Enhancing Game AI Behaviors with Large Language Models and Agentic AI

Ciprian Paduraru
Gameloft &
University of Bucharest
Romania

Miruna Paduraru
Electronic Arts
Romania

Alin Stefanescu
University of Bucharest &
Institute for Logic and Data Science
Romania

Abstract

Integrating advanced AI behaviors is central to creating immersive and dynamic video game experiences. This paper presents a novel approach to improving AI behaviors in games using Large Language Models (LLMs) and agent-based AI. By orchestrating various interconnected parts, we propose a framework that facilitates the creation of complex behavior trees (BTs) for non-player characters (NPCs). Our method bridges the gap between source code and visual tools in game engines and enables both technical and non-technical stakeholders to effectively contribute to the development process. We also aim to increase the diversity of observable behaviors and testability of games through the same methods. The proposed architecture is designed to be adaptable to different game engines to ensure scalability and flexibility. In a collaboration between industry and academia, we validate our approach and demonstrate its potential to improve game AI development and make it more accessible and efficient. To promote the adoption of the methods, we consider small-sized models that run on typical developer platforms without the need for external solutions or expensive computing resources.

CCS Concepts

• **Human-centered computing** → *Human computer interaction (HCI)*; **Interaction techniques**; • **Computing methodologies** → **Knowledge representation and reasoning**.

Keywords

Game AI, Large Language Models, Behavior Trees, Agentic AI, Generative AI, Non-Player Characters (NPCs).

ACM Reference Format:
Ciprian Paduraru, Miruna Paduraru, and Alin Stefanescu. 2025. Enhancing Game AI Behaviors with Large Language Models and Agentic AI. In *33rd ACM International Conference on the Foundations of Software Engineering (FSE Companion '25), June 23–28, 2025, Trondheim, Norway.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3696630.3728553

1 Introduction

In the traditional approach to game development, programmers often write the game mechanics and AI behavior directly into the source code. This practice can be a significant barrier to active participation in the development process for those without programming skills. This limits the scalability of production and limits the ability of player communities to contribute their ideas [6].

Recent advances in game engines have introduced visual tools for building complex AI systems in games [28] and simulation applications, such as behavior trees (BTs), finite state machines (FSMs), and goal-oriented action planning (GOAP). These tools aim to find a middle ground that makes it easier for a variety of users to engage in game development. However, requirements from game companies such as Electronic Arts[1], Gameloft[2], and Amber[3] reveal that integration between the source code base and visual tools remains insufficient, leading to a divide between technical and non-technical stakeholders. The game development industry is facing an increasing demand for AI systems that provide realistic and varied NPC behaviors. However, traditional development practices often require extensive programming knowledge, creating a bottleneck for non-technical stakeholders. Additionally, integrating generative AI methods into existing visual tools and source code is complex and resource-intensive. These barriers hinder scalability and reduce the creative contributions of different stakeholders.

In a close collaboration between game development companies and academia, we worked to better understand the problem and prototype different solutions and architectures to facilitate the creation of game AI and mechanisms for multiple non-programmers. In addition, we have explored the use of generative AI to meet the growing demand, reduce costs, and minimize human effort. It is worth noting that our solution can be extended to other applications such as education, simulations, entertainment, and more, as they utilize the same toolset of game engines [26] as the publicly available well-known Unreal Engine[4] and Unity Engine[5].

In the current study, we focus exclusively on the generation of Behavior Trees (BTs), as they offer the strongest semantics among all other available methods and can imitate all other methods in the game domain [28], or other industries such as robotics.Beyond games and robotics, the proposed methods have potential applications in education and simulation. For example, adaptive NPC behaviors could be used to create realistic training scenarios or personalized educational tools. By making AI development more

[1]https://www.ea.com
[2]https://www.gameloft.com
[3]https://amberstudio.com
[4]www.unrealengine.com
[5]https://unity.com/

accessible, we aim to empower creators across industries the opportunity to leverage advanced AI capabilities.

We summarize our contributions as follows:

- First open-source work considering generative AI for behavior trees using both basic fine-tuning and Retrieval Augmented Generation (RAG) on large code databases of nodes. Our methods have been developed and evaluated for the gaming industry, but can also be applied to other domains such as robotics or simulation. The validation of our results was carried out in collaboration with game developers and projects. Source code available at: https://github.com/unibuc-cs/GameAILLM.
- We leverage the state-of-the-art capabilities of large language models (LLMs) and Agentic AI concepts [12] to generate new behaviors based on user queries which can increase the diversity of observable actions of the NPCs, and the testability of the products at the same time. Methods such as self-reflection, human-in-the-loop, and corrective methods are made through a combination of Agentic
- We leverage the state-of-the-art capabilities of large language models (LLMs) and agent-based AI concepts [12] to generate new behaviors based on user queries that can increase the variety of observable actions of NPCs while increasing the testability of products. Methods such as self-reflection, human-in-the-loop, and corrective methods are developed through a combination of Agentic AI and software engineering. AI and software engineering efforts.
- On a technical level, the proposed architecture is abstract and provides a separation between the generation process and the source code, making it cross-compatible and adaptable to other academic or industrial projects. Moreover, in line with the latest trends in game development, we provide a solution to bridge the gap between the source code base and the available visual tools.
- To ensure the sustainability of our proposals from a computational point of view, we use various NLP-based filtering and fine-tuning techniques on small LLMs such as Llama 3.1 (8B) [5]. Our goal is to enable the efficient generation of AI mechanisms for games on user devices. We refer to our fine-tuned LLM as *BTreeGenLLM*, while the AI agent that supervises the entire process is called *BTreeGenAIAgent*.

The rest of the article is organized as follows. The next section describes related work in the same field of interest. Section 3 provides an introduction to behavior trees and shows that the techniques used provide a separation between the product to which they are applied, the generation process, and the visualization of the results. The proposed generative methods and the pipeline are described in detail in Section 4. Their evaluation follows in Section 5. Finally, conclusions and future work are presented in the last section.

2 Related work

The literature review focuses on two perspectives: a) the previous work related to behavior trees and generation with LLMs and b) specific topics related to LLMs or agentic AI that were considered in the development and implementation of the methods proposed in our paper.

Behavior Trees and LLMs. In our research, we found that previous work, while informative to our work, does not meet the requirements for applying BT and LLM generation to generate AI behaviors for games, as they have different goals. As described below, the only documented applications have been in robotics for short-term action planning with a limited number of nodes and without the ability to extend the fine-tuned model with additional nodes and reasoning at runtime.

The work in [1], presents a novel framework that uses Large Language Models (LLMs) to generate Behavior Trees (BTs) for robotic assembly tasks. The framework leverages the natural language processing and inference capabilities of LLMs to create task plans in BT format, reducing manual effort and ensuring robustness. Similar to our case, the authors evaluate the performance of fine-tuned LLMs with fewer parameters and show improved success rates in BT generation through contextual learning and supervised fine-tuning. The article is informative for our work. However, it uses a handful of nodes (less than 20) and engineering mechanisms that might work in practice for some types of applications in robotics, but not for game AI development. The authors train the LLM on this limited number of nodes, without the support of dynamically added new nodes, which is also a requirement given our goals. As mentioned earlier, training small models on new data not only leads to a loss of previous information [8], but it is also impractical to fill LLM's models with large context. We address these problems with NLP techniques, filtering, and RAG methods as described in Section 4.

