arXiv:2506.22028v1 [cs.RO] 27 Jun 2025

# LMPVC and Policy Bank: Adaptive voice control for industrial robots with code generating LLMs and reusable Pythonic policies

Ossi Parikka[1] and Roel Pieters[1]

***Abstract*— Modern industry is increasingly moving away from mass manufacturing, towards more specialized and personalized products. As manufacturing tasks become more complex, full automation is not always an option, human involvement may be required. This has increased the need for advanced human robot collaboration (HRC), and with it, improved methods for interaction, such as voice control. Recent advances in natural language processing, driven by artificial intelligence (AI), have the potential to answer this demand. Large language models (LLMs) have rapidly developed very impressive general reasoning capabilities, and many methods of applying this to robotics have been proposed, including through the use of code generation. This paper presents *Language Model Program Voice Control* (LMPVC), an LLM-based prototype voice control architecture with integrated policy programming and teaching capabilities, built for use with Robot Operating System 2 (ROS2) compatible robots. The architecture builds on prior works using code generation for voice control by implementing an additional programming and teaching system, the *Policy Bank*. We find this system can compensate for the limitations of the underlying LLM, and allow LMPVC to adapt to different downstream tasks without a slow and costly training process. The architecture and additional results are released on GitHub (`https://github.com/ozzyuni/LMPVC`).**

## I. Introduction

In the EU and other similar areas, where mass manufacturing is no longer viable in the global market due to factors such as labor costs, industry is looking towards increasingly complex and customizable products as a way to the future. To make this future possible, industrial robots face new, challenging requirements. In particular, these developments have exposed the need for human-robot collaboration (HRC) on a level beyond what current technology allows. [1], [2], [3] Improvements are needed in many areas, one of which is more natural interaction through voice control [4].

Natural language understanding in the context of voice controlled robots may be considered as a problem of *grounding* human-intelligible linguistic concepts in the space of robot actions and perception. Reducing the entirety of human expression into a finite set of actions is quite a challenging problem, but recent advances in language modeling using artificial intelligence (AI) present new avenues for solving it [5], [6], [7], [8]. One such system is **Code as Policies** (CaP) [9], which introduced a method for grounding natural language with code generation, using pre-trained Large Language Models (LLMs). The CaP model performs well for many general tabletop manipulation tasks by interacting with an open vocabulary object detector, but this combination alone is not directly applicable to an industrial setting. Open vocabulary detectors may be fine-tuned to a specific industrial scope if necessary, although adopters might prefer to keep using whatever system they might already have. More importantly, many industrial tasks involve more than manipulation, and an LLM often lacks the necessary knowledge to effectively answer prompts such as "Inspect the part" or "Weld the bottom seam".

Some fragments of a solution are already present in CaP, in the form of the possibility to make use of libraries and, under limited circumstances, previously generated code. Vemprala et. al [10] further hint at the possibility of using the latter to teach the robot progressively more complex tasks. Based on a previous M.Sc. thesis [11], this paper seeks to exploit these possibilites to improve the interactivity and adaptability of industrial voice control systems, as well as to provide tools for easier prototyping with real robots and local LLMs. To this end, we propose the following:

1) **LMPVC:** Open, modular, LLM-based voice control architecture with support for ROS 2.
2) **Policy Bank:** Framework for creating Pythonic policies through voice-based teaching and programming, and automatically prompting an LLM to use them.

These systems create a simple and effective approach to adapt voice control to new domains through learning from descriptive commands, as summarized in Fig. 1.

## II. Related Work

The rapid advancement of artificial intelligence in recent years has opened the floodgates for research on new approaches to natural language human-robot interaction (HRI). The catalyst for this revolution, **Large Language Models**, introduced the world to machines with remarkable capabilities in natural language understanding and language-based general reasoning [12]. However, as the name suggest, LLMs such as ChatGPT predominantly work in the realm of language, with additional work required to bridge their capabilities with other domains.

Initially, the application of LLMs to robotics reflected this divide, with the most natural direction being in the realm of social robotics, where LLMs could provide social interaction with text and speech. More recently, advances in models have made them more desirable for instruction following and task completion purposes as well [13]. The approaches to bridging the language-action cap may be seen as two-fold: On the one hand, dedicated models using technology from LLMs have been developed to work in the robotics

[1]Cognitive Robotics group, Unit of Automation Technology and Mechanical Engineering, Tampere University, 33720, Tampere, Finland; firstname.surname@tuni.fi

