Context-Aware Hint Generation for Serious Games Using Large Language Models in Gidget

Ryan Ahlborn

Supervisors:

Michael Miljanovic

Cristiano Politowski

April 24, 2026

Abstract

Serious games about programming education help beginner learners develop programming skills, but players may still become stuck due to syntax errors, misunderstanding game mechanics, or uncertainty about how to proceed. Traditional hint systems often rely on manually written rules and predefined solutions, which can be difficult to scale.

This work investigates the use of a large language model to generate context-aware hints within the educational programming game Gidget. A hint generation pipeline was implemented that extracts game state information and the player's current attempt at the level from the game environment, converts this information into structured prompts, and returns generated hints directly within the game interface.

The system was evaluated through formative user testing involving 88 recorded hint requests. Results showed that 80.7% of generated hints were correct and immediately helpful, while 93.2% were factually correct overall.

These findings provide preliminary evidence that large language models can serve as a viable foundation for context-aware hint generation in structured environments of serious games when given enough explicit context of the current game state and mechanics of the game.

Contents

1 Introduction

Educational serious games have become an increasingly popular tool for introducing beginner programming learners to programming concepts through interactive problem-solving environments. By embedding programming tasks within game-like systems, these environments aim to improve engagement while providing learners with opportunities to develop programming skills.

One such environment is Gidget, a grid-based educational serious game where players solve puzzles by writing commands in a custom in-game language [1]. While Gidget is designed to sequentially introduce the learning process through progressively introduced mechanics, beginner players may still encounter difficulty when they become stuck due to misunderstanding game mechanics, struggling with syntax, or failing to identify the next logical step toward a solution. Prior work on Gidget has shown that framing programming tasks around purposeful, meaningful goals can significantly improve learner engagement in educational serious games [2].

Traditional hint systems for educational software often rely on manually written rules, predefined solution paths, or other heavily engineered domain-specific techniques, which can require significant developer effort and may be difficult to scale across games with a larger set of rules and mechanics [3]. Large language models (LLMs) present a potential alternative by enabling generation of context-aware hints without requiring handcrafted hints for every possible scenario.

This project investigates whether an LLM can be integrated into Gidget to generate useful context-aware hints based on the current game state and the player's current attempt at the level. To explore this, a hint generation pipeline was developed that extracts structured game-state information and player code from the Gidget environment, converts this information into a prompt suitable for LLM inference, and returns generated hints directly within the game interface.

The primary research question guiding this work is:

Can an LLM-based hint helper generate context-aware hints for players in a serious game?

To evaluate this question, the implemented system was tested through formative user play sessions in which generated hints were classified according to correctness and practical usefulness. The results of this evaluation provide insight into both the feasibility of LLM-based hint generation in structured environments of serious games.

2 Background: The Gidget Environment

2.1 Overview of Gidget

Gidget is an educational game designed to introduce novice users to fundamental programming concepts, primarily debugging, through an interactive, grid-based environment [1]. Players control a player character, Gidget, by writing commands in a custom in-game language. These commands are executed within the game world with the goal of meeting the win condition of the current level.

Gidget emphasizes learning through problem-solving through debugging [1]. The player writes a sequence of commands that they can choose to execute step-by-step, line-by-line, or all at once, allowing them to observe the effects of their code in real time and get a visual example of what each part of their code is doing to the environment. This style of execution visualization is commonly used in beginner programming environments to help learners understand how their code affects state during execution [4].

