From Code to Action: Hierarchical Learning of Diffusion-VLM Policies

Markus Peschl Qualcomm AI Research*
Pietro Mazzaglia Qualcomm AI Research
Daniel Dijkman Qualcomm AI Research
{mpeschl, pmazzagl, ddijkman}@qti.qualcomm.com

Abstract: Imitation learning for robotic manipulation often suffers from limited generalization and data scarcity, especially in complex, long-horizon tasks. In this work, we introduce a hierarchical framework that leverages code-generating vision-language models (VLMs) in combination with low-level diffusion policies to effectively imitate and generalize robotic behavior. Our key insight is to treat open-source robotic APIs not only as execution interfaces but also as sources of structured supervision: the associated subtask functions - when exposed - can serve as modular, semantically meaningful labels. We train a VLM to decompose task descriptions into executable subroutines, which are then grounded through a diffusion policy trained to imitate the corresponding robot behavior. To handle the non-Markovian nature of both code execution and certain real-world tasks, such as object swapping, our architecture incorporates a memory mechanism that maintains subtask context across time. We find that this design enables interpretable policy decomposition, improves generalization when compared to flat policies and enables separate evaluation of high-level planning and low-level control.

1 Introduction

The field of robotics has increasingly embraced imitation learning and the expansion of data collection as pivotal research avenues, inspired by the recent successes of generative models in language and vision domains [1, 2, 3, 4]. Unfortunately, however, the challenge of obtaining high-quality and diverse data necessary for training robots to perform a wide array of tasks remains a problem due to the need for accurate language annotations and corresponding expert demonstrations [5]. On the other hand, many robotics tasks share a common trait of compositionality, which is akin to functional programming: Sophisticated programs may appear to exhibit highly complex behavior that is difficult to imitate, but they are usually compositions of simpler functions that are easy to understand. Similarly, navigating and manipulating objects can result in long-horizon, complex patterns that, when broken down into simple skills, become easy to learn. Once learned, skills can then be dynamically composed to potentially achieve greater adaptability and generalize to new tasks.

This idea is not novel; the robotics community has extensively studied pick-and-place tasks because they are fundamental building blocks for interacting with the world [6]. Nonetheless, learning atomic skills and composing them into complex behaviors is challenging for a variety of reasons. Firstly, one needs to either rely on unsupervised learning to decompose long-horizon tasks, or assume access to labeled demonstrations for each subtask, which can be costly to obtain. Secondly, simply having access to a skill library is not sufficient when dealing with high level instructions, as they too first need to be translated into skills, which is exacerbated by the difficulty of long-horizon planning [7, 8].

To address the former, this paper builds on the insight that open-source robot control APIs can be a valuable source of data collection, as they not only provide expert demonstrations, but also come with annotations in the form of a code trace of their action. This code naturally exhibits a hierarchy of complexity and compositions of simple functions, making it well-suited for automating

* Qualcomm AI Research is an initiative of Qualcomm Technologies, Inc.

9th Conference on Robot Learning (CoRL 2025), Seoul, Korea.

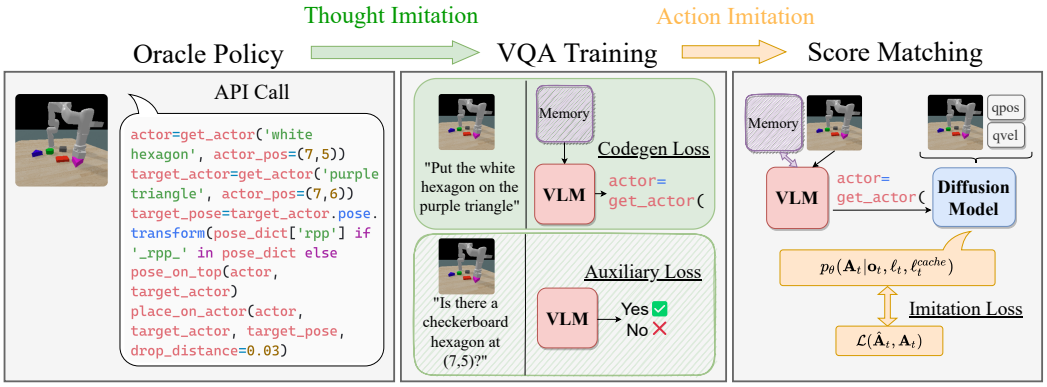

Figure 1: An illustration of our hierarchical learning approach combining thought imitation and action imitation. An oracle policy consisting of Python API calls collects demonstration data including corresponding code snippets per executed action. During the visual-question-answering (VQA) stage, a VLM is trained on the oracle demonstrations to generate the underlying API code (*codegen loss*) as well as recognize objects in the scene (*auxiliary loss*). Finally, a diffusion model, conditioned on the generated code, is trained to imitate low-level actions of the oracle.

the collection of sub-task labels. Unlike natural language, which tends to be under-specified on the end state of an instruction [9], these sub-task code labels are precise and unambiguous, making them ideal for robust concatenation. In order to utilize code as instructions for an end-to-end imitation learning system, we propose a hierarchical framework involving a code **generating** vision-language model (VLM) trained to imitate the language descriptions of API policies and a code **guided** low-level policy based on diffusion models [10] learning to dynamically map code to actions. Our training scheme first trains a VLM to generate API calls from successful demonstrations of an oracle policy. Subsequently, we distill the low-level action part of the oracle policy into a custom language-conditioned diffusion policy (DP) while conditioning on VLM generated code. This ensures that any generated code trace during training mimics those observed during inference, where both models operate simultaneously. We find that this approach effectively mitigates distribution shift and improves generalization compared to a policy that relies solely on high-level task descriptions. Furthermore, by incorporating a memory mechanism into both the high- and low-level policies, we demonstrate that our model can handle non-Markovian tasks, as well as the inherently stateful nature of oracle policy code, which requires memory to function correctly.

This work serves two purposes. Firstly, we compare the performance of diffusion policy under natural language conditioning with verifiably correct text conditioning such as executable code. Secondly, we present a method to distill existing scripted robot policies into learned policies. The applied use-case of this is to distill a classical robotic setup which relies on many sensors, precise calibration and scripted policies into an AI-based system, which merely relies on cameras and proprioception. Our **contributions** can be summarized as follows:

- We introduce a novel VLM training scheme for code generation of robotic control primitives, including auxiliary losses and a memory buffer of past actions to tackle state tracking.
- We present a hierarchical framework for training code-conditioned diffusion models on VLM-labeled demonstration data, as well as a custom encoder based on learned attention pooling layers for processing multimodal conditioning information.
- We find that by accurately composing sub-tasks at inference time, our hierarchical policy generalizes better than flat variants on various tasks of the ClevrSkills benchmark.

2 Related Work

Language-Guided Imitation Learning. Modern imitation learning (IL) benchmarks typically require learning a single language-conditioned policy for a variety of tasks [11, 12, 13]. Diffusion policies [10] offer a strong IL baseline and have since been adapted to tackle this by adding pretrained

language encoders [2, 14, 15]. Similarly, vision-language-action (VLA) models have been proposed, processing language and vision instructions through a more close integration of pretrained foundation models into the policy. Architectural choices commonly range from using diffusion heads [16, 17, 18] and flow matching [19] to directly predicting action tokens through language [1, 20].

Hierarchical Policies. Hierarchical models aim to generalize to new tasks by factorizing their action distribution into high and low-level predictions with varying choices of intermediate representations. Hierarchical diffusion models [21, 22] split action generation into key-step prediction and inpainting steps, while VLM-based models have been used to predict a large variety of representations [23, 24, 25, 26, 27, 28] as well as natural language [29, 30, 31]. More closely to our work, several works have explored using code to represent policies [32, 33, 34, 35, 36, 37]. Typically, a pretrained (vision-)language model is leveraged to generate code corresponding to a multi-step plan, given a natural language description of a task. The focus hereby mostly lies on improving the high-level planning capabilities, whereas the low-level policy is obtained by directly executing robot API code. In our paper, code serves merely as an intermediate representation, with the goal of learning both high and low-level policies entirely through neural networks.

