POLITECHNIC OF TURIN

MASTER's Degree in Computer Engineering

Politecnico di Torino

MASTER's Degree Thesis

Cheap Fun: Accelerating Game Pre-Production via LLM-Based Vibe Coding

Supervisors

Prof. MARCO MAZZAGLIA

Prof. FRANCESCO STRADA

Prof. EDOARDO BATTEGAZZORRE

Candidate

MARCO RIGGIO

A.Y. 2025-2026

Abstract

The emergence of "Vibe Coding" promises a democratization of software development, empowering individuals with no programming knowledge to create functional software. This thesis investigates the feasibility and effectiveness of this approach specifically for rapid game prototyping using Large Language Models (LLMs).

The study aims to quantify the "Quality Gap" between human-crafted and AI-generated games while identifying the optimal tools and prompting techniques for this new workflow. We established a rigorous methodology for performance analysis, generating three distinct game prototypes via LLMs to assess development efficiency, while selecting two for strictly comparative benchmarking against their human-authored counterparts. Through blind playtesting with 27 participants and comprehensive development logs, we collected quantitative and qualitative data to measure the efficiency of the development process and the quality of the resulting prototypes.

Results validate Vibe Coding as a robust and viable methodology for preproduction, offering a $\sim$40x Velocity Multiplier, while producing games of sufficient quality and fun. This rapid iteration, although it still reveals a "Quality Gap", results in the creation of "Cheap Fun" (i.e., prototypes that are remarkably cost-effective to produce, yet offer a constrained level of appeal compared to human-authored experiences). By accepting lower initial quality in exchange for rapid iteration, designers can leverage LLMs for quickly building game prototypes while radically reducing the cost of failure in the early stages of design. Furthermore, the study confirms the LLM's role as a highly effective "Creative Partner", successfully proposing novel mechanics and enabling a paradigm shift where the developer's role evolves from manual implementation to high-level curation and design.

Table of Contents

Chapter 1

Introduction

The game development industry faces a fundamental challenge in the early stages of design: how to rapidly prototype and validate novel game mechanics without investing significant time and resources in full-scale development. Traditional digital prototyping requires proficiency in game engines such as Unity or Unreal Engine, creating a substantial barrier between initial concept and playable prototype [1]. Research indicates that while these high-fidelity tools offer immense power, their steep learning curves and programming requirements often force a reliance on technical specialists, significantly delaying the validation of design concepts [2]. This bottleneck can stifle innovation, as promising ideas may be abandoned before their potential is properly assessed [3].

The prototyping phase (often termed "pre-production") represents a massive financial investment rather than a quick sketch. Standard industry frameworks indicate that this phase typically consumes 10% to 25% of a project's total schedule, requiring months of work from cross-functional teams [4]. The cost to simply validate a mechanic is prohibitive; for a standard indie project with a $100,000 budget, pre-production alone can burn through $10,000 to $15,000 before full production even begins [5]. This high "price of entry" creates a risk-averse environment where studios are reluctant to experiment with unproven concepts, knowing that a failed prototype represents an irrecoverable sunk cost.

While traditional pre-production suffers from bloated schedules and financial risks, game jams offer a radical alternative by condensing the entire development

cycle into a 48–72 hour sprint. These events are defined by their nature as "accelerated, constrained, and opportunistic game creation events" [6]. However, this acceleration creates a paradox: while it removes the financial cost of time, it drastically increases the immediate demand for technical proficiency. Although the format is often praised for fostering the learning of technical skills and new tools [7, 8], empirical studies suggest this learning is typically limited to lightweight utilities rather than deep engine mastery; in fact, core technical skills do not statistically increase during the event [9]. This confirms that despite the potential for experimentation, the format still relies on pre-existing engineering competence to produce functional prototypes, rather than serving as a viable environment for overcoming the steep learning curves of professional engines. Consequently, the technical barrier acts as a persistent gatekeeper, meaning the game jam format inherently favors established engineering competence over pure design innovation.

Recent advances in Large Language Models (LLMs) have introduced a paradigm shift in software development through a practice termed "vibe coding" [10]—a methodology where developers create functional software primarily through natural language instructions rather than direct code manipulation [11, 12]. Originally coined by Andrej Karpathy [10], vibe coding represents a fundamental reimagining of the human-computer interaction in software creation, where the developer focuses on high-level intentions while the AI handles the implementation details.

