# **Towards Improving Large Language Models' Planning Capabilities on WoT Thing Descriptions by Generating** Python Objects as Intermediary Representations<sup>\*</sup>

Lukas Kinder*<sup>1</sup>* , Tobias Käfer*<sup>1</sup>*

*1 Institute AIFB, Karlsruhe Institute of Technology (KIT), Kaiserstr. 89, 76133 Karlsruhe, Germany*

#### **Abstract**

This paper presents a novel method for plan generation with Large Language Model(LLM). We propose utilizing Web-of-Thing Thing Descriptions (WoT TD) to inform the LLM about available devices for interaction. We investigate a novel pipeline in which: (1) The WoT TD gets translated into a Python class, (2) The task in natural language gets translated into code that interact with this class, (3) The generated code gets executed to obtain the plan. We evaluate our approach featuring 6 different state of the art LLMs. We get performance improvements of up to 12%, when comparing our approach against existing LLM-based planning methods. Furthermore, we explore the influence of different aspects of WoT TDs on planning capabilities. This research paves the way towards more potent LLM planning models by introducing Python classes as intermediaries, simplifying the final planning task while leveraging standardized and accessible domain knowledge provided by WoT TDs.

#### **Keywords**

Planning, Large Language Models, Semantic Web

## **1. Introduction**

Robotic systems in households and industry that interact with humans not only need to be able to interpret natural language, but can also benefit from general world knowledge. For this reason, recent studies utilised Large Language Models (LLMs) that are able to perform planning by generating sequences of elemental actions that satisfy some given goal [\[1,](#page--1-0) [2,](#page--1-1) [3,](#page--1-2) [4,](#page--1-3) [5,](#page--1-4) [6\]](#page--1-5). However, even though LLMs have been trained on vast amounts of web knowledge, in a specific environment they may lack the required domain-knowledge. For example, robots interacting with devices need to know exactly what the device does and which actions can be performed with it, in order to include it in a plan.

Providing a LLM with domain-specific knowledge is usually done by a human expert through detailed instruction or specialized training data [\[5,](#page--1-4) [1,](#page--1-0) [7\]](#page--1-6). However, this can be expensive and the expert needs to be consulted every time a new device is added to the system. To address this issue, we consider a novel case in which the domain knowledge of devices is provided by Web-of-Things Thing Descriptions (WoT TD). WoT TDs have been a W3C recommendation since 2020, as a human readable and machine interoperable interface for devices[\[8\]](#page--1-7). They provide information about some properties of the device and actions that can be performed with it. This makes them a useful resource for LLMs to generate plans that involve these devices. A new device can be easily introduced to the system by providing it with the corresponding WoT TD. However, these descriptions are still not detailed enough to explain all the workings of the device with requirements and outcomes of actions. Therefore it is challenge for the LLM to use its general world knowledge to fill these information gaps.

Tasking a LLM to generate a plan requires it to divide the problem into multiple elemental actions that can be executed one by one. This is a process similar to coding a program, in which elemental instructions are chained to fulfill an overall goal. Inspired by existing literature, we translate the plan generation by the LLM into a code generation problem  $[1, 9, 4]$  $[1, 9, 4]$  $[1, 9, 4]$  $[1, 9, 4]$  $[1, 9, 4]$ . The idea is that if solving a task requires a LLM to have the mindset of a programmer, it may be best to also translate it into a programming task  $[10]$ .

[0009-0008-8275-1284](https://orcid.org/0009-0008-8275-1284) (L. Kinder); [0000-0003-0576-7457](https://orcid.org/0000-0003-0576-7457) (T. Käfer)

CEUR-WS.org/Vol-3749/akr3-06.pdf

*First International Workshop on Actionable Knowledge Representation and Reasoning for Robots, May 27, 2024, Hersonissos, Crete*  $\bigcirc$ [lukas.kinder@kit.edu](mailto:lukas.kinder@kit.edu) (L. Kinder); [tobias.kaefer@kit.edu](mailto:tobias.kaefer@kit.edu) (T. Käfer)

<sup>©</sup> 2024 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).



