

Kollect: Retrieval-Based Candidate Proposal for Interactive Command Annotation in Personal Assistants

CHENG FANG, City University of Hong Kong, China

ZIMU ZHOU[✉], City University of Hong Kong, China

BOYI LIU, City University of Hong Kong, China

KE MA, Northwestern Polytechnical University, China

BIN GUO, Northwestern Polytechnical University, China

Personal assistants increasingly use large language models to map natural-language commands to actions grounded in a user's devices, APIs, and preferences. Adapting these assistants to individual users requires executable supervision, in which user commands are paired with user-approved executable labels. Collecting such supervision is difficult because user commands are often flexible and under-specified, leaving the intended label ambiguous without feedback. Multiple-choice confirmation can reduce user effort if the assistant can quickly propose a small candidate set that contains the intended label or recovers enough of its executable structure to reduce correction. Existing fixed-head parsers are efficient but constrained by predefined output inventories, while generation-based methods are more flexible but incur high latency by decoding labels token by token. This paper presents Kollect, a retrieval-based framework for interactive command annotation in personal assistants. Kollect exploits the fact that valid executable labels are bounded by the user's schemas, retrieving candidates rather than generating them during interaction. To improve coverage for ambiguous commands, Kollect uses a query-adaptive grouped retriever that distributes the display budget across plausible API groups and selects query-relevant, non-redundant candidates within each group. To reduce proposal latency, Kollect further introduces a retriever-aligned early-exit encoder that stops once the API groups selected for display become stable. We evaluate Kollect on smart-home and smartphone assistant scenarios. Kollect achieves 92.1% and 91.9% API accuracy, 89.4% and 89.0% Slot F1, and 653 ms and 668 ms average proposal latency on the two scenarios, respectively. These results show that Kollect provides near-instant candidate proposal while retaining competitive annotation quality compared with much slower generation-based methods. Its collected supervision also improves downstream personalization through fine-tuning, retrieval-augmented generation, and agent memory.

CCS Concepts: • **Human-centered computing** → **Ubiquitous and mobile computing**; • **Computing methodologies** → **Machine learning**.

Additional Key Words and Phrases: personalized assistants, user command annotation, retrieval-based confirmation, interactive data collection, edge intelligence

ACM Reference Format:

Cheng Fang, Zimu Zhou, Boyi Liu, Ke Ma, and Bin Guo. 2026. Kollect: Retrieval-Based Candidate Proposal for Interactive Command Annotation in Personal Assistants. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 10, 4, Article 212 (December 2026), 36 pages. <https://doi.org/10.1145/3857974>

Authors' Contact Information: [Cheng Fang](mailto:chengfang9-c@my.cityu.edu.hk), Department of Data Science, City University of Hong Kong, Hong Kong, China, chengfang9-c@my.cityu.edu.hk; [Zimu Zhou](mailto:zimuzhou@cityu.edu.hk) (corresponding author), Department of Data Science, City University of Hong Kong, Hong Kong, China, zimuzhou@cityu.edu.hk; [Boyi Liu](mailto:boyi.liu@my.cityu.edu.hk), Department of Data Science, City University of Hong Kong, Hong Kong, China, boyi.liu@my.cityu.edu.hk; [Ke Ma](mailto:nwpumarco@gmail.com), School of Computer Science, Northwestern Polytechnical University, Xi'an, Shaanxi, China, nwpumarco@gmail.com; [Bin Guo](mailto:guob@nwpu.edu.cn), Northwestern Polytechnical University, Xi'an, China, guob@nwpu.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-ART212

<https://doi.org/10.1145/3857974>

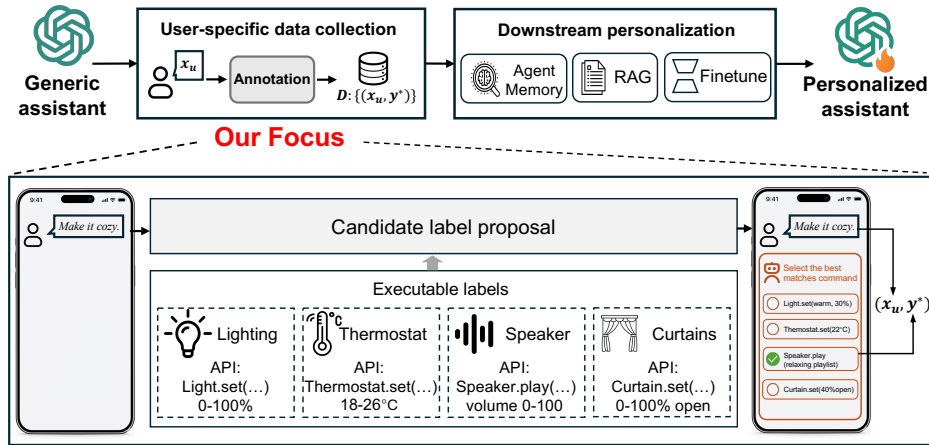


Fig. 1. Our focus in the executable-supervision collection loop. Given a user command, the assistant proposes and displays candidate executable labels over the user’s bounded label space. The confirmed pair supports downstream personalization.

1 Introduction

Large Language Models (LLMs) are increasingly used as personal assistants that interpret natural-language requests and trigger actions in smart homes, smartphones, and other ubiquitous computing environments [5, 9, 56, 68]. To serve a particular user, however, an assistant must adapt to that user’s devices, APIs, and preferences [22, 59]. Such adaptation requires **executable supervision**, *i.e.*, natural-language commands paired with user-approved executable labels, where each label specifies a structured action such as an API call with concrete arguments. For example, “make it cozy” may mean dimming a bedroom light, setting a thermostat, or playing a saved playlist, depending on the user. These command-label pairs support downstream personalization through fine-tuning [20, 51], retrieval-augmented generation (RAG) [27, 50, 51], and agent memory [59].

Collecting executable supervision is difficult because user commands are often under-specified [4, 31]. A user may describe a desired state without naming the target device, API, or parameter values. Hence, the intended executable label cannot always be inferred from the command alone and must be resolved through user feedback. A practical collection mechanism must obtain this feedback with little effort and low latency, as cumbersome annotation can undermine usability of personal assistants [1, 55].

Interactive annotation provides a natural way to collect such feedback [24, 65]. Correction-based interfaces [12, 24, 31] ask users to explain how a wrong prediction should be repaired, while binary confirmation [24, 65] may require multiple rounds. Multiple-choice confirmation [4] offers a lighter alternative. The assistant presents a small set of candidate executable labels, from which the user selects or completes the intended action. The resulting technical problem is **candidate proposal**, *i.e.*, quickly constructing a small candidate set that recovers useful parts of the intended executable structure while minimizing the user’s remaining correction. Fig. 1 shows this collection loop and our focus.

Existing candidate-proposal mechanisms face an *accuracy-latency* trade-off. Fixed-head intent-slot parsers [8, 60] are efficient but rely on predefined output inventories and may not accommodate user-specific devices, APIs, and parameter combinations. Generation-based methods, *e.g.*, seq2seq models [35, 62] and LLM prompting [22, 31, 62], are more flexible but synthesize labels token by token, introducing substantial interaction latency. Since valid labels are already bounded by the user’s schemas, generating every candidate from scratch is unnecessary. This motivates a **retrieval**-based alternative that maintains a command library and retrieves labels for confirmation.

However, standard top- K retrieval [27, 29] presents two limitations. First, under-specified commands such as “make it cozy” can make several API groups plausible, causing flat nearest-neighbor retrieval to fill the display with redundant variants from one dense group [6, 7]. Second, every request must be encoded before candidates can be shown. A large encoder is too slow for interactive use, whereas a smaller encoder may alter the retrieval decisions that determine the displayed set [41, 43, 47]. Therefore, retrieval-based candidate proposal must jointly address **candidate coverage** and **query-encoding latency**.

This paper presents Kollect, a retrieval-based framework for collecting executable supervision through multiple-choice confirmation. Kollect maintains an API-grouped command library whose entries represent schema-valid executable labels. For each incoming command, it retrieves a small candidate set for user confirmation and records the resulting command-label pair for downstream personalization.

Kollect consists of two core designs. First, Kollect introduces a **query-adaptive grouped retriever**. Instead of ranking all library entries globally, it estimates API-group relevance, allocates the display budget across plausible groups, and fills each group quota with a query-conditioned coverage objective. This improves coverage under API ambiguity while reducing redundant candidates within each group. Second, Kollect develops a **retriever-aligned early-exit encoder**. The deepest exit defines the reference embedding space used by the command library, while shallow exits are trained to preserve the group-level retrieval behavior needed by the grouped retriever. At runtime, encoding stops when the API groups receiving display slots stabilize across consecutive exits. Together, these designs improve candidate coverage while keeping proposal latency low.

We evaluate Kollect on personalized-assistant applications in smart-home (Home) and smartphone (Phone) scenarios. On Jetson Orin Nano, Kollect achieves 92.1% and 91.9% API accuracy, 89.4% and 89.0% Slot F1, and average proposal latencies of 653 ms and 668 ms in the two scenarios, respectively. Across baselines spanning fixed-head prediction [8, 24], seq2seq generation [4, 49], and LLM-powered generation [12, 22, 31], Kollect notably reduces proposal latency while retaining strong annotation quality.

The main contributions of this paper are summarized as follows.

- We formulate executable-supervision collection for personalized assistants as retrieval-based interactive annotation. We identify candidate proposal as a bottleneck in multiple-choice confirmation, where the system must balance candidate coverage, user ambiguity, and proposal latency.
- We design Kollect, a novel retrieval-based candidate-proposal framework for personalized assistants. Its query-adaptive grouped retriever allocates display slots across plausible API groups and selects candidates by query-conditioned intent coverage. Its retriever-aligned early-exit encoder reduces query-encoding latency while preserving the retrieval behavior needed for confirmation.
- We evaluate Kollect on smart-home and smartphone assistant applications. Results show that Kollect improves the accuracy-latency trade-off over fixed-head prediction [8, 24], seq2seq generation [4, 49], and LLM-powered generation [12, 22, 31] baselines. We also show that Kollect has low on-device overhead, provides favorable user interaction, and collects data that improves downstream personalization through fine-tuning, RAG, and agent memory.

2 Problem Setting

2.1 Executable Supervision

We consider a target user with a bounded executable label space Y , defined by the user’s installed devices, APIs, and valid argument domains. The space is bounded because the assistant can execute only labels permitted by the user’s current schemas, rather than arbitrary natural-language outputs. Each executable label $y \in Y$ is a structured action instance that specifies an API with its required arguments, e.g., turning on a bedroom light or setting a thermostat value.

Table 1. Representative combinations of interaction modes and candidate-proposal mechanisms.

Interaction mode	Candidate-proposal mechanism	Representative methods
None	Prediction	JointBERT [8]
None	Generation	Seq2Seq-T5 [49], DroidCall [62]
Binary	Generation	Feedback-SP [24]
Binary	Prediction	MISP [65]
Multiple-choice	Generation	Guided-K [4], Home-Llama-3B [22]
Natural-language	Generation	NL-EDIT [12], Sasha [31]
Multiple-choice	Retrieval	Kollect (Ours)

A personal assistant needs executable supervision, *i.e.*, command-label pairs that ground the user’s own natural-language commands in the executable labels intended by that user. We denote the collected supervision dataset by D . Such data can be used by downstream personalization methods, including fine-tuning [20, 22, 51], RAG [27, 50, 51], and agent memory [50, 59].

This paper studies the **annotation problem** of collecting such supervision, particularly on **under-specified** commands. Let x_u be an incoming user command and $y^* \in Y$ the intended executable label. Given x_u , the annotation mechanism should recover y^* with user feedback and add the confirmed pair (x_u, y^*) to D . An under-specified command may describe a desired state without naming the target API, device, or parameter values. Consequently, y^* cannot always be recovered from x_u alone and must be resolved through *interaction*. The resulting confirmation is particularly informative because the approved label reveals default preferences that are not expressed in the command. By contrast, a fully specified command such as “set the AC to 24.1 °C” already determines the API, device, and value, so its confirmation provides little additional preference supervision. Although we can process fully specified commands (see Sec. 7.5), they are outside our primary collection focus.

2.2 Interactive Annotation Design Space

Executable supervision collection can be viewed as interactive semantic parsing (ISP) [65], where the system maps a natural-language command to a structured executable label with user feedback.

An ISP system involves two design choices. The first is the *interaction mode*, *i.e.*, how the user provides feedback. Common modes include no feedback, binary confirmation, multiple-choice confirmation, and natural-language correction. The second is the *candidate-proposal mechanism*, *i.e.*, how the system produces the executable label or candidate label set shown to the user. Common mechanisms include fixed-head prediction and generation-based proposal. Tab. 1 summarizes representative combinations.

Interaction mode. In this work, we adopt **multiple-choice confirmation**. Given a command x_u , the system presents a candidate label set $Y_S \subseteq Y$ with $|Y_S| = K$. Compared with natural language correction [12, 24, 31], this interface does not require the user to explain how an incorrect label should be repaired. Compared with binary confirmation [24, 65], it exposes several plausible interpretations in a single interaction. Thus, multiple-choice confirmation provides a bounded and lightweight interface for collecting executable supervision.

Interaction outcomes. The displayed candidate set determines how much additional input is required to obtain the intended label y^* :

- (1) *Exact hit*: $y^* \in Y_S$. The user directly selects and confirms the intended label.
- (2) *Same-API correction*: $y^* \notin Y_S$, but at least one displayed label uses the intended API. The user selects such a candidate and corrects its mismatched device or argument fields.
- (3) *Full fallback*: no displayed label uses the intended API. The user selects “None of the above” and specifies the intended API and its required fields.

All three paths produce a confirmed executable label, but they require *different amounts of user input*. An exact hit permits direct confirmation, a same-API candidate limits the correction to mismatched fields, and full fallback requires the user to provide the complete executable structure. Thus, the technical problem is to design **candidate proposal mechanisms** that minimize the user’s remaining specification effort while satisfying the latency requirements of interactive use.

2.3 Candidate Proposal Objective and Metrics

We assess candidate proposal by **annotation quality** and **proposal latency**. Annotation quality measures how much of the intended executable structure is recovered by Y_S , which determines the correction burden under the three interaction outcomes above. Proposal latency is measured from receiving x_u to displaying the candidate set.

We use two *set-level* metrics to measure annotation quality. **API accuracy** indicates whether the candidate set contains a label with the intended API: $\text{APIAcc}(Y_S, y^*) = \max_{y \in Y_S} \mathbb{I}[\text{API}(y) = \text{API}(y^*)]$, whereas **Slot F1** measures the argument-level similarity between y^* and the best-matching candidate: $\text{SlotF1}(Y_S, y^*) = \max_{y \in Y_S} \text{SlotF1}(y, y^*)$. Both metrics follow prior semantic parsing evaluation [8]. The two maxima are computed independently and may be attained by different candidates. They evaluate the partial executable structure recovered by the candidate set rather than simulating which candidate a user would select. We use them as our primary offline metrics because they distinguish useful partial matches from candidates that are unrelated to the intended label.

Aligned with previous studies [31, 53] on under-specified commands, we do not use **exact-label Hit@K** ($\text{Hit@K}(Y_S, y^*) = \mathbb{I}[y^* \in Y_S]$) as a primary measure of annotation quality, because any mismatch in the executable label counts as a complete miss and ignores useful partial matches captured by API accuracy and Slot F1. Instead, we report it as a complementary measure of direct confirmation opportunity on human-authored commands (see Sec. 7.5) and in a user study (see Sec. 7.9).

2.4 Why Retrieval-Based Candidate Proposal

Having fixed the interaction mode and candidate proposal objective, we next consider which proposal mechanism can provide high annotation quality with low proposal latency. Fixed-head approaches, such as intent-slot classifiers [8, 60], are efficient but depend on predefined output inventories. They are less suitable when executable labels are compositional, involving different APIs, devices, and parameter values. Generation-based approaches [35, 46, 62] are more flexible, but they construct executable labels through token-level decoding. This can introduce substantial latency in the annotation loop, especially when larger models are used to improve candidate quality.

Fig. 2 compares the accuracy and latency of representative candidate proposal mechanisms. Fixed-head methods such as JointBERT [8, 60] operate near-instantly (<1 s) but achieve only 82% Slot F1, as they rely on predefined output inventories and fail to fill unspecified slot values. Generation-based methods [35, 46, 62] improve accuracy to 88-93% but incur 4-30 s latency, since they decode executable labels token by token and fall into the disruptive or patience-lost zones. These results motivate retrieval-based candidate proposal. Because valid labels are constrained by the user’s executable label space Y , the system need not synthesize every candidate from scratch. Instead, it can retrieve candidates from a structured library and show their executable labels for confirmation.

However, direct adoption of standard retrieval [25, 27, 29] is insufficient. A flat top- K retriever may over-select redundant variants from dense API groups and leave other plausible groups uncovered. Moreover, retrieval still requires query encoding before candidates can be shown. The next section introduces the retrieval-based formulation used by Kollect and the two design challenges that follow from this formulation.