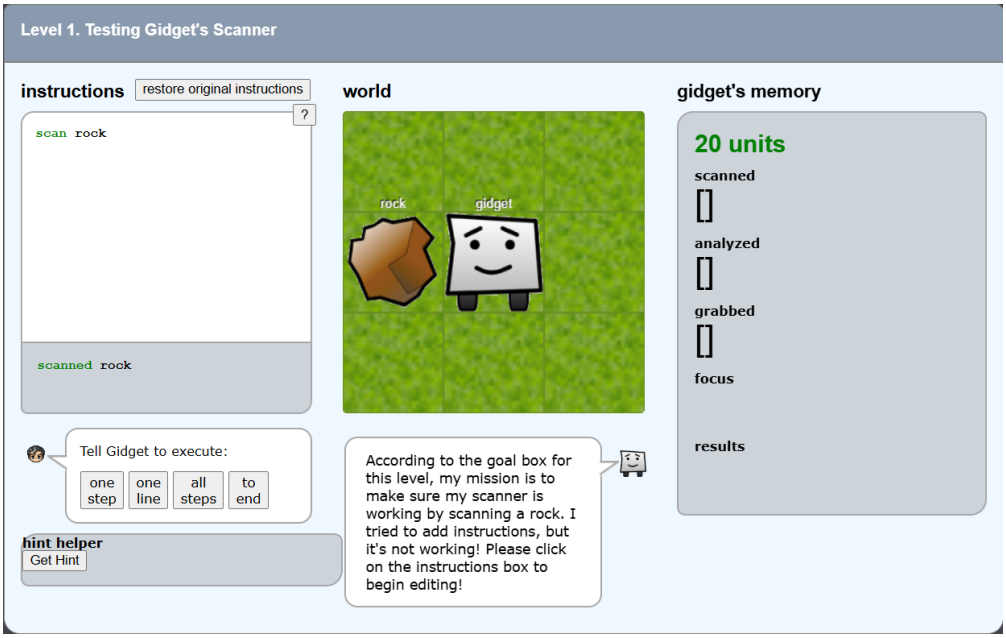

Figure 1: Example of the UI elements in Gidget

The left panel is where the player inputs commands, can see the win condition, and ask for hints. In the original version of the game, there is no section for hints as this section was added for this study. The central panel displays the playing field of the game world. The right panel shows Gidget's energy and the state of every object in terms of how Gidget is currently able to interact with them.

2.2 Game Mechanics

The game operates on a grid-based system in which objects occupy one of the grid squares. Gidget can move between adjacent tiles, interact with objects on the same tile, and perform actions depending on the current game state.

An important mechanic in Gidget is that the player character does not recognize the state of the objects within the current environment by default. This means interactions with objects require the player to gather information about them using various commands. This is typically done through the `scan` command, which allows Gidget to recognize and reference objects within the environment. Without scanning, many commands will fail or

have no effect.

Another important aspect is that commands are executed sequentially, meaning that earlier commands may influence the outcome of later ones. This mirrors the execution model of most modern programming languages and aligns with how code is typically expected to behave, allowing skills developed in Gidget to transfer naturally to more traditional programming environments.

However, unlike conventional programming languages, Gidget does not halt execution when an error occurs. Instead, when a command fails to apply to the current context, the current object is removed from focus and execution continues with the next command. This allows attempts to continue even in the presence of errors.

2.3 Command Language

Gidget uses a simplified custom in-game programming language that allows players to control the world through a sequence of commands [4, 5]. Understanding the behaviour and constraints of these commands is needed for both solving levels, as well as for generating meaningful hints.

2.3.1 `scan <object>`

The `scan` command is used to identify objects within the game environment. An object must always be scanned before it can be meaningfully referenced or interacted with as the player character does not recognize the state of objects in the world by default.

This requirement introduces a dependency between commands, where scanning often acts as a prerequisite for future commands such as movement or interaction. Failure to scan an object before attempting to interact with it in another command can result in error in the execution of the player code.

2.3.2 `goto <object A> avoid <object B>`

The `goto` command moves Gidget to the location of a specified object. Movement consumes one energy per tile travelled plus one energy per object being held by Gidget. The game determines the shortest path to use for movement and attempts to do so.

The `goto` command is often used in combination with other commands, as many interactions require Gidget to be on the same tile as the target object.

This command refers to all objects of the same name and as such Gidget will attempt to move to all objects of that name.

the player has the option to add `avoid <object B>` to the command to make Gidget's pathfinding not enter the same tile as any object with that name.

