# Use of Large Language Models in Game Control

Harsh Mahajan[1], Sai Krishna Dontaraju[1], David Sacharny[2], and Thomas C. Henderson[1]

***Abstract*— The advent of Large Language Models has sparked a paradigm shift in natural language processing, enabling machines to comprehend and generate human-like text with unprecedented accuracy and fluency. We have seen an explosive increase in research on their potential and use in various domains, yet one of the underexplored areas is the role and potential of LLMs in games. Emerging large language models (LLMs) provide a previously impossible opportunity to automatically translate a task description in a natural language to a scenario configuration, i.e., a JSON configuration that the game executes. We investigate the use of LLMs to generate a JSON configuration file that initializes the game environment and orchestrates the game according to the behavior mentioned by the user in natural language.**

## I. INTRODUCTION

The proliferation of Large Language Models (LLMs) marks a significant milestone in the field of natural language processing, opening up new avenues for automation and innovation across a myriad of domains. As researchers and practitioners continue to harness the capabilities of LLMs, one area that warrants further exploration is their integration into the realm of gaming. While the potential of LLMs has been extensively studied in various contexts, their application in game development remains relatively untapped.

This study delves into our exploration of harnessing the power of LLMs, specifically focusing on our utilization of LLMs to generate a game configuration file. This configuration file serves as the blueprint for initiating the game, encapsulating crucial parameters and settings. Our approach aims to empower users to shape the design and specifications of their envisioned game through natural language interaction. By leveraging open-source LLMs adept at orchestrating complex tasks, we enable users to articulate their game design preferences through descriptive prompts. In response, LLMs dynamically generate a JSON configuration file that adheres to the specified schema, laying the foundation for the subsequent stages of game development.

At the heart of this endeavor lies the seamless integration of user input and advanced AI capabilities to streamline the game design process. Rather than relying on traditional manual methods for configuring game settings, our approach empowers users to directly influence the game's parameters through intuitive natural language prompts. Subsequently, Mistral translates these inputs into structured JSON format, ensuring game compatibility and facilitating seamless gameplay transition.

[1]University of Utah, Salt Lake City, UT tch@cs.utah.edu

[2]Chief Technology Officer, Blyncsy, Inc., 175 W 200 S STE 1000, Salt Lake City, UT david.sacharny@gmail.com

## II. PREVIOUS WORK

Integrating Large Language Models into video gaming marks a shift, particularly in creating dynamic portrayals of game characters, their attributes, and strategic placements. This technological evolution can redefine the creative processes within game development, offering a deeper, more nuanced approach to game deployment and interactive gameplay.

### A. *The architecture of LLMs*

At their core, LLMs operate on the principles of machine learning through transformer architectures, which are adept at processing large amounts of data simultaneously. This capability is crucial in understanding and generating nuanced text sequences, making LLMs particularly suitable for tasks that require a deep understanding of context and detail, such as writing and dialogues in games. Initially designed for tasks like translation, summarization, and question-answering, LLMs have been trained on vast and diverse internet text datasets. This training imbues them with a rich linguistic and cultural knowledge repository, which they can creatively leverage to generate diverse game content.

### B. *Influence of LLMs in games*

According to Gallotta et al., [1] the emergence of LLMs has drastically expanded the potential applications of these models in various domains, including gaming. The authors outline how these models have moved from primarily text-based tools to versatile systems capable of engaging in and enhancing gaming experiences. This transition highlights the adaptability of LLMs and sets the stage for their use in generating dynamic game content.

Careful prompting directs LLMs to generate coherent and contextually relevant text, making them crucial for crafting richer storytelling and character development in games through narratives and dialogues. These techniques enhance the model's ability to generate content that is not only high quality but also tailored to specific gaming scenarios, making LLMs effective for creating detailed game characters and environments.