Akin to our paper, recent works such as HAMSTER [26], HiRobot [30], Gr00t N1[38] and DexVLA [29] fully realize high and low-level policies within conditional generative models. Our research diverges by focusing on the generalization performance in an idealized framework, where we obtain perfect access to subtask labels by generating code-annotated demonstration data using robot APIs. This approach precisely specifies high-level thoughts for each time step, unlike 2D path representations [26], natural language [30, 29] or latent thoughts [38]. As a result, we can not only isolate the success rate of the high-level planner from the success rate of the low-level policy, but also automate data collection by directly letting the high-level planner act in the environment. The latter advantage has already been realized in the case of training non-hierarchical policies [2, 39, 40].

3 Preliminaries

Imitation Learning Language conditioned imitation learning aims to learn a policy $\pi_\theta : \mathcal{O} \times \mathcal{L} \rightarrow \Delta\mathcal{A}$ mapping observations $\mathbf{o}_t \in \mathcal{O}$ and task descriptions $\ell_t \in \mathcal{L}$ to a probability distribution over actions $\mathbf{A}_t \in \mathcal{A}$. More specifically, we assume to always predict a sequence of actions (*action chunk*), i.e. $\mathbf{A}_t = [\mathbf{a}_t, \mathbf{a}_{t+1}, ..., \mathbf{a}_{t+H}] \in \mathbb{R}^{H \times D_a}$, where H is the prediction horizon [41, 10] and $\mathbf{o}_t = [\mathbf{X}_t^b, \mathbf{X}_t^w, \mathbf{s}_t]$ consists of image inputs $\mathbf{X}_t \in \mathbb{R}^{H' \times W \times C}$ corresponding to base and wrist cameras as well as low-dimensional proprioception features $\mathbf{s}_t \in \mathbb{R}^{D_s}$.

Generative Models for Robotics. Vision-language models (VLMs) are versatile models pretrained on large-scale, multimodal internet data [42]. For our purposes, we assume VLMs to model a distribution $p_\phi(\ell^{out} | \mathbf{X}^\mathbf{b}, \ell^{in})$ trained using next-token prediction. Given a single (base camera) image $\mathbf{X}^b$ and a task description ℓ^{in}, a language suffix ℓ^{out} is predicted autoregressively $p_\phi(\ell^{\text{out}} \mid \mathbf{X}^b, \ell^{\text{in}}) = \prod_{t=1}^{T} p_\phi(l_t \mid l_1, \ldots, l_{t-1}, \mathbf{X}^b, \ell^{\text{in}})$ via a decoder-only Transformer architecture. Moreover, our work utilizes diffusion models for policy learning. For a primer on the latter, refer to Appendix D.

4 From Code to Action

To optimally facilitate thought and action imitation respectively, our training pipeline splits data generation and training into two stages, which we visualize in Figure 1. Firstly, we train a code-generating VLM on an oracle dataset generated using API calls from hard coded policies, as described in Section 4.1. Using a visual-question-answering (VQA) format, the VLM is trained to predict the current action in the form of API code, given an image and a task prompt. We also introduce auxiliary losses for bounding box predictions and a memory mechanism for state tracking, which we elaborate on in Section 4.2. Secondly, we train a conditional diffusion model to predict low-level actions, given code instructions generated from the VLM, which we outline in Section 4.3.

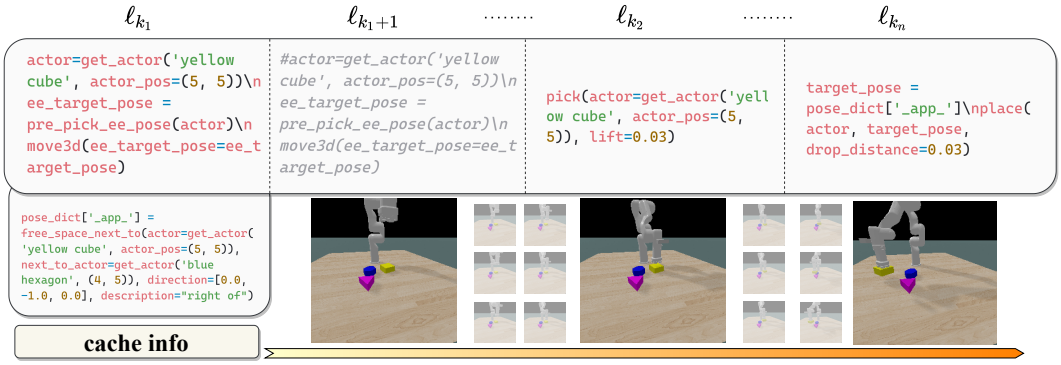

Figure 2: Illustration of a code trace on the task *PlaceNextTo*. Key-steps ℓ_{k_i} form unique subtask labels, while in-between steps correspond to the most recent key-step. To condition the low level policy on historical information, we extract commands that write to an internal dictionary `pose_dict` and save them to a cumulative cache (*cache info*).

4.1 Data Generation

To obtain the oracle dataset $\mathcal{D}_{oracle} = \{\tau_i\}_{i=1}^{N}$, we utilize the ClevrSkills environment [13], which comes with a variety of open-source scripted policies (called *solvers*) for each task. Since the policies are not perfect, we filter out any unsuccessful trajectories. Each trajectory consists of a sequence of observations, actions and language instructions $\tau = (\mathbf{o}_1, \mathbf{a}_1, \ell_1, \mathbf{o}_2, \mathbf{a}_2, \ell_2, \dots)$, where ℓ_t corresponds to the API code that was executed at time t to produce action $\mathbf{a}_t$.

The policies (and hence the annotations) that ClevrSkills provides are hierarchical. For example, there is a `pick_move3d_place` policy, which internally uses `pick`, `move3d` and `place` policies, and utility functions such as `get_actor`. We chose to use the annotations at their most fine-grained level to provide detailed conditioning to the diffusion policy. For more details we refer to API in Appendix B. We pre-process the API calls $\ell_t \in \tau$ into key-step instructions corresponding to the first time an API call is executed, and comment out code using the # symbol in any subsequent time-step with the same API call. We visualize one example of a code trace corresponding to a demonstration on the *PlaceNextTo* task in Figure 2.

4.2 Code Generation VLM

Architecture. We build on the LLaVa framework [42], employing a Phi-3 language model backbone [43] due to its efficient inference. Our objective is to construct a high-level VLM that maps image-valued inputs $\mathbf{X}^b$ and a natural language prompt ℓ^{in}, which specifies the overall task, to an API call that, when executed, would lead to the completion of the current subtask. In practice, however, mapping cannot rely solely on the current observation, as most API-based policies operate in a non-Markovian regime - retaining state information such as previous object poses or task-relevant events across timesteps to ensure correct behavior in the future.

To effectively imitate such non-Markovian policies, we augment our VLM with a lightweight memory mechanism. Specifically, we implement a caching strategy that maintains a memory buffer m_t, which accumulates generated API calls over time. At each timestep t, the model appends the most recent API call to the buffer only if it corresponds to a key-step. This memory is then incorporated into future predictions, enabling coherent, temporally-aware code generation. We'll provide a more detailed formalization of how and when this memory mechanism is used in the following paragraphs.

Training scheme. Our VLM is trained on two different objectives: Code generation and auxiliary losses such as bounding box prediction and object recognition. For code generation, the general prompt structure combines memory information m_t, the task prompt ℓ^{in} and an optional key-step

request ℓ^{key}. The goal is to minimize the loss

$$\mathcal{L}_{code}(\phi) = -\mathbb{E}_{t\sim U([T])}\left[p_\phi(\ell_t|\mathbf{X}_t^b, \ell^{in}, m_{t-1})\right] - \mathbb{E}_{i\sim U([n])}\left[p_\phi(\ell_{k_i}|\mathbf{X}_{k_i}^b, \ell^{in}, \ell^{key}, m_{k_{i-1}})\right],$$

where $m_j := (\ell_{k_1}, \ldots, \ell_{\max\{k_i \leq j\}})$ is the memory buffer of previous key-step instructions and ℓ^{key} is an additional prompt (*Please give a keystep reply*). Both m_j and ℓ^{key} are processed by the VLM by appending them to the instruction ℓ^{in}. For the auxiliary losses that facilitate localization, we refer to Appendix C.1.

