A Human-in-the-Loop Confidence-Aware Failure Recovery Framework for Modular Robot Policies

Rohan Banerjee[‡]
Cornell University

Krishna Palempalli[*]
Cornell University

Bohan Yang[*]
Cornell University

Jiaying Fang
Cornell University

Alif Abdullah
Cornell University

Tom Silver
Princeton University

Sarah Dean[†]
Cornell University

Tapomayukh Bhattacharjee[†]
Cornell University

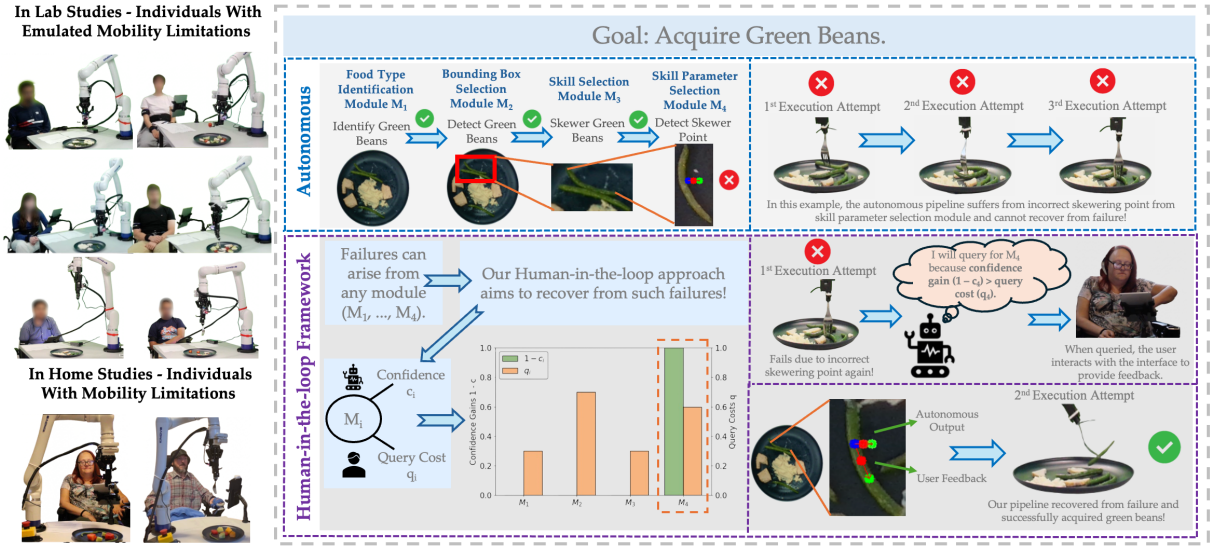

Figure 1: Our human-in-the-loop framework for failure recovery leverages confidence estimates from a modular policy, along with predicted estimates of user workload, to decide what to ask and when to act autonomously.

Abstract

Robots operating in unstructured human environments inevitably encounter failures, especially in robot caregiving scenarios. While humans can often help robots recover, excessive or poorly targeted queries impose unnecessary cognitive and physical workload on the human partner. We present a human-in-the-loop failure-recovery framework for modular robotic policies, where a policy is composed of distinct modules such as perception, planning, and control, any of which may fail and often require different forms of human feedback. Our framework integrates calibrated estimates of module-level uncertainty with models of human intervention cost to decide which module to query and when to query the human. It separates these two decisions: a module selector identifies the module most likely responsible for failure, and a querying algorithm determines whether to solicit human input or act autonomously. We evaluate several module-selection strategies and querying algorithms in controlled synthetic experiments, revealing trade-offs between recovery efficiency, robustness to system and user variables, and user workload. Finally, we deploy the framework on a robot-assisted bite acquisition system and demonstrate, in studies involving individuals with both emulated and real mobility limitations, that it improves recovery success while reducing the workload imposed on users. Our results highlight how explicitly reasoning about both robot uncertainty and human effort can enable more efficient and user-centered failure recovery in collaborative robots. Supplementary materials and videos can be found at: **emprise.cs.cornell.edu/modularhil**.

CCS Concepts

• **Human-centered computing** → **Accessibility technologies**;
• **Computer systems organization** → **Robotics**.

Keywords

Human-in-the-loop Methods, Failure Recovery

ACM Reference Format:
Rohan Banerjee[‡], Krishna Palempalli[*], Bohan Yang[*], Jiaying Fang, Alif Abdullah, Tom Silver, Sarah Dean[†], and Tapomayukh Bhattacharjee[†]. 2026. A Human-in-the-Loop Confidence-Aware Failure Recovery Framework for Modular Robot Policies. In *Proceedings of the 21st ACM/IEEE International Conference on Human-Robot Interaction (HRI '26), March 16–19, 2026,*

---

[*] Equal Contribution † Equal Advising ‡ rbb242@cornell.edu

Acknowledgement: This work was partly funded by NSF CCF 2312774 and NSF OAC-2311521, a LinkedIn Research Award, NSF IIS-244213, NSF IIS #2132846, CAREER #2238792, a PCCW Affinito-Stewart Award, an AI2050 Early Career Fellowship program at Schmidt Sciences, and NIH #T32HD113301. Full acknowledgements in Appendix I.

HRI '26, Edinburgh, Scotland, UK

ACM ISBN 979-8-4007-2128-1/2026/03
https://doi.org/10.1145/3757279.3788668

Edinburgh, Scotland, UK. ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3757279.3788668

1 Introduction

Robots deployed in the wild inevitably fail, especially when assisting individuals with mobility limitations in performing activities of daily living (ADLs) in unstructured home environments [13, 37, 40]. While humans can help robots recover [3, 10], excessive querying imposes cognitive and physical workload on the human partner [16, 36]. Designing robots that know what and when to ask humans is therefore central to effective human–robot interaction. We focus on modular robot systems, which are more interpretable than end-to-end vision–language–action (VLA) policies, and thus more amenable to structured failure recovery. However, failures can arise in any of the perception, planning, or control modules, each requiring different forms of human feedback.

Failure recovery in modular systems is difficult for two reasons. First, identifying which module has failed is non-trivial; a perception error can cascade into planning and execution, making it unclear where intervention is most effective. Second, module confidence scores are often miscalibrated and do not reliably predict whether a task will succeed. Over-querying risks frustrating users, while under-querying risks repeated failures that undermine both task performance and trust. Properly balancing task success and human workload[1] lies at the heart of collaborative autonomy. By combining *confidence-aware reasoning* with models of human workload [3, 11, 32], we can design *workload-aware interaction* strategies that recover from failures more efficiently with high user satisfaction. This is crucial in assistive settings such as physical robot caregiving, where reliable task performance and user experience affect the acceptability of robot assistance.

To address the challenge of balancing recovery efficiency with user satisfaction, we propose a human-in-the-loop failure recovery framework for modular robot policies that integrates calibrated module-level uncertainty with models of human workload. Our framework is broadly applicable to modular architectures, even VLAs themselves (which can be treated as unitary modules). The framework makes two key decisions at every recovery attempt. First, a *module selector* determines which component of the modular policy to query, e.g. perception or planning. Second, a *querying algorithm* determines whether to query the human at all or to act autonomously. This decision is critical for trading off the expected gain in task-level success against user workload. We formulate these decisions within a sequential decision-making framework, and evaluate several module-selection strategies and querying algorithms. We then deploy the framework on a robot-assisted bite acquisition system [3, 13, 14, 37], demonstrating improved recovery success and reduced user querying workload across studies involving individuals with both emulated and real mobility limitations. Our work makes the following novel contributions:

- We propose a human-in-the-loop failure-recovery framework for modular robotic policies that makes two complementary decisions: a *module selector* chooses which component of the policy to query, and a *querying algorithm* decides whether and when to solicit human input versus act autonomously.
- We incorporate calibrated estimates of module-level uncertainty together with models of human intervention cost to guide both decisions, enabling robots to recover more efficiently and to avoid unnecessary queries.
- We conduct systematic evaluations of several module-selection strategies (e.g. brute-force, graph-based, binary-tree, and mixed-integer programming (MIP) selectors) and querying algorithms (e.g. execute-first, query-then-execute, query-until-confident, and workload-aware variants) in controlled synthetic experiments, revealing trade-offs in robustness to system and user variables, efficiency, and user workload.
- We demonstrate the approach in a real-world robot-assisted bite acquisition task, achieving higher recovery success and lower user querying workload compared to baselines in two in-lab studies with emulated limitations and an in-home study involving users with real mobility limitations.

[1] In this work, we use the terms *workload* and *query cost* interchangeably.

2 Related Work

Anomaly detection and recovery in robotics. Robotic policies, whether end-to-end or modular, are prone to failure during execution, especially in unstructured environments or when encountering novel objects. Despite recent advances in end-to-end robot learning, these approaches can be difficult to interpret [1, 8, 41]. In contrast, modular methods can offer more predictable and controllable performance in certain scenarios. In this work, we focus on full-stack modular methods that consider failure detection and recovery across different modules of a robot system. While some human-in-the-loop approaches [2, 10, 22] assume that the system is in a failure state where recovery is required, our approach includes failure detection using confidence estimates.

It is natural to represent such problems using a graph representation, as in prior work on online approaches for robot failure detection and recovery [2, 7, 29]. Additionally, the overall interaction process can be modeled by certain types of graphs, ranging from finite-state machines [7], MDPs/POMDPs [12, 29], and Behavior Trees [2, 6, 26]. Although graph representations have been widely studied in prior work, our method goes further by not only modeling systems using graphs but also leveraging graph algorithms to select which modules to query for information or intervention. In particular, two of our four proposed methods for selecting which modules to query (which we denote module selectors) are graph-based methods.

Robots that ask for help. Robot systems may decide to query humans on the basis of several different criteria. Some approaches ask for help when a failure is detected after a robot action has been executed [21, 38], some based on a confidence estimate about potential robot actions to take [27, 33], and others based on an estimate about the potential information gain that could arise from soliciting human feedback [10, 30]. Departing from existing work, our query rules depend on both module confidence values and estimated human workload costs of querying.

The types of feedback the robot can ask for can also vary, ranging from natural language [5, 33, 35], to actions [21, 38, 39], to labels [5]. Our framework that detects the module to be queried is agnostic to the specific types of feedback. Unlike prior work that queries for a single action-selection module [19, 33] or reward model [10], we simultaneously consider asking for help for multiple modules in a full-stack policy architecture.

3 Problem Formulation

Module graph. We assume that our robot policy architecture $\mathcal{P} : S \to A$ is structured as a module graph G_M, which describes the relationships between the N modules in the architecture. Here, S represents the policy input state (e.g. RGB-D images), and A represents the policy action output (e.g. robot end-effector pose). The vertices in G_M are modules, and an edge exists from one module M_i to another module M_j if the output of M_i is an input to M_j.

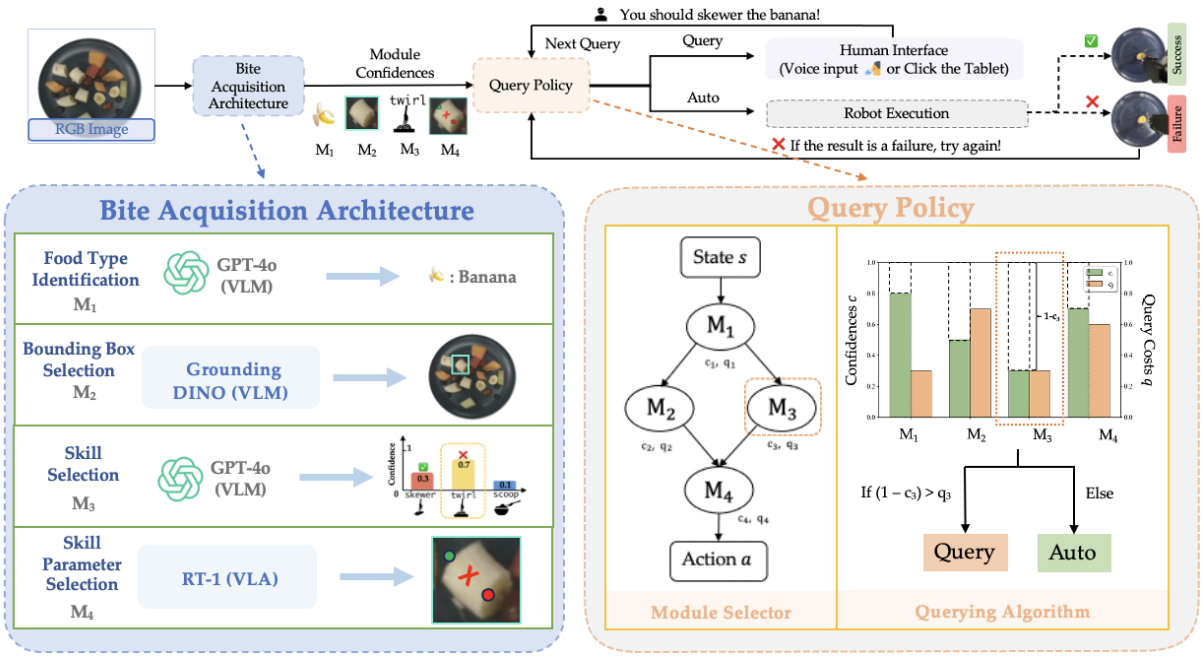

Figure 2: Overall human-in-the-loop decision failure recovery framework, grounded in the robot-assisted bite acquisition domain. The recovery framework first calls a module selector to decide which of the modules to query for (e.g. the skill selector). The framework then calls a querying algorithm, which decides whether to ask the user for help or act autonomously.