Formally, vibe coding is defined as an "AI-assisted software development approach in which the programmer communicates their functional intent, emotional tone or style (vibe), and contextual constraints to an intelligent agent". Unlike traditional coding, which is syntactic, vibe coding can be conceptualized mathematically as a mapping function

$$\mathcal{A} : (I, E, C) \mapsto P$$

where $\mathcal{A}$ is the AI agent, I represents functional requirements, E denotes vibe descriptors (e.g., "playful", "minimalist"), C represents constraints, and P is the resulting executable program [13]. In this paradigm, the developer's role shifts from writing syntax to managing high-level specifications, relying on the agent to select algorithms and UI patterns that align with the requested "vibe".

A key distinction between "vibe coding" and "AI-assisted coding" is the developer's relationship with the codebase. While AI-assisted coding functions as a

hybrid model where the developer remains the pilot—reviewing and maintaining responsibility for every line—vibe coding implies generating code without the intent to understand the underlying implementation. In this model, the developer treats the code as a "black box", focusing entirely on the high-level functional outcome rather than the syntax. Consequently, the process shifts from a deterministic workflow to an inherently probabilistic one, where the same prompt may yield different results upon repeated execution [14]. This approach differs fundamentally from traditional code completion tools, which function primarily as advanced syntax assistants. These traditional tools operate by predicting code token-by-token in a single pass based on local context, leaving the developer firmly in control of the micro-logic and architectural structure. In contrast, the core methodology of vibe coding alters the input mechanism entirely: the developer articulates high-level goals and constraints in natural language, relying on the model to autonomously generate the underlying architecture and syntax. This effectively transfers the human's cognitive load from syntactic typing to systemic design. Furthermore, advanced implementations of this paradigm increasingly rely on autonomous agentic workflows to plan, reason, and self-correct [13].

Vibe coding has already proven highly effective in business domains where immediate utility supersedes long-term maintainability. In marketing, teams are adopting "vibe marketing" to generate disposable, hyper-personalized landing pages and interactive demos for specific client pitches [15]. Similarly, product managers are increasingly bypassing static wireframes to build functional "vibe-coded" prototypes that validate business logic prior to engineering handoff [16]. Ultimately, as this paradigm gains traction across diverse sectors, it highlights the possibility of a future where software creation is democratized.

Applied to the game development context, vibe coding offers a potentially transformative approach to the prototyping challenge. If effective, it could enable designers without extensive programming knowledge to rapidly iterate on game mechanics, test multiple variations, and validate concepts before committing to engine-based development [17, 18]. Advanced systems have already begun exploring LLMs to grow game prototypes semi-autonomously [19].

The concept of artificial intelligence functioning as a creative collaborator in the game development field is not entirely new; past studies in Automated Game

Design (AGD) and Procedural Content Generation (PCG) have long investigated the potential for mixed-initiative co-creation, where bespoke AI systems assist in generating level layouts and game rulesets [20], [21]. However, as LLMs represent the next major evolutionary step in generalized computational logic, their vast generative capabilities suggest they might fundamentally elevate this collaborative dynamic. Instead of merely executing explicit developer instructions within narrow, predefined parameters, modern LLMs possess the potential to actively propose novel mechanical variations and conceptual improvements to the design itself—a dynamic this research explicitly investigates.

1.1 Research Questions and Objectives

This research investigates the viability and effectiveness of vibe coding as a methodology for rapid digital prototyping in game development. Specifically, three primary research questions are addressed:

- **RQ1 (Feasibility & Quality):** Is it possible to create working digital prototypes through vibe coding without touching a line of code, but simply through natural language instructions? Are these prototypes comparable to human-crafted ones?
- **RQ2 (Efficiency & Tools):** Which LLMs and prompting techniques among the currently used ones are best suited for rapid game prototyping tasks, and what is the quantitative gain in performance?
- **RQ3 (Creative Agency):** Is an LLM capable of and effective in proposing novel game mechanic iterations that improve upon the initially proposed version of a game?

