POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Multimodal Behavior Tree Generation: A Lightweight Vision-Language Approach to Robotic Task Planning

Tesi di Laurea Magistrale in
Computer Engineering - Ingegneria Informatica

Cristiano Battistini, 10705270

Abstract: This thesis explores the use of lightweight Vision-Language Models (VLMs) to generate behavior trees for robots from a visual observation of the scene and a natural language instruction. Motivated by recent advances in vision-language models, we investigate whether visual perception of the scene can be exploited to generate robotic plans that account for the observed state of the environment. While recent works have explored vision-language models for robotic task planning, none has targeted the generation of behavior trees from compact, open-source multimodal models. Since no existing dataset links visual observations and instructions to executable behavior trees, we propose a method to construct one starting from existing robotic datasets. A large-scale model serves as a teacher in a multi-stage generation pipeline guided by prompt engineering. The resulting dataset is used to fine-tune VLMs ranging from 500 M to 4 B parameters via Parameter-Efficient Fine-Tuning (PEFT). We compare multiple VLM architectures, both fine-tuned and in their base form, against significantly larger closed-source models. Our results show that this approach effectively closes the performance gap while requiring only a fraction of the computational resources. The quality of the generated behavior trees, compatible with BehaviorTree.CPP, is assessed both offline, through structural, action-level, and lexical-overlap metrics, and online, through execution of household tasks in a state-of-the-art embodied simulator.

Advisor:
Prof. Matteo Matteucci

Co-advisors:
Dr. Gianluca Bardaro
Riccardo Andrea Izzo

Academic year:
2025-2026

Key-words: vision-language models, behavior trees, robotic task planning, robot learning

1. Introduction

Robots are taking on increasingly complex roles in domestic and service environments, from logistics to personal care. Operating in an unstructured setting demands flexible task planning, capable of handling dynamic and unpredictable surroundings where objects and human presence change continuously. The development of embodied benchmarks such as BEHAVIOR-1K [23], which defines over one thousand everyday activities in a realistic simulation, confirms the breadth of this challenge. Different formal representations exist to describe robotic tasks [16], and among these, behavior trees (BTs) have gained traction as a formalism that combines modularity with reactive execution [11, 18], widely adopted in the ROS2 ecosystem through the BehaviorTree.CPP library[1].

[1] `https://www.behaviortree.dev/`

Recently, large language models (LLMs) have been applied to robotic planning, as they can encode commonsense knowledge and interpret natural language instructions [49]. Along this line, a few works have shown that LLMs can generate behavior trees in XML format from textual task descriptions [26], and one of them [19] demonstrated that compact fine-tuned models can produce valid behavior trees for robotic tasks. More recently, a few approaches have begun to integrate vision-language models (VLMs) into BT generation [41, 48], but they rely on large closed-source models, without fine-tuning and without a dedicated dataset.
However, text-only LLM approaches rely entirely on a textual description of the environment, with no direct perception of the scene. Indeed, a limitation of BTGenBot [19] was that the system had no access to visual data: the robot could not observe its surroundings and had to rely on a manually provided description of the task, making it unable to adapt the plan to the actual state of the environment. This was identified as an open direction for future work. Moreover, no existing dataset links visual observations and natural language instructions to executable behavior trees. To the best of our knowledge, no prior work has addressed behavior tree generation using compact, fine-tuned vision-language models.
In this work, we bridge this gap by proposing a method to construct a multimodal dataset from existing robotic episodes (Open X-Embodiment [30]), using a large-scale model as a teacher in a multi-stage generation pipeline. The resulting dataset is then used to fine-tune a vision-language model with 4 billion parameters via Parameter-Efficient Fine-Tuning (PEFT). The quality of the generated behavior trees is evaluated both offline, through syntactic and semantic metrics, and online, through execution in an embodied simulator.
The main contributions of this work are the following:

(i) a multimodal dataset that pairs visual observations and natural language instructions with executable behavior trees, constructed from real robotic episodes through a generation pipeline guided by a large-scale teacher model;
(ii) multiple open-source vision-language models spanning from 500 M to 4 B parameters, fine-tuned via PEFT and directly deployable on robotic platforms;
(iii) a multi-stage offline evaluation comprising domain-specific metrics on the test set and standard lexical-overlap scores on a subset that includes frontier-scale closed-source models, showing that a compact fine-tuned model can match the output fidelity of systems orders of magnitude larger;
(iv) a comprehensive evaluation in which the generated behavior trees are executed inside an embodied simulator on realistic household activities, benchmarked against a frontier-scale model that serves as a performance ceiling.

The remainder of this work is organized as follows. Section 2 surveys prior work on behavior tree generation, vision-language models, and embodied datasets. Sections 3 and 4 then introduce the VLM architectures and the proposed method, covering problem formulation, dataset construction, and fine-tuning. Results are presented in Section 5, followed by conclusions and future directions in Sections 6–7.

2. Related Work

This section reviews the building blocks of the proposed method. We first introduce Large Language Models (LLMs) and Vision-Language Models (VLMs), then define behavior trees and survey existing approaches to their automatic generation, identifying the gap that this thesis targets. The section closes with the fine-tuning technique used to adapt the models and the robotic dataset from which the training data is derived.

2.1. Large Language Models

Large language models (LLMs) are neural networks that generate text by predicting the next token in a sequence, built on the Transformer architecture [40].
Before a Transformer can process text, the input string must be converted into a sequence of discrete units called *tokens*. A tokenizer splits the text into sub-word pieces drawn from a fixed vocabulary [47]; for example, the word `"robotics"` might become two tokens, `"robot"` and `"ics"`. Each token is then mapped to a dense, real-valued vector through a learned *embedding matrix*: a lookup table of size $|V| \times d$, where $|V|$ is the vocabulary size and d the model dimension [40]. All subsequent computations in the model operate on these vectors rather than on the original text. The same idea reappears in vision-language models (Section 2.2), where visual features are projected into the *same* d-dimensional space so that the language model can process images and text as a single sequence.
Because the attention mechanism is permutation-invariant, the model must receive explicit information about the order of tokens. The original Transformer adds fixed sinusoidal vectors to the embeddings [40]; modern LLMs instead use *Rotary Position Embeddings* (RoPE) [38], which encode relative distances directly into the attention scores and generalize better to sequences longer than those seen during training.
Each Transformer layer is built around *self-attention*, which allows every token in the sequence to look at every

other token and weight their contributions. Given the sequence of embedding vectors, three learned linear projections produce matrices of queries Q, keys K, and values V. The scaled dot-product attention is:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V, \tag{1}$$

where d_k is the dimensionality of each key vector and acts as a scaling factor to prevent the softmax from saturating.

Rather than computing a single attention function, the model runs h parallel *heads*, each with its own set of projections, so that different heads can capture different types of relationships. A common efficiency variant is *Grouped-Query Attention* (GQA) [3], in which several query heads share a single key-value pair; this reduces the memory needed to cache past keys and values during generation. After attention, a position-wise feed-forward network is applied to each token independently. Modern LLMs replace the original ReLU with gated activation functions such as SwiGLU [36], where one of the two projections acts as a learned gate on the other (Figure 1). Each sub-layer (attention and feed-forward) is wrapped in a residual connection followed by normalization; current models favor RMSNorm [45] over the original layer normalization [6] for its lower computational overhead and comparable training stability.

A complete Transformer block therefore consists of a (multi-head) attention layer and a feed-forward layer, each with its own residual connection and normalization. The full model stacks N such blocks.

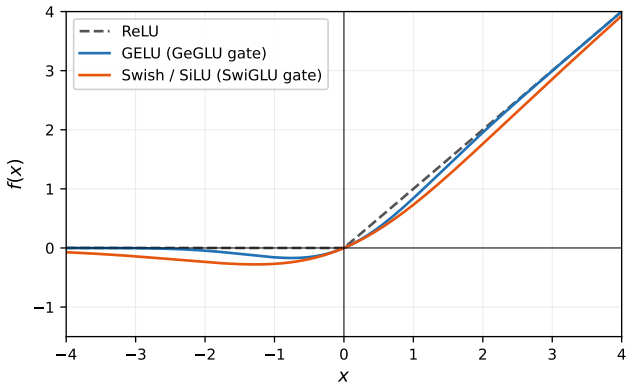


Figure 1: Activation functions used as gating functions in modern Transformer feed-forward blocks.

The original Transformer follows an encoder-decoder design [40]. Modern LLMs simplify this into a decoder-only architecture [47]: input and output form a single token sequence, and the model learns to predict each next token given all preceding ones. Scaling these models to hundreds of billions of parameters has led to qualitatively new behaviors. GPT-3 [9], with 175 billion parameters, showed that a sufficiently large model can solve new tasks by conditioning on a few input-output examples provided in the prompt, without updating its weights, a property known as *in-context learning*. Two further capabilities define modern LLMs: *instruction-following*, where fine-tuning on datasets of task-instruction pairs teaches the model to generalize to unseen tasks described in natural language [46]; and *chain-of-thought (CoT) reasoning*, where asking the model to produce intermediate reasoning steps improves performance on problems that require multi-step inference [43]. The current landscape includes both proprietary and open-source models, the latter reaching competitive performance while allowing fine-tuning and local deployment.

However, LLMs operate exclusively on text. When applied to robotic task planning, they require a textual description of the scene, yet such descriptions are not always available and cannot fully convey spatial relationships or object states.

2.2. Vision-Language Models

Vision-language models (VLMs) extend LLMs by incorporating visual perception into the generative process. Instead of relying on a textual description of the scene, a VLM takes an image as additional input and produces text conditioned on both modalities. A modern generative VLM comprises three main components [8]:

- **Visual encoder.** Typically, a Vision Transformer (ViT) [13], which splits the input image into fixed-size patches, linearly projects each patch into an embedding, and processes the resulting sequence through a Transformer encoder to produce a set of visual feature vectors.
- **Projection module.** A trainable connector that maps the visual features into the embedding space of the language model, so that they can be treated as additional input tokens. Common designs range from

a single linear layer to a multi-layer perceptron; the specific choices adopted in this work are detailed in Section 3.

- **Language model backbone.** A decoder-only LLM that receives the projected visual tokens concatenated with the text tokens and generates the output autoregressively, one token at a time.

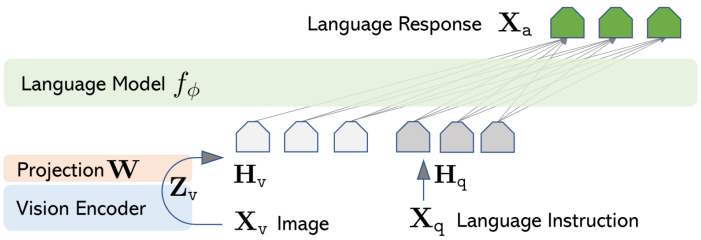


Figure 2: Architecture of the LLaVA vision-language model [25].

At inference time, the image is split into patches by the visual encoder, which produces one embedding per patch. The projection module maps these embeddings into vectors of the same dimensionality used by the language model, and the resulting visual tokens are concatenated with the text tokens to form a single mixed sequence. The LLM then processes this sequence with its standard autoregressive mechanism, attending to both visual and textual tokens as if they belonged to the same modality.

Training a generative VLM typically follows a two-stage procedure [25]. In the first stage, only the projection module is updated on a large set of image-caption pairs, so that visual features become aligned with the language model's embedding space. In the second stage, both the projection module and the language model are fine-tuned end-to-end on instruction-following data, teaching the model to generate detailed responses conditioned on the image. The visual encoder is usually kept frozen throughout both stages, since it already provides strong visual representations from its own pre-training.

The visual encoder can be pre-trained according to two different paradigms:

- **Contrastive pre-training.** CLIP [33] and its variants such as SigLIP [44] train an image encoder and a text encoder jointly so that matching image-text pairs are mapped to nearby points in a shared embedding space, while non-matching pairs are pushed apart. These models perform well on zero-shot classification and retrieval, but they are discriminative and cannot produce free-form text.
- **Generative visual instruction tuning.** LLaVA [25] connects a pre-trained ViT (often the visual encoder from a contrastive model) to a decoder-only LLM through a projection module, and fine-tunes the system end-to-end on instruction-following data that pairs images with conversational text (Figure 2). The result is a model that can produce arbitrary textual output conditioned on an image.

The VLMs fine-tuned in this thesis follow the generative paradigm, because producing a complete behavior tree in XML requires open-ended text generation rather than classification or retrieval.

A related line of work extends VLMs to produce low-level motor commands. These *vision-language-action* (VLA) models, such as RT-2 [51] and OpenVLA [22], output continuous end-effector actions rather than interpretable symbolic plans. This thesis instead uses VLMs to generate structured behavior trees, so that the plan remains readable, modular, and grounded in visual input. Section 2.3 introduces the behavior tree formalism, and Section 2.4 reviews prior work on combining language models with behavior tree generation.

2.3. Behavior Trees in Robotics

A behavior tree (BT) is a plan representation based on a directed rooted tree [11]. The idea originated in the video game industry, where BTs replaced finite state machines for controlling non-player characters, and was later adopted in robotics thanks to its modular structure and its ability to react to changes in the environment [18].

A BT has two kinds of nodes: *internal nodes*, which decide the execution order, and *leaf nodes*, which perform actions or check conditions. Execution follows a protocol called *ticking*: at a fixed frequency a signal called *tick* starts at the root and travels downward through the tree. Each node that receives a tick runs and then reports one of three statuses back to its parent [11]: Success if it completed its task, Failure if it could not, or Running if it is still working and needs more time.

The two main control-flow nodes are Sequence ($\rightarrow$) and Fallback (?) [11]:

- **Sequence.** Ticks its children left to right. It stops at the first child that returns Failure and reports Failure itself. If every child succeeds, the Sequence reports Success.
- **Fallback.** Ticks its children left to right. It stops at the first child that returns Success and reports Success itself. If every child fails, the Fallback reports Failure.

The execution policies of these two nodes are compared in Table 1. The BehaviorTree.CPP library defines additional variants such as `SequenceWithMemory` and `ReactiveFallback` [14].

Table 1: Control-flow node policies in BehaviorTree.CPP.

	Child returns SUCCESS	Child returns FAILURE
Sequence	Tick next child	Stop, return FAILURE
Fallback	Stop, return SUCCESS	Tick next child

A *Decorator* is an internal node with exactly one child. It wraps the child and changes how or how often it is ticked [11]. The two decorators most relevant to this work are:

- **RetryUntilSuccessful.** Keeps ticking its child up to N times. If the child succeeds on any attempt, the node reports SUCCESS; if all N attempts fail, it reports FAILURE.
- **Timeout.** Lets the child run, but if execution takes longer than a given time limit the node interrupts the child and reports FAILURE.

The library also provides `Inverter` (flips SUCCESS and FAILURE), `Repeat`, `ForceSuccess`, `ForceFailure`, and `Delay`; a complete reference is available in [11].

Leaf nodes are the only nodes that interact with the robot and the world. There are two types [11]:

- **Action.** Tells the robot to do something (e.g., navigate to a position, grasp an object). An Action can return any of the three statuses.
- **Condition.** Asks a yes/no question about the current state (e.g., "is the door open?"). It returns SUCCESS for yes and FAILURE for no; it never returns RUNNING because the check is instantaneous.

A typical pattern pairs a Condition with an Action inside a Fallback: the Condition checks whether a precondition already holds, and the Action attempts to satisfy it if it does not. More generally, a Fallback with several Action children acts as a list of alternative strategies: if the first plan fails, the tree automatically tries the next [11].

