Digital Game Development Using Large Language Models (LLMs): An Exploratory Study

Cristiano Barroso Serra[1], **Gabriel Mattos Barroso Serra**[2], **Tadeu Moreira de Classe**[1]

[1]Research Group on Games to Complex Contexts (JOCCOM)
Graduate Program in Informatics (PPGI)
Federal University of the State of Rio de Janeiro(UNIRIO)
Rio de Janeiro - RJ - Brazil

[2]Veiga de Almeida University (UVA)
Rio de Janeiro - RJ - Brazil

cristianoserra@edu.unirio.br, gabrielmbserra@gmail.com
tadeu.classe@uniriotec.br

Abstract. Introduction: *Large Language Models (LLMs) are powerful tools for automating tasks like documentation, code generation, and prototyping in computer science, but their integration into game development pipelines is an opportunity by, also a challenge.* ***Objective:*** *This paper presents the development and implementation of PromptingGameCraft (PGC), a tool that uses Large Language Models (LLMs) to automate key steps in digital game development. The tool takes a Game Design Document (GDD) as input and automatically generates a Game Design File (GDF) in JSON format, along with a custom class diagram, directory and file structure, and game code.* ***Methodology:*** *The architecture was implemented through a web interface connected to the DeepSeek-reasoner model API hosted on Google Cloud. As a proof of concept, a 2D ball-catching game with progressive difficulty was developed.* ***Results:*** *The automated generation process demonstrated efficiency in the transition from design to code, promoting modular organization, logical clarity, and reusability. In addition to productivity and standardization, PGC has the potential to democratize access to game development in educational, training, and community contexts. By enabling the accessible transformation of ideas into working prototypes, it promotes creative expression, supports active learning, and enhances participation among diverse groups.*

Keywords. *Large Language Models, Game Design, Game Prototyping, AI-Assisted Development*

1. Introduction

The digital games industry has experienced rapid growth, surpassing $175 billion in revenue in 2021 and consolidating itself as a major segment of global entertainment [Newzoo 2021]. This expansion has been driven by factors such as mobile gaming, the evolution of game engines, and new monetization models like microtransactions and subscriptions [Tower 2021, Worldpay 2021]. In this increasingly competitive landscape, optimizing the development process has become essential, given the complexity involved across areas such as design, programming, and narrative [Redmond et al. 2024].

To address these challenges, structured design approaches such as the Entity-Component System (ECS) and Unified Modeling Language (UML), for instance, have been widely adopted [Muratet e Garbarini 2020]. More recently, Large Language Models (LLMs) have emerged as powerful tools for automating documentation, code generation, and

prototyping in different fields of computer science [Xu et al. 2022]. These models have shown potential in tasks ranging from character creation to full code synthesis [Xu et al. 2022], yet their integration into complete game development pipelines remains limited.

This paper investigates how LLMs can be systematically employed to automate early stages of digital single game creation. It introduces *PromptingGameCraft* (*PGC*), a tool that integrates semantic interpretation, architectural inference, and scaffold instantiation into a unified pipeline powered by LLMs. By automating the transformation of high-level design into structured implementation, *PGC* reduces cognitive and technical effort, promotes modularity and reuse, and supports prototyping in educational or experimental settings.

The proposal is part of a broader **project model** aimed at developing intelligent, semi-autonomous systems for digital game development and learning. In contrast to approaches focused on narrative generation or agent-based orchestration, this work contributes an original, implementation-oriented architecture grounded in LLM capabilities.

The remainder of the paper is organized as follows: Section 2 presents the fundamental concepts; Section 3 discusses related work; Section 4 details the proposed architecture; Section 5 reports the implementation results; and Section 6 presents the final considerations and future directions.

2. Fundamental Concepts

This section presents the essential concepts for understanding the proposed solution, including Artificial Intelligence and Large Language Models (LLMs), Prompt Engineering, and the Game Design Document (GDD).