### C. *Mistral usage in generating structured output*

Mistral [2] is designed to handle complex sequences and structures, making it particularly suitable for tasks that require more than simple text generation. Mistral can be trained on specific game-level data in game development, learning the underlying patterns and logic that define successful game design. This training allows Mistral to generate game levels that are structurally sound, engaging, and challenging for players. Its capability to process and output structured JSON makes it ideal for direct integration into game engines, where generated levels can be directly imported and used within games.

For instance, when generating levels for a puzzle game like Sokoban, LLMs can be configured to understand and replicate the game's essential elements—players, goals, obstacles, and navigable paths.

### D. Leveraging LLMs to generate game levels

The paper by Todd et al., [3] explores how attention-based LLMs, which have shown exceptional performance in natural language processing tasks, can also generate non-linguistic artifacts such as video game levels. This exploration aligns with the growing trend of employing LLMs like Mistral for more than traditional text-processing applications in the gaming industry. The paper discusses the challenge and feasibility of using LLMs to generate game levels, acknowledging the complexity due to games' spatial and functional requirements, which differ significantly from the text data typically used in LLM training. Through their experiments with the Sokoban game, the authors demonstrate that LLMs can generate functional game levels. However, the performance heavily depends on the size of the dataset provided for training.

Building on the insights from Todd et al., [3] and Gallotta et al., [1] using Mistral [2] to generate game characteristics such as JSON structures with Pydantic can be seen as part of a broader movement towards integrating advanced LLM capabilities into game development. This process can begin with a simple prompt describing the desired game characteristics, which Mistral interprets to produce a JSON output. This output, validated through Pydantic, ensures that the game elements are correctly formatted and functional, illustrating a practical application of LLMs in automating and enriching game content creation.

## III. Method

### A. System Architecture

Our tool adopts a client-server architecture to facilitate the integration of Large Language Models (LLMs) into game development workflows. The LLM is hosted on a server equipped with GPU capabilities and exposes an open API endpoint for seamless interaction. Clients interact with the system through a user-friendly chatbox interface developed using Streamlit. Within the chatbox, users input their queries, which are subsequently transmitted to the server. The server processes the queries using the LLM and responds with a JSON configuration file tailored to initialize the game environment and control in-game agents.

This architecture streamlines the interaction between users and the LLM-powered system, providing a straightforward and intuitive interface for scenario configuration. By leveraging the computational power of GPUs and the flexibility of an open API, our tool empowers developers to harness the capabilities of LLMs without the need for extensive technical expertise or infrastructure setup.

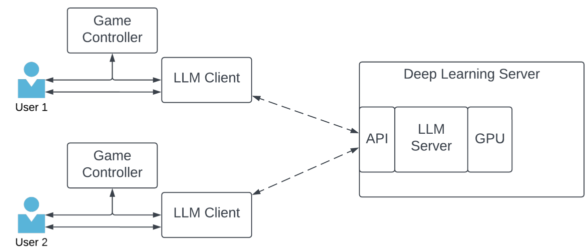


Fig. 1. System Architecture design

### B. Defining JSON structure using Pydantic

Generating structured JSON outputs solely through Large Language Models (LLMs) poses significant challenges. While LLMs excel in generating human-like text and performing various language tasks with remarkable accuracy, ensuring the syntactic correctness and adherence to a predefined schema is non-trivial. As such, before engaging LLMs to generate JSON configurations, it becomes imperative to establish a clear JSON structure that aligns with the desired output format.

To address this challenge, we leveraged Pydantic [4], the leading data validation library for Python. Pydantic offers robust capabilities to define and validate JSON schemas and is compatible with LangChain [5].

By employing Pydantic, we crafted a JSON schema that encapsulates the desired structure of the JSON outputs. This schema serves as a blueprint for LLMs, ensuring that the generated outputs conform to the predefined structure and uphold syntactic correctness. By establishing a clear schema upfront, we mitigate the risk of encountering errors or nonsensical outputs from the LLMs.