**Figure 1:** The pipeline of our approach. First, the (simplified) Web of Things Thing Description gets translated into a Python class. The actions that can be performed with the device become functions of this class. Secondly, the task in natural language gets translated into code that is calling functions of this Python class. Finally, all the generated code gets executed to generate the plan. The color coding in later figures maps them to this pipeline.

In our approach a LLM first translates the WoT TD jsonld-file of a device into a Python class. The functions of the Python class correspond to the actions that can be performed by the device. Afterwards, the LLM tries to accomplish a task in natural language by calling functions of this class. The plan is generated when running all the generated code. In a real world scenario this plan can be mapped to the real device using the WoT TD scripting API provided by the W3C[\[11\]](#page-9-0).

We evaluate different approaches using our own benchmark with 123 tasks involving 29 devices, including real-world WoT TDs. For each device we use a simulator to verify that a plan was successful. Our contributions in this paper are:

- We introduce a novel method that utilises WoT TDs as domain knowledge for LLM planning systems, by writing representations of devices as Python classes.
- We evaluate our approach against the state of the art models Progprompt[\[1\]](#page-8-0) and SayCan[\[5\]](#page-8-1).
- We study how different augmentations of the Web of Things Thing Descriptions influence the performance of our approach.

All the code, the benchmark and the results of our experiments are available on the projects Git repository<sup>[1](#page-1-0)</sup>.

## **2. Related Work**

This section will provide an overview of existing work, starting with traditional planning methods and how LLMs have been used in recent years for planning tasks. Finally, we give an overview of how LLM have been used as code generators.

## **2.1. Traditional Planning:**

Traditionally, symbolic approaches have been used for robotic planning tasks such as STRIPS or SHOP [\[12,](#page-9-1) [13,](#page-9-2) [14\]](#page-9-3). Since its introduction in 1998 the Plan Domain Definition Language (PDDL) and extensions of it have been predominantly been used for planning [\[15,](#page-9-4) [16,](#page-9-5) [17,](#page-9-6) [18,](#page-9-7) [19\]](#page-9-8). These approaches usually require a fully observed and unambiguously defined environment. Formal information of the state of the world and the possible actions must be provided. This makes these approaches unsuitable for planning based on WoT TDs as they do not contain information about preconditions and outcomes of actions.

<span id="page-1-0"></span><sup>1</sup> [https://gitlab.kit.edu/lukas.kinder/plannning\\_with\\_thing\\_descriptions\\_akr3](https://gitlab.kit.edu/lukas.kinder/plannning_with_thing_descriptions_akr3)

#### **2.2. LLMs as Planners:**

LLMs have been a promising approach for robotic planning, if the problem at hand benefits from general world knowledge. The type of useful general information they provide could be for example: "Food can usually be found in the fridge" or "In order to use the device I probably have to turn it on" etc. A challenge with plan generation by the means of a LLM is that even though they are usually able to produce good sounding high-level plans, they struggle to translate the plan into a sequence of executable elemental actions [\[20,](#page-9-9) [3\]](#page-8-2). This limitation arises because the data they are trained on may lack the detailed domain-specific knowledge necessary for generating a grounded plan.

In early work, LLMs were tasked with directly producing action sequence as plans that satisfy a task description in natural language. Grounding the model to the domain-specific actions can be done for example by pre-training [\[21,](#page-9-10) [7\]](#page-9-11). However, this requires a lot of training data and computational scalability. In the model SayCan, grounding is achieved the LLM scores the likelihood for action being the next step in a plan [\[5\]](#page-8-1). But this is only feasible, if it is possible to enumerate all possible actions.

One method to mitigate the limitations of LLMs is to refrain from allowing the LLM to generate the plan entirely, instead letting it make calls to external solvers [\[22,](#page-9-12) [23\]](#page-9-13). This strategy can be applied in the planning domain, by letting the LLM translate a problem into PDDL and let a symbolic solver determine the correct plan [\[24\]](#page-9-14). This however requires precise definition of a formal world-model by the LLM.

#### **2.3. LLMs as Code Generators:**

With their training data usually containing code data, LLMs can be reasonable code generators [\[25,](#page-9-15) [26,](#page-9-16) [27\]](#page-10-0). Therefore it may be beneficial rewrite a problem into a code generation task. Writing code requires to determine small individual steps that together achieve some high level goal. This is the type of thinking that is often required for planning or structured reasoning. Therefore, a LLM may perform better in the familiar framework of a code generation task, if presented with tasks that requires a solution made up by many elemental actions [\[28\]](#page-10-1).