A continuation in the field of robotics is the work in [24], which pursues the same goal of partitioning and finding action plans for robot control. It uses LLMs to correct errors that are beyond the capabilities of the task planner, both in planning and execution. The method improves the transparency and auditability of robot policies, ensuring that they are readable and adaptable to new tasks or unpredictable environments. The study shows that the method is capable of solving various tasks and errors and permanently updating the policy to solve similar problems in the future. Their work is insightful for us and drives the ideas for incorporating the human-in-the-loop concept.

In [13], the authors explore the use of large language models (LLMs) to automate the creation of behavior trees (BTs) for complex tasks. The authors mention that the traditional manual creation of BTs is inefficient and relies heavily on domain knowledge while existing automatic methods struggle with task complexity and model adaptability. Their study explores how LLMs can support these operations and what advantages they have in representation and inference, which includes data synthesis, model training, application development, and multi-level data validation. Although the presentation is insightful, it is only a theoretical exploration of the possibilities, with no implementation, evaluation, or target application.

LLMs related work. The work [20] gives an overview of the development and methods of fine-tuning Large Language Models (LLMs) and serves as a reference point for our methods. It compares different approaches, presents a seven-step pipeline, and explores advanced techniques such as Low-Rank Adaptation (LoRA) and Mixture of Experts (MoE). The paper also addresses the challenges of scalability, privacy, and accountability and provides valuable insights for researchers and practitioners in the field.

Recent developments have significantly improved the efficiency, accuracy, and factuality of large language models (LLMs) with retrieval extension. Our framework includes three main techniques to realize Adaptive-RAG in the proposed BTreeGenAIAgent, as detailed in Section 4. One technique introduces an adaptive QA framework that selects appropriate strategies based on query complexity to improve performance on open QA datasets [10]. Another method, Corrective Retrieval Augmented Generation (CRAG) [27], evaluates the quality of documents, uses web searches, and applies an algorithm to refine RAG-based approaches. In addition, the Self-Reflective Retrieval-Augmented Generation (Self-RAG) [2] framework combines adaptive retrieval and self-reflection to improve quality and factuality, outperforming models such as ChatGPT [17] in various tasks. Taken together, these studies illustrate significant progress in the retrieval-based generation of LLMs. Technically, we use the Faiss library [4], [11], known for its fast vector similarity search, to segment the documents of the dataset into chunks and support the indexing operations in RAG. Faiss provides a comprehensive toolkit of indexing techniques and related primitives for tasks such as search, clustering, compression, and vector transformation.

Agentic AI refers to advanced AI systems that solve complex, multi-step problems autonomously using sophisticated reasoning and iterative planning. The work in [12] focuses on the development and evaluation of AI agents and emphasizes the importance of optimizing accuracy and cost in real-world applications. From the user's perspective, our goals are similar to those of the commercial tool Unity Muse[6], which provides an assistant within the Unity Engine as a subcomponent to help users create BTs. However, we cannot make a comparison as it is not documented and is only available in the parent environment.

3 Behavior Trees: Basics and Abstractions

3.1 Introduction to Behavior trees

Behavior trees (BTs) are an innovative and adaptable approach to artificial intelligence (AI) development that is widely used, particularly in the field of video game development. Although these frameworks originated in robotics, their application in games has increased significantly due to the modular, scalable, and user-friendly nature of BTs. This method allows the creation of complex and dynamic behaviors for non-player characters (NPCs) within a structured and maintainable system.

At their core, behavior trees are hierarchical architectures with nodes assigned to specific actions, conditions, or behaviors. These nodes are systematically arranged in a tree structure, with the root node representing the overarching behavior and the leaf nodes representing specific actions or conditions. The evaluation of these nodes follows predefined rules and priorities, which generally run from left to right and from top to bottom. An example is shown in Figure 1. Each such data structure can be reused by many agents in a game or simulation application. For example, all persons in a soccer game could use the same BT. Differences in the reactions of individual people can be handled with the data structures *Blackboard*, which are dictionaries with keys/values that can be read and written by the source code and BT together.

[6]https://unity.com/products/muse

Behavior trees offer several advantages over conventional finite state machines (FSMs) in the development of game AI. One notable advantage is their modularity, which enables the development of reusable and interchangeable behavior modules. This modular approach simplifies the design and modification of AI, as developers can customize individual nodes without disrupting the entire framework. BTs have greater scalability than FSMs, making it easier to manage complex and nested behaviors. In addition, behavior trees are ideal for processing parallel and concurrent actions. Composite nodes, such as selectors, parallels, and sequences (Figure 1), allow AI systems to evaluate multiple behaviors simultaneously or follow a specific sequence. This capability improves the realism and responsiveness of NPCs and allows them to exhibit more complicated and adaptive behaviors in response to evolving game conditions.

In short, the semantics of composite nodes is as follows. The selector node executes by picking up each of its children in any order until one of them succeeds or all of them fail. In contrast, a sequence node must execute all children from left to right to be considered successful. If one of the children fails, the execution is halted and considered failed. Parallel nodes allow the execution of two or more nodes without dependencies between them, in contrast to sequences, which must always be executed from left to right.

In summary, behavior trees provide a robust framework for implementing sophisticated AI behaviors in video games. Their modular design, scalability, and ability to manage concurrent actions make them an optimal choice for creating dynamic and engaging NPCs. By integrating them, game developers can create AI that enhances the player's experience and increases the complexity of the story.

3.2 Adaptation of BT nodes for being consumed by LLMs

The intermediate representation of BTs in JSON format uses four categories of nodes:

(1) Composite nodes: define the root of a branch and the basic rules for the execution of this branch, e.g. selectors, parallels, sequences, loops, etc. An example is also the node *RepeatUntilFailure* in Figure 1, which executes the subordinate branch until it fails (search for a new waypoint in a certain room until no more can be found).
(2) Task nodes: the leaves of a BT. These nodes are the actions that can be executed and have no output connection. For example, execute another behavior tree (hierarchically), go from point A to B, play an animation, display a chat dialog, etc.
(3) Conditional nodes: are used to change the environment and test certain things. For example, whether the player is blocked, whether a certain item is nearby, etc.
(4) Modifiers: These types of nodes change the behavior of subordinate branches. For example, they reverse the result, make a child run for a certain time, etc.