Efficient Inference One of the main motivations behind splitting $\mathcal{L}_{code}$ into a key-step and an intermediate instruction objective is to enable two modes of inference.

- **VLM + Oracle Policy** Using the key-step mode $p_\phi(\ell_{k_i}|\mathbf{X}_{k_i}^b, \ell^{in}, \ell^{key}, m_{k_{i-1}})$ is useful for enabling tool usage [44]. In our case, the tools are Python calls to invoke the oracle policies. Although a perfectly executed code trace does not result in a 100% success rate due to failure cases of the oracle policies, using the VLM in this mode gives us a robust policy, as well as a close-to-optimal metric for measuring performance of the high-level policy.
- **VLM + Diffusion Policy** The intermediate prediction $\hat{\ell}_t \sim p_\phi(\cdot|\mathbf{X}_t^b, \ell^{in}, m_{t-1})$ is used when using the code outputs merely as conditioning information for a learned low level policy. In this mode, we query the VLM at each timestep. To update m_t, we verify if $\hat{\ell}_t$ is a key-step request by checking for non commented-out code blocks. If this is not the case, m_t is not updated. In practice, we also use this mechanism for speeding up inference: When the first $l = 20$ characters of $\hat{\ell}_t$ match a commented version of the last key-step in m_{t-1}, we truncate the auto-regressive generation through early stopping and use the last key-step instead. Although l is a hyperparameter, we found it to only have a minimal impact on performance.

4.3 Hierarchical Diffusion Policy

The low level part of our hierarchical model consists of a custom language-conditioned diffusion policy architecture [10] and is visualized in Figure 5. Instead of directly feeding the history m_t of memory into the policy at each timestep, we opt to preprocess m_t into a single prompt ℓ^{cache} which contains information about stored variables that are relevant for future frames. For details of this preprocessing step as well as further architecture modifications, we refer the reader to Appendix D.

Initially, we found that naively training the low level policy directly on $\mathcal{D}_{oracle}$ leads to unstable performance. For this reason, we choose to modify the input trajectories at training time in a way that better matches the distribution encountered at inference time. Specifically, we replace each oracle code instruction ℓ_t with generated code instructions $\hat{\ell}_t \sim p_\phi(\cdot|\mathbf{X}_t^b, \ell^{in}, m_{t-1})$. As we will show in section 5.2, training the low level policy on these generated high-level instructions leads to substantial improvements in overall performance.

Overall, the hierarchical policy is instantiated as

$$p_{\theta,\phi}(\mathbf{A}_t|\mathbf{o}_t, \ell) = p_\theta(\mathbf{A}_t|\mathbf{o}_t, \ell_t, \ell_t^{cache})p_\phi(\ell_t, \ell_t^{cache}|\mathbf{X}_t^b, \ell, m_{t-1}),$$

where $\mathbf{A}_t$ is an action chunk of size 8. We choose to run p_ϕ at every step for two reasons. Firstly, the memory m_t needs to be updated alongside the execution of $\mathbf{A}_t$. Secondly, we found that blind execution of an action chunk can lead to detrimental performance when the action chunk spans multiple subtasks. To mitigate this, we can stop execution of $\mathbf{A}_t$ whenever a generated instruction $\ell_k, k \in \{t, \ldots, t+8\}$ contains a key-step command and regenerate a new chunk $\mathbf{A}_k$.

5 Experiments

We evaluate our method on various tasks of the ClevrSkills benchmark [13]. Aside from open-source oracle solvers, which allow training our code generating VLM, ClevrSkills is built to benchmark compositional reasoning and generalization to higher level tasks. In section 5.1 we provide our main results where we aim to train a single hierarchical multitask policy and compare it to a flat baseline as

Table 1: A performance comparison per task and low level training dataset. Mean success rates (%) and standard deviations are shown, computed over 64 seeds with 2 runs each.

Task	Task Prompt Only			VLM Generated Code			VLM+Oracle
	L0	L1	L0+L1	L0	L1	L0+L1	
PlaceNextTo	21.9±2.2	7.0±2.3	25.8±2.3	55.4±0.8	10.2±2.4	**66.1**±1.1	83.1±3.7
PlaceOnTop	14.1±2.2	0.0±0.0	17.2±1.7	29.0±0.9	31.3±6.3	**53.1**±4.2	75.0±1.0
Topple	93.0±1.1	9.3±0.0	94.5±0.8	94.5±0.8	9.4±9.4	**99.0**±1.0	100±0.0
Push	74.2±5.6	2.3±0.8	69.5±3.9	87.4±1.6	0.0±0.0	85.9±0.0	91.5±1.5
SingleStack	0.0±0.0	14.1±3.1	15.6±3.1	0.0±0.0	22.6±0.8	**43.9**±4.7	81.5±1.5
StackTopple	0.0±0.0	0.0±0.0	0.0±0.0	0.0±0.0	17.2±1.6	**38.9**±3.1	71.0±1.0
PushToTarget	2.3±1.1	30.4±5.5	8.6±2.3	0.8±0.8	87.5±1.6	82.5±7.5	75.3±0.3
Unseen in L0+L1							
Pick	35.2±5.6	0.0±0.0	35.9±1.6	59.0±3.6	67.1±0.0	**78.0**±3.0	87.0±1.0
ReverseStack	0.0±0.0	0.0±0.0	0.0±0.0	0.0±0.0	21.8±4.7	**41.4**±2.4	80.0±1.0
NovelNoun	14.8±1.1	0.0±0.0	14.8 ±1.1	26.5±1.5	26.5±7.5	**50.7**±0.8	63.4±6.6
Average	25.55	6.31	28.19	35.24	29.36	**63.95**	80.78

well as an oracle-based baseline, whereas in section 5.2 we analyze design choices such as action chunking regeneration, dataset generation strategies and data scaling properties of our method.

5.1 Multitask Benchmark

Setup. To evaluate the performance of our hierarchical policy, we are not only interested in assessing general success rates per task, but also in the quantification of generalization through composing simpler subtasks to achieve new behaviors. For this purpose, we slightly deviate from the taxonomy of the compositionality of tasks introduced in ClevrSkills and simplify the benchmark into **L0** and **L1** tasks, corresponding to primitive behaviors and more complex, long horizon tasks respectively. The tasks in **L1** are chosen such that they can be achieved by composing multiple subtasks of **L0** together (for details, refer to [13] Appendix A). To be precise, we include the tasks *PlaceNextTo*, *PlaceOnTop*, *Topple* and *Push* into the **L0** dataset, whereas the tasks *SingleStack*, *StackTopple* and *PushToTarget* are part of the **L1** datasets. Aside from *Topple* and *Push*, all tasks have 3 objects chosen at random from a collection of 32 different combinations of colors and shape.

We first generate 500 trajectories for each task of the entire ClevrSkills suite to train our high level policy. Here, we also include additional tasks such as *Pick*, *ReverseStack* and *NovelNoun* which we hold out from the training set of the low level policy as they are mostly testing language understanding and can be readily solved by reusing behaviors from **L0** and **L1** tasks mentioned above. For the low level policy, we generate 2000 trajectories for each task and we train separate policies for the **L0**, **L1** and combined **L0+L1** datasets respectively. As a comparison, we also train a flat diffusion policy with the same architecture, where language conditioning is set to $\ell_t = \ell, \ell_t^{cache} = \ell$, i.e. we replace the low level commands with identical high level *natural language* descriptions of the task.

Results. Table 1 shows success rates per task, separated by training dataset of the low level policy, for hierarchical and flat variants, as well as the performance of **VLM+Oracle** which is obtained by executing the key-step policy $p_\phi(\ell_{k_i}|\mathbf{X}^b_{k_i}, \ell^{in}, \ell^{key}, m_{k_{i-1}})$. The flat variant only receives the task prompt ℓ^{in} in the form of natural language. Overall, we find that using code instructions generated through the VLM is highly beneficial, with success rates improving across all tasks. We observe that this holds even when there is only a small overlap across tasks. For example, this can be seen when comparing success rates on the **L0** dataset with a flat variant. Here, *PlaceNextTo* sees the biggest improvement in performance with a greater than 30% increase, while the only shared primitive with other tasks is picking up the correct object, which is also found in *PlaceOnTop*. Similarly, *Push* does not share any primitives with other **L0** tasks, but still benefits from the decomposition of subtasks.