2.1. Large Language Models

Large Language Models (LLMs) stand out for their ability to process and generate natural language using deep learning architectures such as transformers [Annepaka e Pakray 2025]. Trained on extensive text corpora, LLMs excel in tasks such as content generation, summarization, and programming assistance, enabled by their autoregressive learning mechanisms.

In the context of game development, LLMs expand these possibilities by enabling automated narrative generation, dialogue design, and even game logic definition [Kumaran et al. 2023].

2.1.1. Prompt Engineering

Prompt engineering plays a key role in harnessing the full potential of Large Language Models (LLMs), especially in complex or domain-specific tasks [Liu et al. 2021]. Well-crafted prompts guide the model's behavior, enhance accuracy, and can even activate advanced capabilities.

Among the core strategies, **Zero-Shot** prompting provides only a task description without examples, while **Few-Shot** offers a small number of input-output pairs to improve contextual understanding [Kojima et al. 2023]. **Chain-of-Thought** prompting encourages step-by-step reasoning, enhancing performance in logical tasks [Kojima et al. 2023], and **ReAct** (Reason+Act) combines reasoning with actions for interactive workflows [Yao et al. 2022].

For the execution of this study, several emerging prompt engineering techniques were examined [Sahoo et al. 2024]: **Scratchpad** prompting introduces intermediate annotations to

support complex reasoning [Nye et al. 2021], **Program of Thoughts (PoT)** expresses logic in code-like structures [Chen et al. 2022], and **Structured Chain-of-Thought (SCoT)** adds predefined reasoning stages [Li et al. 2023b]. Finally, **Chain of Code (CoC)** uses code fragments to represent structured reasoning paths, benefiting computational tasks [Li et al. 2023a].

2.2. Game Design Document

The *Game Design Document* (GDD) is a structured reference document that describes all core aspects of a game, including mechanics, narrative, scoring, interface, and visual elements [Fullerton 2018]. It serves as a central artifact to guide development teams (designers, programmers, artists, and audio engineers), ensuring consistency across the project lifecycle [Salazar et al. 2012].

Maintaining an up-to-date GDD improves **communication and transparency** among team members, supports **scope control** by clearly defining functionalities and milestones, and facilitates **change tracking** through revision histories. These aspects contribute to better project alignment, risk management, and collaborative efficiency [Salazar et al. 2012].

3. Related Work

This section reviews recent efforts that apply artificial intelligence to automate aspects of game development. While several approaches focus on narrative generation or multi-agent collaboration, this work distinguishes itself by targeting the implementation phase through structured code generation from a Game Design Document (GDD).

Chen et al. [Chen et al. 2023] propose a multi-agent framework where roles such as Development Manager, Engineer, and Tester are simulated to collaboratively produce game components. Lima et al. [Lima et al. 2023] present ChatGeppetto, a tool that adapts literary plots into interactive stories using a story-generation agent (ChatGPT) and an illustration agent (Stable Diffusion).

In contrast, *PGC* automates the derivation of semantic structures, class diagram, and executable code directly from the GDD. This shifts the focus from narrative or visual adaptation to the technical abstraction and realization of game logic, offering a reproducible pipeline for design-to-code automation.

4. PromptingGameCraft - Architecture

The proposed *PromptingGameCraft* (*PGC*) architecture conceptualizes a pipeline that automates the early stages of digital single game development by transforming high-level design specifications into a structured foundation for implementation. Unlike traditional workflows, where design concepts are manually translated into system components, this architecture abstracts the transformation process and orchestrates the interplay between design intent and software realization through the use of Large Language Models (LLMs). This approach enables the systematic derivation of modular and reusable components from a design-oriented artifact, promoting consistency, traceability, and rapid iteration. It is particularly suited for educational and experimental contexts, where quick prototyping and alignment between team members are essential.

Table 1 outlines the sequential phases of the *PGC* architecture.

