

FloWeaver: Validation-guided Synthesis of Low-Latency Execution Flows for Cross-Device IoT Automation

SIYUAN LIU, Hong Kong University of Science and Technology (Guangzhou), China

HUANGXUN CHEN*, Hong Kong University of Science and Technology (Guangzhou), China

Smart-environment routines are expected to deliver instant and coherent cross-device experiences, yet today's home-automation platforms typically execute routines sequentially across independently maintained device integrations. Our empirical study of 200 real-world routines shows that more than 90% take over 1s to complete and over 30% exceed 5s, causing noticeable user-perceived delays. This paper presents FloWeaver, a validation-guided synthesis system for low-latency execution flows in cross-device automation. Instead of treating a routine as a chain of independent device operations, FloWeaver decomposes workflows into minimal self-contained semantic units and weaves them into a latency-efficient execution flow while preserving observable routine behavior within comprehensive routine scenarios. To achieve this, FloWeaver introduces three key designs: a primitive extractor that identifies semantic units based on routine specifications and platform logic, a constraint-aware orchestrator that represents semantic dependencies and resource constraints as directed acyclic graphs and performs two-stage scheduling, and an equivalence validator that checks scenario-conditioned observational equivalence and progressively rolls back transformations that introduce semantic divergences. Across the four latency groups, FloWeaver achieves pass rates of 88.89%–100% on the smart-home dataset and 87.50%–100% on the long-chain dataset. Among accepted routines, its average latency reductions range from 41.12% to 66.00% and from 73.11% to 75.53%, respectively. FloWeaver provides the strongest combined pass-rate and latency performance compared with HA-native, HEFT, NaiveTopo-Parallel, and a state-of-the-art LLM baseline. An independent audit finds matching device and platform behavior across all 150 audited paired executions, while the validator rejects all 150 injected semantic-breaking mutations. FloWeaver generates optimized routines within 13.66–17.75 s on average, and an $N = 16$ within-subject user study shows improved perceived responsiveness and acceptance. Code is publicly available at <https://github.com/X-EASys/FloWeaver>.

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

Additional Key Words and Phrases: Large Language Model, Personalized Memory Management

ACM Reference Format:

Siyuan Liu and Huangxun Chen. 2026. FloWeaver: Validation-guided Synthesis of Low-Latency Execution Flows for Cross-Device IoT Automation. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 10, 4, Article 241 (December 2026), 41 pages. <https://doi.org/10.1145/3857995>

1 Introduction

A great smart-environment experience features the instant activation of user-specific routines. For example, a morning routine involving multiple physical devices may include: “*turn off all living-room lights, open the curtains, and, when motion is detected in the living room, start preparing a latte.*” In home automation platforms [3, 15, 29, 32], such routines are often delivered by default in a sequential manner: *lights are turned off one by one, the curtains*

*Corresponding author.

Authors' addresses: Siyuan Liu, Hong Kong University of Science and Technology (Guangzhou), Guangzhou, Guangdong, China, sliu268@connect.hkust-gz.edu.cn; Huangxun Chen (corresponding author), Hong Kong University of Science and Technology (Guangzhou), Guangzhou, Guangdong, China, huangxunchen@hkust-gz.edu.cn.



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-ART241

<https://doi.org/10.1145/3857995>

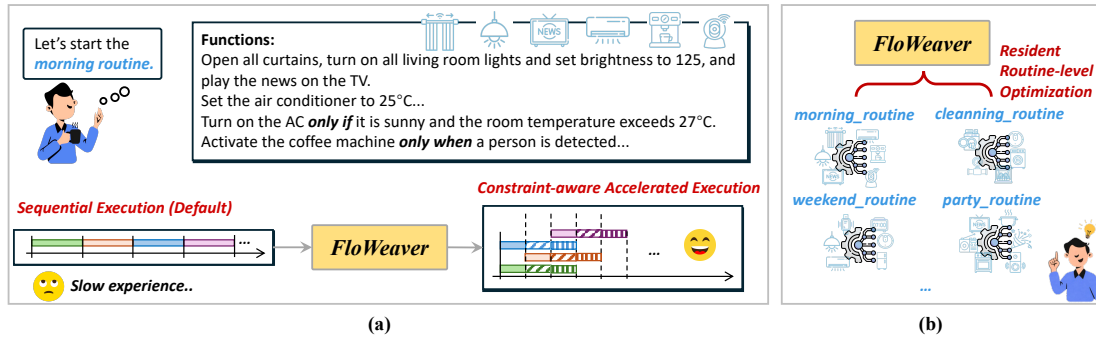


Fig. 1. Overview of FloWeaver's usage scenarios. (a) Each user may have personalized routines that involve multiple devices. The default sequential execution model incurs long execution latency and degrades user experience. FloWeaver is designed to accelerate the execution pipeline while preserving semantic dependencies and satisfying system resource constraints. (b) FloWeaver runs as a resident process to continuously perform routine-level optimization.

start moving seconds later, and the coffee machine starts only after the user has already entered the living room, failing to deliver most optimal user experience. In our empirical investigation of 200 real-world routines, more than 90% take over 1s to complete, while over 30% exceed 5s, incurring noticeable waiting time and motivating the need for optimization. The underlying reason is that, at the system level, each device and associated integration is maintained independently. At runtime, a routine is realized as a chain of independent device operations, where each involved function is wrapped within its own session-management logic and executed sequentially. Through detailed examination, we find substantial room for execution acceleration. For example, if two devices use different control modalities (e.g., BLE and cloud-api), incur no resource contention, and have no semantic dependency implied by the routine, their operations can be executed in parallel. More broadly, even after accounting for constraints such as resource contention and semantic dependencies, significant optimization opportunities remain, which will be elaborated in §2.

Thus, we propose FloWeaver, which acts as an orchestrator that decomposes each individual device operations into finer-grained semantic units and weaves them into a latency-efficient execution flow, as shown in Figure 1(a). This flow still satisfies the relevant system resource constraints and semantic dependencies. In this way, a routine is no longer treated as a chain of independent device operations, but instead behaves as *a unified virtual device with a more efficient internal execution flow* for realizing the user's intended routine. FloWeaver can run as a resident process that continuously performs routine-level optimization to accelerate user-specific and dynamically defined routines, as shown in Figure 1(b). Although the core idea of FloWeaver appears straightforward, we encounter three key challenges in ensuring the efficiency and correctness of the resulting execution flow.

CH1: Semantic-unit Decomposition. As shown in Figure 1, a prerequisite for enabling acceleration is to decompose each individual device workflow into finer-grained semantic units. However, workflows represented in code form do not expose obvious semantic boundaries. If the decomposition is too coarse-grained, the available acceleration space becomes limited. If it is too fine-grained, each unit may no longer be self-contained with the necessary execution context, including session objects, state write-backs, and lifecycle hooks. In this case, reordering such units may violate equivalence with the original routine intent. Thus, we need an effective decomposition method to extract proper semantic units for acceleration-oriented orchestration.

CH2: Multi-aspect Constraint Satisfaction. Fundamentally, FloWeaver accelerates execution by rearranging and parallelizing semantic units, but this optimization must be performed under multiple types of constraints. First, FloWeaver must respect *dependencies enforced either by program logic or by routine intent*. For example,

a data write-back must occur only after the corresponding data has been refreshed, and turning on the air conditioner may be executed after checking the temperature sensor. Secondly, FloWeaver should account for *system resource constraints*. For instance, if two BLE-controlled devices share only one available BLE connection slot, they cannot be operated simultaneously. Similarly, multiple cloud-API-controlled devices may be subject to rate limits imposed by the cloud service. Thus, we need a way to incorporate these constraints into the orchestration process to generate a feasible execution flow.

CH3: Routine-Semantics Validation and Revision. While orchestration aims to reduce latency, it may introduce semantic shifts relative to the default sequential execution, as reordering semantic units can alter the original routine behavior. Thus, we need a reliable validation method to check whether the new execution flow is equivalent to the default one in terms of key observable behaviors, together with an accompanying revision mechanism to correct detected violations, enabling the entire process to be fully automated without human intervention.

To the best of our knowledge, no prior work has enabled similar capabilities. Some studies [18, 19, 40] focus primarily on understanding user intent and composing existing device resources to realize the desired routines. Other prior works [25, 41] study IoT program synthesis, but do not focus on accelerating routine execution. Thus, FloWeaver is designed to fill this gap by reducing the end-to-end latency of user-specified routines, thereby improving user-perceived experience. FloWeaver addresses the above challenges through three dedicated designs. First, for **CH1**, FloWeaver features a primitive extractor that leverages both routine-level clues and automation-platform logic clues to identify minimal self-contained semantic units (MSSUs). Then, to address **CH2**, FloWeaver features a constraint-aware orchestrator that operates on directed acyclic graphs (DAGs), where MSSUs are represented as nodes and semantic dependencies and system resource constraints are explicitly represented as edges. It further derives an action-level DAG from the MSSU-level DAG. The orchestrator then applies a two-stage strategy, with macro-level scheduling and micro-level scheduling, to generate a latency-optimized execution flow. Finally, to address **CH3**, FloWeaver features an equivalence validator that performs differential comparison between the default sequential execution and the optimized execution by comparing key execution traces. When differences are identified, the localized divergence point guides a progressive rollback scheme to revise the execution flow until the violations are resolved or the rollback budget is reached. In summary, we make the following contributions:

- (1) To the best of our knowledge, FloWeaver is the first to enable validation-guided synthesis of low-latency execution flows for cross-device automation, autonomously improving user experience in smart environments. The key idea is to decompose individual device operations into finer-grained semantic units and weave them into a latency-efficient execution flow.
- (2) To ensure the efficiency and correctness of the resulting execution flow, FloWeaver features three key modules: primitive extractor, constraint-aware orchestrator, and equivalence validator. Specifically, FloWeaver adopts a graph in which semantic units are represented as nodes and their associated constraints are represented as edges. Combined with a two-stage execution-flow optimization strategy and trace-based validation, FloWeaver generates optimized execution flows that achieve lower latency than the original sequential execution while preserving routine semantics.
- (3) We conduct a systematic evaluation to demonstrate the effectiveness and efficiency of FloWeaver. On a realistic routine dataset with moderate lengths, FloWeaver achieves a pass rate of 88.89%–100% and reduces average latency by approximately 41.12%–66% relative to the default sequential execution. When routines become longer, FloWeaver achieves a pass rate of 87.5%–100% and reduces latency by 73.11%–75.53%. In addition, FloWeaver is efficient, taking only 13–18s on average to generate an optimized routine. The second-best baseline, a state-of-the-art LLM, Claude 4.6 Sonnet, achieves only a pass rate of 66.67%–90.48% for short routines and 75%–87.5% for long routines, with latency reductions of 37.62%–67.01% for short routines

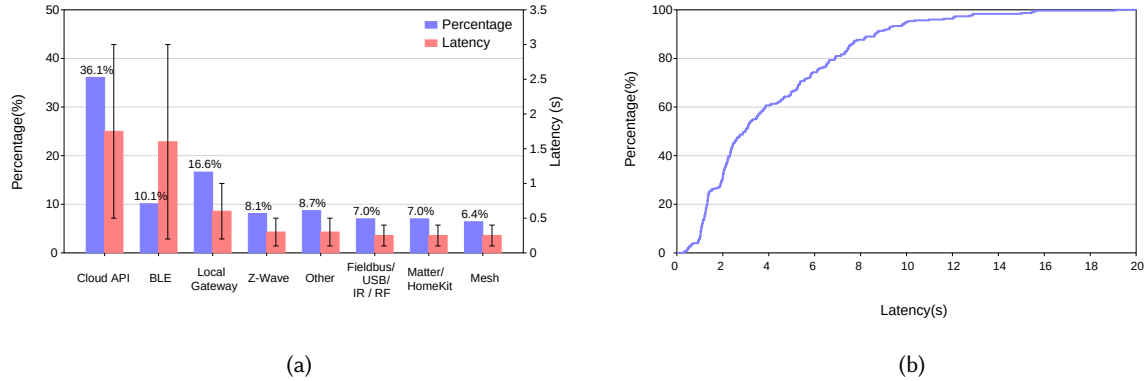


Fig. 2. Preliminary investigation on execution efficiency. (a) Device function-level execution efficiency: the percentage of devices supported in Home Assistant across different control modalities and average latency of invoking a device function; (b) Routine-level execution efficiency: the cumulative latency distribution of 200 realistic routines sampled from IFTTT.

and 33.56%–62.25% for long routines, which are much lower than those achieved by FloWeaver, while requiring $2.3\times$ – $3.6\times$ longer optimization time and 28,780–68,117 tokens per routine; FloWeaver makes no per-routine LLM calls. In addition, we recruited participants to assess the user experience of the optimized executions, further demonstrating the superiority of our design.

2 Preliminary Investigation and Design Motivation

Before presenting the design of FloWeaver, we first describe our preliminary investigation to characterize the current execution efficiency of cross-device IoT routine.

Device Function-level Execution Efficiency. In smart-environment automation ecosystems, smart devices can be controlled through diverse modalities. We examine the devices supported by Home Assistant [3], the largest open-source home-automation platform, and report the distribution of their control modalities in Figure 2(a), including cloud APIs, local gateways, BLE, Z-Wave, Matter, mesh networks, and others. Notably, Cloud API integrations account for the largest share (36.1%), followed by local gateways (16.6%) and BLE (10.1%). Then, for each control modality, we purchase representative devices, measure the average latency of invoking each function, and report the results in Figure 2(a). Unsurprisingly, cloud-API-based devices exhibit high latency, ranging from roughly 500ms to more than 2s, since control commands must traverse vendor cloud services before reaching the physical device. BLE-controlled devices also exhibit high variability, ranging from approximately 200 ms to several seconds, with more pronounced tail latency than other modalities. For local-gateway-based control, although the latency of individual functions is lower than that of cloud-API- and BLE-based control, it still incurs non-negligible latency. Given its large device share, it can also substantially contribute to the overall routine latency.

Routine-level Execution Efficiency. An automation routine often involves multiple devices. To understand the execution efficiency of realistic routines, we randomly sample 200 routines from the IFTTT platform [17], covering control modalities including cloud APIs, BLE, and local gateways. We then deploy these routines on Home Assistant and measure their end-to-end latency. Classical user-tolerance theory [4] suggests that delays above 1s may interrupt user attention and increase concern about system progress, while delays above 10s are often perceived as unacceptable. As shown by the latency distribution in Figure 2(b), most routines complete in more than 1s: approximately 60% fall between 1–5s, where users may begin to experience mild waiting; 25.4% fall

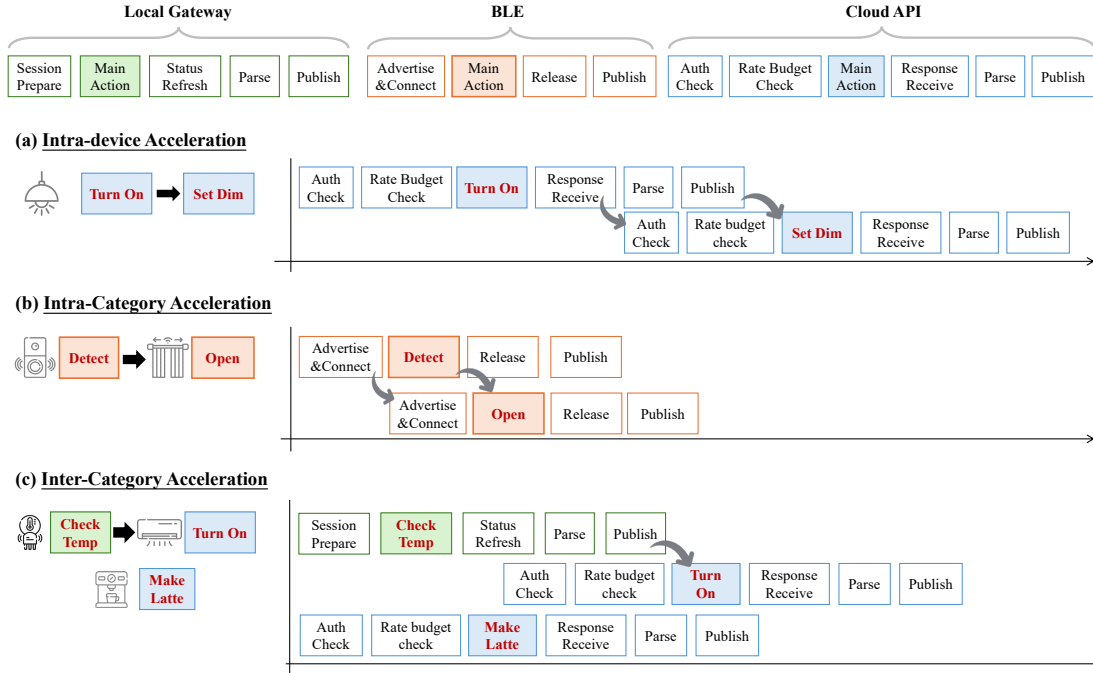


Fig. 3. Acceleration targets in FloWeaver. FloWeaver supports devices controlled through local gateways, BLE, and cloud APIs. (a) *Intra-device Acceleration*: When multiple functions are involved on the same device, the default execution model sequentially issues operations. FloWeaver identifies independent operations and executes them in parallel while preserving device-level semantic constraints. (b) *Intra-category Acceleration*: When multiple devices are controlled through the same manner, FloWeaver accounts for resource contention, such as the constraint that only one BLE advertisement can be issued at a time, as well as potential semantic dependencies implied by the routine intent, to accelerate the execution pipeline. (c) *Inter-category acceleration*: When multiple devices are controlled through different manners, they may not incur direct resource contention. Thus, FloWeaver primarily considers potential semantic dependencies implied by the routine intent to accelerate the execution pipeline.

between 5–10s, where waiting becomes salient; and 5% exceed 10s, which could be difficult to tolerate. These results suggest that **the execution efficiency of cross-device automation routines remains unsatisfactory, motivating the need for optimization to improve users' perceived experience.**

Design Motivation and Rationale. During this investigation, we find that the root cause of routine execution inefficiency is that a routine is realized as a chain of independent device operations, where each involved function is wrapped within its own session-management logic and executed sequentially as a whole. As shown at the top of Figure 3, each control modality involves specific supporting logic beyond the essential device-control command. For cloud-API-based devices, authentication and rate-budget checks must be performed before issuing the command, followed by response handling and state updates on the automation platform. For BLE devices, advertisement and connection establishment are required before the main command can be issued, followed by result collection. For local-gateway-based devices, session preparation is required beforehand, followed by status refresh and result updates afterward. In the default sequential execution, the overall latency accumulates across all steps and increases substantially with more involved device functions. Fortunately, we identify optimization opportunities across multiple perspectives.