A *module* is a tuple $M = (\pi, \mathcal{X}, \mathcal{Y}, c, q(t))$ where $\pi : \mathcal{X} \to \mathcal{Y}$ is a policy function from input space $\mathcal{X}$ to output space $\mathcal{Y}$, $c : \mathcal{X} \to [0, 1]$ is a confidence score, and $q(t)$ is a time-dependent cost for querying the human about the module. For instance, a module could be a perception module that extracts geometric information about the robot's environment from RGB-D input, or a planning module that extracts a robot trajectory given obstacle states in the environment. We assume that one module in G_M (which we denote M_1) takes in the state $s \in S$ (i.e. $\mathcal{X} = S$), and one module (which we denote M_N) produces a final action $a \in A$ (i.e. $\mathcal{Y} = A$), with other modules having arbitrary input/output spaces.

Research problem. Our goal is to decide what and when to query, which we decompose into (1) a module selector algorithm ψ_{ms} which chooses a single module, and (2) a querying algorithm ψ_q that decides whether to query or to execute the autonomous action $a = \mathcal{P}(s) \in \mathcal{A}$. For a particular state $s \in S$, the module selector algorithm ψ_{ms} and querying algorithm ψ_q should minimize $J(\psi_{ms}, \psi_q)$, the weighted sum of the query cost $J_{\text{query}}(t; \psi_{ms}, \psi_q)$ and the task cost $J_{\text{task}}(t; \psi_{ms}, \psi_q)$, over the interaction horizon T:

$$J(\psi_{ms}, \psi_q) = \frac{1}{T} \sum_{t=1}^{T} \left(w J_{\text{query}} + (1 - w) J_{\text{task}} \right)$$
$$= \frac{1}{T} \sum_{t=1}^{T} \left(w \sum_{M_i \in \Omega(t)} q_i(t) + (1 - w)(1 - r(t)) \right) \quad (1)$$

where $w \in [0, 1]$ determines the trade-off between minimizing human workload and recovering from failures efficiently, $\Omega(t)$ denotes the set of modules for which we query at time t, M_i refers to an individual module in this set, $q_i(t)$ is the query cost for module i, and $r(t)$ is the binary task reward, which is equal to 1 if the execution succeeds, and 0 otherwise.

Human feedback. We assume that for every module M_i, the user can provide feedback $f \in \mathcal{Y}_i$ by choosing a possible output of that module (e.g. classifying an object or selecting a high level skill). When we receive this feedback, we replace the module output with the expert feedback f, along with an expert confidence $c = c_{\text{expert}}$.

Module redundancy. The overall task reward $r(t)$ depends on the success of individual modules. The modules can be structured in two ways, depending on the robot system architecture:

- A *redundant* manner, where at least one module M_i must succeed for their combination to succeed, so success is a union. For example, a policy architecture may include two redundant modules that predict object material properties: an off-the-shelf foundation model, and a domain-specific neural network [40].
- A *non-redundant* manner, where all modules M_i must succeed for their combination to succeed, so success is an intersection. For example, a goal-reaching manipulation policy architecture may require two modules to succeed: a goal inference module, and a planning module that infers a trajectory to the goal [25].

The overall success $r(t)$ depends on redundant and non-redundant combinations of modules, based on the robot system architecture.

4 Module Selectors

The module selector algorithms ψ_{ms} select which module to query by considering the 1-timestep version of the overall objective (Eq. 1). While we cannot determine *a priori* whether each module M_i is in failure or success state, we have access to confidences c_i. We consider three *proxy objectives* to minimize:

Proxy objective 1 (product-of-confidences):
$$w \textstyle\sum_{M_i \in \Omega(t)} q_i(t) + (1 - w)(1 - \prod_{M_i \notin \Omega(t)} c_i) \quad (2)$$

Proxy objective 2 (sum-of-uncertainties):
$$w \textstyle\sum_{M_i \in \Omega(t)} q_i(t) + (1 - w) \sum_{M_i \notin \Omega(t)} (1 - c_i) \quad (3)$$

Proxy objective 3 (redundancy-dependent):

$$w \sum_{M_i \in \Omega(t)} q_i(t) + (1 - w)(1 - \hat{r}(\Omega(t)) \quad (4)$$

where reward estimate $\hat{r}(\Omega(t))$ combines sums (redundant) and products (non-redundant) of confidences for modules $M_i \notin \Omega(t)$ (see Appendix A for justifications for how each proxy objective approximates J_{task}).

We consider four module selector algorithms—two direct proxy optimization algorithms (**Mixed-Integer Programming** and **Brute-Force**) and two graph-based algorithms (**Binary Tree Query** and **Graph Query**). We also consider three baseline module selectors, two of which do not consider query costs or confidence scores, and one that only considers confidence scores. The baselines are (1) **Never Query** which does not choose any module to query, (2) **Topo Query** which simply selects the first module in the topological ordering of the module graph G_M which has not been previously selected, and (3) **Confidence Query** which selects the module with the lowest confidence score.

4.1 Proxy Objective Optimization

We consider two module selectors that directly use the redundancy-dependent proxy objective (Eq. 4). The **Mixed-Integer Programming (MIP)** module selector directly minimizes this objective using a mixed-integer solver (SCIP [4]). The **Brute-Force** module selector evaluates this objective for querying each module M_i in the module graph G_M, and then selects the module M_i that achieves the minimal objective.

4.2 Binary Tree Query

This algorithm uses a binary tree graph to model the yes/no decision of querying each module. The edge weights are defined so that paths from the root to the leaves correspond to the product-of-confidences proxy objective (Eq. 2). We then treat the module optimization problem as a shortest-cost path problem, running Dijkstra's algorithm on the graph to obtain the module M_i.

The binary tree graph $B = (V, E)$ (shown in Fig. 3 (left)) has edges E corresponding to the actions of querying a_{query} or acting autonomously a_{auto} for module M_i, and vertices V that encode the querying decisions of modules prior to module M_i in the topological ordering of the module graph G_M. The edge costs for query actions are $C(e) = q_i(t)$, representing the query cost for module M_i. The edge costs for autonomous actions are chosen to create a telescoping sum with the correct product of confidences. In particular, $C(e) = (1 - c_i) \prod_{j \in prev} c_j$, where $prev$ are prior autonomous modules on the path from root to M_i, and if $prev$ is empty, $C(e) = 1 - c_i$.

4.3 Graph Query

This algorithm constructs a directed graph representing the modular policy and treats module selection as a shortest path problem, running Dijkstra's algorithm on the graph to select the module M_i (equivalent to optimizing the sum-of-uncertainties objective, Eq. 3).

Given the topological ordering of modules in the module graph G_M, the graph $G = (V, E)$ is shown in Fig. 3 (middle), where vertices V correspond to the state of each module, and edges E correspond to actions, both query a_{query} and autonomous a_{auto}. The vertices V consist of an initial state s_{init} and pairs of success and failure states for each module M_i ($s_{M_i,\text{success}}$ and $s_{M_i,\text{failure}}$). The edge costs for query actions are a scaled version of the query cost $C(e) = \epsilon q_i(t)$, with scaling factor $\epsilon > 0$ (for module selectors). The edge costs for autonomous actions are the module uncertainties $C(e) = 1 - c_i$.

5 Querying Algorithms

Querying algorithms decide whether to make a query or execute the action a produced by the final action module M_N (Sec. 3). We consider four querying algorithms and one baseline, illustrated in Fig. 3 (right). They use the following primitives:

- ForwardPass: represents passing the current state $s \in \mathcal{S}$ and query set $\Omega(t)$ through all of the modules in the module graph G_M to yield the final action a and updated confidences. In the manipulation setting, this could represent passing the current perceptual state (e.g. RGB-D camera data) to the policy architecture, producing the desired end-effector pose, along with confidences for all modules in the policy architecture.
- Execute: represents executing the action a and observing overall boolean $outcome$. In the manipulation setting, this could represent commanding the robot to the desired end-effector pose and assessing whether the manipulation task was successful.

All querying algorithms call the module selector ψ_{ms} to select a candidate module to query, but differ in how many queries are made prior to execution. The **Execute-First** algorithm initially executes the autonomous action a from the ForwardPass primitive. If a fails, it alternates between calling the module selector ψ_{ms} and executing until success. **Query-then-Execute** always alternates between calling the module selector ψ_{ms} and executing until success.

Algorithm 1 Execute-First and Query-then-Execute.

```
if Execute-First then
    a = ForwardPass(∅)
    outcome = Execute(a)
else if Query-then-Execute then
    outcome = FALSE
end if
while outcome = FALSE do
    Ω(t) = ψ_ms(G_M)
    a = ForwardPass(Ω(t))
    outcome = Execute(a)
end while
```

Algorithm 2 Query-until-Confident and Query-until-Confident-Workload-Aware; Query-For-All baseline.

```
outcome = FALSE
while outcome = FALSE do
    repeat
        Ω(t) = ψ_ms(G_M)
        if Query-until-Confident then
            StopCondition = [r̂(Ω(t)) > τ]
        else if Query-until-Confident-Workload-Aware then
            StopCondition = [c_expert − c_i < λ q_i(t)]
        else if Query-For-All then
            StopCondition = [Ω(t) = ∅]
        end if
    until StopCondition = TRUE
    a = ForwardPass(Ω(t))
    outcome = Execute(a)
end while
```

Query-until-Confident repeatedly calls ψ_{ms} until the proxy task reward estimate $\hat{r}(\Omega(t))$ (Eq. 4) exceeds a certain threshold τ, then executes the action a. **Query-until-Confident-Workload-Aware** repeatedly calls ψ_{ms} until the confidence gain due to querying module M_i, $c_{\text{expert}} - c_i$ (c_{expert} defined in Sec. 3), is less than the scaled cost of querying $\lambda q_i(t)$, for scaling factor $\lambda > 0$ (for querying algorithms).

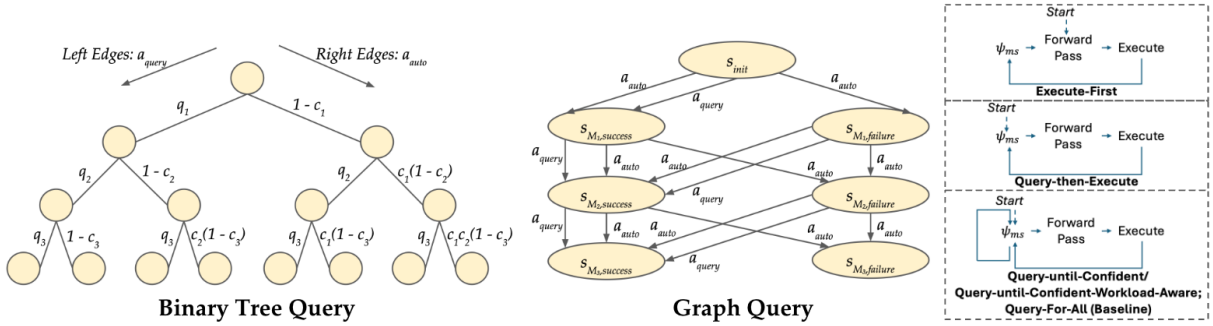

Figure 3: (left) BinaryTreeQuery graph example for N=3, (middle) GraphQuery graph example for N=3, (right) Querying algorithms and Query-For-All baseline, which decide when to query (calling module selector ψ_{ms}) and when to execute actions (calling ForwardPass to get action a, then calling Execute). Querying algorithms include Execute-First, which executes once prior to starting to query, Query-then-Execute, which alternates between querying and execution, and Query-until-Confident/Query-until-Confident-Workload-Aware, both of which repeatedly query until a stopping condition is met. The Query-For-All baseline queries for all modules before execution.

Finally, the **Query-For-All** baseline repeatedly calls ψ_{ms} until it has queried for all modules, then executes the resultant action a.