### C. Language Models

The capacity of the LLMs influences the quality of the generated JSON. Consequently, we explored various LLMs to assess their impact on output quality. We tested with GPT-3.5, Llama 2, and Mistral models. Given our commitment to leveraging open-source models for this project, we opted not to adhere to GPT-3.5 despite its satisfactory performance. During our experimentation with Llama 2 – 13B and Mistral – 7B, we observed that the JSON configurations produced by these models did not meet our expectations compared to those generated by GPT-3.5.

Consequently, we directed our focus towards models specifically finetuned for generating structured outputs. Our search led us to Hermes-2 Pro [2], a model finetuned from Mistral-7B-v0.1. This model retains its impressive performance across various tasks and conversations while excelling, particularly in Function Calling and JSON Structured Outputs. The evaluation conducted revealed a function calling score of 90% and an 84% score for structured JSON output as described on their webpage.

Hermes-2 Pro adopts ChatML as its prompt format and is trained on a dedicated system prompt tailored for structured outputs. The model strictly produces JSON object responses adhering to the prescribed JSON schema. Our approach involved querying the model with the designated system prompt and supplying our JSON schema, crafted using Pydantic, alongside the prompt. Subsequently, when presented with a standard user prompt, the model generates JSON responses. Despite its JSON output, there were instances where the model produced invalid JSON or irrelevant output that deviated from the user's query. We utilized the grammar feature available with llama.cpp to handle this issue.

### *D. Llama.cpp*

Language models typically require sophisticated hardware and extensive dependencies, which makes their adoption difficult in constrained environments. Llama.cpp [6] provides a lighter, more portable alternative to heavyweight frameworks. Developed by Georgi Gerganov, the primary objective of llama.cpp is to facilitate Language Model (LLM) inference with minimal setup while delivering state-of-the-art performance across diverse hardware configurations, whether local or cloud-based. This library supports inference using a range of models and their finetuned versions. For our project, we used llama-cpp-python [7], a Python binding for llama.cpp. For inference tasks with llama.cpp, the model must be obtained in the GGUF file format. GGUF is a binary file format specifically designed for storing models intended for inference using GGML, a tensor library. This format facilitates rapid loading and saving of models while offering ease of readability.

Llama.cpp supports context-free grammars, enabling precise control over the model's output. This capability allows us to impose constraints such as generating valid JSON that adhere to specified schema requirements. This technique is highly effective to ensure that the output matches your specific grammar. It achieves this by directly influencing the selection logic of the next token, restricting the model's choices to tokens that fulfill the grammar rules at each step [8]. Because of this feature, we don't have to go through the complexities of prompt engineering or post-processing efforts to ensure the generated response is a well-formed JSON [9].

GBNF serves as a format for specifying formal grammar. We crafted a grammar tailored to our requirements to ensure that our language model produces responses that adhere to our defined JSON schema. While the llama.cpp repository offers a Python script (json-schema-to-grammar.py) capable of generating grammar from a JSON schema, it currently lacks support for enums within the schema [8]. Consequently, we utilized this script to generate grammar for supported types and manually augmented the grammar to incorporate support for the enum type. Employing llama.cpp for inference ensured the generation of accurate JSON responses and significantly enhanced the inference speed compared to previous methods.

### *E. Building application using LangChain*

LangChain [5] is a framework for building LLM-powered applications. LangChain provides various functionalities that allow you to craft prompts, add conversational memory, create chains, and parse the output, thus making application development streamlined. We utilized the prompt templates provided by LangChain, which allowed us to create parameterized prompts that can be dynamically formatted with input values to create the final prompt. This flexibility allowed us to seamlessly pass different inputs multiple times, utilizing the template to assemble prompts efficiently.

Since we are developing a chatbox application, it was imperative to have memory to retain previous messages. Therefore, we integrated conversation memory, wherein each query to the LLM dynamically updates the prompt template, such that all the prior messages are added in place of a placeholder variable called "chat_history." Additionally, we created a chain so that prompt formatting and response generation from LLM can be invoked in a single function call. Finally, we utilized an output parser to ensure that the generated JSON matches the required schema. We also deployed our application as REST API using LangServe.