Collectively, these questions explore a significant paradigm shift for two key demographics: non-technical creators, who may now bypass traditional programming barriers to build functional games, and independent developers, who can leverage these tools to prototype experimental concepts with the velocity of a full studio. Each question addresses a distinct facet of this transformation:

- **RQ1:** If validated, this methodology fundamentally shifts the creator's role from a technical translator (writing syntax) to a creative director (evaluating outcomes), effectively democratizing the field of game programming for designers, artists, writers, and producers lacking formal engineering skills.
- **RQ2:** As the market floods with AI coding assistants, this study provides an empirical snapshot of how current architectures handle complex game logic. Acknowledging the rapid evolution and obsolescence of specific model benchmarks, the primary objective is to establish a robust experimental framework—providing practitioners with a methodological "heuristic kit" to systematically evaluate the next wave of LLM models.
- **RQ3:** This question investigates the potential of AI as a collaborative design partner rather than a mere execution engine. By testing the LLM's capacity to propose mechanic improvements, an evaluation is conducted to determine whether vibe coding can enhance the creative process itself, expanding the technical feedback loop to serve simultaneously as a generative brainstorming session.

To address these questions, systematic game development through vibe coding has been combined with empirical user evaluation. Three games from game jam submissions have been selected, each representing distinct mechanical paradigms, and recreated using exclusively natural language prompting across multiple LLM platforms. The resulting prototypes were then evaluated against their original counterparts through structured user testing protocols.

1.2 Contributions

This research makes the following contributions:

1. A systematic methodology for rapid game prototyping through vibe coding, including prompting techniques and tool selection criteria.
2. Empirical evidence regarding the capabilities and limitations of current LLMs in game mechanical implementation through natural language.

3. A comparative evaluation framework for assessing functional equivalence and player experience between vibe-coded and traditionally developed games.
4. An assessment of LLM creative contribution in mechanical design and iteration.
5. Practical guidelines for game designers and developers considering vibe coding for rapid prototyping workflows.

1.3 Structure of the Research

The remainder of this document is organized as follows. Chapter 2 reviews related work in LLM-assisted game development, rapid prototyping methodologies, and game experience evaluation. Chapter 3 details the research methodology, including tool selection, prompting techniques, game selection criteria, and the iterative development process. Chapter 4 describes the experimental protocol for user evaluation. Chapter 5 presents the findings addressing each research question. Chapter 6 draws conclusions on the research and, finally, Chapter 7 discusses implications, limitations, and future directions.

Chapter 2

Background and Related Work

This section establishes the theoretical and practical foundations of this research by examining three interconnected research areas: (1) the emergence of vibe coding and LLM-assisted development, (2) rapid prototyping methodologies in game design, and (3) techniques for evaluating game mechanics and player experience.

2.1 Vibe Coding and LLM-Assisted Game Development

The term "vibe coding" emerged in 2025 as a descriptor for a development methodology where programmers leverage Large Language Models to generate code from high-level natural language descriptions, minimizing direct code manipulation [10, 11].

The roots of assisted coding lie in IntelliSense, introduced by Microsoft in 1996. For decades, these tools relied on static analysis—querying in-memory databases of class definitions to suggest valid variable names or functions [22]. A significant shift occurred in 2017 with the release of IntelliCode, which introduced machine learning to the process. Unlike its predecessors, IntelliCode did not just list available options; it used statistical models trained on thousands of open-source

projects to infer the "most likely" language feature a developer intended to use [23]. In 2022, the release of GitHub Copilot marked the transition from "completion" to "generation". By utilizing large language models (LLMs) rather than simple statistical inference, these systems could suggest entire lines or blocks of logic based on surrounding context [24]. This technological foundation paved the way for the "Agentic" era (2024–present), where tools evolved from predicting text to managing entire workflows—enabling a paradigm where developers direct high-level intent rather than managing low-level syntax [25].

Vibe coding relies on a specific set of recent architectural breakthroughs. While early models were constrained by short memory and shallow pattern matching, modern LLMs leverage three critical capabilities that allow them to function as autonomous engineers:

- **Context Windows:** This refers to the maximum amount of information a model can process in a single pass. Modern architectures now support massive windows—ranging from 128,000 to over 1 million tokens—effectively giving the AI a "working memory" large enough to ingest entire repositories. This is critical for context, as it allows the model to see project-wide dependencies and architectural patterns rather than just isolated files [26].

- **Semantic Code Understanding:** Beyond processing natural language, these models possess deep code understanding [27], allowing them to map abstract human intent (e.g., "make it thread-safe") to precise executable logic.

- **Goal-Oriented Autonomy:** Rather than simple text prediction, advanced vibe coding utilizes an agentic, goal-oriented architecture (often described as "Reasoning and Acting"). The system does not just output code; it identifies a high-level objective, decomposes it into a plan of intermediate steps (e.g., "write test", "implement function", "debug error"), and executes them sequentially to reach the desired state [28].

