On the Challenges of Using Large Language Models for NCL Code Generation

Daniel de Sousa Moraes
TeleMídia/PUC-Rio
danielmoraes@telemidia.puc-rio.br

Polyana Bezerra da Costa
TeleMídia/PUC-Rio
polyana@telemidia.puc-rio.br

Antonio J. G. Busson
BTG Pactual
antonio.busson@btgpactual.com

José Matheus Carvalho Boaro
TeleMídia/PUC-Rio
boaro@telemidia.puc-rio.br

Carlos de Salles Soares Neto
TeleMídia-MA/UFMA
carlos.salles@ufma.br

Sergio Colcher
TeleMídia/Departamento de Informática/PUC-Rio
colcher@inf.puc-rio.br

Abstract

A significant concern raised in the domain of authoring tools for interactive Digital TV (iDTV) has been their usability when considering the target audience, which typically consists of content creators and not necessarily programmers. NCL (Nested Context Language), the declarative language for developing interactive applications for Brazilian Digital TV and an ITU-T Recommendation for IPTV services, is a simple declarative language but not an easy tool for non-technical authors. The proliferation of Large Language Models (LLMs) has recently instigated substantial transformations across several domains, including synthesizing code with remarkable potential. This paper proposes an investigation into the challenges of using LLMs to aid automatic NCL code generation/synthesis in authoring tools for iDTV content production. It shows initial evidence that current pre-trained LLMs cannot synthesize NCL code with satisfactory quality. In this context, we raise the main challenges for NCL code generation using LLMs and some issues related to the good practices for engineering prompts and integrating pre-trained LLMs into multimedia authoring tools.

Keywords: NCL, LLMs, Code Generation, Authoring

1 Introduction

Authoring tools [3, 6, 11, 12, 18, 21, 23, 25, 26] have been the subject of extensive study when addressing the development of applications in NCL (Nested Context Language), the declarative language for developing interactive applications for the Brazilian Digital TV and ITU-T IPTV systems.

A significant concern within this domain has been the usability of these tools for their target audience, which typically consists of content creators rather than programmers. Consequently, these tools have focused on creating visual abstractions, rather than textual, to facilitate their utilization. However, these abstractions can introduce inherent limitations and a certain level of complexity, requiring users to invest time in learning and adapting to the provided functionalities.

The proliferation of Large Language Models (LLMs) has recently instigated substantial transformations across several domains. These models have facilitated the creation of chatbots capable of appropriately responding to diverse requests within diverse contexts, such as chatGPT[1] and Google Bard[2]. LLMs have been applied to program or code synthesis tasks and presented remarkable potential [8, 9, 14, 19, 22]. Thus, it seems that, if embedded in a multimedia authoring tool, LLMs could facilitate the development of interactive NCL applications, allowing authors to intuitively define application requirements in natural language.

This paper shows initial evidence that current pre-trained LLMs cannot synthesize NCL codes with satisfactory quality. It mainly fails with syntax and language rules while not generating content that meets specific application requirements. In this context, we raise the main challenges for NCL code generation using LLMs. We also raise challenges related to good practices for engineering prompts and integrating pre-trained LLMs into multimedia authoring tools.

2 NCL Code Generation

We conducted experiments employing current LLMs to assess whether their performance in the NCL code generation task is satisfactory or whether it is necessary and plausible to fine-tune a pre-trained model to generate NCL code. For this, we wrote three different prompts for NCL code generation and conducted tests with 4 LLM-based services: chatGPT, Bard, Llama-v2 [28], and PaLM2 [2]. These prompts are as follows in Listing 1.

We utilize these prompts as inputs for each model and subsequently compare their respective outputs. Following this, the responses were manually evaluated by two NCL experts based on a rating scale of 0 to 4, which signifies the code generation quality. Each response score was given in

In: Workshop Futuro da TV Digital Interativa, Ribeirão Preto, Brasil. Anais Estendidos do Simpósio Brasileiro de Sistemas Multimídia e Web (WebMedia). Porto Alegre: Sociedade Brasileira de Computação, 2023.