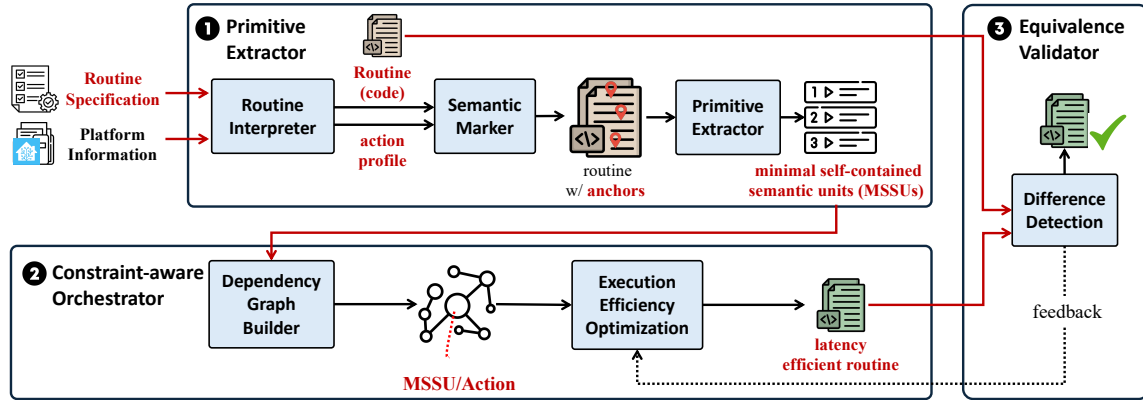


Fig. 4. Overview of FloWeaver, which consists of three main modules: (1) *Primitive Extractor*, which takes the routine specification as the main input and uses platform information as a reference to identify the default routine code. It then marks key semantic anchors related to routine execution and extracts the corresponding primitives, denoted as MSSUs, as inputs to the subsequent orchestration stage. (2) *Constraint-aware Orchestrator* organizes MSSUs into a graph structure and performs constraint-aware optimization to generate a latency-efficient routine. (3) *Equivalence Validator* compares the latency-efficient routine against its original version to examine functional equivalence. If divergences are detected, it rolls back to the previous stage and re-optimizes the routine until validation passes or the rollback budget is exhausted.

- *Intra-device Acceleration*: When a routine involves multiple functions on the same device, independent operations can be executed in parallel, such as performing authentication checks early, while preserving device-level semantic constraints, as shown in Figure 3(a).

- *Intra-category Acceleration*: When a routine involves multiple devices using the same control modality, operations without resource contention can be executed in parallel. As shown in Figure 3(b), the advertise-and-connect phase of the second device can be executed after the first device has completed its own connection phase and entered the device-command transmission phase.

- *Inter-category Acceleration*: When a routine involves multiple devices with different control modalities, more optimization opportunities may arise because they typically incur less direct resource contention. As shown in Figure 3(c), if no additional semantic dependency is implied by the routine intent, operations on different devices, such as reading a temperature sensor and operating a coffee machine, can be executed in parallel.

Thus, this work aims to develop a solution that can automatically identify these optimization opportunities within a routine and orchestrate all involved stages into a more latency-efficient execution flow, while still preserving the same semantics as the original workflow, thereby improving users' perceived experience. Our design finally achieves up to 74% latency reduction compared with the default sequential execution, as detailed in §4.

3 System Design

3.1 Overview of FloWeaver

Figure 4 provides an overview of FloWeaver. The main objective of FloWeaver is to take an existing routine code as input and synthesize a new version that preserves the same functionality while incurring lower execution time. Toward this goal, FloWeaver features three main modules: primitive extractor, constraint-aware orchestrator, and equivalence validator. *Primitive Extractor* aims to extract primitive code fragments that realize routine semantics and serve as the basic building blocks for orchestration. Specifically, it operates in two steps: first, it locates

key anchors based on clues from the routine specification and platform information; then, it expands around these anchors to segment the code into minimal self-contained units. *Constraint-aware Orchestrator* organizes all extracted primitives and their dependencies into a directed acyclic graph (DAG) representation, and performs execution-efficiency optimization to generate a latency-efficient routine. *Equivalence Validator* ensures that the final output routine preserves the same functionality as the original version. It compares the execution traces of the two versions, if divergences are detected, it triggers re-optimization until validation passes or the revision budget is exhausted.

3.2 Obtaining Platform-specific Information (One-Time Effort per Platform)

Before presenting three modules, we first describe how the platform information shown in Figure 4 is obtained. In practice, a variety of IoT automation platforms are actively used, including Home Assistant, openHAB, SmartThings, etc. These platforms differ in their implementation architectures, particularly in the code logic used to manage IoT devices and interact with them at runtime. To systematically characterize such platform-specific behavior, we construct a structured profile for each platform. Specifically, the resulting profile is represented as a JSON-based dictionary that feature the following aspects:

- (1) **Marker Sets and Function Mappings:** A key component of the platform profile is a set of semantic markers, such as BLE_CONNECT, CLOUD_HTTP_CALL, each of which is mapped to concrete functions specified in the platform documentation. For example, read_gatt_char, write_gatt_char, start_notify, stop_notify $\rightarrow$ BLE_GATT_OP indicates that, these functions, although performing distinct tasks, belong to GATT communication operations conducted after a connection to a BLE device has been established. It is worth noting that these markers serve as symbolic abstractions that enable lifecycle, ordering, and resource constraints to be expressed concisely. Thus, different platforms need not define identical sets of markers.
- (2) **Lifecycle Templates:** This category defines a set of templates that characterize the valid lifecycle of a feasible interaction session. For example, BLE_CONNECT $\rightarrow$ BLE_GATT_OP $\rightarrow$ BLE_DISCONNECT specifies that, for a BLE-controlled IoT device, the platform requires to first establish a BLE connection, then perform GATT operations such as reading or writing, and finally terminate the connection.
- (3) **Inter-Operation Ordering Rules.** These rules specify when one operation must finish before another can begin. For example, when a Cloud access token must be renewed, CLOUD_TOKEN_REFRESH must complete before CLOUD_HTTP_CALL; otherwise, the request may fail because it uses an invalid token. The scheduler preserves this order when generating an execution plan.
- (4) **Resource-Contention Rules.** These rules describe limits on shared platform or device resources. For example, if a platform supports at most K concurrent BLE connections, BUDGET_K prevents the scheduler from exceeding that limit. If two BLE operations use the same exclusive connection or adapter, NO_OVERLAP prevents them from running at the same time. Operations may still run in parallel when these resource limits are not violated.
- (5) **Other Heuristics:** Profile heuristics provide optional optimization guidance rather than mandatory correctness constraints. For example, when brightness and color are updated for the same light within a short interval and no operation observes the intermediate state, a heuristic may suggest combining them into a single async_write_ha_state() call to reduce redundant writeback overhead. If the first state write triggers a notification, validation step, or another routine, the heuristic is skipped.

We utilize each platform's documentation and code repositories to extract the corresponding structured profile described above. We adopt the following LLM-assisted pipeline for platform profile extraction:

- **Initialization Stage:** We first define a set of seed markers based on common patterns in IoT automation, such as BLE_CONNECT and BLE_DISCONNECT. We can also initialize mappings for functions with prior known knowledge,

such as `async_setup_entry` $\rightarrow$ `ENTRY_SETUP`. We then perform abstract syntax tree (AST) analysis on the platform's code repository to identify all function and organize them into the structured bank.

- **LLM-Driven Bootstrap Stage:** Using the structured record bank generated in the previous stage, together with the associated documentation and original codebase, we employ an LLM (Qwen3-14B [35] in our work) to reason over each record. The LLM may extend the initial marker set and infer additional function mappings. For example, it may propose the new marker `CLOUD_HTTP_CALL` and derive the mapping `request` $\rightarrow$ `CLOUD_HTTP_CALL` during the process. If it further maps `async_refresh_token` $\rightarrow$ `CLOUD_TOKEN_REFRESH` and determines that `async_refresh_token()` must precede `session.request()`, it can infer the ordering rule `CLOUD_TOKEN_REFRESH` $\rightarrow$ `CLOUD_HTTP_CALL`. This reasoning process continues until every record in the bank has been analyzed. Finally, the LLM performs self-consistency checks across marker types, source-code evidence, and JSON formatting before compiling the results into the final platform profile.

Notably, this procedure needs to be performed only once for each platform unless its underlying framework undergoes substantial changes. The resulting platform profile can therefore be reused across all routines deployed on that platform. Although LLM outputs can be unreliable, our empirical evaluation indicates that a moderately sized LLM can effectively summarize platform-specific information and construct a profile of acceptable quality at a modest token cost. Recent studies have similarly employed LLMs to generate program specifications [8, 27]. The instruction prompt is provided in Appendix B. This is the only stage that involves LLM usage. Because it is performed offline only once per platform, FloWeaver requires no online LLM invocations. Moreover, because the profile is stored in a human-readable plain-text format, people can readily audit, validate, and revise its contents if needed. The resulting profile will serve as a critical reference for *Primitive Extractor*, *Constraint-Aware Orchestrator*, and *Equivalence Validator*. We further discuss the boundaries of system automation and its portability to other platforms in §3.6.

3.3 Primitive Extractor

To improve efficiency by reordering action execution, we ultimately need to reorder the execution of the corresponding code fragments. As shown in Figure 4, a prerequisite is to locate where each action is implemented in the source code, so as to decompose the full code into “puzzle pieces” for subsequent execution orchestration. Thus, the desired primitives are a set of code units that collectively realize the routine semantics. The key challenge lies in ensuring both completeness and granularity. The extractor should capture all essential components; otherwise, optimization based on the extracted primitives cannot produce an equivalent routine. Meanwhile, the granularity of each primitive should be appropriate: each unit should be self-contained to facilitate orchestration, yet sufficiently minimal to expose a large optimization space. Our key idea is to mine the routine specification to obtain an action profile, which serves as the primary guidance for locating the positions and scopes of primitives, as demonstrated in Figure 5.

3.3.1 Routine Interpreter. Routines are often specified in structured formats such as JSON or YAML [3, 17], which explicitly describe the code paths, the involved devices, the actions with function names to be invoked, e.g., `switchbot:curtain.close_cover`, and user-declared semantic dependencies among actions, e.g., `if/else` or `when/then` logic. With reference to platform profile, our designed routine interpreter parses the routine specification and generates an action profile, as shown in Figure 5(a). The action profile contains a list of actions(functions) together with their control modalities, user-declared dependencies, implicit dependencies, relevant code files, and other execution-related metadata. It therefore defines the scope of the subsequent optimization.

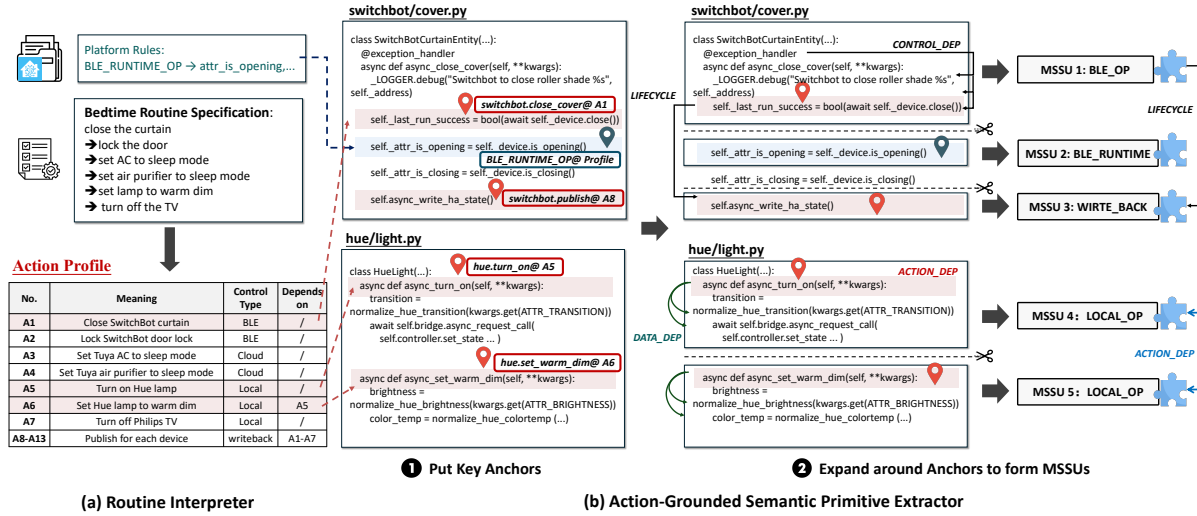


Fig. 5. An example demonstrating the workflow of Primitive Extractor. Due to space limitations, we only show code snippets for SwitchBot curtain and Hue lamp in Bedtime Routine. First, the routine interpreter derives the action profile from the routine specification. The extractor then identifies key anchors based on clues from the action profile and platform information. Finally, it performs static analysis to derive dependencies and expands around the anchors to generate self-contained semantic units.

3.3.2 Action-Grounded Semantic Primitive Extractor. Using the action profile generated by the routine interpreter as the primary guidance, we locate the key code fragments that realize the routine semantics. As shown in Figure 5, this process consists of the following concrete steps:

- **Placing Action-Grounded Anchors.** First, the extractor parses the target source files specified in the routine specification, such as `hue/light.py`, into abstract syntax trees (ASTs) and transforms them into a dependency graph $G = (V, E)$ using static analysis tools [12], where each vertex in V represents a statement and each edge in E encodes a statement-level dependency arising from typed data flow, control flow, exception handling, or other program semantics. Each node v is a candidate anchor locations. It first places anchors at statements that directly implement actions/functions in the action profile, as illustrated by the red pins in Figure 5(b). For example, the call to `async_set_warm_dim()` implements the requested lighting action. Based on the control modality of each action, we then utilize the platform profile described in §3.2 to identify necessary contextual operation enforced by the platform framework. For example, for an action performed over BLE, the operation-ordering rules in the platform profile indicate that a connection must first be established, typically via `await self._client.connect()`. Our algorithm then traverses G to identify statements that match the profile rules, designates the matched node v as context anchors automatically, as illustrated by the blue pins in Figure 5(b). Collectively, for each routine, we annotate the locations of all action anchors and their associated context anchors, as illustrated in Figure 5(b). Each identified anchor m records its marker type, such as `CLOUD_HTTP_CALL` for a Cloud API request, together with its source file, enclosing function, statement-level source range, and binding to the corresponding actions in the routine’s action profile.

- **Expanding around Anchors to Form MSSUs.** We then need to partition the source code into multiple units that are both semantically self-contained and as compact as possible. We refer to these units as *minimal self-contained*

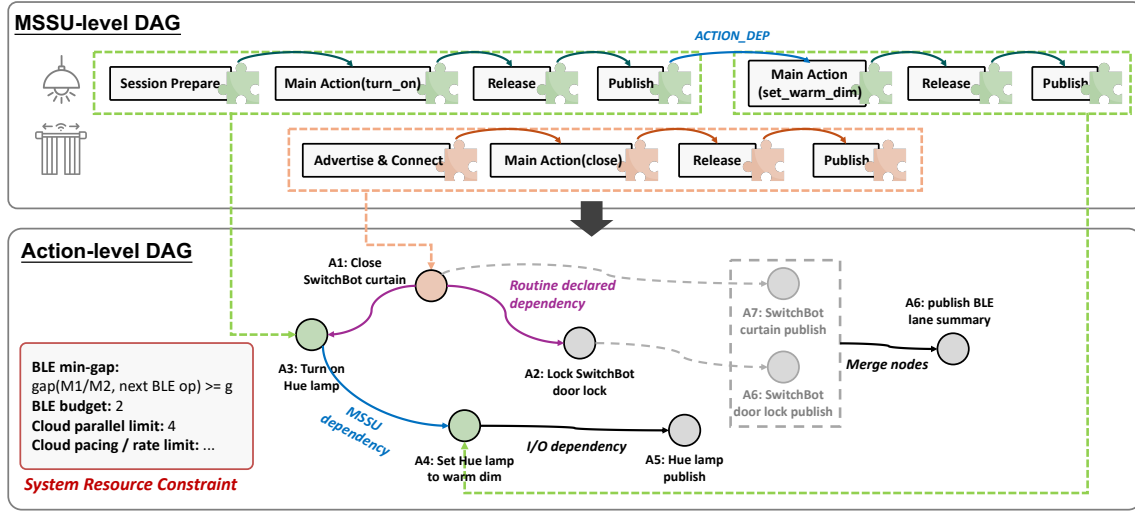


Fig. 6. An example demonstrating the workflow of the dependency-graph builder. We maintain a two-level dependency graph: a micro-level MSSU graph and a macro-level action graph. Different sources of dependencies are explicitly represented in graphs, including MSSU-derived dependencies, routine-declared dependencies, and system-resource-related constraints.

semantic units (MSSUs). To this end, we use the identified anchors as reference points for deriving safe slicing closures, rather than naively partitioning the code at the boundary preceding each subsequent anchor.