2.3.3 `grab <object>`

The `grab` command allows Gidget to pick up an object located on the same tile. Once an object is grabbed, it is carried by Gidget and moves with the character.

Carrying objects affects other game mechanics, most commonly energy use during movement.

2.3.4 `drop <object>`

The `drop` command releases a currently held object onto the tile occupied by Gidget. This is commonly used to place objects in specific locations required to satisfy level objectives.

Proper use of `drop` is often necessary to complete tasks that involve positioning objects relative to other objects.

Notably, an object need not be dropped for it to be considered on a tile for the purpose of fulfilling a win condition, however, this is often necessary when needing different objects on different tiles.

2.3.5 `analyze <object>`

The `analyze` command reveals additional information about an object, including traits the object may have and potential actions that can be performed using it.

Objects can be referred to by trait without analyzing them, but actions that the object can perform require the object to be analyzed before being unlocked for use.

2.3.6 `ask <object> to <action> <target A> <target B>`

The `ask` command allows Gidget to request that an object perform a specific action. The available actions depend on the properties of the object and are typically revealed and unlocked by the `analyze` command.

For example, certain objects may be able to move other entities or modify the game state in ways that are not achievable through player commands alone.

Some actions that can be requested with the `ask` command require two targets and will cause an error if only one target is given.

2.3.7 `if <object> is(n't) <condition>, <command>`

The `if` command enables conditional execution of other commands based on a specified condition. This allows for more complex logic, where actions are performed only when certain criteria are met.

The use of `is` executes the command if the condition is True. The use of `isn't` executes the command if the condition is False.

2.3.8 Command Chaining and Object Referencing

Commands in Gidget can be combined using commas, allowing multiple actions to be executed on a single line. Within these compound commands, the keyword `it` can be used to refer to the most recently referenced object in place of the object name.

This mechanic allows for the player to write more concise code and the ability to use multiple commands referring to each object in an iteration command such as `it`.

For example, in a level with three goop objects, the player would not be able to get Gidget to pick up all goops without the use of command chaining.

Example use of command chaining:

```
scan goop, goto it, grab it
```

2.4 Game Objects

The Gidget environment introduces objects gradually across levels. For the scope of this project, the implemented hint helper focused on the objects and mechanics appearing up to and including level 11, as these were the levels used during development and testing. This subset of levels includes the core objects required to evaluate whether the hint helper could reason about movement, object interaction, hazards, and win conditions.

2.4.1 Gidget

Figure 2: The player character: Gidget

Gidget is the player character controlled through commands. The player does not directly control Gidget through mouse or keyboard inputs. When attempting to interact with any object currently in the world, the object's state is checked based on info already acquired and thus known by Gidget.

2.4.2 Rock

Figure 3: A rock object.

Rocks are basic objects with no traits or actions. They only appear in the early levels that serve as tutorials and although they do not require use of the `grab` or `drop` command on them to complete any win condition.

2.4.3 Goop

Figure 4: A goop object

The goop is one of the primary objects used in level objectives. Many levels require the player to move goop to another object, such as a bucket. Some goops have traits on them that do not change how they are treated in the world outside of conditions in player code. These traits instead are used to check if the correct goop matches what is required by the win condition.

2.4.4 Bucket

Figure 5: A bucket object

The bucket is commonly used as a destination object. Many early objectives involve matching object location with another object, such as requiring a goop on the same tile as the bucket.

2.4.5 Kitten

Figure 6: A kitten object

The kitten is an object used in level objectives. Levels that use the kitten often require the kitten to be on the same tile as some other object like a crate. Kitten objects do not have any special traits or commands associated with them and as such do not need use of the `analyze` command.

2.4.6 Crate

Figure 7: A crate object.

The crate is often used as a destination object. These are introduced as another object that serves as a location to bring another object to. The player must differentiate which object must match the location of different location objects.