Behavior trees offer several advantages over finite state machines (FSMs) in the context of a robotic task planning [15, 29]:

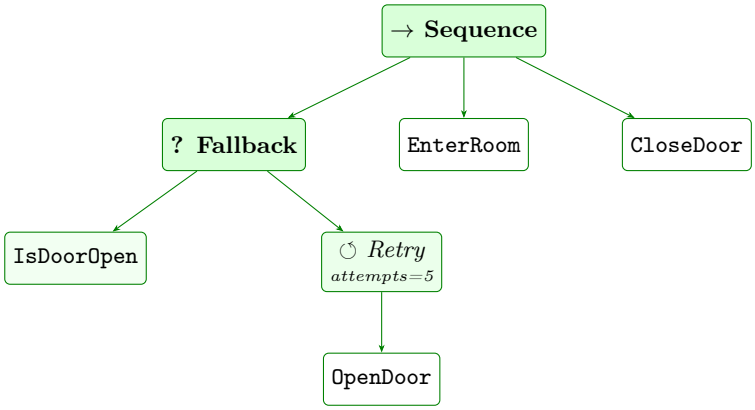
- **Concurrency.** A BT can run multiple branches at the same time through parallel nodes, using the returned status of each branch to coordinate the overall behavior [29].
- **Expressivity.** Sequences, fallbacks, decorators, and parallel nodes can be combined freely, making it possible to encode diverse behaviors within a single formalism [11].
- **Modularity.** A subtree is self-contained: it can be developed and tested once, then plugged into different trees without rewriting any transition logic [11].
- **Reactivity.** In an FSM, once the system enters a state it follows a fixed transition and cannot reconsider until that transition completes. In a BT, the tick travels downward and the status travels upward at every cycle, so the tree can respond to new information at any time [18].
- **Readability.** Both the graphical and the XML form of a BT carry direct semantic meaning, making the plan easier to inspect than an equivalent set of FSM transitions. The tree layout also makes the success/failure logic visible at a glance [11].

Every behavior tree can be written as a unique XML document [11]. The BehaviorTree.CPP library [14] is a C++ framework that loads and runs these XML trees at runtime, and is the standard BT engine in the Robot Operating System 2 (ROS 2), where it powers the Nav2 navigation stack [27]. In the XML format, internal nodes become XML elements with their children nested inside, while leaf nodes become self-closing tags with an `ID` attribute that tells the library which C++ class to instantiate at load time. Because this mapping is resolved at load time, the same XML file can run on different robots as long as each robot registers the required nodes. A `SubTree` tag lets one tree call another by name, so complex behaviors can be split into smaller, reusable pieces. Figure 3 shows a concrete example. The tree opens a door before entering a room. A Fallback first checks whether the door is already open (`IsDoorOpen`); if not, a `RetryUntilSuccessful` decorator tries the `OpenDoor` Action up to five times. Once the Fallback reports SUCCESS, the Sequence moves on to `EnterRoom` and then `CloseDoor`.

Because BehaviorTree.CPP is the standard execution engine in ROS 2, any model that produces valid XML in this format yields plans that can be loaded and executed on a physical robot without an intermediate conversion step.

2.4. Behavior Tree Generation

Generating behavior trees automatically from task descriptions is an open problem that sits at the intersection of natural language understanding and robot planning. The approaches proposed so far differ along two axes: how the model is trained (fine-tuning on BT data versus prompting a general-purpose model) and what the model receives as input (text alone, or text together with visual observations of the scene).




```
<root main_tree_to_execute="MainTree"
      BTCPP_format="4">
 <BehaviorTree ID="MainTree">
  <Sequence>
   <Fallback>
    <IsDoorOpen/>
    <RetryUntilSuccessful
        num_attempts="5">
     <OpenDoor/>
    </RetryUntilSuccessful>
   </Fallback>
   <EnterRoom/>
   <CloseDoor/>
  </Sequence>
 </BehaviorTree>
</root>
```

Figure 3: Behavior tree for a door-opening scenario: graphical representation (left) and XML encoding (right) [14].

From text to trees with compact models. The earliest attempt was LLM-BrAIn [26], which fine-tuned a 7B-parameter Alpaca model on 8 500 pairs of task descriptions and behavior trees, both generated synthetically by `text-davinci-003`. The study found no perceived difference between human-written and machine-generated trees, but the evaluation stopped there: no syntactic or semantic correctness check was performed, and the training data came entirely from another language model [26].
BTGenBot [19] tackled both weaknesses by training on roughly 600 behavior trees collected from real open-source robotics projects [15], each paired with a natural-language description produced by GPT-3.5. The model (LLaMA-2 7B, fine-tuned with LoRA) was evaluated through syntactic validation, semantic review by a human expert, a custom ROS2-based validator, simulation on a TurtleBot3, and execution on a physical robot [19].
BTGenBot-2 [20] pushed the same idea further by shrinking the model to LLaMA-3.2 1B and switching to QLoRA. The training set grew to 5 204 instruction/BT pairs through synthetic augmentation, and the evaluation moved to a standardized benchmark in simulation, where the 1B model surpassed the 7B predecessor in success rate at a fraction of the inference cost [20]. Both BTGenBot versions output BehaviorTree.CPP-compatible XML directly executable through a ROS2 pipeline.
The trajectory from LLM-BrAIn to BTGenBot-2 shows that fine-tuned compact models can generate valid behavior trees with decreasing computational cost. All three systems, however, take only text as input: someone or something must first describe the scene in words before the model can plan.

Large models, no fine-tuning. A parallel line of work uses closed-source models through prompting, avoiding the cost of collecting training data at the price of depending on external APIs. LLM-as-BT-Planner [5] feeds GPT-4 with a structured scene description and a formal action model to produce BTs for robotic assembly; the same study also tried fine-tuning GPT-3.5, Mistral-7B, and LLaMA-2-13B, but found limited improvement on tasks where the tree's internal dependencies had to be correct. LLM-BT [50] takes a different angle: it expands the tree at execution time to recover from unexpected disturbances, using a 3D camera and a recognition network to build a textual semantic map that ChatGPT reads alongside the user instruction. Although a perception module is present, the language model itself still receives only text.
Two other systems go a step further and let a classical algorithm build the tree, restricting the language model to interpreting the instruction. IntentBT [10] uses GPT-3.5 to translate natural language into a formal goal specification; a separate planning algorithm then constructs a provably optimal BT from that specification. BETR-XP-LLM [37] follows a similar split: GPT-4 produces goal conditions, and a classical planner assembles the tree, with an automatic failure-resolution loop that permanently updates the policy when execution fails. In both cases, the language model never touches the tree structure, which gives formal guarantees but limits the system to domains where a complete action model is available.
What all these systems share is that the language model operates on text alone. Most depend on closed-source APIs, and none is fine-tuned on a dedicated BT dataset with visual input. The scene must be described manually, extracted by a separate perception module, or encoded in a formal world state before the model can act on it.

Adding vision. Two recent systems break this pattern by feeding visual observations directly to the model. VLM-BT [41] prompts GPT-4o with a textual instruction, a map, and a skill list to generate a BT in JSON format. During execution, egocentric camera images are sent back to GPT-4o to evaluate visual condition

nodes (for instance, checking whether a cup contains liquid), so that the tree can branch depending on what the robot actually sees. The system was validated on a real robot in a café setting [41]. Video-to-BT [48] goes further: the pipeline takes a human demonstration video as input and uses GPT-4o, Gemini-1.5-flash, or Qwen-2.5VL-72B to decompose the demonstration into reactive BT skeletons, while off-the-shelf object detection and pose estimation models handle perception at execution time. The system was tested on a real-robot assembly task [48].

These results confirm that grounding BT generation in visual input improves context-awareness and enables reactive branching that text-only pipelines cannot express. Both systems, however, share two key limitations. First, they depend on very large models used without fine-tuning on BT data: VLM-BT relies entirely on GPT-4o, which is called both at planning time to produce the tree and at every evaluation of a visual condition node during execution [41]; Video-to-BT tests GPT-4o, Gemini-1.5-flash, and the open-weight Qwen-2.5VL-72B, but none of the three is trained on BT corpora, and the best-performing variant remains GPT-4o [48]. Although Qwen-2.5VL-72B is open-weight, its scale makes on-device deployment impractical. In all cases the models' knowledge of behavior-tree structure comes solely from in-context examples provided in the prompt. Second, neither system releases a dataset that pairs visual observations with executable behavior trees, preventing the community from using their data to train smaller, specialized models.

Vision-language-action models such as RT-2 [51] and OpenVLA [22] (introduced in Section 2.2) also combine visual input with language instructions, but they output low-level motor commands rather than a structured plan. A behavior tree, by contrast, can be inspected and edited by an operator before execution, supports modular reuse of sub-trees across tasks, and runs on the BehaviorTree.CPP ROS2 library already deployed in real robotic systems.

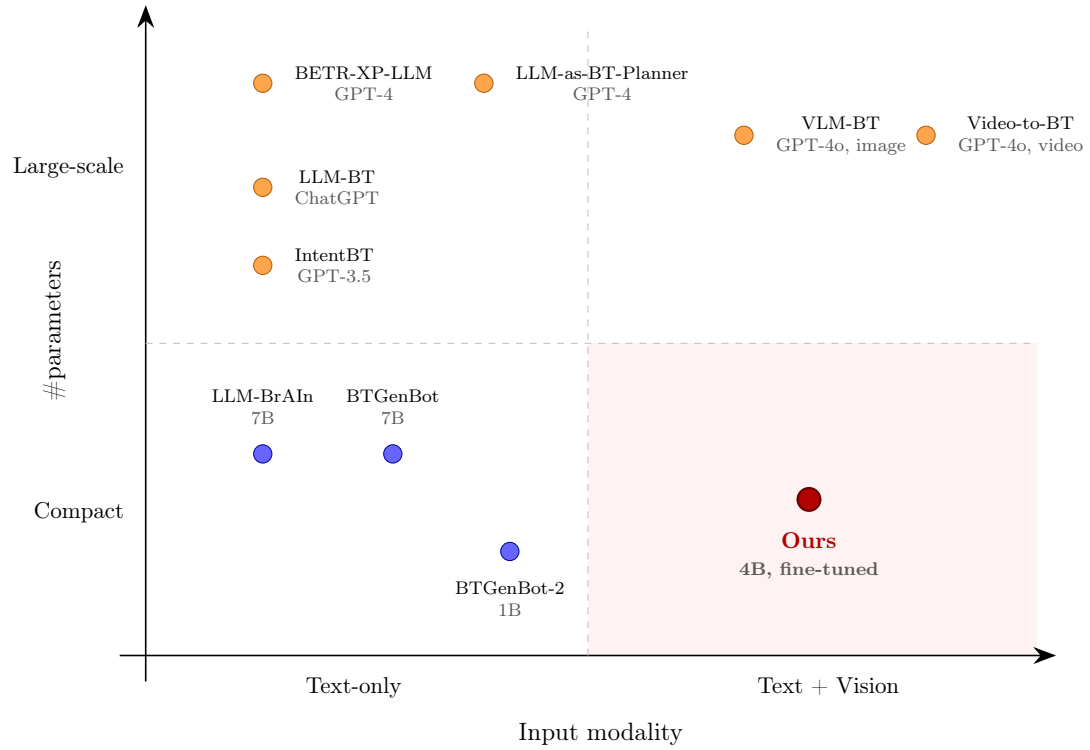


Figure 4: **Comparison of BT generation methods by model size and input type.** Existing methods are categorized by input modality (Text-only vs. Text + Vision) and model scale (a few billion parameters vs. large-scale models). The highlighted shaded quadrant identifies a research gap: previous works have not addressed BT generation using on-device compact Vision-Language Models (VLMs).

No existing work occupies the intersection of compact fine-tuned model and visual input (see Figure 4). This thesis targets precisely that space: a 4B-parameter vision-language model, fine-tuned on a purpose-built multi-modal dataset, receives an image and a textual instruction and produces BehaviorTree.CPP-compatible XML. Because the model is open-source and small enough to run on the robot's own hardware, it removes the dependency on external APIs that all prompt-based approaches share.

2.5. Parameter-Efficient Fine-Tuning

Adapting a pre-trained language model to a specific target task requires updating its parameters on task-specific data. Consider a model $P_\Phi(y \mid x)$ parameterized by Φ, and a training set of context-target pairs $\mathcal{Z} = \{(x_i, y_i)\}_{i=1,\dots,N}$, where each x_i and y_i are token sequences. Standard supervised fine-tuning updates the full parameter set from its pre-trained value Φ_0 to $\Phi_0 + \Delta\Phi$ by maximizing the conditional language modeling objective:

$$\max_{\Phi} \sum_{(x,y)\in\mathcal{Z}} \sum_{t=1}^{|y|} \log P_\Phi\big(y_t \mid x, y_{<t}\big). \tag{2}$$

The update $\Delta\Phi$ has the same cardinality as Φ_0. For models with billions of parameters, storing $\Delta\Phi$ together with the optimizer states exceeds the memory of most available hardware. This cost has motivated *Parameter-Efficient Fine-Tuning* (PEFT), a family of techniques that adapt a pre-trained model by training only a small number of additional or re-parameterized weights while the original parameters remain frozen [32]. Among the PEFT strategies proposed in the literature, Low-Rank Adaptation (LoRA) [17] has become the method of choice for large language and vision-language models, because it adds no inference latency and matches or approaches full fine-tuning accuracy across a range of benchmarks.

LoRA. The theoretical basis for LoRA comes from the observation that pre-trained weight matrices exhibit a low *intrinsic dimensionality*: models can be fine-tuned within a subspace of a few hundred dimensions without degrading task performance [1]. LoRA exploits this property by re-parameterizing the update $\Delta\Phi$ as a function of a much smaller parameter set Θ with $|\Theta| \ll |\Phi_0|$. Concretely, for every dense layer whose pre-trained weight matrix is $W_0 \in \mathbb{R}^{d\times k}$, the update is constrained to have low rank:

$$W_0 + \Delta W \;=\; W_0 + B\,A, \tag{3}$$

where $B \in \mathbb{R}^{d\times r}$, $A \in \mathbb{R}^{r\times k}$, and the rank $r \ll \min(d, k)$. During training W_0 is frozen; only A and B receive gradient updates. The initialization sets A from a random Gaussian and B to zero, so that $\Delta W = BA = 0$ at the start and the model departs from exactly the pre-trained weights. The forward pass through the adapted layer is:

$$h \;=\; W_0\,x \;+\; \frac{\alpha}{r}\,B\,A\,x, \tag{4}$$

where α is a fixed scalar that controls the magnitude of the low-rank update; setting $\alpha = r$ yields a unit scaling factor [17]. The total number of trainable parameters added by a single adapter pair is $r\,(d + k)$, which for typical Transformer dimensions and small ranks ($r \in \{4, 8, 16\}$) is orders of magnitude below $d \times k$. Although the original formulation targets only the self-attention projections [17], the technique is applicable to any linear layer; extending it to the feed-forward matrices can improve task accuracy at the cost of a moderately larger trainable parameter count [32]. Once training is complete, the adapter weights can be merged back into the base model ($W_{\text{merged}} = W_0 + \frac{\alpha}{r}\,B\,A$), so that inference cost remains identical to the original network [17].

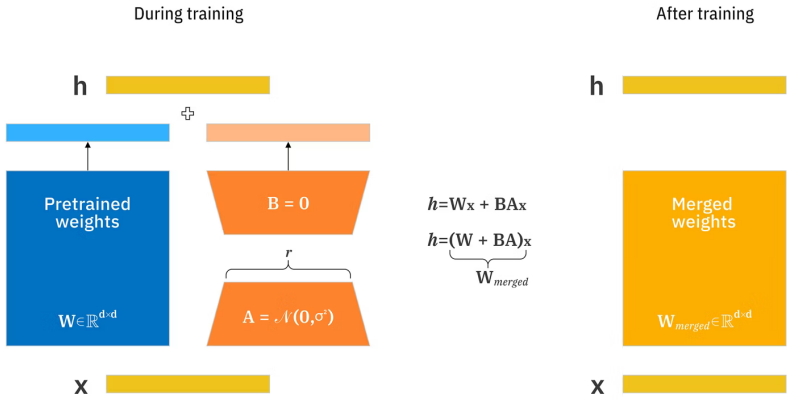


Figure 5: LoRA reparametrization [17].