Each identified anchor m corresponds to a seed statement node $v = \text{seed}(m)$ in G . We then compute the code closure associated with each anchor m based on G as follows:

$$C(m) = \text{Closure}_{d_m, B_m}(\text{seed}(m), E_m).$$

$E_m \subseteq E$ defines the dependency types and traversal directions required for anchor m , d_m denotes the maximum dependency depth and B_m specifies the maximum number of nodes that may be retained in a closure. The closure expansion begins at $\text{seed}(m)$ and progressively incorporates additional statement nodes from G by traversing the permitted dependency edges until no further required dependencies can be identified or either limit, d_m or B_m , is reached. The resulting closure $C(m)$, i.e., a subgraph of G , preserves the code segment required by anchor m and is treated as an MSSU. Any remaining edges in G that cross closure boundaries are subsequently elevated to MSSU-level dependency edges. Notably, when two closures overlap, their shared nodes are assigned only once to avoid redundant inclusion.

3.4 Constraint-aware Orchestrator

Given the extracted code segments, i.e., MSSUs, our next objective is to identify optimization opportunities, as illustrated in Figure 3, and generate a more efficient execution flow that still preserves the original routine semantics. Here, the challenge arises from the tension between efficiency and correctness. Improving efficiency favors parallelizing operations as much as possible, while correctness requires respecting dependencies, including system-resource constraints and semantic dependencies implied by the routine intent. To achieve both objectives, our method explicitly represents MSSUs and their associated dependencies in directed acyclic graphs (DAG), and develops schedulers that optimizes over this graph representation to derive an optimized execution flow.

Table 1. Rules for promoting dependencies to hard edges.

Dependency	Promoted to hard when
DATA_DEP	The destination MSSU uses a value produced by the source MSSU.
LIFECYCLE_DEP	Platform profile requires the lifecycle order, and both MSSUs refer to the same device or session.
CONTROL_DEP	The routine explicitly states that the destination action can run only after the source action or condition.
EXCEPTION_DEP	The destination MSSU is a required error-handling or cleanup step for the source MSSU.

3.4.1 Dependency Graph Builder. As shown in Figure 6, we maintain a two-level dependency graph: an MSSU-level graph and an action-level graph. The action-level DAG is derived from the MSSU-level graph and captures more macro-level execution patterns. The rationale for maintaining two graphs is to enable progressive optimization, first at the macro level and then at the micro level, thereby reducing optimization complexity and achieving better execution efficiency at the end. In the following, we describe how to construct them in detail.

- *Building MSSU-level DAG.* We use the output of Primitive Extractor to initialize MSSU-level DAG. As described in §3.3.2, each MSSU corresponds to a subgraph of G . By contracting each MSSU into a single node, the remaining cross-MSSU edges $E' \subseteq E$ are directly inherited by the resulting MSSU-level DAG. In addition, we augment the DAG with inter-MSSU dependencies inferred from the platform profile (including lifecycle templates, inter-operation ordering rules, and resource-contention constraints) as well as dependencies specified in the action profile. Notably, multiple anchors may share the same marker. For example, a routine with two BLE devices may contain a distinct BLE_CONNECT anchor for each device. To prevent dependency constraints from being erroneously applied across unrelated anchors, we distinguish them using composite identifiers derived from attributes such as device_id, entity, endpoint, host, or the subscription handle. Not all identified dependencies require a strict execution order. FloWeaver keeps a dependency as a soft constraint unless there is clear evidence that its order must be preserved. A hard edge cannot be reordered by the scheduler, while a soft constraint only guides scheduling and may be checked through validation. Table 1 summarizes when each dependency is promoted to hard; all other cases remain SOFT_ORDER.

- *Building the action-level DAG from the MSSU-level DAG.* We then map adjacent MSSUs to the explicit actions specified in the routine to construct an action-level graph. Each action node in the graph corresponds to one or more MSSUs. Figure 6 shows an example of a *bedtime routine*: first closing the curtain, then locking the door, turning on the lamp, and setting it to warm dim. In addition to dependencies inherited from the MSSU-level DAG, the action-level graph also captures routine-declared dependencies from the action profile. We further incorporate I/O-based hard edges when a value produced by one action node is consumed by another. This step is combined with scope checks to avoid incorrectly introducing hard dependency between unrelated actions across different control modalities. In addition, resource constraints derived from platform profile are explicitly recorded and initially treated as soft dependencies, whose enforcement is determined by the scheduler. These constraints include budget, timing, and session constraints, such as BUDGET_K, NO_OVERLAP, MIN_GAP, SAME_SESSION_GROUP, MUST_REUSE_CONTEXT, RATE_LIMIT, BACKOFF_WINDOW, etc. It is worth noting that we perform cycle checks over all hard edges, since a cyclic hard-edge subgraph would be infeasible for scheduling. If a cycle is detected, we break it by downgrading one of the involved hard edges to a soft edge and mark it with a cycle_break reason for later reference by the scheduler.

3.4.2 Macro-level Scheduler. Given the dependency-graph representation, our scheduler operates in two stages: (i) a macro-level scheduler that schedules at the action granularity, and (ii) a micro-level scheduler that schedules within each control modality at the MSSU granularity. Action-level scheduling yields moderate but stable latency

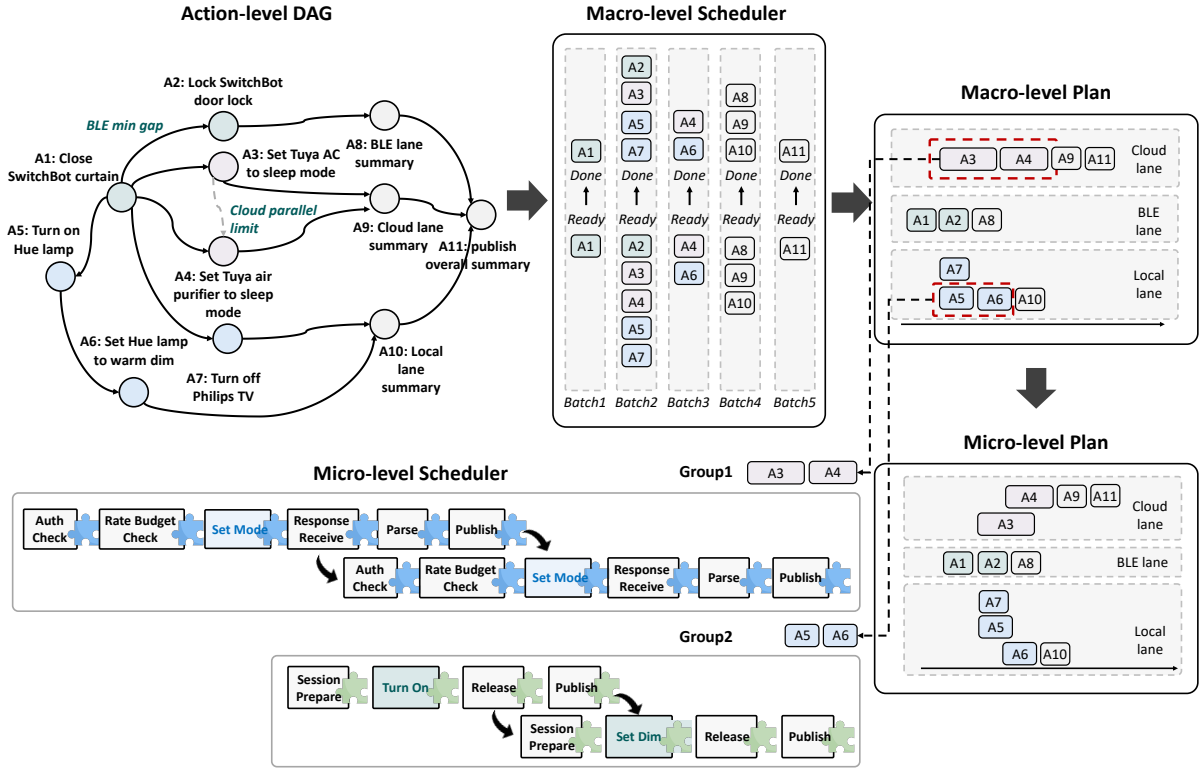


Fig. 7. An example demonstrating the workflow of the macro-level and micro-level schedulers. The macro-level scheduler applies a local greedy strategy on action-level DAG, selecting nodes and grouping them into execution batches. The micro-level scheduler then selects specific control groups, decomposes their actions into MSSUs, and performs intra-control-modality refinement based on MSSU-level DAG.

benefits, whereas finer-grained optimization can further improve efficiency but also introduces a higher risk of semantic violations. Therefore, we separate optimization into these two levels to balance optimization efficiency and correctness.

Intuitively, the macro-level scheduler aims to parallelize as many mutually compatible actions as possible, so that low-cost operations can be hidden behind slower network or device waits, while still respecting constraints and budgets. Figure 7 illustrates the workflow of the macro-level scheduler on a bedtime routine. It adopts a local greedy strategy to optimize the execution flow. Initially, we treat the system resource constraints R in the DAG as the constrained scheduling space, including concurrency budgets such as $BUDGET_K$ and $RATE_LIMIT$. The scheduler maintains a topological frontier, $Ready_t$, defined as the set of remaining action nodes with zero in-degree. At each step, the scheduler extracts a small control-group window W from the current $Ready_t$ and performs a bounded lookahead search within this window to determine the next execution batch. During lookahead, the scheduler first selects an initial feasible action node from the control-group window, and then sequentially scans for additional nodes that do not violate the constraints in R . Once the window has been examined, all selected actions are packed into a committed batch. The detailed action-selection procedure operates as follows:

Algorithm 1 Macro-level Scheduler with Bounded Frontier Search

Input: Action-level dependency graph $G = (A, E)$, scheduler policy π
Output: Execution plan $\mathcal{P} = \langle \text{Batch}_1, \dots, \text{Batch}_T \rangle$

```

1   Initialize  $\mathcal{P} \leftarrow \emptyset$  and compute action indegrees.
2   while unfinished actions remain do
3        $\text{Ready}_t \leftarrow \{a \in A \mid a \text{ is unfinished and } \text{indeg}(a) = 0\}$ 
4        $W \leftarrow \text{SELECTWINDOW}(\text{Ready}_t, G, \pi)$ 
5        $\text{Batch}_t \leftarrow \text{BOUNDEDSEARCH}(W, \pi)$ 
6       if no feasible batch is found then
7            $\text{Batch}_t \leftarrow \text{GREEDYFEASIBLEBATCH}(\text{Ready}_t, \pi)$ 
8       end if
9       Commit  $\text{Batch}_t$  and update  $\mathcal{P}$ .
10      Update indegrees and the ready set.
11  end while
12  return  $\mathcal{P}$ 

```

- **Feasibility Check:** At each selection step, we first eliminate candidates that would violate any hard dependencies or constraints with respect to the actions already selected.
- **Reordering-Sensitivity Check:** For actions that pass the feasibility check, we quantify their reordering sensitivity, i.e., the risk that advancing an action in the current schedule would cross an unresolved ordering relation. Read operations and side-effect-free setup operations are considered less sensitive, whereas an operation with an unsatisfied `SOFT_ORDER` predecessor inferred from the action-level DAG is considered more sensitive. We therefore prioritize actions with lower reordering sensitivity.
- **Tie-Breaking:** If there is a tie, we favor the action order in the original routine. If offline per-action latency estimates are available, they are used as a secondary tie-breaker, with preference given to longer-running operations.

Consider the example in Figure 7, in Batch 2, A3 is selected first after passing the feasibility check. A4 is not subsequently added because it shares the same control modality (cloud API) as A3 and their concurrent execution would exceed the permitted cloud-concurrency limit. A4 is therefore excluded by the feasibility check.

After committing one batch, the scheduler repeats this process to select the next batch. After all batches have been selected, the scheduler further consolidates actions within the same batch when they can be executed concurrently. For example, if multiple cloud devices in a batch involve publish operations, the scheduler can merge them into a single action that publishes a cloud-lane summary and performs the corresponding writeback, as long as doing so does not violate hard edges or the resource constraints R . Algorithm 1 summarizes the detailed steps of the macro-level scheduler.

3.4.3 Micro-level Scheduler. Macro-level scheduling cannot exploit efficiency gains within an action, i.e., at the MSSU level, such as BLE advertisement-side preparation, cloud authentication and budget checks, or local-API endpoint/session setup. As a result, these phases are often paid as pure waiting time. For example, when the request has been sent and is waiting for a response or data transfer is in progress, the next action can often safely complete some side-effect-free preparation in advance. The micro-level scheduler is designed to exploit such opportunities. Therefore, the micro-level scheduler operates in two steps. It first identifies candidate action groups $\text{Group}(a_i, a_j)$ for further refinement, such as A3 and A4 in Figure 7. The selection only considers actions that are temporally contiguous and share the same control modality. Then, within each candidate group, we expand each action to the MSSU level and introduce parallelism to accelerate execution. Specifically, we allow the preparation phase of the next action to borrow a safe window, i.e., the waiting interval after the current action

Algorithm 2 Micro-level Scheduler

Input: Selected candidate action groups $Group(a_i, a_j)$, coarse execution plan $\mathcal{P}$, protocol policy π
Output: Refined micro-level plan $\mathcal{P}$

```

1  Initialize  $\mathcal{P} \leftarrow \emptyset$ .
2  for each candidate action group  $g \in Group$  do
3     $\Phi_g \leftarrow \text{EXPANDMICROPHASES}(g, \pi)$ 
4    for each adjacent action pair  $(a_{\text{cur}}, a_{\text{nxt}})$  in  $g$  do
5      if  $\text{ADMISSION}(a_{\text{cur}}, a_{\text{nxt}}, \pi)$  passes then
6        Shift only the prepare-like phases of  $a_{\text{nxt}}$  into the safe I/O window of  $a_{\text{cur}}$ .
7      end if
8      Keep all core effectful phases serialized.
9    end for
10   if  $\text{VALIDATE}(\Phi_g, \pi)$  passes then
11     Accept the refined candidate action group and add its overlay to  $\mathcal{P}$ .
12   else
13     Roll back  $g$  to its original candidate action group.
14   end if
15 end for
16 return  $\mathcal{P}$ 

```

has entered an I/O stage, such as after a request has been sent and is waiting for a response, or when BLE is in the transfer/settle stage. We never execute two core phases concurrently. Similarly, we do not allow the next action's core send/transfer phase to cross the completion boundary of the current action. This reordering must satisfy the constraint rules, e.g., it must not bypass backoff, violate BLE slot constraints, or break same-device/same-session constraints. Moreover, the overlap does not change the internal MSSU order within each action. The overall procedure is summarized by the pseudo-code in Algorithm 2.

3.5 Equivalence Validator

Although we only optimize the execution flow of primitives sliced from the original routine code, such orchestration may still introduce semantic shifts. Thus, we develop an equivalence validator to determine whether the optimized routine code remains semantically equivalent to the ground truth. Since the orchestrator features both macro-level and micro-level schedulers, we accordingly design the equivalence validator at two levels, as shown in Figure 8. The micro-level validator checks whether refinements introduced by the micro-level scheduler violate the original routine semantics, while the macro-level validator compares the observable traces and final outputs of the ground-truth routine and the optimized routine to determine equivalence.

3.5.1 Micro-level Validator. This validator focuses only on orchestration within each control-modality group. For each group, which serves as the optimization scope of the micro-level scheduler, we compare its optimized execution flow against the version before optimization, and apply lightweight rule-based checks to determine whether the following properties hold:

- The completion order of operations should remain equivalent.
- The relative order of micro-phases should remain equivalent.

Beyond these general logical requirements, control modality-specific rules are also checked. For example, BLE transfers must remain serialized, and API request sends must respect their serialization or rate-limit constraints. If any check fails, the validator rolls back the execution flow of the current control group to its original version enforced by the macro-level scheduler.