These design decisions in the intermediate format were made after experimenting with how LLM could better understand and create correct and diverse trees. For example, the nodes of type *Service* in the Unreal Engine, which are tasked to perform background checks, can be customized in our representation by a union of *Conditional* and *Task* nodes. This split was necessary to make it

```
Description:
I want to create a behavior tree for an agent that has the following blackboard (memory) and types:
name: string; speed: float; current animation: string; current room: string
Create a behavior for a patrolling officer that performs the following actions.

Behavior:
Randomly select one of the following three branches below.
Branch 1: If a door is open, go and check it.
Branch 2: Move between all patrolling waypoints in the room and then play an idle animation for 5 seconds
Branch 3: Find a chair in the room, wait on it for 2 minutes, and check your phone."

Nodes Discarded:
FindRoom

Nodes Recommended:
RepeatUntilFailure
FindObject
```

Listing 1: An example of a natural language user request to create a behavior tree with a given specification of both the behavior and the memory used by the AI agent that uses it during its execution. The query must be split into two parts to understand the processing of features using NLP techniques, as highlighted in the example: a general description and the actual desired behavior. Optionally, the user can recommend discarding or using certain nodes to help the AI agent make a decision.

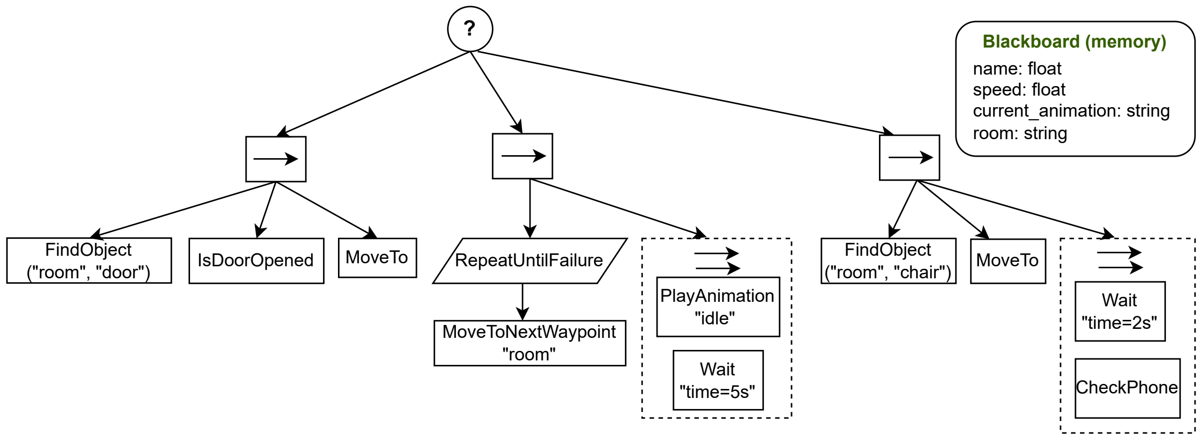

Figure 1: The behavior tree to be created by the proposed methods using the user query in Listing 1. It contains sequence nodes (arrow sign and rectangle), selectors (question marks, circles), parallels (two arrow signs), and leaf nodes.

easier for the LLM process to recognize the purpose of each node. Compared to Unity, the *Modifiers* class was needed so that the LLM explicitly understands how to control the results or execution management of the children. However, the intermediate representation is used internally for the generation process, while the final output can be converted back to the format specific to each game engine.

3.3 Format abstraction

In our proposed approach, we have developed a high-level solution to decouple the creation of behavior trees from specific game engines and their underlying source code. This solution uses an intermediate representation of nodes aggregated by reflection mechanisms, Figure 2. The nodes are made visible directly in the source code through the use of decorators and tags. In addition, a textual description of each node must be given, corresponding to a natural language comment, so that a match can be made between the user's intentions and the functions implemented and available for LLM output. A few illustrative examples can be found in Listing 3.

In order to close the gap between the intermediate representation and the visual tools of the various game engines, an adapter must be developed for each engine. This adapter translates the JSON-based intermediate representation into a format that is compatible with the visual tool (as shown in the right-hand side of Figure 2). In this way, the methods ensure that the format of the intermediate representation of the behavior trees is dynamically adaptable and universally applicable to different game development platforms, allowing for greater flexibility and scalability in the development of AI behaviors.

The repository contains examples of implemented adapters for the most commonly used public game engines, Unity and Unreal.

4 Methods

4.1 Datasets and fine-tuning

Dataset building. Since there is no public dataset for behavior trees, we had to create one from scratch (we plan to clean it of confidential data and publish it after review). To build a robust dataset for fine-tuning, we combined human-annotated examples with synthetic BTs generated by GPT-4o. Synthetic BTs are validated through compilation and testing. This dual approach reduces manual effort while maintaining high-quality training data.

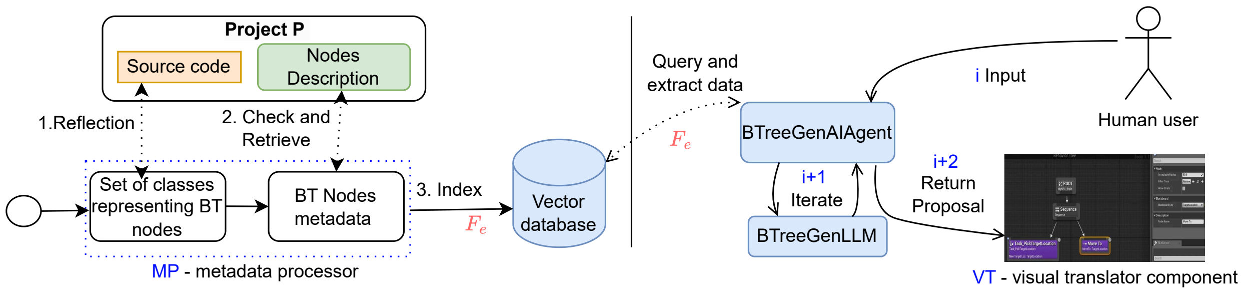

Figure 2: The left part of the figure shows the process for processing (indexing) existing and new node metadata and related items that are further required for the LLM-based agent to create behavior trees from natural language. Updates to the node database can be made at any time. The right side of the image shows the use of the LLM model and high-level AI agent orchestration through multiple successive iterations to create or edit BTs based on user requests. The output consists of two parts: a JSON format and a visual BT representation.

Specifically, we looked at the available implementations of two projects: Gangstar New Orleans[7] and Asphalt[8], which consist of 57 already implemented BTs with a total of 342 nodes.