3 System Overview

We present Kollect, a retrieval-based framework for collecting executable supervision through multiple-choice confirmation. Given a user command x_u , Kollect selects a size- K set S of library entries from a command library C

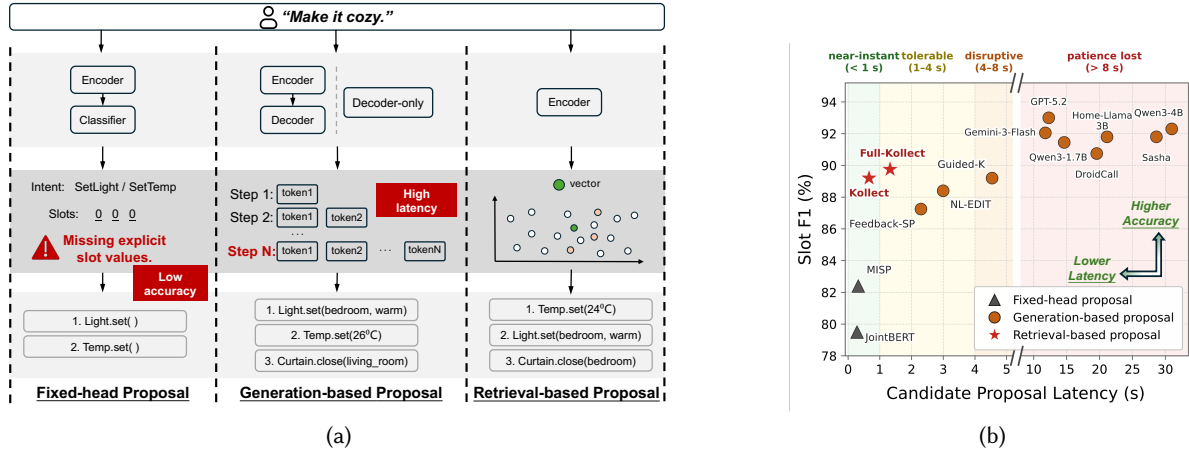


Fig. 2. Motivation for retrieval-based candidate proposal. (a) Fixed-head proposal predicts from a predefined intent-slot inventory, generation-based proposal decodes executable labels token by token, and retrieval-based proposal searches over a bounded executable label space. (b) Accuracy-latency comparison of representative proposal mechanisms shows that our method approaches generation-based quality while remaining in the low latency zone.

and shows their executable labels $Y_S = \{y_c : c \in S\} \subseteq Y$. If the intended label y^* appears in Y_S , the user can select it in one round, and Kollect records (x_u, y^*) in the supervision dataset D . The goal is to cover y^* while keeping proposal latency low on personal devices. This section introduces the retrieval-based formulation (Sec. 3.1), the design challenges (Sec. 3.2), and the resulting architecture and workflow (Sec. 3.3).

3.1 Primer on Retrieval-Based Candidate Proposal

Following Sec. 2, let Y be the user's bounded executable label space. Given an incoming command x_u , let $y^* \in Y$ be the intended executable label and let K be the number of labels shown for confirmation.

Retrieval-based candidate proposal has an *offline* phase and an *online* phase. Offline, the system builds a finite command **library** C . Each library entry $c \in C$ stores a *natural-language command* x_c paired with an *executable label* $y_c \in Y$. An **encoder** maps each library command x_c to a reference embedding v_c . Online, the same encoder maps x_u to a query embedding v_u . A **retriever** searches over library entries using the similarities between v_u and stored embeddings $\{v_c : c \in C\}$. The selected objects are entries $S \subseteq C$, while the objects shown to the user are their executable labels $Y_S = \{y_c : c \in S\}$. A naive retrieval baseline [27, 47] ranks all entries in C by similarity to v_u and returns the top- K entries.

In Kollect, candidate proposal is *closed with respect to the current library* C , i.e., every displayed label must be represented by at least one entry in C . The library can be incrementally updated over time to add newly confirmed labels for subsequent requests (see Sec. 6.3). Each entry also stores two attributes that describe the executable label:

$$c = (x_c, y_c, g_c, p_c, v_c) \quad (1)$$

where $g_c = \text{API}(y_c)$ is the *API group ID* of y_c , and $p_c = \text{Pattern}(y_c)$ is the *slot pattern* of y_c . The slot pattern p_c records the slot structure within an API group while ignoring concrete slot values. For example, `Thermostat.set(AC, 24)` and `Thermostat.set(heater, 20)` share the pattern `Thermostat.set(device, value)`, while `Thermostat.turn_on(AC)` has a different pattern.

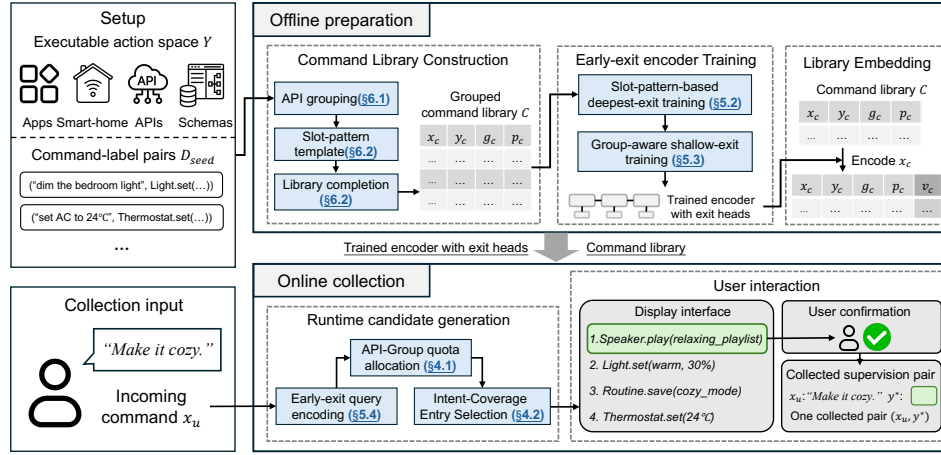


Fig. 3. Collect overview. *Offline*, Collect builds a grouped command library C from seed pairs D_{seed} via API grouping, slot-pattern templating, and library completion, then trains a retriever-aligned early-exit encoder and encodes each entry into a fixed reference embedding v_c . *Online*, the early-exit encoder embeds an incoming command x_u ; the grouped retriever allocates group quotas and performs intra-group selection to form a candidate set $\mathcal{Y}_S$ for user confirmation, recording the confirmed pair (x_u, y^*) in dataset D .

The executable label space in personal assistants is naturally structured by APIs. Collect exploits this structure by grouping entries according to their API group IDs, with $C_g = \{c \in C : g_c = g\}$ for $g \in G$. Thus, C is partitioned into disjoint API groups $\{C_g\}_{g \in G}$. This grouped view makes candidate proposal depend on two coupled decisions, which API groups to consider under budget K and which entries to select within those groups.

3.2 Design Challenges

As discussed in Sec. 2.4, retrieval reduces candidate-proposal latency, but naive top- K retrieval [17, 23, 26] is insufficient. Collect must address two challenges.

- **Challenge #1: Candidate coverage under API ambiguity.** A simple retriever ranks all entries in C by similarity to v_u and returns the top- K entries. However, direct confirmation is possible only if the displayed set contains the intended label y^* ; otherwise, the user completes the label through same-API correction or full fallback. For under-specified commands such as “make it cozy”, several APIs may be plausible, e.g., lighting, thermostat, and speaker. Simple top- K retrieval can therefore spend most slots on near-duplicate entries from one dense group. The challenge is to maintain useful coverage under a small display budget, both across plausible API groups and within each group.
- **Challenge #2: Fast encoding with retriever-compatible embeddings.** Candidate proposal still requires encoding the user command into v_u before any labels can be shown. Running a large model on the confirmation path is too slow for interactive use on personal devices. Simply replacing it with a smaller encoder is infeasible, because the returned embedding is used by the downstream retriever to select entries within and across API groups. Thus, the encoder must reduce latency while preserving the retrieval decisions induced by its embedding.

3.3 Collect Architecture and Workflow

Architecture. Collect addresses the two challenges with two modules supported by one command library.

- **Query-adaptive grouped retriever (Sec. 4).** To address Challenge #1, Kollect does not treat retrieval as flat top- K nearest-neighbor search [7, 21, 27, 47]. Instead, it treats display slots as a budget to allocate across API groups. The retriever first performs adaptive API-group quota allocation, which scores each API group and assigns integer quotas under the total budget K . It then performs intent-coverage entry selection, which fills each non-zero quota with entries that cover different query-relevant regions within the group. This design keeps multiple plausible API groups available while reducing redundant candidates inside each group.
- **Retriever-aligned early-exit encoder (Sec. 5).** To address Challenge #2, Kollect adapts early-exit networks (EENs) [28, 57] to accelerate query encoding. The encoder has multiple exit heads, so shallow exits can return earlier with lower latency. Unlike classification-oriented EENs [57, 63], each exit in Kollect returns an embedding consumed by the retriever. The deepest exit defines the fixed reference embedding space for the library, while shallow exits are trained to preserve the retrieval behavior needed by grouped candidate proposal. At runtime, Kollect stops early only when the group-level retrieval decision is stable across consecutive exits.

The two modules rely on the **command library (Sec. 6)**. Kollect initializes it from observed command-label pairs, groups entries by API, completes missing schema-valid value assignments, and attaches reference embeddings for retrieval. Each confirmed pair is added to D , whereas C requires an update only when the confirmed label is not already represented.

Workflow. Fig. 3 illustrates the end-to-end workflow. Offline, Kollect first constructs the symbolic command library C from the seed supervision set D_{seed} . It then trains the early-exit encoder and encodes every library command with the deepest exit, adding the fixed reference embedding v_c to each entry c . Online, given a user command x_u , the encoder evaluates exits sequentially. At each exit, it produces a candidate query embedding and checks whether the induced group-level retrieval decision has stabilized. Once an exit stops, the returned embedding is denoted by v_u . The grouped retriever then uses v_u and C to construct a size- K entry set S . Finally, Kollect shows Y_S to the user and obtains the confirmed label y^* through one of the three interaction outcomes defined in Sec. 2.2. It records (x_u, y^*) in D and, if y^* is not yet represented in C , incrementally adds a canonical entry for subsequent requests.

4 Query-Adaptive Grouped Retriever

This section addresses Challenge #1, *i.e.*, how to construct a size- K confirmation set that is likely to contain the user's intended executable label even when the target API group is not explicit in the user command. Given the user-command embedding v_u produced by the encoder (see Sec. 5), the retriever selects a size- K entry set $S \subseteq C$ and shows the corresponding executable labels Y_S to the user. Since y^* is unknown before confirmation, the retriever uses query relevance and coverage as surrogate signals to increase the chance that Y_S contains y^* under the fixed display budget K .

The retriever works in two steps (Fig. 4). First, **adaptive API-group quota allocation** (Sec. 4.1) estimates the relevance of each API group to the query embedding and assigns integer quotas under the total budget K . Second, **intent-coverage entry selection** (Sec. 4.2) fills each non-zero quota with entries that add new query-relevant coverage within the group. Together, these steps distribute the display budget across plausible API groups and reduce redundant candidates within each group.

4.1 Adaptive API-Group Quota Allocation

The retriever must allocate the K display slots across API groups. Two naive strategies define the extremes of this allocation. *Global retrieval* [27, 47] ranks all entries in C and returns the top- K results, which can concentrate the display budget in large or dense groups. By contrast, *hard group filtering* [8, 60] predicts a single API group and

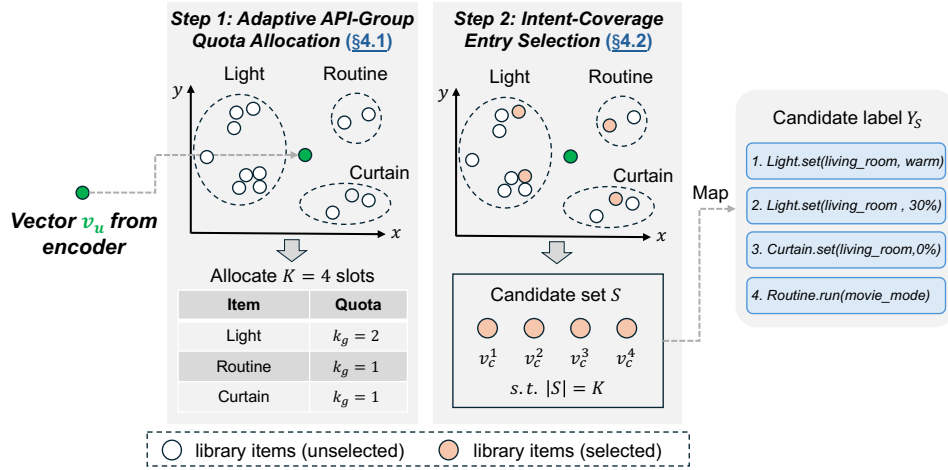


Fig. 4. Illustration of the query-adaptive grouped retriever. Given a query embedding v_u from the encoder, Step 1 (Sec. 4.1) scores each API group by relevance and allocates integer quotas $\{k_g\}$ under the total display budget K , distributing slots across plausible groups rather than concentrating them in one. Step 2 (Sec. 4.2) fills each non-zero quota using a query-conditioned coverage objective, selecting entries that cover different high-relevance regions within the group. The K selected library entries form candidate set S , which maps to the candidate label set Y_S shown to the user.

retrieves only from that group, excluding the intended label whenever the prediction is incorrect (Fig. 5). Instead, Kollect uses a query-adaptive soft allocation. It estimates the relevance of each API group and converts these scores into group-level quotas, preserving multiple plausible groups while assigning more slots to those most relevant to the query.

Group relevance scoring. Given v_u , Kollect computes the query-to-entry similarity for each library entry:

$$s_c = \cos(v_u, v_c), \quad c \in C \quad (2)$$

For each group $g \in G$, let Q_g be the top- q entries in C_g ranked by s_c . The group relevance score is defined as

$$r_g = \frac{1}{|Q_g|} \sum_{c \in Q_g} s_c \quad (3)$$

This score measures how well the most relevant entries in group g match the query. It does not assume that all entries in the same API group form a single compact cluster, which matters because one API group may contain multiple slot patterns or parameter regions (see Sec. 6).

Quota allocation. Kollect converts the group scores $\{r_g\}_{g \in G}$ into integer quotas $\{k_g\}_{g \in G}$ with $\sum_{g \in G} k_g = K$. Kollect first rectifies the scores into non-negative masses $\mu_g = \max(0, r_g)$. If at least one group has positive mass, the masses are normalized into proportions and converted into integer quotas by largest-remainder rounding. If all groups have zero mass, Kollect falls back to the groups with the largest raw scores. Groups with $k_g = 0$ are discarded. If a group receives more quota than its available entries, the unused slots are redistributed to the remaining groups by the same rounding rule. Only groups with non-zero quotas are passed to intent-coverage entry selection. We denote this allocation procedure by $\text{Quota}_K(\cdot)$.

Take $K = 4$ as a toy example. A specific request such as “brighten the light a bit” may assign most slots to the lighting group. By contrast, an under-specified request such as “make it cozy” may spread slots across lighting,

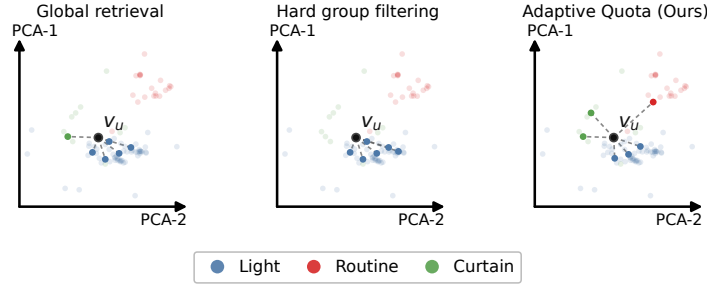


Fig. 5. Comparison of group-quota allocation strategies in PCA-projected embedding space with $K = 6$ setting. Global retrieval [47] selects the K nearest library entries globally, over-allocating display slots to the densest API group (Light) and leaving other plausible groups (Routine, Curtain) uncovered. Hard group filtering [8] restricts retrieval to a single predicted API group, which risks excluding the intended label entirely when the group prediction is wrong. Kollect’s adaptive quota allocation distributes the display budget across all plausible API groups, ensuring coverage of multiple candidate intents under the same budget K .

thermostat, speaker, and routine groups. Thus, adaptive API-group quota allocation replaces naive top- K ranking with query-adaptive budget allocation (see Fig. 4).

4.2 Intent-Coverage Entry Selection

Why ranking and diversity are insufficient. After group g receives quota k_g , the retriever must choose which k_g entries in C_g should be shown. Two standard choices are insufficient.

- A *nearest-neighbor* strategy [27, 47] selects the top- k_g entries by s_c . This maximizes individual relevance, but can spend several slots on near-duplicate parameter variants from the same label region (see Sec. 7.7.1).
- A *diversity-based* strategy [7, 21, 33] encourages selected entries to be far apart. This can reduce redundancy, but pairwise diversity is not conditioned on which regions are relevant to the current query. It may select geometrically different entries that are weakly related to the query.