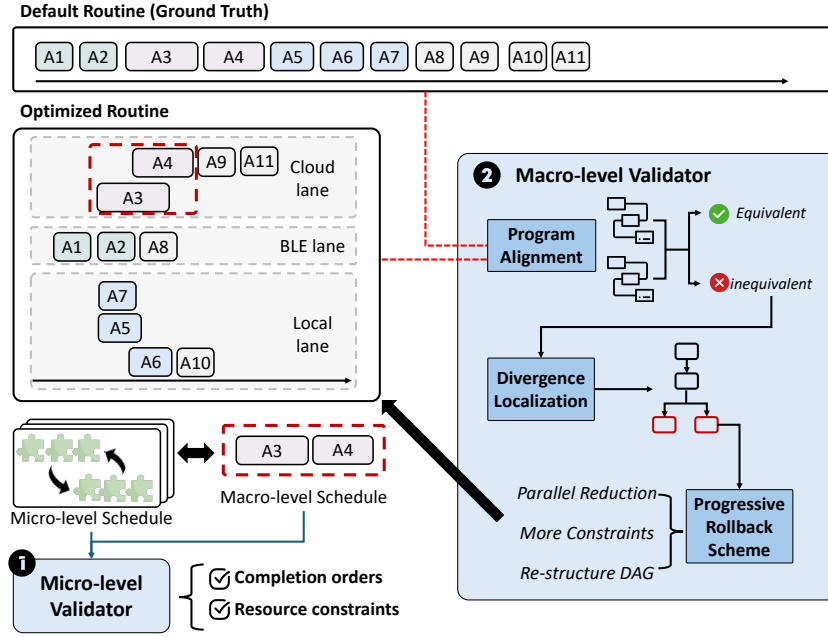


Fig. 8. Equivalence Validator Overview. We first apply the micro-level validator to check whether the micro-level schedule preserves the completion order specified by the macro-level schedule and does not violate resource constraints. Then, the macro-level validator performs program alignment between the optimized routine and the default routine to determine equivalence. If inequivalence is detected, it further applies divergence localization to guide the progressive rollback scheme in revising the optimized routine.

3.5.2 Macro-level Validator. The macro-level validator audits the macro-level execution flow. We design it to determine the equivalence between the optimized routine and the ground-truth routine, and to identify the source of divergence when inequivalence is detected. This validator draws inspiration from program-alignment techniques [10, 23], while adapting the techniques to the domain of IoT automation routines.

Validation Scope: The Validator aims to ensure that the original and optimized code preserve equivalent behavior observable to the automation platform, physical devices, and end users, rather than proving complete implementation equivalence between them. For example, suppose the original routine turns on a light, sets its brightness to 125, publishes the resulting state to Home Assistant, and releases the BLE connection. We consider the optimized routine observationally equivalent if it produces the same device state (the light is on with brightness 125), platform-visible state updates, exception behavior, and resource-protocol outcomes. This validation scope is consistent with our system objective: FloWeaver may change the internal execution order, but it must preserve the routine behavior observable to users, physical devices, and the automation platform.

Validation Workflow: The Validator first constructs replay scenarios through automatic enumeration. Each scenario fixes the routine inputs, initial device and platform states, environmental outcomes, resource budgets, and relevant concurrent execution orders under which the original and optimized routines are compared. Numeric values are selected according to their valid ranges and platform-defined precision. For a threshold predicate, the Validator enumerates representative values immediately below, at, and above the boundary. Initial device states and external API or device outcomes are enumerated according to the contracts in the platform profile. For concurrent execution, the Validator enumerates semantically distinct relative orders that satisfy the happens-before and

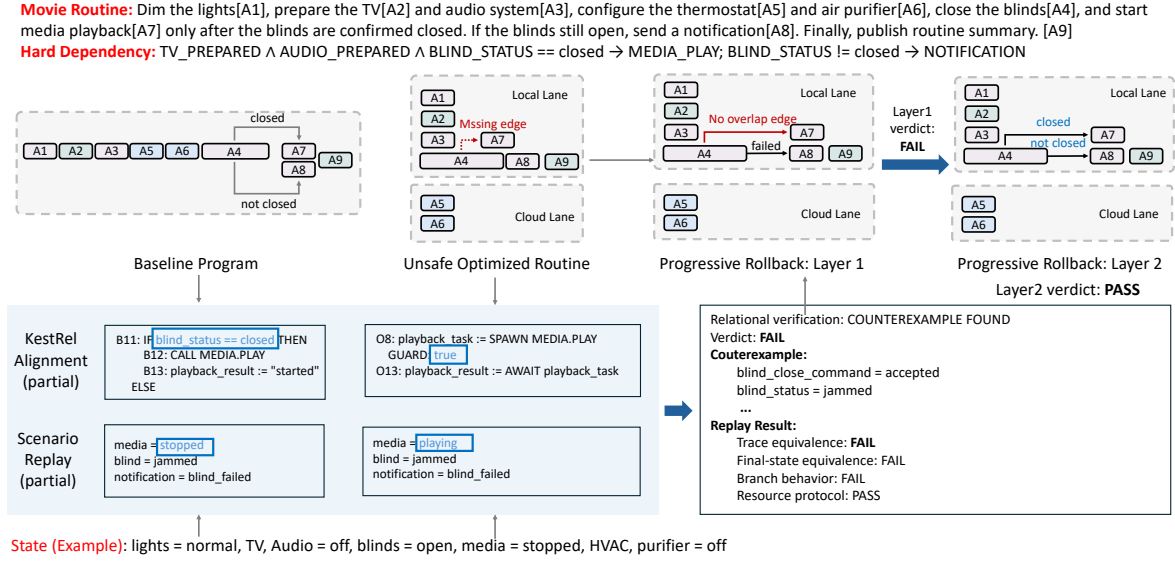


Fig. 9. An example of routine macro-level validation and rollback. The initial unsafe routine omits the hard dependency between A4 and A7. KestRel alignment and scenario replay detect the mismatch and generate a counterexample. Layer 1 only reduces parallelism and still fails validation. Layer 2 uses the counterexample to restore the conditional hard edge from A4 to A7, after which the routine passes validation.

resource constraints within the configured bounds. Figure 8 and Figure 9 presents an overview and representative working example of the Validator. For each scenario, the Macro-Validator performs two complementary checks: a KestRel-inspired relational alignment of the original and optimized program representations [10], and an equivalence comparison of their replay traces. A pass indicates that no modeled difference in observable behavior is found under that scenario. The process is repeated for every generated scenario. If validation fails, the Validator applies a procedure inspired by unified-and-split execution [23]: aligned regions are processed together, while suspicious or unmatched regions are explored separately to generate a concrete counterexample for the subsequent rollback mechanism. Figure 9 illustrates this process using the Movie Routine. The initially generated Unsafe Optimized Routine omits the control dependency between closing and verifying the blinds (A4) and starting media playback (A7). Consequently, its DAG lacks the conditional hard edge $A4[status = closed] \rightarrow A7$. The Validator compares the unsafe routine with the original routine through KestRel-inspired program alignment and replay-trace alignment. Both checks identify inconsistent observable behavior, producing a failed validation result and a counterexample that localizes the missing dependency. We provide more details about the validation scope in Appendix C.

3.5.3 Progressive Rollback Scheme. When the Validator detects inequivalence, it enters a three-layer progressive rollback process. Each layer applies a stronger repair while preserving as much latency reduction as possible. The process is fully automatic and requires no manual intervention.

- **Layer 1:** the Validator revises the unsafe optimized routine by reducing parallelism or disabling session reuse and batching. For an ordering or resource conflict, FloWeaver applies NO_OVERLAP to the affected protocol group. For MIN_GAP, FloWeaver uses the interval specified in the platform profile. If no numeric interval is available, it falls back to the original serial order. These changes are local and attempt to resolve the failure

with minimal rollback. In the example in Figure 9, Layer 1 only reduces parallelism, but the routine still fails validation because the required condition is missing.

- Layer 2: FloWeaver uses the counterexample to locate the cause and adjusts the relevant constraints. If the counterexample implicates a protocol optimization rule, that rule is disabled. If it implicates a semantic rule, FloWeaver adds a stricter guard. In Figure 9, Layer 2 uses the culprit actions identified by the counterexample to restore the missing conditional hard dependency $A4[\text{status} = \text{closed}] \rightarrow A7$. The repaired routine then passes validation.
- Layer 3: the Validator applies broader changes to the DAG. For `order_violation` or `missing_prerequisite` errors, FloWeaver maps the implicated actions back to the original routine, recovers their relative order, and adds scoped `SOFT_ORDER` constraints between the corresponding MSSUs.

After each layer, FloWeaver reschedules the routine and runs the Validator again. The process stops when validation succeeds or when the routine still fails after Layer 3. Only a validated plan is used to generate the final optimized code; otherwise, the routine is reported as failed.

3.6 System Automation Boundary and Portability

Automation Boundary. Based on the preceding description of FloWeaver’s design, we summarize the system’s automation boundaries. As detailed in §3.2, before routine optimization begins, the official documentation and code repositories of each target platform must be provided once. An LLM-assisted process then analyzes these resources offline to construct a platform-specific profile. Unless the underlying platform framework undergoes substantial changes, this profile can be reused by every routine to support subsequent stages. Specifically, routine parsing, contextual-anchor identification, MSSU extraction, DAG building, scheduling, validation, and rollback all refer the platform profile and proceed automatically. Manual intervention is limited to optionally auditing the platform profile, implement a platform-specific routine interpreter and configuring deployment-specific resource parameters, such as BLE concurrency limits and cloud-rate budgets. These parameters characterize the target environment and need to be configured only once. FloWeaver does not require manual annotation or constraint injection for individual routines.

Portability. The optimization core of FloWeaver is platform-independent. Once a routine and its associated platform knowledge have been represented through the action-profile and platform-profile abstractions, MSSU extraction, dependency-graph construction, scheduling, validation, and rollback can all reuse the same core algorithms. Thus, porting FloWeaver to another automation platform involves four steps: (a) implementing a platform-specific routine interpreter/parser, which our experience suggests requires modest engineering effort, particularly with assistance from code LLMs; (b) collecting the platform documentation and code repositories to construct a platform profile, followed by optional expert auditing; (c) configuring platform-specific resource budgets; and (d) implementing platform-specific execution and observation hooks for validation. These components are constructed once for each platform and subsequently reused across all routines on that platform. Owing to space constraints, we leave an empirical evaluation on additional platforms to future work.

4 Evaluation

In this section, we conduct a detailed evaluation of FloWeaver. We first introduce the routine datasets and ground-truth routine code used in our evaluation. We then describe the evaluation metrics, baseline methods, and deployment configuration. Finally, we present the experimental results to show the overall system performance. Our evaluation aims to answer the following research questions:

RQ1: What is the overall success rate of FloWeaver in producing optimized code that runs correctly and delivers measurable latency improvements?

RQ2: How much can FloWeaver reduce the average latency of real-world routines?

Table 2. Physical devices used in our evaluation.

Vendor	Device Type	Control Modality
Airthings	Thermostat, Air quality detector	BLE
Blink	Camera	Cloud
deCONZ	Light	Local
ecobee	Thermostat	Cloud
ESPHome	Thermostat, Motion sensor, Light	Local
Govee	Light, Plug	BLE
Hue	Light	Local
Nanoleaf	Light panels	Local
Netatmo	CO ₂ monitor, Motion sensor	Cloud
Philips	TV	Local
Roborock	Robot vacuum	Cloud
SmartThings	Door lock, Light, Media Player	Cloud
SwitchBot	Curtain, Motion Sensor	BLE
TP-Link	Light, Door bell	Local
Tuya	Light, Door bell, Motion sensor, AC	Cloud

RQ3: Does the Equivalence Validator reliably accept semantics-preserving optimizations and reject semantics-breaking ones?

RQ4: How often does FloWeaver trigger the rollback scheme in the Equivalence Validator, and how deep do the rollbacks go?

RQ5: How long does FloWeaver take to generate optimized routine code?

RQ6: How does each core component of FloWeaver contribute to preserving correctness and reducing latency?

4.1 Evaluation Dataset

Routine Specification Dataset. We consider two representative smart-environment settings that reflect different routine demands: a daily smart-home setting and a larger-scale setting, such as a smart office or smart industrial environment, with longer routines. For the *smart-home setting*, we randomly sample 50 routines from IFTTT [17], a leading and widely used open community that hosts thousands of official and user-contributed routines that are actively deployed and frequently executed. These routines involve one or more of the targeted control-modality groups: BLE, Cloud API, and Local Gateway. For the *long-chain routine setting*, we triple the number of devices in each routine from the smart-home dataset to mimic longer execution chains. The full routine datasets are provided in Appendix A.

Routine-code dataset. Table 2 lists the devices used in our evaluation, covering 15 vendors. These devices span multiple brands that are officially supported by Home Assistant and include top-selling products on Amazon. We treat the corresponding implementations in the official Home Assistant repository as the ground-truth code for realizing routine semantics.

4.2 Evaluation Metric

We evaluate FloWeaver from multiple perspectives. For clarity, we introduce the following notation. Let the routine dataset be $\mathcal{X}$, and let $\mathcal{X}_{\text{pass}} \subseteq \mathcal{X}$ denote the subset of samples whose optimized version pass Equivalence Validation. For each sample $x \in \mathcal{X}$, we record:

- (i) routine execution times: $t_{\text{start}}(x)$, the time when the (baseline or optimized) code begins executing, and $t_{\text{done}}(x)$, the time when the last device action in the routine completes;
- (ii) optimization processing times: $T_{\text{sys_start}}(x)$, when FloWeaver starts processing x , and $T_{\text{sys_end}}(x)$, when FloWeaver outputs a final optimized implementation that passes the Equivalence Validator. We define the end-to-end optimization runtime as $T_{\text{opt}}(x) = T_{\text{sys_end}}(x) - T_{\text{sys_start}}(x)$.

For rollback-related metrics, let $d(x)$ denote the maximum rollback level reached for x before the system either passes or terminates. Based on these definitions, our evaluation metrics cover the following dimensions:

- **Pass Rate (PR).** The probability that FloWeaver produces a final optimized code that passes Equivalence Validation.

$$\text{PR} = \frac{1}{|\mathcal{X}|} \sum_{x \in \mathcal{X}} \mathbf{1}_{\text{pass}}(x).$$

- **Fail Rate (FailR).** A task is considered failed if the optimization never passes Equivalence Validation.

$$\text{FailR} = 1 - \text{PR}.$$

- **Rollback Rate (RR).** The probability that an optimization enters the k -level rollback loop in Equivalence Validation.

$$\text{RR}_k = \frac{1}{|\mathcal{X}|} \sum_{x \in \mathcal{X}} \mathbf{1}_{L_k}(x), \quad k \in \{1, 2, 3\}.$$

- **Mean Rollback Depth (MRD).** The average maximum rollback depth reached before the system either passes or terminates.

$$\text{MRD} = \frac{1}{|\mathcal{X}|} \sum_{x \in \mathcal{X}} d(x), \quad d(x) \in \{0, 1, 2, 3\}$$

- **Average end-to-end routine latency (Latency).** The mean end-to-end latency of the executed routine over the set of samples that pass validation.

$$\text{Latency} = \frac{1}{|\mathcal{X}_{\text{pass}}|} \sum_{x \in \mathcal{X}_{\text{pass}}} L_{\text{e2e}}(x), \quad L_{\text{e2e}}(x) = t_{\text{done}}(x) - t_{\text{start}}(x).$$

- **Average runtime of optimization (ART).** The mean time FloWeaver spends to produce an optimized routine (among passed samples).

$$\text{ART} = \frac{1}{|\mathcal{X}_{\text{pass}}|} \sum_{x \in \mathcal{X}_{\text{pass}}} T_{\text{opt}}(x).$$

4.3 Baselines

To the best of our knowledge, FloWeaver is the first system to support validation-guided synthesis of low-latency execution flows for cross-device automation. Because no directly comparable baseline exists, we select established scheduling algorithms that can operate on the action-level DAG. Moreover, because our system ultimately transforms code, we include an LLM-based approach as a strong state-of-the-art baseline for general code optimization. Accordingly, we evaluate FloWeaver against the following four baselines.

(1) *HA-native baseline.* Home Assistant executes each routine according to its default runtime semantics and explicit action order.

(2) *HEFT scheduling baseline* [36]. HEFT operates on a basic action-level DAG that captures explicit routine dependencies, evident same-device conflicts, targeted write-back ordering constraints, and coarse-grained concurrency budgets for different control modalities. It treats each action as an indivisible unit and performs

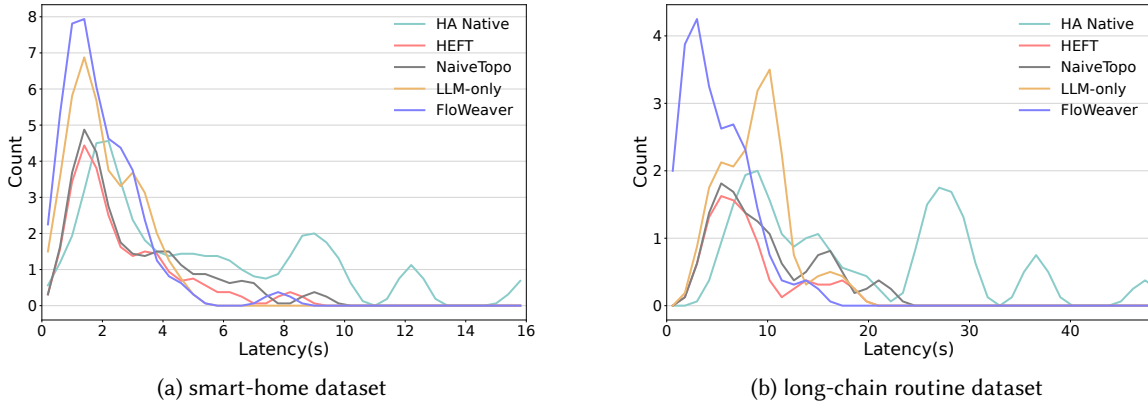


Fig. 10. Latency distribution of optimized routines in the smart-home and long-chain routine datasets.

latency-aware global scheduling. Specifically, HEFT computes the upward rank of each action, which estimates its remaining cost along the longest downstream path, and prioritizes actions in descending order of their upward ranks.