6 Synthetic Simulation: Systematic Ablations

We develop a synthetic module simulation to investigate how system and user variables affect module selector and querying algorithm performance (Secs. 4, 5). Our simulation uses N modules connected via a random module graph G_M, where modules are implemented as logic gates (AND/OR) operating on Boolean inputs (including a Boolean state s). Individual modules are set to be in either a success or failure state: modules in success states output their logic gate value, while modules in failure states always output False. See Appendix F.1 for hyperparameter details (including proxy objective weighting and GraphQuery ϵ).

Independent variables. We examine four variables covering key system and user factors that affect robot and human-robot interaction performance (full settings in Appendix F.1). We consider three system variables:

- **The number of modules N (Sec. 6.1).** The number of modules in a robot policy architecture can vary, ranging from simple modular perception, planning, and control architectures, to more complex modular architectures involving up to 100 modules.
- **Module redundancy/graph structures G_M (Sec. 6.2).** Robot policy architectures include both redundant and non-redundant structures (Sec. 3). Different module selectors have different redundancy assumptions based on their proxy objective. We consider 4 redundancy structures with varied module types and orderings: fully redundant (all-OR), fully dependent (all-AND), fully redundant followed by fully dependent (OR-then-AND), fully dependent followed by fully redundant (AND-then-OR).
- **Confidence values c_i (Sec. 6.3).** Robot policy architectures can have a diversity of confidence scores across the modules. We assign either a high or a low confidence value to each module in G_M. We additionally treat a module's confidence value as the probability that the module is in a success state.

And we consider one user variable:

- **Query costs q_i (Sec. 6.4).** The costs of querying for each module in a policy architecture can vary, due to the diversity of feedback types. We consider different values of uniform query costs q_i for all modules.

We vary each independent variable in isolation. When a variable is not varied, we fix its value as follows: (1) N=10 modules, (2) Fully dependent (all-AND) module structure, (3) 3 modules with confidence 0.1 and all other modules with confidence 1, (4) Uniform query cost of 0.32. We also assume no noise in the expert (c_{expert}=1.0). Additionally, we use a fixed querying algorithm (Query-Until-Confident-Workload-Aware) when analyzing module selectors, and a fixed module selector (GraphQuery) when analyzing querying algorithms.

Metrics. We evaluate the methods using 5 metrics (lower is better):

- **Task Cost.** 0 if the agent successfully recovered from the failure after T timesteps, 1 otherwise. This represents whether the robot could recover from the failure given its time horizon constraint.
- **Query Cost.** $\sum_{t=1}^{T} b_t q_i(t)$, where T is the number of timesteps needed to achieve success or the maximum time horizon (which is proportional to the number of modules N), and b_t is a binary indicator variable for whether the robot queried at timestep t. This represents the total user workload used to recover from the robot's failure.
- **Number of Failed Attempts.** $\sum_{t=1}^{T} g_t$, where g_t is a binary indicator variable for whether a failed execution occurred at timestep t. This represents the number of failed robot executions encountered before the robot recovered from the failure.
- **Computation Time.** $\sum_{t=1}^{T}(Time(t))$, where $Time(t)$ is the total runtime between successive EXECUTE statements. This represents the total computation time used by the robot to run the module selector and the querying algorithm at each timestep (not including the time required to query).
- **Total Timesteps T.** This includes both the number of queries and the number of failed executions encountered before the robot recovered from the failure.

Fig. 4(a) compares the algorithm performance for each variable, highlighting one metric[2] (full results detailed in Appendix F). We additionally highlight results in Appendix F.6 for varying an additional user variable, the expert confidence level c_{expert} (introduced in Sec. 3), where we find that the BruteForce and GraphQuery module selectors and both Query-Until-Confident

[2]Because the computation time of the GraphQuery exceeded 1 second for graph sizes $N \geq 50$, we omit the computation time for this module selector. Additionally, we omit reporting MIP results for all graph sizes because its computation time also exceeds this threshold.

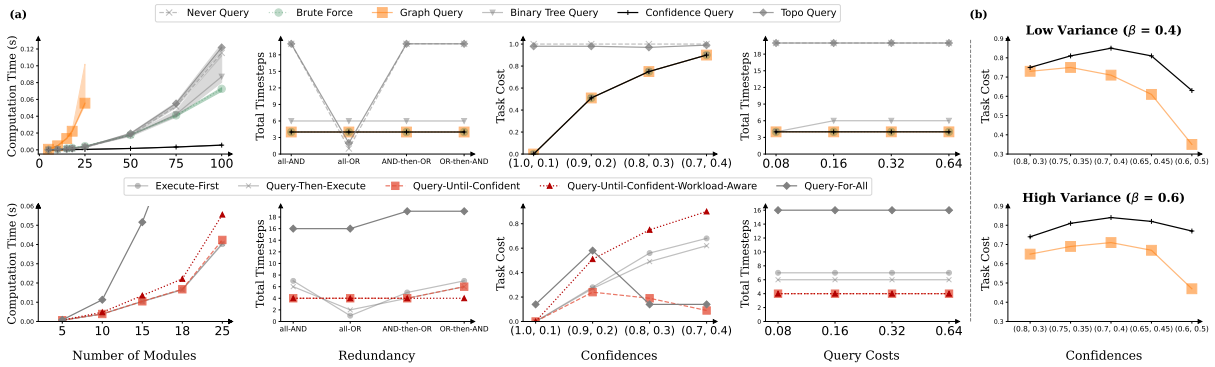

Figure 4: (a) Systematic ablation experiments, showing the 4 independent variables in our simulations: (1) number of modules, (2) graph redundancies, (3) confidences, (4) query costs, with median values across 100 trials (mean for Task Cost)[2]. We find that the BruteForce and GraphQuery module selectors (along with the ConfidenceQuery baseline) are the most robust to varying redundancy, confidences, and query costs, with BruteForce and ConfidenceQuery being the most scalable. Additionally, we find that the Query-until-Confident and Query-until-Confident-Workload-Aware querying algorithms are the most robust across redundancy and query costs, with Query-until-Confident having the best scalability and robustness to confidences; (b) Module heterogeneity experiments. We find that GraphQuery outperforms ConfidenceQuery, particularly when module confidences overlap and workload variance β is high. Detailed results in Appendix F.

querying algorithms perform best in high expert confidence regimes, with performance degrading as the confidence decreases.

6.1 Varying number of modules N

Module selectors. We find that all metrics except for *Computation Time* are invariant to N, for all module selectors except NeverQuery and TopoQuery. ConfidenceQuery is the most scalable module selector with linear time complexity in N (Fig. 4(a)), making it the most computationally efficient for large policy architectures. Since *Computation Time* scales with T, methods that require many timesteps (i.e. NeverQuery and TopoQuery) incur higher computation time.

Querying algorithms. We find that all metrics except *Computation Time* are invariant to N, for all querying algorithms except Query-For-All, with *Task Cost* and *Query Cost* of 0 and 0.96, respectively. We note that Query-Until-Confident-Workload-Aware has the highest *Computation Time* (Fig. 4(a)), due to the additional step to predict the cost of querying, while the other three querying algorithms have nearly-identical values for *Computation Time*. For large robot policy architectures, we recommend not selecting Query-Until-Confident-Workload-Aware if runtime is crucial.

6.2 Varying graph structure G_M

Module selectors. We find that all metrics are lower for the all-OR structure across module selectors (except *Query Cost*, which is invariant to the structure). For this redundant structure, there is little benefit to ask for help to recover from failures, leading NeverQuery and TopoQuery to perform well in *Total Failed Attempts* and *Total Timesteps*. Module selector rankings (Fig. 4(a)) remain invariant for the other redundancy structures. We find that GraphQuery is competitive with the other module selectors, even when its proxy objective does not match the redundancy structure.

Querying algorithms. We find that all metrics are lower for the all-OR structure across querying algorithms. In general, both Query-until-Confident variants have the lowest *Total Failed Attempts*, except in the all-OR setting (where Execute-First and Query-then-Execute outperform them). Thus, if the robot policy architecture is not fully redundant, we recommend either Query-until-Confident variant to maximize recovery efficiency.

6.3 Varying confidence values c_i

Module selectors. We find that all metrics (e.g. *Task Cost*, Fig. 4(a)) are higher across module selectors when confidences overlap. As BruteForce, GraphQuery, and ConfidenceQuery are the most competitive module selectors regardless of confidence level, we recommend any of these module selectors.

Querying algorithms. We find that all metrics (e.g. *Task Cost*, Fig. 4(a)) are higher across querying algorithms when confidences overlap, except for Query-Until-Confident and Query-For-All. We find that the Query-Until-Confident querying algorithm is the most robust to confidence variations. We recommend any querying algorithm if the confidence scores are well-separated in the robot policy architecture, and Query-Until-Confident if confidence values are overlapping.

6.4 Varying query costs q_i

Module selectors. We find that all metrics except *Task Cost* are invariant to q_i, with BruteForce, GraphQuery, and ConfidenceQuery having lower *Total Timesteps* (Fig. 4(a)) compared to the other module selectors. Regardless of the level of user querying workload, we recommend either of these module selectors.

Querying algorithms. We find that besides *Query Cost*, all metrics are largely invariant to q_i across querying algorithms. As both Query-Until-Confident variants have the lowest *Total Failed Attempts* and generally lower *Total Timesteps* (Fig. 4(a)), we recommend either of these variants.

7 Synthetic Simulation: Module Heterogeneity

We now consider a simulation setting with an additional user variable, **query cost variance** β, allowing the query costs q_i to vary across modules. This models more realistic cost variation across feedback types. We sample each q_i uniformly around a nominal value Q, i.e., $q_i \sim \text{Uniform}[(1-\beta)Q, (1+\beta)Q]$, with Q=0.32 (the query cost value used when not varied in Sec. 6). We compare the best performing module selectors from Sec. 6

based on *Task Cost*: the query cost-aware method GraphQuery, and the confidence-only baseline ConfidenceQuery (with the Query-Until-Confident-Workload-Aware querying algorithm).

We find that GraphQuery consistently outperforms ConfidenceQuery in the *Total Timesteps* and *Task Cost* metrics, especially when module confidences overlap or workload variance is high (*Task Cost* in Fig. 4(b)). Under these conditions, upstream modules with lower confidence and high workload may succeed, whereas downstream modules with higher confidence but low workload may fail. ConfidenceQuery incorrectly selects these upstream modules first, whereas GraphQuery correctly selects the downstream module (full results in Appendix G).

8 Robot-Assisted Bite Acquisition Experiments

Our real robot experiments use a robot-assisted bite acquisition architecture with $N = 4$ modules: food type identification (GPT-4o), bounding box selection (GroundingDINO), skill selection (GPT-4o), and skill parameter selection (RT-1). The first three modules are VLMs, while the skill selection module is a VLA. Food type identification and skill selection support learning from feedback via retrieval-augmented generation (RAG) (architecture and RAG details in Appendix C). We develop calibrated confidence scores for each module using a population-based interval procedure (detailed in Appendix C.1-C.2). We estimate module query costs using a predictive workload model from prior work [3] (Appendix E).

To select a module selector and query algorithm for bite acquisition, we adopt the recommendation from the closest synthetic setting as follows. We use the N=10 setting to approximate system scale; we model bite acquisition as non-redundant (all-AND), since all modules must be correct for success (Sec. 3); we use the confidence value setting $(1, 0.1)$ to match our binary calibrated module confidence scores (Appendix C.2); we select q_i=0.32 to match the empirical mean of workload model predictions; we assume c_{expert}=1.0 as expert feedback is available for all modules. Based on these conditions (Sec. 6), along with the query cost variance experiments (Sec. 7), we use the GraphQuery module selector with the Query-Until-Confident-Workload-Aware querying algorithm, which performed best in *Total Timesteps* and *Task Cost*.[3]

User Studies. We conducted three IRB-approved studies to evaluate algorithm task success and querying workload: two in-lab studies, and one in-home study involving two individuals with severe mobility limitations. All studies use a Kinova Gen3 6-DoF arm with a Robotiq 2F-85 gripper, and we replicate a custom-designed feeding tool for the user studies [18]. A key methodological feature of both of our in-lab studies, which supports ecological validity, is the emulation of mobility constraints in participants without pre-existing mobility impairments using occupational therapy resistance bands [24] (Fig. 5 (left)).

For each method, the robot acquires 5 food items from a plate, with a maximum of 3 acquisition attempts per item to maintain a reasonable study duration. The robot may ask 4 types of queries, corresponding to each module in the bite acquisition system (Appendix C.4). The method and plate sequences are both counterbalanced across the users in each study (Fig. 5 (middle, top)).

Metrics. We report 4 subjective metrics (Mental/Physical Demand, Effort, Subjective Success, Satisfaction) and 3 objective metrics (Mean Queries/Executions/Successful Bites Per Plate), detailed in Appendix H.