In the planning domain, this has the advantage that the generated code can often be more easily mapped to the environment. For example by letting a model use template-functions that map to elemental actions [\[1,](#page-8-0) [4,](#page-8-3) [29,](#page-10-2) [30\]](#page-10-3) or by letting it use predefined class structures that translate well into real-world concepts of interest [\[9\]](#page-9-17). The generated code can also be used to solve a variety of reasoning problems. This method extends the traditional Chain of Thought approach [\[31\]](#page-10-4), because the code can be compiled and executed to calculate an answer [\[10,](#page-9-18) [32,](#page-10-5) [33\]](#page-10-6).

## **3. Approach**

We will describe the task of the model and the challenges with it in Section [3.1.](#page-2-0) Afterwards, Section [3.2](#page-3-0) explains how the WoT TD is translated into a Python class and Section [3.3](#page-4-0) explains how the task in natural language gets translated into Python . Finally, we explain in Section [3.4](#page-4-1) how the entire code is executed in order to generate the plan.

#### <span id="page-2-0"></span>**3.1. Task Description**

The model has to generate a plan in the form of a sequence of actions that can be performed with a device. The plan should fulfill a task that we give the model in natural language. Since the task description it is in natural language, the model needs to interpret what it should do. Examples of task descriptions are "Turn on the lights.", "Play me a song on the radio with maximum volume." or "Today I have five friends coming over. Make coffee for all of us".

We also give the model a WoT TD, that is has to use to extract domain knowledge about the device. An example of a WoT TD is shown in Figure [2a.](#page-3-1) Generally a WoT TD gives information about three interaction affordances: *Properties*, *Actions* and *Events*. A *Property* expresses the state of the device. They can be retrieved and possibly updated. *Actions* can be called in order to invoke an operation of the device. This can for example cause the state of the device to change. *Events* describe a (random) incident with a device that has an impact on it (For example a lamp may overheat). In this paper we will ignore *Events* for our evaluations, as environmental feedback is out of the scope for this paper. The action descriptions in the WoT TD are not detailed enough to explain the entire inner working of the corresponding device. Usually the outcome of an action is not formally given and sometimes an action has requirements that are not indicated. Furthermore, the workings of a device may be influenced by hidden variables not mentioned in the WoT TD.

## <span id="page-3-0"></span>**3.2. Translating the Thing Description into a Python Class**

```
{ "title": "Coffee Machine",
  "properties": {
    "water": {
      "description": "Illustrates water
    level of the machine.",
      "type": "boolean"
    },
    "storage": {
      "description": "Indicates the
    remaining coffee ground storage.",
      "type": "boolean"
    },
    "on": {
      "description": "Displays the current
    power state of the machine.",
      "type": "boolean"
    }
  },
  "actions": {
    "toggle": {
      "description": "Toggles on property of
     the device."
    },
    "refill": {
     "description": "Refills the water tank
     of the machine."
    },
    "empty": {
      "description": "Empties coffee ground
    container."
    },
    "make coffee": {
      "description": "Creates a medium cup
   of coffee."
}}}
                                               class CoffeeMachine:
                                                  def \_init_{\_} (self):self.water = True # Illustrates water
                                                  level of the machine.
                                                     self.storage = True # Indicates the
                                                   remaining coffee ground storage.
                                                      self.on = False # Displays the current
                                                   power state of the machine.
                                                  def toggle(self):
                                                      # Toggles on property of the device.
                                                      self.on = not self.on
                                                   def refill(self):
                                                      # Refills the water tank of the machine.
                                                      self.water = True
                                                   def empty(self):
                                                      # Empties coffee ground container.
                                                      self.storage = True
                                                   def make coffee(self):
                                                      # Creates a medium cup of coffee.
                                                      if not self.on:
                                                          self.toggle()
                                                      if not self.water:
                                                          self.refill()
                                                      if not self.storage:
                                                          self.empty()
```
(a) A WoT TD of a coffee machine.

(b) The corresponding Python class.