2.4.7 Battery

Figure 8: A battery object.

The battery introduces the idea that some objects have special actions. To unlock the ability to use the actions of objects with special actions, the `analyze` command must be used on the object first. After analyzing the battery, the player can use the `ask` command to ask the battery to energize Gidget, restoring energy.

2.4.8 Shrub

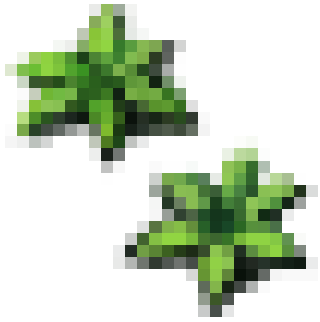

Figure 9: A shrub object.

Shrubs are objective objects. They often have traits that can be seen with the `analyze` command, and they have no special actions. These are in levels that require use of conditional commands to differentiate between objects based on their traits.

2.4.9 Tree

Figure 10: A tree object.

Trees are impassable objects that block movement for both Gidget and other characters. Trees do not need to be scanned because the pathfinding system naturally routes around impassable obstacles.

2.4.10 Crack

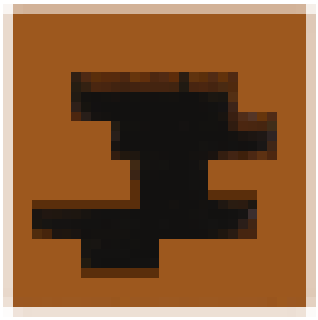

Figure 11: A crack object.

Cracks are hazards that Gidget can walk over. If Gidget moves over a crack while carrying an object, the held object is dropped. Cracks can be referenced to using the optional `avoid` command as part of the `goto command` and the levels that utilize cracks require the player to do so.

2.4.11 Bird

Figure 12: A bird object.

The bird appears in a level where it functions as part of a positioning-based win condition. Similar to objects such as a goop or a kitten, the bird needs to be moved or placed relative

to another object. It has no special traits or actions that it can do.

2.4.12 Rat

Figure 13: A rat object.

The rat is a moving hazard. It moves one space toward Gidget every other step of code execution using the same pathfinding as the `goto` command. If the rat reaches Gidget, Gidget's energy is set to 0 which automatically fails the level. This introduces a timing and path-planning constraint that differs from static objects.

2.4.13 Dog

Figure 14: A dog object.

The dog is an interactive object with a special `carry` action. The carry action uses the following syntax: `ask dog to carry <target A> <target B>`. The dog must be on the same tile as `<target A>`. `<target A>` and the dog will then move to `<target B>` using the

same pathfinding as the `goto` command. When moving in this way, Gidget does not spend energy per tile. The intended solution in levels with the dog often involves use of the dog's carry action to move gidget without spending energy.

2.4.14 Button

Figure 15: A button object.

The button is an interactive object that can modify the environment, such as lowering a gate. Like the battery and the dog, the button requires use of the `analyze` command and the `ask` command to get use out of the button's special action. The special commands associated with the button do not require targets when using the `ask` command.

2.5 Energy System

Gidget operates under an energy constraint that limits the number of actions that can be performed. Gidget starts with some initial value of energy that varies per level. Different actions cost different amounts of energy. By default, most commands cost one energy. If Gidget's energy is less than or equal to zero at any point in code execution, the attempt is stopped and the result is a failure.

Using the `goto` command, movement consumes one unit of energy per tile traversed, plus an additional unit for each object being carried by Gidget. This constraint is often used in levels where there are ways to mitigate energy usage through the use of other objects.

Notably, if Gidget and a rat are on the same tile, Gidget's energy is set to zero which results as a loss for the current attempt.

2.6 Relevance to Hint Generation

The mechanics described above directly impact the design of the hint generation system. In order to provide meaningful guidance, the system must account for:

- Dependencies between commands (e.g., the need to scan before interaction)
- Object-specific behaviors and actions
- Environmental constraints such as hazards and obstacles

These factors create a structured problem space in which the player must operate. The hint generation system must therefore interpret both the current game state and the player's code in order to provide useful and context-aware guidance.