The pipeline for data collection follows. First, the nodes and descriptions of each project are collected using the automated methods described above (where descriptions were missing, they were added by software engineers). Then, each of the available BT is translated into a JSON format required by the methods and provided with a query description similar to the one in Listing 1. This pair forms the human-annotated fine-tuning dataset $\mathcal{D}_H$, as shown in Eq. (1). Starting from this set, the algorithm in Listing 2 is used to create a synthetic dataset consisting of synthetic BTs. Until a certain number of tests $NS = 500$ are created, the teacher model (the GPT-4o model in our experiments) is used to create synthetic tests as follows. First, the model is asked to create natural language variations of the ground truth behavior, keeping the same semantics. Then it is asked to create a new synthetic behavior tree using the methods described in the following subsections. If the test passes the compilation and the tests available in the original reference, it is added to the synthetic dataset $\mathcal{D}_S$. The final fine-tuning dataset is the union of the human-annotated and synthetic datasets.

$$\begin{aligned}
&\mathcal{D}_H = \{(instruction = \text{"}Create\ a\ ..\text{"},\ answer = \text{"}\{..JSONbtree..\}\text{"})\}\\
&\mathcal{D}_S = create\_synthetic\_data(\mathcal{D}_H, NS)\\
&\mathcal{D} = \mathcal{D}_H \cup \mathcal{D}_S \qquad (1)
\end{aligned}$$

```
def create_synthetic_data($\mathcal{D}_H$, $NS$):
  $\mathcal{D}_S = \emptyset$
  while $|\mathcal{D}_S| < NS$:
    $bt$ = uniform sample of $\mathcal{D}_H$
    # Get a varied natural language description
    # with the same meaning
    $bt_{desc\_new}$ = TeacherModel(vary $bt$'s instruction)
    # Create a new btree with the same nodes as bt
    $bt\_new$ = TeacherModel($bt_{desc\_new}$, nodes($bt$))
    # Test whether the new tree is
    # syntactically correct.
    if not compiles($bt\_new$): continue
    # are tests available and have they passed?
    if not testbed($bt\_new$): continue
    # add to the final output
    $\mathcal{D}_S$ = $\mathcal{D}_S \cup bt\_new$
  return $\mathcal{D}_S$
```

Listing 2: Pseudocode for creating a synthetic data set.

[7]https://www.gameloft.com/game/gangstar-new-orleans
[8]https://www.gameloft.com/game/asphalt-8

Fine-tuning. Based on the Llama 3.1-8B-Instruct model [5], we use the dataset $\mathcal{D}$ to fine-tune the *BTreeGenLLM* model. Samples from the dataset are processed by the model and their similarity to the corresponding ground truth is evaluated using cross entropy as a loss function [9]. To measure the performance of the model at certain intervals, we use the perplexity metric [15]. The fine-tuning method used is LoRA (Low-Rank Adaptation) [7], which allows to add updated weights to each selected layer of the base model while keeping the computational cost manageable. Since the completed prompts can become very large, a completion-only training [20] was used for fine-tuning to reduce the number of tokens fed into the LLM.

Fine-tuning a model to generate behavior trees (BTs) using large language models (LLMs) is important for the accurate and efficient generation of AI behavior, as the evaluation section shows. Fine-tuning allows the small LLMs to capture the complex structure of BTs and enable the correct construction and modification of behavior trees using a certain number of nodes [1], [20]. In addition, knowledge of the basic composition of the nodes and the structures of the BTs, which are the same for different projects, is important, even if they behave differently from case to case. This approach ensures a basic understanding of the BT components and improves the versatility and adaptability of the model to different scenarios. As suggested by [13], when training LLMs to create behavior trees, this basic knowledge greatly improves the model's ability to handle complex tasks. Understanding the use of modifier nodes and their role, especially when reusing nodes to reverse branching results or to execute within a certain time limit, is essential. According to the study in [1], leveraging the robust reasoning capabilities of LLMs to build behavior trees can lead to significant improvements in efficiency and reliability. These scientific findings provide a solid theoretical basis for fine-tuning LLMs to ensure that they are well-equipped to effectively generate and adapt behavior trees.

4.2 Internal knowledge representation and RAG.

The Retrieval Augmented Generation (RAG) process ensures that newly added nodes of a project under development can be used without retraining the model. By indexing node metadata and project documentation in a vectorized database, our framework dynamically retrieves relevant information during runtime, enabling continuous updates of the NPC behavior trees.

Two sources of indexed data are considered (but more can be added dynamically using the existing pipelines):

- The documentation of the project, design documents, manuals, (format examples: pdfs, docs, urls, video transcripts).
- The textual description of the implemented functionality, as shown in Listing 3.

To apply the general notion of RAG in this context, the knowledge of the project is indexed and stored for quick retrieval, as shown in the left side of Figure 2. The first step is to use the reflection mechanisms of programming languages and frameworks, which means that metadata about classes, functions, and types can be retrieved after compilation. Even though C++ does not have reflection by default, there are certain open-source frameworks or methods that do, such as those built into the Unreal Engine. One requirement for our methods is to provide a container in which each of the BT nodes identified by reflection (taking into account base classes and type hierarchies) must have a JSON-formatted description within a special container. Examples of this can be found in Listing 3. The information about all BT-related nodes is retrieved and converted into an embedding space (floating point numbers) using a record conversion model [21] - using the F_e function - and then indexed for fast retrieval. The embedded data is then uploaded to a vectorized database using the Faiss [4] library to efficiently store data, support indexing, and manage continuous updates. The pipeline also supports adding new nodes or removing previous nodes as needed. During inference, as the human iterates over the results with the agent and the LLM model, the database is queried with the human's input, and the same embedding function is used to index the data.

4.3 BT creation from natural language with Agentic AI

The main flow of constructing a behavior tree from the user input described in natural language is highlighted in Figure 3. It comprises several steps, which are described below and are executed by an AI agent prototyped with the LangGraph framework[9].

A complete example of the user input can be found in Listing 1 The user input contains a general description of the entity on which the behavior is to be executed, along with the variables that represent the table (memory) required for the BT. This is followed by the actual description of the intended behavior of the entity that owns it. Optionally, a selection of node names can be specified that are to be discarded or recommended from the user's point of view.

[9]https://langchain-ai.github.io/langgraph/

4.3.1 Filtering. First, a filtering process takes place based on the user's input, which we further denote as *Task*.
There are two outputs in this step:

A. The nodes from the database that are closely related to the $Task$. This is needed to create a smaller context for the agent, as it would be impractical to send the description of all nodes as a context.
B. A set of examples from a data set that together provide structural examples for the $Task$. These are reused as context for the LLM, using the few-shot examples prompting strategy.

A. Filtering nodes. As motivated by the study in [23], semantic extraction of the meaning of the behavior from the user description can be done with Semantic Role Labeling (SLR) [16], which assigns semantic roles in the form of predicate arguments, e.g. who did what, where and how?

More specifically, the filtering method in our case requires an SRL model to annotate the user's query based on some selected roles: Agent (the doer), Patient (the entity performing the action), Predicate (the verb describing the action), Instrument (the means by which an action is performed), Condition, Place, Time, and Goal (the purpose of the action).

Listing 4 shows an example of annotating part of the user input. Technically, several methods can be used to obtain such annotations. For example, we initially used the method in [25] or the GPT-4o model to check the state of the art. However, from a practical point of view, using a different model, e.g. a deep learning-based model (e.g. DeepStruct[25] with 10B parameters), or an external model would require significant computational effort or financial resources.