**Figure 2:** An example of how a WoT TD can be translated into a Python class. Of special interest is the function make\_coffee(). It is written such that requirements to make a coffee are automatically fulfilled if necessary.

Translating the WoT TD into Python code is done using a LLM. We instructed the LLM to translated the WoT TD into a Python class as follows:

- The class has a variable for each WoT TD *Property*.
- The class has a function for each WoT TD *Action*.
- Descriptions within the WoT TD are preserved in the Python class as comments.
- If an action has prerequisites, the corresponding function should call other functions of the class, such that these prerequisites are fulfilled.

An example of how to write the WoT TD of a coffee machine into a Python class is shown in Figure [2.](#page-3-1) Checking the prerequisites and fulfilling them within the functions, allows to write the subsequent plan in a shorter and less detailed manner. Some actions require a sub-plan to be able to be performed. While creating the overall plan it is not necessary to write out these types of sub-plans anymore. For example, making a coffee requires the sub-plan to turn the machine on, and if necessary refill water and empty the storage. Generating the Python class also necessitates that the LLM decides how actions change the state of the device and what the input or actions are.

The prompt that we give the LLM also includes information about the initial state of the device. In addition to that, the LLM is provided with three examples of how to translate WoT TD into Python classes (Few-shot learning). All prompts can be seen in our Git repository.

#### <span id="page-4-0"></span>**3.3. Translating the Natural Language Task into Python Code**

After the WoT TD is translated into a Python class, this class used is as a prompt. In addition, the LLM receives the task description in natural language that it has to translate into Python code. This is done using few-shot learning. In the prompt we also tell the model that it is allowed to use loops or utility functions.

For example, the task "Make 5 cups of coffee" using the Python class in Figure [2b](#page-3-1) would correspond to:

```
coffeeMachine = CoffeeMachine()
for i in range(5):
   coffeeMachine.make_coffee()
```
Note that making a coffee may require a sub-plan to turn the device on, refill or empty the machine. This does not need to be included because it is handled withing the function of the class. The model also does not repeat the command "make\_coffee()" but can use a loop instead. This is useful because LLMs are notoriously bad in counting tasks [\[34\]](#page-10-7).

#### <span id="page-4-1"></span>**3.4. Running the Code**

In a final step all the generated code is executed. The plan is obtained by tracking the timing of function entries and exits within the Python class. The functions of the Python class correspond to the action of the device. Whenever a function was executed during runtime, we append the corresponding action to the plan.

## **4. Experiment**

The benchmark we created contains 123 planning tasks. It involves 29 devices that have between 2 and 9 tasks each. Of the WoT TDs 24 of them were written by us and 5 of them are real-world descriptions of companies that provide them for their devices. For each device, we used a device-simulator to verify the success of the plan. These simulators followed the structure of the WoT TDs but could have hidden variables or interactions. For example, the coffee-machine-simulator runs out of water after every 10 cups of coffee (This is not mentioned in the WoT TD in Figure [2a\)](#page-3-1). The number of elemental steps to fulfill a task ranges between 1 and 173 steps, which is considerably longer than in existing robotic planning benchmarks [\[35,](#page-10-8) [36\]](#page-10-9).

The LLMs for our experiments were selected such that they had a range of different sizees. We were also interested if models that are already pretrained on code generation would perform better. Therefore

we used GPT-3.5, GPT-4, Llama-7b<sup>[2](#page-5-0)</sup>, Llama-70b<sup>[3](#page-5-1)</sup>, CodeLlama-7b<sup>[4](#page-5-2)</sup> and CodeLlama-70b<sup>[5](#page-5-3)</sup>. CodeLlama-7b and CodeLlama-70b are version of Llama-7b and Llama-70b that were further fine tuned on code corpora [\[37\]](#page-10-10). In our experiments, we used 4-bit quantized versions of the models.

We will discuss an experiment in which we compare our approach with state of the art models in Section [4.1.](#page-5-4) An experiment in which we investigate which aspect of the Wo TD are most important for plan generation is shown in Section [4.2.](#page-6-0)

#### <span id="page-5-4"></span>**4.1. Experiment 1: Comparing our Approach with Existing Work**

For comparison we used the models SayCan [\[5\]](#page-8-1) and Progprompt [\[1\]](#page-8-0). The model SayCan is generating the actions in the form of "1. action1, 2. action2, ... n. done.". The model Propgprompt is generating a plan via code generation: After the actions are provided as an Python import-statement, the LLM is asked to generate code that fulfills a task.