QLoRA. QLoRA [12] combines LoRA with aggressive weight quantization to reduce memory consumption further. The frozen base model is stored in a 4-bit format called *NormalFloat4* (NF4), whose sixteen quantization levels are derived from the quantiles of a standard normal distribution, matching the empirically observed distribution of pre-trained weights better than uniform schemes. The LoRA matrices A and B remain in BFloat16, and gradients flow through the dequantized weights to update only the adapter parameters. To handle memory peaks during gradient computation, QLoRA introduces *paged optimizers* that offload Adam states to CPU RAM via unified memory. The combination of 4-bit base storage, BFloat16 adapters, and paged optimizers allows models that would require more than 30 GB of VRAM in half precision to be fine-tuned on a single consumer GPU with 24 GB of memory, with accuracy comparable to 16-bit LoRA training [12].

2.6. The Open X-Embodiment Dataset

Most robotic learning datasets focus on a single robot platform or task domain. BridgeData V2, for instance, collects over 60 000 tabletop trajectories on a single low-cost arm [42], while DROID gathers 76 000 demonstrations with a Franka Panda across 13 institutions [21]. Open X-Embodiment unifies 60 such datasets into a single collection that spans 22 robot embodiments, 527 skills, and over one million episodes, all stored in a common data format [30] (Figure 6). Each episode typically contains RGB observations from one or more cameras, a natural language instruction, and a sequence of low-level actions expressed as 7-dimensional end-effector commands.
Although the dataset was originally designed for training end-to-end manipulation policies, in this work, we use it differently: we extract only the visual observations and the associated language instructions, discarding the low-level action sequences. These image-instruction pairs serve as input for building the multimodal dataset on which our VLMs are fine-tuned. The details of this construction are presented in Section 4.1.

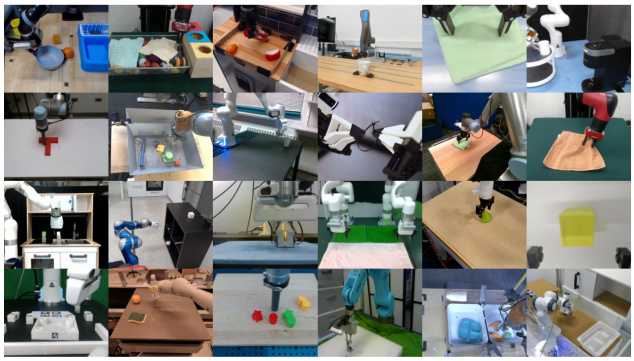


Figure 6: Overview of the Open X-Embodiment dataset [30].

3. Models

The extension from text-only LLMs to vision-language models requires selecting architectures that can ground behavior-tree generation in visual observations while remaining trainable on a single consumer GPU. We choose three open-source VLMs that span an order of magnitude in parameter count: SmolVLM2-500M (∼500 M parameters) by Hugging Face [28], Qwen2.5-VL-3B (∼3.7 B) by Alibaba Cloud [7], and Gemma 3 4B (∼4.3 B) by Google DeepMind [39]. All three follow the generative VLM paradigm introduced in Section 2.2, comprising a visual encoder, a projection module, and a decoder-only language backbone. They differ in image resolution handling (SmolVLM2 tiles the input at a fixed 512×512 per tile, Qwen2.5-VL processes images at their native resolution, and Gemma 3 resizes to a fixed 896×896), as well as in visual-token compression and the balance between vision and language capacity. All three models are loaded in 4-bit quantization (QLoRA) to fit within GPU memory constraints during training.

3.1. SmolVLM2-500M

SmolVLM2 is a family of compact VLMs developed by Hugging Face, explicitly designed for resource-efficient inference [28]. We select the 500M variant, which pairs a 93 M-parameter SigLIP-B/16 encoder with a SmolLM2-360M language backbone [4]. With a batch size equal to one, the model requires only 1.2 GB of GPU memory [28], making it the smallest in our selection by a large margin. Figure 7 illustrates the full pipeline.

As shown on the left side of Figure 7, an input image is first *split* into a grid of sub-images, each resized to 512×512 pixels, alongside a downsized overview of the original [28]. This image-splitting strategy, handled automatically by the Hugging Face processor, allows the model to handle inputs larger than the encoder's native resolution; for our robotic-scene images the processor determines the appropriate number of tiles based on the input size.

The *visual encoder* is SigLIP-B/16 [44], a 93 M-parameter vision Transformer. It processes each tile with a patch size of 16×16, producing 1,024 patch embeddings per tile. The encoder is kept frozen during fine-tuning. The most distinctive step follows immediately after: *pixel shuffle* (space-to-depth) with a ratio $r = 4$ collapses each 4×4 block of spatially adjacent embeddings into a single token whose channel dimension is multiplied by r^2 [28]. This reduces the 1,024 patches to just 64 visual tokens per tile, an aggressive $16\times$ compression that is particularly beneficial at sub-billion parameter scales [28]. The compressed tokens are then mapped to the language model's embedding space through a *linear projection*, as shown in the middle of the figure.

On the right side of Figure 7, the visual tokens from all tiles (labeled by their grid coordinates) are concatenated with the text embeddings produced by the tokenizer and fed into the *language backbone*: SmolLM2-360M [4], a decoder-only Transformer with 8,192-token context window and RoPE base frequency raised to 273,000 to support this extended length [28]. Combined with the 64-token-per-tile compression, the 8K context is sufficient for the visual tokens and the structured textual prompt used in our pipeline. In our setting, each input is a single robotic-scene image; the video capabilities of SmolVLM2 are not used.

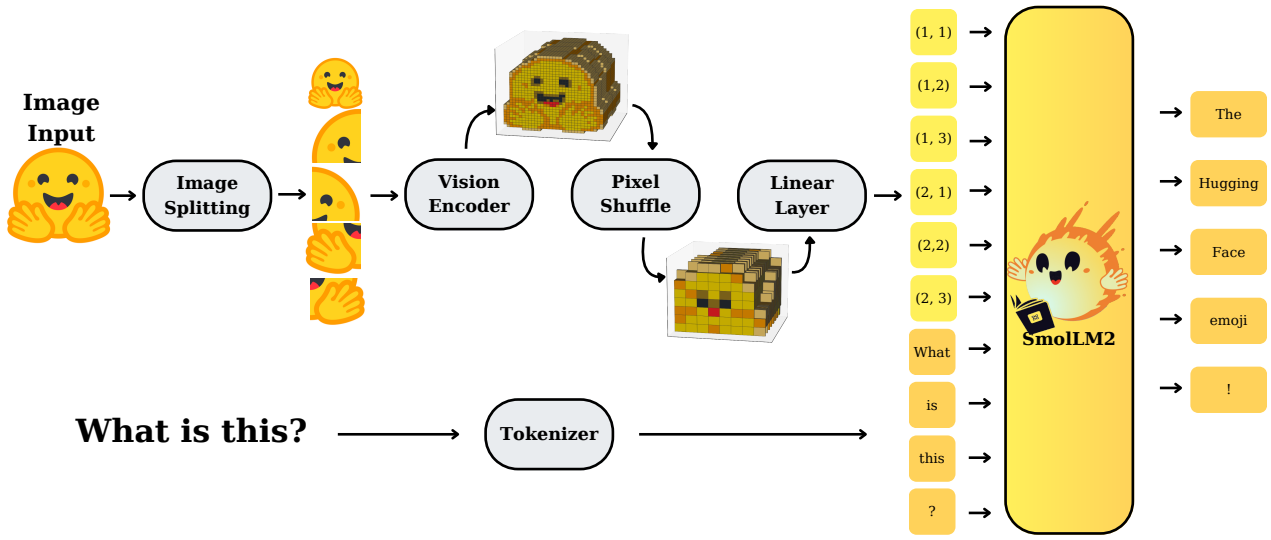


Figure 7: Architecture of SmolVLM2 [28].

3.2. Qwen2.5-VL-3B

Qwen2.5-VL belongs to the Qwen vision-language series developed by Alibaba Cloud [7] and is available in 3B, 7B, and 72B variants. We adopt the 3B model, which was the smallest available at the time of our experiments. Figure 8 provides an overview of the full framework; although the figure also depicts video and multi-image inputs, our pipeline uses only the single-image path described below.

Unlike SmolVLM2, which resizes every tile to a fixed 512×512 resolution, Qwen2.5-VL processes images at their *native resolution*, only rounding height and width to the nearest multiples of 28 pixels so that the subsequent 2×2 spatial merging divides evenly [7]. The visual token count therefore varies with each input, preserving fine-grained detail without the distortions caused by forcing a single canonical size.

The *visual encoder* is a 32-layer Vision Transformer with hidden dimension 1,280 and 16 attention heads, trained from scratch rather than initialized from a frozen contrastive checkpoint [7]. The *projection module* is a two-layer MLP that groups every 2×2 block of adjacent patch features and projects the result to the decoder hidden size (2,048), yielding a $4\times$ reduction in visual tokens [7].

The *language backbone* is a 36-layer Qwen2.5 decoder with Grouped-Query Attention (16 query heads, 2 KV heads) [7]. A distinctive feature is *Multimodal Rotary Position Embedding* (M-RoPE), which decomposes the rotary encoding into three independent axes (temporal, height, width) so that image tokens carry explicit 2D spatial positions directly in the attention computation [7]. In our pipeline, a single RGB image of the robotic scene is provided as input; the video and multi-image capabilities are not used. In 4-bit quantization the model fits comfortably on a single consumer GPU.

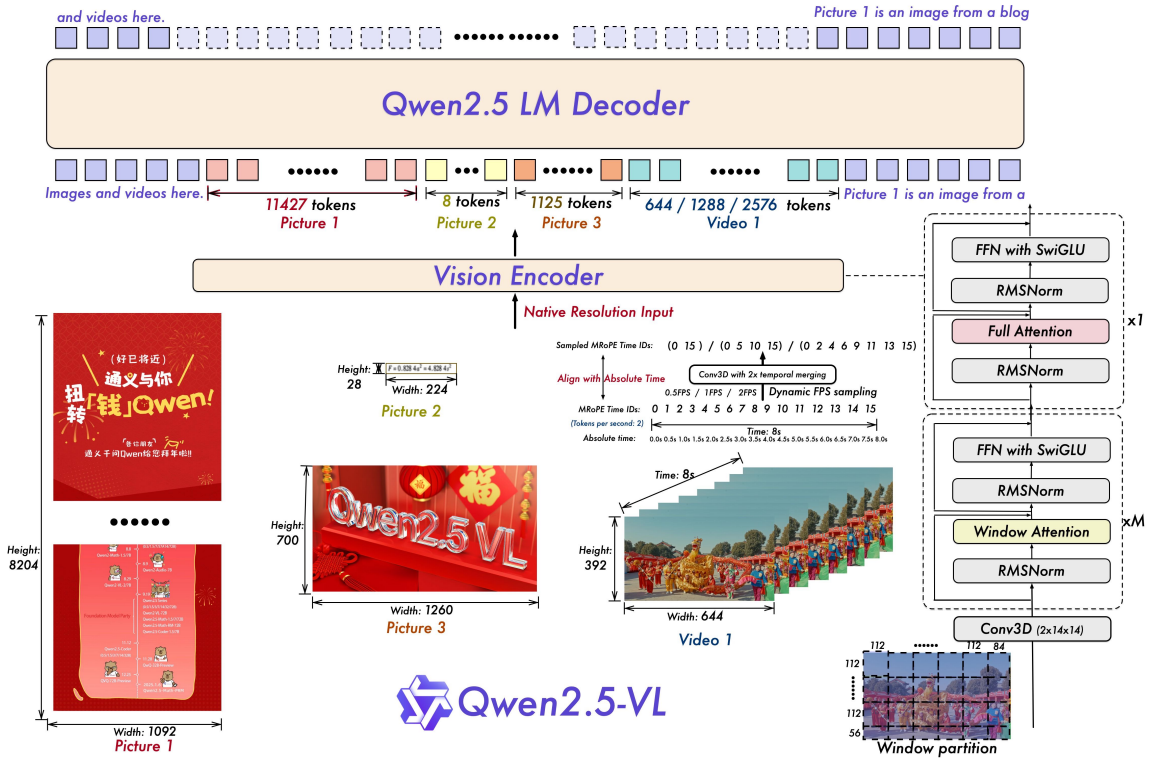


Figure 8: Architecture of the Qwen2.5-VL framework [7]. Only the single-image path is used in our work.

3.3. Gemma 3 4B

Gemma 3 is the third generation of the Gemma family of open models, co-designed with the Gemini frontier models [39], and the first Gemma release to include vision capabilities. The 4B variant we adopt totals approximately 4.3 billion parameters (417 M in the vision encoder, 675 M in embeddings, and 3,209 M in non-embedding weights) [39]. In 4-bit quantization the model occupies roughly 2.6 GB of GPU memory [39].

The *visual encoder* is a 417 M-parameter, 27-layer SigLIP model [44], the largest variant of the same encoder family used by SmolVLM2 (Section 3.1). It is kept frozen during training. The input image is resized to a fixed 896×896 resolution, split into 14×14-pixel patches, yielding $64 \times 64 = 4{,}096$ embeddings, and 4×4 average pooling produces a compact sequence of 256 visual tokens [39]. The *projection module* normalizes these 256 tokens with RMSNorm and linearly projects them to the decoder hidden dimension ($d = 2{,}304$) [39]. The projected visual tokens are then concatenated with the text token embeddings to form a single mixed-modality sequence of $256 + N$ tokens.

The *language backbone* is a 26-layer decoder-only Transformer with Grouped-Query Attention (8 query heads, 4 KV heads). Its layers interleave sliding-window and full-context attention, keeping the KV-cache small while supporting a 128K-token context window [39]. In our pipeline, a single RGB image of the robotic scene is resized to 896×896 and processed through this path; we do not employ multi-crop strategies.

4. Method

Given a single RGB image of the robot workspace and a natural-language instruction, the goal is to generate an executable behavior tree in BehaviorTree.CPP XML format. A model f_θ must learn the mapping

$$f_\theta : (\mathbf{I},\ t,\ \mathcal{A}) \longrightarrow (SA,\ BT), \tag{5}$$

where $\mathbf{I} \in \mathbb{R}^{H \times W \times 3}$ is the RGB observation, t is the instruction string, $\mathcal{A} \subseteq \mathcal{P}$ is the subset of allowed primitives drawn from a fixed action library $\mathcal{P}$, SA is a structured state analysis of the scene, and BT is the behavior tree in XML. The state analysis acts as a chain-of-thought signal: the model is trained to produce it before the XML output, making the generation process interpretable.

The rest of this section is organized as follows. Section 4.1 describes the construction of the training dataset. Section 4.2 presents the fine-tuning procedure and the inference pipeline. Section 4.3 covers the simulation environment used for online evaluation.

4.1. Dataset Generation

This section describes how the training dataset is constructed. Figure 9 provides an overview of the full pipeline. We first present the source data and the frame selection procedure (Section 4.1.1), then the teacher pipeline that generates the behavior trees (Section 4.1.2) and the primitive vocabulary they rely on (Section 4.1.3). We then describe the two augmentation strategies applied to the base trees (Sections 4.1.4 and 4.1.5) and conclude with the final dataset format and statistics (Section 4.1.6).