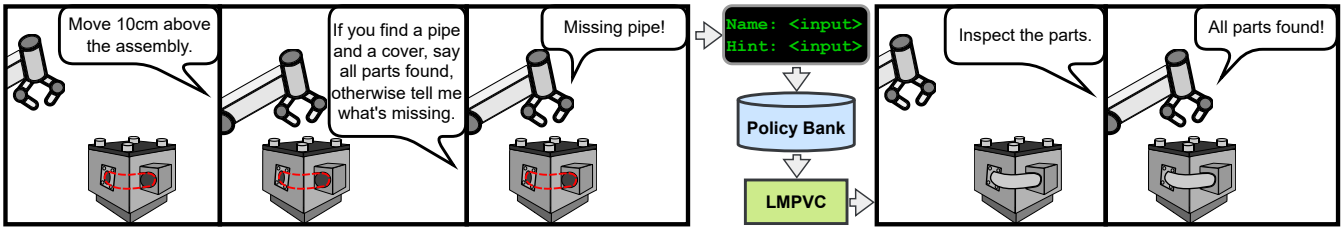


Fig. 1: With the Policy Bank, a user can instruct the robot to complete a task step by step, and save the resulting action code as a new Policy. This Policy is then integrated into the prompting structure of LMPVC, allowing it to be intelligently recalled when needed. Policies can also be programmed, edited and combined, making the system arbitrarily expandable.

domain directly, and on the other, various methods have been discovered to apply LLMs as they are.

In the first case, solutions commonly involve **Vision-Language Models** (VLM) and **Large Vision Models** (LVM), which deploy Transformer architectures similar to LLMs, but adjusted to also process visual data. An earlier solution is CLIPort [14], which combines existing visual and robotics models for an end-to-end solution. By contrast, recent solutions such as RVT-2 [6], SAM2Act [5] and others [15], [16] are fully developed with robotics in mind, and further improve visual precision by employing methods such as using point clouds to render the scene from different angles instead of relying on cameras alone. SAM2Act is currently state of the art in RLBench with an average score of 86.8%, showcasing impressive performance in complex manipulation tasks, but despite these results, it and other similar models are not without disadvantage. Notably, because the models are end-to-end and entirely bespoke, it can be difficult to interact with how they operate, or add functionality which they do not possess. RVT-2 and SAM2Act do both involve few-shot learning capabilities, but this is not an ideal solution to all problems.

On the side of using LLMs more directly, much progress has also been made. Text2Motion [17] uses PDDL to allow an LLM to reason about an environment in terms of symbolic states and known skills. RobotGPT [18] uses ChatGPT to distill a smaller, dedicated model also based on generating executable actions, achieving higher stability but sacrificing some flexibility. **Code as Policies** [9] uses OpenAI Codex to generate robot code from natural language prompts, *grounding* natural language to a set of executable queries and actions. CaP can also leverage its code-based nature by incorporating libraries and hierarchically building behaviour from individual functions. ProgPrompt [8] similarly takes advantage of code generation for task planning. Vemparala *et al.* [10] further investigate the collaborative aspects of using ChatGPT to generate code, allowing the user to request changes to generated tasks. Some, such as PaLM-E [7] also incorporate visual data more directly through encoding. In comparison to VLMs, these LLM based solutions place less emphasis on environmental modeling and precise manipulation, but more on interactivity and the role of the human operator. They also offer more points of entry for programmers, and do not necessarily require the training of task-specific models.

## III. Problem Definition

Recent dedicated robotics models such as SAM2Act are highly effective at precisely grounding natural language in various manipulation tasks, but may not be as easily applicable to specialized industrial use cases. It is not trivial to precisely predict or modify their behaviour, and their bespoke nature requires potential users to commit to a specific system that might not meet all future needs. For industrial actors looking for stability and configurability, this can be a difficult proposition. For the moment, the flexibility and upgradibility advantages of an LLM based solution may be worth the compromises, provided they can be adapted to perfom the task actually required by a given industrial application.

Fortunately, the code-based approach of Code as Policies offers many opportunities for implementing adaptable and specialized behaviour. Instead of purely relying on model performance, we believe a hybrid approach with first party libraries and reused generated code may hold advantages not fully explored by CaP. Furthermore, most existing solutions, including CaP, target SotA cloud-based models, with little regard for latency or data security. In real industrial applications these can be critical, and to better answer this need, the focus of this paper will instead be on LLMs available for local deployment. Finally, we find that such research could benefit from more freely available tooling for rapid prototyping. [9]

## IV. System

The LMPVC architecture is designed to be modular, with individual components being implemented separately using their own interfaces, as shown in Fig. 2. Most components also reside in their own Robot Operating System (ROS) 2 packages and nodes. This structure allows various components to be modified or rebuilt entirely without affecting the rest of the system, provided the existing interfaces of the modules are respected. Some consideration has also been given to uses outside of ROS: Where possible, ROS-specific code is isolated from internal logic, and if desired, can in many cases be replaced entirely by direct calls between Python modules. The object detection module is a special case, as the reference implementation only contains interfaces, which can be adapted to any common object detection model as necessary.