The implementation in the original papers are not suitable for our benchmark, as they did not use WoT TD and they also did not run with instruction-tuned LLMs. Therefore we had to adapt their work as follows:

- The prompt is restructured to include a system message and a user-assistant dialogue for the few-shot examples.
- The WoT TD is provided in the prompt.
- The system message of the prompt explicitly warns the LLM that some action have requirements.
- For SayCan, the model is simply generating the action sequence instead of using the model as a heuristc.
- For Progprompt the actions of the Wot TD are are given as an import statement.

In the results below we write SayCan\* and Progprompt\* to indicate the difference between the original work and our adaptations.

The original work for Progprompt did not write loops to repeat an action sequence multiple times. However, this ability is very useful for some of the tasks in our benchmark. Therefore we tested out two versions: For Progprompt\* it is stated in its system-message that the LLM should not write loops and for Progprompt\* + Loops the LLM is instructed to use loops.

The results for different models with different LLMs is shown in Table [1.](#page-6-1) Success was measured using the state of the device-simulator after the plan was executed. Overall our method performed best except for when using GPT-4.

Overall, the GPT-3.5 and GPT-4 models performed considerably better on the benchmark than the Llama models. Our approach result in a considerable improvements for most of the LLMs. This could indicate that representing the device as a Python class in an intermediate step, causes the LLMs to be better in writing plans for it. An obvious reason SayCan under-performed could be that this method does not enable the use of loops. The bad performance of SayCan may also showcase that code-generation models are in general better planners.

Interestingly, there is no clear trend that the models fine-tuned with code-generation perform better. This could be because the programming that needs to be done is not very complicated. There are no advanced mathematical operations, highly nested loops, recursion or knowledge about specialized libraries required. Rewriting a WoT TD into Python code may be a different type of task the CodeLlamaversion were trained with, preventing them to generalize well for this new task.

Based on our inspection, most planning mistakes originate from the class generation task and not from translating the task description into code. For example, the models may miss requirements of actions or misinterpret how a variable is used. Hidden variables of the devices that cause it to behave unexpectedly are also often preventing a plan to be successful, because they are hard to predict by the LLM and the model does not allow for re-planning.

<span id="page-5-0"></span><sup>2</sup> <https://huggingface.co/TheBloke/Llama-2-7B-Chat-GPTQ>

<span id="page-5-1"></span><sup>3</sup> <https://huggingface.co/TheBloke/Llama-2-70B-Chat-GPTQ>

<span id="page-5-2"></span><sup>4</sup> <https://huggingface.co/TheBloke/CodeLlama-7B-Instruct-GPTQ>

<span id="page-5-3"></span><sup>5</sup> [https://huggingface.co/TheBloke/CodeLlama-70B-Instruct-GPTQ](https://huggingface.co/TheBloke/CodeLlama-70B-Instruct-GPTQ )

#### <span id="page-6-1"></span>**Table 1**

Results for the 123 planning tasks for different models with different LLMs. SUCCESS indicates the number of times running the code resulted in a successful plan, verified by the device-simulators. FAILURE indicates the number of times the plans were not successful, ERROR is the number of times the plan could not be executed and TIMEOUT is the number of times running the code caused an infinite loop.



## <span id="page-6-0"></span>**4.2. Experiment 2: Exploring WoT TD Performance Through Ablation Analysis**

In addition, we also performed an ablation study to investigate which aspects of the WoT-TD are most important for the the LLMs abilities to generate a plan. For this purpose we augmented the WoT TD and measured performance under the following conditions:

- **Only Task**: This is a baseline in which we already provide a (correctly implemented) Python class to the LLM and only the task needs to translated into Python code.
- **Unaltered**: The normal WoT TDs are used.
- **No Descriptions**: All descriptions within the WoT TDs are removed.
- **Generic Action names**: The names of the actions within the WoT TDs are replaced with the generic names "action1", "action2" etc.
- **No Descriptions and Generic Action Names**: There are no descriptions and the actions names are generic. Note that under this condition there is no indication of what an action is actually doing.
- **RDF**: The WoT TD are serialised into RDF data in turtle syntax.