(3) *NaiveTopo-Parallel baseline*. This baseline uses the same input as HEFT but adopts a different scheduling policy. Specifically, at each step, it examines the topological frontier of the action-level DAG and assigns eligible actions to parallel execution batches according to their control modalities, provided that they exhibit no evident same-device conflicts and do not exceed the concurrency budgets.

(4) *LLM-only baseline*. An LLM is prompted to inspect the routine implementation and directly generate optimized code. We use Claude 4.6 Sonnet [2], the strongest general-purpose model available to us.

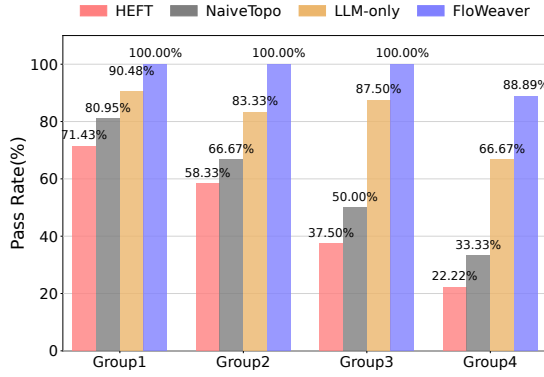
All experiments ran on one PC with an AMD 7950X CPU, an NVIDIA RTX 4090 GPU, 64 GB RAM, a 1 TB SSD, and Ubuntu 24.04. FloWeaver, Home Assistant, and all routines ran on this machine, where routine execution latency was measured from dispatch to the last required observable event. The PC used a BLE 5.0 USB adapter, 1-Gbps Ethernet, a Wi-Fi 6 local network, and a fixed broadband connection for Cloud APIs.

In the following, we present the evaluation results for RQ1–RQ6.

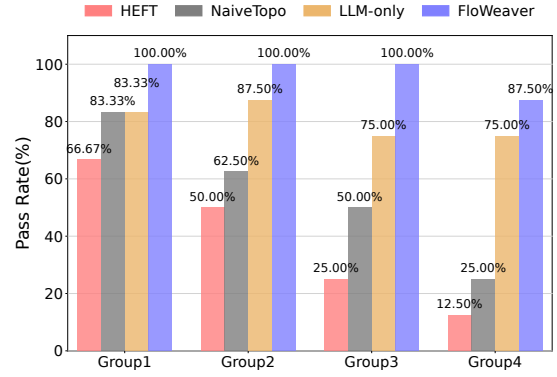
4.4 Overall pass-rate and latency reduction performance (RQ1, RQ2)

We first compare FloWeaver against the baselines on two routine datasets in terms of pass rate and average latency. Fig. 10 reports the latency distributions across routines. To present results across different complexity levels, we use the latency, measured by HA-native baseline, as a proxy for routine-chain complexity. For the smart-home dataset, we partition routines into four groups according to HA-native latency: $(0, 3]$ s, $(3, 6]$ s, $(6, 9]$ s, and $(9, \infty)$ s. For the long-chain routine dataset, we apply the same strategy with wider bins: $(0, 9]$ s, $(9, 18]$ s, $(18, 27]$ s, and $(27, \infty)$ s.

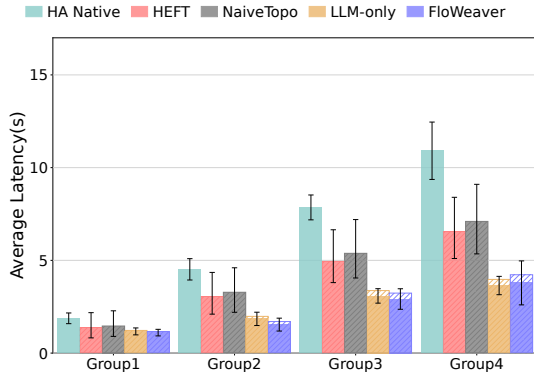
Observation 1: FloWeaver maintains a high pass rate on both routine datasets (RQ1). As shown in Fig. 11(a-b), FloWeaver achieves 100% pass rate for the first three groups in both datasets. For the most complex group (Group 4), FloWeaver attains a pass rate of 88.89% on the smart-home dataset and 87.5% on the long-chain routine dataset. This suggests that FloWeaver produces a consistently high validation pass rate across most routines. However, LLM-only baseline never reaches 100% pass rate. For the smart-home dataset, its pass rate is at least 83.3% for the first three groups, while for long-chain routines, its pass rate is at least 75% for the first three groups. For Group 4, LLM-only drops to 66.67% (smart-home) and 75% (long-chain), which is 22.22% and 12.5%



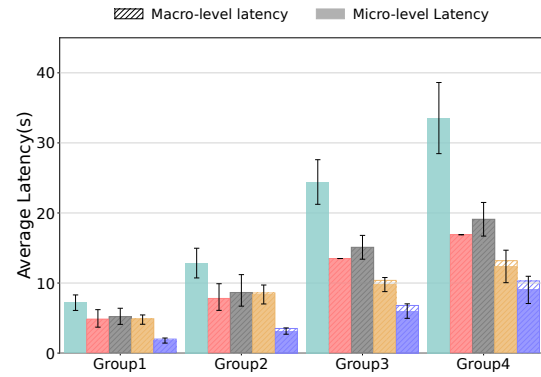
(a) Pass Rate in smart-home routine dataset.



(b) Pass Rate in long-chain routine dataset.



(c) Average Latency in smart-home routine dataset.



(d) Average Latency in long-chain routine dataset.

Fig. 11. Pass Rate and Average Latency performance.

lower than FloWeaver, respectively. This indicates that even state-of-the-art code LLMs struggle to perform reliable latency-oriented routine reordering without careful system design and distilled domain knowledge. In terms of NaiveTopo-Parallel and HEFT, the former consistently outperforms the latter in pass rate, while both degrade as routine complexity increases. From Group 1 to Group 4, NaiveTopo-Parallel drops from 80.95% to 33.33% on the smart-home dataset and from 83.33% to 25.00% on the long-chain dataset; HEFT drops more sharply, from 71.43% to 22.22% and from 66.67% to 12.50%, respectively. NaiveTopo-Parallel applies stable topological batching and waits for each batch to complete, whereas HEFT uses upward-rank and earliest-finish scheduling to overlap ready actions more aggressively. This gives HEFT more latency-reduction opportunities but also exposes more hidden lifecycle and ordering constraints that are absent from the action-level DAG so as to reduce pass rate.

Observation 2: FloWeaver substantially reduces routine latency, with larger gains on longer chains (RQ2). Fig. 11(c-d) reports average latency results, where hatched bars represent the macro schedule latency and solid bars represent the micro-level schedule latency. HEFT and NaiveTopo-Parallel only produce action-level schedule, thus, their macro-level and final latencies are identical. Both FloWeaver and the LLM-only baseline support optimization at the macro and micro levels.

Among validated smart-home routines, FloWeaver reduces latency by 41.12% relative to the HA-native baseline in Group 1, and by 62.88%–66% in Groups 2–4. This demonstrates that FloWeaver delivers substantial latency reductions. LLM-only is close to FloWeaver on the easier groups: it reduces latency by 37.62%, 59.15%, and 60.75% in Groups 1–3, trailing FloWeaver by 2.13%–6.85%. In Group 4, LLM-only achieves a 67.01% reduction relative to HA-native; however, since its pass rate in Group 4 is lower than FloWeaver, this comparison is less direct. For four groups, NaiveTopo-Parallel reduces latency by 19.78%–34.56% relative to the HA-native baseline, with average latency increasing from 1.46s to 7.10s. HEFT reduces latency by 21.14%–38.79% relative to the HA-native baseline, with average latency increasing from 1.38s to 6.55s. Both remain behind FloWeaver because they lack MSSU-level modeling, fine-grained lifecycle constraints, micro-level scheduling.

Among validated long-chain routines, the performance gain of FloWeaver becomes more pronounced as routines become longer and more complex. FloWeaver consistently achieves a 73.11%–75.53% latency reduction relative to the HA-native baseline across all four groups. In Groups 3 and 4, FloWeaver reduces latency by 18.425s and 23.744s, respectively, differences that are clearly perceptible to users. This again confirms that FloWeaver achieves the more gains on longer chains. LLM-only method reduces latency by 33.56%–34.9% relative to the HA-native baseline in Groups 1–2 and by 59.95%–62.25% in Groups 3–4. Relative to the HA-native baseline, NaiveTopo-Parallel reduces latency by 26.24%–39.94% on the long-chain dataset, with its average latency increases from 5.20s in Group 1 to 19.10s in Group 4; HEFT achieves latency reductions of 29.71%–45.48%, with its accepted latency increasing from 4.85s to 16.90s. The widening latency gap between these baselines and both LLM-only and FloWeaver indicates that coarse-grained action-level batching scales less effectively as routine chains grow longer.

Finally, we compare macro versus micro-level schedules. On the smart-home dataset, micro refinement reduces latency by an additional 0.059–0.438s for FloWeaver and 0.051–0.33s for LLM-only. On the long-chain dataset, the additional reduction is 0.208–1.274s for FloWeaver and 0.119–0.822s for LLM-only. Neither HEFT nor NaiveTopo-Parallel support micro-level schedule. Overall, in our evaluation, micro-level refinement provides consistent incremental gains, while the majority of latency reduction is captured by the macro-level schedule.

4.5 Performance of Equivalence Validator (RQ3–RQ4)

To evaluate the reliability of FloWeaver’s equivalence validator, we conduct two complementary correctness assessments on the resulting optimized routine code. First, an independent audit determines whether the optimized code preserves externally observable device and platform behavior. Second, a mutation-injected test evaluates whether the validator successfully rejects optimized routines that have been deliberately rendered incorrect.

4.5.1 Independent Audit (RQ3). We select 30 pairs of validator-accepted optimized routines and their corresponding baseline versions, comprising 20 smart-home routines and 10 long-chain routines. The sample is stratified by HA-native latency group, control modality, optimization pattern, and rollback depth. It encompasses BLE-only, local-only, cloud-only, and mixed-modality routines, including routines accepted either directly or after rollback. Each routine pair is executed five times, yielding 150 paired baseline–optimized executions.

Before each execution, we reset and verify the initial device and platform states. We then execute the baseline and optimized routines under identical scenario inputs and compare their final device states, platform-visible states, required intermediate events, exception categories, and observable lifecycle and resource events. Whenever possible, we query vendor APIs or direct device interfaces to obtain device-side final states; otherwise, we use Home Assistant state and event logs as the platform-visible oracle. The audited fields include light power state, brightness, color temperature, curtain position, switch state, thermostat mode and setpoint, camera arm/disarm state, vacuum state, sensor readings, Home Assistant state write-backs, notifications, and lifecycle events such as BLE connection/disconnection and subscription/unsubscription.

Table 3. Independent audit on optimized routine code.

Metric	smart-home	long-chain	Overall
Dev-State Match	100.0%	100.0%	100.0%
Plat-State Match	100.0%	100.0%	100.0%
Run-State Match	100.0%	100.0%	100.0%

Table 4. Results of mutation-injected test. MRR denotes mutation rejection rate, and FPR denotes false-positive rate.

Dataset	# Routine.	# Mutation.	MRR	FPR
Smart-home	20	105	100.0%	0.0%
Long-chain	10	45	100.0%	0.0%
Overall	30	150	100.0%	0.0%

Table 3 summarizes the results. *Dev-State Match* denotes device-side final-state equivalence, *Plat-State Match* denotes platform-visible state equivalence, and *Run-State Match* denotes whether all final-state fields match within one paired execution. The optimized code achieves a 100% audit pass rate. We observe no mismatches in final states, required events, exception behavior, or lifecycle behavior. Detailed state-field comparisons, event-level checks, and results for all audited cases are provided in Appendix D.

4.5.2 Mutation-Injected Test (RQ3). We further examine whether the validator rejects optimized routines that have been intentionally rendered incorrect. This experiment directly assesses the validator’s false-positive risk, where a false positive occurs when a mutated optimized routine is accepted without producing either a rejection signal or a counterexample. We use the routine set from the preceding independent audit and inject semantics-breaking mutations into the validator-accepted optimized routines. Rather than introducing arbitrary code corruption, we model plausible but unsafe transformations that may arise during execution-flow optimization, as shown in Table 5. For each mutation, we compare the baseline routine against the mutated optimized routine. A mutation is classified as a false positive only if the validator accepts the mutated routine without issuing a rejection signal or producing a counterexample.

We report the mutation rejection rate (MRR), defined as the fraction of injected mutations rejected by the validator, and the false-positive rate (FPR), defined as the fraction of mutations incorrectly accepted. As shown in Table 4, the validator rejects all injected mutations, achieving an MRR of 100.0% and an FPR of 0.0% on both datasets. Table 5 further demonstrates that this result holds across all mutation categories. Thus, within the evaluated mutation suite, the validator does not incorrectly accept any representative semantics-breaking optimization.

Together with the independent audit, these results demonstrate the reliability of FloWeaver’s equivalence validator and the correctness of the optimized routines it accepts.

4.5.3 Progressive Rollback Analysis (RQ4). Here we examine how FloWeaver and the LLM-only baseline behave inside the Equivalence Validator’s progressive rollback process. In our evaluation, both HEFT and NaiveTopo-Parallel baselines employ the Equivalence Validator in check-only mode: any counterexample is treated as a failure, and neither baseline invokes the rollback mechanism. The results are shown in Fig.12.

Observation 3: Long-chain routines trigger rollbacks more frequently and tend to roll back deeper. (RQ4) As shown in Fig.12(a), on the smart-home dataset, 62.00% of FloWeaver routines enter at least rollback

Table 5. Mutation results by mutation type. Each type represents a semantic-breaking unsafe optimization pattern.

Mutation Type	Cases	MRR	observed FPR
M1 Dependent-action swap	20	100.0%	0.0%
M2 Premature state writeback	20	100.0%	0.0%
M3 Missing refresh/status query	20	100.0%	0.0%
M4 BLE serialization violation	12	100.0%	0.0%
M5 Cloud rate/session violation	18	100.0%	0.0%
M6 Wrong batching/writeback merge	20	100.0%	0.0%
M7 Same-device ordering violation	20	100.0%	0.0%
M8 Cleanup/lifecycle violation	20	100.0%	0.0%
Overall	150	100.0%	0.0%

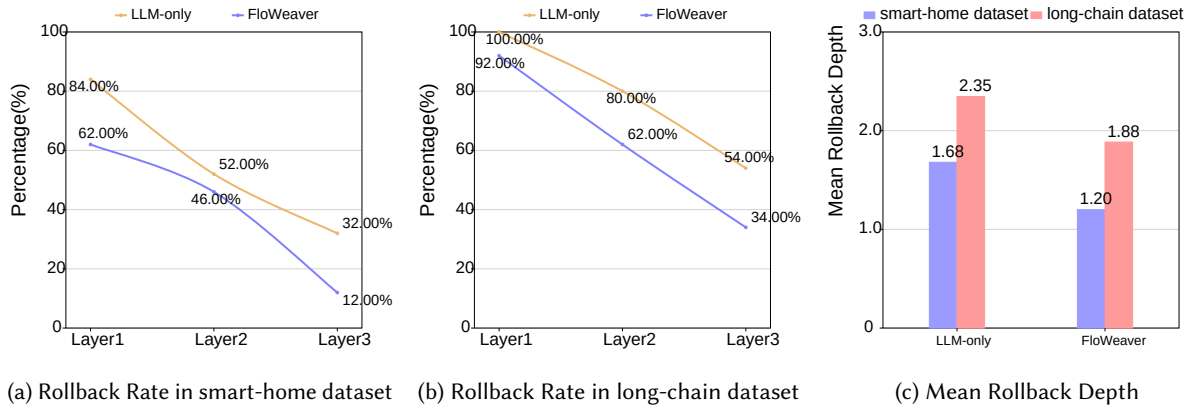


Fig. 12. Rollback Rate and Mean Rollback Depth for rollback-enabled methods in the smart-home and long-chain datasets.

Layer 1, 46.00% reach Layer 2, and 12.00% reach Layer 3. The corresponding rates for LLM-only are 84.00%, 52.00%, and 32.00%, respectively.