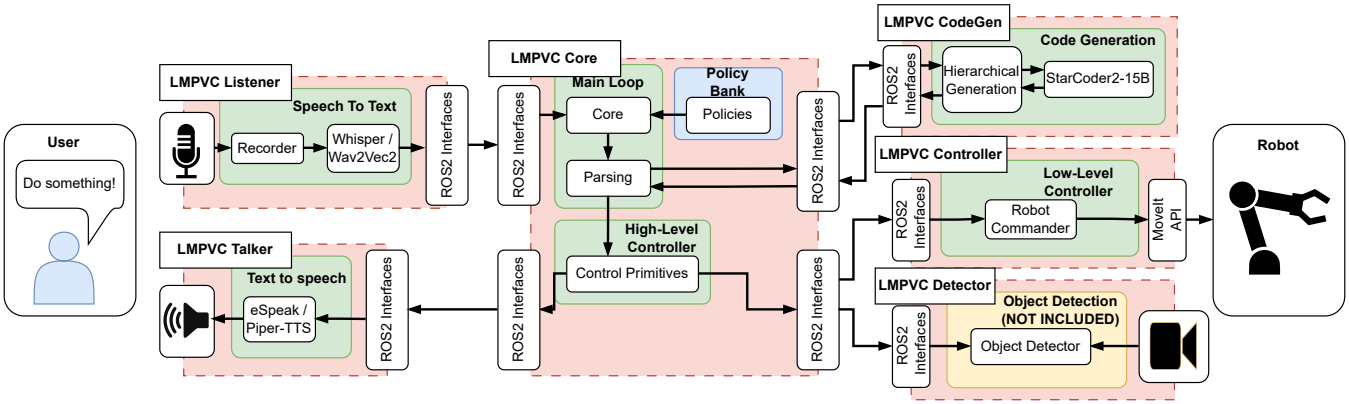


Fig. 2: The modular structure of the LMPVC architecture encourages modification and increases adaptability.

### A. Control flow

At the center of the architecture is the Main Loop, contained within the Core module. In typical usage, the Core first triggers the Listener after detecting a wake event, such as a key press. It waits until the Listener returns transcribed text, or a timeout in case of no detected speech. LMPVC provides two engines by default, Wav2Vec 2.0 [19] and Whisper [20], of which the latter is generally recommended. Any returned text is filtered for manually detected keywords which initiate actions directly, such as "stop", and if none are found, passed to Code Generation. Commands for controlling the teaching capabilities of the Policy Bank are also currently implemented as keywords, although there is no technical reason this could not be done from generated code through dedicated primitives.

Once the command is parsed and necessary code has been generated, the High-Level Controller component of the Core module takes over for execution of the high-level control primitives. Different primitives can involve moving the robot through the Controller module, recovering object poses from the Detector module, or providing user feedback with the Talker module. Optionally, generated code can be temporarily retained as context for future actions.

### B. Code generation

Code Generation is based on Code as Policies: As illustrated in Fig. 3, the model is prompted with a static preamble containing examples, combined with a user prompt formatted as a comment. The LLM then generates an executable response. Importantly, our solution replaces OpenAI Codex with a locally running LLM, StarCoder2 [21]. Liang *et al.* refer to the results of complete generation cycles as Language Model Programs (LMP), a practice we adopt to distinguish between LMPs and the policies handled by the Policy Bank. [9]

Beyond simple control primitives such as *say* or *move*, CaP demonstrates that generated code can depend on both first- and third-party libraries for more complex functionality, such as using NumPy for mathemathical operations. Perhaps most interestingly, because the model works by generating code, it can also make use of its own outputs under certain

```
#define function: say thank you
def say_thank_you(robot):
    robot.say("thank you")
#define function: tell me the first law of robotics
def tell_me_the_first_law_of_robotics(robot):
    robot.say("A robot may not injure a human being
    , or through inaction, allow a human being to
    come to harm.")
#end of function
```

Fig. 3: Basic operating principle of the code generation.

circumstances. In CaP, this is used in two ways: hierarchical generation and LMP composition. We implement hierarchical generation to improve the performance of code generation, particularly in complex tasks. This works by prompting the model to call *undefined* functions, meaning it can freely decide how a given task should be divided. Then, undefined functions are detected, and the model is called again to define them after the fact. This bears some resemblance to Chain of Thought prompting [22], but Liang *et al.* find it can work even better in programming benchmarks such as HumanEval. LMP composition is not directly implemented they way it is used in CaP: LMPVC does not use different prompts to separately process parts of the voice command. Rather, the concept is a part of the inspiration for the Policy Bank. [9]

### C. Policy Bank

To extend the capabilities inherited from CaP, we propose the Policy Bank. Combining parts of the LMP composition and use of libraries demonstrated by Liang *et al*, the Policy Bank is asolution for dynamically registering new Pythonic policies with LMPVC, and automatically incorporating them into the prompts sent to Code Generation. A programmer can simply provide a code file containing import statements, one or more function definitions, and a short common sense hint, separated by special comment blocks. These files are then registered with the system through a JSON configuration file, which the Policy Bank reads to determine which files should be included when building the prompt for Code Generation.