3 Methodology

The core objective of this project was to design and implement a system capable of generating useful contextual hints for players of Gidget using an LLM. To accomplish this, a pipeline was developed to convert the structured state of the game and the player's current attempt at the level into a prompt suitable for LLM inference.

3.1 System Overview

The implemented system follows a multi-stage pipeline:

1. Extract the current game state from the Gidget environment
2. Extract the player's current code from the interface

3. Parse the player's current code for syntax errors

4. Format both into a structured prompt

5. Send the prompt to an LLM

6. Display the generated hint to the player

This pipeline allows the hint system to generate feedback based on the context of the player's current progress and the exact state of the level.

3.2 Game State Extraction

To provide meaningful hints, the system must first understand the current state of the game. Relevant information extracted includes:

- Current level name
- Remaining player energy
- Win condition
- Present object types
- Positions of relevant objects

This information is converted into structured text before being included in the prompt. Providing explicit state information reduces ambiguity and improves the model's ability to reason about what the player must do to improve their attempt.

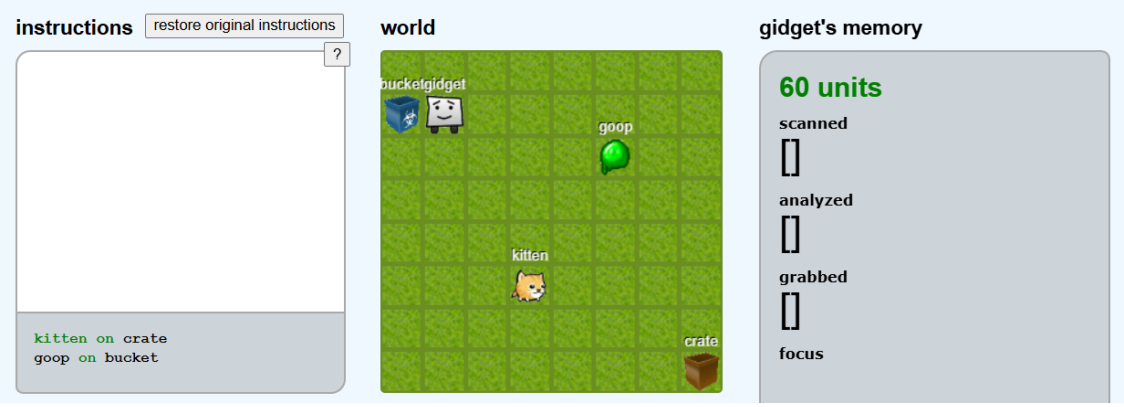

Figure 16: Game state as seen by the player

```
GAME STATE: Level: learnGrabAndDrop
            Energy: 60
            Win Condition: kitten on crate, goop on bucket
            Objects: - gidget @ (1, 1)
- goop @ (2, 5)
- crate @ (7, 7)
- bucket @ (1, 0)
- kitten @ (5, 3)
```

Figure 17: Extracted structured game state

3.3 Player Code Extraction

The player's current code is extracted directly from the code input panel within the Gidget interface. This allows the model to reason not only about the level itself, but also about the player attempt's current progress toward a solution.

Including the player's code allows for producing incremental hints rather than generic level-solving advice, as the system must identify what the player has already accomplished before determining an appropriate suggestion.

In addition to extracting the player's current code, the system uses Gidget's built-in parser to identify syntax and grammar errors within the player's attempt. Gidget already includes parsing functionality to support its step-by-step execution interface, where parsing

failures are used to indicate why a given instruction cannot be executed [5]. Rather than implementing a separate parsing system, this existing parser was reused to detect errors in the player's code and provide that information to the hint generation system. This allows the model to distinguish between strategic mistakes and issues caused by syntax errors, enabling it to prioritize grammar-related feedback when the player's code is invalid.