8.1 In-lab real-robot user study

We first conducted an in-lab study with 10 participants without mobility limitations (7 male, 3 female; ages 19–33; 40% with prior robot experience) (Fig. 5 (right)). We compared our method (GraphQuery module selector and Query-Until-Confident-Workload-Aware querying algorithm) against two additional baselines: *Never Query* (NeverQuery module selector and Query-then-Execute querying algorithm) and *Always Query* (TopoQuery module selector and Query-For-All querying algorithm). The study evaluated whether our method (1) improved task success over *Never Query* and (2) reduced querying workload over *Always Query*. Participants interacted with two plates: a "savory salad" (chicken, lettuce, cherry tomatoes) and a "Thanksgiving meal" (chicken, green beans, mashed potatoes).

As shown in Fig. 6(a–b), our method achieved the highest user satisfaction, significantly lower mental/physical workload than *Always Query*, and higher subjective success than *Never Query* (Wilcoxon signed-rank test, $\alpha = 0.05$). Our method is more efficient than *Always Query*, with higher task success than *Never Query*.

8.2 In-lab real-robot user study with module heterogeneity

To distinguish our method from ConfidenceQuery, we introduced module heterogeneity into our bite acquisition setting by incorporating a faulty food detector M_1 (Appendix D), similar to our second simulation study (Sec. 7). We conducted a second in-lab study with 10 additional participants (3 male, 6 female, 1 non-binary; ages 20-30; 30% with prior robot experience) using the "savory salad" plate (Fig. 5 (right)). We sought to determine whether our method improved task success and querying workload compared to the *Confidence Query* baseline (ConfidenceQuery module selector and Query-Until-Confident-Workload-Aware querying algorithm).

We found that our method produced significantly higher task performance, lower workload, and higher satisfaction than *Confidence Query* (Wilcoxon test, $\alpha = 0.05$; Fig. 5(c-d)), reinforcing the value of principled uncertainty and workload integration.

8.3 In-home real-robot user study

We conducted an in-home user study on 2 individuals with mobility limitations who require feeding assistance (Fig. 5 (middle, bottom)). One is a female, 48 years old, who has had Multiple Sclerosis for 17 years, and the other is a male, 47 years old, who has been paralyzed with a C4-C6 spinal cord injury for 27 years. Based on the in-lab study results (Secs. 8.1, 8.2), we compared our method to *NeverQuery* and *AlwaysQuery*. We evaluated the methods on a "mixed salad" plate with additional food item diversity (watermelons, cantaloupes, honeydew, green beans).

We find that the mental/physical demand and effort of our method are lower compared to *Always Query*, and that users rated our method as more successful compared to *Never Query* (Fig. 6(e-f)). In general, our method achieves the best user satisfaction score.

Both users expressed that they liked the system, as their satisfaction scores were higher for our method compared to both baselines. One user's absolute satisfaction level was reduced due to the need for physical interaction with the interface. They noted that a more accessible option, such as voice control, would significantly improve their experience. They commented that because they use a tablet interface in everyday interactions, answering a few more

[3]While GraphQuery and BruteForce were equally competitive, BruteForce produced false-positive queries with our binary confidence scores. We chose Query-until-Confident-Workload-Aware over Query-until-Confident for better generalization to time-varying workload.

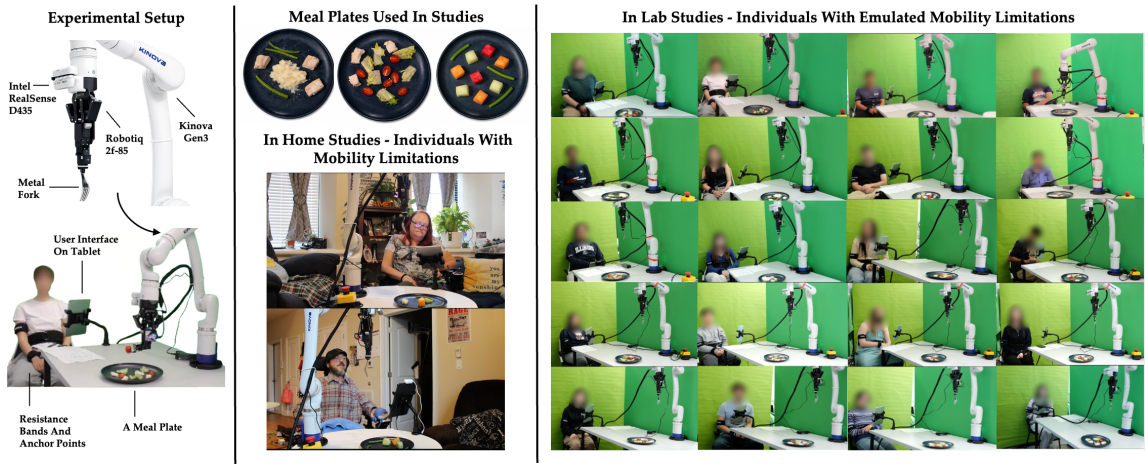

Figure 5: Experimental setup. (left) Robot and user study setup; (middle, top) Meal plates used in user studies, including "Thanksgiving meal", "savory salad", and "mixed salad"; (middle, bottom) Users with mobility limitations from in-home user study; (right) Users with emulated mobility limitations from two in-lab user studies.

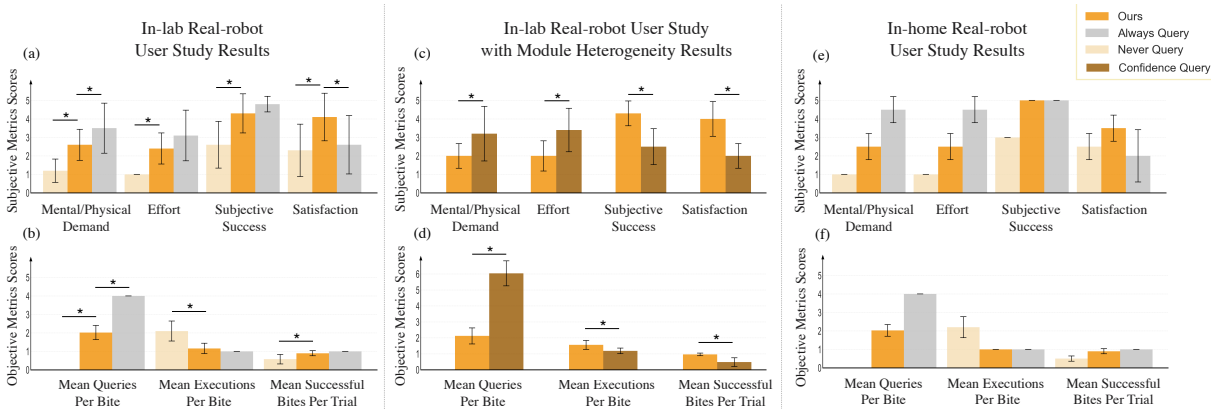

Figure 6: User study metrics. In-lab real-robot study: (a) subjective scores, (b) objective scores; In-lab real-robot study with module heterogeneity: (c) subjective scores, (d) objective scores; In-home real-robot study: (e) subjective scores, (f) objective scores (∗ indicates statistical significance $p < 0.05$.).

queries on the tablet in the study did not require significantly extra effort for the study duration. However, they still rated *AlwaysQuery* as having higher effort than our method. The other user mentioned that the time required for the robot to execute actions (higher when the robot failed more frequently) was an additional source of frustration that increased overall workload.

9 Discussion

Our framework readily extends to other modular robot systems, which we illustrate by considering two additional domains: a feeding architecture with visuo-haptic perceptual redundancy [40], and a large multi-robot swarm system [34]. The feeding system maps naturally to an OR-then-AND redundancy structure (due to the perceptual redundancy) with low query cost ($q_i = 0.16$), yielding the same recommendation as our primary setting (Fig. 4). We can model the swarm domain with $N = 100$ to capture its scale and an OR redundancy structure, leading to the ConfidenceQuery module selector for its superior scalability (Fig. 4, first column), and either Query-until-Confident querying algorithm variant. Both resulting recommendations align with domain intuition.

A key challenge in applying our framework to bite acquisition was developing well-calibrated confidence scores for each module. This involved adapting RT-1 to produce novel confidence scores, and developing a common calibration procedure for all modules that could operate under their varying empirical confidence distributions (Appendix C.1-C.2). While our confidence estimates were reasonably calibrated, future work could explore conformal prediction [27, 33] or data-driven calibration methods [20]. Future work could also consider module selectors that handle more complex redundancy structures. While we studied graph-based module selectors for product-of-confidences and sum-of-uncertainties objectives, a hybrid selector could adaptively combine these structures based on which modules are redundant. Finally, longer-term interactions with end users may further differentiate human-in-the-loop algorithms, beyond our observed satisfaction trends. Future evaluations could also consider more diverse in-the-wild dishes with richer food sets, more complex geometries, and pre-manipulation skills [15, 18, 40].

References

[1] Christopher Agia, Rohan Sinha, Jingyun Yang, Ziang Cao, Rika Antonova, Marco Pavone, and Jeannette Bohg. 2025. Unpacking Failure Modes of Generative Policies: Runtime Monitoring of Consistency and Progress. In *Conference on Robot Learning*. PMLR, 689–723.

[2] Faseeh Ahmad, Matthias Mayr, Sulthan Suresh-Fazeela, and Volker Krueger. 2024. Adaptable Recovery Behaviors in Robotics: A Behavior Trees and Motion Generators (BTMG) Approach for Failure Management. *arXiv preprint arXiv:2404.06129* (2024).

[3] Rohan Banerjee, Rajat Kumar Jenamani, Sidharth Vasudev, Amal Nanavati, Sarah Dean, and Tapomayukh Bhattacharjee. 2024. To Ask or Not To Ask: Human-in-the-loop Contextual Bandits with Applications in Robot-Assisted Feeding. *arXiv preprint arXiv:2405.06908* (2024).

[4] Suresh Bolusani, Mathieu Besançon, Ksenia Bestuzheva, Antonia Chmiela, João Dionísio, Tim Donkiewicz, Jasper van Doornmalen, Leon Eifler, Mohammed Ghannam, Ambros Gleixner, et al. 2024. The SCIP optimization suite 9.0. *arXiv preprint arXiv:2402.17702* (2024).

[5] Maya Cakmak and Andrea L Thomaz. 2012. Designing robot learners that ask good questions. In *Proceedings of the seventh annual ACM/IEEE international conference on Human-Robot Interaction*. 17–24.

[6] Michele Colledanchise and Petter Ögren. 2018. *Behavior trees in robotics and AI: An introduction*. CRC Press.

[7] Cristina Cornelio and Mohammed Diab. 2024. Recover: A Neuro-Symbolic Framework for Failure Detection and Recovery. *arXiv preprint arXiv:2404.00756* (2024).

[8] Jiafei Duan, Wilbert Pumacay, Nishanth Kumar, Yi Ru Wang, Shulin Tian, Wentao Yuan, Ranjay Krishna, Dieter Fox, Ajay Mandlekar, and Yijie Guo. 2024. Aha: A vision-language-model for detecting and reasoning over failures in robotic manipulation. *arXiv preprint arXiv:2410.00371* (2024).

[9] Ryan Feng, Youngsun Kim, Gilwoo Lee, Ethan Gordon, Matthew Schmittle, Shivaum Kumar, Tapomayukh Bhattacharjee, and Siddhartha Srinivasa. 2019. Robot-Assisted Feeding: Generalizing Skewering Strategies across Food Items on a Realistic Plate. (06 2019). doi:10.48550/arXiv.1906.02350

[10] Tesca Fitzgerald, Pallavi Koppol, Patrick Callaghan, Russell Quinlan Jun Hei Wong, Reid Simmons, Oliver Kroemer, and Henny Admoni. 2022. INQUIRE: INteractive querying for user-aware informative REasoning. In *6th Annual Conference on Robot Learning*.

[11] Lex Fridman, Bryan Reimer, Bruce Mehler, and William T Freeman. 2018. Cognitive load estimation in the wild. In *Proceedings of the 2018 chi conference on human factors in computing systems*. 1–9.

[12] Émiland Garrabé, Pierre Teixeira, Mahdi Khoramshahi, and Stéphane Doncieux. 2024. Enhancing Robustness in Language-Driven Robotics: A Modular Approach to Failure Reduction. *arXiv preprint arXiv:2411.05474* (2024).

[13] Ethan K Gordon, Xiang Meng, Tapomayukh Bhattacharjee, Matt Barnes, and Siddhartha S Srinivasa. 2020. Adaptive robot-assisted feeding: An online learning framework for acquiring previously unseen food items. In *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 9659–9666.