Considering this, our research focused on using the same fine-tuned base model, i.e. BTreeGenLLM, to perform this annotation. In short, the input submitted by the user is first parsed using the model to split the phrases into sentences, each with its predicate. Then the same model is asked to annotate the selected roles for each of these sentences, if found. This is done using prompt engineering (for space reasons, the experiments and prompts can be found in the repository).

```
Root: Choose one of the three branches below at random.
"Root": [
 {"agent" : "Entity",
  "predicate" : "choose"
  "patient" : "one the three branches below",
  "condition": random,
 }]
Branch 1: If a door is open, go and check it.
"Branch 1": [
 {"agent": "Entity",
  "predicate": ["go", "check"],
  "patient": "it (the door)",
  "condition": "If a door is open",
 }]
Branch 2: move between all patrolling waypoints in the room
 and then play an idle animation for 5 seconds.
"Branch2": [
 {"agent": "Entity",
  "predicate": "move",
  "patient": "between all patrolling waypoints",
  "location": "in the room"
 },
 {"agent": "Entity",
  "predicate": "play",
  "patient": "an idle animation",
  "time": "5 seconds"
}]
```

Listing 4: An example of user-defined semantic role labeling, shown for the root node and the second branch evaluation.

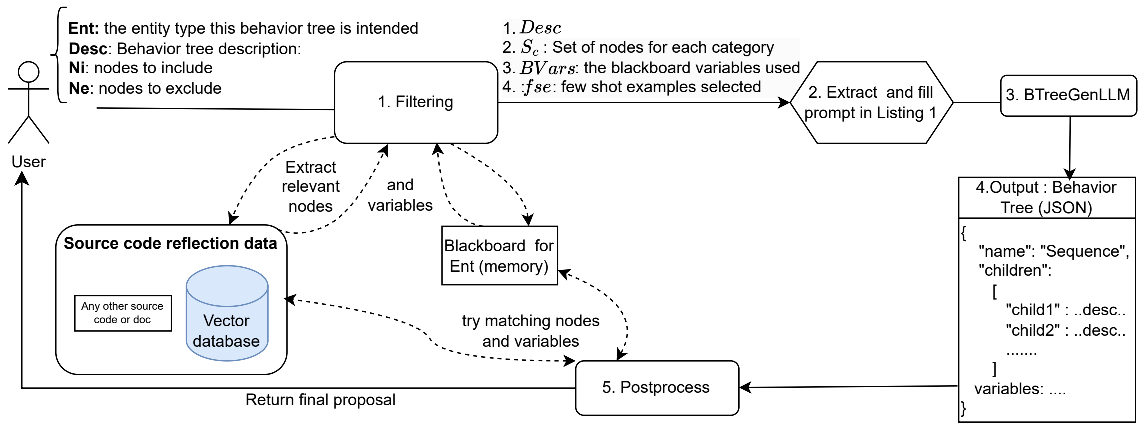
Ent: the entity type this behavior tree is intended
Desc: Behavior tree description:
Ni: nodes to include
Ne: nodes to exclude
User
1. Filtering
1. Desc
2. S_c : Set of nodes for each category
3. $BVars$: the blackboard variables used
4. :fse: few shot examples selected
2. Extract and fill prompt in Listing 1
3. BTreeGenLLM
4.Output : Behavior Tree (JSON)
{ "name": "Sequence", "children": ["child1" : ..desc.. "child2" : ..desc.. ....] variables: .... }
Extract relevant nodes
and variables
Source code reflection data
Any other source code or doc
Vector database
Blackboard for Ent (memory)
try matching nodes and variables
5. Postprocess
Return final proposal

Figure 3: The process of creating a behavior tree from user input. First, a filtering process is performed based on a description of the intended behavior and optionally the nodes that the user wants to forcibly include or exclude. The generated output (step 1) is then used to fill the prompt shown in Listing 5 (step 2). To do this, the BTreeGenLLM is used and an initial suggestion is made (steps 3-4). Before the final response is returned, final processing takes place, attempting to assign the variables on the agent's blackboard (memory) as references and checking whether the proposed nodes can be linked to those in the source code

```
{"AllNodes": {
 "Modifiers": [...
    { "name": "Repeat Until Failure", "description": "Repeats until the child node fails",
      "parameters": {"child": "AnyNode"}
    }],
"Tasks": [...
    {"name": "MoveToNextWaypoint", "description": "Moves the agent to the next waypoint. Returns false when there are no more waypoints",
     "parameters": {"location": "string"},
     "returns": "bool"
    },
    {"name": "FindObject","description": "Finds an object in a location",
     "parameters": {"location": "string", "object": "string"}, "returns": "Target",
    }],
"Conditionals": [...
    {"name": "IsDoorOpened", "description": "Checks if the door is opened",
     "parameters": {"target": "Target"}, "returns": "bool"
    }],
"Composites": [...
    {"name": "Sequence", "description": "Execute branches in order until one fails or all succeed",
     "parameters": {"children": ["AnyNode"]}},
    {"name": "Selector", "description": "Execute branches in order until one succeeds or all fail",
     "parameters": {"children": ["AnyNode"]}
    }]
},
```

Listing 3: A selection of some nodes from each category that will be used in the rest of the presentation. The purpose of the textual description is to establish the link between the user requests and the availability of the required functionalities.

For each of the four node classes used, the agent is prompted (through prompt engineering) to retrieve the nodes that relate to the union of the roles and their mapping to the relevant nodes. While, customization, the current mapping is as follows. The class *conditional nodes* can be retrieved via the semantic role *condition*. The *modifier nodes* are accessed via the roles *time* and *predicate*. Finally, the type *task node* is largely covered by the role *predicate*.

B. Filtering examples. A few examples of behavior trees are retrieved contextually from a set of predefined generic examples (extensible on the user side, ∼ 20 at the moment in the repository) using the previous set of filtered nodes. Each of these examples is tagged to represent the nodes or combinations used, e.g. tags: non-deterministic selector, conditional nodes in sequence, followed by a task, etc. The selection mechanism selects three examples that are as close as possible to the tags of the previously selected nodes by using a similarity formula, Eq. (2). Listing 5 shows the prompt used to create a behavior tree. The four categories of nodes are represented as JSONs. The $few\_shot\_examples$ field is filled with the same format as the intermediate format used for BTs.

$$Score(Ex_i) = \sum_{t \in AllTags} \mathbb{1}(t \in Tags(Ex_i)) \quad (2)$$

4.3.2 LLM interaction. By providing the description of the selected nodes, their parameters, and the Semantic Role Labeling (SRL) processing of the user request, the LLM can have a higher chance to match them and provide a correct output. A concrete example of this matching capabilities can be observed in the output presented in Listing 6, and the SRL sketched in Listing 4. In the description of the second branch of the user's behavior tree input, the SRL identified that the predicate is *move* and the location is *between all* *patrolling waypoints in the room.* While there is a *Move* node in the source-code repository's nodes, there is no one exact match for the second part. However, the model recognizes that it can combine two existing source-code nodes:

- RepeatUntilFailure: according to its description it can run the children until it fails
- MoveToNextWaypoint: its description in Listing **??**) says that the node will move to the available waypoint in a room, returning false when all have been visited.

In the subsequent phase, the filtered output from step 1 is utilized to populate the template prompt, as illustrated in Listing 5. This step leverages the BTreeGenLLM, generating an initial suggestion based on the provided template (steps 3-4).

4.3.3 Post-processing. Before the response is provided in JSON format, additional processing steps are performed to increase the probability that it is correct or expected by the user.