Therefore, Kollect uses a *query-conditioned* coverage surrogate. The surrogate treats entries with higher query similarity as more plausible and selects entries that represent different high-relevance regions within the group.

Query-conditioned coverage surrogate. For group g , Kollect selects a subset $S_g \subseteq C_g$ with $|S_g| = k_g$. If s_c were a calibrated probability that entry c corresponds to the intended label, selecting the top- k_g entries would be natural. In practice, s_c is an embedding similarity. High-scoring entries can concentrate in one local region of the group, especially when many entries differ only by small parameter changes [7]. Thus, ranking entries independently may waste slots on variants that provide little extra coverage.

Kollect uses s_c to define a query-conditioned relevance weight over entries in the group:

$$w_c = \frac{\max(0, s_c)}{\sum_{c' \in C_g} \max(0, s_{c'})} \quad (4)$$

If the denominator is zero, Kollect uses uniform weights within the group. The weight w_c identifies which entries, and thus which regions of C_g , are more relevant to the current query.

Given a selected set S_g , Kollect defines its coverage as

$$F_g(S_g) = \sum_{c \in C_g} w_c \cdot \max\left(0, \max_{c' \in S_g} \cos(v_c, v_{c'})\right) \quad (5)$$

with $F_g(\emptyset) = 0$. This is a query-conditioned weighted facility-location objective [32]. Each entry c contributes according to both its query relevance w_c and how well it is represented by the selected set. The maximum over selected entries introduces diminishing returns, so adding another entry from an already represented local region gives limited additional gain. As a result, the objective favors entries that represent different high-relevance regions within the group.

Greedy entry selection. The objective in Eq.(5) is a weighted facility-location function with non-negative similarities. It is normalized, monotone, and submodular. Hence, under the cardinality constraint $|S_g| = k_g$, the standard greedy algorithm achieves a $(1 - 1/e)$ approximation to the optimal value of this surrogate objective [42]. Collect uses this greedy procedure to fill each group quota.

Starting from $S_g = \emptyset$, the retriever repeatedly adds the entry with the largest marginal gain:

$$\text{Gain}_g(c; S_g) = F_g(S_g \cup \{c\}) - F_g(S_g), \quad c \in C_g \setminus S_g \quad (6)$$

This process continues until $|S_g| = k_g$. The final selected entry set is the union of the per-group selections $S = \bigcup_{g:k_g>0} S_g$. The executable labels $Y_S = \{y_c : c \in S\}$ are then shown to the user.

Runtime cost. The retriever incurs two types of computation. Quota allocation computes query-to-entry similarities and extracts the top- q entries from each group. Intent-coverage selection then evaluates marginal gains using item-to-item similarities within each active group. The query-to-entry similarities are reused in Eq.(4), whereas only the item-to-item similarities in Eq.(5) are specific to coverage selection. Both computations rely on the auxiliary structures described in Sec. 6.1: an exact index over the fixed reference embeddings $\{v_c : c \in C\}$ for query-to-entry search, and precomputed per-group item-to-item similarity matrices for coverage selection. On the per-command path, exact search time and reference-embedding storage grow linearly with $|C|$. In contrast, constructing or refreshing the in-group similarity matrices is an offline or periodic maintenance cost rather than part of a single candidate-proposal request. Sec. 7.8 evaluates both costs.

5 Retriever-Aligned Early-Exit Encoder

This section addresses Challenge #2 via an early-exit encoder for the grouped retriever in Sec. 4. Given a user command x_u , the encoder returns a query embedding v_u that supports API-group scoring, quota allocation, and intent-coverage entry selection. Unlike *classification*-oriented early-exit networks (EENs) [57, 63], where each exit predicts a label, each exit in Collect returns an embedding whose quality is determined by retrieval.

As shown in Fig. 6, Collect adapts early exiting to retrieval through slot-pattern-based deepest-exit training, group-aware shallow-exit training, and a group-stability exit policy. For **exit training**, Sec. 5.2 trains the **deepest** exit as a slot-pattern-based retrieval teacher, and Sec. 5.3 trains **shallow** exits to query the same fixed reference library while preserving the teacher's group-level behavior. For **exit policy**, Sec. 5.4 stops only when the quota support, *i.e.*, the API groups receiving non-zero quotas, become stable across consecutive exits.

5.1 Design Space and Requirements for Retrieval

Following the notations from Sec. 3.1, an early-exit encoder augments a fixed backbone with multiple exit heads. Let E_j be the backbone portion up to exit j , and let h_j be the corresponding exit head. For a library entry $c \in C$ and a user command x_u , exit j produces $v_c^j = h_j(E_j(x_c))$ and $v_u^j = h_j(E_j(x_u))$. If the policy stops at exit j , the returned query embedding is $v_u = v_u^j$.

In standard EENs, each exit outputs a *label* distribution, and the exit policy stops when the current prediction is *confident* [57, 63]. This design is not aligned with Collect, because the exit output is consumed by the grouped retriever, which imposes three requirements.

- **Deepest-exit training should define a retrieval space.** Since Collect stores the deepest-exit embeddings as the fixed reference library, the deepest exit should organize entries by intent similarity rather than by

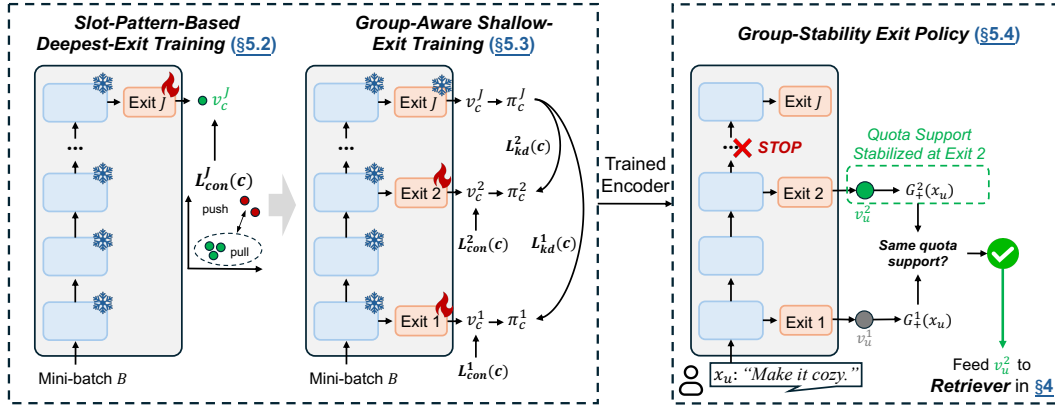


Fig. 6. Retriever-aligned early-exit encoder. Offline, Kollect first trains the deepest exit to define a slot-pattern-based retrieval space whose embeddings serve as the fixed reference library. It then trains shallow exits to query the same reference library while preserving both slot-pattern geometry and API-group ranking behavior. Online, the group-stability exit policy evaluates exits sequentially and stops once the quota support remains unchanged across consecutive exits, after which the stabilized query embedding is passed to the grouped retriever.

a classification boundary. Kollect uses the slot pattern p_c for this purpose. Here, $p_c = \text{Pattern}(y_c)$ is the group-aware slot structure of y_c , ignoring concrete slot values. For example, two thermostat labels that set different temperatures but share the same slots [Device, Value] have the same slot pattern, while a thermostat on/off label with only [Device] has a different slot pattern.

- **Shallow-exit training should preserve retriever behavior.** A shallow embedding must be compatible with the fixed deepest-exit reference library. More importantly, it should preserve API-group relevance for quota allocation and slot-pattern-level geometry for intent-coverage entry selection.
- **Exit policy should align with quota allocation.** The policy should not decide whether a top-1 label prediction is confident. Instead, it should decide whether the current embedding induces a stable group-level retrieval decision.

5.2 Slot-Pattern-Based Deepest-Exit Training

The deepest exit *defines* the reference retrieval space used by Kollect. The backbone is fixed throughout training, so deepest-exit training learns only the deepest projection head h_J . After training, every library entry $c \in C$ is encoded by the deepest exit, and its embedding is stored as the fixed reference vector used by runtime retrieval and shallow-exit training. We train the deepest exit with supervised contrastive learning [30], because the goal is not to predict a class label but to shape an embedding space where entries with similar intent structures are close and unrelated entries are separated.

Positive-pair granularity. The central design choice is the *granularity* of positive pairs. Exact-label positives are too fine-grained because labels with different concrete slot values may still express the same executable-intent structure. API-group positives are too coarse-grained because one API group can contain multiple slot structures and parameter semantics [60]. Kollect therefore uses *slot patterns* as the positive-pair granularity.

Contrastive objective. For a mini-batch B and entry $c \in B$, define the positive set $P_c = \{c' \in B : p_{c'} = p_c, c' \neq c\}$ and the contrast set $A_c = B \setminus \{c\}$. For anchors with $P_c \neq \emptyset$, the deepest exit is trained with

$$L_{\text{con}}^J(c) = -\frac{1}{|P_c|} \sum_{c^+ \in P_c} \log \frac{\exp(\cos(v_c^J, v_{c^+}^J)/\tau)}{\sum_{c' \in A_c} \exp(\cos(v_c^J, v_{c'}^J)/\tau)} \quad (7)$$

where τ is the temperature. This objective pulls entries with the same slot pattern together and separates entries with different slot patterns. We use slot-pattern-balanced sampling so that each mini-batch contains multiple slot patterns and, whenever possible, at least two entries per slot pattern. If a rare singleton slot pattern leaves P_c empty in a batch, the corresponding anchor is omitted from the contrastive term.

Fixed reference library. After training, the deepest exit is frozen. For every library entry c , Kollect stores $v_c = v_c^J = h_J(E_J(x_c))$ as an attribute of c . Thus, Kollect maintains one fixed reference library produced by the deepest exit, rather than a separate vector library for each exit.

5.3 Group-Aware Shallow-Exit Training

Shallow exits are trained to approximate the retrieval behavior of the deepest exit when querying the same fixed reference library. This design is important because early stopping changes only the query embedding v_u^j , while the stored library embeddings $\{v_c : c \in C\}$ and the retriever remain unchanged. We train only the shallow exit heads $\{h_j\}_{j=1}^{J-1}$, keeping the backbone, deepest head, and reference library fixed.

API-group ranking distillation. We first align shallow exits with the teacher on *API-group relevance*. For any query embedding z and group g , let $Q_g(z)$ be the top- q entries in C_g ranked by $\cos(z, v_{c'})$, where $v_{c'}$ is the fixed deepest-exit embedding stored in C . The group-scoring operator is

$$R_g(z) = \frac{1}{|Q_g(z)|} \sum_{c' \in Q_g(z)} \cos(z, v_{c'}) \quad (8)$$

When a library entry c is used as the query during training, we remove c from its own group candidate set whenever another reference entry remains, avoiding a trivial self-match.

For exit j and anchor entry c , the group score induced by the shallow embedding is $r_{c,g}^j = R_g(v_c^j)$, where $g \in G$. Stacking $\{r_{c,g}^j\}_{g \in G}$ gives the group-score vector r_c^j , which is converted into a distribution over groups:

$$\pi_c^j = \text{softmax}(r_c^j / \tau_d) \quad (9)$$

where τ_d is the distillation temperature. The teacher distribution π_c^J is computed in the same way using the deepest-exit embedding $v_c = v_c^J$.

Each shallow exit is trained to match the teacher's group-score distribution:

$$L_{\text{kd}}^j(c) = \text{KL}(\pi_c^j \parallel \pi_c^J) \quad (10)$$

This term preserves the group-ranking behavior used for API-group quota allocation.

Slot-pattern geometry preservation. Group-ranking distillation alone does not preserve the geometry needed inside each active group. The retriever's intent-coverage selector uses item-to-item similarities to avoid spending several slots on redundant local variants. Therefore, each shallow exit also receives the *slot-pattern contrastive loss* $L_{\text{con}}^j(c)$, i.e., Eq.(7) with v^J replaced by v^j . This term encourages shallow embeddings to organize entries by the same slot-pattern structure as the deepest retrieval space, so that the intent-coverage selector remains effective when an early exit is used.

Overall shallow-exit objective. For a mini-batch B , the shallow exits are trained with

$$L_{\text{shallow}} = \sum_{c \in B} \sum_{j=1}^{J-1} \left[\lambda L_{\text{con}}^j(c) + (1 - \lambda) L_{\text{kd}}^j(c) \right] \quad (11)$$

where λ balances slot-pattern geometry preservation and API-group ranking distillation. Anchors with empty positive sets are omitted from the corresponding contrastive terms, as in deepest-exit training.

5.4 Group-Stability Exit Policy

After training, Kollet must decide when a shallow exit is reliable enough to return. Because the grouped retriever consumes an embedding rather than a label distribution, entropy [37, 57, 63], and top-1 prediction stability [70] are inappropriate stopping criteria.

Quota-support criterion. Kollet uses a *quota-support stability* policy. At runtime, exits are evaluated sequentially. For a user command x_u , exit j produces v_u^j . Using Eq.(8), Kollet computes $r_{u,g}^j = R_g(v_u^j)$ for each $g \in G$, and applies the allocation rule in Sec. 4.1 to obtain $\{k_g^j\}_{g \in G} = \text{Quota}_K(\{r_{u,g}^j\}_{g \in G})$, where $\sum_{g \in G} k_g^j = K$. Let $G_+^j(x_u) = \{g \in G : k_g^j > 0\}$ be the resulting quota support, i.e., the API groups passed to intent-coverage selection.

The first exit initializes the quota support. From the second exit onward, Kollet stops at exit j when $G_+^j(x_u) = G_+^{j-1}(x_u)$. Otherwise, computation continues to the next exit, and the deepest exit returns automatically if no shallow exit satisfies the criterion. As shown in Fig. 7, this criterion allows most commands to exit before the deepest head $j = 6$. Conversely, top-1 stability forces nearly all commands through the full encoder because dense, near-duplicate library entries cause the nearest neighbor to change across consecutive exits. The returned embedding is denoted by v_u and passed to the grouped retriever in Sec. 4.

Approximation to deepest-exit retrieval. The policy tests agreement between consecutive exits rather than equality with the deepest exit. Hence, the resulting candidate set is a practical approximation to the deepest-exit result. Exact agreement between query embeddings is unnecessary. The candidate output remains stable when representation differences do not change the allocated quotas or the entries selected within each active group. Disagreement is more likely when candidates near an allocation or selection boundary have similar scores, because a small representation change can reorder them. Sec. 7.7.2 measures the per-command agreement between early-exit and deepest-exit candidate sets.

Runtime cost. The policy performs group scoring and quota allocation at each evaluated exit, but executes full intent-coverage selection only once after stopping. Thus, it reuses the same fixed reference library without repeatedly constructing complete candidate sets at intermediate exits. Proposal latency depends on both how early the quota support stabilizes and the cost of evaluating the API groups. Sec. 7.6 measures the effect of increasing $|G|$ on proposal latency.

6 API-Grouped Command Library

This section constructs the command library C used by the grouped retriever (Sec. 4) and the early-exit encoder (Sec. 5). Starting from an offline seed supervision set $D_{\text{seed}} = \{(x_n, y_n)\}_{n=1}^N$, where each y_n is a canonical executable label, Kollet first initializes a symbolic library with observed command-label pairs, API groups, and slot patterns. It then completes the library within each observed (API, slot-pattern) pair by generating schema-valid but unseen value assignments. This completion broadens retrieval coverage without requiring D_{seed} to contain every concrete parameter combination. After the deepest exit is trained, Kollet encodes each library command and stores its embedding v_c as an additional entry attribute. Fig. 8 shows the workflow of command library construction.