[14] Ethan K Gordon, Sumegh Roychowdhury, Tapomayukh Bhattacharjee, Kevin Jamieson, and Siddhartha S Srinivasa. 2021. Leveraging post hoc context for faster learning in bandit settings with applications in robot-assisted feeding. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 10528–10535.

[15] Nayoung Ha, Ruolin Ye, Ziang Liu, Shubhangi Sinha, and Tapomayukh Bhattacharjee. 2024. REPeat: A Real2Sim2Real Approach for Pre-acquisition of Soft Food Items in Robot-assisted Feeding. In *2024 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 7048–7055.

[16] Sandra G Hart and Lowell E Staveland. 1988. Development of NASA-TLX (Task Load Index): Results of empirical and theoretical research. In *Advances in psychology*. Vol. 52. Elsevier, 139–183.

[17] Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, et al. 2024. Gpt-4o system card. *arXiv preprint arXiv:2410.21276* (2024).

[18] Rajat Kumar Jenamani, Priya Sundaresan, Maram Sakr, Tapomayukh Bhattacharjee, and Dorsa Sadigh. 2024. FLAIR: Feeding via Long-horizon AcquIsition of Realistic dishes. *arXiv preprint arXiv:2407.07561* (2024).

[19] Ulas Berk Karli, Tetsu Kurumisawa, and Tesca Fitzgerald. [n. d.]. Ask Before You Act: Token-Level Uncertainty for Intervention in Vision-Language-Action Models. In *Second Workshop on Out-of-Distribution Generalization in Robotics at RSS 2025*.

[20] Ulas Berk Karli, Ziyao Shangguan, and Tesca FItzgerald. 2025. INSIGHT: INference-time Sequence Introspection for Generating Help Triggers in Vision-Language-Action Models. *arXiv preprint arXiv:2510.01389* (2025).

[21] Ross A Knepper, Stefanie Tellex, Adrian Li, Nicholas Roy, and Daniela Rus. 2015. Recovering from failure by asking for help. *Autonomous Robots* 39 (2015), 347–362.

[22] Ali Larian, Atharv Belsare, Zifan Wu, and Daniel S Brown. 2025. Learner and Teacher Perspectives on Learning Rewards from Multiple Types of Human Feedback. *RSS 2025 Workshop Human-in-the-Loop Robot Learning: Teaching, Correcting, and Adapting* (2025).

[23] Shilong Liu, Zhaoyang Zeng, Tianhe Ren, Feng Li, Hao Zhang, Jie Yang, Chunyuan Li, Jianwei Yang, Hang Su, Jun Zhu, and Lei Zhang. 2023. Grounding DINO: Marrying DINO with Grounded Pre-Training for Open-Set Object Detection. *arXiv preprint arXiv:2303.05499* (2023).

[24] Ziang Liu, Yuanchen Ju, Yu Da, Tom Silver, Pranav N Thakkar, Jenna Li, Justin Guo, Katherine Dimitropoulou, and Tapomayukh Bhattacharjee. 2025. GRACE: Generalizing Robot-Assisted Caregiving with User Functionality Embeddings. In *2025 20th ACM/IEEE International Conference on Human-Robot Interaction (HRI)*. IEEE, 686–695.

[25] Xiao Ma, Sumit Patidar, Iain Haughton, and Stephen James. 2024. Hierarchical Diffusion Policy for Kinematics-Aware Multi-Task Robotic Manipulation. *CVPR* (2024).

[26] Matthias Mayr, Faseeh Ahmad, Konstantinos Chatzilygeroudis, Luigi Nardi, and Volker Krueger. 2022. Skill-based multi-objective reinforcement learning of industrial robot tasks with planning and knowledge integration. In *2022 IEEE International Conference on Robotics and Biomimetics (ROBIO)*. IEEE, 1995–2002.

[27] James F Mullen Jr and Dinesh Manocha. 2024. Towards Robots That Know When They Need Help: Affordance-Based Uncertainty for Large Language Model Planners. *arXiv preprint arXiv:2403.13198* (2024).

[28] Krishna Palempalli, Rohan Banerjee, Sarah Dean, and Tapomayukh Bhattacharjee. 2025. Human-in-the-loop Foundation Model Failure Recovery for Robot-Assisted Bite Acquisition. In *1st Workshop on Safely Leveraging Vision-Language Foundation Models in Robotics: Challenges and Opportunities*. https://openreview.net/forum?id=I7UG7eC11i

[29] Tianyang Pan, Andrew M Wells, Rahul Shome, and Lydia E Kavraki. 2022. Failure is an option: task and motion planning with failing executions. In *2022 International Conference on Robotics and Automation (ICRA)*. IEEE, 1947–1953.

[30] Rafael Papallas and Mehmet R Dogar. 2022. To ask for help or not to ask: A predictive approach to human-in-the-loop motion planning for robot manipulation tasks. In *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 649–656.

[31] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. 2021. Learning Transferable Visual Models From Natural Language Supervision. arXiv:2103.00020 [cs.CV] https://arxiv.org/abs/2103.00020

[32] Akilesh Rajavenkatanarayanan, Harish Ram Nambiappan, Maria Kyrarini, and Fillia Makedon. 2020. Towards a real-time cognitive load assessment system for industrial human-robot cooperation. In *2020 29th IEEE International Conference on Robot and Human Interactive Communication (RO-MAN)*. IEEE, 698–705.

[33] Allen Z Ren, Anushri Dixit, Alexandra Bodrova, Sumeet Singh, Stephen Tu, Noah Brown, Peng Xu, Leila Takayama, Fei Xia, Jake Varley, et al. 2023. Robots that ask for help: Uncertainty alignment for large language model planners. *arXiv preprint arXiv:2307.01928* (2023).

[34] Michael Rubenstein, Alejandro Cornejo, and Radhika Nagpal. 2014. Programmable self-assembly in a thousand-robot swarm. *Science* 345, 6198 (2014), 795–799.

[35] Lucy Xiaoyang Shi, Zheyuan Hu, Tony Z Zhao, Archit Sharma, Karl Pertsch, Jianlan Luo, Sergey Levine, and Chelsea Finn. 2024. Yell at your robot: Improving on-the-fly from language corrections. *arXiv preprint arXiv:2403.12910* (2024).

[36] Joshua Bhagat Smith, Prakash Baskaran, and Julie A Adams. 2022. Decomposed physical workload estimation for human-robot teams. In *2022 IEEE 3rd International Conference on Human-Machine Systems (ICHMS)*. IEEE, 1–6.

[37] Priya Sundaresan, Suneel Belkhale, and Dorsa Sadigh. 2022. Learning visuo-haptic skewering strategies for robot-assisted feeding. In *6th Annual Conference on Robot Learning*.

[38] Stefanie Tellex, Ross Knepper, Adrian Li, Daniela Rus, and Nicholas Roy. 2014. Asking for help using inverse semantics. (2014).

[39] Shivam Vats, Michelle Zhao, Patrick Callaghan, Mingxi Jia, Maxim Likhachev, Oliver Kroemer, and George Konidaris. 2025. Optimal Interactive Learning on the Job via Facility Location Planning. *arXiv preprint arXiv:2505.00490* (2025).

[40] Zhanxin Wu, Bo Ai, Tom Silver, and Tapomayukh Bhattacharjee. 2025. SAVOR: Skill Affordance Learning from Visuo-Haptic Perception for Robot-Assisted Bite Acquisition. *arXiv preprint arXiv:2506.02353* (2025).

[41] Chen Xu, Tony Khuong Nguyen, Emma Dixon, Christopher Rodriguez, Patrick Miller, Robert Lee, Paarth Shah, Rares Ambrus, Haruki Nishimura, and Masha Itkina. 2025. Can we detect failures without failure data? uncertainty-aware runtime failure detection for imitation learning policies. *arXiv preprint arXiv:2503.08558* (2025).

A Proxy Objectives

Justification for proxy objective 1: the second term can be a reasonable approximation for $1 - \mathbb{E}[r_{\text{task}}]$, where the expectation is over model uncertainties (represented by c_i):

$$\begin{aligned}\mathbb{E}[r_{\text{task}}] &= P(\cap_i M_i \text{ succeeds}) \\ &= \prod_{M_i} P(M_i \text{ correct} | \text{predecessor } M_j \text{ correct}) \\ &= \prod_{M_i \notin M_q} c_i\end{aligned}$$

Justification for proxy objective 2: Let $u_i = 1 - c_i$ be the uncertainty of module i. Then we can use union bound to show that:

$$\begin{aligned}P(\text{system fail}) &= P(\cup_i M_i \text{ fails}) \\ &\leq \sum_i P(M_i \text{ fails}) \\ &= \sum_{M_i \notin M_q} u_i + \sum_{M_i \in M_q} 0 \\ &= \sum_{M_i \notin M_q} u_i\end{aligned}$$

Thus, the proxy objective 2 task component is an upper bound in $P(\text{system fail}) = 1 - \mathbb{E}[r_{\text{task}}]$, meaning that proxy objective 2 is a loose upper bound on the original objective, so minimizing proxy objective 2, could also minimize the original objective.

B Algorithm Performance Analysis

Suppose that we have a module graph G_M where we have exactly one module in failure (denoted M_f), with all other modules in success. Additionally, we assume that the cost of querying $q_i = 0.1$ for all modules. The module in failure has confidence $c_f = 0.1$, and all other modules have confidence $c_i = 0.1$.

B.1 GraphQuery

Recall that in GraphQuery, we assign edge costs $C(e) = 1 - c_i$ for the autonomous edges. The graph algorithm thus queries for a module if $1 - c_i > \epsilon q$; or:

$$q < (1 - c_i)/\epsilon \tag{5}$$

We will never query for any of the modules in success because $c_i = 1$ for these modules (as $1 - c_i$ will always be 0, and we assume that q is nonnegative, so condition 5 can never be met), so we don't have to worry about querying for modules that are upstream of M_f.

In our scenario, we will query for M_f since $q = 0.1$; and $(1 - c_i)/\epsilon = (1 - 0.1) = 0.9$; thus condition 5 is met. Thus, GraphQuery will query correctly at the first timestep.

For fixed $q = 0.1$, the critical value of ϵ above which we would cease to query is $\epsilon_{crit} = (1 - c_i)/(q) = (1 - 0.1)/(0.1) = \boxed{9}$.

For fixed $\epsilon = 1$, the critical value of q above which we would cease to query is $q_{crit} = (1 - c_i)/(\epsilon) = \boxed{0.9}$

B.2 MIP

Recall that we approximate the expected task reward $\mathbb{E}[r_{\text{task}}]$ as follows:

$$\mathbb{E}[r_{task}] = \prod_{M_i} P(M_i \text{ correct}) = \prod_{M_i} c_i$$

If we decide to query for a particular module i, we assume that $c_i = 1$ since we're using the expert feedback.

- Hypothetical $J(\psi_{ms}, \psi_q)$ of not querying for any module: $1 - (1^{N-1} \cdot 0.1) = 0.9$
- Hypothetical $J(\psi_{ms}, \psi_q)$ of querying for any of the non-failure modules: $q + 1 - (1^{N-2} \cdot 0.1 \cdot 1) = q + 0.9 = q + 0.9$
- Hypothetical cost of querying for the failure module: $q + 1 - (1^{N-1} \cdot 1) = q$

Thus, we should choose to query for the failure module as long as $q < 0.9$, so in our scenario, MIP will always choose to query for the module in failure at the first timestep.

C Bite Acquisition Architecture

Fig. 2 (left) shows the modular bite acquisition architecture that we use in our work, which is an adaptation of a state-of-the-art architecture [18], including novel foundation model components.

This autonomous system consists of four submodules, each with an associated confidence estimate:

- $\mathbf{M_1}$: GPT-4o [17] food type detector that processes a whole plate RGB image $z_{\text{RGB}} \in \mathbb{R}^{H \times W \times 3}$ and identifies a candidate set of food labels $\mathcal{L} = \{l_1, l_2, \ldots, l_K\}$, where K is the number of unique food categories detected (e.g., cherry tomatoes, pineapple, etc.). From this set, M_1 selects the single label with the highest confidence score as its output:

$$M_1(z_{\text{RGB}}) \rightarrow l^*$$

- $\mathbf{M_2}$: GroundingDINO [23] bounding box detector that takes as input the selected food type label l^* from M_1 together with the plate RGB image z_{RGB}. It outputs one or more bounding boxes $\mathcal{B}(l^*) = \{b_1, b_2, \ldots, b_J\}$, where each b_j corresponds to a detected instance of the food type l^* in the image. Thus,

$$M_2(z_{\text{RGB}}, l^*) \rightarrow \mathcal{B}(l^*).$$

- $\mathbf{M_3}$: GPT-4o [17] skill selector that, given a detected food label l^* and its corresponding bounding box b_i, predicts the optimal skill $a_i^h \in \mathcal{A}_h$, where $\mathcal{A}_h$ is a set of skills (e.g. Skewering, Scooping, Twirling).

$$M_3(l^*, b_i) \rightarrow a_i^h$$

- $\mathbf{M_4}$: A VLA model RT-1 that refines the skill action a_i^h into precise skill parameters $a_i^l \in \mathbb{R}^4$. It takes in the skill action a_i^h and the cropped image from the corresponding bounding box b_i. We adapt RT-1 for modular control by fine-tuning it with a new regression projection head on an aggregated dataset consisting of SPANet samples [9], along with ~1,000 additional labeled images that we collected. This design enables us to attribute RT-1's uncertainty purely to its skill-parameter prediction, while M_1, M_2, and M_3 handle food classification, location estimation, and skill selection.