On the long-chain dataset, 92.00% of FloWeaver routines enter Layer 1, 62.00% reach Layer 2, and 34.00% reach Layer 3. For LLM-only, these rates increase to 100.00%, 80.00%, and 54.00%. Long-chain routines therefore trigger more frequent and deeper rollback because they contain more dependencies, larger graphs, and tighter cross-protocol interactions. Fig. 12(c) further shows that FloWeaver has a lower mean rollback depth than LLM-only: 1.20 versus 1.68 on the smart-home dataset, and 1.88 versus 2.35 on the long-chain dataset. This suggests that FloWeaver produces higher-quality initial schedules and requires less aggressive rollback to pass validation. These results show that FloWeaver achieves a higher pass rate and a lower rollback burden than LLM-only. This highlights the benefit of combining MSSU-level modeling and lifecycle constraints with progressive rollback and constraint tightening.

4.6 Optimization Efficiency (RQ5)

In this section, we evaluate the runtime efficiency of FloWeaver, LLM-only, NaiveTopo-Parallel, and HEFT. We report end-to-end runtime for all methods and per-routine token consumption for LLM-only. FloWeaver has no

Table 6. Runtime and per-routine token consumption. LLM-only token statistics are measured per routine. FloWeaver requires neither an online LLM stage nor per-routine LLM invocations. Constructing the Home Assistant platform profile requires a single invocation of Qwen3-14B and consumes 1,808 input and output tokens in total. This is a one-time offline cost, as the resulting profile is reused across all routines.

Configuration		Runtime (second)			Per-Routine Token Consumption		
Dataset	Method	Average	Max	Min	Average	Max	Min
smart-home routines	HEFT	7.70	11.50	4.10			
	NaiveTopo	6.43	10.20	3.80			
	LLM-only	32.61	47.82	14.43	28780	48606	15008
	FloWeaver	13.66	20.86	6.78	–	–	–
long-chain routines	HEFT	11.27	17.10	6.00			
	NaiveTopo	9.58	15.00	5.20			
	LLM-only	64.55	82.24	45.52	68117	94324	43996
	FloWeaver	17.75	23.88	9.89	–	–	–

online LLM stage; its only LLM invocation is the one-time offline construction of the platform profile. Results are summarized in Table 6.

Observation 4: FloWeaver is efficient in routine code optimization (RQ5). HEFT takes 7.70s on average for smart-home routines and 11.27s on average for long-chain routines. NaiveTopo-Parallel takes 6.43s and 9.58s on two datasets, respectively. It is moderately faster than HEFT because its greedy batching avoids upward-rank computation and associated resource assignment overhead. FloWeaver takes 13.66 s on average on the smart-home dataset, and 17.75s on average on the long-chain dataset, which is still a practical runtime range for routine optimization. Compared with the NaiveTopo-Parallel and HEFT baselines, FloWeaver incurs additional runtime overhead for MSSU extraction, constraint-aware scheduling, equivalence validation, and progressive rollback, which collectively enable the stronger correctness guarantees and greater latency reductions reported in *Observations 1–2*. In contrast, the LLM-only baseline relies on a state-of-the-art long-context reasoning LLM, so its longer runtime is expected: it takes 32.61 s on average on the smart-home dataset and 64.55 s on the long-chain dataset, which introduces noticeable user waiting time. Overall, FloWeaver remains relatively efficient in wall-clock time, making it feasible to be a resident process to optimize new routines online, as indicated in Fig. 1(b). In terms of token consumption, neither HEFT nor NaiveTopo-Parallel employs an LLM. FloWeaver has no online LLM stage and therefore incurs no per-routine token cost. Similar to other long code generation tasks, LLM-only method consumes a large number of tokens: 28,780 tokens on average on the smart-home dataset and 68,117 tokens on the long-chain dataset. These results show that FloWeaver efficiently optimizes routine code, incurring acceptable generation overhead while producing latency-reduced code that preserves the original routine intent.

4.7 Ablation Study (RQ6)

We evaluate three ablated variants of FloWeaver:

- FloWeaver *w/o Rollback*: disables progressive rollback and validates only the initially optimized schedule.
- FloWeaver *w/o Micro*: disables the micro-level scheduler while retaining MSSU extraction, macro-level scheduling, validation, and rollback.
- FloWeaver *w/o MSSU*: eliminates MSSU decomposition and treats each action as the smallest scheduling unit while retaining action-level scheduling, validation, and rollback.

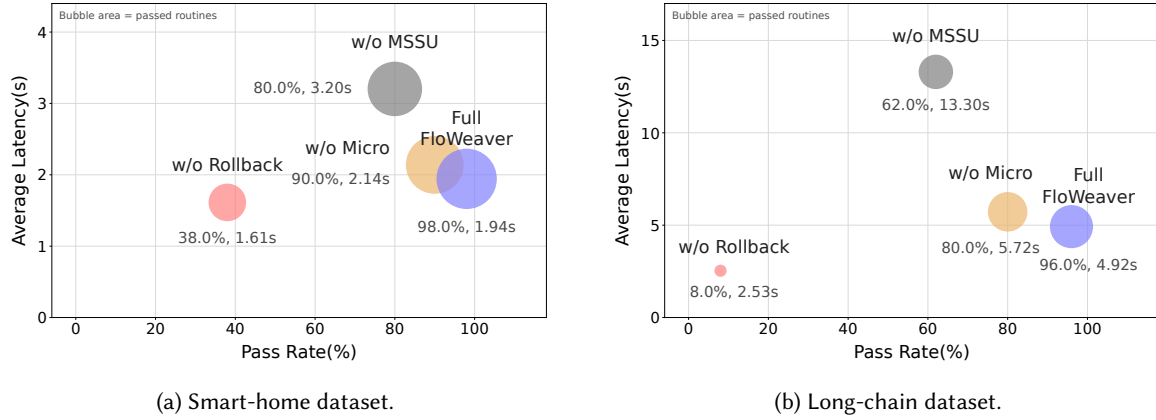


Fig. 13. Pass-rate and latency trade-offs of the ablation variants. Each bubble represents one configuration, and its area indicates the number of validation-passed routines. Latency is averaged only over passed routines.

For all variants, latency reductions are computed over routines that pass validation.

Observation 5: Rollback improves correctness recovery. As shown in Fig. 13, disabling rollback causes the largest pass-rate drop. On the smart-home dataset, w/o Rollback passes only 38.00% of routines, compared with 98.00% for full FloWeaver. On the long-chain dataset, its pass rate drops further to 8.00%, compared with 96.00% for full FloWeaver. This shows that the initial optimized plan is often too aggressive, especially for complex routines, and rollback is necessary to repair unsafe schedules. Although the passed w/o Rollback routines have low average latency, this is a survivor effect: latency is averaged only over the few aggressive schedules that pass validation, while most unsafe schedules are rejected.

Observation 6: Micro-level scheduling provides incremental latency gains. The w/o Micro variant maintains a relatively high pass rate but has higher average latency than full FloWeaver. On the smart-home dataset, its average latency is 2.14s, compared with 1.94s for full FloWeaver; on the long-chain dataset, the corresponding values are 5.72s and 4.92s. This indicates that macro-level scheduling captures most latency reduction, while micro-level scheduling provides stable additional gains by overlapping safe intra-action phases.

Observation 7: MSSU decomposition is important for complex routines. The w/o MSSU variant achieves an overall pass rate of 80.00% on the smart-home dataset and 62.0% on the long-chain dataset, both lower than full FloWeaver. Its average latency also increases from 1.94s to 3.20s on the smart-home dataset and from 4.92s to 13.30s on the long-chain dataset. This shows that action-level scheduling alone cannot expose sufficient safe optimization opportunities or model lifecycle-sensitive constraints at the appropriate granularity. Therefore, MSSU decomposition contributes to both optimization efficiency and correctness stability, particularly for long-chain routines.

Fig. 14 further explains the validation pressure behind these results. w/o Rollback has high diagnostic rollback demand, especially on long-chain routines, where 92.00% of cases require rollback and the mean depth reaches 2.77. w/o Micro has rollback statistics close to full FloWeaver, suggesting that most rollback pressure comes from macro-level ordering and resource constraints, while micro-level scheduling mainly affects latency. w/o MSSU has lower rollback depth than full FloWeaver, but this should not be interpreted as better stability. It reflects a coarser action-level search space with fewer fine-grained optimization attempts, which also leads to higher latency and lower pass rate on complex routines.

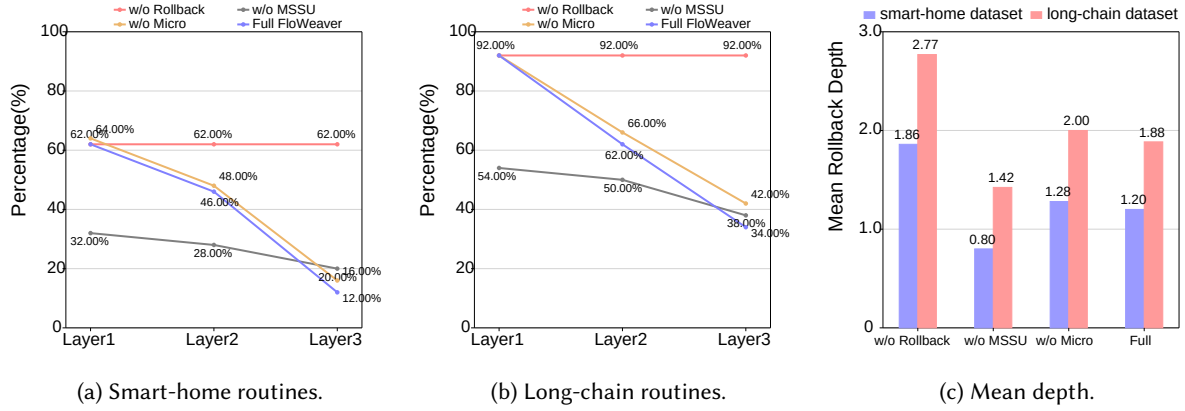


Fig. 14. Rollback rate and mean rollback depth for ablation variants. For w/o Rollback, rollback depth is diagnostic only because rollback is disabled.

Table 7. Runtime comparison for ablation configurations.

Configuration		Runtime (second)		
Dataset	Method	Average	Max	Min
smart-home routines	w/o Rollback	10.23	16.40	5.56
	w/o MSSU	10.80	17.60	5.15
	w/o Micro	12.43	18.98	6.17
	Full FloWeaver	13.66	20.86	6.78
long-chain routines	w/o Rollback	12.88	17.20	7.59
	w/o MSSU	14.19	19.91	7.52
	w/o Micro	16.15	21.73	9.00
	Full FloWeaver	17.75	23.88	9.89

Table 7 shows that removing modules reduces runtime, but the saved time comes with clear tradeoffs. w/o Rollback is faster but loses correctness recovery. w/o Micro is slightly faster but has higher final latency. w/o MSSU is faster than full FloWeaver, but its pass rate and latency degrade substantially. Overall, the ablation study shows that the three components play complementary roles: rollback recovers correctness, micro-level scheduling improves latency, and MSSU decomposition provides the right granularity for reliable and efficient optimization.

5 User Study

We design a controlled user study to addresses the following research question:

RQ7: Does the routine execution produced by FloWeaver provide a better perceived latency experience than the baselines?

Table 8. Demographic characteristics of the participants in the user study.

ID	Age	Education	Routine Usage Frequency
P1	25–34	Graduate	Slightly
P2	25–34	Graduate	Very
P3	18–24	Graduate	Moderately
P4	25–34	Bachelor's	Very
P5	35–44	High school or equivalent	Moderately
P6	18–24		Slightly
P7	18–24	Bachelor's	Very
P8	25–34	Bachelor's	Slightly
P09	25–34	Graduate	Moderately
P10	18–24	Bachelor's	Very
P11	35–44	Bachelor's	Moderately
P12	25–34	Graduate	Slightly
P13	18–24	Graduate	Moderately
P14	25–34	Bachelor's	Very
P15	35–44	Graduate	Slightly
P16	25–34	Bachelor's	Moderately

5.1 Study Design

5.1.1 Participants. We recruited N=16 participants (ages $M = 24.6$ years, $SD = 3.4$ years). Their demographic information is shown in Table 8. Participants were compensated at a rate of \$12 per hour. The study protocol was approved by the Institutional Review Board of our institution.

5.1.2 Methodology. We adopt a within-subject design and use a fixed set of routines across all participants to ensure comparability. The study includes ten routines: five smart-home routines and five long-chain routines. The smart-home subset covered different HA-native latency groups and cloud, local, and mixed cloud-local routines. The long-chain subset contained routines with 18-24 devices and represented longer execution chains. For each routine, every participant is presented with executable and validator-accepted outputs generated by five methods: HA-native, HEFT, NaiveTopo-Parallel, LLM-only, and FlowWeaver. In total, each participant evaluated 10 routines across five methods. To mitigate potential bias toward any method, we randomized the presentation order of the methods. Table 9 lists the ten presentation sequences used in the study. Across participants, each routine was evaluated under all ten sequences once. For each routine, every method appeared in each presentation position three or four times, and every directed adjacent method pair appeared two to four times. Eight participants completed the smart-home routines first, while the other eight completed the long-chain routines first.

5.1.3 Procedure. At the beginning of each trial, a facilitator explained the routine scenario and restored the devices and automation platform to the same initial state. The five routine implementations were then executed under identical scenario inputs in the assigned Williams-design order, with method identities concealed from participants. After observing each execution, participants immediately rated its perceived latency before proceeding to the next one. Device states were reset between executions whenever a routine modified persistent state. Participants were permitted to take short breaks between routines and a longer break between the two datasets. The study measured perceived execution latency and user acceptance, while functional correctness was controlled by including only implementations that had passed validation before the study.

Table 9. Williams sequences used to mitigate potential bias toward any method.

Sequence	Position 1	Position 2	Position 3	Position 4	Position 5
W01	HA-native	HEFT	FloWeaver	NaiveTopo	LLM-only
W02	HEFT	NaiveTopo	HA-native	LLM-only	FloWeaver
W03	NaiveTopo	LLM-only	HEFT	FloWeaver	HA-native
W04	LLM-only	FloWeaver	NaiveTopo	HA-native	HEFT
W05	FloWeaver	HA-native	LLM-only	HEFT	NaiveTopo
W06	LLM-only	NaiveTopo	FloWeaver	HEFT	HA-native
W07	FloWeaver	LLM-only	HA-native	NaiveTopo	HEFT
W08	HA-native	FloWeaver	HEFT	LLM-only	NaiveTopo
W09	HEFT	HA-native	NaiveTopo	FloWeaver	LLM-only
W10	NaiveTopo	HEFT	LLM-only	HA-native	FloWeaver

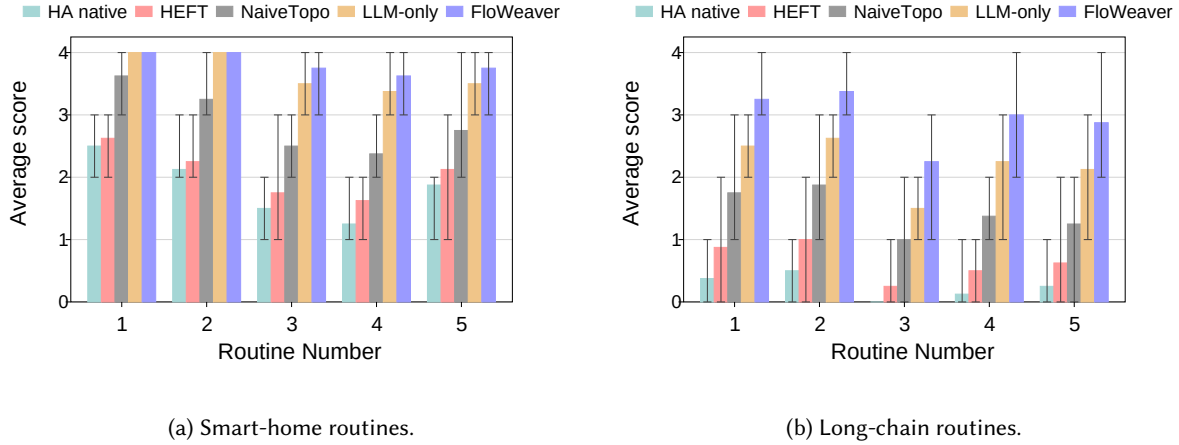


Fig. 15. Latency-acceptance scores for five methods, showing the mean, minimum, and maximum across 16 participants. A score of 0 denotes unacceptable latency, while 4 denotes near-instant execution.

5.1.4 Measures and Analysis. Our rating rubric follows established waiting-time perception thresholds [4]. Participants assigned an integer score from 0 to 4:

- **0:** unacceptable delay; the routine feels unusable;
- **1:** strongly disruptive delay;
- **2:** acceptable but clearly noticeable delay;
- **3:** responsive execution with minor waiting;
- **4:** near-instant execution.

The primary metric is the mean latency-acceptance score. We additionally report the proportion of ratings greater than or equal to 3, denoted as the acceptance rate.