When comparing the performance of training on a combined dataset **L0+L1** with training on only one dataset respectively, we see that the hierarchical policy can readily reuse instructions from lower level tasks to solve longer horizon tasks. This is mostly pronounced in stacking tasks, which require

chaining together *PlaceOnTop* multiple times and optionally using the *Topple* skill at the right time. Interestingly, however, zero-shot generalization of the low level to solve stacking remains challenging, with a success rate of 0% when trained only on **L0**. In this case, while the policy correctly executes the start of stacking, it tends to fail at lifting blocks high enough towards the end of the trajectory as this is not seen in **PlaceOnTop**. Finally, we find that despite the smaller dataset, our hybrid **VLM+Oracle** policy performs strongly, whereas learning the low level actions remains the most challenging part. This allows zero-shot generalization of the low level policy to tasks that were not explicitly seen in its dataset, such as *Pick*, *ReverseStack* and *NovelNoun*.

Non-Markovian Swapping In Table 1, all tested tasks are solvable using Markovian low-level policies. However, this does not always hold for the high level policy p_ϕ, as code traces depend on internal variables such as target poses, which have to be stored in memory m_t even when the task itself follows a Markov decision process.

To explicitly test the memorization capabilities of our policy, we train on small and large datasets of 1000 and 2000 trajectories of swapping two objects respectively, which requires remembering initial positions of both objects. This is a challenging long-horizon task with many subtasks, as the robot needs to (i) first remember the position of one object, (ii) pick and place it onto a free position, (iii) save the position of the other object before moving it onto the remembered initial position and (iv) pick and place the second object on the last remembered position. We note that the actual number of subtasks is closer to 12, as each moving, picking and placing instruction form their own subtasks.

We compare a flat variant trained on natural language (DP), our hierarchical policy (VLM+DP), the high-level policy (VLM+Oracle) and our low-level policy on a modified version of the task (DP+Oracle). The latter automatically calls an oracle function for computing initial positions and inserts it into the natural language task prompt. This equates to evaluating our low level policy on a Markovian version of the task. Figure 3 shows success rates for each method. As expected, DP without any high level thoughts or additional information fails regardless of training dataset size. Similar to the Markovian tasks, letting the VLM execute its generated thoughts yields the strongest performance, while learning the low level actions requires more trajectories in terms of scaling. We also observe that giving the low level policy sufficient information yields better performance than relying on the VLM. We hypothesize that this is due to the VLM failing at advancing to the next subtask if the low level policy goes slightly out of distribution.

Data Scaling We further investigate whether scaling the number of trajectories in the **L0** dataset proportionally enhances generalization to **L1** tasks. As shown on the right side of Figure 3, we evaluate a larger variant of the **L0** dataset, which includes twice the number of demonstrations for the *PlaceNextTo* and *PlaceOnTop* tasks. Our results reveal not only improved performance on these specific **L0** tasks, but also a notable increase in success rates on the more complex stacking task - despite the number of stacking demonstrations remaining constant. This cross-task improvement

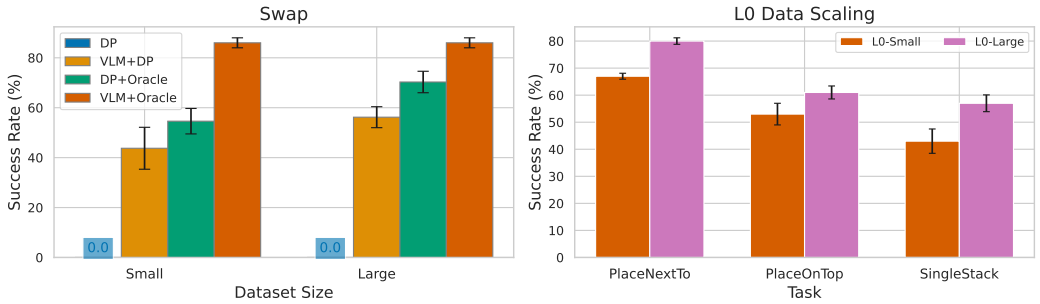

Figure 3: Left: Success rates on *Swap*, a non-Markovian task, divided into small and large datasets used to train the low level policy. Right: Success rates on pick and place tasks when training on a small and a large number of demonstrations for *PlaceNextTo* and *PlaceOnTop*.

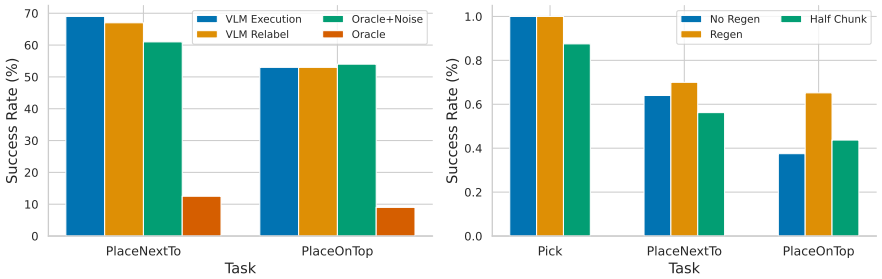

Figure 4: Left: Comparison of different subtask labeling strategies when training on *PlaceNextTo* and *PlaceOnTop*. Right: A comparison of different action chunking strategies during inference. *No Regen* corresponds to always executing the full predicted chunk, whereas *Regen* generates a new chunk when the VLM predicts a new key-step instruction.

provides compelling evidence for compositional generalization, suggesting that the VLM effectively learns to decompose long-horizon tasks into reusable, transferable primitives.

5.2 Ablations

Dataset Generation In Figure 4, we analyze the impact of different strategies for generating the thoughts ℓ_t for training the low level policy. We find that directly using trajectories from $\mathcal{D}_{oracle}$ leads to significantly lower performance. We hypothesize that this is due to a small mismatch in the time at which various subtasks are predicted by the VLM, compared to the start and end times of subtasks when following the oracle policy, we visualize this in Figure 6. However, we found that augmenting oracle thoughts by randomly shifting the start and end times of subtasks by up to 3 steps can mitigate this issue. Our choice of using the VLM to relabel the thoughts performs similarly on *PlaceOnTop*, while slightly better on *PlaceNextTo*. Finally, we also tested using the **VLM+Oracle** policy to generate a completely new demonstration dataset $\mathcal{D}_{exec}$, which yields the best performance on average while being more costly.

Action Chunking As outlined in section 4.3, we regenerate action chunks whenever a new subtask is predicted by the VLM. In Figure 4, we demonstrate the performance of the hierarchical policy with and without this regeneration mechanism when trained on a dataset of 2000 *Pick*, *PlaceNextTo* and *PlaceOnTop* trajectories. We find that the regeneration becomes important when there are many subtasks to be chained together. In *Pick*, which consists of only two subtasks (moving to a pose and picking up), both methods achieve a success rate of 100%. On the other hand, both *PlaceNextTo* and *PlaceOnTop* see a decrease in performance when always executing the full action chunk. In *PlaceOnTop* this is especially pronounced, as the information on which object to place only becomes available after picking up the first object. (In *PlaceNextTo* the API solver precomputes free space next to objects of interests and saves it in memory instead). We also test halfing the prediction horizon (without regeneration), but find that it generally worsens performance across all tasks.

6 Conclusion

In this work, we introduced *From Code to Action*, a hierarchical framework that integrates code-generating vision-language models with code-guided low-level policies to enable compositional generalization in robotic manipulation tasks. Our approach leverages the inherent structure of open-source API policies, allowing for automatic data collection without the need for manual subtask annotations. We investigate whether such code can serve as effective subtask supervision and demonstrate that a VLM, when provided with an appropriate memory buffer, can reliably predict the corresponding API code. Building on this, we develop a diffusion policy conditioned on the VLM-generated code and show that it significantly outperforms a flat policy baseline. Notably, our system exhibits strong signs of compositional generalization, with performance on long-horizon tasks improving as the number of training examples on simpler tasks increases.