  Skill-specific parameterization. We represent all low-level skills using a unified 2D action vector $a_i^l = (x_1, y_1, x_2, y_2)$, with its semantics based on the selected skill a_i^h. For *skewering* and *twirling*, the interaction

point is defined as the midpoint between the two predicted points,

$$(x_c, y_c) = \left(\frac{x_1 + x_2}{2}, \frac{y_1 + y_2}{2}\right),$$

and the fork tine direction θ computed as follows:

$$\theta = \arctan(y_2 - y_1, x_2 - x_1).$$

For *scooping*, (x_1, y_1) denotes the start of the scooping motion and (x_2, y_2) denotes the end of the scooping trajectory. This parameterization follows prior work on vision-conditioned manipulation primitives (e.g., FLAIR [18]), while enabling a unified continuous action space across heterogeneous skills.

$$M_4(a_i^h, b_i) \rightarrow a_i^l$$

C.1 Calibration Datasets and Confidence Intervals

To support uncertainty-aware decision-making in our bite-acquisition pipeline, we adapt the work mentioned in [28]. We construct calibration datasets and associated confidence intervals for all modules: M_1, M_2, M_3, M_4. Each calibration dataset records per-instance model outputs, their confidence scores, and the corresponding ground-truth labels.

Calibration for M_1. We use dishes and all the available plate images from [18]. For each food type in an image, M_1 generates a ranked list of candidate tokens (labels) with log probabilities. These are exponentiated, scaled to percentages, and rounded to two decimal places to yield confidence scores, and the top token (token with the highest confidence score) is selected as the predicted label for that food type.

- If the top token matches the ground-truth label for the food type, we add an entry containing the top-5 tokens, their confidence scores, and the ground-truth label to M_1's calibration dataset.
- If the ground-truth label is not among the predictions, we substitute a label: either an unmatched food type prediction (if available) or, as a fallback, a matched one from the same image. The substituted token and its associated scores are stored alongside the ground-truth label.

This substitution procedure ensures that every target food type contributes an entry to the calibration dataset, which is necessary for computing reasonable reference intervals. This yields a confidence-based calibration dataset for M_1 with 96 samples.

Calibration for M_2. For the same set of images used in M_1, we iterate over all bounding boxes generated by M_2. Each bounding box is manually assessed to determine whether it is *successful* (accurately capturing a target food type) or *unsuccessful* (e.g., enclosing the wrong food type, empty regions, or background noise). For each bounding box, we record the confidence score produced by GroundingDINO. If the bounding box is deemed successful, its confidence score is added to the success list; if unsuccessful, its confidence score is added to the failure list. In this setup, the success scores play the role of the "top-choice" confidence values, while the failure scores serve as the counterpart to the "second-choice" values used in M_1 and M_3. This yields a calibration dataset of 76 samples for M_2 consisting of success (61 samples) and failure (15 samples) confidence distributions that are used to construct intervals analogous to those for the other modules.

Calibration for M_3. The identified (or substituted) label from M_1, together with the corresponding bounding box image from M_2 of that food type, is passed to M_3. To ensure reliable inputs, we restrict this step to bounding boxes deemed valid—those accurately capturing the food type without just empty regions and such. This module outputs candidate skill tokens and their log probabilities, which are processed into percentage confidence scores using the same procedure as M_1. Similar to the procedure in M_1, the top token is taken as the predicted skill. For each food type and bounding box image pair, we store the top 3 tokens, their confidence scores, and the ground-truth skill label in the calibration dataset for M_3. This yields a confidence-based calibration dataset for M_3, with 66 samples, analogous to that of M_1.

Calibration for M_4. Since RT-1 is deterministic, we introduce stochasticity into its predictions using Monte Carlo Dropout. Specifically, we enable dropout during inference and run the model in batches of 16 forward passes. This produces a distribution over the M_4 output space, from which we compute the variance. The variance then serves as our measure of the model's confidence in its prediction. We used the same bounding boxes as we used in M_3, but with the human annotated ground truth label for skills. This yields a confidence-based calibration dataset for M_4 with 66 samples.

Confidence Intervals. From each calibration dataset, we compute the mean (μ^*) and standard deviation (σ^*) of the top tokens' confidence scores. We do the same for the second top tokens, and get μ^+ and σ^+. Using these values, we define the following confidence intervals:

$$I_{\text{TopConf}} = [\mu^* - \sigma^*,\ \mu^* + \sigma^*], \quad I_{\text{SecondTopConf}} = [\mu^+ - \sigma^+,\ \mu^+ + \sigma^+].$$

These intervals capture the expected distribution of confidence scores for top and second-ranked predictions across all calibration instances.

C.2 Calibrated Confidence Scores

To obtain calibrated confidence scores for all the modules, we apply a rule that maps raw confidence values into a stable binary value. For a given confidence score c^* (corresponding to the top prediction for an instance), we check whether c^* falls outside the calibrated top-token interval or within the second-token interval:

$$Conf_{cal}(c^*) = \begin{cases} 0, & \text{if } (c^* \notin I_{\text{TopConf}}) \ \vee \ (c^* \in I_{\text{SecondTopConf}}), \\ 1, & \text{otherwise.} \end{cases}$$

Here, the function $Conf_{cal}(c^*)$ produces a binary calibrated confidence score: 0 when the prediction is considered low-confidence (uncertain), and 1 when the prediction is considered high-confidence (confident). These calibrated scores are later consumed by other components of the pipeline that determine whether or not to query the human for that particular module.

Grounding the decision in confidence intervals estimated from calibration data allows the system to identify both underconfident correct predictions and overconfident incorrect predictions, producing a more stable confidence signal than raw probabilities alone. By introducing calibrated confidence scores, we ensure that the system reasons over interpretable, statistically grounded intervals rather than noisy probability values, providing more reliable and consistent confidence estimates across all the modules.

C.3 Retrieval-Augmented Feedback for Error Recovery

To enable the system to recover from past errors and adapt over time, we integrate a retrieval-augmented generation (RAG) component into both the food type identification module (M_1) and the skill selection module (M_3). The goal of this component is to incorporate

human-provided feedback into a persistent store, allowing the pipeline to retrieve and reuse corrections when similar inputs are encountered in the future. This mechanism provides a means of continual learning from mistakes and reduces repeated queries to the human user.

Embedded Feedback Store. We implement an EmbeddedFeedbackStore that records feedback entries consisting of the plate image (for m_1) or bounding box image and food label (for m_3), together with the corrected output provided by the human. For m_1, embeddings are computed directly from the plate image using the CLIP vision encoder [31], resulting in a purely visual representation. For m_3, embeddings are computed jointly from the bounding box image of the target food item and its textual label using the CLIP vision–language model [31], capturing both visual and semantic context for skill selection. All embeddings are normalized and stored persistently as vectors alongside the corresponding ground-truth correction, which in our setup refers to the a correct food type label for M_1 and the correct skill for M_3. So whenever the system queries the human for feedback for a particular module, the corresponding input (image or bounding box image with label) together with its corrected output is added to the feedback store for future retrieval and reuse.

Retrieval. At inference time, when M_1 or M_3 produces a prediction, the feedback store computes the CLIP embedding for the current input and retrieves the most similar past entry using cosine similarity. If the best match exceeds a similarity threshold, the stored correction is reused. This correction is converted into a probability distribution: the retrieved food type label (or skill) is assigned a probability proportional to the similarity score, and the remaining probability mass is evenly distributed across the other candidate tokens. If no sufficiently similar correction is found, the pipeline defaults to the module's prediction.

This RAG-based mechanism allows the pipeline to learn incrementally from its mistakes. By leveraging similarity search, the system avoids repeating prior errors on similar inputs and reduces unnecessary human queries. This provides a lightweight but effective form of continual adaptation.

C.4 Query questions and feedback in user study

Four types of questions can be asked when executing the *Always Query* method and our selected method in the user study:

(1) The robot will ask the user for help by asking the following question: "Could you tell me a food item that is on the plate?" Users can respond by providing any valid food item that is present on the plate.
(2) The robot will ask the user for help by displaying the plate image on the tablet. Users can respond by tapping 2 opposite corners of the box on the screen to create a bounding box.
(3) The robot will ask the user for help by asking you the following question: "For the food item on the plate, what skill should I use?". Users can respond by telling the robot the skill that it should use.
(4) The robot will ask the user for help by displaying the bounding box image on the tablet. Users can respond by tapping 2 points in the following manner: for skewering, tap two points that define the longer edge of the food item; for scooping, tap the start and the end of the scoop; for twirling, tap two points around where the user thinks the fork should twirl the food item.

D Bite Acquisition Architecture with Module Heterogeneity

This architecture is similar to that described in Sec. C, with the following modifications:

- $\mathbf{M_1}$: The food detector is faulty, producing a random food label with probability 0.9 (which lies outside of the set of food items encountered in the "savory salad" and "Thanksgiving meal" plates). In this scenario, the food detector assigns a confidence of $c_1 = 0.65$. In the event of a success, M_1 produces the label with the highest confidence score ℓ^* as before, but with a confidence score $c_1 = \frac{c^*}{c_{max,M_1}}$, where c^* is the raw confidence score (Sec. C.2) and c_{max} is the maximum raw confidence score observed in the M_1 calibration set.
- $\mathbf{M_2}$: Similar to Sec. C, but with a confidence score $c_2 = \frac{c^*}{c_{max}}$, where c^* is the raw confidence score (Sec. C.2) and c_{max,M_2} is the maximum raw confidence score observed in the M_2 calibration set.
- $\mathbf{M_3}$: Similar to Sec. C, but with a confidence score $c_3 = \frac{c^*}{c_{max}}$, where c^* is the raw confidence score (Sec. C.2) and c_{max,M_3} is the maximum raw confidence score observed in the M_3 calibration set.
- $\mathbf{M_4}$: Unchanged compared to Sec. C.

E Workload Model Conditioning

To predict workload for each of the bite acquisition modules (Appendix C), we set the query type variables for the predictive workload model as follows [3]:

- M_1: d_t = easy, $resp_t$ = MCQ, $dist_t$ = "no distraction task"
- M_2: d_t = easy, $resp_t$ = BB, $dist_t$ = "no distraction task"
- M_3: d_t = easy, $resp_t$ = MCQ, $dist_t$ = "no distraction task"
- M_4: d_t = hard, $resp_t$ = BB, $dist_t$ = "no distraction task"

F Additional Synthetic Experiments: Systematic Ablations

F.1 Additional experimental setup details.

We set the GraphQuery hyperparameter $\epsilon = 1$, and we assume equal weight between querying and task reward ($w = 0.5$).

Independent variable settings considered:

- Number of modules N: [3, 5, 10, 15, 18, 25, 50, 75, 100]
- Graph redundancy structure G_M: [fully redundant (all-OR), fully dependent (all-AND), fully redundant followed by fully dependent (OR-then-AND), fully dependent followed by fully redundant (AND-then-OR)]
- Module confidences c_i: [(1.0, 0.1), (0.9, 0.2), (0.8, 0.3), (0.7, 0.4)], where (c_h, c_l) are the high confidence value and low confidence value, respectively, described in Sec. 6. We assume that 3 of the modules in the module graph G_M are assigned the low confidence value, and the rest are assigned the high confidence value.
- Query costs q_i: [0.08, 0.16, 0.32, 0.64]

Below, we vary each independent variable in isolation, fixing the other variables to the setting described in Sec. 6.