Additionally, the Policy Bank interacts with the Core's keyword system to implement a voice-based teaching

method. When the user says a specific keyword (by default: "record policy"), Core enables an additional step to its main loop: Every time a result is received from Code Generation, it is also passed back to the Policy Bank through a special recording method. This code is then slightly reformatted in order to be saved as a part of a new policy. Once the user indicates the new policy is ready with another keyword (by default: "save policy"), the Core asks the user for a name and a hint, which are also passed to the Policy Bank. With this information, a new policy can be created, and incorporated into the system in exactly the same way as a handwritten one would be. These learned policies are also saved to disk in the same text format, allowing further editing by a programmer.

## V. Results

The primary goals of our experiments are to (i) show how LMPVC as a whole performs with common natural language voice commands, (ii) demonstrate how the *Policy Bank* can be used to apply CaP concepts to specialized industrial use cases, and (iii) evaluate how this combination performs in practical tasks. Experiments V-B, V-D and V-E are mostly from the original M.Sc. thesis [11], with additional ones conducted for this paper.

### A. Experimental setup

The setup used for practical experiments consists of a **Franka Emika Panda** 7-DOF robot with a hand-mounted **Intel RealSense D435** camera, **three PCs** running the various ROS components, and an **industrial pump** used for collaborative assembly tasks. PC 1 is dedicated to controlling the robot, and needs no special hardware. PC 2 performs object detection using an Nvidia GTX 1080Ti 11GB GPU, while PC 3 contains an Nvidia RTX 3080 10GB GPU powering both StarCoder2-15B, and Whisper Small. The object detection implementation used for testing is based on the one developed by Ojanen for a thesis project [23].

### B. Direct robot control

Starting simply, **LMPVC can respond to various simple directives**. This includes parsing common units of measurement, based on inferring from examples that the base unit of the coordinate system is a meter. Code generation is also **resilient to some mistakes**. In the case of *"Two... twenty centimeters to the left"*, the user's hesitation is transcribed by Whisper as *"twotwenty"*. Fortunately, this is resolved correctly when it comes to setting the coordinate in code.

```
USER: Two... twenty centimeters to the left.
LMPVC:
def twotwenty_centimeters_to_the_left(robot):
    waypoint = robot.get_pose()
    waypoint.position.y -= 0.2
    robot.add_waypoint(waypoint)
    robot.go()
```

Similarly, it **can resolve ambiguity and imprecision** by making reasonable assumptions, such as interpreting moving *a little* as 5 centimeters. Assumptions can also be directed by including some of them in the examples.

```
USER: Move a little down.
LMPVC:
def move_a_little_down(robot):
    waypoint = robot.get_pose()
    waypoint.position.z -= 0.05
    robot.add_waypoint(waypoint)
    robot.go()
```

As with CaP, **sensors can be integrated** through their own Python interfaces. Since our solution uses a simple object detection solution, our examples direct code generation to only pass names of objects, which are then mapped to classes and converted to poses. The high-level interface is defined by the prompt, simplifying integration of different sensors.

```
USER: Can you see the big bolt?
LMPVC:
def can_you_see_the_big_bolt(robot):
    (big_bolt_pose, big_bolt_found) = robot.find('
    big_bolt')
    if(not big_bolt_found):
        robot.say("Can't find the big bolt!")
    else:
        robot.say("Found the big bolt!")
```

The most common occurrence of hierarchical generation within LMPVC is in the context of **multi-stage commands**. Separate functions are often called for sub-tasks, and subsequently generated. The output processing of our solution always places the top-level function at the bottom to make generated code more readable.

```
USER: Move a little down, and then draw a circle
    with radius 35 millimeters.
LMPVC:
def draw_circle_of_radius_35_millimeters(robot):
    waypoints = 25
    for i in range(waypoints + 1)):
        waypoint = robot.get_pose()
        waypoint.position.x += (0.035 * math.cos
    ((((2 * math.pi) * i ) / 25)))
        waypoint.position.y += (0.035 * math.sin
    ((((2 * math.pi) * i ) / 25)))
        robot.add_waypoint(waypoint)
    robot.go()

def move_a_little_down(robot):
    waypoint = robot.get_pose()
    waypoint.position.z -= 0.05
    robot.add_waypoint()
    robot_go()

def move_a_little_down_and_then_draw_a_circle...
    _with_radius_35_millimeters(robot):
    move_a_little_down(robot)
    draw_a_circle_of_radius_35_millimeters(robot)
```