All results with augmented WoT TD are shown in Table [2.](#page-7-0)

### <span id="page-7-0"></span>**Table 2**

The results of our approach with different augmentations of the WoT TDs. The condition "Only Tasks" serves as a baseline in which a correct WoT TD to Python class translation was provided and only the code for the task needs to be written.

![](_page_7_Picture_455.jpeg)

The general trend seems to be that the models perform well as long as there is either a description or the actions have non-generic names. Providing the correct class implementation gives a performance boost for all models. This indicates that even though the LLMs are able to extract a lot of relevant knowledge from the WoT TDs, having a human expert give the correct model is still a superior approach. Providing the WoT TD in RDF does not change the performance much compared to the standard json format.

## **5. Conclusion**

This work presented a novel method of how WoT TD can be utilised as a resource to provide LLMs with domain knowledge for planning, emitting the necessity for a human expert to ground LLMs to the environment.

We show that LLMs are successfully able to use WoT TDs as a source of domain knowledge, as long as the WoT TDs either contain descriptions or actions have meaningful names. Even though the WoT TDs do not give a detailed formal description of the workings of a device, the LLMs are able to fill these information gaps using their general world knowledge.

We show the benefit of our approach in which devices are represented as Python classes as an intermediate step. This separates the task of interpreting the workings of a device and generating the plan. We believe that this makes our method more explainable and it is simpler for humans to intervene (For example by changing the Python class). Turning the plan generation into a code generation problem allows to use loops, preventing the LLMs from having to count and having to produce long repeated sequences that may excess the token-limits of LLMs.

Future work may also include mechanisms that allow for incorporating environmental feedback. We also want to further investigate the benefit of generating Python classes that contain hierarchical task structures.

## **Acknowledgments**

This work has been supported in part by the German federal ministry of education and research (BMBF) in NeSyPlan (FKZ 01IS23052B). We also thank Jan Henze, Nicholas Popovic, and David Paul Mark for fruitful discussions and for help during preparation of the dataset and LLM set-up.