F.2 Number of modules N

Full results for varying the number of modules N across module selectors and querying algorithms are shown in Fig. 7. Key takeaways:

- No metrics (besides *Computation Time*) vary as a function of number of modules, except in different variable settings

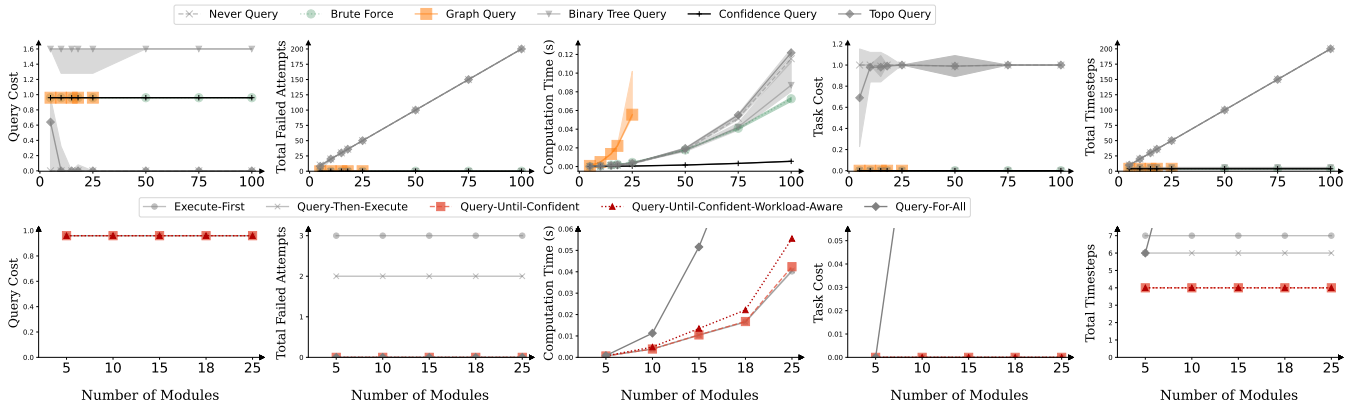

Figure 7: (top) Module selector comparison for varying the number of modules N in the module graph G_M, for the QUERY-UNTIL-CONFIDENT-WORKLOAD-AWARE querying algorithm. (bottom) Querying algorithm comparison for varying the number of modules N in the module graph G_M, for the GRAPHQUERY module selector. Plots show median, upper quartile, and lower quartile values across 100 trials (mean for Task Cost).

(e.g. closer confidence values) where numerical stability is a concern.
- When confidence values differ from $(1, 0.1)$, module selector performance degrades at larger graph sizes due to numerical underflow. We find that BINARYTREEQUERY is the most sensitive, followed by GRAPHQUERY, followed by BRUTEFORCE.

Recommendations. For module selectors, we recommend using either BRUTEFORCE, GRAPHQUERY, or CONFIDENCEQUERY, with the latter being the most scalable to increasing module graph size. For querying algorithms, we recommend QUERY-UNTIL-CONFIDENT or QUERY-UNTIL-CONFIDENT-WORKLOAD-AWARE as they minimize *Total Failed Attempts* and *Total Timesteps*.

F.3 Graph structure G_M

Full results for varying the graph structure G_M across module selectors and querying algorithms are shown in Fig. 8. Key takeaways:

F.3.1 *Module selectors.*

- *Query Cost.* For the variable setting shown in Fig. 8, we find that *Query Cost* is insensitive to the redundancy structure. Across other variable settings, we generally observe that the query cost is lower for the all-OR redundancy structure, compared to the other redundancy structures, because fewer queries are needed for the overall system output to be correct. Additionally, the query cost for the OR-then-AND structure is slightly higher than that for the AND-then-OR redundancy structure (because overall system success is more likely when the final module is an OR module).
- *Total Failed Attempts.* For the variable setting shown in Fig. 8, we find that *Total Failed Attempts* is insensitive to the redundancy structure (except for NEVERQUERY). Across other variable settings, we observe that *Total Failed Attempts* is lower for the all-OR redundancy structure. We additionally observe that the relative ordering of module selectors does not generally depend on the redundancy structure, and proceeds roughly as follows (in descending order): NEVERQUERY, BINARYTREEQUERY, BRUTEFORCE, followed by GRAPHQUERY.
- *Computation Time.* We observe that *Computation Time* is lower for the all-OR redundancy structure. GRAPHQUERY tends to have a higher *Computation Time* than the other module selectors (regardless of the redundancy structure) due to the computational complexity of parallel graph creation + parallel shortest-path searches.
- *Task Cost.* For the variable setting shown in Fig. 8, we find that *Task Cost* is insensitive to the redundancy structure (except for NEVERQUERY). In other settings, NEVERQUERY and BINARYTREEQUERY only achieve complete success for the all-OR architecture, and have no success for the other redundancies.
- *Total Timesteps.* For the variable setting shown in Fig. 8, we find that *Total Timesteps* is insensitive to the redundancy structure (except for NEVERQUERY), with BINARYTREEQUERY having a slightly higher value across-the-board compared to the other module selectors.

F.3.2 *Querying algorithms.*

- *Query Cost.* For the variable setting shown in Fig. 8, generally lower for most querying algorithms in all-OR redundancy structure, compared to other structures. In other low query cost settings: (1) higher for QUERY-UNTIL-CONFIDENT-WORKLOAD-AWARE compared to other methods, (2) higher in the hybrid structures (compared to the pure structures) across-the-board.
- *Total Failed Attempts.* Lower for all querying algorithms in all-OR redundancy structure, compared to other structures. We note the following rough ordering across querying algorithms, regardless of graph structure, in descending order: EXECUTE-FIRST > QUERY-THEN-EXECUTE > QUERY-UNTIL-CONFIDENT ~ QUERY-UNTIL-CONFIDENT-WORKLOAD-AWARE. We also notice the following ordering across graph structures, regardless of querying algorithm, in descending order: OR-then-AND > all-AND > AND-then-OR > all-OR.
- *Computation Time.* Lower for all querying algorithms in all-OR redundancy structure, compared to other structures. We note that QUERY-UNTIL-CONFIDENT-WORKLOAD-AWARE

has slightly higher computation time than other querying algorithms, independent of graph structure.
- *Task Cost.* Fairly consistent across graph structures. In other settings, Query-Until-Confident-Workload-Aware tends to have the highest success rate across all querying algorithms, as it is able to recover in some settings where other querying algorithms fail (e.g. AND-then-OR, and OR-then-AND structures).
- *Total Timesteps.* Lower for most querying algorithms in all-OR redundancy structure, compared to other structures. WE note the following roughly consistent ordering across querying algorithms, regardless of graph structure, in descending order: Execute-First > Query-then-Execute > Query-until-Confident ~ Query-until-Confident-Workload-Aware, consistent with the *Total Failed Attempts* trend.

Recommendations. For module selectors, we recommend using either BruteForce, GraphQuery, or ConfidenceQuery, as all 3 have the best performance across redundancy structures. For querying algorithms, we recommend Query-until-Confident or Query-until-Confident-Workload-Aware perform best across the metrics (for all redundancies except all-OR, where Execute-First and Query-and-Execute outperform the other querying algorithms in *Query Cost* and *Total Timesteps*).

F.4 Confidence values c_i

Full results for varying the confidence values c_i across module selectors and querying algorithms are shown in Fig. 9. Key takeaways:

F.4.1 Module selectors.

- *Query Cost.* Generally higher when confidences are closer together. Regardless of confidence level, generally follows the following pattern in descending order: BinaryTreeQuery > GraphQuery ~ BruteForce > Never Query.
- *Total Failed Attempts.* Generally higher when confidences are closer together. Regardless of confidence level, generally follows the following pattern in descending order: NeverQuery, GraphQuery ~ BruteForce, BinaryTreeQuery.
- *Computation Time.* Generally higher when confidences are closer together. GraphQuery has much higher Computation Time, followed by BruteForce, BinaryTreeQuery and NeverQuery.
- *Task Cost.* All module selectors (except NeverQuery) tend to increase from 0.0 to 1.0 with less-separated confidences. NeverQuery always has a value of 1.0, except in the all-OR setting (not shown in Fig. 9).
- *Total Timesteps.* Generally higher when confidences are closer together (for all module selectors except NeverQuery).

F.4.2 Querying algorithms.

- *Query Cost.* Generally higher when confidences are closer together. For low query cost settings (including Fig. 9), Query-until-Confident-Workload-Aware has the highest *Query Cost* compared to other querying algorithms.
- *Total Failed Attempts.* Generally higher across all querying algorithms when confidences are closer together. Lower for Query-until-Confident than Execute-First and Query-then-Execute.
- *Computation Time.* Generally higher when confidences are closer together. Slightly higher for Query-until-Confident-Workload-Aware compared to the other querying algorithms.
- *Task Cost.* Generally higher when confidences are closer together. Lower for Query-until-Confident than Execute-First and Query-then-Execute.
- *Total Timesteps.* Generally higher when confidences are closer together. Regardless of confidence score, *Total Timesteps* is generally lower for Query-until-Confident compared to Execute-First and Query-then-Execute.

Recommendations. For module selectors, we recommend using either BruteForce or GraphQuery, as they are the most competitive across metrics regardless of the confidence level. For querying algorithms, we recommend Query-until-Confident (unless in the all-OR setting, in which case Execute-First or Query-then-Execute are the best at minimizing *Total Timesteps*).

F.5 Query Cost q_i

Full results for varying the confidence values c_i across module selectors and querying algorithms are shown in Fig. 10. Key takeaways:

F.5.1 Module selectors.

- *Query Cost.* Higher *Query Cost* across-the-board as we increase the cost of querying, which is expected. BinaryTreeQuery generally has the highest *Query Cost* (regardless of module query cost), while BruteForce and GraphQuery are lower.
- *Total Failed Attempts.* Relatively invariant to the query cost level.
- *Computation Time.* Generally highest for GraphQuery, regardless of the query cost level. In ascending order for the other module selectors, generally see NeverQuery, then BruteForce, then BinaryTreeQuery.
- *Task Cost.* Generally consistent for all methods, regardless of query cost level. For higher query cost settings and confidence settings with less separation (not shown in Fig. 10), we do observe consistent a *Task Cost* value of 0.0 across all settings for BinaryTreeQuery and NeverQuery.
- *Total Timesteps.* Values are also generally invariant as a function of query cost, with BinaryTreeQuery having a slightly higher value compared to BruteForce and GraphQuery.

F.5.2 Querying algorithms.

- *Query Cost. Query Cost* generally increases as query cost increases, as expected.
- *Total Failed Attempts.* Typical trend is Execute-First > Query-then-Execute > Query-until-Confident ~ Query-until-Confident-Workload-Aware, regardless of the query cost setting. In some settings, *Total Failed Attempts* increases slightly for Query-until-Confident-Workload-Aware as a function of query cost.
- *Computation Time.* Generally comparable across querying algorithms, invariant to query cost setting. In some scenarios (e.g. in Fig. 10), Query-until-Confident-Workload-Aware has a higher computation time than the other querying algorithms.
- *Task Cost.* Generally comparable across querying algorithms (0.0), invariant to query cost setting. Sometimes observe degradation to 1.0 (in module

Figure 8: Varying the graph redundancy structure for fixed querying algorithm (Query-until-confident-workload-aware; top) and fixed module selector (GraphQuery; bottom). Plots show median values across 100 trials (mean for Task Cost).

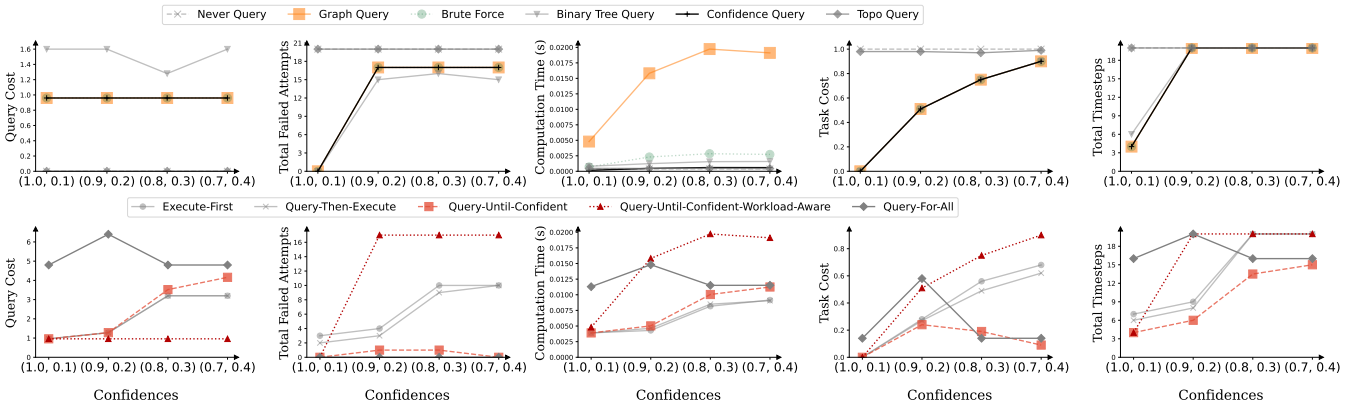

Figure 9: Varying the confidence levels for fixed querying algorithm (Query-until-confident-workload-aware; top) and fixed module selector (GraphQuery; bottom). Plots show median values across 100 trials (mean for Task Cost).