### C. Context and conversational modification

By retaining earlier code as context, LMPVC **can react to or modify previous behaviour**. For example, context can influence how ambiguities are solved, making it possible to use commands like *"Just a bit more"* immediately after the previous *"Move a little down"*. Some more advanced modifications are also possible, such as directing the robot to regenerate an action with different parameters. This functionality is reminiscent of examples Vemprala *et al.* [10] use to discuss the potential of the conversational capabilities of ChatGPT.

```
USER: Draw a small circle.
LMPVC:
def draw_a_small_circle(robot):
    waypoints = 25
    for i in range(waypoints + 1)):
        waypoint = robot.get_pose()
        waypoint.position.x += (0.05 * math.cos
    ((((2 * math.pi) * i ) / 25)))
        waypoint.position.y += (0.05 * math.sin
    ((((2 * math.pi) * i ) / 25)))
        robot.add_waypoint(waypoint)
    robot.go()
USER: Double the radius.
LMPVC:
def double_the_radius(robot):
    waypoints = 25
    for i in range(waypoints + 1)):
        waypoint = robot.get_pose()
        waypoint.position.x += (0.1 * math.cos((((2
     * math.pi) * i ) / 25)))
        waypoint.position.y += (0.1 * math.sin((((2
     * math.pi) * i ) / 25)))
        robot.add_waypoint(waypoint)
    robot.go()
```

The StarCoder2 LLM core used by this version of LMPVC is unquestionably more limited, but it is interesting that even far smaller models not fine-tuned for conversation can still be used in this manner.

### D. Policy bank

**Policies may be programmed by hand** to explicitly define desired behaviour. This is convenient for users with programming experience, and allows adding functionality which the LLM struggles with or can't access. In the following example, a handover policy is created, using both object detection and movement commands through the *High Level Controller*.

```
import time
#BODY
def handover(robot):
    # Find predefined handover pose
    (handover_pose, success) = robot.find('handover
    ')
    # Move to it
    robot.add_waypoint(handover_pose)
    robot.go()
    robot.say("In handover position, releasing in
    two seconds!")
    start = time.time()
    end = time.time()
    while(end - start < 2.0):
        time.sleep(0.1)
        end = time.time()
    robot.open_hand()
# HINT
# define function: give me the held item
def give_me_the_held_item(robot):
    handover(robot)
# end of function
```

Going forward, prompts to *Code Generation* will now include the section below the *#HINT*-tag, allowing the function *handover(robot)* to appear in generated code when appropriate. When they exist, import statements above the *#BODY*-tag are included as well.

**Policies can be taught** by recording generated policy output as the user guides the robot to complete a task with descriptive commands. Below is an automatically generated policy in response to the command *"If you see a pipe and a cover, say all parts found, otherwise tell me what's missing."*, as saved to disk by the *Policy Bank*.

```
# Generated code
def if_you_see_a_pipe_and_a_cover_say_all_parts...
    _found_otherwise_tell_me_whats_missing(robot):
    (pipe_pose, pipe_found) = robot.find('pipe')
    if (not pipe_found):
        robot.say("Can't find the pipe!")
    else:
        (cover_pose, cover_found) = robot.find('
    cover')
        if (not cover_found):
            robot.say("Can't find the cover!")
        else:
            robot.say('All parts found!')
# Added by Policy Bank based on user input
def parts_check(robot):
    if_you_see_a_pipe_and_a_cover_say_all_parts...
    _found_otherwise_tell_me_whats_missing(robot):
# HINT
# define function: inspect all parts
def inspect_all_parts(robot):
    parts_check(robot)
# end of function
```

**Policies can also include multiple commands** and even **references to existing policies**. Here, a policy is constructed from three commands: *"Find the assembly and move thirty centimeters above it."*, *"Check parts."* and *"Check bolts."*. The first command generates new behavior, while the other call existing policies, including a previous example.

```
# Generated code based on 1st command
def find_the_assembly_and_move_thirty_...
    centimeters_above_it(robot):
    (assembly_pose, assembly_found) = robot.find('
    assembly')
    if (not assembly_found):
        robot.say("Can't find the assembly!")
    else:
        waypoint = robot.get_pose()
        waypoint.position.x = assembly_pose.
    position.x
        waypoint.position.y = assembly_pose.
    position.y
        waypoint.position.z = (assembly_pose.
    position.z + 0.3)
        robot.add_waypoint(waypoint)
        robot.go()
# Generated code based on 2nd command
def check_parts(robot):
    parts_check(robot)
# Generated code based on 3rd command
def check_bolts(robot):
    bolts_check(robot)
# Added by Policy Bank based on user input
def full_check(robot):
    find_the_assembly_and_move_thirty...
    _centimeters_above_it(robot)
    check_parts(robot)
    check_bolts(robot)
# HINT
# define function: do a full inspection
def do_a_full_inspection(robot):
    full_check(robot)
# end of function
```

Multiple commands and reusing existing policies combine to make the *Policy Bank* a powerful tool for extending the capabilities of the LLM with robust and potentially very complex behaviour blocks. With the further addition of hand-

Fig. 4: Experimental assembly of a pump, demonstrating practical human-robot collaboration aided by the Policy Bank. [11]

written components and the ability to fix or modify the blocks later, the resulting architecture becomes highly customizable and less reliant on the capabilities of LLMs alone. If desired, the complete policies could even be triggered by some other, more robust voice control method, leaving the LLM only as a teaching tool and secondary control method.