## **References**

- <span id="page-8-0"></span>[1] I. Singh, V. Blukis, A. Mousavian, A. Goyal, D. Xu, J. Tremblay, D. Fox, J. Thomason, A. Garg, Progprompt: Generating situated robot task plans using large language models, in: 2023 IEEE International Conference on Robotics and Automation (ICRA), IEEE, 2023, pp. 11523–11530.
- [2] W. Huang, F. Xia, T. Xiao, H. Chan, J. Liang, P. Florence, A. Zeng, J. Tompson, I. Mordatch, Y. Chebotar, et al., Inner monologue: Embodied reasoning through planning with language models, arXiv preprint arXiv:2207.05608 (2022).
- <span id="page-8-2"></span>[3] W. Huang, P. Abbeel, D. Pathak, I. Mordatch, Language models as zero-shot planners: Extracting actionable knowledge for embodied agents, in: International Conference on Machine Learning, PMLR, 2022, pp. 9118–9147.
- <span id="page-8-3"></span>[4] A. Zeng, M. Attarian, B. Ichter, K. Choromanski, A. Wong, S. Welker, F. Tombari, A. Purohit, M. Ryoo, V. Sindhwani, et al., Socratic models: Composing zero-shot multimodal reasoning with language, arXiv preprint arXiv:2204.00598 (2022).
- <span id="page-8-1"></span>[5] M. Ahn, A. Brohan, N. Brown, Y. Chebotar, O. Cortes, B. David, C. Finn, C. Fu, K. Gopalakrishnan, K. Hausman, et al., Do as i can, not as i say: Grounding language in robotic affordances, arXiv preprint arXiv:2204.01691 (2022).
- [6] C. H. Song, J. Wu, C. Washington, B. M. Sadler, W.-L. Chao, Y. Su, Llm-planner: Few-shot grounded planning for embodied agents with large language models, in: Proceedings of the IEEE/CVF International Conference on Computer Vision, 2023, pp. 2998–3009.
- <span id="page-9-11"></span>[7] S. Li, X. Puig, C. Paxton, Y. Du, C. Wang, L. Fan, T. Chen, D.-A. Huang, E. Akyürek, A. Anandkumar, et al., Pre-trained language models for interactive decision-making, Advances in Neural Information Processing Systems 35 (2022) 31199–31212.
- [8] S. Kaebisch, T. Kamiya, M. McCool, V. Charpenay, E. Korkan, M. Kovatsch, Web of Things (WoT) Thing Description 1.1, W3C Recommendation, W3C, 2020. URL: [https://www.w3.org/](https://www.w3.org/TR/wot-thing-description11/) [TR/wot-thing-description11/.](https://www.w3.org/TR/wot-thing-description11/)
- <span id="page-9-17"></span>[9] X. Wang, S. Li, H. Ji, Code4struct: Code generation for few-shot event structure prediction, arXiv preprint arXiv:2210.12810 (2022).
- <span id="page-9-18"></span>[10] C. Li, J. Liang, A. Zeng, X. Chen, K. Hausman, D. Sadigh, S. Levine, L. Fei-Fei, F. Xia, B. Ichter, Chain of code: Reasoning with a language model-augmented code emulator, arXiv preprint arXiv:2312.04474 (2023).
- <span id="page-9-0"></span>[11] Z. Kis, D. Peintner, C. Aguzzi, J. Hund, K. Nimura, Web of Things (WoT) Scripting API, W3C Recommendation, W3C, 2023. URL: [https://www.w3.org/TR/wot-scripting-api/.](https://www.w3.org/TR/wot-scripting-api/)
- <span id="page-9-1"></span>[12] N. J. Nilsson, et al., Shakey the robot, Technical Note 323, SRI A.I. Center (1984).
- <span id="page-9-2"></span>[13] R. E. Fikes, N. J. Nilsson, Strips: A new approach to the application of theorem proving to problem solving, Artificial intelligence 2 (1971) 189–208.
- <span id="page-9-3"></span>[14] D. Nau, Y. Cao, A. Lotem, H. Munoz-Avila, Shop: Simple hierarchical ordered planner, in: Proceedings of the 16th international joint conference on Artificial intelligence-Volume 2, 1999, pp. 968–973.
- <span id="page-9-4"></span>[15] C. Aeronautiques, A. Howe, C. Knoblock, I. D. McDermott, A. Ram, M. Veloso, D. Weld, D. W. Sri, A. Barrett, D. Christianson, et al., Pddl| the planning domain definition language, Technical Report, Tech. Rep. (1998).
- <span id="page-9-5"></span>[16] M. Fox, D. Long, Pddl2. 1: An extension to pddl for expressing temporal planning domains, Journal of artificial intelligence research 20 (2003) 61–124.
- <span id="page-9-6"></span>[17] C. R. Garrett, T. Lozano-Pérez, L. P. Kaelbling, Pddlstream: Integrating symbolic planners and blackbox samplers via optimistic adaptive planning, in: Proceedings of the international conference on automated planning and scheduling, volume 30, 2020, pp. 440–448.
- <span id="page-9-7"></span>[18] Y.-q. Jiang, S.-q. Zhang, P. Khandelwal, P. Stone, Task planning in robotics: an empirical comparison of pddl-and asp-based systems, Frontiers of Information Technology & Electronic Engineering 20 (2019) 363–373.