References

[1] M. J. Kim, K. Pertsch, S. Karamcheti, T. Xiao, A. Balakrishna, S. Nair, R. Rafailov, E. Foster, G. Lam, P. R. Sanketi, Q. Vuong, T. Kollar, B. Burchfiel, R. Tedrake, D. Sadigh, S. Levine, P. Liang, and C. Finn. Openvla: An open-source vision-language-action model. *ArXiv*, abs/2406.09246, 2024. URL https://api.semanticscholar.org/CorpusID:270440391.

[2] H. Ha, P. Florence, and S. Song. Scaling up and distilling down: Language-guided robot skill acquisition. In *Conference on Robot Learning*, pages 3766–3777. PMLR, 2023.

[3] O. M. Team, D. Ghosh, H. Walke, K. Pertsch, K. Black, O. Mees, S. Dasari, J. Hejna, T. Kreiman, C. Xu, et al. Octo: An open-source generalist robot policy. *arXiv preprint arXiv:2405.12213*, 2024.

[4] A. Brohan, N. Brown, J. Carbajal, Y. Chebotar, J. Dabis, C. Finn, K. Gopalakrishnan, K. Hausman, A. Herzog, J. Hsu, et al. Rt-1: Robotics transformer for real-world control at scale. *arXiv preprint arXiv:2212.06817*, 2022.

[5] N. Blank, M. Reuss, M. Rühle, Ö. E. Yağmurlu, F. Wenzel, O. Mees, and R. Lioutikov. Scaling robot policy learning via zero-shot labeling with foundation models. *arXiv preprint arXiv:2410.17772*, 2024.

[6] B. Siciliano, O. Khatib, and T. Kröger. *Springer handbook of robotics*, volume 200. Springer, 2008.

[7] Z. Chen, J. Yin, Y. Chen, J. Huo, P. Tian, J. Shi, Y. Hou, Y. Li, and Y. Gao. Deco: Task decomposition and skill composition for zero-shot generalization in long-horizon 3d manipulation. *arXiv preprint arXiv:2505.00527*, 2025.

[8] I. Mishani, Y. Shaoul, and M. Likhachev. Mosaic: A skill-centric algorithmic framework for long-horizon manipulation planning. *arXiv preprint arXiv:2504.16738*, 2025.

[9] J. Gu, S. Kirmani, P. Wohlhart, Y. Lu, M. G. Arenas, K. Rao, W. Yu, C. Fu, K. Gopalakrishnan, Z. Xu, et al. Rt-trajectory: Robotic task generalization via hindsight trajectory sketches. *arXiv preprint arXiv:2311.01977*, 2023.

[10] C. Chi, Z. Xu, S. Feng, E. Cousineau, Y. Du, B. Burchfiel, R. Tedrake, and S. Song. Diffusion policy: Visuomotor policy learning via action diffusion. *The International Journal of Robotics Research*, 2023.

[11] O. Mees, L. Hermann, E. Rosete-Beas, and W. Burgard. Calvin: A benchmark for language-conditioned policy learning for long-horizon robot manipulation tasks. *IEEE Robotics and Automation Letters*, 7(3):7327–7334, 2022.

[12] H. R. Walke, K. Black, T. Z. Zhao, Q. Vuong, C. Zheng, P. Hansen-Estruch, A. W. He, V. Myers, M. J. Kim, M. Du, et al. Bridgedata v2: A dataset for robot learning at scale. In *Conference on Robot Learning*, pages 1723–1736. PMLR, 2023.

[13] S. Haresh, D. Dijkman, A. Bhattacharyya, and R. Memisevic. Clevrskills: Compositional language and visual reasoning in robotics. In *The Thirty-eight Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2024.

[14] M. Reuss, Ö. E. Yağmurlu, F. Wenzel, and R. Lioutikov. Multimodal diffusion transformer: Learning versatile behavior from multimodal goals. *arXiv preprint arXiv:2407.05996*, 2024.

[15] H. Li, Q. Feng, Z. Zheng, J. Feng, and A. Knoll. Language-guided object-centric diffusion policy for collision-aware robotic manipulation. *arXiv preprint arXiv:2407.00451*, 2024.

[16] J. Wen, M. Zhu, Y. Zhu, Z. Tang, J. Li, Z. Zhou, C. Li, X. Liu, Y. Peng, C. Shen, et al. Diffusion-vla: Scaling robot foundation models via unified diffusion and autoregression. *arXiv preprint arXiv:2412.03293*, 2024.

[17] S. Liu, L. Wu, B. Li, H. Tan, H. Chen, Z. Wang, K. Xu, H. Su, and J. Zhu. Rdt-1b: a diffusion foundation model for bimanual manipulation. *arXiv preprint arXiv:2410.07864*, 2024.

[18] J. Wen, Y. Zhu, J. Li, M. Zhu, Z. Tang, K. Wu, Z. Xu, N. Liu, R. Cheng, C. Shen, Y. Peng, F. Feng, and J. Tang. Tinyvla: Toward fast, data-efficient vision-language-action models for robotic manipulation. *IEEE Robotics and Automation Letters*, 10:3988–3995, 2024. URL https://api.semanticscholar.org/CorpusID:272753287.

[19] K. Black, N. Brown, D. Driess, A. Esmail, M. Equi, C. Finn, N. Fusai, L. Groom, K. Hausman, B. Ichter, et al. Pi0: A vision-language-action flow model for general robot control. *arXiv preprint arXiv:2410.24164*, 2024.

[20] M. Zawalski, W. Chen, K. Pertsch, O. Mees, C. Finn, and S. Levine. Robotic control via embodied chain-of-thought reasoning. *arXiv preprint arXiv:2407.08693*, 2024.

[21] X. Ma, S. Patidar, I. Haughton, and S. James. Hierarchical diffusion policy for kinematics-aware multi-task robotic manipulation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 18081–18090, 2024.

[22] C. Chen, F. Deng, K. Kawaguchi, C. Gulcehre, and S. Ahn. Simple hierarchical planning with diffusion. *arXiv preprint arXiv:2401.02644*, 2024.

[23] M. Pan, J. Zhang, T. Wu, Y. Zhao, W. Gao, and H. Dong. Omnimanip: Towards general robotic manipulation via object-centric interaction primitives as spatial constraints. *arXiv preprint arXiv:2501.03841*, 2025.

[24] F. Liu, K. Fang, P. Abbeel, and S. Levine. Moka: Open-world robotic manipulation through mark-based visual prompting. *arXiv preprint arXiv:2403.03174*, 2024.

[25] A. Stone, T. Xiao, Y. Lu, K. Gopalakrishnan, K.-H. Lee, Q. H. Vuong, P. Wohlhart, B. Zitkovich, F. Xia, C. Finn, and K. Hausman. Open-world object manipulation using pre-trained vision-language models. In *Conference on Robot Learning*, 2023. URL https://api.semanticscholar.org/CorpusID:257280290.

[26] Y. Li, Y. Deng, J. Zhang, J. Jang, M. Memme, R. Yu, C. R. Garrett, F. Ramos, D. Fox, A. Li, et al. Hamster: Hierarchical action models for open-world robot manipulation. *arXiv preprint arXiv:2502.05485*, 2025.

[27] C. Pan, K. Junge, and J. Hughes. Vision-language-action model and diffusion policy switching enables dexterous control of an anthropomorphic hand. *arXiv preprint arXiv:2410.14022*, 2024.

[28] N. Ingelhag, J. Munkeby, J. van Haastregt, A. Varava, M. C. Welle, and D. Kragic. A robotic skill learning system built upon diffusion policies and foundation models. In *2024 33rd IEEE International Conference on Robot and Human Interactive Communication (ROMAN)*, pages 748–754. IEEE, 2024.

[29] J. Wen, Y. Zhu, J. Li, Z. Tang, C. Shen, and F. Feng. Dexvla: Vision-language model with plug-in diffusion expert for general robot control. *arXiv preprint arXiv:2502.05855*, 2025.