3.4 Prompt Construction

The extracted game state and player code are combined with manual descriptions of relevant game mechanics to construct the final prompt.

The prompt contains:

- A system instruction defining the model's role as a hint assistant
- Structured game rules and mechanics
- The current game state
- The player's current code
- Output formatting constraints

Prompt engineering plays a significant role in the performance of LLMs on structured reasoning tasks, with prompt wording, context, and examples often substantially affecting output quality [6, 7]. This structured prompt design was chosen to encourage the model to reason over the problem in an organized manner, as prior work has shown that structured prompting can improve large language model performance on programming-related reasoning tasks [8].

4 Implementation

4.1 Frontend Integration

The hint system was integrated directly into the Gidget interface through changes to the game's HTML and JavaScript files. A hint button was added to the user interface, allowing the player to request assistance at any point during gameplay.

When pressed, this button triggers the prompt construction pipeline and sends the generated prompt to the model API.

4.2 Model Selection

Several locally hosted language models available through Ollama were initially explored during development in an attempt to create a fully local implementation of the hint generation system. However, despite iterative prompt refinement, none of the tested local models demonstrated sufficient consistency or reasoning ability to produce reliable hints for user tests. As a result, these models were not included in formal user testing.

Following this, the implementation transitioned to an API-hosted model provided through OpenAI. The model evaluated in user testing was GPT-4o, which demonstrated substantially improved reasoning ability and consistency compared to the locally hosted alternatives. All reported user testing and evaluation results in this work correspond exclusively to the GPT-4o implementation.

5 Results

5.1 Evaluation Overview

Given the prototype-oriented nature of this work, evaluation focused on formative assessment of hint quality and implementation feasibility rather than large-scale measurement of pedagogical outcomes. The purpose of this evaluation was to determine whether an LLM

could provide context-relevant hints within Gidget at a level sufficient to justify further development.

Testing was conducted through supervised user play sessions involving 12 participants. A total of 88 hint requests were recorded during these sessions.

To evaluate hint quality, each generated hint was manually classified into one of three categories:

- Correct advice and immediately helpful
- Correct advice but not immediately helpful
- Incorrect advice

A hint was classified as *correct advice and immediately helpful* if it made no incorrect claims about the game's mechanics or game state and the player made progress toward the win condition on their subsequent attempt. A hint was classified as *correct advice but not immediately helpful* if it made no incorrect claims, but the player failed to make meaningful progress after receiving the hint. Finally, a hint was classified as *incorrect advice* if it misrepresented the game's mechanics, provided guidance inconsistent with the current game state, or otherwise suggested an invalid course of action.

This classification scheme allowed hint quality to be evaluated based not only on factual correctness, but also on practical usefulness in helping the player progress.