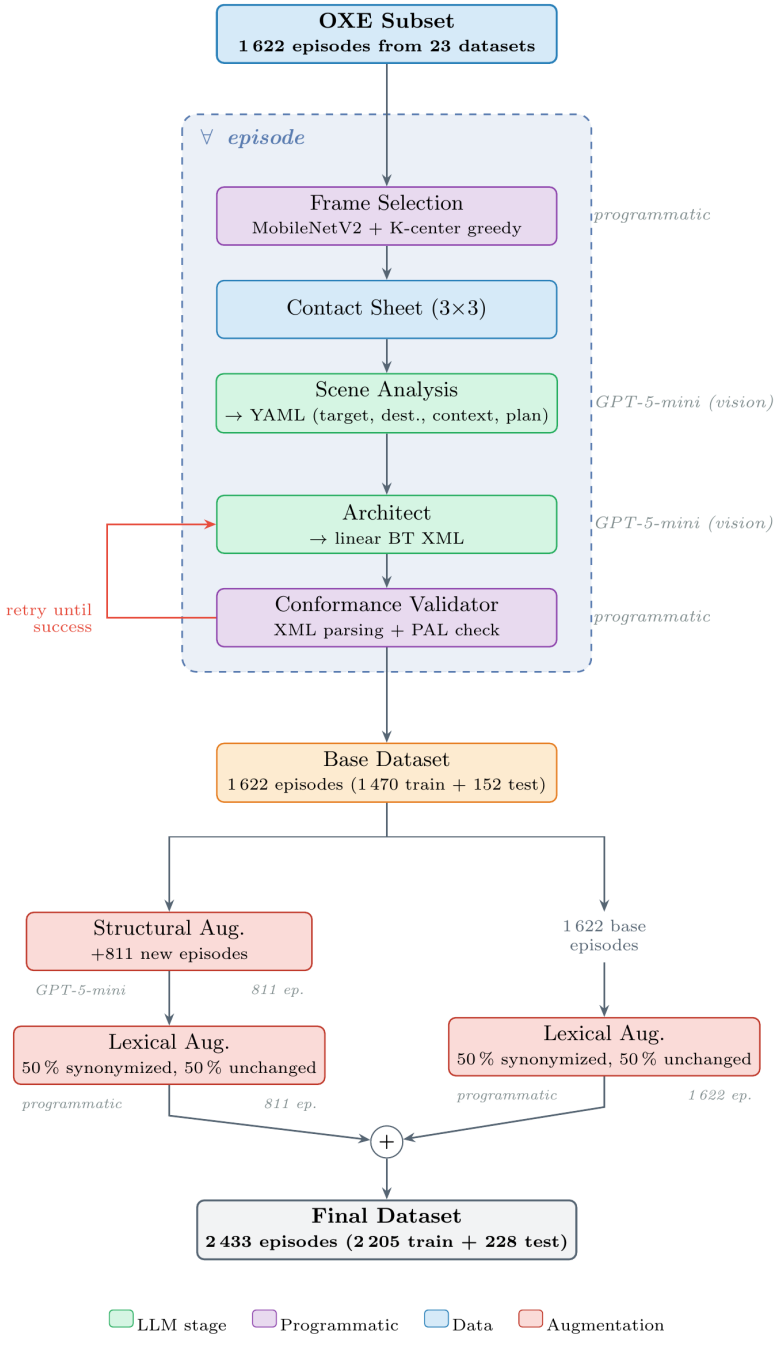


Figure 9: Dataset generation pipeline.

4.1.1. Source Data from Open X-Embodiment

As introduced in Section 2.6, Open X-Embodiment collects over one million robotic episodes across 22 embodiments. From this collection we selected 23 sub-datasets [30], covering tabletop manipulation (e.g. Bridge, Fractal), assistive and shared control (DLR EDAN, DLR SARA), mobile manipulation (CMU Stretch), bimanual tasks (Stanford Hydra, UTokyo xArm), and kitchen activities (UCSD Kitchen, UTokyo PR2), contributed

by 21 institutions; the complete list is provided in Appendix A. Only the visual observations and the language instructions are retained; all low-level action sequences are discarded, since the goal is to generate high-level behavior trees rather than end-effector commands.

Task and instruction distribution. The selected episodes cover a broad range of manipulation skills, though with a pronounced skew towards prehensile tasks: approximately 76 % of the instructions involve picking, grasping, placing, or relocating objects, reflecting the natural composition of Open X-Embodiment. The remaining 24 % form a long tail of eleven other skills, including opening and closing articulated objects, pushing, pouring, folding, flipping, toggling, and inserting. Linguistically, instructions tend to be short and direct (median length of six words): 76 % follow a simple verb-object pattern such as "pick apple" or "open drawer", about 12 % include spatial references ("move the fork to the left of the pot"), and 11 % describe multi-object or multi-step tasks. The five largest contributing sub-datasets (jaco_play, imperialcollege_sawyer, ucsd_kitchen, asu_table_top, and bridge) account for roughly 45 % of the episodes and contribute a disproportionate share of simple grasping instructions.

Instruction filtering. Many OXE sub-datasets repeat the same instruction across hundreds of episodes. To avoid over-representing frequent commands and to encourage diversity in the resulting behavior trees, we impose a cap of 25 episodes per unique instruction. Instructions shorter than 5 characters, or matching uninformative patterns (e.g. `NOT MOVING`, `Video format`, `N/A`) are discarded automatically.

Frame selection. Each OXE episode consists of a variable-length sequence of RGB frames. Rather than sampling uniformly or selecting a fixed stride, we adopt a feature-based selection strategy that maximizes the visual diversity of the chosen frames. The procedure operates in four steps:

1. **Candidate decimation.** The raw frame sequence is sub-sampled with a stride of 10, retaining approximately 10 % of the frames as candidates. If the resulting pool contains fewer than nine frames, the pipeline falls back to nine uniformly spaced indices across the full episode.
2. **Feature extraction.** Each candidate frame is resized to 224×224 pixels and passed through a MobileNetV2 backbone [34] pre-trained on ImageNet, with the classification head removed and global average pooling applied. This produces a 1280-dimensional embedding per frame, which is then L2-normalized to a unit vector. The resize and normalization are applied only for computing embeddings; the frames saved to disk retain their original resolution.
3. **K-center greedy selection.** From the candidate pool, nine frames are selected using the K-center greedy algorithm [35]. The procedure first picks the frame closest to the centroid of all candidate embeddings, then iteratively selects the frame with the lowest maximum cosine similarity to the already-selected set. This guarantees a deterministic, maximally diverse subset of real frames. The selected indices are sorted in temporal order.
4. **Contact sheet assembly.** The nine selected frames are arranged into a 3×3 grid image. Each tile is rescaled to a uniform width of 640 pixels (preserving the aspect ratio) and annotated with a tile index (`[0] ... [8]`) and the original frame number.

Student and teacher images. The dataset stores two images per episode. The *teacher image* is the full 3×3 contact sheet, which provides temporal context across the entire episode. The *student image* is the first frame of the episode (frame 0), copied without any modification. At inference time the student model has access only to the current camera view, so training on a single frame mirrors this constraint.

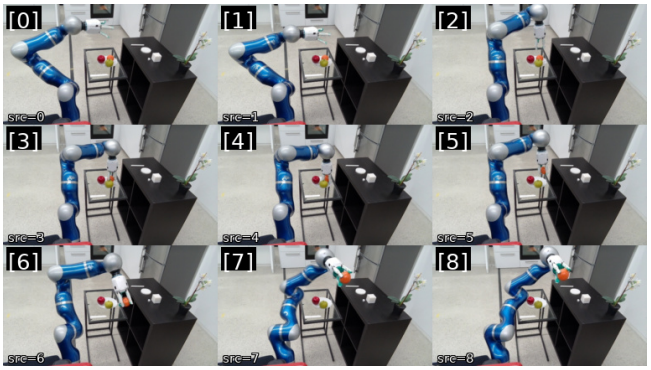


Figure 10: Example 3×3 contact sheet. The student model receives only frame 0; the teacher receives the full grid.

4.1.2. Teacher Pipeline for BT Generation

Since no existing dataset pairs visual observations with behavior trees, the training labels must be generated automatically. For each episode selected in Section 4.1.1, the goal is to produce a valid BehaviorTree.CPP XML plan that solves the task described by the language instruction. Generating the XML directly from the instruction alone, however, would ignore the visual context of the scene. We therefore split the generation into two steps: first, a *state analysis* extracts structured information from the image (which object to manipulate, where to place it, what the initial scene looks like, and what sequence of actions is expected); then, an *architect* uses this analysis together with the image to produce the XML tree. Both steps are carried out by GPT-5-mini, accessed through the OpenAI API, and each receives the 3×3 contact sheet described in Section 4.1.1 as visual input. A programmatic conformance check at the end accepts or rejects the result. Table 2 summarizes the three stages.

Table 2: Teacher pipeline stages.

Stage	Input	Output	LLM involved
Scene Analysis	instruction, contact sheet	YAML with target, destination, scene context, expected action sequence	GPT-5-mini (vision)
Architect	instruction, contact sheet, scene analysis	linear BT XML (Sequence + Action nodes only)	GPT-5-mini (vision)
Conformance	BT XML	ACCEPT or REJECT	none

Scene Analysis stage. The first stage receives the instruction and the contact sheet and produces a structured YAML block containing five fields: `target` (the main object to manipulate), `destination` (where to place it, if applicable), `expanded_instruction` (a more detailed version of the instruction grounded in visual details), `scene_context` (a description of the initial state of the scene), and `expected_sequence` (a natural-language plan of the actions the robot should perform). The `scene_context` field is restricted to describe only the initial state visible in the first frame, so that no future information leaks into the training signal. The `expected_sequence` field serves as a chain-of-thought that the student model will learn to reproduce at inference time.

Scene Analysis prompt (condensed)

Role: Scene analysis for an embodied robot planner.
Inputs: Language instruction + contact sheet (3×3, 9 frames with tile indices).
Output: YAML block with fields `target`, `destination`, `expanded_instruction`, `scene_context`, `expected_sequence`.
Rules: `scene_context` describes only the initial state. `target` supports both single strings and arrays for multi-object tasks. All object names in `snake_case`.

Architect stage. The second step receives the instruction, the contact sheet, and the scene analysis YAML. It generates a behavior tree in BehaviorTree.CPP XML format, constrained to be *linear*: only a single `Sequence` node containing `Action` leaf nodes is allowed. Fallback, RetryUntilSuccessful, Timeout, and SubTree constructs are deliberately excluded at this stage; they are introduced later through data augmentation (Section 4.1.4). Keeping the base trees linear avoids a bias where models learn to always produce complex structures even for simple tasks. The prompt maps natural-language verbs to the primitives defined in the action library (Section 4.1.3) using semantic verb groups: for instance, *pick*, *grab*, *lift*, and *take* all map to `GRASP`. Each `Action` node in the output is preceded by an XML comment that describes the action in natural language.

Architect prompt (condensed)

Role: Behavior tree generator for embodied robots.
Inputs: Instruction + contact sheet + scene analysis YAML.
Output: Linear BT XML (`Sequence` + `Action` nodes only).
Rules: No Fallback, Retry, Timeout, or SubTree. Use only primitives from the action library (Section 4.1.3). Primitives that act on an object take `obj` = target; primitives that place a held object take `obj` = destination. `NAVIGATE_TO` before any manipulation; `GRASP` before any placement; `RELEASE` only at the end.

Conformance validator. The final stage is a purely programmatic check with no LLM involvement. It parses the XML with a standard XML parser, verifies that the root structure conforms to the BehaviorTree.CPP schema, and validates every `Action` node against the primitive action library (Section 4.1.3): each primitive must exist in the library and all required parameters must be present. Episodes that pass receive an `ACCEPT` verdict; on `REJECT`, the Architect stage is re-invoked until a conforming tree is produced.

Design rationale. Splitting the generation into two LLM calls lets each prompt focus on a single objective rather than handling scene understanding and plan generation at the same time. The Scene Analysis stage also serves the student model directly: at inference time the student first produces a state analysis of what it observes in the image and only then generates the XML plan, so the two-step output acts as a chain-of-thought that encourages reasoning about the scene before committing to an action sequence.

4.1.3. Primitive Action Library

Every behavior tree in the dataset draws its actions from a fixed vocabulary of 22 primitives that we refer to as the Primitive Action Library. These primitives represent common manipulation actions (grasping, placing, opening, toggling) that cover the range of household tasks found in the OXE episodes. The names were chosen to be natural and self-explanatory (e.g. `GRASP`, `OPEN`, `PLACE_INSIDE`).
Each primitive takes a single `obj` parameter whose meaning depends on the action. For primitives that act on an object (e.g. `GRASP`, `OPEN`, `TOGGLE_ON`), the parameter identifies the target of the action. For primitives that involve placing or moving a held object to a location (e.g. `PLACE_ON_TOP`, `PLACE_INSIDE`, `POUR`), the parameter identifies the destination; the held object is implicit, determined by the most recent `GRASP` in the tree. `RELEASE` is the only primitive that takes no parameters.

4.1.4. Structural Augmentation

The base behavior trees produced by the teacher are deliberately linear: a single `Sequence` containing `Action` nodes. Real-world robotic plans, however, often require constructs such as `RetryUntilSuccessful` (re-attempt a failed action), `Fallback` (try alternative strategies), `Timeout` (abort after a time limit), `SubTree` (encapsulate reusable sub-plans), and `Condition` nodes (check whether a precondition holds before acting). Rather than asking the teacher to generate these constructs directly, which would risk producing them even when unnecessary, we introduce them through a separate augmentation step applied to the accepted base trees.
The key design principle is that augmentation is *dual*: each template modifies the behavior tree and the text prompt simultaneously. If a `RetryUntilSuccessful` wrapper is added around a `GRASP` node in the XML, the instruction is extended with a phrase such as "Retry grasping up to 3 times if it fails." Without this correspondence, the model would see decorators in the target output with no linguistic cue in the input, making it impossible to learn when to produce them. Depending on the construct, the augmentation may also extend the list of allowed actions (e.g. adding a fallback action that was not in the original set), introduce an `Allowed Conditions` field for `Condition` nodes, and append a `CONSTRAINT` line that makes the expected structural pattern explicit.
A set of 17 prompt templates defines the patterns to be injected: 5 templates target a single construct (timeout, retry, fallback, condition, or subtree) and 12 combine two or more of these constructs. For each base tree, the augmentation code selects the applicable templates based on the instruction and the actions present in the tree, and samples one according to a target probability distribution. The selected template, the original instruction, and the original BT are then sent to GPT-5-mini, which produces both the modified instruction and the augmented tree in a single call. This probabilistic selection ensures a balanced distribution of constructs across the dataset rather than applying every template to every tree.
Figure 11 illustrates the process on a concrete example. The base episode has instruction *"Place the teapot on the stove"* and a linear tree of four actions. After augmentation with a Retry template, the instruction is extended with *"Retry grasping up to 3 times if it fails."*, a constraint line is added, and the `GRASP` node (highlighted in red) is wrapped in a `RetryUntilSuccessful` decorator. In a Fallback variant, the instruction would instead read *"... if placing fails, use an alternative strategy as Plan B"*, the allowed actions would be extended, and the tree would contain a `Fallback` node with a retry branch and an alternative strategy branch.
This procedure generates 811 augmented episodes (735 training, 76 held-out evaluation). Table 3 reports the distribution of decorator types across the augmented set. Retry and Condition are the most frequent individual constructs, while mixed templates, which combine two or three constructs in a single episode, account for roughly a third of the total and span 11 subtypes (e.g. `condition_retry`, `timeout_retry_fallback`).

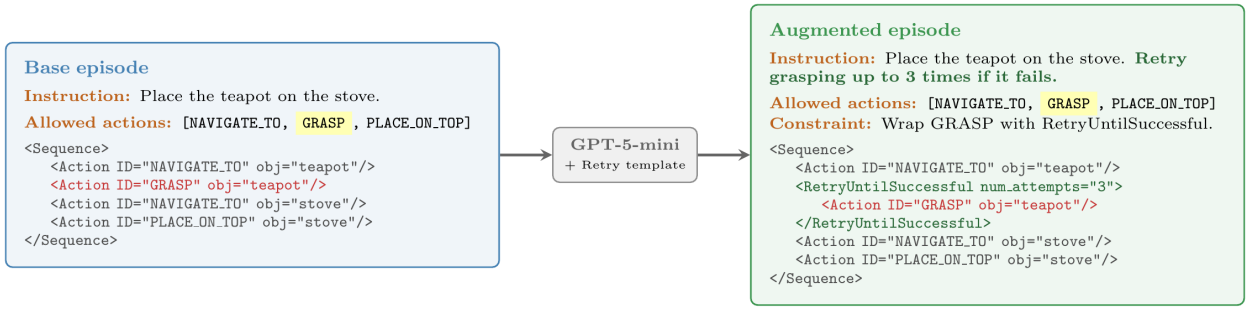


Figure 11: Structural augmentation example. The target primitive (`GRASP`) is highlighted in yellow in the action list and in red in the XML; added constructs are shown in green.