- <span id="page-9-8"></span>[19] J. A. Baier, F. Bacchus, S. A. McIlraith, A heuristic search approach to planning with temporally extended preferences, Artificial Intelligence 173 (2009) 593–618.
- <span id="page-9-9"></span>[20] H. Zhang, W. Du, J. Shan, Q. Zhou, Y. Du, J. B. Tenenbaum, T. Shu, C. Gan, Building cooperative embodied agents modularly with large language models, arXiv preprint arXiv:2307.02485 (2023).
- <span id="page-9-10"></span>[21] P. A. Jansen, Visually-grounded planning without vision: Language models infer detailed plans from high-level instructions, arXiv preprint arXiv:2009.14259 (2020).
- <span id="page-9-12"></span>[22] T. Schick, J. Dwivedi-Yu, R. Dessì, R. Raileanu, M. Lomeli, E. Hambro, L. Zettlemoyer, N. Cancedda, T. Scialom, Toolformer: Language models can teach themselves to use tools, Advances in Neural Information Processing Systems 36 (2024).
- <span id="page-9-13"></span>[23] L. Pan, A. Albalak, X. Wang, W. Y. Wang, Logic-lm: Empowering large language models with symbolic solvers for faithful logical reasoning, arXiv preprint arXiv:2305.12295 (2023).
- <span id="page-9-14"></span>[24] B. Liu, Y. Jiang, X. Zhang, Q. Liu, S. Zhang, J. Biswas, P. Stone, Llm+ p: Empowering large language models with optimal planning proficiency, arXiv preprint arXiv:2304.11477 (2023).
- <span id="page-9-15"></span>[25] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, H. Wang, Large language models for software engineering: A systematic literature review, arXiv preprint arXiv:2308.10620 (2023).
- <span id="page-9-16"></span>[26] P. Denny, V. Kumar, N. Giacaman, Conversing with copilot: Exploring prompt engineering for solving cs1 problems using natural language, in: Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1, 2023, pp. 1136–1142.
- <span id="page-10-0"></span>[27] S. Bubeck, V. Chandrasekaran, R. Eldan, J. Gehrke, E. Horvitz, E. Kamar, P. Lee, Y. T. Lee, Y. Li, S. Lundberg, et al., Sparks of artificial general intelligence: Early experiments with gpt-4, arXiv preprint arXiv:2303.12712 (2023).
- <span id="page-10-1"></span>[28] A. Madaan, S. Zhou, U. Alon, Y. Yang, G. Neubig, Language models of code are few-shot commonsense learners, arXiv preprint arXiv:2210.07128 (2022).
- <span id="page-10-2"></span>[29] M. Skreta, Z. Zhou, J. L. Yuan, K. Darvish, A. Aspuru-Guzik, A. Garg, Replan: Robotic replanning with perception and language models, arXiv preprint arXiv:2401.04157 (2024).
- <span id="page-10-3"></span>[30] J. Liang, W. Huang, F. Xia, P. Xu, K. Hausman, B. Ichter, P. Florence, A. Zeng, Code as policies: Language model programs for embodied control, in: 2023 IEEE International Conference on Robotics and Automation (ICRA), IEEE, 2023, pp. 9493–9500.
- <span id="page-10-4"></span>[31] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou, et al., Chain-of-thought prompting elicits reasoning in large language models, Advances in Neural Information Processing Systems 35 (2022) 24824–24837.
- <span id="page-10-5"></span>[32] W. Chen, X. Ma, X. Wang, W. W. Cohen, Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks, arXiv preprint arXiv:2211.12588 (2022).
- <span id="page-10-6"></span>[33] X. Wang, Y. Chen, L. Yuan, Y. Zhang, Y. Li, H. Peng, H. Ji, Executable code actions elicit better llm agents, arXiv preprint arXiv:2402.01030 (2024).
- <span id="page-10-7"></span>[34] L. Parcalabescu, A. Gatt, A. Frank, I. Calixto, Seeing past words: Testing the cross-modal capabilities of pretrained v&l models on counting tasks, arXiv preprint arXiv:2012.12352 (2020).
- <span id="page-10-8"></span>[35] M. Shridhar, J. Thomason, D. Gordon, Y. Bisk, W. Han, R. Mottaghi, L. Zettlemoyer, D. Fox, Alfred: A benchmark for interpreting grounded instructions for everyday tasks, in: Proceedings of the IEEE/CVF conference on computer vision and pattern recognition, 2020, pp. 10740–10749.
- <span id="page-10-9"></span>[36] C. Li, R. Zhang, J. Wong, C. Gokmen, S. Srivastava, R. Martín-Martín, C. Wang, G. Levine, M. Lingelbach, J. Sun, et al., Behavior-1k: A benchmark for embodied ai with 1,000 everyday activities and realistic simulation, in: Conference on Robot Learning, PMLR, 2023, pp. 80–93.
- <span id="page-10-10"></span>[37] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin, et al., Code llama: Open foundation models for code, arXiv preprint arXiv:2308.12950 (2023).