5.2 Perceived Latency Results (RQ7)

Fig. 15 presents the per-routine mean together with the observed minimum–maximum range across participants. For the smart-home routines, FloWeaver achieves the highest aggregate mean score of 3.83, followed by LLM-only at 3.68, NaiveTopo at 2.90, HEFT at 2.08, and HA-native at 1.85. Both FloWeaver and LLM-only obtain an acceptance rate of 100.0%. Their mean scores reach 4.0 on the first two routines. NaiveTopo improves over HEFT and HA-native, but its acceptance rate is lower at 71.2%. These results suggest that advanced optimization provides a near-instant experience for most daily routines, while simple action-level optimization produces less consistent improvements.

The differences become larger for long-chain routines. FloWeaver obtains an aggregate mean score of 2.95 and an acceptance rate of 78.8%. LLM-only reaches a mean of 2.20 and an acceptance rate of 32.5%, while NaiveTopo reaches 1.45 and 2.5%, respectively. HA-native and HEFT receive mean scores of 0.25 and 0.65, with no rating at or above 3. On the most demanding long-chain routine, FloWeaver receives a mean score of 2.25, compared with 1.50 for LLM-only, 1.00 for NaiveTopo, 0.25 for HEFT, and 0 for HA-native. The observed pattern indicates that perceived latency differences increase with routine length. Simple scheduling provides limited improvement on long routines, while FloWeaver maintains substantially higher acceptance. The lower FloWeaver score on the most demanding routine also shows that long routines still remain noticeable after optimization.

6 Related Work

IoT routines and trigger–action programming. IoT routines, which can be represented as trigger–action programming, have been widely studied due to their popularity and potential logical complexity. However, most existing work in this area does not focus on accelerating routine execution. Some work focuses on program synthesis and repair. Work [40] allows end users to describe desired smart-home behaviors as properties, translates them into trigger–action programs, and repairs them when needed. Work [41] synthesizes trigger–action programs from temporal behaviors. Work [25] investigates how to automate the synthesis of device components to reduce developer effort and improve coverage. There are also studies [26, 33, 42] that focus on how users understand, express, and author routines in practice, emphasizing usability, mental models, and in-situ programming workflows. Work [14, 22] further introduces LLMs to assist in writing routine rules, improving authoring efficiency and accuracy. In addition, some studies focus on analyzing semantic constraints and repairing unsafe routines [7, 28, 31, 39]. For example, work [39] studies interaction vulnerabilities in home-automation rules. To the best of our knowledge, little prior work targets runtime acceleration of cross-device routines. FloWeaver is designed to fill this gap by reducing the end-to-end latency of user-specified routines, thereby improving user perceived experience.

Code Transformation and Acceleration. Automatically optimizing code to obtain additional beneficial properties is not a new concept. Work [20] develops a tool that automatically rewrites low-level assembly code to insert carefully designed checkpoint and restore operations, enabling Arduino programs to run on batteryless hardware. There are also extensive work on compiler optimization that transforms conventional programs into more efficient executions. Work [13] focuses on the most likely hot path and schedules operations across basic blocks to reduce execution time. Follow-up work extends this idea by organizing hot regions or overlapping loop iterations to improve throughput at a broader scope [16, 30, 38]. Recent work [24] leverages runtime profiling data to guide compiler decisions, such as code layout, branch optimization, inlining, and hot-path tuning. Overall, these techniques optimize programs at the level of basic blocks, loops, or compiler IR. However, they cannot be directly applied to the routine-code acceleration problem targeted in this work, because a critical point here is to well capture routine-level semantics and constraints, such as BLE connection and transfer constraints, Cloud API rate limits, platform session lifecycles, and observable state writebacks. Therefore, dedicated system design is needed to enable efficient and semantics-preserving routine acceleration.

Graph Representation for Optimization. Graph representations are widely used to characterize workloads and facilitate resource allocation, orchestration, and workflow optimization [1, 5, 11, 21, 34, 37]. For example, work [6] characterizes application workloads as DAGs and optimizes container resource management to meet SLOs. Work [9] models sequences of serverless functions and optimizes their execution to reduce latency caused by cascading cold starts. Inspired by these works, we also adopt a graph representation as the intermediate abstraction, where dependencies and constraints are directly encoded in the graph and optimization is performed over this representation.

7 Discussion and Limitation

We discuss the potential challenges associated with deploying FloWeaver in real-world environments, its current limitations, and promising directions for future research.

Practical Benefits for Home Automation Developers. For users, FloWeaver accelerates long-running automation routines, reducing perceived latency and enhancing the overall user experience. For developers, optimizing diverse routines typically entails substantial manual effort in code analysis, optimization planning, and implementation. FloWeaver automates this end-to-end process, thereby significantly reducing the development burden. In dynamic home environments, where devices are frequently added or removed and users continually adapt their routines to evolving needs and habits, manually devising tailored optimizations for every configuration is impractical. Even with advanced coding agents, repeatedly optimizing each configuration may incur substantial API calling costs. In contrast, FloWeaver operates persistently within the system, efficiently performing recurring optimization tasks with minimal human intervention and without incurring runtime LLM inference costs.

Scheduling-Constraint Scope and Completeness. FloWeaver distinguishes between two categories of constraints. (1) *Deployment-specific resource parameters*, such as BLE connection capacity and cloud rate limits, depend on the target environment and are specified in the platform configuration. Thus, the values used in our evaluation are not assumed to be universally applicable. (2) *semantic constraints*, including lifecycle and ordering relations, are encoded in a platform profile constructed from platform documentation, source code, etc. The generated profile is bounded by the available evidence and is therefore not assumed to be universally complete. Insufficiently supported rules are retained as soft constraints and applied conservatively, while validation and rollback mitigate the risk of unsafe optimizations. Future work could further improve constraint coverage through richer execution traces, more robust LLM-assisted discovery, and targeted expert review.

Validation Completeness. The guarantees provided by our validator are bounded by the generated replay scenarios. Although these scenarios are systematically constructed from routine semantics to cover diverse execution paths and achieve strong empirical results, they may not exercise every feasible behavior. Consequently, successful validation does not establish all-path equivalence. Future work could improve validation coverage through symbolic execution, coverage-guided scenario generation, richer real-world execution traces, etc.

Component-Level Evaluation. FloWeaver comprises multiple important stages, not all of which are evaluated independently. For example, because no manually annotated anchor/MSSU dataset is currently available, we do not conduct a standalone evaluation of Primitive Extractor. Its correctness is instead assessed indirectly through our end-to-end evaluation. In particular, the independent audit (§4.5.1) shows that all sampled routines optimized by FloWeaver preserve the audited device and platform behaviors. These results provide indirect evidence that MSSU extraction operates correctly for the evaluated routines, although they do not constitute a component-level guarantee. Constructing dedicated component-level benchmarks remain as future work.

8 Conclusion

This paper presents FloWeaver, a validation-guided synthesis system for low-latency execution flows in cross-device smart-environment automation. We show that existing automation platforms often execute user routines

as sequential chains of independently managed device operations, leading to noticeable delays and degraded user experience. To address this limitation, FloWeaver decomposes workflows into minimal self-contained semantic units, represents their semantic dependencies and system resource constraints through graph structures, and applies two-stage scheduling to synthesize latency-efficient execution flows. Across the four latency groups, FloWeaver achieves pass rates of 88.89%–100% on the smart-home dataset and 87.50%–100% on the long-chain dataset. Among accepted routines, its average latency reductions range from 41.12% to 66.00% and from 73.11% to 75.53%, respectively. FloWeaver provides the strongest combined pass-rate and latency performance compared with HA-native, HEFT, NaiveTopo-Parallel, and LLM-only. The ablation study demonstrates the contributions of MSSU decomposition, micro-level scheduling, and progressive rollback, while the N = 16 within-subject user study shows improved perceived responsiveness and acceptance. FloWeaver provides a practical optimization pipeline that can be integrated into automation platforms, extending routine composition toward semantics-preserving execution-flow optimization for more responsive smart-environment experiences.

Acknowledgments

We sincerely thank the anonymous AEs and reviewers for their valuable comments and suggestions. This work is supported by Guangdong General Higher Education Institution (Young Innovative Talents, No. 2025KQNCX113), Guangzhou-HKUST(GZ) Joint Funding Program (No. SL2024A03J01192), Guangzhou Municipal Science and Technology Project (Young Scholar Set-Sail Research, No. SL2024A04J01392), the Department of Education of Guangdong Province (Innovation Team, No. 2024KCXTD008), and Guangdong Provincial Key Lab of Integrated Communication, Sensing and Computation for Ubiquitous Internet of Things (No. 2023B1212010007).

References

- [1] Niket Agrawal, Jan Rellermeyer, and Aaron Yi Ding. 2019. IoT Resource-aware Orchestration Framework for Edge Computing. In *Proceedings of the 15th International Conference on Emerging Networking EXperiments and Technologies* (Orlando, FL, USA) (CoNEXT '19 Companion). Association for Computing Machinery, New York, NY, USA, 62–64. doi:10.1145/3360468.3368179
- [2] Anthropic. 2026. Claude 4.6. <https://www.anthropic.com/news/claude-sonnet-4-6>
- [3] Home Assistant. 2017. Home Assistant. <https://www.home-assistant.io/>, Accessed:2026-04-14.
- [4] Siddhartha Asthana, Pushpendra Singh, and Parul Gupta. 2015. Survival analysis: objective assessment of wait time in HCI. In *Proceedings of the 33rd annual acm conference on human factors in computing systems*. 367–376.
- [5] Luciano Baresi, Davide Yi Xian Hu, Giovanni Quattrocchi, and Luca Terracciano. 2024. NEPTUNE: A Comprehensive Framework for Managing Serverless Functions at the Edge. *ACM Trans. Auton. Adapt. Syst.* 19, 1, Article 7 (Feb. 2024), 32 pages. doi:10.1145/3634750
- [6] Vivek M. Bhasi, Jashwant Raj Gunasekaran, Prashanth Thinakaran, Cyan Subhra Mishra, Mahmut Taylan Kandemir, and Chita Das. 2021. Kraken: Adaptive Container Provisioning for Deploying Dynamic DAGs in Serverless Platforms. In *Proceedings of the ACM Symposium on Cloud Computing* (Seattle, WA, USA) (SoCC '21). Association for Computing Machinery, New York, NY, USA, 153–167. doi:10.1145/3472883.3486992
- [7] Lei Bu, Qiuping Zhang, Suwan Li, Jinglin Dai, Guangdong Bai, Kai Chen, and Xuandong Li. 2023. Security Checking of Trigger-Action-Programming Smart Home Integrations. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle, WA, USA) (ISSTA 2023). Association for Computing Machinery, New York, NY, USA, 639–651. doi:10.1145/3597926.3598084
- [8] Yufan Cai, Zhe Hou, Xiaokun Luan, David Miguel Sanan Baena, Yun Lin, Jun Sun, and Jin Song Dong. 2024. Towards Large Language Model Aided Program Refinement. arXiv:2406.18616 [cs.SE] <https://arxiv.org/abs/2406.18616>
- [9] Nilanjan Daw, Umesh Bellur, and Purushottam Kulkarni. 2020. Xanadu: Mitigating cascading cold starts in serverless function chain deployments (*Middleware '20*). Association for Computing Machinery, New York, NY, USA, 356–370. doi:10.1145/3423211.3425690
- [10] Robert Dickerson, Prasita Mukherjee, and Benjamin Delaware. 2025. Kestrel: Relational verification using e-graphs for program alignment. *Proceedings of the ACM on Programming Languages* 9, OOPSLA1 (2025), 1073–1100.
- [11] Boyu Du, Jingya Zhou, Jin Wang, Jiangwei Wang, and Zhijun Li. 2025. Joint Task Scheduling and Resource Allocation in Cloud-Edge Collaborative Computing Systems. In *Proceedings of the 54th International Conference on Parallel Processing (ICPP '25)*. Association for Computing Machinery, New York, NY, USA, 586–596. doi:10.1145/3754598.3754646
- [12] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.* (July 1987), 319–349.

- [13] Stefan M. Freudenberger, Thomas R. Gross, and P. Geoffrey Lowney. 1994. Avoidance and suppression of compensation code in a trace scheduling compiler. *ACM Trans. Program. Lang. Syst.* 16, 4 (July 1994), 1156–1214. doi:10.1145/183432.183446
- [14] Mathyas Giudici, Alessandro Sironi, Ismaele Villa, Samuele Scherini, and Franca Garzotto. 2025. Generating HomeAssistant Automations Using an LLM-based Chatbot. arXiv:2505.02802 [cs.HC] <https://arxiv.org/abs/2505.02802>
- [15] Google. 2024. Android Things. <https://developer.android.com/things/>. Accessed: 2025-04-12.
- [16] Richard E. Hank, Scott A. Mahlke, Roger A. Bringmann, John C. Gyllenhaal, and Wen-mei W. Hwu. 1993. Superblock formation using static program analysis. In *Proceedings of the 26th Annual International Symposium on Microarchitecture* (Austin, Texas, USA) (MICRO 26). IEEE Computer Society Press, Washington, DC, USA, 247–255.
- [17] IFTTT. 2024. IFTTT:Automation for Business and Home. <https://ifttt.com/>. Accessed:2024-09-06.
- [18] Negin Jahanbakhsh, Mario Vega-Barbas, Iván Pau, Lucas Elvira-Martín, Hírad Moosavi, and Carolina García-Vázquez. 2025. Leveraging Retrieval-Augmented Generation for Automated Smart Home Orchestration. *Future Internet* 17, 5 (2025), 198.
- [19] 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 (March 2024), 1–38. doi:10.1145/3643505
- [20] Christopher Kraemer, William Gelder, and Josiah Hester. 2024. User-directed Assembly Code Transformations Enabling Efficient Batteryless Arduino Applications. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 8, 2, Article 44 (May 2024), 32 pages. doi:10.1145/3659590
- [21] Ye Lei. 2025. Intelligent Prediction and Dynamic Scheduling Optimization Strategy for Cloud Computing Resources under Burst Load Scenarios. In *Proceedings of the 2025 International Symposium on Machine Learning and Social Computing (MLSC '25)*. Association for Computing Machinery, New York, NY, USA, 59–67. doi:10.1145/3778450.3778459
- [22] Fu Li, Jiaming Huang, Yi Gao, and Wei Dong. 2023. ChatIoT: Zero-code Generation of Trigger-action Based IoT Programs with ChatGPT. In *Proceedings of the 7th Asia-Pacific Workshop on Networking* (Hong Kong, China) (APNet '23). Association for Computing Machinery, New York, NY, USA, 219–220. doi:10.1145/3600061.3603141
- [23] Hongliang Liang, Luming Yin, Wenying Hu, Yuxiang Li, and Wuwei Shen. 2025. Unified and Split Symbolic Execution for Exposing Semantic Differences. *ACM Trans. Softw. Eng. Methodol.* 34, 6, Article 168 (July 2025), 27 pages. doi:10.1145/3705299
- [24] Bingxin Liu, Yinghui Huang, Jianhua Gao, Jianjun Shi, Yongpeng Liu, Yipin Sun, and Weixing Ji. 2025. From Profiling to Optimization: Unveiling the Profile Guided Optimization. arXiv:2507.16649 [cs.PF] <https://arxiv.org/abs/2507.16649>
- [25] 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. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 9, 2, Article 40 (June 2025), 26 pages. doi:10.1145/3729482
- [26] Xiaoyi Liu, Yingtian Shi, Chun Yu, Cheng Gao, Tianao Yang, Chen Liang, and Yuanchun Shi. 2023. Understanding In-Situ Programming for Smart Home Automation. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 7, 2, Article 66 (June 2023), 31 pages. doi:10.1145/3596254
- [27] Lezhi Ma, Shangqing Liu, Yi Li, Xiaofei Xie, and Lei Bu. 2025. SpecGen: Automated Generation of Formal Program Specifications via Large Language Models. arXiv:2401.08807 [cs.SE] <https://arxiv.org/abs/2401.08807>
- [28] McKenna McCall, Eric Zeng, Faysal Hossain Shezan, Mitchell Yang, Lujo Bauer, Abhishek Bichhawat, Camille Cobb, Limin Jia, and Yuan Tian. 2023. Towards usable security analysis tools for trigger-action programming. In *Proceedings of the Nineteenth USENIX Conference on Usable Privacy and Security* (Anaheim, CA, USA) (SOUPS '23). USENIX Association, USA, Article 17, 20 pages.
- [29] openHAB Foundation. 2024. openHAB. <https://www.openhab.org/>. Accessed: 2025-04-12.
- [30] B. Ramakrishna Rau. 1994. Iterative modulo scheduling: an algorithm for software pipelining loops (MICRO 27). Association for Computing Machinery, New York, NY, USA, 63–74. doi:10.1145/192724.192731
- [31] Piero Romare, Victor Morel, Farzaneh Karegar, and Simone Fischer-Hübner. 2023. Tapping into Privacy: A Study of User Preferences and Concerns on Trigger-Action Platforms. arXiv:2308.06148 [cs.CY] <https://arxiv.org/abs/2308.06148>
- [32] Samsung Electronics. 2024. SmartThings. <https://www.samsung.com/global/galaxy/apps/smartthings/>. Accessed: 2025-04-12.
- [33] 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:2408.12687 [cs.HC] <https://arxiv.org/abs/2408.12687>
- [34] Shrijan Subedi, Satish Kumar, Nawar Jawad, and Ah Lian Kor. 2026. Energy-Aware Latency Optimization for Scheduling Serverless Workload in Edge Computing Environment. In *Proceedings of the 18th IEEE/ACM International Conference on Utility and Cloud Computing (UCC '25)*. Association for Computing Machinery, New York, NY, USA, Article 21, 10 pages. doi:10.1145/3773274.3774276
- [35] Qwen Team. 2025. Qwen3 Technical Report. arXiv:2505.09388 [cs.CL] <https://arxiv.org/abs/2505.09388>
- [36] Haluk Topcuoglu, Salim Hariri, and Min-You Wu. 2002. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE transactions on parallel and distributed systems* 13, 3 (2002), 260–274.
- [37] Zhengrong Wu, Hua Li, Wenqin Chen, Zilong Liu, and Shaohui Du. 2025. Resource Allocation for Power IoT Edge Devices Based on Edge Node Collaboration. In *Proceedings of the 4th International Conference on Computer, Internet of Things and Control Engineering (CITCE '24)*. Association for Computing Machinery, New York, NY, USA, 121–127. doi:10.1145/3705677.3705698

