and the retrieved capability schemas are provided to the generator. The generated output is converted to the unified TAP representation.

A.5 AutoloT

We implement a task-adapted AutoIoT [19] baseline while preserving its Generate–Verify–Repair workflow. The generator produces one TAP, followed by verification and up to two repair iterations when necessary. The original multi-rule conflict verification is omitted because our benchmark evaluates one TAP per request.

B Performance Across Different LLMs

To evaluate whether KGIOT depends on a particular LLM family, we additionally replace the planning backbone with GPT-4o, Qwen3.6-Plus, and DeepSeek-V4-Pro while keeping the KGIOT framework unchanged.

B.1 Performance Analysis

Table 8 reports executable correctness, platform validity, grounding precision/recall, efficiency, and category-wise performance across the three home environments. Overall, GPT-4o achieves the highest executable correctness across all home scales. Qwen3.6-Plus and DeepSeek-V4-Pro also maintain high platform validity (90–100%) and grounding quality, with device-level recall above 94% across all settings. Performance differences are mainly reflected in executable TAP generation rather than grounding accuracy.

Table 8. Performance comparison of KGIoT with different LLM backbones. Corr.: executable correctness; Valid.: platform validity. P^D/R^D and P^C/R^C denote device- and capability-level grounding precision/recall. RC1–RC4 denote four request categories. Lat. denotes latency.

LLM	Overall								Request Category					
	Corr.	Valid.	P^D	R^D	P^C	R^C	Tokens	Lat. (s)	RC1	RC2	RC3	RC4-Expl.	RC4-Impl.	RC4-Corr.
Home-S														
gpt-4o	100.0	100.0	100.0	100.0	94.67	100.0	3656	2.58	100.0	100.0	100.0	100.0	100.0	100.0
qwen3.6-plus	95.0	100.0	100.0	100.0	87.67	100.0	3732	8.89	100.0	80.0	100.0	100.0	100.0	100.0
deepseek-v4-pro	95.0	95.0	100.0	100.0	85.75	100.0	3743	12.031	100.0	80.0	100.0	100.0	100.0	100.0
Home-M														
gpt-4o	100.0	100.0	93.75	100.0	85.75	100.0	5232	3.30	100.0	100.0	100.0	100.0	100.0	100.0
qwen3.6-plus	85.0	95.0	92.09	100.0	74.59	95.0	5312	9.21	80.0	100.0	60.0	100.0	100.0	100.0
deepseek-v4-pro	95.0	100.0	94.59	100.0	86.17	100.0	5413	12.60	100.0	80.0	100.0	100.0	100.0	100.0
Home-L														
gpt-4o	85.0	95.0	92.09	94.59	83.74	93.96	10750	3.32	100.0	80.0	60.0	100.0	100.0	100.0
qwen3.6-plus	80.0	90.0	91.68	97.50	69.16	87.71	10858	11.40	100.0	60.0	80.0	80.0	80.0	80.0
deepseek-v4-pro	80.0	95.0	97.34	96.67	85.39	97.09	11400	14.37	100.0	60.0	60.0	100.0	100.0	100.0

Category-wise results show that backbone differences are most evident in Conditional TAPs (RC2) and Multi-Action TAPs (RC3), especially in larger homes. In contrast, Constraint-Aware TAPs remain relatively stable because KGIoT provides explicit capability grounding and deterministic constraint injection. GPT-4o and DeepSeek-V4-Pro achieve 100% implicit-constraint completion across all homes, while Qwen3.6-Plus only decreases on Home-L (80%). These results suggest that backbone choice mainly affects explicit request interpretation and structured TAP generation.

Token usage remains similar across backbones because all models receive the same structured context and generate the same TAP representation. The main efficiency difference lies in inference latency: GPT-4o requires approximately 2.6–3.3 s per request, compared with 8.9–11.4 s for Qwen3.6-Plus and 12.0–14.4 s for DeepSeek-V4-Pro.

B.2 Representative Failure Cases

The quantitative results indicate that changing the planner backbone primarily affects structured-output reliability, trigger formulation, fine-grained capability selection, and action assignment in longer requests. We present three representative cases below. The KGIoT pipeline and evaluation protocol are kept unchanged across backbones. For readability, only the TAP atoms relevant to each error are shown.

Case 1: Trigger and output-structure errors

(Home-S, HS_COND_007, Conditional TAP)

Request. When the entrance door is opened, if it is after 9 PM, turn on the entrance light.

Ground truth.

```
trigger: event:entrance_door_sensor.magnet_sensor.open
condition: time>=21:00
action: entrance_light.light.on=true
```

Qwen3.6-Plus

```
trigger: event:entrance_door_sensor.magnet_sensor.door_not_closed
condition: time>=21:00
action: entrance_light.light.on=true
```

Executable correctness: ✗ Platform validity: ✓

Observation. Qwen generates a structurally valid TAP, but uses the non-equivalent event name `door_not_closed` instead of the annotated open event or an accepted state-predicate equivalent. The time condition and action are generated correctly.

DeepSeek-V4-Pro

Executable correctness: ✗ **Platform validity:** ✗

No parseable TAP was returned.