### *F. Game*

We used a game [10] similar to the multiplayer strategic battle game Clash of Clans. In this game, players have the opportunity to design their base environment, having resources and defensive structures. Each defense structure has its own defense and striking capabilities. The game enables the following structures:

- Tower – This defense structure can attack enemies on the ground and in the air.
- Mortars and Cannons – These defense structures can attack enemies only on the ground.
- Headquarters and Resource Buildings – These resource structures do not attack the enemies.

All three defense structures have different shot speed, range, and damage. All the defense and resource structures have some health.

The other player can attack this base using the attack structures they have. The goal is to destroy all the resources and defense structures on the base and earn stars based on the percentage of damage they have done. The player has the following attack structures:

- Shooters and Tanks – These attack structures travel on the ground and can be attacked by all three defense mechanisms.
- Helicopters – This attack structure travels in the air and can only be attacked by the Tower (that can attack enemies in the air).

Each attack structure has its moving speed, shooting speed, health, and the damage it can do. Each attack structure has a priority target that it will attack first.

Users input information regarding defense structures and their placements, as well as details about attack structures and their deployment strategies, including priority targets. This input, provided in natural language format, is then processed by the language model to generate a JSON configuration. The resulting JSON configuration delineates the base environment, outlining the types and locations of defense and resource structures. Moreover, it specifies the attack structures along with their quantities and deployment locations, while also defining their behavior, including priority targeting.

The game initiates the environment setup based on the specifications outlined in the JSON configuration. Subsequently, all attackers are deployed, prioritizing their assigned targets as per the directives provided in the JSON.

### G. Behavior Trees

Behaviour tree is a tree of hierarchical nodes that control the flow of decision making of an AI entity. At the extents of the tree, the leaves, are the actual commands that control the AI entity, and forming the branches are various types of utility nodes that control the AI's walk down the trees to reach the sequences of commands best suited to the situation. The trees can be extremely deep, with nodes calling subtrees which perform particular functions, allowing for the developer to create libraries of behaviours that can be chained together to provide very convincing AI behaviour [11]. BTs are a very efficient way of creating complex systems that are both modular and reactive. These properties are crucial in many applications, which has led to the spread of BT from computer game programming to many branches of AI and Robotics [12].

In our initial attempts, we encountered challenges when attempting to generate behavior trees in JSON format, resulting in significant setbacks. The inherent issue stemmed from the fact that behavior trees are traditionally constructed using XML format. Consequently, we were compelled to devise a custom JSON format for generating behavior trees. Due to the scarcity of examples and resources tailored to this specific format, refining our approach through model fine-tuning was unfeasible.

Initially, we attempted to guide the Language Model (LLM) by providing example prompts within the system input to generate behavior trees according to our requirements. However, the LLM's outputs did not align with our expectations, prompting a shift in focus towards generating behavior trees exclusively in XML format.

The base Mistral model (Mistral-7B-Instruct-v0.2) performed great and generated valid behavior trees for the prompts we tried. We wanted to try and fine-tune the model to see if that improves the performance in anyway. Fortunately, we stumbled upon a dataset of behavior trees, courtesy of researchers engaged in a similar endeavor utilizing LLM (specifically, Llama2) [13]. Leveraging the LoRA (Low-Rank Adaptation) technique, we conducted fine-tuning on the base Mistral model.

LoRA is a PEFT (Parameter Efficient Fine Tuning) method that decomposes a large matrix into two smaller low-rank matrices in the attention layers. This reduces the number of parameters that need to be fine-tuned. These matrices constitute the LoRA adapter. This fine-tuned adapter is then loaded on top of pre-trained model and used for inference. We fine-tuned the base model on the dataset for 4 epochs with learning rate of 3e-4, and batch size of 2. The fine-tuning took around 11 hours to complete. The results with the fine-tuned model are presented in the evaluation section.