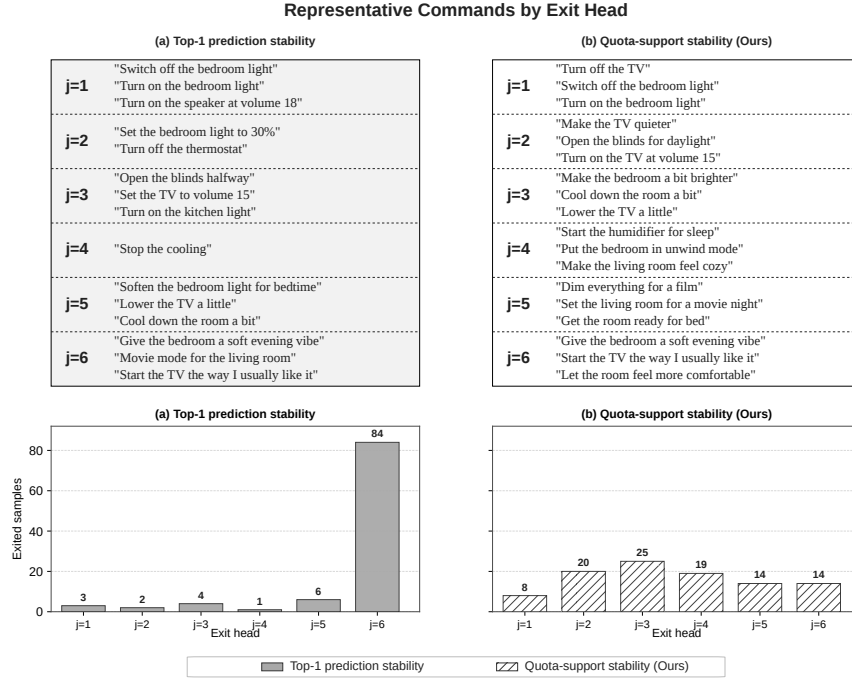


Fig. 7. Early-exit behavior under two stopping criteria on the Home scenario. (*Top*) Commands at each exit head for top-1 prediction stability (left) and Kollekt’s quota-support stability (right). Unambiguous commands exit early; under-specified or multi-intent commands require deeper encoding to stabilize the group-level retrieval decision. (*Bottom*) Distribution of exit heads under top-1 prediction stability (left, solid bars) versus Kollekt’s quota-support stability (right, hatched bars). Top-1 stability forces 84 of 100 samples to the deepest exit $j = 6$, as near-duplicate entries cause the top-1 neighbor to shift across exits. Quota-support stability spreads exits across all heads, with many samples stopping at $j = 2-3$.

6.1 Grouped Library Initialization

Observed entries. For each observed pair $(x_n, y_n) \in D_{\text{seed}}$, Kollekt creates or updates the canonical entry associated with y_n . The entry stores the symbolic attributes $x_c = x_n$, $y_c = y_n$, $g_c = \text{API}(y_n)$, and $p_c = \text{Pattern}(y_n)$. As defined in Sec. 3.1, x_c is the canonical surface form, y_c is the executable label, g_c is the API group ID, and p_c is the slot pattern. The library retains at most one canonical entry for each executable label. When multiple observed commands map to the same label, one is retained as the canonical surface form and the remaining commands are used as training examples.

API grouping. The grouped retriever requires a library partition in which each group corresponds to one executable API. Exact-label grouping is too fine-grained, because it creates one group for each concrete parameter assignment and makes the group count scale with the executable label space. Hence, Kollekt uses the API attribute as the group ID. For an entry c , $g_c = \text{API}(y_c)$, and entries with the same group ID form $C_g = \{c \in C : g_c = g\}$. Thus, C is partitioned into disjoint API groups $\{C_g\}_{g \in G}$.

Embedding attachment. The initial library is symbolic. After training the deepest exit, Kollekt attaches to each entry a 256-dimensional, ℓ_2 -normalized FP16 reference embedding $v_c = v_c^J = h_J(E_J(x_c))$, where $h_J(E_J(\cdot))$ is the deepest encoder exit defined in Sec. 5.1. The symbolic attributes and slot patterns supervise encoder training, whereas runtime retrieval uses the resulting fixed reference embeddings.

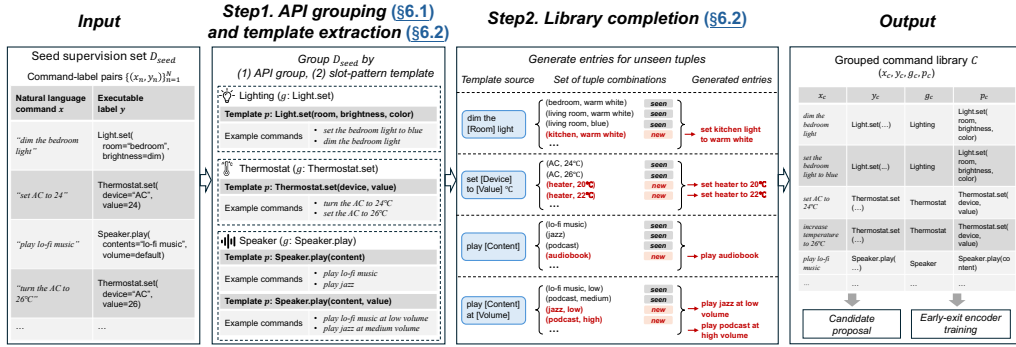


Fig. 8. Command library construction from seed supervision set D_{seed} . Step 1 groups observed command-label pairs by API group g and extracts a slot-pattern template p for each group, partitioning D_{seed} into disjoint API groups $\{C_g\}_{g \in G}$. Step 2 completes the library by enumerating schema-valid value assignments $\Omega(g, p)$ for each observed (API, slot-pattern) pair, generating new entries for parameter combinations not covered by D_{seed} . The resulting symbolic library C stores each entry as a tuple (x_c, y_c, g_c, p_c) and serves as the training corpus for the early-exit encoder; after encoder training, each entry is augmented with its fixed reference embedding v_c to form the complete canonical entry $c = (x_c, y_c, g_c, p_c, v_c)$.

Auxiliary retrieval structures. For exact query-to-entry search, Kollekt stores all reference embeddings $\{v_c : c \in C\}$ in a FAISS Flat index (IndexFlatIP) [11]. Because the query and reference embeddings are ℓ_2 -normalized, inner-product search is equivalent to cosine-similarity search. The index requires neither training nor approximate-search hyperparameters, and each indexed entry retains its API-group metadata for the relevance scoring and quota allocation in Sec. 4.1. For intent-coverage selection, Kollekt also stores the upper-triangular part of the symmetric item-to-item cosine-similarity matrix for each group $M_g[a, b] = \cos(v_a, v_b)$ for $a, b \in C_g$. These matrices allow the weighted facility-location objective in Sec. 4.2 to reuse item-to-item similarities. Their total storage is $O(\sum_{g \in G} |C_g|^2)$ and is approximately 0.27 MB at $|C| = 2,000$.

6.2 Schema-Guided Slot-Pattern Completion

A retrieval-based system can only display labels represented in C , yet seed supervision typically covers only frequent commands and popular parameter values, leaving many schema-valid assignments absent. Enumerating all argument combinations under each API is insufficient, because different commands under the same API may require different slot structures. For example, one command may specify only a device, while another specifies both a device and a value. Hence, Kollekt completes the library at the *slot-pattern* level, filling missing assignments only within (API, slot-pattern) pairs observed as valid in D_{seed} .

- (1) **Template extraction.** For each observed (API, slot-pattern) pair (g, p) , Kollekt constructs one natural-language template. It selects a representative command from the observed entries with group g and pattern p , preferring more frequent command forms and breaking ties by shorter command length. Concrete parameter mentions are replaced with slot placeholders to form a template t_g^p . For example, the command "set the bedroom AC to 24°C" with group AC.set and pattern [Device, Value] yields the template "set the [Device] AC to [Value] °C".
- (2) **Value-assignment enumeration.** For each pair (g, p) , Kollekt uses the API schema to construct the set $\Omega(g, p)$ of valid slot assignments. Each $\omega \in \Omega(g, p)$ assigns one valid value to every slot in p .

- Discrete slots, including categorical and high-cardinality fields such as devices, contacts, and playlists, are enumerated directly from the values declared in the schema. Each declared value remains a distinct slot value and is not quantized.
 - Continuous slots are discretized to keep the library finite. For a slot with valid range $[l_s, u_s]$, Collect enumerates values beginning at l_s with interval Δ_s , using the resolution declared by the API or device schema and defaulting to one native unit when no resolution is provided. For example, an interval of 1°C represents 24, 25, and 26°C , but not 24.1°C . Numeric values are compared exactly and without tolerance. Thus, a candidate `Thermostat.set(AC, 24)` for $y^* = \text{Thermostat.set(AC, 24.1)}$ matches the API and device but mismatches the value field. The interval Δ_s determines which values can be proposed directly from the current library, not which schema-valid values can be confirmed. A missing value can be supplied through same-API correction or full fallback at the precision supported by the schema. The confirmed pair is recorded in D , and a subsequent incremental update may add a canonical entry to C for later requests (Sec. 6.3).
- (3) **Filling uncovered combinations.** For each assignment $\omega \in \Omega(g, p)$ whose executable label is not already represented in C , Collect instantiates the template t_g^p and constructs the corresponding executable label. The generated entry is added to C with $x_c = t_g^p(\omega)$, $y_c = \text{Label}(g, p, \omega)$, $g_c = g$, and $p_c = p$.

After completion, C contains both observed entries from D_{seed} and generated entries derived from schema-valid assignments. Observed entries preserve natural command phrasing, while generated entries extend coverage to executable labels absent from the seed supervision.

6.3 Incremental Library Update

Update policy. Each confirmed pair (x_u, y^*) is appended to the supervision dataset D . A candidate-set miss triggers library maintenance only when y^* is not already represented in C . Otherwise, x_u is retained as an additional training example. Maintenance occurs after confirmation and may be applied in periodic batches, so any change to C affects only subsequent requests.

Maintenance operations. Library maintenance supports three operations on a canonical entry.

- **Add.** For a newly confirmed label, Collect creates an entry with $x_c = x_u$, $y_c = y^*$, $g_c = \text{API}(y^*)$, and $p_c = \text{Pattern}(y^*)$. It generates the deepest-exit embedding v_c , inserts it into the FAISS index, and appends its similarities to the existing entries in C_{g_c} to the corresponding in-group matrix.
- **Modify.** If the canonical surface form changes, Collect regenerates the deepest-exit embedding, replaces the indexed vector, and recomputes the affected row and column of the in-group matrix. If only the executable label changes, the existing embedding is reused while y_c , g_c , and p_c are updated. When the API group changes, the entry is removed from the old group's matrix and its similarities within the new group are computed.
- **Delete.** Collect removes the entry from C and the FAISS index, updates the entry-to-index mapping, and deletes the corresponding row and column from its in-group matrix.

Only Add and a Modify that changes the canonical surface form require a new reference embedding and thus incur encoder-dependent cost.

7 Evaluation

7.1 Experimental Setup

Datasets. We construct preference-conditioned executable-supervision datasets where different user profiles map commands to preferred labels over the same scenario action space. This evaluation requires (i) consistent

user preferences across the collection and test splits, and (ii) under-specified commands whose intended labels depend on those preferences. No existing dataset provides both properties at sufficient scale.

Following Sasha [31], we use an LLM-assisted pipeline to construct natural-language commands for two scenarios. For Home, we adapt Sasha’s most complex smart-home template, which contains 17 devices, 7 areas, diverse slot types, single-device commands, and multi-device routines. For Phone, we derive an action ontology from SLURP [3] intents and slots and augment it with multi-API templates. The pipeline varies action templates, user profiles, APIs, slot combinations, command phrasing, and degrees of under-specification, including commands that omit preference-dependent fields.

The commands are divided into a seed pool and a personalization pool. The seed pool forms D_{seed} , which is used for library initialization, encoder training, and baseline setup. The personalization pool is annotated under five fixed preference profiles and divided into a collection set D_{col}^s and a test set D_{test}^s . In each scenario, D_{seed} contains 2,000 generic examples. The personalization pool contains 300 examples, with 60 examples for each profile, and is split into 180 collection examples and 120 test examples.

This construction enables reproducible evaluation but may not fully capture naturally occurring expressions. Hence, we also evaluate on human-authored commands mapped from HWU64 [38] and SLURP_real [3] (Sec. 7.5).

Candidate-proposal methods. We compare methods by how they propose an executable label or a candidate label set for an incoming command x_u . For methods that output a single label, we denote the proposed label by $\hat{y}$. For methods that output a candidate set, we denote the displayed executable labels by Y_S when they are selected from the library, or by $\hat{Y}_K$ when they are generated directly.

- **Non-interactive proposal.** These methods predict one executable label from x_u without user feedback.
 - **Intent-slot classification.** JointBERT [8] predicts an intent and its slots from a fixed output space. This baseline represents efficient fixed-head semantic parsing.
 - **Text generation.** Seq2Seq-T5 [49] fine-tunes a T5 seq2seq model to generate the executable label. DroidCall [62] represents an LLM-based parser that generates an executable action from the input command. These baselines test whether direct generation can provide sufficient quality under interactive latency constraints.
- **Interactive proposal.** These methods involve user feedback before the final label is recorded.
 - **Binary confirmation.** MISP [65] and Feedback-SP [24] present one candidate at a time and receive accept/reject feedback. In our one-round evaluation, they are scored using the first proposed executable label.
 - **Multiple-choice confirmation.** Guided-K [4], Home-Llama-3B [22], Qwen3-1.7B [64], Qwen3-4B [64], Gemini-3-Flash [18], and GPT-5.2 [44] generate a size- K candidate label set $\hat{Y}_K$. The user then selects one label from the displayed set.
 - **Natural-language correction.** NL-EDIT [12] refines an initial prediction using natural-language corrective feedback. Sasha [31] uses a multi-call LLM pipeline for clarification and planning; we instantiate it with Qwen3-1.7B as the backend LLM. These baselines represent richer but potentially slower correction-style interaction.
- **Kollet variants.** Full-Kollet uses the same grouped command library and query-adaptive grouped retriever as Kollet, but always runs the deepest encoder exit. Kollet uses the group-stability exit policy in Sec. 5 to return earlier when the quota support stabilizes.

Kollet implementation. We implement the Kollet encoder with Qwen3-Embedding-0.6B [69] as the backbone, attaching six early exits at transformer layers {4, 9, 14, 19, 23, 27}. Each exit projects its intermediate representation to a 256-dimensional, ℓ_2 -normalized query embedding. The deepest exit encodes the fixed reference embeddings stored in a FAISS Flat index [11], and shallow exits are trained to preserve retrieval behavior in this reference

space. At inference, every exit searches the same index, and encoding stops when the API-group support stabilizes. We set the group-scoring pool size to $q=10$ (Eq.(3)).

Qwen3-4B-Instruct-2507 [64] performs offline weak labeling, executable-label canonicalization, and command paraphrase synthesis. It also serves as the downstream task model but never runs on Collect's online candidate-proposal path. Candidate-proposal latency is measured on a Google Pixel 10 and an NVIDIA Jetson Orin Nano to represent mobile and edge-device deployments. Unless otherwise stated, the Home and Phone scenarios use the same encoder architecture, grouped retriever, and early-exit policy.

Training and validation. The Qwen3-Embedding-0.6B backbone remains frozen while the six exit projection heads are trained with AdamW. We use a learning rate of 2×10^{-4} , weight decay 0.01, batch size 256, 10 epochs, 10% linear warmup, and gradient clipping at 1.0. The contrastive objectives use temperature $\tau = 0.07$. Group-ranking distillation uses temperature $\tau_d = 2.0$ and weight $\lambda = 0.5$, assigning equal weight to slot-pattern geometry and API-group ranking.

Encoder training and downstream task-model fine-tuning are performed on 8×NVIDIA A800 GPUs. All hyperparameters are selected using validation data and then fixed across users within each scenario.

Evaluation protocol and metrics. For offline evaluation, each pair $(x_u, y^*) \in D_{\text{col}}^s$ is treated as one annotation request. A method returns either a single executable label or a size- K candidate set. For candidate-set methods, we score the displayed set directly without simulating which candidate a user would select. We refer to this protocol as *offline candidate-set quality evaluation*.

Annotation quality is measured using API accuracy and Slot F1, as defined in Sec. 2.3. Proposal latency is measured from receiving x_u to returning the single label or displaying the candidate set. The offline weak-labeling model is used to provide scalable initialization supervision. Annotation quality is computed against the fixed reference label y^* in each evaluation pair rather than against the weak-labeling model's output at evaluation time.

Downstream personalization. The supervision collected on D_{col}^s is used to personalize a fixed downstream task model, which is then evaluated on D_{test}^s . Unless otherwise stated, *fine-tuning* is used for adaptation, so the downstream model and adaptation procedure remain fixed when candidate-proposal methods are compared.

We additionally instantiate RAG and agent memory to test whether the collected supervision remains useful beyond fine-tuning. In these offline experiments, the confirmed dataset $D = \{(x_u, y^*)\}$ is the sole interface between candidate collection and downstream personalization. The three strategies are implemented as follows.

- **Fine-tuning.** We adapt the downstream task model using LoRA on all projection modules. We use rank $r = 16$, scaling factor $\alpha = 32$, and dropout 0.05. Optimization uses AdamW with learning rate 2×10^{-5} , weight decay 0.01, batch size 4 with 8 gradient-accumulation steps, 3 epochs, cosine learning-rate decay, 3% warmup, maximum sequence length 1024, bf16 precision, and gradient clipping at 1.0. Fine-tuning updates the task-model weights using the confirmed pairs and requires no retrieval at inference.
- **RAG.** Following a LaMP-style in-prompt augmentation pipeline [50], each confirmed pair is stored as one entry in a flat user profile. A separate facebook/contriever encoder [25] embeds the current command and the stored commands. We retrieve the top $k = 4$ entries by embedding similarity from a separate FAISS index and prepend the retrieved command-label pairs as input-output demonstrations to the task instruction and current command.
- **Agent memory.** We adapt the hierarchical preference memory of Me-Agent [59]. Level 1 stores one descriptor for each API group, while each group's Level 2 memory stores the confirmed pairs associated with that group. Each Level 2 record contains x_u, y^* , and the slot-value assignments parsed from y^* . We use Contriever to build one FAISS index over the Level 1 descriptors and a separate Level 2 index for each API group. For a new command, the system first retrieves the most relevant API group and then searches the corresponding Level 2 memory. The retrieved records are inserted into the task-model prompt for final label generation.