This architecture abstracts the transformation logic while encapsulating domain knowledge within adaptive prompts and representations mediated by the LLM. All artifacts

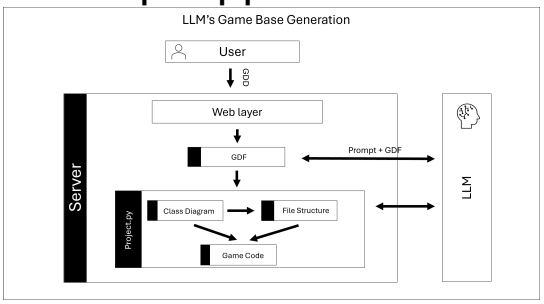
Table 1. The *PGC* Sequential Phases

Phase	Description	Main Objective	Output
Design Input	Reception of the Game Design Document (GDD) containing the conceptual specification of the game.	Collect structured game design information.	GDD artifact.
Semantic Interpretation	The Language Model interprets the GDD and generates a formal semantic model of the game.	Translate design intent into a structured and interpretable representation.	Game semantic model.
Architectural Inference	Inference of an abstract blueprint defining component roles, interactions, and structural patterns.	Organize the game logic into an architectural configuration.	Architectural blueprint.
Scaffold Instantiation	Instantiation of an implementation-oriented scaffold based on the inferred blueprint.	Create a reusable and modular project foundation.	System scaffold aligned with design logic.

produced throughout the process are logically structured and can be reused, modified, or extended in future iterations.

This pipeline is a core component of a broader **project model** focused on the development of intelligent, semi-autonomous systems for game development. It aims to reduce the cognitive and technical burden traditionally associated with early-stage design translation, fostering greater accessibility, innovation, and pedagogical support in the field of digital single games. Figure 1 illustrates the conceptual pipeline of the proposed architecture. The pipeline starts with users (i) applying the GDD to our *PGC* server; next, (ii) our architecture derives a GDF (game design file) converting the GDD information to *PGC* expected field patterns. Then, the (iii) GDF is processed by LLMs systems using generic coding design patterns that will be used to generate the specific game design pattern, source code and, file structures, which are integrated with LLM reasoning and human creativity.

Figure 1. Conceptual pipeline - *PGC* Architecture



(i) Game Design Document

The **Game Design Document** (GDD) consolidates gameplay elements—mechanics, narrative, interactions, and behaviors—into a unified, human-readable artifact that expresses the game's intent. Its role is to communicate design intent in a way that supports both human understanding and automated interpretation. Table 2 details the main sections of the GDD used in the *PGC* architecture.

(ii) Game Design File

The **Game Design File** (GDF) is a formalized representation derived from the GDD, in logical patterns. It organizes design semantics into categorized structures that support interpretation,

Table 2. Game Design Document (GDD) Sections

Section	Description
Overview	Main goal, game type, and duration
Platform and Target Audience	Intended platforms and player demographics
Game Mechanics	Movement, interactions, object logic, progression
Scoring and Penalties	Rules for gaining or losing points
Game Over Conditions	Criteria that trigger the end of the game
User Interface (HUD)	On-screen information and player interface
Technical and Physical Parameters	Speed, collisions, FPS, resolution
Visual Style and Art Requirements	Artistic direction and graphic needs

validation, and downstream generation processes. It serves as an intermediary between conceptual design and technical realization. Table 3 shows the semantic categories defined in the GDF. Additionally, Table 4 shows examples of structured identification fields from the GDF used to map the modular code in the project. Its primary objective is to ensure higher fidelity between the designer's vision and the outputted code. By pre-formalizing mechanics, the GDF acts as a semantic control layer, mitigating recurring failures. Language models carry an established risk of context inconsistency, which may lead to omission or oversimplification of core mechanics during generation [Gallotta et al. 2024b, Gallotta et al. 2024a]