## IV. Evaluation

### A. Game Control using LLM

Our experimentation employed the finetuned Mistral LLM to generate JSON configurations across diverse game scenarios. These scenarios included alterations in defense and structure placements, attacker deployment strategies, and target priorities. Subsequently, we meticulously scrutinized the generated JSON configurations for each scenario before executing the corresponding game instances. Our chosen LLM demonstrated exceptional proficiency in comprehending user queries and generating structured outputs reflective of the query's intent. In cases where the generated output deviated from our expectations, we iteratively queried the LLM, refining our output as necessary, leveraging the flexibility afforded by the integrated chatbox interface. Below is an example of a user query:

*Query: Generate a game configuration with three defenses. Of these three defenses, two are mortars placed at the center and center-right position. One tower is placed at the top-right corners. There are ten shooters available to us. I want to deploy five shooters from the top-right corner and five from the bottom-right corner. Also, shooters will attack the cannons first.*

The above query resulted in the following JSON –


```
{
 "environment": {
  "defenses": {
   "defense1": {
          "structure": "mortar",
          "position": "center"
   },
   "defense2": {
          "structure": "mortar",
          "position": "center-right"
         },
   "defense3": {
          "structure": "tower",
          "position": "top-right"
         }
  }
 },
 "agents": {
  "shooters": {
         "noShooters": 10
  }
 },
 "deployment": {
  "deployments": {
   "troop1": {
       "troop": "shooters",
          "DeployedNumbers": 5,
          "DeploymentPosition": "top-right"
```

```
    },
    "troop2": {
        "troop": "shooters",
        "DeployedNumbers": 5,
        "DeploymentPosition": "bottom-right"
    }
  }
 },
 "behavior": {
        "shooters": "cannon",
        "tanks": "mortar",
        "helicopters": "tower"
 }
}
```

The above JSON configuration initializes the game defense and it looks as shown in the following image.

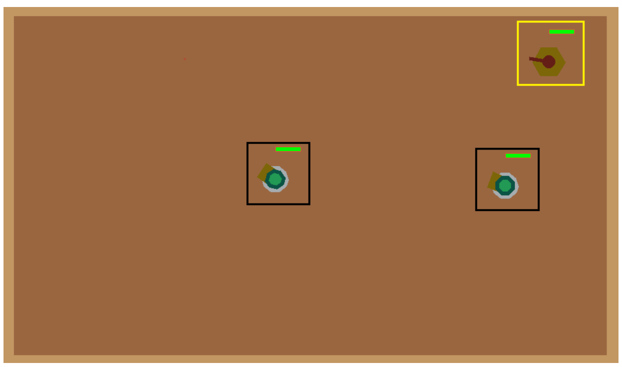

Fig. 2. The black boxes highlights the mortars placed at center and center-right position. The yellow box highlights tower at the top-right position.

The game further executes with the deployment of attackers as specified in the JSON, and they first attack their priority target as mentioned in the behavior in JSON.

### B. Behavior tree generation using LLM

The following examples shows the behavior tree generated using the base model as well as the fine-tuned model.

*Query: The behavior tree represents a robot's navigation system with arm activity. The robot must visit the location "ArucoStand", then follow the arucos given their IDs: 10, 1, and 7. Finally the robot reset the manipulator to the parking position and goes to the location "Parking". The only available actions that must be used in the behavior tree are: "MoveTo", "FollowAruco" and "ResetManipulator".*

This query was passed to both the base model and the fine-tuned model.