Despite persistent skepticism regarding code quality and maintainability [29, 30, 31], the practice has gained traction, particularly in rapid prototyping contexts where speed of iteration outweighs long-term codebase concerns [16]. By 2025, AI tool adoption reached near-saturation, with 84% of developers reporting usage and

51% relying on them daily [32]. However, this ubiquity masks a critical "trust gap": positive sentiment for AI tools decreased from over 70% in 2023 to just 60% in 2025. Furthermore, more developers actively distrust the accuracy of AI tools (58%) than trust it (42%), with only a fraction (4%) reporting that they "highly trust" the output [32].

Several researchers have explored LLM applications in game development contexts. [33] demonstrated procedural generation of browser-based JRPGs using LLMs for narrative and world-building, though mechanical implementation remained manually coded. [18] introduced an automated framework for card game prototyping that uses LLMs to generate novel game variations and consistent gameplay environments. Their work focused specifically on card games with discrete rule systems, leaving open questions about more complex interactive mechanics.

[19] developed an AI-driven assistant for game design that elaborates initial concepts into implementation plans for Unreal Engine, demonstrating semi-autonomous game prototyping. However, their approach requires significant domain expertise to interpret and refine the generated plans. [34] explored LLM-generated behavior trees for game-playing agents, showing that language models can reason about game mechanics at abstract levels.

[35] introduced GVGAI-LLM, a benchmark adapted from the General Video Game AI framework to evaluate the reasoning and problem-solving capabilities of LLMs. By testing agents in a zero-shot setting across diverse rule-based games, they revealed fundamental limitations in symbolic reasoning, spatial understanding, and planning capabilities of LLMs when playing games.

[36] introduced LIGS (LLM-Infused Game System), a framework that uses an LLM as the core engine to manage emergent storytelling. By allowing players to interact via natural language, LIGS significantly enhances agency, transforming the game loop from rigid state machines to fluid, conversational simulations and demonstrating the potential for new types of design tools. However, this approach also introduces challenges regarding unpredictability and hallucination, suggesting that LLMs are not an all-encompassing solution. Consequently, further research is necessary to clearly define the function of LLMs in game development and understand how their integration redefines the role of the game developer itself.

[37] provides a comprehensive typology of LLM roles in games, and classifies AI

design assistants into three levels of autonomy: *conceptual assistance*, where the AI provides high-level ideation and suggestions; *procedural assistance*, where the AI iteratively refines artifacts through dialogue; and *production assistance*, where the system directly generates game content under designer supervision. They outline a roadmap that moves beyond simple asset generation, stimulating research into tools that foster the creative process and enable novel game mechanics. However, they caution that widespread adoption is currently hindered by technical limitations such as hallucination, limited context windows, and prohibitive inference costs.

A critical gap in existing research is the systematic evaluation of whether vibe coding can produce functionally equivalent game prototypes compared to traditional development methods, particularly for games with complex interactive mechanics.

2.2 Rapid Prototyping in Game Development

Game prototyping serves as the critical bridge between concept and production, allowing designers to "find the fun" before committing substantial resources [4, 5, 38]. Traditional approaches span a spectrum from paper prototypes for testing rule systems to digital prototypes built in game engines for evaluating feel and interaction dynamics.

Central to this process is the concept of the Minimum Viable Product (MVP), defined as "a version of a new product, which allows a team to collect the maximum amount of validated learning about customers with the least effort" [39]. Unlike traditional software MVPs that prioritize functional utility, a game MVP—often termed a "Minimum Viable Game"—aims to isolate and validate the core gameplay loop to ensure it is intrinsically engaging before broader content production begins [40, 41]. In executing this, developers typically choose between vertical and horizontal slices [42]. A *vertical slice* serves as a proof of quality, delivering a narrow but fully polished cross-section of the game (e.g., a single complete level with final art, sound, and mechanics) to test the production pipeline and player experience. Conversely, a *horizontal slice* focuses on scope, implementing a specific functional layer across the entire game (such as the full UI flow or all level layouts with placeholder assets) to evaluate pacing and volume. Early validation through these methods is crucial for risk management, as it allows teams to "fail fast"—identifying

fundamental design or technical flaws when the cost of correction is minimal, rather than discovering them after significant budget has been exhausted [43, 44].