Table 3: Distribution of decorator types in the augmented set. *Episodes* counts how many episodes use each type as primary selection; *Instances* counts every individual construct occurrence.

Decorator	Episodes	%	Instances	%
Retry	147	18.1	340	29.2
Condition	104	12.8	264	22.7
Timeout	80	9.9	219	18.8
Fallback	104	12.8	208	17.9
SubTree	113	13.9	133	11.4
Mixed (2–3 above)	263	32.4	–	–
Total	**811**		**1,164**	

4.1.5. Lexical Augmentation

Once the base and structurally augmented episodes are ready, a lexical augmentation step is applied independently to both subsets. A risk of training on a fixed primitive vocabulary is that the model may memorize rigid action names rather than learning the underlying semantics of each action. To mitigate this, the augmentation rewrites the primitive names in the behavior trees before training, exposing the model to multiple surface forms for the same action. Three types of transformations are applied. All transformations update both the behavior tree and the list of allowed primitives in the user prompt, so that input and output remain consistent.

The first is synonym expansion: each canonical primitive is associated with three high-frequency synonyms (e.g. `GRASP` → `GRAB`, `PICK`, `GRIP`; `PLACE_INSIDE` → `PUT_IN`, `INSERT_INTO`, `SET_IN`). When augmenting an episode, each action node may be replaced by one of its synonyms. The XML comment above the node is updated accordingly, so that the natural-language description remains consistent with the primitive name.

The second targets two-parameter variants for placement primitives. In the canonical form, the held object is implicit (e.g. `PLACE_INSIDE(obj="fridge")`); in the expanded form, both the held object and the destination are explicit (e.g. `PUT_IN(obj="bottle", dest="fridge")`). The held object is resolved by tracking the most recent `GRASP` in the tree. This transformation is applied with 50 % probability when applicable and teaches the model to handle both parameter conventions.

The third introduces solitary actions: rare alternative verb forms (e.g. `CLAMP`, `LATCH_ONTO`, `ADVANCE_TO`) that appear with low probability (5 %), forcing the model to generalize beyond the most common verb forms.

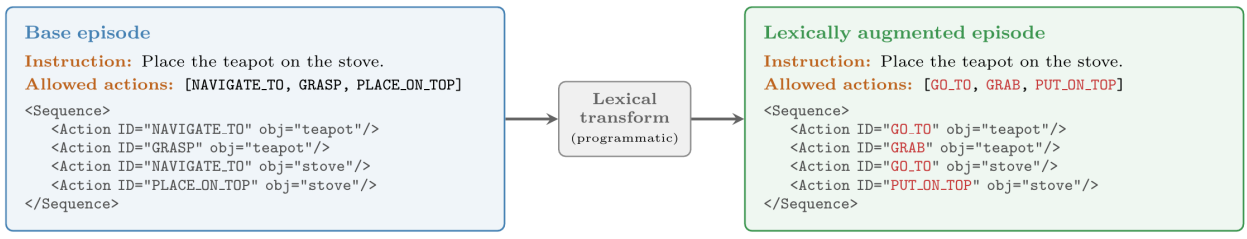


Figure 12: Lexical augmentation example. Canonical primitives (left) are replaced by synonyms (right, in red); the allowed-actions list in the user prompt is updated accordingly.

Input (User Turn)

Instruction: *"Place the teapot on the stove"*

Allowed actions: `NAVIGATE_TO(obj)`, `GRASP(obj)`, `PLACE_ON_TOP(obj)`

Output (Assistant Turn)

State Analysis (YAML)

```yaml
scene_analysis:
  target: "teapot"
  destination: "stove"
  expanded_instruction:
    "Pick up the blue teapot from
    the countertop and place it
    on the stove"
  scene_context:
    "Teapot sits on countertop
    right of sink; stove surface
    is clear."
  expected_sequence:
    "Approach the teapot, grasp
    it, move to the stove, set
    it down."
```

Behavior Tree (XML)

```xml
<root main_tree_to_execute=
      "MainTree">
 <BehaviorTree ID="MainTree">
  <Sequence>
   <!-- Navigate to teapot -->
   <Action ID="NAVIGATE_TO"
           obj="teapot"/>
   <!-- Grasp teapot -->
   <Action ID="GRASP"
           obj="teapot"/>
   <!-- Navigate to stove -->
   <Action ID="NAVIGATE_TO"
           obj="stove"/>
   <!-- Place on stove -->
   <Action ID="PLACE_ON_TOP"
           obj="stove"/>
  </Sequence>
 </BehaviorTree>
</root>
```

Figure 13: Example training sample from the student dataset.

To preserve a stable learning signal, 50 % of the episodes are kept unchanged. All transformations use a fixed random seed for reproducibility.

4.1.6. Student Dataset Format and Statistics

After the teacher pipeline, structural augmentation, and lexical augmentation, all episodes are merged into a single dataset. Each record is formatted as a two-turn conversation suitable for supervised fine-tuning of chat-based VLMs.

The *user turn* contains two elements: the student image (frame 0 of the episode, with no modifications) and a text prompt that includes the natural-language instruction and the list of allowed primitives to be used for the task. The *assistant turn* contains the expected output: a state analysis YAML block followed by the behavior tree in XML. Figure 13 shows a complete training sample. By including the state analysis in the training target, the model learns to reason about the scene before producing the plan.

Table 4 summarizes the final dataset composition. The base episodes come from the teacher pipeline (Section 4.1.2) and the structural augmentation adds episodes with decorator constructs (Section 4.1.4). Lexical augmentation (Section 4.1.5) does not create new episodes; instead, it transforms the primitive names in-place in approximately 50 % of the episodes within each subset, stratified by sub-dataset to preserve balance. In total, 1,228 out of 2,433 episodes (50.5 %) undergo lexical transformation.

Table 4: Dataset composition. The final dataset merges base and structurally augmented episodes. Lexical transformation is applied in-place to roughly half the episodes in each subset.

Subset	Train	Eval	Total	Lex. transformed
Base (linear trees)	1,470	152	1,622	817 (50.4 %)
Structural augmentation	735	76	811	411 (50.7 %)
Total	**2,205**	**228**	**2,433**	**1,228 (50.5 %)**

4.2. Training Configuration

Full fine-tuning of multi-billion-parameter VLMs exceeds the memory of consumer-grade GPUs. We therefore adopt QLoRA [12] (Section 2.5), which keeps the base weights in 4-bit NF4 precision while the LoRA adapter matrices remain in BFloat16. The LoRA hyperparameters are identical across all three models: rank $r = 16$, scaling factor $\alpha = 16$ (effective multiplier $\alpha/r = 1$), and no dropout. We inject LoRA adapters into every linear layer of the model, covering language backbone, visual encoder, and projection module; Table 5 lists the adapted layers. For Gemma 3 4B and Qwen2.5-VL 3B, adapters are injected through the Unsloth library [2] with `target_modules = "all-linear"`; for SmolVLM2-500M, the standard Hugging Face PEFT library is used with an equivalent set of target modules listed explicitly.

Table 5: Linear layers adapted by LoRA in each model component. Each receives a rank-16 adapter pair.

Component	Sub-module	Adapted layers
Language backbone	Self-attention	`q_proj`, `k_proj`, `v_proj`, `o_proj`
	Feed-forward	`gate_proj`, `up_proj`, `down_proj`
Visual encoder	Self-attention	`q_proj`, `k_proj`, `v_proj`, `out_proj`
	Feed-forward	`fc1`, `fc2`*
Projection module	–	Linear mapping visual tokens to language hidden dim.†

* Qwen2.5-VL uses `gate/up/down_proj` in the encoder as well, matching its language backbone.
† Architecture-specific: single linear (Gemma 3), two-layer MLP (Qwen2.5-VL), `modality_projection.proj` (SmolVLM2).

Hyperparameter choice. The rank $r = 16$ was selected after preliminary experiments with $r \in \{8, 16\}$: the higher rank improved the model's ability to reproduce the structured YAML-plus-XML output, likely because visual and textual representations must be jointly reshaped. Setting $\alpha = r$ yields a unit scaling factor, so the adapter contribution is controlled entirely by the learning rate. Dropout is set to zero because the small dataset (2,205 training samples) limits overfitting risk and the 4-bit quantization of the base weights acts as an implicit regularizer [12].

Training setup. Each model is trained on 2,205 samples; the remaining 228 are reserved for evaluation (Section 5). Every sample is a two-turn conversation: the user turn contains frame 0 and the text prompt; the assistant turn contains the state analysis YAML followed by the behavior tree XML. All hyperparameters are reported in Table 6. Experiments run on Google Colab with NVIDIA L4 (Gemma 3, SmolVLM2) and Tesla T4 (Qwen2.5-VL) GPUs; training curves are logged to Weights & Biases and checkpoints are saved at the end of each epoch.

4.3. Simulation and Execution Environment

The fine-tuned models described in the previous section produce behavior trees from an image and an instruction. To determine whether the generated plans actually achieve a physical goal, we execute them inside OmniGibson [23], the reference simulator of the BEHAVIOR-1K benchmark, built on NVIDIA Isaac Sim. The simulated agent is R1, a bimanual mobile manipulator with a holonomic base (3-DOF), a 4-DOF torso, two 6-DOF arms, and parallel-jaw grippers. The RGB observation fed to the VLM is captured from the left wrist-mounted camera (eye-in-hand perspective), so the gripper is visible in the foreground of every frame.
At the start of each episode, the simulation workstation captures an RGB observation from the robot's camera and sends it to the VLM together with the task instruction and the list of allowed primitives (see Section 4.3). The model is loaded in 4-bit quantized form with the LoRA adapter merged at load time. The pipeline extracts

Table 6: Training hyperparameters. All models share the same learning rate (2×10^{-4}), number of epochs (3), and effective batch size (16).

Parameter	Gemma 3 4B	Qwen2.5-VL 3B	SmolVLM2 500M
Epochs	3	3	3
Per-device batch size	8	8	4
Gradient accumulation	2	2	4
Optimizer	AdamW (fused)	AdamW (8-bit)	AdamW
LR scheduler	cosine	linear	linear
Weight decay	0.001	0.001	0.001
Max sequence length	2 048	2 048	–
Max gradient norm	0.3	0.3	0.3

the last `<root>...</root>` XML block from the response; if no valid block is found, the episode is marked as a generation failure.

Inference prompt (condensed)

Role: Embodied planner for a robot operating in a home environment.
Inputs: Scene image (current camera view) + task instruction + list of allowed actions.
Output: `scene_analysis` YAML block, then BehaviorTree.CPP XML plan.
Constraints: (1) Produce the analysis before the XML. (2) Use *only* actions from the allowed set.

All episodes use *symbolic execution mode*: each primitive action is realized as an instantaneous state change rather than a physics-based motor command. `NAVIGATE_TO` teleports the robot next to the target, `GRASP` attaches the object to the end-effector, and placement primitives reposition the held object at the destination. By eliminating low-level control variance and physics noise, a failed episode can be attributed to the plan itself (wrong actions, wrong objects, or wrong ordering) rather than to motor execution. Evaluating plan correctness is the scope of this work; how each primitive would be realized on a physical robot is orthogonal and deferred to future work.

Figure 14 illustrates the complete pipeline from observation to verdict. After the VLM generates a response, an XML parser attempts to extract a valid BehaviorTree.CPP tree. If parsing fails (malformed tags, missing attributes, or non-XML content), the episode terminates immediately with zero ticks and is recorded as a generation failure. If parsing succeeds, the behavior tree is instantiated and ticked: at each tick the executor traverses the tree, dispatching each action node to the corresponding OmniGibson symbolic primitive (Section 4.1.3). Ticking continues until the tree returns SUCCESS, FAILURE, or a maximum tick limit is reached.

Once execution completes, the simulator evaluates the BDDL goal conditions associated with the task. Each BEHAVIOR-1K activity defines a set of goal predicates describing the desired final state, for instance `inside(drill, toolbox)` or ¬`open(toolbox)` (i.e. the toolbox must be closed). The task is considered solved if and only if *every* goal predicate is satisfied; no partial credit is awarded.

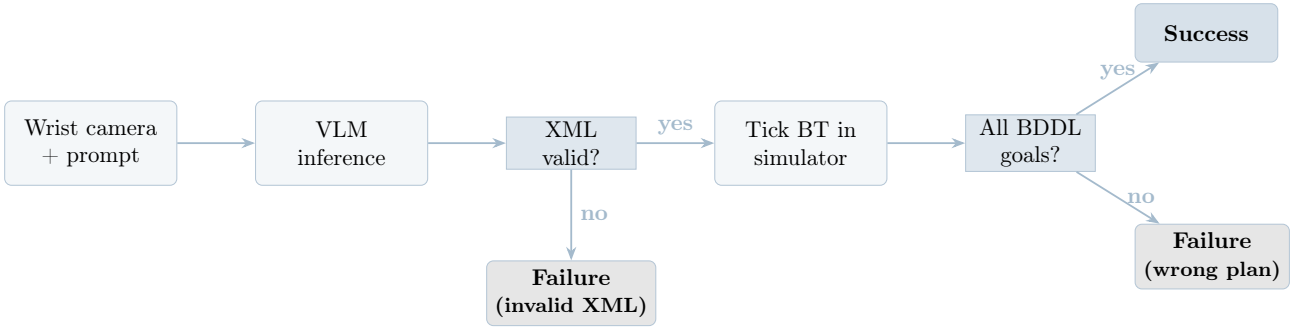


Figure 14: Single-episode execution pipeline: from wrist-camera observation to BDDL goal verification.

The simulation runs on a workstation with an Intel Core i7, an NVIDIA RTX 3080 Ti, and Ubuntu 22.04 (Isaac Sim 4.2). VLM inference runs on a separate machine with an Intel Core i9, an NVIDIA RTX 5080, and Ubuntu 24.04. The two communicate over a local network: the simulation workstation captures the observation, sends it to the inference endpoint, and executes the returned behavior tree locally.

5. Experimental Results

This section presents the experimental evaluation of the proposed system. The three fine-tuned models are compared against their base counterparts and frontier-scale closed-source models through two complementary stages. Section 5.1 assesses the generated behavior trees *offline*, through structural, action-level, and lexical-overlap metrics; these measure how well the model learns the BT-CPP format and how faithfully it follows the task prompt, but they cannot verify whether the resulting plan actually achieves the goal. Section 5.2 closes this gap by executing the generated trees inside the OmniGibson simulator on 15 BEHAVIOR-1K tasks, measuring success against the BDDL goal predicates. A behavior tree that is syntactically valid and lexically close to the ground truth may still fail in simulation if the actions are misordered or target wrong objects; conversely, a plan that differs textually from the reference may succeed if it satisfies the goal conditions. The two stages are therefore complementary: the first measures generation quality, the second measures execution correctness.

5.1. Offline Evaluation

The offline evaluation assesses the quality of the generated behavior trees without executing them. It operates at two granularities. First, Section 5.1.1 evaluates every output on five metrics that probe progressively deeper properties of the generated XML, from surface-level well-formedness to the correctness of the action vocabulary; this analysis runs on the full held-out test split of 228 examples. Second, Section 5.1.2 complements these domain-specific metrics with standard text-overlap scores (BLEU and ROUGE), computed on a 30-example subset that also includes frontier-scale closed-source models; the smaller subset reflects the cost of querying commercial APIs.

5.1.1. Generation Quality Metrics

Each of the three fine-tuned models and its corresponding base version (six configurations in total) is evaluated on the 228-example held-out split (152 linear, 76 with decorators). Five metrics are computed by comparing the extracted XML against the ground-truth behavior tree.