### *E. Pump assembly*

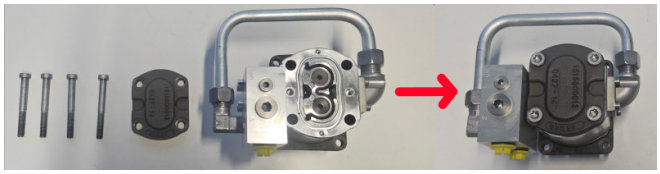

Fig. 5: The goal of the collaborative task.

To demonstrate how LMPVC and the Policy Bank can enable HRC in practice, we conduct an experimental assembly task: A human and a robot arm collaboratively assembling a pump. For brevity, the experiment focuses on the last step of this assembly, which consists of **installing the top cover** and **securing it with four screws**, as shown in Fig. 5. This task makes use of the policies created in the previous section, along with some more general commands. Screenshots of the experiment are shown in Fig. 4, and a full video is provided on GitHub.

First, the user asks the robot to *"Do a full inspection"*, **causing code generation to call the generated policy** from the end of the previous section. Note that this policy also checks for the presence of a *pipe*, which at this stage is already attached to the side of the pump. After the policy is executed, the robot responds with *"Cover not found, everything secured."*, and the user promptly places the cover on top of the assembly. Here, the **context sensitivity** of LMPVC becomes useful, as the user can trigger the inspection policy once more by simply saying *"Check again."*. After listening to the updated response *All parts found, missing bolts."*, the user moves the robot out of the way with *"30cm to the left."*.

While working on the assembly with both hands, the user can then direct the robot to perform a parallel task, sending it to *"Pick up the big bolt."*. This **activates the hand-written policy** *pick*, which takes care of how the robot approaches the target and picks it up. Similarly, the following command *"Give it to me."* calls the *handover* policy shown in the earlier

examples. Finally, the user calls the inspection policy two more times with and without context, making the corrections the robot requests until the assembly is complete.

This experiment demonstrates that the Policy Bank can allow LMPVC to adapt to a practical HRC situation where purely AI-based alternatives might struggle. The inspection functionality was achieved through voice-based teaching, demonstrating how users with limited programming experience can make adaptations of their own.

### F. Code generation latency

An additional concern for the practical viability of LMPVC is the time it takes for the local LLM to generate code. If the use of an LLM leads to excessive command latency, user experience could be significantly impacted. To quantify the impact, we measure the latency of the code generation component for a set of 50 commands ranging from movement and manipulation to more general knowledge-based tasks. A handful examples of these commands are listed in table I, the rest can be found on GitHub with the other experiment materials.

| N | Command |
|---|---|
| 11. | Rotate 33 degrees clockwise. |
| 14. | Draw an ellipse with radii five and four centimeters. |
| 18. | What's the cube root of three rounded to two decimal places? |
| 31. | Move a little to the left and then right, repeat three times. |
| 50. | What tool should I use to tighten bolts to correct torque? |

TABLE I: Examples of the commands used for the latency and accuracy measurements

Latency is measured 10 times for each command in the set, using a script with a short wait time between runs to avoid congestion. The results will vary between hardware and software configurations, and should not be taken as absolute values. Rather, the intent is to present some representative samples on relatively modern but still accessible hardware. This experiment is performed on an Nvidia RTX 3080 10GB, running StarCoder2-15B in GGUF IQ4_XS format [24]. The model is loaded with Llama.cpp [25], using FlashAttention-2 [26] for optimal performance.

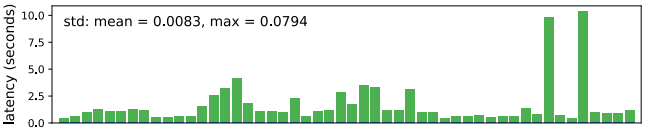


Fig. 6: Mean command latency in the test set.

The mean latency results for each command can be seen in Fig. 6, along with aggregate information of the standard deviations between runs for readability. It is evident that the results are consistent, with longer latencies resulting in the highest variability. More importantly, the mean latencies are quite low, below 1.5 seconds for majority of the test cases, and just over 4 seconds at maximum for successful commands. Latencies generally depend on the amount of code required to accomplish the task, corresponding to the *tokens/s* performance of the model, and in this case two of the failures generated very long responses, resulting in higher latency. Even accounting for the minor additional latency from Whisper not included in this test, the successful results compare quite well to a simple, non-AI based voice control solution [27]. This supports the viability of LLMs for real-time voice control, which we believe is promising for industrial applications.

### G. Command completion performance

Due to the specialized focus of our work, LMPVC is not well suited to benchmarks such as RLBench, which primarily test performance in complex, open-ended manipulation tasks. In the absence of suitable standardized tools, we use the same 50 commands from the latency test to quantify reliability in a more appropriate scope. We choose test commands which are possible to complete with the tools the system has access to, with varying complexity. In order to make the results more meaningful, all commands were chosen and then tested at once. To collect results, we analyze the generated code for each command, calculate the success rate, and make note of relevant failures.