ISSN 2596-1683

[1]https://openai.com/blog/chatgpt
[2]https://bard.google.com/

mutual agreement by both evaluators. Results are presented in Table 1.

```
Prompt 1: Write an NCL (Nested Context Language)
    code that starts a video file named "video.mp4".

Prompt 2: Write an NCL code to start a video in full
    screen using a port, and after 10 seconds, using
    a link, it starts an image in the top-right
    corner of the screen.

Prompt 3: Write an NCL code to start a video in full
    screen using a port, and after 10 seconds, using
    a link, it starts an image in the top-right
    corner of the screen. The following code
    exemplifies a basic application where a video is
    initiated using a port element:
   <?xml version="1.0" encoding="ISO-8859-1"?>
   <ncl id="exemplo01"
       xmlns="http://www.ncl.org.br/NCL3.0/EDTVProfile">
      <head>
         <regionBase>
            <region id="rgVideo1" zIndex="1" />
         </regionBase>
         <descriptorBase>
            <descriptor id="dVideo1"
                region="rgVideo1" />
         </descriptorBase>
      </head>
      <body>
         <port id="pVideoAbertura"
             component="videoAbertura" />
         <media id="videoAbertura"
             src="media/abertura.mpg"
             descriptor="dVideo1" />
      </body>
   </ncl>
```

Listing 1. Experiments prompts to evaluate generic LLM on NCL code generation

LLM	Prompt 1	Prompt 2	Prompt 3
ChatGPT (GPT-3.5)	0	2	1
Bard	1	1	1
Llama-v2 70B	0	0	0
PaLM2 (Bison)	1	1	3

Table 1. LLMs rating for each prompt. (0) Can not generate code; (1) Invalid NCL code; (2) NCL code with many errors; (3) NCL code with few errors; (4) NCL code as expected.

In **Prompt 1**, the task was to define an application code with one element media referring to a video and start it. The ChatGPT answered that it could not generate an NCL code response, alleging that "*...there is no widely known or standardized programming language called "Nested Context Language" (NCL)..*". Conversely, the Bard and PaLM2 models demonstrated the ability to generate code-form responses. However, they hallucinated, producing responses in languages other than NCL. Bard generated a Python response referencing an inexistent API named "ncl," while PaLM2 just generated a JSON object defining attributes of a video.

Prompt 2 demands a slightly more elaborated task. It explicitly says that the video presentation must be initialized using a port element. Also, after 10 seconds of playing the video, an image on a specific screen region must be started using a link. ChatGPT was able to generate NCL codes but with many syntactic errors and semantically far from the expected response. Bard and PaLM2 suffered from hallucinations again. This time, PaLM2 generated what looks like Python code. Bard generated a Python code similar to the previous task, using a made API named "ncl".

Lastly, in **Prompt 3**, we used the same task of **Prompt 2**, but this time adding an example of an NCL code, in which a video media is defined and initiated through a port element. In this round, the PaLM2 model performed better than others, generating a code close to the expected response. Both GPT-3.5 and Bard generated invalid NCL codes. For all the prompts, the Llama-v2 model could not generate a response code in any programming language.

This experiment shows early signs that even LLMs trained on large amounts of data still cannot be adequately used for NCL code generation tasks. We hypothesize that this happens because NCL is a domain language for a particular niche. Thus, it is necessary to fine-tune such LLMs for the proper generation of NCL codes.

3 Challenges of Using LLMs for NCL code generation

Generally, LLMs denote Transformer-based language models that contain hundreds of billions of parameters or even more [16]. These models undergo training on extensive textual datasets. With this in mind, we can identify a set of specific, though not unique, challenges that need to be addressed to implement LLMs for the generation of NCL codes successfully.

3.1 Adjusting the LLMs

Using LLMs trained on generic data and task-agnostic has demonstrated strong performances across a wide range of tasks [5, 10, 28]. However, NCL code generation tasks might perform differently than desired, as demonstrated in the experiment in section 2.

For instance, Listing 2 shows that even the best response for the prompts, using natural language, fails to generate a correct code in the experiment. This code was acquired as a response to the **Prompt3** by the PaLM2 model in the experiment, and even though it is the most coherent and closest response to the desired code, it still fails to complete the task.