Table 2. Primary collection results across Home and Phone scenarios under the top-1 proposal setting ($K = 1$). This is the most stringent candidate-proposal setting because only one executable label is displayed. **API Acc.** measures whether the API group of the proposed label matches the intended label; **Slot F1** evaluates slot-value correctness at the parameter level; **Avg Latency** is the proposal latency measured on Jetson Orin Nano 8GB. Latency bands are indicated by superscript letters: ^a<1 s (near-instant [40, 55]); ^b1–4 s (tolerable [16]); ^c4–8 s (disruptive [16]); ^d>8 s (patience lost [1, 55]).

Type	Method	Home Scenario			Phone Scenario		
		API Acc.(%)	Slot F1(%)	Avg Latency (ms)	API Acc.(%)	Slot F1(%)	Avg Latency (ms)
Non-ISP	JointBERT	85.2	80.4	291 ^a	83.5	78.6	274 ^a
	Seq2Seq-T5	90.4	87.0	2,527 ^b	89.8	86.5	2,411 ^b
	DroidCall	92.9	90.6	19,184 ^d	93.4	90.9	19,963 ^d
ISP	Feedback-SP	89.5	86.8	2,319 ^b	90.7	87.7	2,276 ^b
	MISP	86.5	82.0	333 ^a	87.4	82.8	309 ^a
	Guided-K	92.3	89.2	4,627 ^c	92.0	89.2	4,452 ^c
	Home-Llama-3B	94.2	91.8	21,136 ^d	—	—	—
	Qwen3-1.7B	93.4	91.2	14,528 ^d	93.7	91.7	14,732 ^d
	Qwen3-4B	94.8	92.5	30,683 ^d	94.3	92.1	31,264 ^d
	Gemini-3-Flash	94.1	92.0	11,587 ^d	94.2	92.1	11,946 ^d
	GPT-5.2	95.3	93.2	12,374 ^d	95.0	92.8	12,191 ^d
	NL-EDIT	90.8	88.2	3,126 ^b	91.3	88.6	2,868 ^b
	Sasha	94.3	92.2	28,173 ^d	93.6	91.4	29,108 ^d
	Full-Kollect	92.6	89.8	1,278 ^b	92.3	89.7	1,369 ^b
	Kollect	92.1	89.4	653 ^a	91.9	89.0	668 ^a

RAG and agent memory build separate retrieval structures over D and do not access Kollect’s command library, grouped retriever, or early-exit embeddings. Sec. 7.4 tests the utility of the collected supervision under all three strategies, while the user study in Sec. 7.9 uses agent memory.

7.2 Main Results

Tab. 2 compares Kollect with non-interactive and interactive baselines on Home and Phone. We first evaluate top-1 proposal ($K = 1$), which isolates the accuracy-latency trade-off when only one label can be displayed. Larger display budgets, corresponding to the full multiple-choice interface, are studied in Sec. 7.3. Under $K = 1$, API Acc. and Slot F1 compare the single proposed label with the intended label y^* . For all methods, API Acc. and Slot F1 are averaged over five independent runs. Following prior HCI studies on response-time perception [1, 16, 40, 55], we categorize latency as near-instant (< 1 s), tolerable (1–4 s), disruptive (4–8 s), or patience-lost (> 8 s).

The key results and takeaways are as follows.

- **Quality among near-instant methods.** Kollect provides the strongest proposal quality among near-instant methods in both scenarios. On Home, JointBERT and MISP require only 291 ms and 333 ms, respectively, but achieve 85.2%/80.4% and 86.5%/82.0% API Acc./Slot F1. At 653 ms, Kollect improves these results to 92.1% API Acc. and 89.4% Slot F1. Thus, Kollect notably improves annotation quality over efficient fixed-head methods while remaining in the near-instant latency zone.
- **Comparison with generation-based methods.** Compared with Guided-K, the strongest generation-based candidate-set baseline below the patience-lost threshold, Kollect achieves similar API accuracy (92.1% vs. 92.3%) while reducing latency from 4,627 ms to 653 ms, a $7.1\times$ speedup. It is also $3.6\times$ – $4.8\times$ faster than Seq2Seq-T5, Feedback-SP, and NL-EDIT while matching or exceeding their proposal quality. Large local and cloud LLM baselines achieve higher absolute accuracy but incur substantially longer latency. Qwen3-4B, Gemini-3-Flash, GPT-5.2, and Sasha require 11.6–30.7 s on Home, compared with 653 ms for Kollect. Across these methods, Kollect is $17.8\times$ – $47.0\times$ faster, with gaps of only 2.0–3.2 points in API accuracy and 2.6–3.8

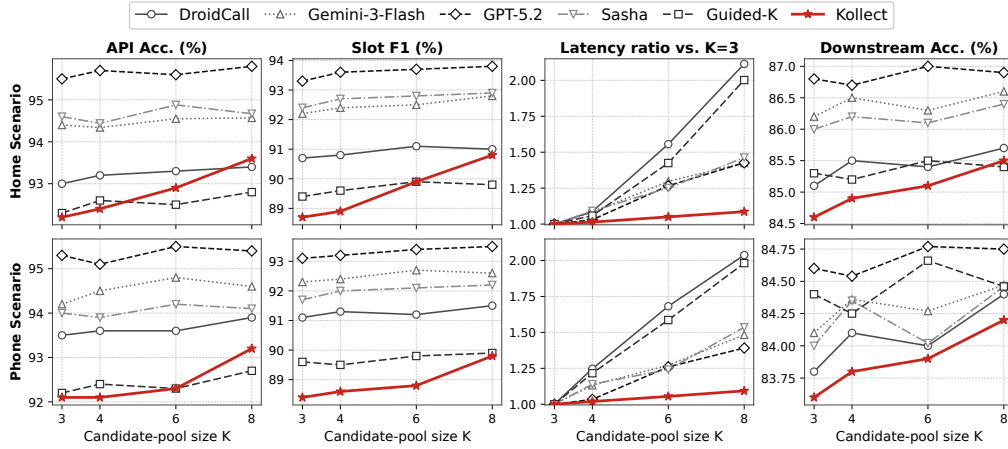


Fig. 9. Effect of display budget K on API accuracy, Slot F1, normalized proposal latency, and downstream accuracy. Rows correspond to Home and Phone scenarios. Latency is normalized by each method's value at $K=3$. Kollect improves consistently as K increases, with little additional latency.

points in Slot F1. DroidCall follows the same pattern. It improves API accuracy by 0.8 points over Kollect but is $29.4\times$ slower.

- **Effect of early-exit encoding.** Full-Kollect uses the same command library and grouped retriever as Kollect but always runs the deepest encoder exit. On Home, it achieves 92.6% API accuracy and 89.8% Slot F1 at 1,278 ms. The group-stability exit policy reduces latency to 653 ms while decreasing API accuracy and Slot F1 by only 0.5 and 0.4 points, respectively. Thus, early exiting moves candidate proposal from the tolerable to the near-instant zone with little loss in annotation quality.
- **Cross-scenario consistency.** The Phone results follow the same pattern. Kollect achieves 91.9% API accuracy and 89.0% Slot F1 at 668 ms, outperforming the near-instant fixed-head baselines in quality and substantially reducing latency relative to generation-based methods. Overall, Kollect consistently keeps top-1 proposal below one second while retaining most of the quality of slower generation-based methods.

7.3 Effect of Display Budget K

The main results in Sec. 7.2 use top-1 proposal ($K=1$). Here, we evaluate the full multiple-choice setting with $K \in \{3, 4, 6, 8\}$. For each method, we report API accuracy, Slot F1, downstream accuracy, and proposal latency normalized by its value at $K=3$. This normalization highlights how each proposal mechanism scales as the interface allows more candidate labels.

The key results and takeaways are as follows.

- **Collection quality.** Fig. 9 shows that Kollect benefits consistently from larger display budgets. On Home, API accuracy increases from 92.2% at $K=3$ to 93.6% at $K=8$, Slot F1 increases from 88.7% to 90.8%, and downstream accuracy increases from 84.6% to 85.5%. The Phone scenario shows the same upward trend. These gains support the design of the grouped retriever. When more slots are available, Kollect can allocate candidates to additional plausible API groups and cover more query-relevant slot-pattern regions within each group. Thus, increasing K does not merely show more labels. It gives the retriever more room to resolve API ambiguity.
- **Comparison with generation-based candidate sets.** Generation-based methods do not always improve monotonically with larger K . When a model is prompted to output more candidates, the generated list for a

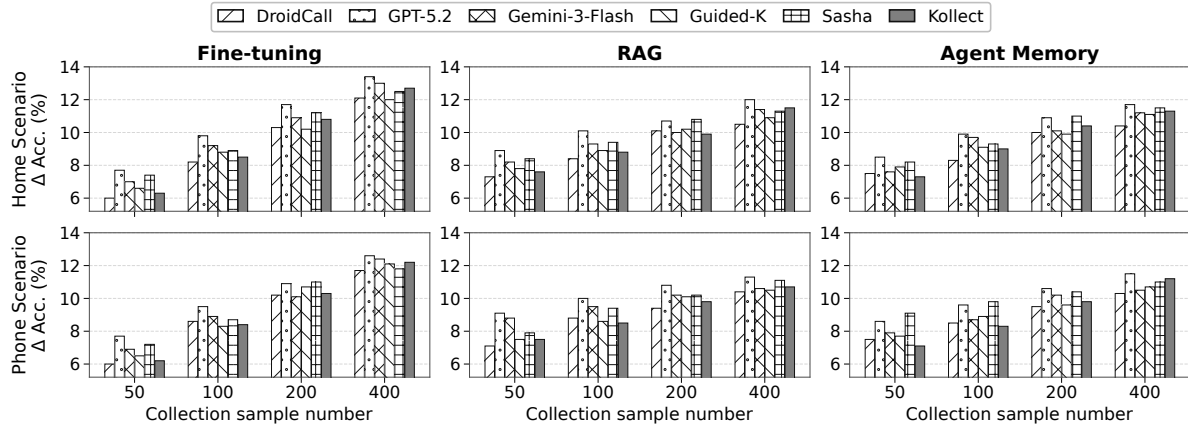


Fig. 10. Downstream accuracy gains as the number of collected command-label pairs increases. Rows correspond to Home and Phone, and columns to fine-tuning, RAG, and agent memory. Each bar reports the gain over the unadapted task model for $n \in \{50, 100, 200, 400\}$ collected pairs.

larger K is not constrained to contain the list generated for a smaller K . Earlier token choices and candidate ordering can change, so accuracy may fluctuate rather than steadily increase. For example, Guided-K’s downstream accuracy drops from 85.3% to 85.2% when K changes from 3 to 4 on Home, and from 84.4% to 84.25% on Phone. In contrast, Kollect retrieves from the same bounded executable label space and uses the additional budget through quota allocation and coverage selection. Empirically, this leads to stable quality improvements across the K values.

- **Latency scaling.** Local generation-based methods such as DroidCall and Guided-K require longer outputs as K increases, and their latency at $K=8$ is roughly twice that at $K=3$. Cloud LLM baselines show smaller relative increases because network overhead dominates, but their absolute latency remains high. Kollect is less sensitive to K : the command is encoded once, and only the retrieval and selection budget changes. Thus, its latency remains close to the $K=3$ value across all evaluated budgets. Overall, Kollect uses larger displays to improve collection quality while preserving near-instant proposal latency.

7.4 Effect of Collected Supervision Size

Fig. 10 examines how the amount of collected supervision affects downstream personalization. Since $K=1$, each method adds its sole proposal for each command to its downstream personalization dataset. For each method, we sample $n \in \{50, 100, 200, 400\}$ pairs from this dataset, augmented with diverse paraphrases for each profile, and use them to adapt the same downstream task model. We consider fine-tuning, RAG, and agent memory to test whether the collected supervision is useful across different adaptation strategies. We report the downstream accuracy gain over the unadapted task model, whose accuracy is 73.1% on Home and 71.7% on Phone. This protocol isolates supervision quality at a fixed data volume, while Tab. 2 reports per-example collection latency.

The key results and takeaways are as follows.

- **More supervision improves personalization.** All candidate-proposal methods improve the unadapted model, and the gains increase with the number of collected pairs. Under fine-tuning, the gain grows from roughly 6 points at $n=50$ to about 13 points at $n=400$. RAG and agent memory show the same overall trend, although their gains increase more gradually. Thus, the supervision collected through the annotation loop benefits downstream personalization across all three adaptation strategies.

Table 3. Top-1 proposal quality on synthesized and human-authored commands. API accuracy and Slot F1 evaluate the proposed label under $K=1$. Average latency is measured on Jetson Orin Nano.

Scenario	Test Set	API Acc. (%)	Slot F1 (%)	Avg. Lat. (ms)
Home	Synthetic	92.1	89.4	653
Home	HWU64 (mapped)	92.4	91.6	679
Phone	Synthetic	91.9	89.0	668
Phone	SLURP_real (mapped)	93.7	91.3	691

Table 4. Exact-label Hit@4 and candidate-set outcome frequencies on the fully specified command subsets selected from the mapped HWU64 and SLURP_real corpora, under $K=4$.

Scenario	Method	Exact Hit@4 (%)	Same-API correction (%)	Full fallback (%)
Home	Guided-K	78.1	18.8	3.1
	Qwen3-4B	83.4	14.8	1.8
	GPT-5.2	86.9	11.8	1.3
	Sasha	85.6	12.2	2.2
	Kollect	84.5	13.8	1.7
Phone	Guided-K	76.9	19.5	3.6
	Qwen3-4B	82.6	15.2	2.2
	GPT-5.2	85.5	12.8	1.7
	Sasha	84.8	12.5	2.7
	Kollect	83.5	14.5	2.0

- **Utility across adaptation strategies.** The collected data are useful under different data regimes. At $n=50$ on Home, supervision collected by Kollect yields 80.7% accuracy with RAG and 80.4% with agent memory, compared with 79.4% with fine-tuning. At $n=400$, fine-tuning reaches 85.8%, compared with 84.6% for RAG and 84.4% for agent memory. These results indicate that retrieval- and memory-based adaptation can use small collections effectively, while fine-tuning benefits more from larger collections.
- **Supervision quality and collection latency.** At the same n , higher-accuracy LLM-based proposal methods retain a small downstream advantage. Across the three adaptation strategies, Kollect trails GPT-5.2 by approximately 1.4-1.6 points at $n=50$, but by only 0.3-0.7 points at $n=400$. The narrowing gap indicates that additional supervision can partly compensate for small differences in per-example collection quality. The fixed- n comparison isolates the quality of the collected supervision. In practice, collection is also constrained by interaction time. Kollect’s sub-second proposal latency can support more confirmations within a fixed time budget than slower generation-based methods.

7.5 Evaluation on Human-Authored Commands

The previous experiments use LLM-synthesized commands to support controlled, preference-conditioned evaluation. We complement them with human-authored commands from public corpora. We first evaluate top-1 proposal quality on all compatible mapped commands and then report $K=4$ interaction outcomes on fully specified subsets, for which the complete intended label is recoverable from the command itself.

For Home, we map compatible device-control commands from HWU64 [38] to our executable action space. For Phone, we use compatible commands from SLURP_real [3], whose intent and slot ontology underlies our Phone action space. We retain only commands that map uniquely to an executable action and construct the reference label y^* by mapping the annotated intent or action to an API and its entities to slot values. The encoder, command library, grouped retriever, and all other evaluation settings remain unchanged.

Note that these mapped subsets evaluate written commands that align unambiguously with our executable action spaces. Long-term preference changes, spontaneous speech, and ASR errors lie outside their scope.

The key results and takeaways are as follows.

Table 5. Sensitivity to continuous-slot granularity on 1,000 synthetic, fully specified Home commands. The library is rebuilt at each interval width.

Method	Interval	C	FP16 Index (MB)	Sim. Matrix (MB)	Avg. Latency (ms)	API Acc. (%)	Slot F1 (%)
GPT-5.2	—	—	—	—	12,281	94.8	92.9
Kollect	1	1,000	0.51	0.07	656	91.8	89.6
Kollect	0.1	10,000	5.12	6.75	707	92.1	91.5

- **Top-1 proposal quality.** As shown in Tab. 3, Kollect achieves 92.4% API accuracy and 91.6% Slot F1 on mapped HWU64 commands, and 93.7% API accuracy and 91.3% Slot F1 on mapped SLURP_real commands. Proposal latency remains below one second in both scenarios. These results match or exceed those on the corresponding synthetic splits, showing that Kollect transfers to compatible human-authored expressions.
- **Exact coverage on fully specified commands.** We additionally select commands that explicitly specify the intended API and all required fields and evaluate the five candidate-set methods under the $K=4$ interface. This setting isolates exact candidate coverage because the complete executable label can be recovered from the command without inferring omitted preferences. As shown in Tab. 4, Kollect achieves exact-label Hit@4 of 84.5% on Home and 83.5% on Phone. It outperforms Guided-K and Qwen3-4B in both scenarios and remains within 1.1-2.4 percentage points of Sasha and GPT-5.2. Only 1.7% and 2.0% of its proposals require full fallback, respectively.