With this methodology, LMPVC successfully completes 39 out of 50 tasks, resulting in an overall success rate of **78%**. The system generally does well in tasks involving moving the robot arm, calling policies and reacting to object detection, including making use of the received poses when required. It can also provide some advice, for example, the question *"What tool should I use to tighten bolts to correct torque?"* is appropriately answered with *"I would use a torque wrench to tighten bolts to the correct torque"*.

One notable failure involves the use of methods not present on the high-level controller. When asked to *"press the red button"* the non-existent method *robot.set_digital_output(0, True)* is called, indicating that the system failed to construct a pressing action from the known object detection and movement components. Similarly, something interesting happens in response to the command *"Build a house"*. This is quite clearly an out-of-scope instruction, without any relevant policies to call. Often, the default answer of the model is to repeat the unknown task back at the user, but in this case, it attempts to build a house by picking and placing a variety of components, such as *"bricks"*, *"roof"* and *"chimney"*. These examples highlight how the flexibility of AI can sometimes be an issue with unexpected commands.

In two further cases, the robot struggles to parse the phrase "back the other way", failing to move in the correct contextual direction. The LLM appears to often not understand the difference between "back" in the context of the previous answer, and "back" as in "backwards" in world space, although other contextual commands usually work quite well. The remaining six failures mostly involve incorrect information retrieval, and two cases of unexpected output on tasks similar to other, successful ones. These issues are more directly indicative of the limitations of a small LLM, although the 4-bit quantization of the model may also be a contributing factor.

## VI. Discussion

Although LMPVC is at a disadvantage when relying purely on code generation, with the Policy Bank it can adapt to specialized industrial use cases not presently covered by Code as Policies [9] or other similar code generation solutions [8]. Conversational behavior and contextual awareness reminiscent of the work of Vemprala *et al.* [10] can also be achieved with adjustments to prompting, despite the small model and lack of fine-tuning. Our solution additionally benefits from increased data security and low latency compared to works relying on larger cloud-based models, with results similar to certain non-AI alternatives [27].

We also identify several limitations, most commonly related to the capabilities of the local LLM, as well as the simplifications made to object detection and the architecture in general. Despite the advantages of augmenting live generation with existing policy, the system would still benefit from improved generation performance. While research on improved models continues to advance rapidly, we also suggest further investigating strategies for structured problem solving, such as LMP Composition [9], Chain of Thought and ReAct, as well as augmentation techniques such as Retrieval-Augmented Generation [28]. The Policy Bank itself could additionally benefit from a robust solution for policies to send and receive data from each other and external processes, enabling more dynamic sequences of actions.

## VII. Conclusion

In this paper, we proposed LMPVC, an open voice control architecture built for ROS 2, and Policy Bank, a framework for intelligently integrating voice-based teaching and high-level Python programming with LLM-based natural language robot control. In experiments involving individual commands, policy usage and an assembly task, this combination achieves promising performance using very limited hardware resources. While our solution does not compete with SotA methods in benchmarks measuring complex manipulation, it has potential advantages in industrial applications and other use cases requiring data security, low latency and adaptability to specialized tasks.

## Acknowledgements

This project has received funding from the European Union's Horizon Europe research and innovation programme under grant agreement no. 101135708 (JARVIS).

## References

[1] A. K. Inkulu *et al.*, "Challenges and opportunities in human robot collaboration context of industry 4.0 - a state of the art review," *Industrial robot*, vol. 49, no. 2, pp. 226–239, 2022.

[2] S. Kumar, C. Savur, and F. Sahin, "Survey of human–robot collaboration in industrial settings: Awareness, intelligence, and compliance," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 51, no. 1, pp. 280–297, 2021.

[3] W. Bauer, M. Bender, M. Braun, P. Rally, and O. Scholtz, "Lightweight robots in manual assembly–best to start simply," *Frauenhofer-Institut fur Arbeitswirtschaft und Organisation IAO, Stuttgart*, vol. 1, 2016.

[4] M. Marge *et al.*, "Spoken language interaction with robots: Recommendations for future research," *Computer Speech &; Language*, vol. 71, p. 101255, 2022. [Online]. Available: http://dx.doi.org/10.1016/j.csl.2021.101255

[5] H. Fang *et al.*, "Sam2act: Integrating visual foundation model with a memory architecture for robotic manipulation," 2025. [Online]. Available: https://arxiv.org/abs/2501.18564

[6] A. Goyal, V. Blukis, J. Xu, Y. Guo, Y.-W. Chao, and D. Fox, "Rvt-2: Learning precise manipulation from few demonstrations," 2024. [Online]. Available: https://arxiv.org/abs/2406.08545

[7] D. Driess *et al.*, "Palm-e: An embodied multimodal language model," in *Proceedings of the 40th International Conference on Machine Learning, ICML 2023*, vol. 202, 2023, pp. 8469–8488.