The link created does not correspond to the prompt, it is starting the image presentation after the video stops. In the prompt, we ask to start an image ten seconds after the video begins. Besides, the connector used is not declared or imported.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<ncl id="exemplo02"
    xmlns="http://www.ncl.org.br/NCL3.0/EDTVProfile">
   <head>
      <regionBase>
         <region id="rgVideo1" zIndex="1" />
         <region id="rgImagem1" zIndex="2" top="0"
             right="0" width="200" height="200" />
      </regionBase>
      <descriptorBase>
         <descriptor id="dVideo1"
             region="rgVideo1" />
         <descriptor id="dImagem1"
             region="rgImagem1" />
      </descriptorBase>
   </head>
   <body>
      <port id="pVideoAbertura"
          component="videoAbertura" />
      <media id="videoAbertura"
          src="media/abertura.mpg"
          descriptor="dVideo1" />
      <link xconnector="onEndStart">
         <bind role="onEnd"
             component="videoAbertura" />
         <bind role="start" component="imagem1" />
      </link>
      <media id="imagem1" src="media/imagem.jpg"
          descriptor="dImagem1" />
   </body>
</ncl>
```

Listing 2. PaLM2 generated code for Prompt 3

Many factors can explain the results in section 2, but mainly because the model was not trained for code generation. Thus, the models may sometimes suffer from hallucinations and include inaccurate information in their responses [16]. To mitigate this, some works adapt generic LLMs, fine-tuning them to specific tasks, such as code generation. There are works for particular languages such as Codex [8], a GPT-3 LLM fine-tuned to generate Python functions from doc strings. Also, phi-1 [14], a smaller LLM, which is also trained to generate Python functions from doc strings. There are also models trained for multiple languages, such as CodeGen LLMs [22] that use C, C++, Go, Java, JavaScript, and Python code datasets, and the AlphaCode LLM [19], pre-trained on a C++, C#, Go, Java, JavaScript, Lua, PHP, Python, Ruby, Rust, Scala and TypeScript source code dataset collected from GitHub.

All the works mentioned positively impact the performance of their specific tasks. Thus, it is clear the necessity of experiments to confirm the need to train LLMs with specific examples of NCL code, comparing their performance with that of generic LLMs. Such an experiment demands the creation of a dataset with examples of NCL code.

One of the main challenges is the creation or use of a sufficiently extensive dataset for training the LLM or fine-tuning a pre-trained model, as done in [8, 9, 14, 19, 22]. This entails defining and collecting a substantial number of code examples that effectively represent the core characteristics of the NCL language and its potential applications. By assembling a diverse set of code samples, we enable the model to grasp the intricacies of the language and adopt coding best practices specific to NCL. It is essential to recognize that the more exposure the model has to pertinent examples, the better equipped it becomes to produce high-quality code.

Moreover, in the experiment of specifying an LLM for NCL code generation, we can also highlight questions about what type of adaptation in the LLM would be feasible, given the context of available data about the language and the resources available for model implementation and execution.

We can mention two widely used methods (among many others) [30], fine-tuning and few-shot training. The fine-tuning, used in several works already mentioned, is a process of re-training on a task-specific dataset, a pre-trained model, updating all parameters, which, depending on the model architecture, will require significant memory resources to store parameters, model activations, etc. [16]. Besides, it also requires a fair amount of task-specific data to optimize its performance, adding to the need for dataset construction.

There are also the Parameter-efficient fine-tuning (PEFT) methods, such as Adapters [15], that add learnable layers into the Transformer architecture to be updated during the fine-tuning, keeping the rest of the network from change. Another method is prefix-tuning, when token embeddings are added to an input to be learned during the fine-tuning without changing the rest of the model's parameters. These methods can reduce resource usage while maintaining competitive performance to a full fine-tuning.

Another promising approach is using *zero, one-or-few shot training* [29]. In this method, a small set of examples is given along with a prompt query. The LLM uses the information received to teach itself to complete the demanded task. According to Ahmed and Devanbu [1], this method does not require weight adjustments. This method enables the specialization of a generic LLM for NCL code generation without the need for ample labeled examples and the availability of many computing resources.