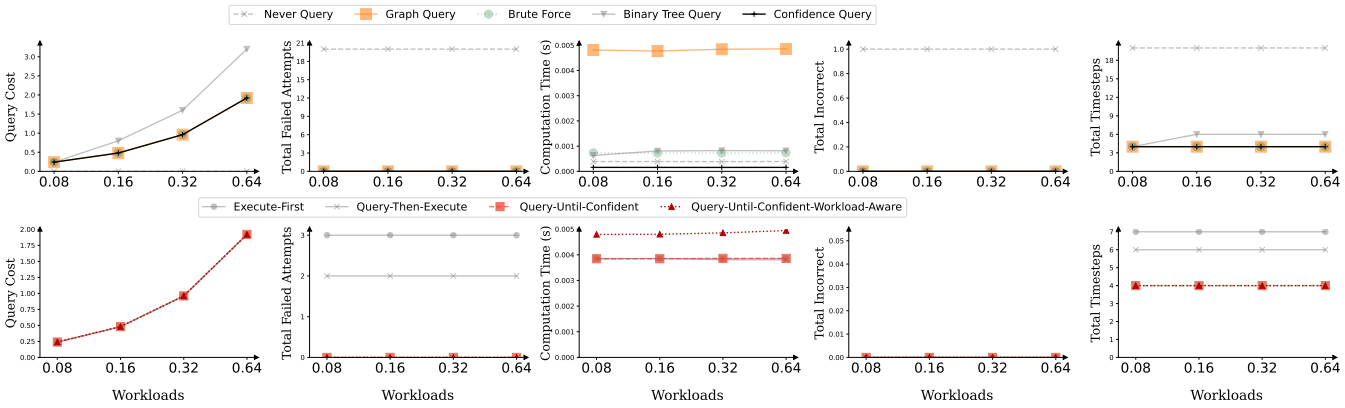

Figure 10: Varying the module query costs for fixed querying algorithm (Query-until-confident-workload-aware; top) and fixed module selector (GraphQuery; bottom). Plots show median values across 100 trials (mean for Task Cost).

settings with smaller separation in confidences, not shown in Fig. 10) for all methods besides Query-until-Confident-Workload-Aware.

- *Total Timesteps.* Generally invariant to query cost setting. Regardless of confidence score, *Total Timesteps* is generally lower for Query-until-Confident-Workload-Aware and

Query-until-Confident compared to Execute-First and Query-then-Execute (except for the all-OR redundancy structure, where the trend is flipped).

Recommendations. For module selectors, we recommend using either BruteForce, GraphQuery, or ConfidenceQuery as they are the most competitive across metrics regardless of the module query cost. For querying algorithms, we recommend Query-until-Confident-Workload-Aware or Query-until-Confident (Query-until-Confident-Workload-Aware if minimizing *Query Cost* and *Total Failed Attempts* is most important; Query-until-Confident if minimizing *Computation Time* is the most important).

F.6 Additional Ablation: Expert Confidence

We consider an ablation over an additional user variable: the expert confidence value (c_{expert}, introduced in Sec. 3), to understand how imperfect human feedback affects the performance of the module selectors and querying algorithms. In our simulation, the expert confidence c_{expert} represents the probability that a module query produces the correct logical value for that module. We consider the following settings for c_{expert}: $[1, 0.8, 0.6, 0.4]$, which range from a perfect expert (considered in Sec. 6) to a less confident expert. Additionally, we adapt the module selectors to assign a confidence of c_{expert} to modules that have already been queried.

F.6.1 Module selectors. We find that performance degrades for all module selectors as c_{expert} decreases (Fig. 11), with *Task Cost* degrading to 0 at the lowest confidence setting ($c_{\text{expert}} = 0.4$). For sufficiently low expert confidence, the module selectors cannot determine whether a module has received the correct feedback, leading to redundant querying of the same (upstream) module. Across the module selectors, we find that Brute Force, GraphQuery, and ConfidenceQuery perform best in the high expert confidence regime. We also find that BinaryTreeQuery tends to under-query regardless of expert confidence value, leading to high *Task Cost*.

F.6.2 Querying algorithms. We again find that all querying algorithms degrade in performance as c_{expert} decreases (Fig. 11). For higher expert confidence values, we find that the two Query-Until-Confident variants are the best in *Total Timesteps* and *Total Failed Attempts*. For lower expert confidence values, we note that Query-Until-Confident tends to over-query, because the proxy task objective never becomes high enough to stop querying, and Query-Until-Confident-Workload-Aware tends to under-query, as the confidence gain due to querying is insufficient to overcome the query cost.

Recommendations. For intermediate-to-high expert confidence values, we would recommend either the BruteForce, GraphQuery, or ConfidenceQuery module selectors, as they achieve the lowest *Task Cost* with low *Total Timesteps*. Additionally, we would recommend the two Query-Until-Confident querying algorithms if minimizing *Total Timesteps* is the most important, and the Execute-First and Query-then-Execute querying algorithms in other scenarios.

G Additional Synthetic Experiments: Module Heterogeneity

Fig. 12 compares the GraphQuery vs ConfidenceQuery module selectors for additional values of the module confidences and query cost variance β (with fixed querying algorithm Query-Until-Confident-Workload-Aware). We find that GraphQuery performs at least as well as ConfidenceQuery across these experimental settings.

H User Study Metrics

H.1 Subjective metrics

We asked the participants the following questions in terms of Mental/Physical Demand, Effort, Subjective Success, and Satisfaction, all on a Likert scale from 1-5:

(1) **Mental/Physical Demand**: For the last method, how mentally/physically demanding was it for the robot to query you?
(2) **Effort**: For the last method, how hard did you have to work to make the robot pick up food items?
(3) **Subjective Success**: For the last method, how successful was the robot in picking up food items?
(4) **Satisfaction**: For the last method, how satisfied are you with how the robot balanced between trying to pickup independently when possible and asking for help when required?

H.2 Objective metrics

We define the following 3 objective metrics:

(1) **Mean Queries Per Plate**: $\frac{1}{B}\sum_{b=1}^{B}\sum_{t=1}^{T_b}\mathbf{1}\{\text{queried at } t\}$, where B is the number of bites per plate and T_b is the number of timesteps needed for bite b.
(2) **Mean Executions Per Plate**: $\frac{1}{B}\sum_{b=1}^{B}\sum_{t=1}^{T_b}\mathbf{1}\{\text{executed at } t\}$, where B and T_b are defined above.
(3) **Mean Successful Bites Per Plate**: $\frac{1}{B}\sum_{b=1}^{B}\mathbf{1}\{\text{b succeeded}\}$, where B is defined above and we define bite b to have succeeded if the robot acquired the bite after T_b timesteps.

I Full Acknowledgements

This work was partly funded by NSF CCF 2312774 and NSF OAC-2311521, a LinkedIn Research Award, NSF IIS-244213, NSF IIS #2132846, CAREER #2238792, a PCCW Affinito-Stewart Award, and by an AI2050 Early Career Fellowship program at Schmidt Sciences. Research reported in this publication was additionally supported by the Eunice Kennedy Shriver National Institute Of Child Health & Human Development of the National Institutes of Health and the Office of the Director of the National Institutes of Health under Award Number T32HD113301. The content is solely the responsibility of the authors and does not necessarily represent the official views of the National Institutes of Health.

The authors would like to additionally thank Yuanchen Ju and Zhanxin Wu for helping with figure creation, and all of the participants in our two in-lab user studies, as well as the two participants in our in-home user study.

Received 2025-09-30; accepted 2025-12-23

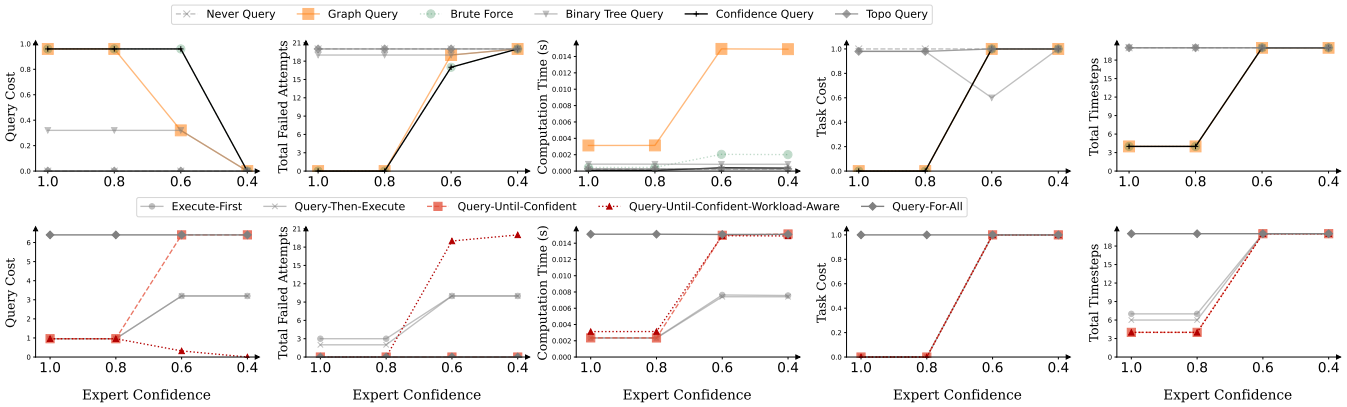

Figure 11: Varying the expert query confidence for fixed querying algorithm (QUERY-UNTIL-CONFIDENT-WORKLOAD-AWARE; top) and fixed module selector (GRAPH-QUERY; bottom). Plots show median values across 100 trials (mean for Task Cost).

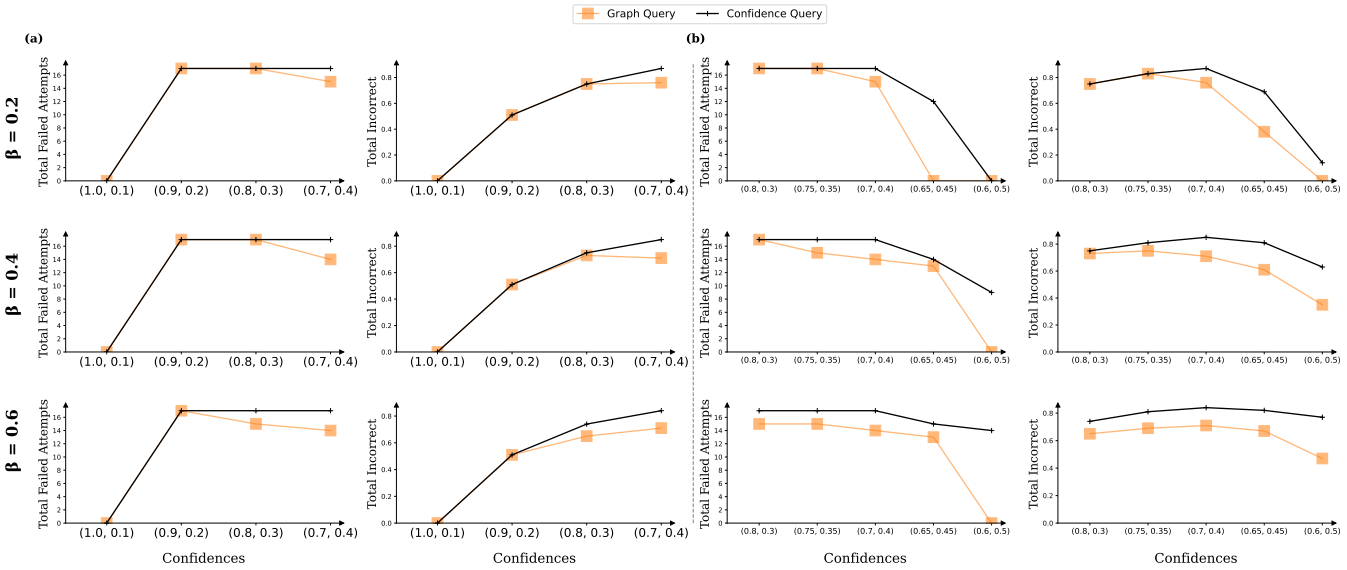

Figure 12: Comparison between GRAPHQUERY and CONFIDENCEQUERY, as a function of varying module confidences and query cost uniform noise β (for fixed querying algorithm: QUERY-UNTIL-CONFIDENT-WORKLOAD-AWARE), both in less overlapping (left) and more overlapping (right) confidence regimes. Plots show median values across 100 trials (mean for Task Cost).