Table 3. Game Design File Categories

Categories	Description
Game descriptions	Themes, objectives, narrative, enemy paths, difficulty levels, environment, and story progression
System interactions	Menus, pause, settings, onboarding, game over
Game interactions	Core gameplay such as movement, item collection, combat, puzzles, and abilities

Table 4. Identification Fields

Item	Description
`id`	Unique identifier (e.g., "GAME_01")
`name`	Interaction name (e.g., "Player Movement")
`type`	Category (e.g., "task", "event", "action")
`description`	Brief explanation of the in-game action
`parameters`	Input variables (e.g., "direction", "speed")
`next`	Next interaction(s) in the sequence

(iii) Coding Class Diagram

A custom notation [1] was defined for this project to enhance clarity and semantic precision when modeling software architecture. Unlike standard UML diagrams, this notation introduces additional symbols such as `->M` for method calls, `->C` for object instantiation, and `->O` for observer patterns. These conventions aim to simplify the visual representation of interactions between components, classes, and modules while remaining intuitive and readable. The full set of relationship symbols is summarized in Table 5.

4.1. Prototype Presentation - Demonstration

To demonstrate the applicability of the *PGC* architecture, a functional prototype was implemented and deployed in a cloud environment. It integrates a web-based interface with a Python backend, allowing interaction between the user, a Game Design Document (GDD), and a Large Language Model (LLM), automating the generation of a structured game project.

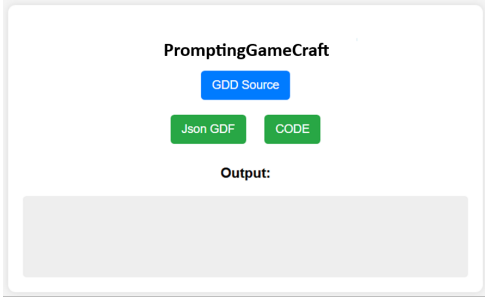
[1] `https://docs.google.com/document/d/1N4VCOgoQtXbkcIGqOPF9C85uQEV59bDzHpzmCXFvWZw/edit?usp=sharing`

Table 5. Custom Notation

Relationship	Symbol	Example	Description
Inheritance	`->`	`Enemy -> Character`	Class inherits from another
Implements	`>`	`Service > Interface`	Implements interface or protocol
Composition	`-*`	`House -* Wall`	Strong ownership: one object fully owns another
Aggregation	`-o`	`Team -o Player`	Weak ownership: object contains others without full control
Instantiates	`->C`	`Factory ->C Product`	Creates an instance of another class
Calls Methods	`->M`	`Scene ->M loadAssets`	Calls a method from another module or class
Observes	`->O`	`UI ->O Subject`	Implements the observer pattern
Depends On	`-->`	`Controller --> Repository`	High-level dependency
Provides	`=>`	`Service => Controller`	Provides resources or logic to another component
Association	`--`	`Client -- Order`	General association

Upon receiving the GDD, the system performs semantic interpretation to produce a formalized intermediate representation: the Game Design File (GDF). Based on this, a coding class diagram and an implementation scaffold are derived, which serve as a base for generating complete source code using the Phaser[2] framework. The prototype interface encapsulates these stages, streamlining the user experience and enabling developers to focus on design aspects. Figure 2 illustrates this interface.

Figure 2. *PGC* prototype interface.



The prototype provides initial indications of the feasibility of using the proposed architecture, operationalizing key principles such as semantic interpretation, architectural inference, and scaffold instantiation with executable code generation.

5. Proof of Concept

The proof of concept suggests *PromptingGameCraft* (*PGC*) may support the automation of the transition from game design to implementation. By transforming natural language specifications in a GDD into a structured game scaffold, the system demonstrates its ability to support early-stage development.

Using structured prompts, the system generated Game Design Files (GDFs) that encapsulate mechanics, interface interactions, system states, and progression logic. These were then used to infer class diagram and generate full JavaScript code in Phaser, representing all key components specified in the GDD. *PGC*'s pipeline relies on prompts designed to address each stage of the architecture.