The first three characterize the output at the surface level. *Inference time* is the wall-clock time (in seconds) from the start of tokenization to the end of output decoding, reported as mean and standard deviation; it is relevant because robotic applications require near-real-time planning. *XML validity* is the percentage of outputs whose extracted `<root>...</root>` block can be parsed without error by a standard XML parser; it represents the minimum threshold of usability. *BT-CPP validity* is the percentage of outputs that, beyond being XML-valid, conform to the BehaviorTree.CPP schema [14]: the XML must contain at least one `<BehaviorTree ID="...">` element with a mandatory `ID` attribute, and each such node must have exactly one child (typically `<Sequence>`). The distinction matters because a model may produce well-formed XML (e.g. an HTML fragment) that is entirely unusable as a behavior tree.

The remaining two metrics require a BT-CPP valid output and probe deeper properties of the generated tree. *Structural compliance* verifies whether the *typological structure* of the generated tree matches that of the ground truth. For each pair (o_i, g_i) of generated and reference trees, we extract the set of decorator tags present (from the vocabulary $\mathcal{D} = \{$`RetryUntilSuccessful`, `Fallback`, `Condition`, `Timeout`, `SubTree`$\}$) and require exact set equality:

$$\textsc{StructMatch}(o_i, g_i) \;=\; \begin{cases} 1 & \text{if } \mathcal{D}(o_i) = \mathcal{D}(g_i), \\ 0 & \text{otherwise.} \end{cases} \tag{6}$$

This metric captures whether the model executes exactly what the user requests: a plain sequence when the instruction has no special requirements ($\mathcal{D}(g_i) = 0$), or the appropriate decorators when the prompt contains specific cues (e.g. "retry if the grasp fails" $\rightarrow$ `RetryUntilSuccessful`). We report both the overall percentage and the disaggregation into linear and decorator subsets.

Action Jaccard similarity measures the overlap between the sets of action types extracted from the generated and reference trees:

$$J(o_i,\, g_i) \;=\; \frac{|\,\mathcal{A}(o_i) \,\cap\, \mathcal{A}(g_i)\,|}{|\,\mathcal{A}(o_i) \,\cup\, \mathcal{A}(g_i)\,|}, \tag{7}$$

where $\mathcal{A}(\cdot)$ denotes the set of distinct action primitives (e.g. `NAVIGATE_TO`, `GRASP`, `PLACE_ON_TOP`) present in a tree. Structural tags (`Sequence`, `BehaviorTree`, `root`) and decorator nodes are excluded. The index ranges from 0 (no action in common) to 1 (identical action vocabularies). Because sets disregard multiplicity, the metric captures whether the model selected the correct *types* of primitives for a given task, independently of how many times each primitive appears.

In a robotic setting, the model must select actions exclusively from the allowed primitive list provided in the prompt (see Section 4.3), without hallucinating non-existent primitives (e.g. `STACK` instead of the supported `PLACE_ON_TOP`) or choosing supported but inappropriate ones (e.g. `TOGGLE_ON` for a placement task). The Jaccard index captures both failure modes, since either error inflates the union without contributing to the intersection. This metric is computed only on BT-CPP valid outputs.

Table 7 compares the three models with and without fine-tuning on inference time and syntactic validity. All three fine-tuned models jump from zero to high BT-CPP validity; Gemma-3 FT and Qwen2.5-VL FT reach 100 %. A LoRA adapter adding roughly 2 % of trainable parameters and trained on 2 205 examples is therefore sufficient to teach a format that none of the models has ever encountered during pre-training. SmolVLM2 FT does not reach full validity, indicating that sub-billion-parameter models can acquire the schema but with a residual failure rate.

The baseline column contextualizes this result. No base model produces a single BT-CPP valid output. Qwen2.5-VL base is instructive: it achieves high XML validity, showing that its pre-training corpus contains enough XML-like content to produce well-formed markup, yet the output never conforms to the BT-CPP schema. XML validity is therefore a necessary but not sufficient condition; BT-CPP validity is the actual gate for usability.

Inference times reveal a further effect of fine-tuning. Each model generates both a scene analysis, which acts as a chain-of-thought reasoning step, and the BT XML plan, so generation is not instantaneous, even after fine-tuning. The fine-tuned models produce structured, concise output and stop promptly, whereas the base models, lacking knowledge of the expected format, tend to generate long unstructured text before reaching the stop token. Gemma-3 baseline is roughly five times slower than its fine-tuned counterpart, with a standard deviation that exceeds the mean, reflecting a highly unpredictable generation length. SmolVLM2 FT shows a similar pattern in miniature: a standard deviation nearly double the mean points to a bimodal distribution, where most examples are fast but the malformed outputs correlate with prolonged generation.

Table 7: Inference time and syntactic validity (228-example test split).

Metric	SmolVLM2-500M	Gemma-3 4B	Qwen2.5-VL-3B
Adapter (fine-tuned)			
Inference time (s)	12.7 ± 24.4	20.4 ± 5.5	17.2 ± 4.9
XML valid (%)	88.60	100.00	100.00
BT-CPP valid (%)	87.72	100.00	100.00
Baseline (no fine-tuning)			
Inference time (s)	39.0 ± 31.6	104.6 ± 114.2	13.9 ± 15.1
XML valid (%)	27.19	17.54	82.89
BT-CPP valid (%)	0.00	0.00	0.00

Table 8 moves beyond format correctness and reports structural compliance and action Jaccard for the fine-tuned models. Because no baseline output is BT-CPP valid, these metrics are only meaningful for the adapter configurations.

The results suggest two distinct levels of learning. All three fine-tuned models achieve high structural compliance on linear examples: a flat `<Sequence>` of `<Action>` leaves is a simple structural template that can be acquired even with 500 M parameters, regardless of which specific primitives fill it. Decorator examples expose a sharp capacity threshold: SmolVLM2 FT correctly reproduces the control-flow constructs in only a fraction of cases, whereas the two larger models succeed on the vast majority. The mapping from natural-language cues to structured control flow is therefore largely achieved by the 3–4 B models but remains out of reach for the smallest one.

Gemma-3 FT and Qwen2.5-VL FT show complementary strengths. Gemma-3 FT is the most disciplined on structure: it achieves the highest structural compliance and never adds superfluous decorators to linear examples. Qwen2.5-VL FT is more precise on action selection: it achieves the highest Jaccard with the lowest variance, meaning it consistently picks the right primitives from the allowed set. SmolVLM2 FT shows an interesting dissociation: its action Jaccard is substantially better than its structural compliance, indicating that the model learns which actions to use before it learns how to organize them into the correct control-flow structure.

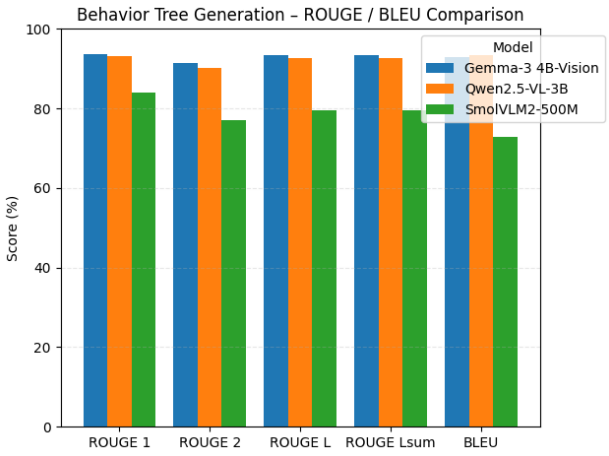
5.1.2. Lexical Metrics

The generation quality metrics of Section 5.1.1 assess structural and action-level properties of the behavior tree but do not capture token-level fidelity to the ground truth. We complement them with BLEU [31] and ROUGE-1, ROUGE-2, ROUGE-L, ROUGE-Lsum [24], computed on a fixed subset of 30 examples from the same test

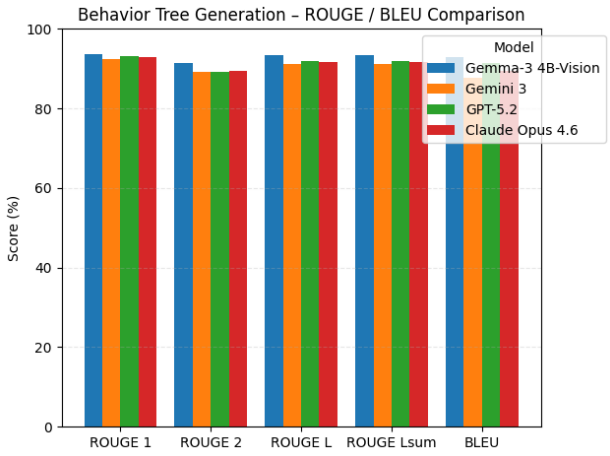
Table 8: Structural compliance and action correctness (fine-tuned models only; n = BT-CPP valid outputs).

Metric	SmolVLM2-500M	Gemma-3 4B	Qwen2.5-VL-3B
Structural match (%)	66.67	96.93	94.74
Linear (of 152)	142	152	146
Decorator (of 76)	10	69	70
Action Jaccard (n)	0.886 ± 0.197 (200)	0.971 ± 0.129 (228)	0.984 ± 0.079 (228)

split: 10 with a purely linear structure and 20 that include one or more control-flow constructs introduced by the structural augmentation stage of the pipeline (Section 4.1.4). ROUGE and BLEU are lexical-overlap metrics and do not directly measure the functional correctness of a behavior tree: a valid plan that follows a different action ordering may receive a low score, while a high score does not guarantee successful execution. In our setting, however, these metrics capture something more specific. Since the evaluation subset includes trees whose control-flow constructs are triggered by stochastically varied natural-language cues, a high score indicates that the model correctly interprets the intent of the prompt and produces the matching XML structure, not merely that it reproduces surface-level tokens. The offline evaluation therefore serves as a first-stage filter for prompt-to-plan fidelity; execution-level correctness is assessed through simulation (Section 5.2). The three fine-tuned lightweight models achieve clearly separated performance tiers (Figure 15a). Gemma-3 4B FT and Qwen2.5-VL-3B FT obtain comparable scores across all metrics, with differences never exceeding 1.5 percentage points on ROUGE. SmolVLM2-500M FT, roughly one eighth the size of Gemma-3, drops substantially on every metric, suggesting that below a certain model-capacity threshold the LoRA adapter cannot fully compensate for the reduced representational power of the backbone. Since the gap between Gemma-3 FT and Qwen2.5-VL FT is marginal at this stage, the final model selection is deferred to the simulation-based comparison in Section 5.2. We also evaluate the same 30 episodes with the three frontier-scale closed-source models, namely GPT-5.2 (OpenAI), Gemini 3 Pro (Google), and Claude Opus 4.6 (Anthropic), each receiving the same guided prompt and scene image with no XML examples or few-shot demonstrations (Figure 15b). The fine-tuned Gemma-3 4B-Vision achieves scores competitive with, and in several cases marginally higher than, those of the frontier models, indicating that a 4B-parameter model fine-tuned with LoRA on a synthetic dataset can reach comparable lexical fidelity to the ground-truth references as systems orders of magnitude larger.



(a) Fine-tuned Lightweight models.



(b) Gemma-3 FT vs. frontier models.

Figure 15: Offline ROUGE and BLEU scores on 30 held-out episodes.

5.2. Simulation

The offline metrics of Section 5.1 measure how closely a generated tree matches a reference, but they cannot determine whether the plan actually works. This section closes the loop: the VLM-generated behavior trees are executed inside OmniGibson using the pipeline described in Section 4.3, and success is measured by BDDL goal satisfaction. Seven models are compared: the three fine-tuned adapters, their three base models (Gemma-3 4B, Qwen2.5-VL-3B, SmolVLM2-500M), and GPT-5 as a frontier reference without any fine-tuning. Each model is tested under two prompting strategies across 15 household tasks.

5.2.1. Task Selection and Prompting Strategies

Fifteen BEHAVIOR-1K activities were selected to cover a progressive range of planning complexity.[2] Table 9 lists each task with its BDDL goal conditions in compact notation, a difficulty rating, and a short type descriptor. The goal conditions are the predicates that must all hold at the end of execution for the task to count as solved; reading them gives an immediate sense of what the robot must accomplish. The difficulty rating reflects how many planning levels the behavior tree must encode.

Table 9: BEHAVIOR-1K tasks used for simulation evaluation. Goal conditions use a compact notation: $\forall$ denotes universal quantification, $\neg$ negation, $\geq n$ a cardinality constraint. Difficulty reflects the number of planning levels required: *Easy* = flat action sequence, *Medium* = container state management, *Hard* = nested or strict ordering constraints.

Task	Diff.	Goal conditions (BDDL)	Type
turning on radio	Easy	`toggled_on(radio)`	State-change
picking up trash	Easy	$\forall$ `can_of_soda: inside(trash_can)`	Repeated pick-and-place
setting mousetraps	Easy	$\forall$ `mousetrap: ontop(floor)`; ≥ 2 `nextto(sink)`	Multi-object placement
hiding Easter eggs	Easy	$\forall$ `egg: nextto(tree)` $\wedge$ `ontop(lawn)` $\forall$ `egg:` $\neg$`inside(basket)`	Shared-target placement
bringing in wood	Easy	$\forall$ `plywood: ontop(floor)`	Repeated transport
moving boxes to storage	Easy	`ontop(box1,floor)` $\wedge$ `ontop(box2,box1)`	Stacking
tidying bedroom	Easy	`nextto(sandal1,bed); nextto(sandal2,sandal1)` `ontop(book,table)`	Chained spatial
picking up toys	Medium	$\forall$ `puzzle, board_game, tennis_ball:` `inside(toy_box)`	Many objects, one container
putting up Christmas decorations	Medium	$\forall$ `gift_box: nextto(tree)`; $\forall$ `candle: ontop(table)` ≥ 1 `candy_cane: ontop(table)`; $\forall$ `wreath: ontop(sofa)` ≥ 2 `candy_cane: ontop(sofa)`	Many objects, many surfaces
preparing lunch box	Medium	$\forall$ `apple, sandwich, cookie, tea: inside(box)` $\neg$`open(fridge)`	Container, mixed sources
bringing water	Medium	$\forall$ `bottle: ontop(coffee_table)`; $\neg$`open(fridge)`	Container-mediated

outfit a basic toolbox	Medium	`inside({drill,pliers,flashlight,wrench,`
		`screwdriver},toolbox);`
		`ontop(toolbox,tabletop)`
		$\neg$`open(toolbox)`
			Repetitive container

putting away Halloween decorations	Hard	$\forall$ `pumpkin, candle: inside(cabinet)` `nextto(caldron,table)`; $\forall$ `cabinet:` $\neg$`open`	Mixed placement, open/close
loading the car	Hard	`inside(camera,container); inside(container,car)` `inside(racket,car)`; $\neg$`open(car)`	Nested containment
carrying in groceries	Hard	`inside(tomato,fridge); inside(milk,fridge)` $\neg$`open(car)` $\wedge$ $\neg$`open(fridge)`	Strict ordering

The seven *Easy* tasks require a flat action sequence: navigate, grasp, place, repeated for each target object. No container state needs tracking and the actions can be executed in any order. These are short-horizon tasks for which the training data provides close analogues, so a fine-tuned model can generalize by scaling a familiar pattern. *Turning on radio* is the simplest case (a single `TOGGLE_ON`); *picking up trash* and *bringing in wood* test repeated cross-room transport; *tidying bedroom* introduces chained spatial constraints.
The five *Medium* tasks add state management: the robot must open or close containers at the right moment, interleaving placement with state-change actions. *Preparing lunch box* requires opening the fridge, placing ingredients in the box, and closing the fridge; *outfit a basic toolbox* repeats the same insert pattern five times

[2]The complete list of BEHAVIOR-1K activities and their definitions is available at `https://behavior.stanford.edu/challenge/tasks/`.

and must close the toolbox at the end. The training data contains individual open, grasp, and place episodes but rarely the full sequence, so the model must compose short-horizon primitives into a longer plan.
The three *Hard* tasks require multi-level reasoning or strict ordering. *Loading the car* demands nested containment: the camera must go inside the container *before* the container is loaded into the car. *Carrying in groceries* has only three goal predicates but the strictest implicit constraint: the robot must free its hands before opening the refrigerator, because `OPEN` fails if the gripper is occupied. *Halloween decorations* combines six objects, two placement types, and a post-condition that all cabinets must be closed. In these tasks, the model must reason about preconditions that are not stated in the goal but emerge from the environment's physics.
Both prompting strategies share a common prefix that assigns the model the role of *Embodied Planner* and specifies the expected output format: a YAML `scene_analysis` block followed by a BehaviorTree.CPP XML plan. The `scene_analysis` is a structured reasoning step in which the model identifies the target objects, the destination, the initial state of the scene, and the expected action sequence in natural language. This intermediate representation grounds the subsequent XML generation in the visual observation.
The difference between the two strategies lies in how much planning guidance the prompt provides. Figure 16 shows both prompts for *loading the car*, where the robot must place a camera inside a container, load the container and a tennis racket into the car, and close the trunk. The *Chain-of-Thought* (CoT) variant includes a numbered `Workflow` that decomposes the task into the exact sequence of primitives; the model's task is then primarily to translate this workflow into valid XML. The *Zero-Shot* variant provides the same goal as a natural-language paragraph but no explicit decomposition: the model must infer the full action sequence from the instruction and the scene image alone.