[8] I. Singh *et al.*, "Progprompt: Generating situated robot task plans using large language models," in *2023 IEEE International Conference on Robotics and Automation (ICRA)*, 2023, pp. 11 523–11 530.

[9] J. Liang *et al.*, "Code as policies: Language model programs for embodied control," in *2023 IEEE International Conference on Robotics and Automation (ICRA)*, vol. 2023-. IEEE, 2023, pp. 9493–9500.

[10] S. H. Vemprala, R. Bonatti, A. Bucker, and A. Kapoor, "Chatgpt for robotics: Design principles and model abilities," *IEEE access*, vol. 12, pp. 55 682–55 696, 2024.

[11] O. Parikka, "Voice control of industrial robots with large language models trained on code," Tampere University, 2024. [Online]. Available: https://urn.fi/URN:NBN:fi:tuni-2024112910646

[12] T. Teubner, C. Flath, C. Weinhardt, W. Aalst, and O. Hinz, "Welcome to the era of chatgpt et al.: The prospects of large language models," *Business & Information Systems Engineering*, vol. 65, 03 2023.

[13] C. Zhang, J. Chen, J. Li, Y. Peng, and Z. Mao, "Large language models for human–robot interaction: A review," *Biomimetic Intelligence and Robotics*, vol. 3, no. 4, p. 100131, 2023.

[14] M. Shridhar, L. Manuelli, and D. Fox, "Cliport: What and where pathways for robotic manipulation," in *Proceedings of the 5th Conference on Robot Learning, CoRL 2021*, vol. 164, 2021, pp. 894–906.

[15] J. Duan *et al.*, "Manipulate-anything: Automating real-world robots using vision-language models," 2024. [Online]. Available: https://arxiv.org/abs/2406.18915

[16] M. Shridhar, L. Manuelli, and D. Fox, "Perceiver-actor: A multi-task transformer for robotic manipulation," in *Proceedings of the 6th Conference on Robot Learning, CoRL 2022*, vol. 205, 2023, pp. 785–799.

[17] K. Lin, C. Agia, T. Migimatsu, M. Pavone, and J. Bohg, "Text2motion: from natural language instructions to feasible plans," *Autonomous Robots*, vol. 47, no. 8, p. 1345–1365, Nov. 2023. [Online]. Available: http://dx.doi.org/10.1007/s10514-023-10131-7

[18] Y. Jin, D. Li, Y. A, J. Shi, P. Hao, F. Sun, J. Zhang, and B. Fang, "Robotgpt: Robot manipulation learning from chatgpt," *IEEE Robotics and Automation Letters*, vol. 9, no. 3, pp. 2543–2550, 2024.

[19] A. Baevski, H. Zhou, A. Mohamed, and M. Auli, "wav2vec 2.0: A framework for self-supervised learning of speech representations," in *34th Conference on Neural Information Processing Systems, NeurIPS 2020*, vol. 2020-, 2020.

[20] A. Radford *et al.*, "Robust speech recognition via large-scale weak supervision," in *Proceedings of the 40th International Conference on Machine Learning, ICML 2023*, vol. 202, 2023, pp. 28 492–28 518.

[21] A. Lozhkov *et al.*, "Starcoder 2 and the stack v2: The next generation," 2024. [Online]. Available: https://arxiv.org/abs/2402.19173

[22] J. Li, G. Li, Y. Li, and Z. Jin, "Structured chain-of-thought prompting for code generation," *ACM transactions on software engineering and methodology*, vol. 34, no. 2, pp. 1–23, 2025.

[23] R. Ojanen, "Human-robot collaboration by speech in an industrial assembly task," Tampere University, 2024. [Online]. Available: https://urn.fi/URN:NBN:fi:tuni-202409188790

[24] Mradermacher, "mradermacher/starcoder2-15b-i1-gguf," 2024. [Online]. Available: https://huggingface.co/mradermacher/starcoder2-15b-i1-GGUF

[25] G. Gerganov, "llma.cpp," 2024. [Online]. Available: https://github.com/ggerganov/llama.cpp

[26] T. Dao, "Flashattention-2: Faster attention with better parallelism and work partitioning," in *12th International Conference on Learning Representations, ICLR 2024*, 2024.

[27] W. Kaczmarek, J. Panasiuk, S. Borys, and P. Banach, "Industrial robot control by means of gestures and voice commands in off-line and on-line mode," *Sensors*, vol. 20, no. 21, 2020. [Online]. Available: https://www.mdpi.com/1424-8220/20/21/6358

[28] H. Kang, M. Ben Moussa, and N. M. Thalmann, "Nadine: A large language model-driven intelligent social robot with affective capabilities and human-like memory," *Computer animation and virtual worlds*, vol. 35, no. 4, 2024.