[30] L. X. Shi, B. Ichter, M. Equi, L. Ke, K. Pertsch, Q. Vuong, J. Tanner, A. Walling, H. Wang, N. Fusai, et al. Hi robot: Open-ended instruction following with hierarchical vision-language-action models. *arXiv preprint arXiv:2502.19417*, 2025.

[31] Y. Zhong, X. Huang, R. Li, C. Zhang, Y. Liang, Y. Yang, and Y. Chen. Dexgraspvla: A vision-language-action framework towards general dexterous grasping. *arXiv preprint arXiv:2502.20900*, 2025.

[32] S. Xie, H. Wang, Z. Xiao, R. Wang, and X. Chen. Robotic programmer: Video instructed policy code generation for robotic manipulation. *arXiv preprint arXiv:2501.04268*, 2025.

[33] J. Liang, W. Huang, F. Xia, P. Xu, K. Hausman, B. Ichter, P. Florence, and A. Zeng. Code as policies: Language model programs for embodied control. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pages 9493–9500. IEEE, 2023.

[34] B. Li, P. Wu, P. Abbeel, and J. Malik. Interactive task planning with language models. *ArXiv*, abs/2310.10645, 2023. URL https://api.semanticscholar.org/CorpusID:264172138.

[35] I. Singh, V. Blukis, A. Mousavian, A. Goyal, D. Xu, J. Tremblay, D. Fox, J. Thomason, and A. Garg. Progprompt: Generating situated robot task plans using large language models. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pages 11523–11530. IEEE, 2023.

[36] J. Varley, S. Singh, D. Jain, K. Choromanski, A. Zeng, S. B. R. Chowdhury, K. A. Dubey, and V. Sindhwani. Embodied ai with two arms: Zero-shot learning, safety and modularity. *2024 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3651–3657, 2024. URL https://api.semanticscholar.org/CorpusID:268889821.

[37] P. Zhi, Z. Zhang, M. Han, Z. Zhang, Z. Li, Z. Jiao, B. Jia, and S. Huang. Closed-loop open-vocabulary mobile manipulation with gpt-4v. *ArXiv*, abs/2404.10220, 2024. URL https://api.semanticscholar.org/CorpusID:269157231.

[38] J. Bjorck, F. Castañeda, N. Cherniadev, X. Da, R. Ding, L. Fan, Y. Fang, D. Fox, F. Hu, S. Huang, et al. Gr00t n1: An open foundation model for generalist humanoid robots. *arXiv preprint arXiv:2503.14734*, 2025.

[39] J. Duan, W. Yuan, W. Pumacay, Y. R. Wang, K. Ehsani, D. Fox, and R. Krishna. Manipulate-anything: Automating real-world robots using vision-language models. *arXiv preprint arXiv:2406.18915*, 2024.

[40] M. Ahn, D. Dwibedi, C. Finn, M. G. Arenas, K. Gopalakrishnan, K. Hausman, B. Ichter, A. Irpan, N. Joshi, R. Julian, et al. Autort: Embodied foundation models for large scale orchestration of robotic agents. *arXiv preprint arXiv:2401.12963*, 2024.

[41] T. Z. Zhao, V. Kumar, S. Levine, and C. Finn. Learning fine-grained bimanual manipulation with low-cost hardware. *arXiv preprint arXiv:2304.13705*, 2023.

[42] H. Liu, C. Li, Q. Wu, and Y. J. Lee. Visual instruction tuning. *Advances in neural information processing systems*, 36:34892–34916, 2023.

[43] M. Abdin, J. Aneja, H. Awadalla, A. Awadallah, A. A. Awan, N. Bach, A. Bahree, A. Bakhtiari, J. Bao, H. Behl, et al. Phi-3 technical report: A highly capable language model locally on your phone. *arXiv preprint arXiv:2404.14219*, 2024.

[44] C. Qu, S. Dai, X. Wei, H. Cai, S. Wang, D. Yin, J. Xu, and J.-R. Wen. Tool learning with large language models: A survey. *Frontiers of Computer Science*, 19(8):198343, 2025.

[45] J. Ho, A. Jain, and P. Abbeel. Denoising diffusion probabilistic models. *Advances in neural information processing systems*, 33:6840–6851, 2020.

[46] D. Kingma and R. Gao. Understanding diffusion objectives as the elbo with simple data augmentation. *Advances in Neural Information Processing Systems*, 36:65484–65516, 2023.

[47] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of machine learning research*, 21(140):1–67, 2020.

[48] E. Perez, F. Strub, H. De Vries, V. Dumoulin, and A. Courville. Film: Visual reasoning with a general conditioning layer. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32, 2018.

[49] J. Song, C. Meng, and S. Ermon. Denoising diffusion implicit models. *arXiv preprint arXiv:2010.02502*, 2020.

A Limitations

Our experiments are limited to simulation only and limited to the API of the ClevrSkills benchmark. Future work includes real-world deployment as well as testing the approach on different open-source APIs. Furthermore, our low-level policy vision encoder is trained from scratch and thus naturally limited in generalization. It remains to be explored if large pretrained policies equally benefit from the hierarchical architecture proposed in this paper.

B API Description

Below is a description of the API which the ClevrSkills oracle uses to solve the tasks used in this paper. The VLM is trained to mimic the use of this API, and it is used as conditioning for the diffusion policy.

```

# *************  Utility function API *************

def get_actor(
    actor: str,
    actor_pos: Optional[Tuple[int, int]] = None
) -> sapien.ActorBase:
    """
    :param actor: The name of the actor. The name is matched to the names
    of actors in the scene using Bleu score.
    :param actor_pos: Optional position of the actor in the observation
    image, relative to a coarse 10 by 10 grid. This can be used to
    disambiguate when there are multiple identical actors.
    :return: The actor which matches the description most closely.
    """


def get_pose(actor: sapien.ActorBase) -> sapien.Pose:
    """
    :param actor: A Sapien actor.
    :return: The pose of the actor .
    """

def free_space(actor: sapien.ActorBase) -> sapien.Pose:
    """
    :param actor: The actor to be put in free space.
    :return: A pose for actor in free space.
    """

```

```python
def free_space_next_to(
    actor: sapien.ActorBase,
    next_to_actor: sapien.ActorBase,
    direction: List,
    description: str
) -> sapien.Pose:
    """
    :param actor: the actor to be placed.
    :param next_to_actor: the actor to be placed next to.
    :param direction: direction (list of floats) where to
    place actor relative to next_to_actor.
    :param description: Natural language description of the direction
    (does not influence returned pose).
    :return: A pose for actor, next to next_to_actor, in free space.
    """

def pre_pick_ee_pose(actor: sapien.ActorBase) -> sapien.Pose:
    """
    :param actor: The actor to be picked.
    :return: End-effector pose to move to, to perform a picking operation.
    """

def pre_place_ee_pose(
    actor: sapien.ActorBase,
    target_pose: sapien.Pose
) -> sapien.Pose:
    """
    :param actor: actor to be place. Assumed to be grasped by the agent.
    :param target_pose: The pose to place the actor in.
    :return: the pose where end-effector should move to place the
    actor at target_pose. It is assumed that the EE is currently holding
    the actor.
    """

def pre_push_pose(
    actor: sapien.ActorBase,
    topple: bool = False,
    target_pose: sapien.Pose = None,
) -> sapien.Pose:
    """
    :param actor: The actor to be pushed
    :param topple: When true, the returned pose will be closer to
    the top of the actor, because the goal is to push-to-topple.
    :param target_pose: The target to push towards. Used to compute
    the pushing direction.
    :return: the pose that the end-effector should move in order
    to push actor towards the target_pose.
    """