7.6 Scaling Sensitivity

We examine two ways in which a deployment may outgrow the default configuration: (i) finer discretization of continuous slots and (ii) expansion of the API-group and command-library sizes. Both analyses keep the encoder and display budget $K=1$ fixed and report average proposal latency on Jetson Orin Nano.

The continuous-slot analysis uses a separate set of 1,000 synthetic, fully specified commands constructed by filling the Home action-space templates with devices, discrete arguments, and temperature, brightness, or volume values respecting API-defined ranges and precision, paired with executable labels preserving the exact values. We rebuild the library using interval widths $\Delta_s \in \{1, 0.1\}$ native units and evaluate both configurations on the same commands. The resulting libraries contain 1,000 and 10,000 entries, respectively. GPT-5.2 provides a generation-based reference. The action-space analysis starts from the default Home configuration and independently expands $|G|$, $|C|$, or both by factors of 5 and 10. Its $1 \times / 1 \times$ setting reproduces the results in Tab. 2.

The key results and takeaways are as follows.

- **Continuous-slot granularity.** As shown in Tab. 5, reducing the interval from 1 to 0.1 increases $|C|$ by $10\times$ and the FP16 reference index from 0.51 to 5.12 MB. The in-group similarity matrices increase from 0.07 to 6.75 MB, consistent with their $O\left(\sum_{g \in G} |C_g|^2\right)$ storage cost. Despite this growth, proposal latency increases by only 51 ms, from 656 to 707 ms. API accuracy remains nearly unchanged, while Slot F1 improves by 1.9 points because the finer library represents continuous values more precisely. GPT-5.2 remains 2.7 points higher in API accuracy and 1.4 points higher in Slot F1, but is $17.4\times$ slower.
- **API-group and command-library scaling.** Tab. 6 shows the effects from increasing $|G|$ and $|C|$. Scaling only $|G|$ to $5\times$ and $10\times$ reduces API accuracy from 92.1% to 91.3% and 90.8%, respectively, as more groups compete for the fixed display budget. A larger $|G|$ also delays quota-support stabilization and increases encoding latency. Scaling only $|C|$ increases API accuracy to 92.3% and 92.7% and Slot F1 to 89.6% and 89.9%, showing that broader library coverage offsets the extra search cost. When both increase, the larger library partially compensates for greater group ambiguity. The joint $5\times$ and $10\times$ configurations achieve 91.7%/89.0% and 91.4%/88.6% API accuracy/Slot F1. Average latency remains below 750 ms throughout.

Table 6. Sensitivity to API-group and command-library growth under $K=1$. The $1\times/1\times$ row is the default Home configuration.

G scale	C scale	API Acc. (%)	Slot F1 (%)	Avg. Latency (ms)
1×	1×	92.1	89.4	653
5×	1×	91.3	88.6	684
1×	5×	92.3	89.6	692
5×	5×	91.7	89.0	711
10×	1×	90.8	88.1	701
1×	10×	92.7	89.9	719
10×	10×	91.4	88.6	742

Table 7. Ablations under the default top-1 setting ($K=1$). **Removed** names the Kollect component being ablated, and **Replacement** describes the substitute while keeping other components unchanged. API Acc. measures API-group correctness; Slot F1 evaluates slot-value correctness; Latency is average proposal latency on Jetson Orin Nano; Downstream Acc. reports downstream task accuracy after the default fine-tuning-based personalization with $n=400$ collected pairs.

Removed	Replacement	Home Scenario				Phone Scenario			
		API Acc. (%)	Slot F1 (%)	Latency (ms)	Downstream Acc. (%)	API Acc. (%)	Slot F1 (%)	Latency (ms)	Downstream Acc. (%)
Adaptive API-group quota allocation (§4.1)	Global top- K retrieval	89.1	84.3	648	82.2	88.8	84.1	664	80.8
	Hard group top- K retrieval	86.6	81.7	644	79.6	86.4	81.5	661	78.1
Intent-coverage entry selection (§4.2)	Nearest-neighbor selection	89.6	82.7	638	80.5	89.0	83.4	658	81.3
	Query-agnostic diversity selection	89.8	85.0	682	82.7	89.3	84.7	704	81.1
Group-stability exit policy (§5.4)	Top-entry stability policy	92.4	88.5	1284	85.6	92.2	88.3	1373	83.6
Slot-pattern completion (§6.2)	No completion	88.2	83.1	651	81.3	88.4	82.9	666	79.9
Kollect		92.1	89.4	653	85.4	91.9	89.0	668	83.3

The results above show that Kollect remains effective over the evaluated ranges of continuous-slot precision, API-group count, and library size. However, they do not show performance at enterprise or campus scale, which may require approximate indexing, API-group sharding, or hierarchical API organization to control retrieval and maintenance costs.

7.7 Ablation and Mechanism Analysis

7.7.1 Module Ablations. Tab. 7 isolates four design choices in Kollect under the default top-1 setting ($K=1$). For each ablation, we replace one component while keeping the command library, encoder, retriever, and downstream evaluation protocol otherwise unchanged. Adaptive API-group quota allocation is replaced by either global top- K retrieval or retrieval restricted to the highest-scoring group. Intent-coverage selection is replaced by nearest-neighbor selection or query-agnostic diversity selection. The group-stability exit policy is replaced by top-entry stability, and slot-pattern completion is removed by retaining only observed seed entries.

The key results and takeaways are as follows.

- **Adaptive API-group quota allocation.** Replacing adaptive allocation with global retrieval reduces API accuracy from 92.1% to 89.1% and downstream accuracy from 85.4% to 82.2% on Home. Hard group retrieval performs worse, reducing them to 86.6% and 79.6%, respectively. The same trend holds on Phone. These results show that retrieval should neither allow dense groups to dominate globally nor commit prematurely to a single group.
- **Intent-coverage entry selection.** Nearest-neighbor selection lowers Slot F1 from 89.4% to 82.7% on Home and from 89.0% to 83.4% on Phone. Query-agnostic diversity recovers part of this loss but still reduces downstream accuracy from 85.4% to 82.7% on Home and from 83.3% to 81.1% on Phone. Thus, diversity alone is insufficient. The selected entries must also cover regions relevant to the current command.
- **Group-stability exit policy.** Top-entry stability slightly increases API accuracy but nearly doubles latency, from 653 to 1,284 ms on Home and from 668 to 1,373 ms on Phone. It also produces lower Slot F1 in both

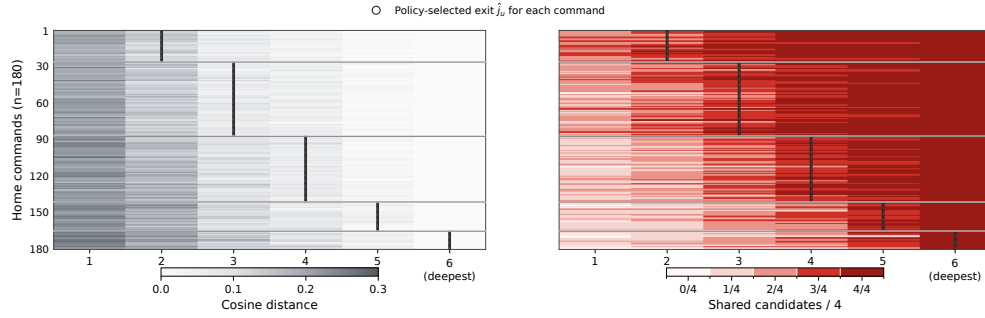


Fig. 11. Per-command comparison of each exit with the deepest exit on the Home collection split. (Left) Cosine distance d_u^j between the query embedding at each exit and the deepest-exit embedding. (Right) Candidate overlap@4 O_u^j between the size-4 candidate set at each exit and the deepest-exit candidate set. Rows are grouped by the policy-selected exit $\hat{j}_u$, and hollow circles mark that exit for each command.

scenarios. Because the nearest entry can change under small representation differences, top-entry stability rarely triggers at a shallow exit. Aligning the stopping rule with quota support instead retains comparable proposal quality at substantially lower latency.

- **Slot-pattern completion.** Removing completion lowers Slot F1 from 89.4% to 83.1% on Home and from 89.0% to 82.9% on Phone. Downstream accuracy decreases by 4.1 and 3.4 points, respectively. The observed seed entries therefore do not cover enough valid parameter combinations, and schema-guided completion improves coverage of executable labels absent from the seed data.

7.7.2 Early-Exit Fidelity. The exit-policy ablation above measures its *aggregate* effect on proposal quality and latency. Fig. 11 further examines whether the candidate set returned at the policy-selected exit remains consistent with that of the deepest exit.

For each of the 180 commands in $D_{\text{col}}^{\text{Home}}$, we record the selected exit $\hat{j}_u$, continue execution to the deepest exit $J=6$, and retrieve a size-4 candidate set $Y_4^j(x_u)$ at every exit. We measure embedding distance as $d_u^j = 1 - \cos(v_u^j, v_u^J)$ and candidate overlap as $O_u^j = |Y_4^j(x_u) \cap Y_4^J(x_u)|/4$.

The key results and takeaways are as follows.

- **Candidate-set agreement.** Of the 180 commands, 165 exit before the deepest head. For these commands, mean candidate overlap increases from 48.0% at exits before $\hat{j}_u$ to 88.4% from $\hat{j}_u$ through the last shallow exit. Moreover, 145 commands (87.9%) retain at least three of the four deepest-exit candidates at the selected exit and every subsequent shallow exit.
- **Residual embedding differences.** Candidate agreement remains high even though the selected-exit embeddings have not converged to the deepest-exit representation. Thus, the policy does not reproduce the deepest exit exactly, but it usually preserves the candidate output required by the grouped retriever. This supports treating early-exit retrieval as a practical approximation of deepest-exit retrieval.

7.8 System Overhead

We evaluate Kollekt’s overhead along two execution paths. The *candidate-proposal path* encodes an incoming command, searches the command library, and displays the candidate set (Sec. 7.8.1). The *library-maintenance path* updates the library and its auxiliary retrieval structures after a label is confirmed (Sec. 7.8.2).

7.8.1 Candidate-Proposal Overhead. We compare Kollekt with five high-quality baselines from Tab. 2: DroidCall, Guided-K, Gemini-3-Flash, GPT-5.2, and Sasha. All methods use the top-1 setting ($K=1$) and are evaluated on

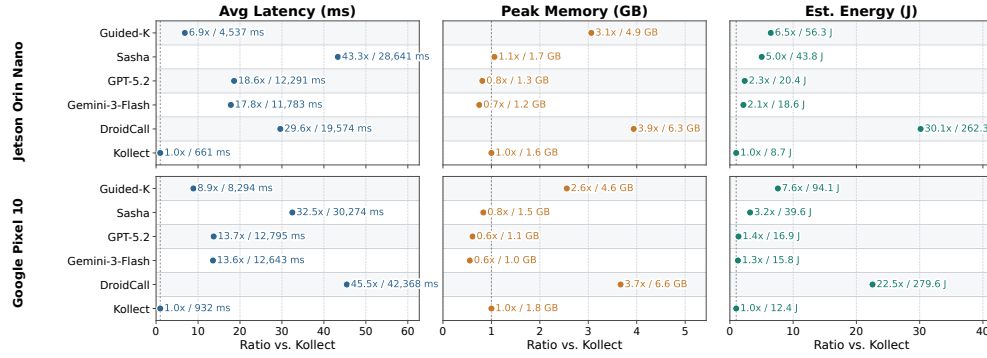


Fig. 12. End-to-end deployment overhead on Jetson Orin Nano and Google Pixel 10. Rows correspond to devices; columns report average proposal latency, peak local memory, and estimated local energy on linear axes. Each marker is annotated with its ratio relative to Collect and its absolute value.

Jetson Orin Nano and Google Pixel 10. Latency is measured from receiving a command to returning the proposed executable label. For local methods, peak memory and energy include on-device model execution. For cloud-API methods, latency includes the network round trip and remote response time, whereas memory and energy include only client-side overhead. Energy is estimated by integrating tegrastats power traces on Jetson and using per-UID batterystats measurements on Pixel, with counters reset before each run. Fig. 12 summarizes proposal latency, peak local memory, and estimated local energy on both platforms.

The key results and takeaways are as follows.

- **Latency.** Collect has the lowest latency on both platforms, requiring 661 ms on Jetson and 932 ms on Pixel. Guided-K, the next-fastest baseline, is 6.9× slower on Jetson and 8.9× slower on Pixel. The largest gaps are 43.3× for Sasha on Jetson and 45.5× for DroidCall on Pixel. The cloud-API methods require 11.8–30.3 s, placing them beyond the 8 s patience-lost threshold.
- **Memory.** Collect uses 1.6 GB of peak memory on Jetson and 1.8 GB on Pixel. On Jetson, this is 3.1× less than Guided-K and 3.9× less than DroidCall because Collect avoids an autoregressive decoder on the online path. Cloud-API methods need less local memory but shift model execution to the cloud.
- **Energy.** Among on-device methods, Collect consumes the least local energy, *i.e.*, 8.7 J on Jetson and 12.4 J on Pixel. Guided-K consumes 6.5× and 7.6× more energy on the two devices, while DroidCall consumes 30.1× and 22.5× more. These results reflect Collect’s single early-exit encoding pass followed by lightweight retrieval, rather than autoregressive generation.

7.8.2 Library-Maintenance Overhead. We measure the latency of maintaining one canonical library entry at three command-library sizes in the Home scenario. An Add operation inserts a new canonical entry, a Modify operation replaces an entry’s canonical surface form, and a Delete operation removes an entry. Both Add and Modify require generating a deepest-exit reference embedding, whereas Delete requires only updates to the command library and its auxiliary retrieval structures.

We compare two approaches to reference-embedding generation. *Full re-encoding* processes the confirmed command from the first backbone layer through the deepest exit J . *Resumed re-encoding* reuses computation from candidate proposal when the new canonical surface form is the just-confirmed command. If proposal stops at exit j , Collect temporarily retains the hidden state $H_u^j = E_j(x_u)$ and, after confirmation, executes only layers $j + 1, \dots, J$ to obtain the deepest-exit reference embedding. The retained state is released after maintenance. When no retained state is available, Collect uses full re-encoding.

Table 8. Single-entry command-library maintenance latency at different $|C|$ in the Home scenario. Add inserts a new canonical entry, Modify replaces its canonical surface form and regenerates its reference embedding, and Delete removes one indexed entry. All reported latencies are measured directly. "Encoding" is the cost of generating the reference embedding; "Total" additionally includes the update to D , when applicable, and to the metadata in C , the FAISS index, and the affected in-group similarity matrix.

Operation	$ C $	Full re-encoding latency (ms)		Resumed re-encoding latency (ms)	
		Encoding	Total	Encoding	Total
Add one entry	10^3	1,075.4	1,075.9	609.6	610.3
	10^4	1,076.8	1,077.9	638.7	639.6
	10^5	1,074.9	1,083.6	621.8	631.2
Modify canonical form	10^3	1,077.2	1,078.1	617.9	619.1
	10^4	1,075.8	1,079.3	612.7	615.8
	10^5	1,078.1	1,108.4	649.3	680.3
Delete one entry	10^3	–	0.6	–	0.6
	10^4	–	3.7	–	3.7
	10^5	–	37.1	–	37.1

We evaluate both modes under *eager maintenance*, in which each library change is applied immediately after confirmation. Before each timed operation, the command library, FAISS index, and in-group similarity matrices have already been constructed, so the reported latency measures only the incremental update. This is a conservative setting because the update policy in Sec. 6.3 can accumulate changes for periodic batch maintenance. Tab. 8 reports the measured encoding and total latency for Add, canonical-surface-form Modify, and Delete at the three command-library sizes.

The key results and takeaways are as follows.

- **Maintenance latency.** At $|C| = 10^5$, resumed re-encoding reduces the total Add latency from 1,083.6 to 631.2 ms, a 41.7% reduction, and the total Modify latency from 1,108.4 to 680.3 ms, a 38.6% reduction. Delete requires no reference encoding and completes in 37.1 ms. Under resumed re-encoding, the non-encoding updates to the command library, FAISS index, and in-group similarity matrices add only 9.4 ms for Add and 31.0 ms for Modify. Thus, deepest-exit reference-embedding generation, rather than auxiliary-structure maintenance, is the dominant cost of adding or modifying an entry.
- **Impact on interaction.** In the user-study logs, consecutive commands were submitted 1.8–14.3 s apart, exceeding the largest measured eager-maintenance latency of 680.3 ms. Thus, every measured eager update would have completed before the next command was submitted. Periodic batch maintenance provides additional timing slack by moving these updates outside the immediate post-confirmation path.