Outside of basic text processing and formatting, the first task is to link the variable names generated by the LLM to references in the agent's memory (Blackboard). As a concrete example in Listing 6- Lines 6, 7 and 21, the reference *door* is the previous object found by calling the function *FindObject*, where all three are given as "Result of FindObject". To match the variable name with the correct reference, a temporary variable is created for the object return, and the mapping between the string variable and the reference is performed with a least common ancestor query to find the corresponding object.

Even with fine-tuning, the model can hardly distinguish when to use parallel nodes efficiently. For example, the output in Listing 6 is intentionally left without this processing step. The Wait and CheckPhone tasks are initially in separate nodes, but there is clearly no dependency and it can allow the runtime mechanism to execute them in parallel. A simple rule states that if two or more sequence tasks at the same level have no dependency, a parallel parent node is added. The result of this correction is shown in Figure 1. Note that the same rule applies to the PlayAnimation and Wait tasks.

Other tokenization rules such as elimination of spaces/tabs (e.g., task "Check Phone" instead of "CheckPhone"), use of plural for objects instead of singular or page number, use of multiple children in different lines in SRL instead of lists, etc. were performed to increase user satisfaction as shown in the Evaluation section.

5 Evaluation

The evaluation of the proposed methods follows two research questions that were discussed with colleagues from the game industry:

- **RQ 1**- *Quantitative*: What is the degree of correctness of the BTreeGenAIAgent?
- **RQ 2**- *Qualitative*: How does it help different stakeholders create AI behaviors for their products? Are the methods suitable for use by the end user?

The motivation for *RQ 1* is that two key problems in game development are the lack of automated testing methods and the diversity of AI characters. The ability to create diverse, correct behavior trees (BTs) on a large scale can address both problems. First, the BTs created can serve as tests, as the nodes used can hierarchically touch a significant portion of the source code. Intuitively, users can release the functionality for testing and let the BTreeGenAIAgent create functional tests. This is also motivated by the work in [18], which considers BTs as a testing method for games.

RQ 2, on the other hand, is the metric that indicates how useful an assistant for writing BTs is in practice for the various stakeholders in production environments. One of the problems observed in the industry is that the lack of diversity of AI characters is a scale problem since usually only software engineers are typically able to write BTs. The main motivation of our work is to create a simpler method for non-technical stakeholders to write good BTs. Observations, improvements, and future work can be derived from these experiments.

5.1 Setup

Since there is no publicly available open-source method for generating BTs with LLMs under the stated objectives, the LLM models used for comparisons within the BTreeGenAIAgent are: a) the teacher model, GPT-4o, b) the base vanilla version LLama3.1-8B-Instr, c) the fine-tuned BTreeGenLLM.

To perform tests in a development environment where the model does not need to be retrained between short iterations of the projects where new nodes may be added, we withhold 50 of the total 342 nodes at each fine-tuning time during the evaluation period of *RQ 1*. The agent only had access to these nodes via the RAG methods (source code reflection, filtering) mentioned in Section 4, while the rest was available during fine-tuning. Simiarly, during *RQ 2* evaluation we not only withhold 50, but also allow developers to write new nodes and ask the agent to use them either explicitly or implicitly from the input query.

The upper limit for the number of nodes per type was restricted by default to $MaxNodes = 10$ for each category, although the value is adjustable and the user even could change it during the experiments at each input query. From our experiments, it is clear that embedding an LLM in a medium with environmental rules is a *must*. In our study, as detailed in Section 4, the BTreeGenAIAgent is an agent that solves various aspects, such as providing insights into the project implementation, performing post-processing tasks to access the source code, self-reflecting the given answers, and asking for user help when needed. This is one of the reasons why we are not directly comparable to the work of [1], which applies the techniques in a constrained robotics environment. Another reason is that fine-tuning the model after each newly added node implementation is firstly impractical and secondly, the small scale will eventually forget the previously stored information [19].

5.2 Quantitative evaluation (RQ 1)

The same teacher model and algorithm for creating the synthetic dataset mentioned in Section 4.1 are used to create an evaluation dataset $\mathcal{D}_{eval}$ consisting of 200 new BTs, but this time without checking if they are correct (compile or pass any test).

Table 1 shows the percentage of correctly generated BTs, where correctness means that the BTs are compiled and executed in a testbed scenario without crashes or asserts. The evaluation is performed for two sets of depth levels given by the input query (e.g. there are two levels in the Listing 1). The results indicate that small models are able to match or come close to the performance of much larger models when fine-tuned even on small datasets.

```
You are given the following set of task, composite, modifier, and conditional nodes in a JSON format.
Note that ``AnyNode`` is the type of any node, and ``string`` is the type of a string.

Available task nodes: {var_tasks}
Available composite nodes: {var_composites}
Available modifier nodes: {var_modifiers}
Available conditional nodes: {var_conditionals}

Provide me with an answer based on a description in the form of a tree structure in a JSON object format.
If the behavior described cannot be found in the available actions, include it in the JSON object anyway (and be sure to create a description for it).
If the description cannot generate a meaningful tree, please answer as follows: Cannot be generated.
The nodes of the modifier set are placed in the hierarchy before the branch they modify. If the modifier has multiple children, first insert a sequence node (single
       child of the modifier) that contains the children nodes.

{few_shot_examples}

Please create the answer to this behavior tree description given by the user:
Q: {description}
A:
```

Listing 5: Prompt to create a behavior tree in JSON format after providing a description in natural language, a set of nodes filtered by category, and some selected examples. The purple-colored variables in which these specifications are made are filled at runtime in the pipeline from Figure 3.

```
{"name":"Selector","parameters":{"type": "random",   # Root
 "children":[
  {"name":"Sequence","parameters":      # Branch 1
   {"children":[
    {"name":"FindObject","parameters":{"location":"room","object":"door"}},
    {"name":"IsDoorOpened","parameters":{"target":"result of FindObject"}},
    {"name":"MoveTo","parameters":{"target":"result of FindObject"}}
  ]}},
  {"name":"Sequence","parameters":      # Branch 2
   {"children":[
     {"name":"Repeat Until Failure","parameters":
      {"children":[
        {"name":"MoveToNextWaypoint","parameters":{"location":"room"}}
     ]}},
     {"name":"PlayAnimation","parameters":{"animation":"Idle","loop":true,"speed":1.0}},
     {"name":"Wait","parameters":{"time":5.0}}
  ]}},
  {"name":"Sequence","parameters":      # Branch 3
  {"children":[
    {"name":"FindObject","parameters":{"location":"room","object":"chair"}},
    {"name":"MoveTo","parameters":{"target":"result of FindObject"}},
    {"name":"Wait","parameters":{"time":120.0}},
    {"name":"CheckPhone","description":"Agent checks their phone for updates or messages"} ,
  ]}}
]}}
```

Listing 6: The result of executing the methods for the example query in Listing 1 (with partial post-processing) corresponds to the tree expected in Figure 1.