- [38] Ziteng Yang, Jun Shirako, and Vivek Sarkar. 2024. Fully Verified Instruction Scheduling. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 299 (Oct. 2024), 26 pages. doi:10.1145/3689739
- [39] Yinbo Yu, Yuanqi Xu, Kepu Huang, and Jiajia Liu. 2024. TAPFixer: Automatic Detection and Repair of Home Automation Vulnerabilities based on Negated-property Reasoning. *ArXiv abs/2407.09095* (2024). <https://api.semanticscholar.org/CorpusID:271161844>
- [40] Lefan Zhang, Weijia He, Jesse Martinez, Noah Brackenburg, Shan Lu, and Blase Ur. 2019. AutoTap: synthesizing and repairing trigger-action programs using LTL properties. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) (*ICSE '19*). IEEE Press, 281–291. doi:10.1109/ICSE.2019.00043
- [41] 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. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 4, 3, Article 104 (Sept. 2020), 26 pages. doi:10.1145/3411838
- [42] Valerie Zhao, Lefan Zhang, Bo Wang, Michael L. Littman, Shan Lu, and Blase Ur. 2021. Understanding Trigger-Action Programs Through Novel Visualizations of Program Differences. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems* (Yokohama, Japan) (*CHI '21*). Association for Computing Machinery, New York, NY, USA, Article 312, 17 pages. doi:10.1145/3411764.3445567

A Routine Dataset Collection and Preprocessing

Sampling. We took a snapshot of the IFTTT Popular pages on February 16, 2026. For each candidate applet, we recorded its IFTTT ID, URL, popularity rank, original title, trigger service and event, and action service and action. This snapshot defines our sampling frame. We first sort the candidates by IFTTT ID and then shuffle them using the fixed random seed 20260216. Candidates are drawn without replacement and checked against the eligibility and deduplication criteria below. If a candidate is excluded, sampling continues from the remaining candidates until 50 eligible routines have been collected. Neither validation outcomes nor optimization performance are used during sampling. The modality restriction applies to device operations scheduled and optimized by FloWeaver: these operations must use BLE, Cloud API, or Local Gateway. Time, weather, location, Webhook, and Button events may serve as external triggers, while Email and Notification may serve as platform outputs; neither is counted as an optimized device modality.

Filtering. Table 10 summarizes the filtering criteria.

Table 10. Routine filtering criteria.

Criterion	Requirement
Complete specification	The trigger, condition, and device action must be sufficiently specified to reconstruct the routine.
Target modality	Every optimized device operation must use BLE, Cloud API, or Local Gateway.
Supported execution	Each optimized device operation must have a semantics-preserving mapping to an integration supported by the evaluation environment.
External trigger	Time, weather, location, Webhook, and Button events are allowed as external inputs and are not counted as device modalities.
Platform output	Email and Notification actions are allowed as platform-visible outputs and are not counted as device modalities.
Unsupported operation	A routine is excluded if any required device operation cannot be implemented without changing its trigger or action semantics.
Duplicate routine	A routine identified as a source, textual, or semantic duplicate is excluded.

Deduplication. We apply three levels of deduplication. A *source duplicate* has the same IFTTT applet ID or URL. An *exact textual duplicate* has the same title after case, punctuation, and whitespace normalization. A *semantic duplicate* expresses the same normalized trigger, condition, target-device type, and action despite differences in wording. For each duplicate set, we retain the routine with the higher popularity rank. If the ranks are equal, we retain the entry with the more complete original metadata. Deduplication does not use latency, validation, or optimization results. Routines with similar structures but different triggers, device operations, or mapped integrations are retained because they exercise different platform paths. Such routines are marked as belonging to the same template family rather than being treated as independent routine patterns.

Integration mapping. Eligible IFTTT services and device operations are mapped to their corresponding Home Assistant integrations before evaluation. The mapping is fixed before running any experiment and follows three rules: it preserves the original trigger and action semantics; it is applied consistently to all routines; and a routine is excluded when no semantics-preserving integration is available. External triggers and platform outputs are recorded separately and are not replaced with device integrations.

Final datasets. The 50 accepted routines form the smart-home dataset. The long-chain dataset is derived from the same routines by tripling the multiplicity of their device operations while preserving their triggers,

actions, integration mappings, and dependency structure. It is therefore a controlled stress dataset rather than an independently sampled IFTTT dataset. The complete routines used in both datasets are listed in Table 11.

B Offline LLM-Assisted Platform Profile Construction

The following presents the prompt used to instruct Qwen3-14B to perform offline LLM-assisted platform-profile construction, as described in §3.6.

Prompt (Partial)

System: You discover Home Assistant integration profile candidates. Return JSON only. Every candidate must cite concrete `source_refs` or `evidence_ids`. Do not invent files, line numbers, marker types, or unsupported concurrency rules.

User: Using the supplied evidence extracted from platform documentation, integration code, and tests, propose candidate runtime carriers, marker detectors, test behaviors, and profile rules. Use only the candidate kinds, marker types, and protocol families defined in the supplied contract. For each candidate, report its type, protocol family, relevant integration and symbol, match pattern, confidence, and supporting source references or evidence identifiers. Source references must point to an existing repository or test file and a valid source line. Rule candidates must describe a supported ordering or constraint relationship. Ground all claims in the supplied evidence summary and static file and function indexes, and return only candidates supported by this local context.

In our evaluation, we explicitly configure Qwen3-14B through an OpenAI-compatible interface with temperature 0.0 and a 120-second timeout. Each request contains at most 120 evidence records, 160 file records, and 160 function records. Candidate confidence must be at least 0.55, with at most six source references per candidate. Other decoding parameters follow the serving endpoint defaults.

For each candidate rule, Profile Builder records how often the rule is matched and applied, the trace-, resource-, and final-state preservation rates, and the number of observed counterexamples. Rules with insufficient evidence or application coverage remain soft. We do not assume that the extracted lifecycle-rule set is universally complete; nevertheless, our empirical end-to-end evaluation indicates that it achieves reasonable coverage and quality for the evaluated routines.

C Equivalence Validation Scope

Our Validator checks *observational equivalence* rather than complete program equivalence. Replay scenarios are generated automatically through bounded enumeration. Each scenario specifies the initial trace state and execution environment used by the Validator. For each scenario, the original and optimized routines are validated under the same initial states, external inputs, API outcomes, resource budgets, and modeled concurrency conditions.

The validation uses a normalized program IR for KestRel-based alignment and a canonical replay trace IR. These intermediate representations preserve behavior observable to physical devices, the automation platform, and end users. Under a scenario, the two routines are considered observationally equivalent when they satisfy the following conditions:

- (1) they produce the same final device states and platform-visible states;
- (2) they preserve the same required commands, notifications, and exception behavior;
- (3) any reordered events are independent under the modeled dependencies and do not change an observable outcome; and
- (4) their resource behavior satisfies the same protocol requirements, including acquire-before-use, valid resource release, concurrency limits, and rate limits.

The optimized routine is accepted only when these conditions hold for every enumerated replay scenario. The result therefore establishes observational equivalence with respect to the enumerated scenario set and the modeled platform contracts.

Table 11. List of smart-home routines used in the evaluation.

ID	Routines
1	Arm Blink cameras when Abode is set to Home mode
2	Turn on deCONZ Hood ambient light daily
3	Automatically turn your Hue lights blue whenever it starts to rain
4	Turn on deCONZ Hood light on a special day
5	Turn on ESPHome Hood ambient light on Sunday
6	Get an Email from deCONZ Hood When Carbon Filter Is Nearly Full
7	Set Philips Hue lights to red when it starts snowing
8	Close all SwitchBot curtains when it is rainy
9	Blink Philips Hue lights and notify if fridge door left open
10	Arm Blink system and run SwitchBot scene when you leave home
11	Turn on Philips Hue lights when you arrive home
12	Turn on Hue lights when it starts raining
13	Turn off SmartThings lights with Button widget
14	Enable Govee plug before grocery trip
15	Turn on Hue lights when Netatmo temperature rises above threshold
16	Notify when ecobee outdoor temperature exceeds threshold
17	Turn on Hue and SmartThings lights when you enter the bedroom
18	Arm Blink with a Netatmo button press
19	Disarm Blink when the first person arrives home
20	Notify when Netatmo temperature rises above threshold
21	Turn off Hue and Nanoleaf lights when you leave home
22	Turn on Hue and Nanoleaf lights at sunset
23	Turn off SmartThings lights and Nanoleaf outlets at bedtime
24	Color-loop Hue lights when indoor temperature gets high
25	Turn on SmartThings scene and Hue lights when someone is sitting on the sofa
26	Color-loop Hue lights when Netatmo CO2 gets high
27	Send Roborock home and disarm Blink when a Webhook event fires
28	Change Philips Hue light color when Netatmo detects rain
29	Turn off SmartThings lights and Hue outlets when leaving home
30	Color-loop Hue lights when Netatmo noise gets loud
31	Turn off SmartThings lights and ESPHome outlets when there is nobody home
32	Turn off Hue and Nanoleaf lights at bedtime
33	Turn on Hue and Govee lights at party time
34	Blink deCONZ lights and notify if entry door left open
35	Turn off SmartThings scene and Hue lights when leaving home
36	Turn on Hue and Nanoleaf lights when someone comes back, then play some music
37	Arm Blink and start Roborock when a Webhook event arrives
38	Turn off lights and Govee plugs with one tap
39	Activate Tuya AC sabbath mode every Friday

Table 11. List of smart-home routines used in the evaluation (continued).

ID	Routines
40	Turn off my TP-Link plug when there's an electricity price spike
41	Automatically turn your lights on at sunset
42	Toggle Hue lights with Button widget
43	Turn off TP-Link light at sunrise
44	Set Tuya door bell to security mode on weekdays
45	Open SwitchBot curtains when your Smart Bed alarm rings
46	Turn on Hue Hood light on the rainy day
47	Switch your ecobee to your chosen comfort profile as you arrive home
48	Turn on Govee plug at sunset
49	Get IFTTT notification when temperature is high
50	Press Button widget to activate Tuya AC Super Cooling

D Additional Results for Independent Audit

We provide additional details for the independent audit in §4.5. Tables 12 to 14 report the audited state fields, required observable events, and mismatch/non-failure breakdown used to support the summary results in Table 3.

Table 12. Required observable-event checks in the independent device-state audit. The table reports the event categories used for Req-Event Match, Except. Match, and Lifecycle Match.

Event Category	Examples	Match
Device command	light command, curtain refresh, AC status, media query	100.0%
State writeback	HA entity update, routine summary write	100.0%
Branch outcome	sensor-triggered branch, condition result	100.0%
Notification	expected notification / no unexpected notification	100.0%
Exception category	timeout, unavailable, auth failure class	100.0%
Lifecycle event	BLE connect/disconnect, cloud session, subscribe/unsubscribe	100.0%

Table 13. Mismatch and non-failure breakdown in the independent device-state audit. Semantic mismatch categories all have zero occurrences; timing-only differences are expected optimization effects and are not correctness failures.

Category	Count	Interpretation
Final-state mismatch	0	No semantic mismatch
Missing required event	0	No lost observable action
Extra unexpected event	0	No extra semantic side effect
Exception mismatch	0	Error behavior preserved
Lifecycle mismatch	0	Resource behavior preserved
Timing-only difference	141 / 150	Expected optimization effect
Environmental invalid run	2 rerun, 0 counted	Device/API instability, excluded after rerun
Oracle read retry	4	Vendor API delayed response, resolved by retry

Table 14. State-field checks in the independent audit. The table lists the audited device/platform state fields, the oracle source used to read them, and the corresponding Dev-State or Plat-State matching result.

Device / Platform Type	Checked Fields	Source	Match
Light / Bulb / Light group	on/off, brightness, color, color_temperature, effect, scene_id	vendor/local API + HA state	100.0%
Curtain / Cover / Blind	open/closed, position, tilt, moving/stopped, last_command_result	device query + HA state	100.0%
Plug / Switch / Relay	on/off, power_state, relay_state, energy_power_w, last_update	local/cloud API + HA state	100.0%
Thermostat / AC / HVAC	power, hvac_mode, target_temperature, current_temperature, fan_mode, preset_mode, swing_mode	cloud/local API + HA state	100.0%
Environmental Sensor	temperature, humidity, co2, voc, pm2.5, battery, signal_strength, last_seen	BLE/vendor read + HA state	100.0%
Motion / Contact / Presence Sensor	motion, opened/closed, presence, occupancy, battery, last_seen	BLE/Zigbee/local gateway + HA state	100.0%
Media Player / TV / AVR	on/off, playing/stopped, source, volume, muted, media_state, input_mode	local API + HA state	100.0%
Camera / Security Device	armed/disarmed, online/offline, motion_detected, recording_state, siren_state	cloud/vendor API + HA state	100.0%
Door Lock / Garage / Access Device	locked/unlocked, open/closed, jammed, battery, last_command_result	vendor/local API + HA state	100.0%
Vacuum / Robot	docked, cleaning, paused, returning, battery, fan_mode, cleaning_area, error_state	cloud/vendor API + HA state	100.0%
Appliance	power, mode, program, remaining_time, door_state, operation_state, error_state	cloud/vendor API + HA state	100.0%
BLE Lifecycle	advertisement_seen, connect_success, read_complete, disconnect/release, last_seen	BLE scanner log + HA state	100.0%
Cloud/API Lifecycle	auth_valid, request_sent, response_received, rate_limit_status, retry/backoff_status	API log + HA event log	100.0%
Local/Gateway Lifecycle	endpoint_resolved, session_ready, subscribe_active, response_received, unsubscribe/release	local API/gateway log + HA event log	100.0%
HA Entity State	entity_state, attributes, last_changed, last_updated, availability	HA state registry	100.0%
HA Writeback / Summary	state_writeback, summary_entity, notification, automation_trace, service_call_result	HA state/event log	100.0%
Exception / Error Behavior	exception_category, timeout, unavailable, auth_error, rate_limit_error, device_error	execution log + HA event log	100.0%

E Additional Results for Mutation-Injected Test

This section provides supplementary results for the mutation-injected test in §4.5.2. Tables 15 to 18 expand the summary results by reporting concrete counts, counterexample categories, detected violation labels, and rollback repair outcomes. Counts are non-exclusive because one mutation may produce multiple witnesses or violation labels.

Table 15. Concrete counts for the mutation-injected test. The table expands the aggregate MRR and observed FPR results reported in the main text.

Metric	Count
Valid injected mutations	150
Rejected mutations	150
Accepted false positives	0
Counterexample-backed rejections	143
Trace-only rejections	7
Type-accurate / compatible localizations	144
Type-imprecise but rejected cases	6
Invalid generated mutations excluded/regenerated	4

Table 16. Counterexample quality breakdown. The table reports the main violation categories found by the validator and representative evidence recorded in the corresponding witnesses.

Violation Category	# Witnesses	Example Evidence
Branch/order mismatch	36	dependent command appears before required read
Observation boundary violation	20	HA state write occurs before device response
Missing prerequisite	20	refresh/status query absent before publish
Resource protocol violation	11	BLE GATT op overlaps with connect/session boundary
Cloud prerequisite / exception mismatch	17	cloud request sent without auth/rate check
State write target mismatch	20	writeback target differs from baseline entity
Command order violation	18	same-device command order reversed
Lifecycle violation	18	cleanup occurs before required receive/read
Trace-only rejection	7	rejected by trace diff without symbolic witness

Table 17. Detected violation labels in the mutation-injected test. The `semantic_delta` label is a fallback category; these cases are still rejected and therefore do not affect the false-positive rate.

Detected Label	Count
<code>order_violation</code>	54
<code>observation_boundary_violation</code>	20
<code>missing_prerequisite</code>	20
<code>resource_protocol_violation</code>	11
<code>cloud_session_or_rate_violation</code>	17
<code>state_write_target_mismatch</code>	20
<code>lifecycle_violation</code>	18
<code>semantic_delta</code>	10

Table 18. Rollback repair outcomes after mutation rejection. Repair success is reported as a secondary result and is not used to compute the false-positive rate.

Metric	Count	Rate
Rejected mutations	150	100.0%
Rollback attempted	150	100.0%
Repaired by rollback	137	91.3%
Not repaired but safely rejected	13	8.7%
Accepted after unsafe mutation without detection	0	0.0%