Zero-Shot prompt (condensed)

Instruction: The car is in the garage with its trunk closed. The tennis_racket and digital_camera are on the table in the living room, and the container is on the living room floor. If you see the car trunk is closed, open it first. Load the tennis_racket, container, and digital_camera into the car. The digital_camera should go inside the container. Close the car when done.
Allowed Actions: `[NAVIGATE_TO, GRASP, PLACE_INSIDE, OPEN, CLOSE]`
Available Objects: `car, container, digital_camera, tennis_racket`

Chain-of-Thought prompt (condensed)

Instruction: [. . . same objects and goal. . .] Load all items into the car and close it when done. The digital_camera should go inside the container.
Workflow:
1. `NAVIGATE_TO(car)` → `OPEN(car)`
2. `NAV` → `digital_camera, GRASP, NAV` → `container, PLACE_INSIDE`
3. `GRASP(container)` → `NAV` → `car, PLACE_INSIDE`
4. `NAV` → `tennis_racket, GRASP, NAV` → `car, PLACE_INSIDE`
5. `CLOSE(car)`

Note: Use ONLY objects from Available Objects.

Figure 16: Zero-Shot and Chain-of-Thought prompts for *loading the car*.

Three metrics are reported for each configuration. *BT Valid* records whether the first-attempt output is parseable as BehaviorTree.CPP XML. *SR* (Success Rate) records whether the first attempt satisfies all BDDL goal predicates. *PASS@3* records whether at least one of up to three attempts succeeds; the retry protocol uses early stopping, so no further attempts are made after a success.

5.2.2. Results

Before examining the aggregate numbers, it is useful to see what a complete episode looks like. Figure 17 shows the Gemma-3 adapter on *picking up trash* (CoT), one of the easy tasks. The robot observes a living room with three soda cans on the floor (Figure 17a). The model first produces the `scene_analysis` shown in Figure 18, identifying the cans as targets and the trash can as destination, then generates a BT XML that repeats the same four-action pattern three times, once per can. After execution, all cans are inside the trash can (Figure 17b) and the single BDDL predicate is satisfied. This is an Easy task with a flat plan; the interesting question is what happens when the tasks grow harder.
The first observation is that fine-tuning is essential. All three base models score 0/15 on every metric: without the adapter, none produces a single BT-CPP-valid tree, making simulation execution impossible. After fine-

(a) Initial observation (b) After execution (goal satisfied)

Figure 17: *Picking up trash*, Gemma-3 adapter (CoT).

Model output (condensed) — Gemma-3 4B (QLoRA), *picking up trash*, CoT

```
scene_analysis:
target: "can_of_soda"
destination: "trash_can"
expanded_instruction: "Pick up the three cans of soda and place each into the trash can in the
kitchen."
scene_context: "Trash can is open and empty on the floor; three cans are on the floor in the
living room and not held."
```

```
<Sequence>
<Action ID="NAVIGATE_TO" obj="can_of_soda_1"/>
<Action ID="GRASP" obj="can_of_soda_1"/>
<Action ID="NAVIGATE_TO" obj="trash_can"/>
<Action ID="PLACE_INSIDE" obj="trash_can"/>
[...same pattern for can_of_soda_2, and can_of_soda_3]
</Sequence>
```

Figure 18: Complete VLM output for the episode in Figure 17: structured scene analysis followed by BT XML.

tuning, Gemma-3 and Qwen both reach 100% BT Valid, meaning the syntactic gap with GPT-5 is fully closed. The remaining gap is in planning quality: on SR and PASS@3, the metrics that measure whether the robot actually completes the task, Gemma-3 approaches but does not match the frontier model (87% vs. 100% SR; 93% vs. 100% PASS@3). The dataset, constructed through the pipeline of Section 4.1, proves effective at teaching the BT-CPP output format to models that have never seen it during pre-training.

This improvement is not gradual: there is a qualitative threshold between 500M and 3B parameters. SmolVLM2 is not a weaker version of Qwen; it is rarely able to produce structured XML, regardless of how much task-specific training it receives. Once the model crosses this threshold, failures shift from syntactic (broken XML) to semantic (valid XML, wrong plan), which is a qualitatively different kind of mistake.

GPT-5 saturates the benchmark in both conditions. This is intentional: the 15 tasks are designed to measure how close lightweight, locally deployable models can get to a known performance ceiling, not to challenge a frontier system.

Table 10: Aggregate simulation results (15 tasks, up to 3 attempts). Open-source models are shown both before (Base) and after (FT) QLoRA fine-tuning; GPT-5 is accessed via API without fine-tuning. Best result per metric in **bold**; best open-source result marked with †.

Model	Variant	Prompt	BT Valid	SR	PASS@3
GPT-5	—	CoT	15/15 (100%)	**15/15 (100%)**	**15/15 (100%)**
		Zero-Shot	15/15 (100%)	**15/15 (100%)**	**15/15 (100%)**
Gemma-3 4B	FT	CoT	15/15 (100%)	†13/15 (87%)	†14/15 (93%)
		Zero-Shot	15/15 (100%)	11/15 (73%)	12/15 (80%)
	Base	CoT	0/15 (0%)	0/15 (0%)	0/15 (0%)
		Zero-Shot	0/15 (0%)	0/15 (0%)	0/15 (0%)
Qwen2.5-VL-3B	FT	CoT	15/15 (100%)	10/15 (67%)	13/15 (87%)
		Zero-Shot	15/15 (100%)	7/15 (47%)	10/15 (67%)
	Base	CoT	0/15 (0%)	0/15 (0%)	0/15 (0%)
		Zero-Shot	0/15 (0%)	0/15 (0%)	0/15 (0%)
SmolVLM2-500M	FT	CoT	1/15 (7%)	0/15 (0%)	0/15 (0%)
		Zero-Shot	4/15 (27%)	0/15 (0%)	1/15 (7%)
	Base	CoT	0/15 (0%)	0/15 (0%)	0/15 (0%)
		Zero-Shot	0/15 (0%)	0/15 (0%)	0/15 (0%)

The gap between Chain-of-Thought and Zero-Shot is itself informative. On Easy tasks the two strategies produce comparable results because the plan is a flat repetition of patterns already present in the training data. The difference emerges on Medium and Hard tasks, where the model must manage container states or respect ordering constraints that are not stated in the goal. The CoT prompt externalizes this reasoning into the Workflow, acting as scaffolding that helps models which can already generate XML but struggle with multi-step planning. For SmolVLM2 the additional tokens are counterproductive: CoT reduces rather than increases its BT Valid rate. More broadly, the size of the CoT-to-Zero-Shot drop reveals how much of a model's performance depends on the prompt rather than on its own reasoning. The larger the drop, the weaker the intrinsic planning ability.

The training data consists predominantly of short-horizon, single-action episodes, yet Gemma-3 and Qwen solve Medium tasks that require composing these primitives into longer sequences with open/close logic. This partial generalization from short-horizon training to longer-horizon evaluation is encouraging, but it reaches its limit on the Hard tasks, where the plan requires reasoning about implicit physical preconditions that were never present in the training data.

To see where this limit manifests concretely, we compare all four models on *carrying in groceries* under CoT. This Hard task requires the robot to place a sack beside the refrigerator, open the fridge with empty hands, store a tomato and a carton of milk inside, close the fridge, and close the car. Figure 19 shows the scene and Figure 20 shows the critical XML fragment generated by each model.

The four outputs show a clear progression. GPT-5 understands the implicit constraint: it places the sack, frees its hands, and only then opens the fridge. Gemma-3 produces valid XML with the right objects and actions, but violates the ordering: it grasps the tomato before opening the fridge, which fails because the gripper is occupied. Qwen generates valid XML too, but the plan is semantically confused: it opens the fridge while still holding the sack, uses the wrong placement action for the tomato, and attempts to grasp the car. SmolVLM2 never reaches the planning stage because it cannot produce well-formed XML in the first place. Each model fails at a different level, and moving from GPT-5 to SmolVLM2 the errors shift from planning to syntax, reflecting the relationship between model scale and the type of mistake.

Figure 19: Scene observation for *carrying in groceries*.

GPT-5 ✓

```
<PLACE_NEXT_TO obj="electric_refrigerator"/>
<OPEN obj="electric_refrigerator"/>     ← hands free, OPEN succeeds
<GRASP obj="beefsteak_tomato"/> ... <PLACE_INSIDE .../> <CLOSE .../> <CLOSE obj="car"/>
```

Gemma-3 4B × ordering error

```
<PLACE_NEXT_TO obj="electric_refrigerator"/>
<GRASP obj="beefsteak_tomato"/>     ← picks up before opening
<OPEN obj="electric_refrigerator"/>     ← fails: hands full
```

Qwen2.5-VL-3B × semantic confusion

```
<GRASP obj="sack"/> <OPEN obj="electric_refrigerator"/>     ← hands full
<PLACE_NEXT_TO obj="electric_refrigerator"/>     ← wrong action
<GRASP obj="car"/>     ← nonsensical
```

SmolVLM2-500M × invalid XML

```
  </Sequence>     ← tree closes here
<Action ID="NAVIGATE_TO" .../>     ← outside tree
<Action ID="PLACE_INSIDE" destination=.../>     ← wrong attribute
```

Figure 20: Four-model comparison on *carrying in groceries* (CoT). Each box shows the critical fragment; annotations mark the error.

This pattern is not specific to a single task. Across the full experiment, SmolVLM2's failures are almost entirely at the XML level, whereas Qwen and Gemma-3 both fail predominantly at the plan level, though Gemma-3 does so less frequently. GPT-5 never fails.
Among the plan failures, two patterns recur. The first is a violation of physical preconditions: the model places an object inside a closed container (`PLACE_INSIDE` before `OPEN`) or tries to open a door while holding something (`GRASP` before `OPEN`). These are not random errors; the models fail to internalize constraints about the physical state of the environment, even when the CoT prompt spells them out. The second is object-name hallucination: SmolVLM2 attempts `GRASP(car)`, Qwen confuses object suffixes, and Gemma-3 occasionally invents names like `small_orange_container` instead of the correct `storage_container_1`. GPT-5 never hallucinates an object name. Both patterns become more frequent as the model size decreases, which suggests that smaller models maintain a less stable representation of the available actions and objects, even when both lists are provided in the prompt.

6. Conclusions

We proposed a method for generating behavior trees from visual observations and natural language instructions using lightweight vision-language models. Since no existing dataset links images and instructions to executable behavior trees, we designed a multi-stage teacher pipeline that constructs such a dataset from real robotic episodes, and used it to fine-tune open-source VLMs via PEFT. We compared three compact models of different scale and selected Gemma-3 4B-Vision [39] as the best performer based on offline metrics and online simulations. We then evaluated it against GPT-5, a state-of-the-art closed-source model of significantly larger scale, and found that the fine-tuned 4B model reaches competitive scores despite the large gap in model scale. After fine-tuning, the model always produces syntactically valid behavior trees in zero-shot on the offline test set, whereas the same model without the adapter is never able to do so.
Beyond syntax, the generated trees are also semantically correct: tested on fifteen household activities from the BEHAVIOR-1K benchmark [23], ranging from simple pick-and-place to multi-step tasks that require opening containers, managing nested containment, and coordinating strict action sequences, the plans were successfully executed in simulation. On Easy and Medium tasks, the compact model performed comparably to GPT-5; on Hard tasks, which require reasoning about implicit physical preconditions such as freeing the gripper before opening a container, the gap widens. The simulation experiments also revealed a qualitative capacity threshold: below approximately 3B parameters, fine-tuning cannot teach reliable structured output, and the dominant failure mode shifts from semantic (wrong plan) to syntactic (broken XML). Above this threshold, Chain-of-Thought prompting acts as effective scaffolding for multi-step planning, while in Zero-Shot the model's intrinsic reasoning ability determines success.
The model generates only the plan: the low-level actions are still implemented through hand-written code, which remains one of the most demanding aspects of the system. Because the output is a behavior tree, the generated plan remains interpretable and open to human-in-the-loop intervention: an operator can inspect, modify, or approve the tree before and during execution.
All fine-tuned weights, training code, and the dataset are released as open-source. The dataset construction stage does depend on a proprietary teacher model (which could in principle be replaced by a sufficiently capable open-source alternative), but this is a one-time cost: once the dataset exists, all subsequent training and inference run entirely on open-source models without sending data to external servers. The generated XML is directly compatible with the BehaviorTree.CPP framework and, by extension, with ROS2: the same plan that runs in simulation can be loaded on a physical robot without changes to its structure. With 4 billion parameters, the model fits on a consumer-grade GPU and on the embedded accelerators already available on robotic platforms, making on-device inference feasible without cloud connectivity. The step from simulation to physical deployment is therefore a matter of integration, not of reengineering.

7. Future Work

The results of this work are encouraging, yet some limitations point to areas for future improvement. The main limitation is that every leaf node in the behavior tree is currently executed through a symbolic simulator action, which abstracts away the difficulty of real manipulation. In the future, these hand-coded primitives could be replaced by learned low-level policies, such as vision-language-action models or reinforcement-learning skills, so that the VLM retains its role as a high-level planner while the actual motor execution is learned from data rather than manually programmed.
A second limitation is that the system relies on a single image captured before execution begins. Providing multiple viewpoints or a short video at planning time would improve spatial understanding. More importantly, keeping the visual channel open during execution would allow the system to detect unexpected states, for

instance an occupied gripper or a fallen object, and react accordingly. A natural extension would be a closed-loop, multi-adapter architecture in which the same lightweight backbone serves multiple roles through different LoRA adapters: a *proposer* adapter that generates the initial behavior tree from the scene, and a *runtime refiner* adapter that receives updated observations during execution, compares them against the expected state, and proposes corrections to the plan. Because adapter switching is inexpensive, such a system could run on the same hardware without requiring a separate monitor model. VLM-BT [41] and LLM-BT [50] explored similar closed-loop strategies with large closed-source models; achieving comparable reactivity with compact fine-tuned models has not yet been demonstrated. Looking further ahead, the closed-loop architecture could evolve into a fully agentic system in which the VLM controls the entire decision loop: deciding whether the current observation provides enough information to plan or whether the robot should first explore the scene (e.g. rotating its head or approaching an object for a closer view), generating the behavior tree, and during execution autonomously choosing whether to retry a failed action, reformulate the plan, or request human assistance. Such a system would use the behavior tree as a living working memory that the model reads and rewrites at runtime in response to visual feedback.

On the structural side, the behavior trees generated in this thesis use only a limited subset of the constructs available in the BehaviorTree.CPP specification: Fallback, RetryUntilSuccessful, Timeout, Condition, and SubTree. Exposing the model to the full vocabulary of decorators and control-flow nodes offered by the framework would allow the planner to select the most appropriate construct for each situation rather than being restricted to a fixed set.