The table also compares the percentage of nodes that were covered (used) during the evaluation $\mathcal{D}_{eval}$. Considering that the original set of 57 tests covered only 53%, we can conclude that the proposed pipeline can indeed solve the problems raised for *improving diversity and testability through behavior generation.* The coverage and correctness can be correlated with the number of nodes retrieved by the filtering method, as shown in the same table. The results, *correctness may be lower if the nodes that could be useful are filtered out, or confuse the model altogether if too many nodes are provided.*

Table 1: Evaluation of correctness, nodes covered and average number of nodes filtered by the model, split by the number of depth levels in the input query and model type.

Depth	Avg correct percent, Nodes covered percent, Avg nodes per query rounded								
Model	**GPT-4o**			**Llama3.1**			**BTreeGen AIAgent**		
Levels 2-3	76%	83%	16	34%	49%	8	69%	78%	14
Levels 4-5	45%	87%	19	8%	54%	9	42%	81%	15

5.3 Quality evaluation and computational resources (RQ 2)

With this in mind, the evaluation tried to assess the extent to which the BTreeGenAIgent helps both technical (group *GT*) and non-technical (group *GN*) people to create BTs. The core of the discussed methods was evaluated over ∼ 8 months by 7 people in *GT* and 13 in *GN*, using the same products mentioned in Section 4.1. However, improvements were made during the experiments based on the feedback received. The models were asked to create an explanation for the output. An example can be found in Listing 7.

First, using the latest version, each person was asked to write 20 BTs with BTreeGenAIAgent in their work area, only 2 to 5 depth levels, and to evaluate the three LLMs in comparison. Note that each node can in fact be an instance of a different behavior three, so the evaluated number of levels does not limit the potential creativity of users.The *GT* group was also asked to implement 10 new nodes in their work area and try to get the model to use them from the input query. The motivation for using a seemingly small number of depth levels for BTs stems from actual use cases where developers prefer to group logic hierarchically. For example, nodes can often represent the invocation of complete other BTs, each with their behavior, descriptions and memory. This is similar to the concept of aggregation in object-oriented programming.

For each group, we measure the averaged preference for each model and show the results in Table 2. The key observation is that the fine-tuned version made it possible to understand the basics of the descriptions of BTs and project nodes, as intuitively explained in Section 4.1. It is clear that the BTreeGenLLM model is preferred over the vanilla version, but still lags behind the GPT-4o model. This was to be expected as the model only has 8B (billion) parameters and our goal was to tune it to users' typical hardware.

In our case, the response on an Nvidia 4090 RTX GPU averaged ∼ 147 seconds per tree generation with an output of ∼ 1518 tokens for depth 4-5 and ∼ 891 for depth 2-3 (the upper limit was set to 8k tokens). Since the dataset for fine-tuning is small, the same GPU was used for fine-tuning over 20 epochs, which took about 82 GPU hours. The literature states that the performance of the LLM training procedure does not increase as expected when the same data set and parameters [22], [14] are iterated multiple times, which was also observed in our case. Note that these parameters were determined empirically after short training evaluations of different parameter ranges.

Table 2: Head-to-head comparison of response preferences for the technical and non-technical groups by the LLMs core model used in the BTreeGenAIAgent.

Group	Avg preference		
Model	GPT-4o	Llama3.1	BTreeGenLLM
GT	51%	8%	41%
GN	63%	2%	35%

When we tried to understand the difference between the two groups, we found that the non-technical group tended to favor the GPT-4o answers because they were better and more fully explained. Although the BTreeGenLLM was small, it also had the problem that some skills were forgotten after fine-tuning. This can be remedied in the future by creating human-annotated explanations for the items in the dataset.

```
'### Explanation:\n'
'- **Selector node**: The root node is a selector - random composite node, which
       randomly selects one of the three branches to execute.\n'
'- **Branch 1**: \n'
'  - **FindObject**: Finds a door in the current room.\n'
'  - **IsDoorOpened**: Checks if the door is open.\n'
'  - **MoveTo**: If the door is open, the agent moves to it.\n'
'- **Branch 2**: \n'
'  - **Repeat Until Failure**: Repeats moving to the next waypoint in the '
'current room until there are no more waypoints.\n'
'  - **PlayAnimation**: Plays an idle animation with looping enabled for 5 '
'seconds.\n'
'  - **Wait**: Waits for 5 seconds.\n'
'- **Branch 3**: \n'
'  - **FindObject**: Finds a chair in the current room.\n'
'  - **MoveTo**: Moves to the chair.\n'
'  - **Wait**: Waits for 120 seconds (2 minutes).\n'
'  - **CheckPhone**: A custom action node where the agent checks their phone. ''
    'This action is described but not implemented in the available nodes.'
```

Listing 7: The explanation given by the agent for the tree generated in Figure 1 with the user input query shown in Listing 1. For example, notice how the missing implementations are suggested.

Next, we describe some conclusions from the human evaluation, the lessons learned, and the improvements based on the feedback. *Software engineering along the application of LLMs.* The effects of post-processing were also evaluated. For example, with reference to Table 1, the correctness results without this component were almost halved for the first row set, with the second being less than 10% for all categories. This proves that development work needs to be done not only on the small models but also on larger ones. When trying to find a compromise between size and performance, the amount of work increases significantly. For example, our 8B model struggled to even adhere to the JSON patterns or the format recommended for SRL operations without post-processing.
Human-in-the-loop and agent-based AI. Experiments and successive iterations revealed that it is important to control the agent processing the pipeline and to have fine-grained mechanisms for correction. For example, the use of SRL with specific roles and some hand-written rules (which could be dynamically injected on the user side) were required to increase the reliability of the results. Self-reflection on the quality of the response and then asking for human help significantly improved the quantitative assessment feedback. The evaluators from the technical group also preferred to confirm what was happening at different steps of the generation process. For example, they were interested in confirming or editing the selection of nodes in the filtering phase. This can be explained by the fact that they know the available implementation functions or have this type of knowledge. In contrast, other project stakeholders prefer to avoid this part and focus on editing/tuning the final visual output.
Explanations of the results and visual support through tools. People's feedback was very sensitive to the explanations given by the model in relation to the BT generated. It was important to not only have a model that outputs a behavior tree but also an explanation for the output so that even if it is not complete/correct, it can be edited afterward. A visual output tool was also highly appreciated by both groups. A better explanation of the results is planned for future work Reporting the missing functionalities and providing an empty node description were also welcomed by both sides. The people from the technical group could then replace these with similar ones and change the tree slightly if necessary. The other group felt that the suggestion should be taken up by applying the test-driven development (TDD) [3] methodology, creating an empty implementation by default and having the missing functionality implemented by the software engineers.

6 Conclusion

In summary, the integration of LLMs and agent-based AI into game development can improve the creation of complex AI behaviors and make the process more accessible and efficient for both technical and non-technical stakeholders. The proposed methods were able to retrieve nodes from the source code with minimal description so that they could be added to the knowledge of LLM through both fine-tuning and RAG. From a computational perspective, we also conclude that the small, fine-tuned model chosen as a reference can be used even on typical developer hardware with an accuracy that is not far behind external and expensive models.