*Response from base model-*

```
<root main_tree_to_execute = "MainTree">
 <BehaviorTree ID="MainTree">
  <Sequence>
   <Action ID="MoveTo" goal="ArucoStand"/>
   <Action ID="FollowAruco" goal_id="10"/>
   <Action ID="FollowAruco" goal_id="1"/>
   <Action ID="FollowAruco" goal_id="7"/>
   <Action ID="ResetManipulator"/>
   <Action ID="MoveTo" goal="Parking"/>
  </Sequence>
 </BehaviorTree>
</root>
```

*Response from fine-tuned model-*

```
<root main_tree_to_execute = "MainTree">
 <BehaviorTree ID="MainTree">
  <Sequence>
   <Action ID="MoveTo" goal="ArucoStand"/>
   <Action ID="FollowAruco" goal_id="10"/>
   <Action ID="FollowAruco" goal_id="1"/>
   <Action ID="FollowAruco" goal_id="7"/>
   <Action ID="ResetManipulator"/>
   <Action ID="MoveTo" goal="Parking"/>
  </Sequence>
 </BehaviorTree>
</root>
```

The lack of discernible improvement in the output suggests that our fine-tuning efforts may have been hindered by insufficient epochs and the scope of parameters trained on the base LLM model. It appears that a more extensive fine-tuning regimen, encompassing a greater number of epochs and parameters, may be necessary to achieve the desired enhancements in performance.

## V. CONCLUSIONS

In summary, our project achieved several milestones. We effectively modified the game to accept JSON configurations, facilitating execution according to defined behaviors. Leveraging LLMs, we successfully generated valid JSON outputs and managed game dynamics accordingly. Additionally, we developed a user interface enabling prompt-driven LLM interactions and aesthetically presented JSON outputs. While full integration of behavior trees into the game proved elusive, we demonstrated the LLM's capacity to generate diverse scenarios for robot interactions. Furthermore, through fine-tuning, we enhanced the LLM's performance in behavior tree generation. In conclusion, our project underscores the potential of LLMs in generating dynamic scenarios and controlling game environments based on user inputs.

## REFERENCES

[1] R. Gallotta, G. Todd, M. Zammit, S. Earle, A. Liapis, J. Togelius, and G. N. Yannakakis, "Large language models and games: A survey and roadmap," *arXiv preprint arXiv:2402.18659*, 2024.
[2] NousResearch. Hermes-2-pro-mistral-7b. [Online]. Available: https://huggingface.co/NousResearch/Hermes-2-Pro-Mistral-7B
[3] G. Todd, S. Earle, M. U. Nasir, M. C. Green, and J. Togelius, "Level generation through large language models," in *Proceedings of the 18th International Conference on the Foundations of Digital Games*, 2023, pp. 1–8.
[4] S. Colvin. Pydantic. [Online]. Available: https://docs.pydantic.dev/latest/

[5] Langchain. [Online]. Available: https://www.langchain.com/
[6] G. Gerganov. llama.cpp. [Online]. Available: https://github.com/ggerganov/llama.cpp
[7] A. Abetlen. llama-cpp-python. [Online]. Available: https://github.com/abetlen/llama-cpp-python
[8] S. Willison. Using llama-cpp-python grammars to generate json. [Online]. Available: https://til.simonwillison.net/llms/llama-cpp-python-grammars
[9] I. Maurer. Using grammars to constrain llama.cpp output. [Online]. Available: https://www.imaurer.com/llama-cpp-grammars/
[10] J. Madav. Gaming in python. [Online]. Available: https://github.com/jatinmandav/Gaming-in-Python/tree/master/Battles
[11] C. Simpson. Behavior trees for ai: How they work. [Online]. Available: https://www.gamedeveloper.com/programming/behavior-trees-for-ai-how-they-work
[12] M. Colledanchise and P. Ögren, *Behavior Trees in Robotics and AI: An Introduction*, ser. Part of: Chapman Hall/CRC Artificial Intelligence and Robotics Series. CRC Press, 2018.
[13] R. A. Izzo, G. Bardaro, and M. Matteucci, "Btgenbot: Behavior tree generation for robotic tasks with lightweight llms," 2024.