Lastly, after defeating all the challenges above, we must evaluate the performance of the chosen methods and models. In this stage, we ensure the model effectively generalizes to new data and performs proficiently. We may adjust large models using PPO (Proximal Policy Optimization) based on a reward model trained to align the models with human preferences [24]. ChatGPT also employed this approach. The reward model is trained using comparison data generated by human labelers who rank the model outputs manually. Based on these rankings, the reward model or a machine labeler calculates a reward that is then utilized to update the FM through PPO.

3.2 Prompt Engineering and User Interaction

Regarding the challenges of embedding an LLM for NCL generation in an authoring tool. Our attention must also include investigating mechanisms within the authoring tool that will enable developers to interact with the model, ensuring the highest level of response accuracy.

The process of designing prompts is challenging in itself, as the performance of the LLM is highly dependent on the quality of the prompt informed [31]. Since natural language is highly expressive and imprecise at times, it can lead to ambiguous prompts, making it difficult for the model to capture user intent [17]. Zhou et al. [31] claim that even though most effective prompts have been human-engineered, using LLMs accurately for more complex tasks is not straightforward since it requires careful development.

One potential way to overcome this is by using formal specifications when designing prompts, as suggested by [4]. However, this approach is not practical if we intend to cater to a broader audience, especially people unfamiliar with programming languages.

Therefore, many other works have explored ways of facilitating human-LLM interaction by either automatizing prompts creation [20, 31]; improving prompts by enforcing specific reliability rules [27]; or providing a prompt editing interface with standard IDE features, such as syntax highlighting and refactoring [13]; or even visual interactions combined with pre-defined workflow operations [7].

Thus, finding the most effective way to allow the user to interact with the LLM without compromising the performance of NCL code generation constitutes another great challenge. The central goal of this challenge revolves around whether to adopt natural language prompts, employ lower-level prompts through the decomposition of tasks into more straightforward, fundamental commands, or leverage visual abstractions to simplify and automate the prompt construction process. We will need to experiment with adapting the mentioned approaches to this specific task and design a different one that suits the requirements well.

Besides, as we are talking about a multimedia authoring tool, we also need to understand how this new paradigm can be combined with the existing and well-established paradigms and how we can make the best use of well-known abstractions used by previous multimedia authoring tools, such as Composer [18], or STEVE [12], allowing the visualization and editing of the generated applications.

Furthermore, aside from our efforts to enhance the LLM's performance through user interactions, we must also prioritize the practical usability of the authoring tool. To achieve this, it becomes necessary to conduct evaluations involving the intended audience. These evaluations will provide valuable insights to determine the most appropriate approach for refining the tool's usability.

4 Risks and Limitations

Besides the challenges in prompt design/engineering, the sole use of LLMs for code generation already brings potential risks. In a hazard assessment article for Codex [17], an LLM for code synthesis, the authors noted that such a tool has the potential for misuse and may offer technological, social, and economic risks. The potential risks/hazards include:

- **Inefficient Code**: Generated code may not be optimized for performance or resource usage, leading to suboptimal results in terms of speed and efficiency;
- **Lack of Creativity**: Models can only generate code based on the examples they were trained on. They may struggle with tasks or languages that were not well-represented in their training data, so they may not be able to come up with innovative solutions or creative problem-solving;
- **Debugging Challenges**: Code generated by LLMs can contain bugs or vulnerabilities. Identifying and fixing these issues can be challenging, especially if the codebase is large;