To illustrate how *PGC* operationalizes these prompt stages, Listing 1 presents an example used to generate the GDF. Additionally, Listing 2 shows an excerpt of a generated GDF, highlighting how the model encodes system interactions in a formalized structure.

The prompt instructs a LLM to act as a game system architect, analyze a given Game Design Document (GDD), and generate a complete, structured Game Design File (GDF)

[2] `https://phaser.io/`

Table 6. Prompt functions used in the *PGC* pipeline

Prompt Name	Description
Prompt_GDF	Generates the Game Design File (GDF) from the GDD
Prompt_Pattern	Infers the modular class diagram based on the GDF
Prompt_Structure	Generates the folder and file structure aligned with the inferred architecture
Prompt_Code	Produces the final game code using the Phaser framework

in JSON format, detailing all game mechanics and interactions in a coherent flowchart-like structure.

Listing 1. Game Design File Prompt

```
prompt_0 = (
  "You are a game system architect. Based on the following Game Design Document (GDD): " + GDD + "\n"
  "1 - Create a complete Game Design File (GDF) in JSON format that covers all the game mechanic logic needed.\n"
  "2 - Divide the structure into two main sections:\n" ...
  "3 - For each interaction, define:\n"
  "  - 'id': unique identifier\n"
  "  - 'name': descriptive name\n" ...
  "4 - Ensure the logic is complete and coherent as a flowchart represented in JSON.\n"
  "5 - Return only the final JSON file representing the GDF.\n"
)
```

Listing 2. Excerpt of generated Game Design File

```
"system_interactions": [
  {
    "id": "S1",
    "name": "Main Menu Trigger",
    "type": "trigger",
    "description": "Opens/closes the main menu when ESC is pressed",
    "parameters": { "key": "Esc", "screen": "MainMenu" },
    "next": ["S2", "S3"]
  }
]
```

After the GDD is converted into a GDF, a class diagram is automatically generated, as illustrated in Listing 3.

Listing 3. Excerpt of Class Diagram

```
[{
  "game/static/src/scenes/MenuScene.js": [
    "-M initMainMenu : Creates Navigation options",
    "-M handleOptionSelect : Processes menu selections",
    "--> SystemInteractions : Depends on system interaction definitions"
  ],
  "game/static/src/scenes/GameScene.js": [
    "-> Basket : Instantiates player-controlled basket entity",
    "-M updateGameState : Handles core game loop",
    "-> DifficultySystem : Observes difficulty progression changes"
  ],
  "game/static/src/entities/Basket.js": [
    "-M handleMovement : Processes input controls",
    "-* PhysicsBody  : Composition with physics representation",
    "-> applyBoundaries : Keeps within screen limits"
  ]
}]
```

The proof of concept suggests that *PGC* can support the transition from design to implementation through a reproducible and structured process, which may be applicable to experimental and educational scenarios.

5.1. Game Implementation - Example

To assess the effectiveness of the *PGC* architecture, a simple 2D game prototype was developed using only the outputs generated by the system. This case study aims to verify whether the GDD, Game Design File (GDF), coding design pattern, and code scaffold can be effectively applied to create a fully functional game.

The implementation used the **Phaser** framework, an open-source HTML5 engine for 2D games. Its modular design, support for scenes, physics, and inputs makes it well-suited

for rapid prototyping, aligning with *PGC*'s focus on producing usable, extensible code for experimentation and education.

The GDD [3] provided as input was written in Markdown and described a fast-paced arcade game, summarized in Table 7.

Table 7. Excerpt from the Game Design Document – Basket Catch Game

Element	Description
Overview	2D game in which the player controls a horizontally movable basket to catch falling balls.
Duration	120 seconds base time, with possible increases from bonus elements and penalties from traps.
Objective	Maximize score by catching beneficial balls while avoiding penalties.
Ball Types	**Common:** +1 or -1 point, some reduce time by 3 seconds. **Golden:** +2 or -2 points. **Time (punitive):** 0 points, -5 seconds when caught, no penalty if missed.
Progression	Difficulty increases every 30 seconds.
Game Over (optional)	Triggered if the player misses 20 balls.