5.2 Quantitative Hint Quality Results

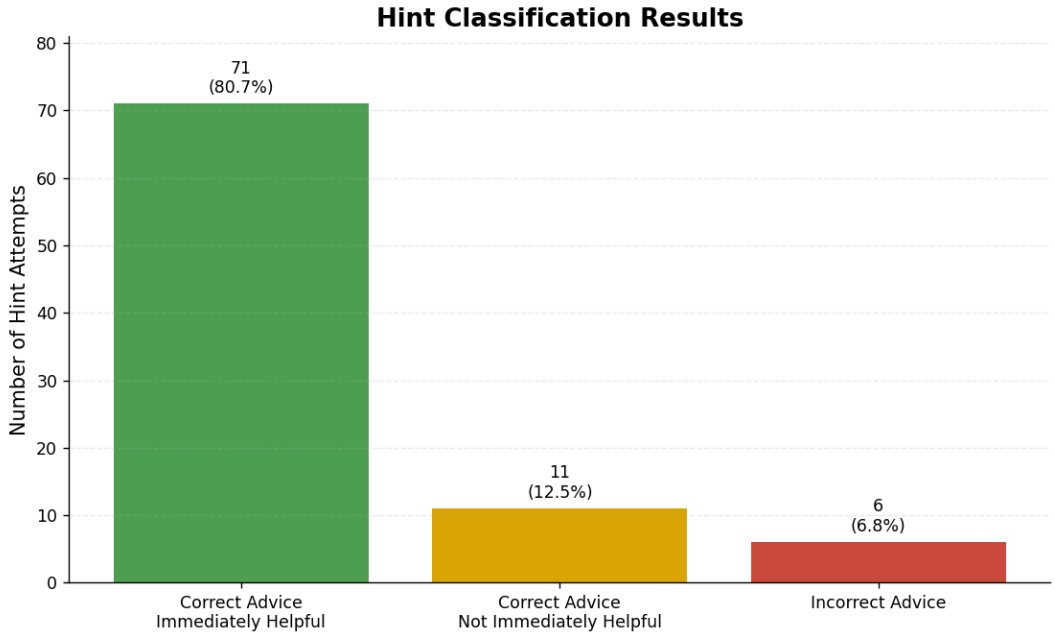

Figure 18: Distribution of generated hint classifications across 88 recorded hint requests during formative user testing.

As shown in Figure 18, the majority of generated hints were classified as *correct advice and immediately helpful*, making up 80.7% of all recorded hint requests. An additional 12.5% of hints were classified as *correct advice but not immediately helpful*. Only 6.8% of observed hints were classified as *incorrect advice*.

Overall, 93.2% of generated hints were factually correct, suggesting that the implemented system was generally able to reason accurately about the current game state and relevant mechanics when generating responses. The high proportion of immediately helpful hints further indicates that the system was frequently capable of turning that reasoning into advice that players could directly apply toward making significant progress in the current level.

These results provide preliminary evidence that LLMs can serve as a viable foundation for contextual hint generation within a structured environment for serious games such as

Gidget.

5.3 Qualitative Failure Case Analysis

Although overall quantitative performance was strong, analysis of incorrect and non-helpful hints revealed several recurring failure patterns.

The most common source of failure occurred when the player's attempted strategy diverged significantly from the intended solution path of the level. In these cases, the language model often struggled to provide effective guidance because the prompt structure and embedded game rules were primarily designed around the expected instructional objective of each level. When players attempted alternative strategies, the model had a tendency to get stuck exploring the player's strategy even when that strategy could not produce a solution.

This limitation was particularly noticeable in levels designed to teach a specific mechanic. Players who did not understand the newly introduced mechanic and would attempt to create a solution using only previously introduced mechanics were more vulnerable to getting this kind of incorrect advice.

A secondary source of failure involved cases where the model provided correct observations that did not meaningfully assist the player's immediate progression. These hints were generally the result of the model identifying a valid mechanic or object interaction, but focusing on information that was either already understood by the player or not directly relevant to the current obstacle.

These findings suggest that while the implemented system performs well when the player's reasoning aligns with the expected pedagogical path of a level, performance was worse when interpreting unconventional or unintended solution strategies. This reflects an inherent challenge in hint generation for open-ended problem solving, where multiple plausible solution paths may exist despite levels being designed around a specific instructional objective. This issue may also be noticed if a similar system was attempted to be implemented in a game where the nature of the game does not require an intended solution from the creator.

5.4 Discussion of Findings

The observed results suggest that LLMs are a viable approach for generating contextual hints in structured environments of serious games such as Gidget.

While this evaluation does not establish long-term pedagogical effectiveness, it demonstrates that a prompt-engineered LLM can provide useful in-context guidance with sufficient reliability to support prototype deployment and further research.

These findings support the feasibility of integrating LLM assistance into serious games and motivate future work involving larger-scale user studies and measurement of learning outcomes.

6 Limitations and Future Work

6.1 Limitations

Although the implemented system demonstrated promising results, several limitations remain.

First, evaluation was conducted through formative testing rather than a large-scale controlled user study. While sufficient for feasibility analysis, this limits the strength of conclusions that can be drawn regarding educational effectiveness.