```

```
79
80  def pose_on_top(
81      actor: sapien.ActorBase,
82      target_actor: sapien.ActorBase
83  ) -> sapien.Pose:
84      """
85      :param actor: the actor to be placed on target_actor.
86      :param target_actor: The target actor.
87      :return: a pose where actor is on top of target_actor.
88      """
89
90  def towards_pose(
91      src_pose:sapien.Pose,
92      dst_pose:sapien.Pose,
93      alpha:float=0.5
94  ) -> sapien.Pose:
95      """
96      :param src_pose: Pose of source actor.
97      :param dst_pose: Pose of destination actor.
98      :param alpha: Blending coefficient between poses.
99      :return: Blended position between src_pose and dst_pose.
100     The orientation of src_pose is used.
101     This function is used to compute how to push source actor
102     towards destination actor.
103     """
104
105 # ************* Policies API *************
106
107 def move3d(
108     ee_target_pose: sapien.Pose = None,
109     match_ori: bool = False,
110     vacuum: bool = False,
111     extend_bounds: float = 0.01,
112     check_done: bool = True,
113 ) -> Move3dSolver:
114     """
115     :param ee_target_pose: the target pose of the end-effector.
116     :param match_ori: Whether the orientation of the ee_target_pose
117     must be matched.
118     :param vacuum: Whether to turn vacuum gripper on or off during moving.
119     :param extend_bounds: By how much to extend the bounds of the grasped
120     actor (in meters) in order to avoid collections.
121     :param check_done: whether the solver should check and self-report
122     that it has completed. In most cases you want to set this to True.
123     :return: A solver (policy) to move the end-effector to the specified
124     pose.
125     """
126
127 def touch(
```

```
    actor: sapien.ActorBase,
    push: bool = False,
    topple: bool = False
) -> TouchSolver:
    """
    :param actor: The actor to be touched, pushed or toppled.
    :param push: Whether to push.
    :param topple: Whether to topple. Toppling takes priority over pushing.
    :return: A solver (policy) to touch/push/topple the actor.
    """


def pick(actor: sapien.ActorBase, lift=0.1):
    """
    :param actor: The actor to be picked.
    :param lift: How much to lift the actor above the initial pose at
    pickup.
    Without lifting a bit, actors could be pushed off the gripper
    during horizontal transport.
    :return: A solver (policy) to pick the actor.
    """

def place(
    actor: sapien.Actor,
    target_pose: sapien.Pose,
    match_ori_2d: bool = False,
    drop_distance: float = 0.02,
) -> PlaceSolver:
    """
    :param actor: Actor to be placed.
    :param target_pose: Absolute pose to place the actor.
    :param match_ori_2d: Match z-axis rotation of target_pose?
    :param drop_distance: The actor will be dropped from this height
    relative to target (in meters).
    :return: A solver (policy) to place the actor in target_pose.
    """

def place_on_actor(
    actor: sapien.Actor,
    target_actor: sapien.Actor,
    target_pose: sapien.Pose,
    match_ori_2d: bool = False,
    drop_distance: float = 0.02,
) -> PlaceOnActorSolver:
    """
    :param actor: The actor to be placed
    :param target_actor: The actor to-be-placed-upon
    :param target_pose: pose (relative to target_actor)
    :param match_ori_2d:  Match z-axis rotation of target_pose?
```

```
    :param drop_distance: From what distance to drop the actor (in meters).
    :return: A solver (policy) to place the actor on target_actor in
    target_pose.
    """


def push_along_path(actor: sapien.ActorBase, target_pose: sapien.Pose) ->
↪  PushAlongPathSolver:
    """
    :param actor: The actor to be pushed.
    :param target_pose: The pose to be pushed towards.
    :return: A solver (policy) to push actor to target_pose,
    while avoiding collisions
    """
```

C Details: High-level VLM

C.1 Auxiliary Loss

We simplify the bounding box representation by dividing images into a 10×10 grid and assigning objects to the nearest patch. Although this approach may compromise some accuracy, our early experiments indicated that predicting two integer values, rather than multiple digits, offered greater robustness while maintaining performance. Consequently, for each image $\mathbf{X}^b$, we obtain a set of bounding boxes $\{(x_i, y_i)\}_{i=1}^k$. These bounding boxes are then utilized to generate a VQA format, where we query the VLM to determine if a randomly selected object is present at a specific location (x_i, y_i). Additionally, we ask the VLM to directly predict (x_i, y_i) based on a given object description. While the prediction of bounding boxes is not directly queried for during inference, it is still utilized when generating code instructions. For example, in the API function `get_actor()`, the location of the actor (object) in the image is used to disambiguate between actors with identical descriptions.

C.2 VLM performance on ClevrSkills

In Table 2 we show the success rate of the **VLM+Oracle** policy on the full ClevrSkills task suite aside from the tasks that require multimodal input prompts. Furthermore, we provide the number of average actions, which denotes the average number of API policy calls which are invoked to solve the task. Each task was evaluated on 100 random seeds.

D Details: Low-level Policy

D.1 Diffusion Policy Preliminaries

Diffusion policy (DP) [10] parametrizes π_θ using diffusion models such as DDPM [45], which entails training a conditional latent variable model $p_\theta(\mathbf{A}^0|\mathbf{o}, \ell) = \int p_\theta(\mathbf{A}^{0:K}|\mathbf{o}, \ell) d_{\mathbf{A}^{1:K}}$. The latents $\mathbf{A}^{1:K}$ are noisy versions of the original data, defined by a forward noise process $q(\mathbf{A}^k|\mathbf{A}^{k-1}) = \mathcal{N}(\mathbf{A^k}; \sqrt{1-\beta_k}\mathbf{A}^k, \beta_k\mathbf{I})$ and $\beta_k > 0$. To reverse the noising process, the model is parametrized as $p_\theta(\mathbf{A}^{k-1}|\mathbf{A}^k, \mathbf{o}, \ell) = \mathcal{N}(\mathbf{A}^{k-1}; \boldsymbol{\mu}_\theta(\mathbf{A}^k, k|\mathbf{o}, \ell), \sigma_k^2\mathbf{I})$ and trained using a weighted Evidence Lower Bound (ELBO) loss [46]. Finally, sampling from $\pi_\theta(\mathbf{o}, \ell)$ is performed by ancestral sampling, starting from $\mathbf{A}^K \sim \mathcal{N}(\mathbf{0}, \sigma^2\mathbf{I})$ and iteratively sampling $\mathbf{A}^{k-1} \sim p_\theta(\cdot|\mathbf{A}^k, \mathbf{o}, \ell)$.

D.2 Architecture

We visualize the low-level policy architecture in Figure 5. The main deviations from the original diffusion policy architecture [10] come from the need to encode lengthy code instructions, as well

Table 2: VLM + scripted policies: performance on a variety of ClevrSkills tasks

Task Name	Level	Success (%)	Avg. #Actions
Pick	0	99	2
Place on top	0	84	2.4
Place next to	0	96	2
Rotate	0	83	3
Throw at	0	71	3
Throw to topple	0	91	3
Touch	0	94	1.9
Push	0	93	2.8
Topple	0	100	2.9
Pick and place on top	1	79	5.3
Pick and place next to	1	96	4.2
Follow_order	1	83	5.2
Follow_order_and_restore	1	55	8.4
Neighbour	1	50	7.2
NovelAdjective	1	31	4.6
NovelNoun	1	58	4.1
NovelNounAdjective	1	56	4.2
Rotate and restore	1	72	4.9
Rotate symmetry	1	58	5.9
Stack	1	90	7.9
Stack in reversed order	1	79	7.5
Sort by texture	1	41	8.2
Swap	1	84	11.2
Throw onto	1	100	2
Balance scale	2	44	10.8
Stack sorted_by_texture	2	57	9.5
Stack and topple	2	81	9.9
Swap by pushing	2	7	9.8
Swap and rotate	2	83	11.3
Throw and sort	2	46	4.5
mean (all levels)	-	72.6	5.3
mean	0	88.3	2.6
mean	1	68.8	6.05
mean	2	53.0	9.3

as the need to enable conditioning on a consistent memory buffer. To encode the code instructions (*task info*) and memory instructions (*cache info*), we use a frozen T5 language model [47], which processes each respectively and produces a sequence of token embeddings.