7.9 User Study

We conducted a two-phase user study to evaluate (i) perceived interaction quality during candidate collection and (ii) how method-specific selected labels affect downstream personalization. As shown in Fig. 13, the study compares three candidate-proposal methods on a Google Pixel phone: *Fixed-head* (JointBERT), *Generation* (Qwen3-1.7B), and *Kollect*. Unlike the main evaluation in Sec. 7.2, which uses the stringent top-1 setting, the user study uses a display budget of $K=4$ to match the intended multiple-choice confirmation interface.

We recruited 20 participants (11 male and 9 female, age 19–42, $\mu = 26.6$, $\sigma = 6.1$), all of whom were regular smartphone users with prior experience using mobile assistants. The study followed a within-subjects design, with condition order counterbalanced using a Latin square. Each participant issued 50 collection commands evaluated by all three methods, yielding 1,000 method-trials per method. The study had two phases.

- (1) In the *collection and adaptation phase*, participants issued natural-language commands, interacted with the displayed candidates, and rated the interaction experience. The method-specific $(x_u, y_{\text{conf},m})$ pairs were

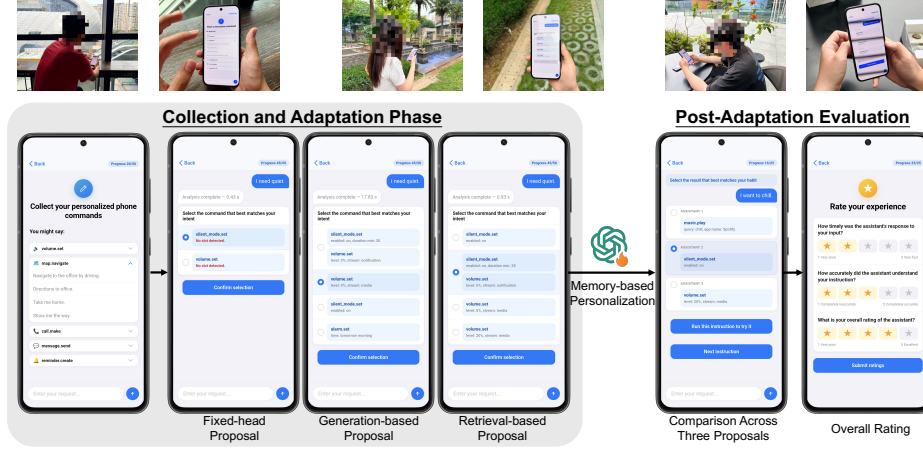


Fig. 13. Overview of the two-phase user study. In the collection and adaptation phase, each participant submits smartphone commands and, for method m , separately selects a method-specific label $y_{\text{conf},m}$ directly from its $K=4$ candidates for personalization. Direct confirmation, same-API correction, or full fallback then produces the final label y^* stored in D . The method-specific $(x_u, y_{\text{conf},m})$ pairs are used to personalize the corresponding assistants with agent memory. In the post-adaptation phase, participants compare the outputs of the three personalized assistants and provide preference judgments.

used to personalize the corresponding assistant with agent-memory personalization, while the final (x_u, y^*) pairs were recorded in D .

(2) In the *post-adaptation phase*, participants compared the outputs of the three personalized assistants.

The study was approved by our Institutional Review Board.

7.9.1 Collection and Adaptation Phase. For each command, the three methods independently proposed $K=4$ executable-label candidates through the same confirmation interface. To isolate how the quality of each method's displayed candidates affects downstream personalization, the participant selected the candidate from method m that best matched the intended action without using correction or fallback. We denote this method-specific selected label by $y_{\text{conf},m}$. The resulting $(x_u, y_{\text{conf},m})$ pairs formed three parallel memories, each of which personalized the corresponding assistant using the agent-memory configuration in Sec. 7.1.

Separately, the confirmation interaction produced the final intended label y^* through direct confirmation, same-API correction, or full fallback. For a given command, y^* represents the same intended action across methods and is recorded as (x_u, y^*) in D . Thus, $y_{\text{conf},m}$ captures the quality of the candidates displayed by method m , whereas y^* is the final executable supervision obtained after any required correction. After each trial, participants rated *response speed*, *candidate quality*, and *overall interaction quality* on 5-point Likert scales.

Tab. 9 reports exact-label candidate coverage, fallback frequency, and post-display completion latency, computed from the timestamped trial-level logs. Each method-trial is assigned to one of the three interaction outcomes defined in Sec. 2.2, relative to the same final intended label y^* . Completion latency is measured from the time the four candidates are fully displayed until y^* is confirmed. The first three rating columns of Tab. 10 report the corresponding subjective interaction results.

The key results and takeaways are as follows.

- **Candidate coverage and fallback burden.** Collect achieves an exact-label Hit@4 of 69.5%, within 0.9 percentage points of Generation at 70.4%, and substantially above Fixed-head at 35.1%. A further 27.4% of Collect trials contain a candidate with the intended API and require only field correction, while 3.1% require full fallback. Thus, 96.9% of Collect trials either contain the complete intended label or preserve its

Table 9. Exact-label Hit@4 and fallback outcomes in the user study ($N=20$, 1,000 method-trials per method, $K=4$). Latency is the mean post-display completion time within each outcome.

Condition	Exact Hit@4		Same-API correction		Full fallback	
	Freq. (%)	Latency (s)	Freq. (%)	Latency (s)	Freq. (%)	Latency (s)
Fixed-head	35.1	1.8	48.2	4.7	16.7	13.8
Generation	70.4	1.4	25.7	3.7	3.9	14.9
Kollect	69.5	1.6	27.4	4.1	3.1	13.4

Table 10. User study results ($N = 20$). Speed, Quality, and Overall are mean 5-point Likert ratings from the collection phase. Pref. reports participant-level first-choice votes in the post-adaptation evaluation.

Condition	Speed	Quality	Overall	Pref.
Fixed-head	4.8	2.8	3.1	0/20
Generation	1.9	4.5	3.9	12/20
Kollect	4.3	4.2	4.5	8/20

Table 11. Post-adaptation selected-prediction counts by command type. All 20 participants evaluated every command, yielding 1,000 participant-command trials. Counts are trial-level selected predictions, whereas the Pref. column of Tab. 10 reports participant-level first-choice votes.

Command type	Commands	Trials	Fixed-head	Generation	Kollect
Fully specified	4	80	3	40	37
Preference under-spec.	8	160	5	74	81
Object under-spec.	10	200	7	99	94
Parameter under-spec.	13	260	6	136	118
Goal-oriented	15	300	10	174	116
Total	50	1,000	31	523	446

API. The mean post-display completion times are 1.6 s, 4.1 s, and 13.4 s for direct confirmation, same-API correction, and full fallback, respectively.

- **Perceived interaction quality.** Kollect receives the highest overall interaction rating, with an average score of 4.5. Fixed-head receives the highest speed rating (4.8) but substantially lower candidate quality (2.8), resulting in an overall rating of 3.1. Generation receives the highest candidate-quality rating (4.5), but its slower response is reflected in a speed rating of 1.9. Kollect remains close to Fixed-head in perceived speed (4.3) and close to Generation in candidate quality (4.2). This balance between speed and candidate quality leads to the highest overall rating during collection.

7.9.2 Post-Adaptation Evaluation. In the post-adaptation phase, the three personalized assistants produced top-1 predictions using their respective memories constructed from $(x_u, y_{\text{conf},m})$. Each participant evaluated all 50 new commands and selected the prediction that best matched the intended action, yielding 1,000 participant-command trials. The Pref. column of Tab. 10 reports participant-level first-choice votes, whereas Tab. 11 reports trial-level selected-prediction counts by command type.

We group the 50 commands into *Fully specified*, *Preference under-specified*, *Object under-specified*, *Parameter under-specified*, and *Goal-oriented* commands. This breakdown shows whether the post-adaptation preference gap is distributed uniformly or concentrated in particular command types.

The key results and takeaways are as follows.

- **Participant-level preference.** Generation is the first choice of 12 participants, Kollect of the remaining 8, and Fixed-head of none. This result reveals a trade-off between the collection interaction and the method-specific labels selected from the displayed candidates. Generation produces somewhat stronger

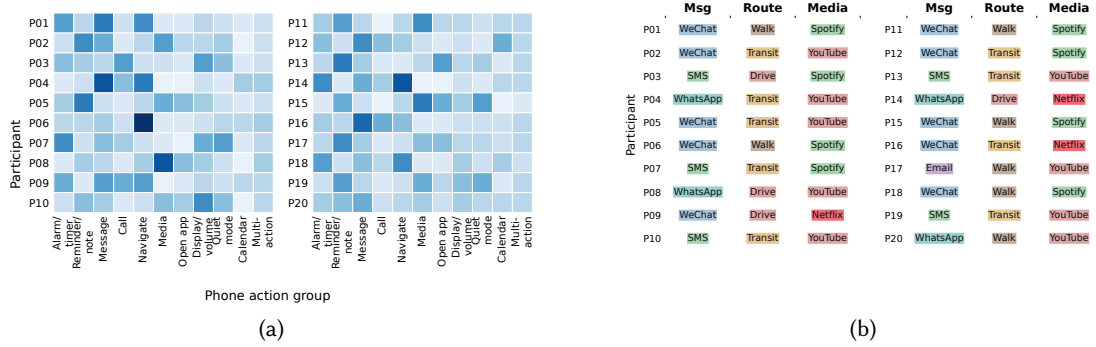


Fig. 14. Personalization profiles of the 20 study participants. (a) Per-participant command frequency across phone action groups, showing heterogeneous action usage. (b) Preferred slot values for messaging client (Msg), navigation mode (Route), and media app (Media), showing within-category divergence in concrete slot preferences.

personalized outputs but receives a perceived-speed rating of only 1.9 during collection, whereas Collect provides competitive post-adaptation utility while receiving the highest overall interaction rating.

- **The preference gap is concentrated in goal-oriented commands.** Generation and Collect are nearly tied on Fully specified, Preference under-specified, and Object under-specified commands, with 213 and 212 selected predictions, respectively, giving Generation a net one-selection advantage across these three types. Collect trails Generation by 18 selections on Parameter under-specified commands and by 58 selections on Goal-oriented commands. The Goal-oriented difference alone accounts for 75.3% of Generation’s overall 77-selection advantage. Fixed-head is selected in 31 individual trials but is not the first choice of any participant.
- **Implication for improvement.** Goal-oriented commands describe a desired outcome without explicitly naming the action or API. Generation can use the pretrained LLM’s goal-to-action reasoning, whereas Collect may distribute retrieval relevance across several semantically related APIs. Thus, a potential extension is to store confirmed goal-oriented expressions as additional retrieval representations of their executable labels, enabling the retriever to match goal descriptions more directly.

The study is limited to 20 participants and a short evaluation period; future work will evaluate Collect in a longitudinal deployment with naturally spoken commands.

7.9.3 Personalization Profiles. The action-group heatmap in Fig. 14a shows that participants differ substantially in the phone actions they invoke. Some concentrate on a few categories such as messaging or navigation, while others distribute commands more evenly, suggesting a single global action prior is insufficient. The slot-preference panel in Fig. 14b further shows that personalization is needed within the same action group. For example, participants prefer different messaging clients (WeChat, WhatsApp, SMS, Email), navigation modes (walking, transit, driving), and media apps (Spotify, YouTube, Netflix). These profiles support the paper’s motivation that executable supervision must capture not only the intended API group but also user-specific slot values.

8 Related Work

8.1 Personalized Assistants with LLMs

LLMs are increasingly used as personal assistants, including smart-home control and smartphone automation [31, 66, 68]. Existing work mainly improves assistants in three directions. The first improves task execution quality, such as goal-oriented reasoning [31], device coordination [66], and API-grounded action execution [48], including

sensor-driven HVAC question answering [34] and structured context memory for smart environments [36]. The second improves response speed through on-device deployment, model compression, small language models, or memorization of repeated subtasks [22, 66]. The third improves personalization by adapting assistants to user preferences, histories, and contexts [50, 51, 59, 62]. For example, Me-Agent [59] stores structured preference records for later reuse. LaMP [50] retrieves relevant user-profile entries and injects them into prompts. HomeLLM [22] and DroidCall [62] fine-tune compact models on user-specific task data.

We focus on the *pre-requisite* of personalization, *i.e.*, collecting user-specific datasets of executable supervision, which can support fine-tuning [20, 51], RAG [27, 50, 51], and agent memory [59] of the personal assistants.

8.2 Semantic Parsing

Our data collection problem can be viewed as semantic parsing, where a natural-language command is mapped to a structured executable label. Existing methods are either *non-interactive* or *interactive*.

Non-interactive semantic parsing predicts a label from the input command without user feedback. Fixed-head approaches, such as intent classification and slot filling, predict over predefined intent and slot inventories [8, 60]. They are efficient for closed-schema tasks, but are less suitable for our setting where executable labels depend on user-specific APIs, devices, and valid parameter combinations. Generation-based approaches, generate structured labels token by token [35, 46, 62]. They can express more complex labels, but increases latency during user interaction. Smaller generators can reduce latency but often weakens generation quality [58]. Retrieval-augmented semantic parsers retrieve similar exemplars as additional context for a generative parser [19, 45], which is orthogonal to Kollect’s use of retrieval for executable-label candidate proposal.

Interactive semantic parsing incorporates user feedback. Binary confirmation methods ask users to accept or reject one candidate label at a time [24, 65]. They can obtain precise feedback, but may require multiple rounds when early candidates are wrong. Natural-language correction methods ask users to revise or clarify incorrect outputs [12, 31]. They provide richer feedback, but require additional language understanding or generation steps. Multiple-choice confirmation asks users to select the intended label from a candidate set [4]. This mode balances user burden and feedback strength, but it relies on whether candidate proposal is accurate and fast.

Kollect adopts multiple-choice confirmation, but differs in how candidates are proposed. Instead of generating candidates during interaction, Kollect retrieves them from the user’s bounded executable label space. This shifts the main challenge from parsing or repair to retrieval-based candidate proposal, *i.e.*, constructing a small set that covers the intended label while keeping proposal latency low.

8.3 Vector Retrieval

When valid labels are bounded and enumerable, candidate proposal can be formulated as selecting a size- K subset from a fixed library rather than decoding labels token by token. Retrieval methods differ in their *selection objective* and *search scope*.

By objective, standard dense retrieval methods perform similarity-based top- K search [25, 27, 29]. Each item is scored independently, and the highest-scoring items are returned. This objective is efficient, but it does not control how the limited display slots are distributed across the executable label space. In our case, many labels under the same API may differ only in device or parameter values. Hence, flat top- K retrieval can over-select redundant variants from one dense API group. Diversity-based methods reduce redundancy by encouraging selected items to be far apart [7, 13, 33]. However, geometric diversity is not query-conditioned intent coverage. It may spread candidates across the space without prioritizing regions that are plausible for the command.

By search scope, retrieval can score all items globally or restrict search to a predicted attribute group. Global search lets all labels compete for the same K slots and is sensitive to group imbalance [2, 7, 52, 67]. Attribute-restricted retrieval first predicts or extracts an attribute, then searches only within the corresponding subset [61,

65]. This avoids some imbalance, but assumes that the target attribute is identifiable before confirmation. That assumption fails for under-specified commands, where several API groups may be plausible simultaneously.

Kollect addresses both limitations with a query-adaptive grouped retriever. It estimates group relevance from the query embedding, allocates display slots across API groups, and fills each non-zero quota using a query-conditioned coverage objective. Thus, Kollect differs from flat similarity search, which lacks group-level budget control, and from generic diversity search, which does not directly model query-relevant intent coverage.

8.4 On-Device Inference

Retrieval-based candidate proposal still requires encoding the user command before any candidates can be displayed. Because this encoding lies on the confirmation path, it must meet strict per-round latency constraints on personal devices. Popular acceleration methods include *static compression* and *adaptive computation*.

Static compression reduces model cost before deployment. Pruning removes weights, attention heads, or structured blocks [14, 39]. Quantization lowers weight or activation precision [10, 15]. These techniques are largely orthogonal to Kollect. They can reduce the cost of the underlying encoder, but they do not decide how much computation an individual command needs at runtime.

Adaptive computation varies compute per input. Conditional computation and mixture-of-experts methods route inputs through selected modules [54]. Early-exit networks attach intermediate exits and stop when the current output is sufficiently reliable, allowing easy inputs to terminate before the deepest layer [37, 57, 63, 70]. Most early-exit criteria are designed for classification, using confidence, entropy, or top-1 label stability.

Kollect also uses early exiting, but the output of each exit is not a class label. It is a query embedding consumed by the *grouped retriever*. Hence, classification-oriented stopping misaligns with the downstream decision. Kollect introduces a retriever-aligned early-exit encoder whose shallow exits are trained to preserve group-level retrieval behavior, and whose runtime policy stops only when the quota support is stable across consecutive exits.