Second, the system relies heavily on manually authored prompt rules describing game mechanics. Expanding the system to any other environment would require a full detailing of all mechanics and implementation of the game state from the new environment.

Finally, despite improvements through prompt engineering, the language model remains imperfect and can still produce incorrect hints.

6.2 Future Work

The most immediate direction for future work is large-scale user evaluation. While the formative testing conducted in this project was sufficient to assess prototype feasibility and identify common strengths and weaknesses of the system, a broader controlled study would allow for more rigorous measurement of pedagogical effectiveness. In particular, larger-scale testing could provide deeper insight into how different players interact with the hint system, which failure cases occur most frequently, and which aspects of hint generation require the most improvement.

Another important avenue for future work is evaluation of alternative models. Due to time and resource constraints, formal user testing in this project was conducted using only GPT-4o. While this model produced sufficiently strong results for the implemented prototype, testing additional models may display differences in reasoning ability or consistency. Comparing multiple models under identical evaluation conditions would provide a clearer understanding of how model selection impacts hint quality.

Together, these future directions would allow the system to be evaluated more rigorously while also informing further refinement of prompt design.

7 Conclusion

This work demonstrates that LLMs can be effectively integrated into structured environments in a serious game to provide context-aware hint generation based on game state information and the current player attempt at the level.

A complete hint generation pipeline was implemented within the Gidget environment, including extraction of a structured game state, player code parsing, prompt construction, and in-game hint generation through an LLM. Formative user testing showed that the system produced correct hints in the majority of observed cases, with most generated hints being immediately helpful to players.

Qualitative analysis further showed that the primary source of failure occurred when player strategies diverged significantly from the intended pedagogical solution path of a level, highlighting an important challenge in generating hints for open-ended problem-solving tasks within serious games.

Overall, the results indicate that LLMs are a viable foundation for context-aware hint generation in structured environments given enough explicit context of the current game state and mechanics of the game. While further large-scale evaluation is required to assess long-term pedagogical impact, this work provides preliminary evidence that LLM-based hint systems can serve as a practical alternative to more traditional manually written hint systems in serious games.

References

[1] M. J. Lee, "Gidget: An online debugging game for learning and engagement in computing education," in *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*. IEEE, 2014, pp. 193–194.

[2] M. J. Lee and A. J. Ko, "Investigating the role of purposeful goals on novices' engagement in a programming game," in *2012 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 2012, pp. 163–166.

[3] J. McBroom, I. Koprinska, and K. Yacef, "A survey of automated programming hint generation: The hints framework," *ACM computing surveys*, vol. 54, no. 8, pp. 1–27, 2022.

[4] C. Kelleher and R. Pausch, "Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers," *ACM computing surveys*, vol. 37, no. 2, pp. 83–137, 2005.

[5] M. J. Lee, F. Bahmani, I. Kwan, J. LaFerte, P. Charters, A. Horvath, F. Luor, J. Cao, C. Law, M. Beswetherick, S. Long, M. Burnett, and A. J. Ko, "Principles of a debugging-first puzzle game for computing education," in *Proceedings (IEEE Symposium on Visual Languages and Human-Centric Computing)*. IEEE, 2014, pp. 57–64.

[6] P. Korzyński, G. Mazurek, P. Krzypkowska, and A. Kurasiński, "Artificial intelligence prompt engineering as a new digital competence: Analysis of generative ai technologies such as chatgpt," *Entrepreneurial Business and Economics Review*, vol. 11, no. 3, pp. 25–38, 2023.

[7] Y.-T. Xiong, W.-J. Lian, Y.-N. Sun, W. Liu, J.-X. Guo, W. Tang, and C. Liu, "Exploring gpt-4o's multimodal reasoning capabilities with panoramic radiograph: the role of prompt engineering," *Clinical oral investigations*, vol. 29, no. 9, pp. 405–, 2025.

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