- **Misleading Solutions**: LLMs can produce code that appears correct but does not perform the intended task accurately. This can lead to unreliable software that does not meet the functional requirements, risking user satisfaction and trust;
- **Lack of Context**: LLMs may not have access to the full context of a project or its requirements, leading to code that does not fully align with the intended functionality;
- **Maintenance Issues**: Code generated by LLMs may be hard to maintain and update over time, especially if the original developers are not familiar with the model-generated code;
- **Vulnerable Code**: Automated code generation may result in vulnerabilities and security flaws if not adequately reviewed. Code generated by these models might contain vulnerabilities that compromise the safety and security of the application, leaving it susceptible to attacks;
- **Legal and Licensing Issues**: There could be legal and licensing challenges when using code generated by LLMs, mainly if the code includes proprietary or copyrighted content;

- **Ethical Concerns**: The use of LLMs for code generation may raise ethical concerns related to plagiarism, copyright infringement, or the unintended use of third-party code without proper attribution;
- **Overreliance on Models**: Developers may become overly reliant on language models, potentially neglecting their programming skills and critical thinking abilities. Relying on language models for code generation depends on their availability and maintenance. If the model becomes obsolete or is discontinued, it can disrupt ongoing projects;
- **Quality Control**: Ensuring the quality of code generated by LLMs requires careful review and testing, which can be time-consuming and resource-intensive;
- **Training Data Bias**: The biases present in the training data used for LLMs can lead to code that reflects these biases, potentially impacting the inclusivity and fairness of software projects.
- **Environmental Impact**: Training and running LLMs require significant computational resources, leading to high energy consumption. This energy-intensive process contributes to environmental harm, including increased carbon emissions, which is a concern given the need for sustainability in technology.
- **Digital Divide**: Using code generation tools necessitates access to appropriate hardware, stable internet connections, and technical expertise. This can exclude economically vulnerable groups who may lack access to these resources, exacerbating disparities in the tech industry and limiting opportunities for underprivileged individuals.

5 Conclusions

This paper proposes utilizing LLMs to generate NCL codes. Initially, we empirically find indications that pre-trained LLMs exhibit suboptimal performance when tasked with generating NCL code using natural language prompts. Subsequently, we outline a non-exhaustive list of challenges that must be addressed to effectively adapt an LLM to the specific task of NCL code generation.

Furthermore, we highlight the potential challenges of achieving user-centric usability while maintaining robust model performance for task resolution. Lastly, we enumerate the inherent risks and limitations of using LLMs for code generation. Through this article, we aspire to lay the groundwork for developing an authoring tool capable of seamlessly merging well-established functionalities from previous authoring tools with the untapped potential offered by LLMs in NCL application development.

In future work, we plan to perform a robust evaluation using current LLMs to properly attest to the efficiency of these models in automatically generating NCL code. Furthermore, we will also test the adaptation of LLMs using a large dataset of NCL codes.

Acknowledgments

The authors would like to thank CAPES and CNPq for partial financial support for this work.

References

[1] Toufique Ahmed and Premkumar Devanbu. 2022. Few-shot training LLMs for project-specific code-summarization. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–5.

[2] Rohan Anil and Andrew M. Dai. 2023. PaLM 2 Technical Report. (2023). arXiv:2305.10403 [cs.CL]

[3] Roberto Gerson Albuquerque Azevedo, Carlos de Salles Soares Neto, Mario Meireles Teixeira, Rodrigo Costa Mesquita Santos, and Thiago Alencar Gomes. 2011. Textual authoring of interactive digital TV applications. In *Proceedings of the 9th European Conference on Interactive TV and Video*. 235–244.

[4] Stephen H Bach, Victor Sanh, Zheng-Xin Yong, Albert Webson, Colin Raffel, Nihal V Nayak, Abheesht Sharma, Taewoon Kim, M Saiful Bari, Thibault Fevry, et al. 2022. Promptsource: An integrated development environment and repository for natural language prompts. *arXiv preprint arXiv:2202.01279* (2022).

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