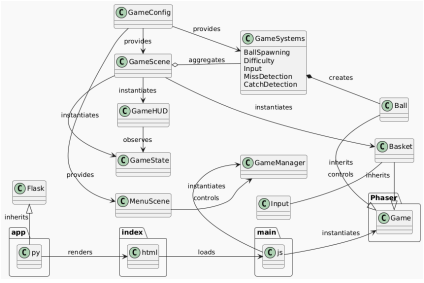
The game was built using the artifacts produced by *PGC*, including the complete folder structure, scaffolding, and JavaScript code written for Phaser. Only minimal manual adjustments were necessary, such as adding '.js' extensions to import paths due to ECMAScript module syntax as showed in the Table 8.

Table 8. Manual Adjustments to LLM-Generated Code

Original LLM Output	Expected Output	Manual Adjustment
`import { MenuScene } from './scenes/MenuScene'`	`import { MenuScene } from './scenes/MenuScene.js'`	Added `.js` extension
`import { GameScene } from './scenes/GameScene'`	`import { GameScene } from './scenes/GameScene.js'`	Added `.js` extension
`import { GameManager } from './core/GameManager'`	`import { GameManager } from './core/GameManager.js'`	Added `.js` extension
`import { GameConfig } from './config/GameConfig'`	`import { GameConfig } from './config/GameConfig.js'`	Added `.js` extension

The automatically inferred modular architecture is illustrated in Figure 3, showing the component relationships and system organization. The game's architecture follows a modular design pattern, structured into presentation, logic, and data layers. Scenes like `MenuScene` and `GameScene` manage flow and gameplay, while entities such as `Basket` and `Ball` encapsulate behavior and physics. Systems handle core mechanics like collision, spawning, and difficulty scaling, interacting through method calls and observer patterns. The `GameManager` coordinates transitions and persistence.

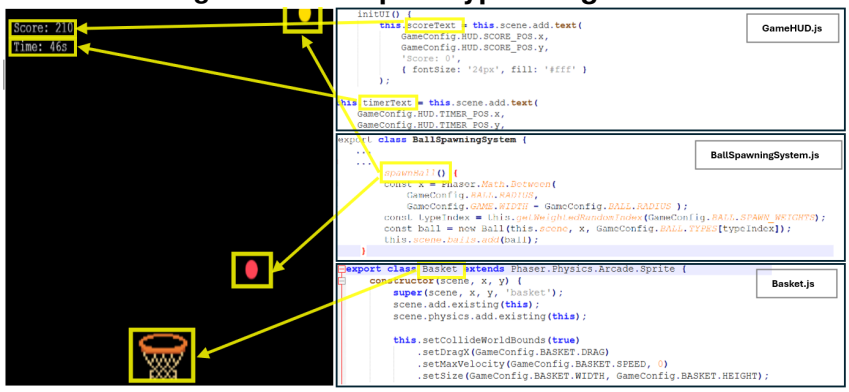
Figure 3. Automatically generated class diagram.



[3] https://docs.google.com/document/d/1g8DLDYpsBB60TfZY55efC4srmhxbia8H/edit?usp=drive_link&ouid=115202591714752543778&rtpof=true&sd=true

The resulting game prototype is shown in Figure 4[4], featuring the game layout and key elements such as the player-controlled basket, falling balls, and HUD displaying time and score. Figure 4 also contains excerpts from the generated code. The ball logic includes type variation and time-based behavior, while the basket module manages user control. Other components like the HUD system were also generated automatically and integrated without modification.

Figure 4. Game prototype using *PGC*.