In addition to language embeddings, we use a lightweight vision encoder based on a standard ResNet-18 to process base and wrist cameras, as well as linear embedding layers for proprioception and extra information corresponding to the gripper state. We treat proprioception and image embeddings as a single token, respectively, and combine them with the language tokens using an attention pooling layer, consisting of several cross-attention blocks. The purpose of the pooling mechanism is to aggregate token-level language embeddings and arrive at a fixed-dimensional embedding, which can then be fed into a diffusion UNet head [10] with FiLM embeddings [48]. Finally, to train the diffusion head, we employ DDPM with a custom loss weighting inspired by [46], using ϵ-prediction with a sigmoid$(-\lambda + 2)$ ELBO weighting and a cosine noise scheduler.

D.3 Memory Preprocessing

We choose to preprocess the history m_t of memory into a single prompt ℓ^{cache} which only contains information about stored variables that are relevant for future frames. For example, in the visualized

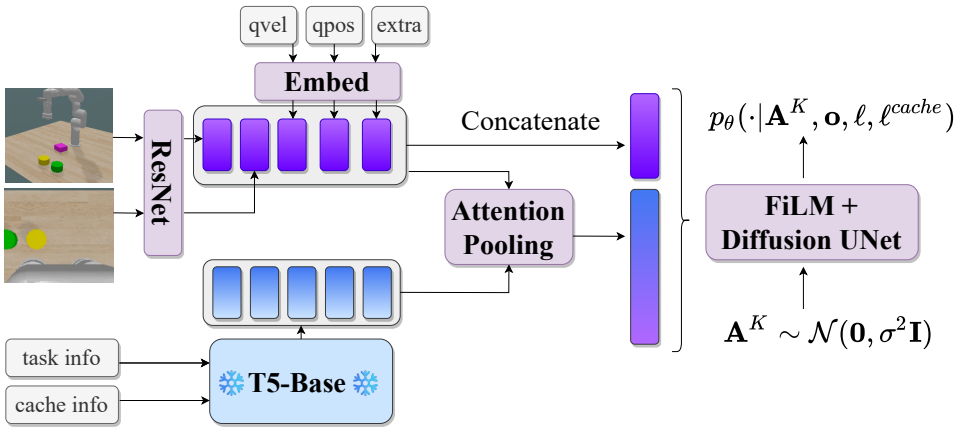

Figure 5: The low level policy is conditioned on proprioception, base and wrist camera images, as well as Python code in the form of task info for code corresponding to the current instruction and cache info for state tracking. Observation embeddings are treated as tokens and cross-attend to language embeddings using an attention pooling mechanism.

trajectory of Figure 2 there is exactly one such instruction which typically occurs at the beginning. The motivation behind this is to allow for greater generalization, since conditioning on a long history of observations can lead to overfitting to specific trajectories, reducing the model's ability to generalize to novel situations. For the same reason, we do not provide the overall task description ℓ^{in}, but force the low level policy to rely only on subtask code instructions.

To separate code instructions from cache information, we use a simple regular expression scanning for `pose_dict` values being set. Algorithm 1 illustrates this behavior. During inference, we can extract caching information from the VLM memory buffer by calling `ExtractMemoryInfo` on its memory of past key-step instructions m_t. This ensures that all instructions that attempt to assign values to some key of `pose_dict` are persistent through time and visible to the low level policy in the form of ℓ^{cache}. If no memory info is returned by the function (i.e. if none of the instructions in m_t were writing to `pose_dict`), we set $\ell^{cache} =$ "null". Figure 2 illustrates the extraction of memory information from the code trace in *PlaceNextTo*. In this task, the oracle (and, as a result, the trained VLM) uses the first timestep to calculate a target placing position alongside outputting a moving instruction. This instruction is then persistent in the memory buffer m_t of the VLM, which we in turn extract in the form of ℓ^{cache} to feed into the low level policy at every time step.

Algorithm 1 Extract Memory Info from Python String

```
procedure ExtractMemoryInfo(python_string)
    Define cache_pattern as regex: pose_dict['.*?']  =
    Split python_string into lines by newline
    Initialize empty list cache_lines
    Initialize empty list remaining_lines
    for each line in lines do
        if regex_pattern matches line then
            Append line to cache_lines
        else
            Append line to remaining_lines
        end if
    end for
    Join remaining_lines into remaining_string
    Join cache_lines into cache_string
    return cache_string, remaining_string
end procedure
```

D.4 Dataset

As described in section 5.1 we train on 2000 trajectories for each of the described tasks on a simple object split using the ClevrSkills simulator. Each object color is randomly chosen from the following list: *cyan*, *red*, *white*, *yellow*, *black*, *blue*, *green*, *purple*, while the object shapes are randomly chosen from a list of *cube*, *cylinder*, *triangle*, *hexagon*. Each dataset is generated using random seeds 12000 to 14000 of the simulator.

In Figure 6 we perform an ablation on the distribution of subtask labels obtained by VLM generations and oracle thoughts. For more details, refer to section 5.2.

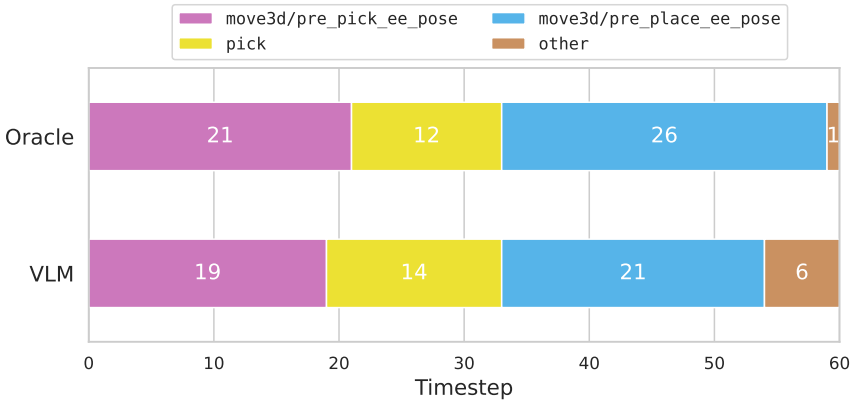

Figure 6: A comparison of subtask labels found in the oracle dataset with those found in VLM generations on a trajectory of the task PlaceNextTo. We find that while the VLM predicts instructions accurately, a small mismatch in start and end times of subtasks can persist. This can lead to a distribution shift when directly training on oracle labels.

D.5 Hyperparameters

We use the AdamW optimizer for all experiments with a learning rate of $1.0e-4$, beta values of $[0.95, 0.999]$, epsilon $1.0e-8$, weight decay $1.0e-6$ and a cosine learning rate scheduler. Furthermore, following the choice of [10] we keep an exponential moving average (EMA) of the model weights using the same hyperparameters. However, we deviate in terms of using historical observations and proprioception values and only provide one timestep of observations into the model. Regarding the diffusion head, we use DDPM [45] with standard hyperparameters:

Table 3: DDPM Noise Scheduler Hyperparameters

Hyperparameter	**Value**
num_train_timesteps	100
beta_start	0.0001
beta_end	0.02
beta_schedule	`squaredcos_cap_v2`
variance_type	`fixed_small`
clip_sample	`True`
prediction_type	`epsilon`

D.6 Testing

During inference, we use 10 denoising steps for faster inference using the DDIM sampler [49]. We always test on 64 random initializations of the environment with seeds 10 to 74.

E Compute Resources

All of our experiments were conducted on a mixture of A100-80GB and V100-32GB GPUs. The high level VLM can be trained on a node of 8 A100s within 48 hours, while the low level policy can be trained separately and requires fewer compute resources. We trained the diffusion policy on a node of 8 V100s for around 24-72 hours depending on the size of the dataset. For our biggest dataset, we train for 250 epochs, taking around 72 hours of walltime. For inference, a single A100 is sufficient to run both the high and lowlevel policy in parallel, i.e. they consume less than 80GB of memory in total.

F Societal Impact

Enabling learning of arbitrary robotic manipulation policies has the potential for societal impact. Our work was performed on simple environments with simple objects, thus limiting the direct potential negative impact and limiting the application to stationary robots in e.g. a warehouse setting. Nonetheless, we acknowledge that the automation of data collection and improving scalability of robot learning can have drastic societal impact due to the possibility to automate previously challenging tasks that required human supervision. This can lead to the replacement of human workers with robots. In the longer term, this can also accelerate the development of arbitrary robot policies which can be used for warfare or other malicious activities.