The training data presents a limitation that the simulation experiments made concrete: the current dataset is built from episodes that are predominantly short-horizon manipulations, and the models generalize well to straightforward tasks that compose these primitives into longer sequences but struggle with more layered plans that require reasoning about implicit physical preconditions never seen during training. Two directions could address this. As the community releases longer and more diverse demonstrations, the same teacher pipeline can be re-run to produce long-horizon behavior trees. Independently of the data, a constraint-aware training objective or a lightweight precondition verifier that checks physical feasibility (e.g. "is the gripper empty before OPEN?") before execution could reduce the ordering violations that account for a large share of the plan failures.

The simulation experiments also highlighted object-name hallucination as the most frequent semantic error, with smaller models inventing or confusing object identifiers even when the allowed list is provided in the prompt. In a real setting, symbolic object names may no longer suffice to identify targets unambiguously, especially when multiple instances of the same category are present; integrating visual grounding modules that produce spatially resolved references, such as bounding boxes or segmentation masks, would connect the symbolic plan to the physical scene and reduce hallucination by anchoring each name to a visual entity.

Finally, the current evaluation is limited to simulation. Developing a complete pipeline that can operate on a physical robot, from visual observation to behavior tree generation to real motor execution, would provide a thorough test of the approach in uncontrolled conditions.

References

[1] Armen Aghajanyan, Sonal Gupta, and Luke Zettlemoyer. Intrinsic dimensionality explains the effectiveness of language model fine-tuning. In *Proceedings of the 59th annual meeting of the association for computational linguistics and the 11th international joint conference on natural language processing (volume 1: long papers)*, pages 7319–7328, 2021.

[2] Unsloth AI, Daniel Han-Chen, and Michael Han-Chen. Unsloth. `https://github.com/unslothai/unsloth`, 2025.

[3] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. Gqa: Training generalized multi-query transformer models from multi-head checkpoints. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 4895–4901, 2023.

[4] Loubna Ben Allal, Anton Lozhkov, Elie Bakouch, Gabriel Martín Blázquez, Guilherme Penedo, Lewis Tunstall, Andrés Marafioti, Hynek Kydlíček, Agustín Piqueres Lajarín, Vaibhav Srivastav, et al. Smollm2: When smol goes big–data-centric training of a small language model. *arXiv preprint arXiv:2502.02737*, 2025.

[5] Jicong Ao, Fan Wu, Yansong Wu, Abdalla Swiki, and Sami Haddadin. Llm-as-bt-planner: Leveraging llms for behavior tree generation in robot task planning. In *2025 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1233–1239. IEEE, 2025.

[6] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

[7] Shuai Bai, Keqin Chen, Xuejing Liu, Jialin Wang, Wenbin Ge, Sibo Song, Kai Dang, Peng Wang, Shijie Wang, Jun Tang, et al. Qwen2. 5-vl technical report. *arXiv preprint arXiv:2502.13923*, 2025.

[8] Florian Bordes, Richard Yuanzhe Pang, Anurag Ajay, Alexander C Li, Adrien Bardes, Suzanne Petryk, Oscar Mañas, Zhiqiu Lin, Anas Mahmoud, Bargav Jayaraman, et al. An introduction to vision-language modeling. *arXiv preprint arXiv:2405.17247*, 2024.

[9] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[10] Xinglin Chen, Yishuai Cai, Yunxin Mao, Minglong Li, Wenjing Yang, Weixia Xu, and Ji Wang. Integrating intent understanding and optimal behavior planning for behavior tree generation from human instructions. *arXiv preprint arXiv:2405.07474*, 2024.

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

[12] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. Qlora: Efficient finetuning of quantized llms. *Advances in neural information processing systems*, 36:10088–10115, 2023.

[13] Alexey Dosovitskiy. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

[14] Davide Faconti. BehaviorTree.CPP: An open-source C++ framework for behavior trees. `https://www.behaviortree.dev/`, 2024. Accessed: 2025.

[15] Razan Ghzouli, Thorsten Berger, Einar Broch Johnsen, Andrzej Wasowski, and Swaib Dragule. Behavior trees and state machines in robotics applications. *IEEE Transactions on Software Engineering*, 49(9):4243–4267, 2023.

[16] Huihui Guo, Fan Wu, Yunchuan Qin, Ruihui Li, Keqin Li, and Kenli Li. Recent trends in task and motion planning for robotics: A survey. *ACM Computing Surveys*, 55(13s):1–36, 2023.

[17] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen, et al. Lora: Low-rank adaptation of large language models. *ICLR*, 1(2):3, 2022.

[18] Matteo Iovino, Edvards Scukins, Jonathan Styrud, Petter Ögren, and Christian Smith. A survey of behavior trees in robotics and ai. *Robotics and Autonomous Systems*, 154:104096, 2022.

[19] Riccardo Andrea Izzo, Gianluca Bardaro, and Matteo Matteucci. Btgenbot: Behavior tree generation for robotic tasks with lightweight llms. In *2024 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 9684–9690. IEEE, 2024.

[20] Riccardo Andrea Izzo, Gianluca Bardaro, and Matteo Matteucci. Btgenbot-2: Efficient behavior tree generation with small language models, 2026.

[21] Alexander Khazatsky, Karl Pertsch, Suraj Nair, Ashwin Balakrishna, Sudeep Dasari, Siddharth Karamcheti, Soroush Nasiriany, Mohan Kumar Srirama, Lawrence Yunliang Chen, Kirsty Ellis, et al. Droid: A large-scale in-the-wild robot manipulation dataset. *arXiv preprint arXiv:2403.12945*, 2024.

[22] Moo Jin Kim, Karl Pertsch, Siddharth Karamcheti, Ted Xiao, Ashwin Balakrishna, Suraj Nair, Rafael Rafailov, Ethan Foster, Grace Lam, Pannag Sanketi, et al. Openvla: An open-source vision-language-action model. *arXiv preprint arXiv:2406.09246*, 2024.

[23] Chengshu Li, Ruohan Zhang, Josiah Wong, Cem Gokmen, Sanjana Srivastava, Roberto Martín-Martín, Chen Wang, Gabrael Levine, Wensi Ai, Benjamin Martinez, et al. Behavior-1k: A human-centered, embodied ai benchmark with 1,000 everyday activities and realistic simulation. *arXiv preprint arXiv:2403.09227*, 2024.

[24] Chin-Yew Lin. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*, pages 74–81, 2004.

[25] Haotian Liu, Chunyuan Li, Qingyang Wu, and Yong Jae Lee. Visual instruction tuning. *Advances in neural information processing systems*, 36:34892–34916, 2023.

[26] Artem Lykov and Dzmitry Tsetserukou. Llm-brain: Ai-driven fast generation of robot behaviour tree based on large language model. In *2024 2nd International Conference on Foundation and Large Language Models (FLLM)*, pages 392–397. IEEE, 2024.

[27] Steve Macenski, Francisco Martín, Ruffin White, and Jonatan Ginés Clavero. The marathon 2: A navigation system. In *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 2718–2725. IEEE, 2020.

[28] Andrés Marafioti, Orr Zohar, Miquel Farré, Merve Noyan, Elie Bakouch, Pedro Cuenca, Cyril Zakka, Loubna Ben Allal, Anton Lozhkov, Nouamane Tazi, et al. Smolvlm: Redefining small and efficient multimodal models. *arXiv preprint arXiv:2504.05299*, 2025.

[29] Petter Ögren and Christopher I Sprague. Behavior trees in robot control systems. *Annual Review of Control, Robotics, and Autonomous Systems*, 5(1):81–107, 2022.

[30] Abby O'Neill, Abdul Rehman, Abhiram Maddukuri, Abhishek Gupta, Abhishek Padalkar, Abraham Lee, Acorn Pooley, Agrim Gupta, Ajay Mandlekar, Ajinkya Jain, et al. Open x-embodiment: Robotic learning datasets and rt-x models: Open x-embodiment collaboration 0. In *2024 IEEE International Conference on Robotics and Automation (ICRA)*, pages 6892–6903. IEEE, 2024.

[31] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318, 2002.

[32] George Pu, Anirudh Jain, Jihan Yin, and Russell Kaplan. Empirical analysis of the strengths and weaknesses of peft techniques for llms. *arXiv preprint arXiv:2304.14999*, 2023.

[33] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al. Learning transferable visual models from natural language supervision. In *International conference on machine learning*, pages 8748–8763. PmLR, 2021.

[34] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4510–4520, 2018.

[35] Ozan Sener and Silvio Savarese. Active learning for convolutional neural networks: A core-set approach. *arXiv preprint arXiv:1708.00489*, 2017.

[36] Noam Shazeer. Glu variants improve transformer. *arXiv preprint arXiv:2002.05202*, 2020.

[37] Jonathan Styrud, Matteo Iovino, Mikael Norrlöf, Mårten Björkman, and Christian Smith. Automatic behavior tree expansion with llms for robotic manipulation. In *2025 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1225–1232. IEEE, 2025.

[38] Jianlin Su, Murtadha Ahmed, Yu Lu, Shengfeng Pan, Wen Bo, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding. *Neurocomputing*, 568:127063, 2024.

[39] Gemma Team, Aishwarya Kamath, Johan Ferret, Shreya Pathak, Nino Vieillard, Ramona Merhej, Sarah Perrin, Tatiana Matejovicova, Alexandre Ramé, Morgane Rivière, et al. Gemma 3 technical report. *arXiv preprint arXiv:2503.19786*, 2025.

[40] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[41] Naoki Wake, Atsushi Kanehira, Jun Takamatsu, Kazuhiro Sasabuchi, and Katsushi Ikeuchi. Vlm-driven behavior tree for context-aware task planning. *arXiv preprint arXiv:2501.03968*, 2025.

[42] Homer Rich Walke, Kevin Black, Tony Z Zhao, Quan Vuong, Chongyi Zheng, Philippe Hansen-Estruch, Andre Wang He, Vivek Myers, Moo Jin Kim, Max Du, et al. Bridgedata v2: A dataset for robot learning at scale. In *Conference on Robot Learning*, pages 1723–1736. PMLR, 2023.

[43] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

[44] Xiaohua Zhai, Basil Mustafa, Alexander Kolesnikov, and Lucas Beyer. Sigmoid loss for language image pre-training. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 11975–11986, 2023.

[45] Biao Zhang and Rico Sennrich. Root mean square layer normalization. *Advances in Neural Information Processing Systems*, 32, 2019.

[46] Shengyu Zhang, Linfeng Dong, Xiaoya Li, Sen Zhang, Xiaofei Sun, Shuhe Wang, Jiwei Li, Runyi Hu, Tianwei Zhang, Guoyin Wang, et al. Instruction tuning for large language models: A survey. *ACM Computing Surveys*, 58(7):1–36, 2026.

[47] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. A survey of large language models. *arXiv preprint arXiv:2303.18223*, 1(2), 2023.

[48] Xiwei Zhao, Yiwei Wang, Yansong Wu, Fan Wu, Teng Sun, Zhonghua Miao, Sami Haddadin, and Alois Knoll. Video-to-bt: Generating reactive behavior trees from human demonstration videos for robotic assembly. *arXiv preprint arXiv:2509.16611*, 2025.

[49] Zirui Zhao, Wee Sun Lee, and David Hsu. Large language models as commonsense knowledge for large-scale task planning. *Advances in neural information processing systems*, 36:31967–31987, 2023.

[50] Haotian Zhou, Yunhan Lin, Longwu Yan, Jihong Zhu, and Huasong Min. Llm-bt: Performing robotic adaptive tasks based on large language models and behavior trees. In *2024 IEEE International Conference on Robotics and Automation (ICRA)*, pages 16655–16661. IEEE, 2024.

[51] Brianna Zitkovich, Tianhe Yu, Sichun Xu, Peng Xu, Ted Xiao, Fei Xia, Jialin Wu, Paul Wohlhart, Stefan Welker, Ayzaan Wahid, et al. Rt-2: Vision-language-action models transfer web knowledge to robotic control. In *Conference on Robot Learning*, pages 2165–2183. PMLR, 2023.

A. OXE Sub-datasets

Table 11: OXE sub-datasets used for dataset construction (23 sources).

Sub-dataset	Institution / Description
bridge	Stanford (tabletop manipulation)
fractal20220817_data	Google RT-1 (kitchen tasks)
berkeley_autolab_ur5	UC Berkeley (grasping)
austin_buds	UT Austin (contact-rich tasks)
austin_sailor	UT Austin (articulated objects)
austin_sirius	UT Austin (long-horizon tasks)
cmu_stretch	CMU (mobile manipulation)
dlr_edan_shared_control	DLR (shared control)
dlr_sara_grid_clamp	DLR (grid clamping)
dlr_sara_pour	DLR (pouring)
imperialcollege_sawyer	Imperial College (wrist cam)
jaco_play	Kinova Jaco (play data)
nyu_rot_dataset	NYU (rotation tasks)
stanford_hydra	Stanford (bimanual)
stanford_kuka_multimodal	Stanford KUKA (multimodal)
tokyo_u_lsmo	U. Tokyo (manipulation)
ucsd_kitchen	UCSD (kitchen tasks)
ucsd_pick_and_place	UCSD (pick and place)
utokyo_pr2_opening_fridge	U. Tokyo PR2 (fridge)
utokyo_pr2_tabletop	U. Tokyo PR2 (tabletop)
utokyo_xarm_bimanual	U. Tokyo xArm (bimanual)
utokyo_xarm_pick_place	U. Tokyo xArm (pick-place)
asu_table_top	ASU (tabletop tasks)

Abstract in lingua italiana

Questa tesi esplora l'impiego di Vision-Language Model (VLM) leggeri per generare behavior tree destinati a robot, a partire da un'osservazione visiva della scena e da un'istruzione in linguaggio naturale. Alla luce dei recenti progressi nei modelli vision-language, si indaga se la percezione visiva della scena possa essere sfruttata per produrre piani robotici che tengano conto dello stato osservato dell'ambiente. Sebbene diversi lavori recenti abbiano impiegato modelli vision-language per la pianificazione robotica, nessuno ha affrontato la generazione di behavior tree a partire da modelli multimodali compatti e open-source. Poiché non esiste un dataset che colleghi osservazioni visive e istruzioni a behavior tree eseguibili, viene proposto un metodo per costruirne uno a partire da dataset robotici esistenti. Un modello di grandi dimensioni svolge il ruolo di insegnante in una pipeline di generazione multi-stadio guidata da prompt engineering. Il dataset risultante viene utilizzato per il fine-tuning di VLM con un numero di parametri compreso tra 500 milioni e 4 miliardi, tramite Parameter-Efficient Fine-Tuning (PEFT). Vengono confrontate più architetture VLM, sia nella versione base sia dopo fine-tuning, con modelli closed-source di scala molto superiore. I risultati mostrano che l'approccio proposto colma efficacemente il divario prestazionale richiedendo solo una frazione delle risorse computazionali. La qualità dei behavior tree generati, compatibili con BehaviorTree.CPP, è valutata sia offline, mediante metriche strutturali, di correttezza delle azioni e di sovrapposizione lessicale, sia online, tramite l'esecuzione di attività domestiche in un simulatore embodied di riferimento.

Parole chiave: modelli multimodali, behavior tree, pianificazione robotica, robot learning

Acknowledgements

I would like to express my sincere gratitude to Professor Matteucci for giving me the opportunity to pursue my thesis in such a fascinating area of research. A special thanks to Riccardo and Gianluca for their dedication and guidance throughout these months, always ready to help and push me in the right direction. I also wish to thank Politecnico di Milano for offering me such a unique and prestigious academic journey.

My deepest thanks go to my parents, Silvia and Pier Francesco, for believing in me and investing in my growth from the very beginning. Everything I have achieved is built on the foundation they gave me. I am equally grateful to my grandparents. Knowing that they are here to witness this milestone and share this moment with me means the world to me. A heartfelt thank you to all my relatives and friends who have supported me along the way, and to my beloved dog Keanu, who stayed by my side throughout this work with his unconditional love.

A kiss and a dedication to those family members who are no longer with us. I am certain they are watching from above, proud of what I have accomplished and of the person I am becoming.

Last but not least, this one is for me. I put my heart into this journey, and I have come to realize that giving it my all is the most meaningful way to repay those who stood by me.

Today, the laurel wreath is no longer just a dream.