In order to measure the impact of recent adjustments made to the source code, a comparative analysis of file metrics including the number of files, lines, words, and characters was conducted. The table 9 summarizes these metrics for both JavaScript and Python files, before (i.e., directly after LLM-based generation) and after the manual adjustments. It also presents the percentage variation observed in the JavaScript code, which was the only one modified.

Table 9. Code Metrics Comparison Before and After Adjustments

Code	Files	Lines	Words	Characters
Before Adjustments				
JavaScript	14	540	1231	15177
Python	1	38	83	1047
After Adjustments				
JavaScript	14	547	1252	15352
Python	1	38	83	1047
Variation (JavaScript)		1%	2%	1%

Table 10 presents a rapid and alternative evaluation of the proposed tool. The code samples were generated by the DeepSeek Reasoner and independently assessed by GPT-o3 to reduce potential bias from the generating language model. Code 1 was produced via a single-shot prompt, resulting in a solution appropriate for educational purposes and rapid prototyping, and received a score of 6.25. In contrast, Code 2 was generated using a structured prompt architecture, with an emphasis on modularity and scalability. It achieved a score of 9.5, standing out as the most technically robust solution. The independent review by GPT-o3 ensures a more objective and reliable assessment of the generated outputs.

This implementation offers preliminary evidence that the architectural pipeline proposed by *PGC* can be operationalized in practice. By enabling the transition from design to code with minimal developer input, the system demonstrates practical potential for modular game development. These results provide preliminary evidence of *PGC* applicability to rapid

[4]The basket rasterized assets was created manually.

Table 10. Code evaluation

Code	Score	Generation Method	Remarks
Code 1	6.25	Single-shot	Suitable for learning and rapid prototyping. Generated with a single prompt execution.
Code 2	9.5	Prompt Architecture	Offers better structure, scalability, and modularity. Created using a multi-stage prompt architecture.

prototyping and educational contexts, motivating a broader discussion of its capabilities and limitations.

6. Discussion and Final Considerations

The implementation of the *PGC* interface suggests the potential feasibility of employing Large Language Models (LLMs) to automate critical stages of digital game development. The proposed architecture integrates semantic interpretation of Game Design Documents (GDDs), structured generation of Game Design Files (GDFs), automatic inference of design patterns, and instantiation of implementation scaffolds, culminating in the production of executable code.

The results suggest that natural language specifications may be translated into reusable and logically coherent software structures. By leveraging the Deepseek-reasoner model and a prompt-driven pipeline, *PGC* aligns with established software development practices while significantly reducing the technical overhead typically required in early-stage prototyping.

The GDF emerged as a pivotal intermediary artifact, enabling the formalization of gameplay logic and supporting the generation of modular code within development frameworks such as Phaser. The case study suggests the system's capacity to bridge the gap between design and implementation with minimal human intervention, as evidenced by negligible variations in codebase metrics after manual adjustments.

In addition to its technical efficacy, *PGC* demonstrates potential for expanding access to game development within educational and community contexts. Its accessible workflow fosters creative experimentation, supports active learning methodologies, and encourages broader participation in the production of digital technologies.

Despite its contributions, the system presents some limitations. Its effectiveness depends heavily on the quality and structure of the input GDD; poorly defined documents can result in incomplete or inconsistent outputs. Although the generated code provides a functional baseline, manual intervention is still necessary for asset integration, adaptation to specific game engines, and adjustments to enhance usability and performance. **Future work** includes extending support for multiple GDD formats, integrating automated validation mechanisms, enhancing compatibility with additional game engines, and evaluating the pipeline with more advanced LLMs to assess potential improvements in semantic precision, reasoning capabilities, and contextual adaptability.

Acknowledgments

The authors thanks to Carlos Chagas Filho Foundation for Research Support of the State of Rio de Janeiro – FAPERJ (proc. E-26/204.478/2024 - SEI-260003/013219/2024) for partially funding this research.