[6] Antonio José G Busson, André Luiz de B Damasceno, Thacyla de S Lima, and Carlos de Salles Soares Neto. 2016. Scenesync: A hypermedia authoring language for temporal synchronism of learning objects. In *Proceedings of the 22nd Brazilian Symposium on Multimedia and the Web*. 175–182.

[7] Yuzhe Cai, Shaoguang Mao, Wenshan Wu, Zehua Wang, Yaobo Liang, Tao Ge, Chenfei Wu, Wang You, Ting Song, Yan Xia, et al. 2023. Low-code LLM: Visual Programming over LLMs. *arXiv preprint arXiv:2304.08103* (2023).

[8] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[9] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128* (2023).

[10] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. 2022. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311* (2022).

[11] André Luiz de B Damasceno, Thacyla de Sousa Lima, Carlos de Salles Soares Neto, et al. 2014. Cacuriá: Uma Ferramenta de Autoria Multimídia para Objetos de Aprendizagem. In *Anais dos Workshops do Congresso Brasileiro de Informática na Educação*, Vol. 3. 76.

[12] Douglas Paulo de Mattos and Débora C Muchaluat-Saade. 2018. Steve: A hypermedia authoring tool based on the simple interactive multimedia model. In *Proceedings of the ACM Symposium on Document Engineering 2018*. 1–10.

[13] Alexander J Fiannaca, Chinmay Kulkarni, Carrie J Cai, and Michael Terry. 2023. Programming without a Programming Language: Challenges and Opportunities for Designing Developer Tools for Prompt Programming. In *Extended Abstracts of the 2023 CHI Conference on Human Factors in Computing Systems*. 1–7.

[14] Suriya Gunasekar, Yi Zhang, Jyoti Aneja, Caio César Teodoro Mendes, Allie Del Giorno, Sivakanth Gopi, Mojan Javaheripi, Piero Kauffmann, Gustavo de Rosa, Olli Saarikivi, et al. 2023. Textbooks Are All You Need. *arXiv preprint arXiv:2306.11644* (2023).

[15] Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. 2019. Parameter-efficient transfer learning for NLP. In *International Conference on Machine Learning*. PMLR, 2790–2799.

[16] Jean Kaddour, Joshua Harris, Maximilian Mozes, Herbie Bradley, Roberta Raileanu, and Robert McHardy. 2023. Challenges and applications of large language models. *arXiv preprint arXiv:2307.10169* (2023).

[17] Heidy Khlaaf, Pamela Mishkin, Joshua Achiam, Gretchen Krueger, and Miles Brundage. 2022. A hazard analysis framework for code synthesis large language models. *arXiv preprint arXiv:2207.14157* (2022).

[18] Rodrigo Laiola Guimarães, Romualdo Monteiro de Resende Costa, and Luiz Fernando Gomes Soares. 2008. Composer: Authoring tool for iTV programs. In *Changing Television Environments: 6th European Conference, EUROITV 2008, Salzburg, Austria, July 3-4, 2008 Proceedings 6*. Springer, 61–71.

[19] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with alphacode. *Science* 378, 6624 (2022), 1092–1097.

[20] Vadim Liventsev, Anastasiia Grishina, Aki Härmä, and Leon Moonen. 2023. Fully Autonomous Programming with Large Language Models. *arXiv preprint arXiv:2304.10423* (2023).

[21] Carlos de Salles Soares Neto, Thacyla de Sousa Lima, André Luiz de B Damasceno, and Antonio José G Busson. 2017. Creating Multimedia Learning Objects. In *Proceedings of the 23rd Brazillian Symposium on Multimedia and the Web*. 19–21.

[22] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474* (2022).

[23] Dina Nogueira, Lois Nascimento, Michael Mello, and Rodrigo Braga. 2020. NuGinga Playcode: A web NCL/NCLua authoring tool for Ginga-NCL digital TV applications. In *Anais Estendidos do XXVI Simpósio Brasileiro de Sistemas Multimídia e Web*. SBC, 75–78.

[24] Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. 2022. Training language models to follow instructions with human feedback. (2022). arXiv:2203.02155 [cs.CL]

[25] Douglas Paulo de Mattos, Júlia Varanda da Silva, and Débora Christina Muchaluat-Saade. 2013. NEXT: graphical editor for authoring NCL documents supporting composite templates. In *Proceedings of the 11th european conference on Interactive TV and video*. 89–98.

[26] Hedvan Fernandes Pinto, Antonio José Grandson Busson, Carlos de Salles Soares Neto, and Samyr Beliche Vale. 2016. Creating Non-Linear Interactive Narratives with Fábulas Model. In *Proceedings of the 22nd Brazilian Symposium on Multimedia and the Web*. 207–210.

[27] Chenglei Si, Zhe Gan, Zhengyuan Yang, Shuohang Wang, Jianfeng Wang, Jordan Boyd-Graber, and Lijuan Wang. 2022. Prompting gpt-3 to be reliable. *arXiv preprint arXiv:2210.09150* (2022).

[28] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971* (2023).

[29] Yaqing Wang, Quanming Yao, James T Kwok, and Lionel M Ni. 2020. Generalizing from a few examples: A survey on few-shot learning. *ACM computing surveys (csur)* 53, 3 (2020), 1–34.

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

[31] Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. 2022. Large language models are human-level prompt engineers. *arXiv preprint arXiv:2211.01910* (2022).