Acknowledgements: We thank our colleagues and collaborators at Electronic Arts, Gameloft, and Amber, for the evaluation and constant feedback. This research was supported by the project "Romanian Hub for Artificial Intelligence - HRIA", Smart Growth, Digitization and Financial Instruments Program, 2021-2027, MySMIS no. 334906.

References

[1] Jicong Ao, Fan Wu, Yansong Wu, Abdalla Swikir, and Sami Haddadin. 2024. LLM as BT-Planner: Leveraging LLMs for Behavior Tree Generation in Robot Task Planning. arXiv:2409.10444 [cs.RO] https://arxiv.org/abs/2409.10444

[2] Akari Asai, Zeqiu Wu, Yizhong Wang, Avirup Sil, and Hannaneh Hajishirzi. 2023. Self-RAG: Learning to Retrieve, Generate, and Critique through Self-Reflection. arXiv:2310.11511 [cs.CL] https://arxiv.org/abs/2310.11511

[3] Kent Beck. 2022. *Test driven development: By example.* Addison-Wesley Professional.

[4] Matthijs Douze et al. 2024. The Faiss library. aaaa. *arXiv preprint arXiv:2401.08281, https://github.com/facebookresearch/faiss* (2024). arXiv:2401.08281 [cs.LG] https://github.com/facebookresearch/faiss

[5] Aaron Grattafiori et al. 2024. The Llama 3 Herd of Models. arXiv:2407.21783 [cs.AI] https://arxiv.org/abs/2407.21783

[6] Chengpeng Hu, Yunlong Zhao, Ziqi Wang, Haocheng Du, and Jialin Liu. 2024. Games for Artificial Intelligence Research: A Review and Perspectives. arXiv:2304.13269 [cs.AI] https://arxiv.org/abs/2304.13269

[7] Edward J Hu et al. 2022. LoRA: Low-Rank Adaptation of Large Language Models. In *International Conference on Learning Representations.* https://openreview.net/forum?id=nZeVKeeFYf9

[8] Jianheng Huang et al. 2024. Mitigating Catastrophic Forgetting in Large Language Models with Self-Synthesized Rehearsal. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers).* Association for Computational Linguistics, Bangkok, Thailand, 1416–1428. doi:10.18653/v1/2024.acl-long.77

[9] Like Hui and Mikhail Belkin. 2020. Evaluation of neural architectures trained with square loss vs cross-entropy in classification tasks. *arXiv preprint arXiv:2006.07322* (2020).

[10] Soyeong Jeong, Jinheon Baek, Sukmin Cho, Sung Ju Hwang, and Jong C. Park. 2024. Adaptive-RAG: Learning to Adapt Retrieval-Augmented Large Language Models through Question Complexity. arXiv:2403.14403 [cs.CL] https://arxiv.org/abs/2403.14403

[11] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2019. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data* 7, 3 (2019), 535–547.

[12] Sayash Kapoor, Benedikt Stroebl, Zachary S. Siegel, Nitya Nadgir, and Arvind Narayanan. 2024. AI Agents That Matter. arXiv:2407.01502 [cs.LG] https://arxiv.org/abs/2407.01502

[13] Fu Li, Xueying Wang, Bin Li, Yunlong Wu, Yanzhen Wang, and Xiaodong Yi. 2024. A Study on Training and Developing Large Language Models for Behavior Tree Generation. arXiv:2401.08089 [cs.CL] https://arxiv.org/abs/2401.08089

[14] Yiheng Liu et al. 2024. Understanding LLMs: A Comprehensive Overview from Training to Inference. arXiv:2401.02038 [cs.CL]

[15] Clara Meister and Ryan Cotterell. 2021. Language Model Evaluation Beyond Perplexity. In *ACL-IJCNLP.* 5328–5339.

[16] Ruslan Mitkov. 2022. *The Oxford Handbook of Computational Linguistics.* Oxford University Press. doi:10.1093/oxfordhb/9780199573691.001.0001

[17] OpenAI et al. 2024. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL] https://arxiv.org/abs/2303.08774

[18] Ciprian Paduraru and Miruna Paduraru. 2019. Automatic Difficulty Management and Testing in Games using a Framework Based on Behavior Trees and Genetic Algorithms. In *2019 24th International Conference on Engineering of Complex Computer Systems (ICECCS).* 170–179. doi:10.1109/ICECCS.2019.00026

[19] Aldo Pareja, Nikhil Shivakumar Nayak, Hao Wang, Krishnateja Killamsetty, Shivchander Sudalairaj, Wenlong Zhao, Seungwook Han, Abhishek Bhandwaldar, Guangxuan Xu, Kai Xu, Ligong Han, Luke Inglis, and Akash Srivastava. 2024. Unveiling the Secret Recipe: A Guide For Supervised Fine-Tuning Small LLMs. arXiv:2412.13337 [cs.LG] https://arxiv.org/abs/2412.13337

[20] Venkatesh Balavadhani Parthasarathy, Ahtsham Zafar, Aafaq Khan, and Arsalan Shahid. 2024. The Ultimate Guide to Fine-Tuning LLMs from Basics to Breakthroughs: An Exhaustive Review of Technologies, Research, Best Practices, Applied Research Challenges and Opportunities. arXiv:2408.13296 [cs.LG] https://arxiv.org/abs/2408.13296

[21] Nils Reimers and Iryna Gurevych. 2019. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP).* Association for Computational Linguistics, 3982–3992. doi:10.18653/v1/D19-1410

[22] Noveen Sachdeva et al. 2024. How to Train Data-Efficient LLMs. arXiv:2402.09668 [cs.LG]

[23] Tim Storer and Ruxandra Bob. 2019. Behave Nicely! Automatic Generation of Code for Behaviour Driven Development Test Suites. In *2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM).* 228–237. doi:10.1109/SCAM.2019.00033

[24] Jonathan Styrud, Matteo Iovino, Mikael Norrlöf, Mårten Björkman, and Christian Smith. 2024. Automatic Behavior Tree Expansion with LLMs for Robotic Manipulation. arXiv:2409.13356 [cs.RO] https://arxiv.org/abs/2409.13356

[25] Chenguang Wang, Xiao Liu, Zui Chen, Haoyun Hong, Jie Tang, and Dawn Song. 2022. DeepStruct: Pretraining of Language Models for Structure Prediction. In *Findings of the Association for Computational Linguistics: ACL 2022.*

[26] Lei Wang and Yan Zhang. 2023. Expanding Horizons: Leveraging Game Engine Toolsets for Educational and Simulation Applications. In *Proceedings of the 2023 International Conference on Game Development Advances.* Game Development Research Association, 102–113. https://gdraconf.org/2023/papers/wang_zhang.pdf

[27] Shi-Qi Yan, Jia-Chen Gu, Yun Zhu, and Zhen-Hua Ling. 2024. Corrective Retrieval Augmented Generation. arXiv:2401.15884 [cs.CL] https://arxiv.org/abs/2401.15884

[28] Georgios N. Yannakakis and Julian Togelius. 2024. *Artificial Intelligence and Games; Second edition public draft.* Springer. https://gameaibook.org/about/.