References

Annepaka, Y. e Pakray, P. (2025). Large language models: A survey of their development, capabilities, and applications. *Knowledge and Information Systems*, 67:2967–3022.

Chen, D., Wang, H., Huo, Y., Li, Y., e Zhang, H. (2023). Gamegpt: Multi-agent collaborative framework for game development. *arXiv preprint arXiv:2310.08067*.

Chen, W. et al. (2022). Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *arXiv preprint arXiv:2211.12588*.

Fullerton, T. (2018). *Game Design Workshop: A Playcentric Approach to Creating Innovative Games*. AK Peters/CRC Press.

Gallotta, R., Liapis, A., e Yannakakis, G. (2024a). Consistent game content creation via function calling for large language models. In *2024 IEEE Conference on Games (CoG)*, pages 1–4. IEEE.

Gallotta, R., Todd, G., Zammit, M., Earle, S., Liapis, A., Togelius, J., e Yannakakis, G. N. (2024b). Large language models and games: A survey and roadmap. *IEEE Transactions on Games*.

Kojima, T., Schmid, P., Li, Q., Alsaidi, A., Tan, C., Lu, X., e Song, D. (2023). What makes chain-of-thought prompting effective? a counterfactual study. In *Proceedings of the 11th International Conference on Learning Representations (ICLR)*.

Kumaran, V., Rowe, J., Mott, B., e Lester, J. (2023). Scenecraft: Automating interactive narrative scene generation in digital games with large language models. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 19, pages 86–96.

Li, X. et al. (2023a). Chain of code: Reasoning with code for interpretability. *arXiv preprint arXiv:2303.08168*.

Li, X. et al. (2023b). Structured prompting: Scaling in-context learning to 1,000 examples. *arXiv preprint arXiv:2303.08774*.

Lima, E. S. d., Feijó, B., Casanova, M. A., e Furtado, A. L. (2023). Chatgeppetto - an ai-powered storyteller. In *Proceedings of the 22nd Brazilian Symposium on Games and Digital Entertainment (SBGames)*, Rio Grande, Brazil. ACM.

Liu, P., Yuan, W., Fu, J., Jiang, Z., Hayashi, H., e Neubig, G. (2021). Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *arXiv preprint arXiv:2107.13586*.

Muratet, M. e Garbarini, D. (2020). Accessibility and serious games: What about entity-component-system software architecture? In *Games and Learning Alliance (GALA)*. Springer.

Newzoo (2021). Global games market to generate $175.8 billion in 2021. Accessed: 2025-04-17.

Nye, M. et al. (2021). Show your work: Scratchpads for intermediate computation with language models. In *Advances in Neural Information Processing Systems*.

Redmond, P., Castello, J., Calderón Trilla, J. M., e Kuper, L. (2024). Exploring the theory and practice of concurrency in the entity-component-system pattern. *Proceedings of the ACM on Programming Languages*, 8(OOPSLA):1–29.

Sahoo, P., Singh, A. K., Saha, S., Jain, V., Mondal, S., e Chadha, A. (2024). A Systematic Survey of Prompt Engineering in Large Language Models: Techniques and Applications. *arXiv e-prints*, page arXiv:2402.07927.

Salazar, M. G., Mitre, H. A., Olalde, C. L., e Sánchez, J. L. G. (2012). Proposal of game design document from software engineering requirements perspective. In *2012 17th International Conference on Computer Games (CGAMES)*, pages 81–85.

Tower, S. (2021). State of mobile gaming 2021. Accessed: 2025-04-17.

Worldpay (2021). Microtransactions: Next big thing? Accessed: 2025-04-17.

Xu, F. F., Alon, U., Neubig, G., e Hellendoorn, V. J. (2022). A systematic evaluation of large language models of code. *arXiv preprint arXiv:2202.13169*.

Yao, S., Zhao, J., Yu, D., Yu, S., Gao, S., Zettlemoyer, L., e Narasimhan, K. (2022). React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*.