9 Conclusion

This paper studied executable-supervision collection for personal assistants, where natural-language commands must be paired with user-approved executable labels for downstream personalization. We identified candidate proposal as a key bottleneck in multiple-choice confirmation and presented Kollect, a retrieval-based framework that avoids token-by-token candidate generation. By combining query-adaptive grouped retrieval with a retriever-aligned early-exit encoder, Kollect improves candidate coverage under API ambiguity while keeping proposal latency low. Experiments on smart-home and smartphone assistant scenarios show that Kollect achieves near-instant candidate proposal with competitive annotation quality, low on-device overhead, and collected supervision that benefits fine-tuning, retrieval-augmented generation, and agent memory. In future work, we plan to update the command library and encoder continuously as users confirm new commands, enabling longer-term adaptation without frequent full retraining. We also plan to incorporate richer contextual signals, such as location, time, device state, and interaction history, into retrieval to further disambiguate under-specified commands.

Acknowledgments

This research is supported by the RGC of Hong Kong SAR, China (Project No. CityU 11206425) and CityU APRC grant (No. 9610633).

References

- [1] Tahir Abbas, Ujwal Gadiraju, Vassilis-Javed Khan, and Panos Markopoulos. 2022. Understanding user perceptions of response delays in crowd-powered conversational systems. *Proceedings of the ACM on Human-Computer Interaction* 6, CSCW2 (2022), 1–42.
- [2] Rakesh Agrawal, Sreenivas Gollapudi, Alan Halverson, and Samuel Jeong. 2009. Diversifying search results. In *Proceedings of the second ACM international conference on web search and data mining*. 5–14.

- [3] Emanuele Bastianelli, Andrea Vanzo, Pawel Swietojanski, and Verena Rieser. 2020. SLURP: A spoken language understanding resource package. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 7252–7262.
- [4] Anton Belyy, Chieh-Yang Huang, Jacob Andreas, Emmanouil Antonios Platanios, Sam Thomson, Richard Shin, Subhro Roy, Aleksandr Nisnevich, Charles Chen Jr, and Benjamin Van Durme. 2022. Guided K-best Selection for Semantic Parsing Annotation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics: System Demonstrations*. 114–126.
- [5] Frank Bentley, Chris Luvogt, Max Silverman, Rushani Wirasinghe, Brooke White, and Danielle Lottridge. 2018. Understanding the long-term use of smart speaker assistants. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 2, 3 (2018), 1–24.
- [6] Allan Borodin, Aadhar Jain, Hyun Chul Lee, and Yuli Ye. 2017. Max-sum diversification, monotone submodular functions, and dynamic updates. *ACM Transactions on Algorithms (TALG)* 13, 3 (2017), 1–25.
- [7] Jaime Carbonell and Jade Goldstein. 1998. The use of MMR, diversity-based reranking for reordering documents and producing summaries. In *Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval*. 335–336.
- [8] Qian Chen, Zhu Zhuo, and Wen Wang. 2019. Bert for joint intent classification and slot filling. *arXiv preprint arXiv:1902.10909* (2019).
- [9] Meghan Clark, Mark W Newman, and Prabal Dutta. 2017. Devices and data and agents, oh my: How smart home abstractions prime end-user mental models. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 1, 3 (2017), 1–26.
- [10] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. 2022. Gpt3. int8 (): 8-bit matrix multiplication for transformers at scale. *Advances in neural information processing systems* 35 (2022), 30318–30332.
- [11] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. 2025. The faiss library. *IEEE Transactions on Big Data* (2025).
- [12] Ahmed Elgohary, Christopher Meek, Matthew Richardson, Adam Fourney, Gonzalo Ramos, and Ahmed Hassan. 2021. NL-EDIT: Correcting semantic parse errors through natural language interaction. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. 5599–5610.
- [13] Erhan Erkut. 1990. The discrete p-dispersion problem. *European Journal of Operational Research* 46, 1 (1990), 48–60.
- [14] Elias Frantar and Dan Alistarh. 2023. Sparsegpt: Massive language models can be accurately pruned in one-shot. In *International conference on machine learning*. PMLR, 10323–10337.
- [15] Elias Frantar, Saleh Ashkboos, Torsten Hoefer, and Dan Alistarh. 2022. Gptq: Accurate post-training quantization for generative pre-trained transformers. *arXiv preprint arXiv:2210.17323* (2022).
- [16] Dennis F Galletta, Scott McCoy, Peter Polak, and Raymond Henry. 2004. Web site delays: How tolerant are users? Association for Information Systems.
- [17] Aristides Gionis, Piotr Indyk, Rajeev Motwani, et al. 1999. Similarity search in high dimensions via hashing. In *Vldb*, Vol. 99. 518–529.
- [18] Google DeepMind. 2025. Gemini 3 Flash Model Card. <https://storage.googleapis.com/deepmind-media/Model-Cards/Gemini-3-Flash-Model-Card.pdf>
- [19] Vivek Gupta, Akshat Shrivastava, Adithya Sagar, Armen Aghajanyan, and Denis Savenkov. 2022. Retronlu: Retrieval augmented task-oriented semantic parsing. In *Proceedings of the 4th Workshop on NLP for Conversational AI*. 184–196.
- [20] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Liang Wang, Weizhu Chen, et al. 2022. Lora: Low-rank adaptation of large language models. *Iclr* 1, 2 (2022), 3.
- [21] Qiang Huang, Yanhao Wang, Yiqun Sun, and Anthony KH Tung. 2024. Diversity-Aware k -Maximum Inner Product Search Revisited. *arXiv preprint arXiv:2402.13858* (2024).
- [22] Xinyu Huang, Leming Shen, Zijiang Ma, and Yuanqing Zheng. 2026. Towards privacy-preserving and personalized smart homes via tailored small language models. *IEEE Transactions on Mobile Computing* (2026).
- [23] Piotr Indyk and Rajeev Motwani. 1998. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*. 604–613.
- [24] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, Jayant Krishnamurthy, and Luke Zettlemoyer. 2017. Learning a neural semantic parser from user feedback. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 963–973.
- [25] Gautier Izacard, Mathilde Caron, Lucas Hosseini, Sebastian Riedel, Piotr Bojanowski, Armand Joulin, and Edouard Grave. 2022. Unsupervised dense information retrieval with contrastive learning. *Transactions on Machine Learning Research* (2022).
- [26] Herve Jegou, Matthijs Douze, and Cordelia Schmid. 2010. Product quantization for nearest neighbor search. *IEEE transactions on pattern analysis and machine intelligence* 33, 1 (2010), 117–128.
- [27] Vladimir Karpukhin, Barlas Oguz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. 2020. Dense passage retrieval for open-domain question answering. In *Proceedings of the 2020 conference on empirical methods in natural language processing (EMNLP)*. 6769–6781.
- [28] Yigitcan Kaya, Sanghyun Hong, and Tudor Dumitras. 2019. Shallow-deep networks: Understanding and mitigating network overthinking. In *International conference on machine learning*. PMLR, 3301–3310.

- [29] Omar Khattab and Matei Zaharia. 2020. Colbert: Efficient and effective passage search via contextualized late interaction over bert. In *Proceedings of the 43rd International ACM SIGIR conference on research and development in Information Retrieval*. 39–48.
- [30] Prannay Khosla, Piotr Teterwak, Chen Wang, Aaron Sarna, Yonglong Tian, Phillip Isola, Aaron Maschiot, Ce Liu, and Dilip Krishnan. 2020. Supervised contrastive learning. *Advances in neural information processing systems* 33 (2020), 18661–18673.
- [31] E. King, H. Yu, S. Lee, and C. Julien. 2024. Sasha: Creative Goal-Oriented Reasoning in Smart Homes with Large Language Models. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 8, 1 (2024), 1–38.
- [32] Andreas Krause and Daniel Golovin. 2014. Submodular function maximization. *Tractability* 3, 71–104 (2014), 3.
- [33] Alex Kulesza and Ben Taskar. 2012. Determinantal point processes for machine learning. *Foundations and Trends® in Machine Learning* 5, 2-3 (2012), 123–286.
- [34] Sungmin Lee, Joonhee Lee, Minju Kang, Seungyong Lee, Dongju Kim, Jingsi Hong, Jun Shin, Pei Zhang, and JeongGil Ko. 2026. JARVIS for HVAC: LLM-based Question-Answer Framework for Sensor-driven HVAC System Interaction. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* (2026).
- [35] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. 2020. BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In *Proceedings of the 58th annual meeting of the association for computational linguistics*. 7871–7880.
- [36] Siyuan Liu and Huangxun Chen. 2026. MemAura: Structured Context Memory for Personalized LLM Reasoning in Smart Environments. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* (2026).
- [37] Weijie Liu, Peng Zhou, Zhiruo Wang, Zhe Zhao, Haotang Deng, and Qi Ju. 2020. Fastbert: a self-distilling bert with adaptive inference time. In *Proceedings of the 58th annual meeting of the association for computational linguistics*. 6035–6044.
- [38] Xingkun Liu, Arash Eshghi, Pawel Swietojanski, and Verena Rieser. 2021. Benchmarking Natural Language Understanding Services for Building Conversational Agents. In *Increasing Naturalness and Flexibility in Spoken Dialogue Interaction*. Springer Singapore, 165–183. doi:10.1007/978-981-15-9323-9_15
- [39] Paul Michel, Omer Levy, and Graham Neubig. 2019. Are sixteen heads really better than one? *Advances in neural information processing systems* 32 (2019).
- [40] Robert B Miller. 1968. Response time in man-computer conversational transactions. In *Proceedings of the December 9-11, 1968, fall joint computer conference, part I*. 267–277.
- [41] Niklas Muennighoff, Nouamane Tazi, Loïc Magne, and Nils Reimers. 2023. Mteb: Massive text embedding benchmark. In *Proceedings of the 17th Conference of the European Chapter of the Association for Computational Linguistics*. 2014–2037.
- [42] George L Nemhauser, Laurence A Wolsey, and Marshall L Fisher. 1978. An analysis of approximations for maximizing submodular set functions—I. *Mathematical programming* 14, 1 (1978), 265–294.
- [43] Jianmo Ni, Gustavo Hernandez Abrego, Noah Constant, Ji Ma, Keith Hall, Daniel Cer, and Yinfei Yang. 2022. Sentence-t5: Scalable sentence encoders from pre-trained text-to-text models. In *Findings of the association for computational linguistics: ACL 2022*. 1864–1874.
- [44] OpenAI. 2025. Introducing GPT-5.2. <https://openai.com/index/introducing-gpt-5-2/>
- [45] Panupong Pasupat, Yuan Zhang, and Kelvin Guu. 2021. Controllable semantic parsing via retrieval augmentation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. 7683–7698.
- [46] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of machine learning research* 21, 140 (2020), 1–67.
- [47] Nils Reimers and Iryna Gurevych. 2019. Sentence-bert: Sentence embeddings using siamese bert-networks. In *Proceedings of the 2019 conference on empirical methods in natural language processing and the 9th international joint conference on natural language processing (EMNLP-IJCNLP)*. 3982–3992.
- [48] Dmitriy Rivkin, Francois Hogan, Amal Feriani, Abhisek Konar, Adam Sigal, Xue Liu, and Gregory Dudek. 2024. AIoT smart home via autonomous LLM agents. *IEEE Internet of Things Journal* 12, 3 (2024), 2458–2472.
- [49] Subendhu Rongali, Luca Soldaini, Emilio Monti, and Wael Hamza. 2020. Don’t parse, generate! a sequence to sequence architecture for task-oriented semantic parsing. In *Proceedings of the web conference 2020*. 2962–2968.
- [50] Alireza Salemi, Sheshera Mysore, Michael Bendersky, and Hamed Zamani. 2024. Lamp: When large language models meet personalization. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 7370–7392.
- [51] Alireza Salemi and Hamed Zamani. 2025. Comparing retrieval-augmentation and parameter-efficient fine-tuning for privacy-preserving personalization of large language models. In *Proceedings of the 2025 International ACM SIGIR Conference on Innovative Concepts and Theories in Information Retrieval (ICTIR)*. 286–296.
- [52] Rodrygo LT Santos, Jie Peng, Craig Macdonald, and Iadh Ounis. 2010. Explicit search result diversification through sub-queries. In *European conference on information retrieval*. Springer, 87–99.
- [53] Yingyu Shan, Zeming Liu, Silin Li, Boao Qian, Jiashu Yao, Yuhang Guo, and Haifeng Wang. 2026. PEC-Home: Interpretation of Progressively Elliptical Commands in Smart Homes. In *Findings of the Association for Computational Linguistics: ACL 2026*. 19989–20006. doi:10.18653/v1/2026.findings-acl.999

- [54] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarczyk, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. 2017. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538* (2017).
- [55] Ben Shneiderman. 1984. Response Time and Display Rate in Human Performance with Computers. *Comput. Surveys* 16, 3 (1984), 265–285. doi:10.1145/2514.2517
- [56] Yunpeng Song, Yiheng Bian, Xiaorui Wang, and Zhongmin Cai. 2024. Learning from User-driven Events to Generate Automation Sequences. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 7, 4 (2024), 1–22.
- [57] Surat Teerapittayanon, Bradley McDanel, and Hsiang-Tsung Kung. 2016. Branchynet: Fast inference via early exiting from deep neural networks. In *2016 23rd international conference on pattern recognition (ICPR)*. IEEE, 2464–2469.
- [58] Chien Van Nguyen, Xuan Shen, Ryan Aponte, Yu Xia, Samyadeep Basu, Zhengmian Hu, Jian Chen, Mihir Parmar, Sasidhar Kunapuli, Joe Barrow, et al. 2025. A Survey on Small Language Models. In *Proceedings of the 15th International Conference on Recent Advances in Natural Language Processing-Natural Language Processing in the Generative AI Era*. 807–821.
- [59] Shuoxin Wang, Chang Liu, Gowen Loo, Lifan Zheng, Kaiwen Wei, Xinyi Zeng, Jingyuan Zhang, and Yu Tian. 2026. Me-Agent: A Personalized Mobile Agent with Two-Level User Habit Learning for Enhanced Interaction. *arXiv preprint arXiv:2601.20162* (2026).
- [60] Henry Weld, Xiaoqi Huang, Siyu Long, Josiah Poon, and Soyeon Caren Han. 2022. A survey of joint intent detection and slot filling models in natural language understanding. *Comput. Surveys* 55, 8 (2022), 1–38.
- [61] Ledell Wu, Fabio Petroni, Martin Josifoski, Sebastian Riedel, and Luke Zettlemoyer. 2020. Scalable zero-shot entity linking with dense entity retrieval. In *Proceedings of the 2020 conference on empirical methods in natural language processing (EMNLP)*. 6397–6407.
- [62] Weikai Xie, Li Zhang, Shihe Wang, Rongjie Yi, and Mengwei Xu. 2024. Droidcall: A dataset for llm-powered android intent invocation. *arXiv preprint arXiv:2412.00402* (2024).
- [63] Ji Xin, Raphael Tang, Jaejun Lee, Yaoliang Yu, and Jimmy Lin. 2020. DeeBERT: Dynamic early exiting for accelerating BERT inference. In *Proceedings of the 58th annual meeting of the association for computational linguistics*. 2246–2251.
- [64] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. 2025. Qwen3 technical report. *arXiv preprint arXiv:2505.09388* (2025).
- [65] Ziyu Yao, Yu Su, Huan Sun, and Wen-tau Yih. 2019. Model-based interactive semantic parsing: A unified framework and a text-to-SQL case study. In *Proceedings of the 2019 conference on empirical methods in natural language processing and the 9th international joint conference on natural language processing (EMNLP-IJCNLP)*. 5447–5458.
- [66] Chaerin Yu, Chihun Choi, Sunjae Lee, Hyosu Kim, Steven Y Ko, Young-Bae Ko, and Sangeun Oh. 2026. Leveraging LLMs for Efficient and Personalized Smart Home Automation. *arXiv preprint arXiv:2601.04680* (2026).
- [67] Meike Zehlike, Francesco Bonchi, Carlos Castillo, Sara Hajian, Mohamed Megahed, and Ricardo Baeza-Yates. 2017. Fa* ir: A fair top-k ranking algorithm. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*. 1569–1578.
- [68] Chi Zhang, Zhao Yang, Jiaxuan Liu, Yanda Li, Yucheng Han, Xin Chen, Zebiao Huang, Bin Fu, and Gang Yu. 2025. Appagent: Multimodal agents as smartphone users. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*. 1–20.
- [69] Yanzhao Zhang, Mingxin Li, Dingkun Long, Xin Zhang, Huan Lin, Baosong Yang, Pengjun Xie, An Yang, Dayiheng Liu, Junyang Lin, et al. 2025. Qwen3 embedding: Advancing text embedding and reranking through foundation models. *arXiv preprint arXiv:2506.05176* (2025).
- [70] Wangchunshu Zhou, Canwen Xu, Tao Ge, Julian McAuley, Ke Xu, and Furu Wei. 2020. Bert loses patience: Fast and robust inference with early exit. *Advances in Neural Information Processing Systems* 33 (2020), 18330–18341.