TAP generation failed: `time_condition.at` cannot be used together with `after/before`.

Observation. The generated intermediate representation violates the structured TAP constraints and is rejected before deployment. This case illustrates that backbone choice affects both semantic trigger formulation and structured-output reliability.

Case 2: Action omission and device assignment errors

(Home-L, HL_MULTI_012, Multi-Action TAP)

Request. *At 11:30 PM, if the master bedroom window is closed, turn off the master bedroom main light, turn on the bedside lamp at 15% brightness, close the bedroom curtain, turn on the air conditioner and set it to 26°C, and turn on the humidifier.*

Ground-truth action atoms (excerpt).

```
bedroom_main_light.light.on=false
bedroom_beside_lamp.light.on=true
bedroom_beside_lamp.light.brightness=15
bedroom_curtain.curtain.motor_control=1
...
```

GPT-4o

Executable correctness: ✗ **Platform validity:** ✓

The time trigger and window condition are correct, but the generated action set omits:

```
bedroom_main_light.light.on=false
bedroom_beside_lamp.light.on=true
bedroom_beside_lamp.light.brightness=15
```

Only the air-conditioner, humidifier, and curtain actions are retained.

Observation. The generated TAP is structurally valid but incomplete because three explicitly requested lighting actions are missing.

DeepSeek-V4-Pro

Executable correctness: ✗ **Platform validity:** ✓

```
bedroom_main_light.light.on=false
bedroom_main_light.light.brightness=15
bedroom_curtain.curtain.motor_control=1
...
```

Observation. DeepSeek generates most of the requested actions, but assigns the 15% brightness value to the main light rather than the bedside lamp. The error therefore arises from an incorrect device–action correspondence within a long multi-device plan.

Case 3: Capability selection ambiguity

(Home-M, HM_COND_009, Conditional TAP)

Request. *When the cooker is turned on, if the kitchen light is on, turn on the kitchen range hood.*

Ground truth.

```
trigger: kitchen_cooker.kitchen_stove.status==2
condition: kitchen_light.light.on==true
action: kitchen_hood.hood.on=true
```

DeepSeek-V4-Pro

Executable correctness: ✗ **Platform validity:** ✓

```
trigger: kitchen_cooker.kitchen_stove.status==2
condition: kitchen_light.ambient_light.on==true
action: kitchen_hood.hood.on=true
```

Observation. The model identifies the correct physical device but selects `ambient_light.on` instead of the required `light.on` predicate. Although both capabilities belong to the same device, they represent different services and are not interchangeable. This case shows that correct device grounding does not guarantee correct capability selection.

Taken together, these cases summarize three major backbone-dependent error patterns: structured-output failure, incomplete or inconsistent multi-action planning, and fine-grained capability selection errors. They are consistent with the quantitative results in Table 8, where grounding quality remains relatively stable while executable TAP generation varies across planner backbones.

Table 9. Mapping from platform metadata to KG elements.

Metadata source	KG element	Purpose
Home Assistant device registry	Device entities and identifiers	Represents the deployed device inventory
Home Assistant area assignments	Area entities and area-to-device relations	Captures spatial organization and supports same-area matching
Home Assistant runtime entities	Executable entity bindings	Connects capabilities to deployable sensing and actuation interfaces
MIoT model identifiers and instance specifications	Services and capability definitions	Specifies supported properties, events, actions, access modes, and value ranges
Capability access metadata	TAP-role annotations and dependency relations	Determines readability, writability, and operational prerequisites
Configurable interlock templates	Instantiated cross-device relations	Encodes environmental requirements for matching device configurations

C Implementation Details of Knowledge Graph Construction

This appendix describes the implementation details of capability-centric KG construction. The implementation integrates Home Assistant deployment metadata with Xiaomi MIoT capability specifications.

C.1 Metadata Integration and Construction Pipeline

At initialization, KGIoT retrieves registered devices from Home Assistant, including device identifiers, models, areas, and runtime entities. Each device model is mapped to its MIoT specification, which defines supported services, properties, events, and actions. These metadata sources are joined by device identity and model information to construct a unified device representation.

For each device, KGIoT derives normalized service-level capabilities, where properties, events, and actions are represented as distinct capability types. Capability access modes determine read/write properties and admissible TAP roles. During construction, KGIoT builds area–device, device–capability, and capability–device mappings together with a global capability catalog.

After capability construction, operational dependencies are inferred from capability metadata, and environmental-interlock templates are instantiated by matching capability types and area information against the synchronized home inventory. These relations are stored directly in the KG as capability-level edges.

When device additions or removals are detected, KGIoT reruns the construction pipeline to refresh device metadata, capability mappings, and constraint relations. The current implementation therefore reconstructs the KG upon inventory changes rather than incrementally updating individual graph elements.

C.2 Resulting Data Organization

The constructed KG contains area, device, and service entities together with service-level capability entries. Structural associations capture area–device membership and device–service relationships. Three direct lookup mappings support constraint-based retrieval: area-to-device, device-to-capability, and capability-to-device. A global capability catalog maintains unique capability types and their semantic annotations. Operational dependencies and environmental interlocks are stored as capability-level relations associated with the relevant device instances.