Game jams represent an important case study in rapid prototyping under extreme time constraints. [45] analyzed how game jams facilitate rapid prototyping through time-bounded competition and multidisciplinary collaboration. They identified key patterns: focusing on core mechanics rather than polish, leveraging existing assets and code libraries, and accepting technical debt (i.e., the implied cost of future rework incurred by choosing quick, suboptimal solutions to meet immediate deadlines) for speed. [46] extended this analysis to educational contexts, proposing "Serious Slow Game Jams" that balance speed with reflection.

In the broader landscape of rapid prototyping, developers must navigate a trade-off between flexibility, ease of use, and implementation speed. Reliance on proprietary custom engines, while offering total architectural control, often presents a paradox for rapid prototyping: significant development time is consumed by building foundational tools rather than iterating on gameplay mechanics, effectively delaying the validation phase [47, 48]. Conversely, industry-standard engines like Unity and Unreal provide immediate functionality but introduce a steep learning curve and significant overhead. Their feature-rich ecosystems—which combine complex visual editors with external Integrated Development Environments (IDEs)—can be daunting for small-scale experimentation, often forcing developers to navigate substantial boilerplate code and intricate asset pipelines merely to test simple ideas [49].

To mitigate these technical barriers, visual environments like GameMaker facilitate faster entry. However, these tools still demand a rigorous grasp of programmer logic (e.g., state machines, event handling), merely replacing textual syntax with visual abstraction without removing the underlying computational requirements [50, 51]. [52] introduced the concept of "Live Game Design", where designers can prototype and playtest simultaneously through immediate visual feedback. Their Vie tool enables rapid iteration on 2D game economies without traditional coding. While promising, such approaches remain limited to specific game genres and require visual programming literacy.

Positioned between these extremes are code-centric frameworks such as Pygame, which was utilized in this study. Pygame minimizes the overhead of full-scale game

engines, offering a lightweight, script-based environment ideal for rapid testing of 2D mechanics. While it avoids the bloat of commercial engines, it remains strictly code-dependent, necessitating a foundational understanding of programming structures similar to standard software development.

Current rapid prototyping methodologies, while faster than full production, still require substantial technical expertise and tool-specific knowledge. The question of whether natural language interfaces can further reduce these barriers is now being actively investigated, with recent academic frameworks and industry experiments demonstrating significant potential. [53] demonstrated the feasibility of generating playable Super Mario Bros levels directly from natural language prompts using fine-tuned GPT-2 models (MarioGPT), experimenting with LLMs working within spatial constraints. [54] documented the use of LLMs to generate functional prototypes directly from Game Design Documents (GDDs) using Cursor/Zed in Agent Mode, reducing the time from concept to playable artifact within the LÖVE game engine. [55] developed LLMR, a framework where GPT-4 receives instructions from a user and converts them into a series of tasks executed within the engine to create and modify interactive scenes in real-time. [56] introduced "Cardiverse", a system that utilizes LLMs to automate the entire prototyping loop for card games, generating both the rule sets and the executable code. [57] provided a comprehensive case study through *Echoes of Somewhere*, an experimental 2.5D game, documenting a hybrid pipeline where generative AI produced everything from background art to code, demonstrating that a single developer can achieve near-production quality prototypes by offloading technical debt to AI agents.

Despite these advances, most existing work focuses on asset generation or isolated scripts rather than holistic, end-to-end prototyping for general-purpose game development.

2.3 Evaluating Game Experience and Mechanics

Assessing whether a prototype successfully captures intended game mechanics requires both objective functionality testing and subjective player experience evaluation. The game user experience research community has developed numerous validated instruments for this purpose.

The Game Experience Questionnaire (GEQ) [58] represents a foundational tool, measuring seven core dimensions: competence, sensory and imaginative immersion, flow, tension/annoyance, challenge, negative affect, and positive affect. Its widespread adoption stems from psychometric validation and sensitivity across diverse game genres. [59] developed the Game User Experience Satisfaction Scale (GUESS) specifically for playtesting contexts, with nine subscales covering usability, narratives, play engrossment, enjoyment, creative freedom, audio aesthetics, personal gratification, social connectivity, and visual aesthetics. GameFlow [60] adapts Csikszentmihalyi's flow theory into a model for heuristic evaluation. It identifies eight core elements—concentration, challenge, skills, control, clear goals, feedback, immersion, and social interaction—often used by expert reviewers to systematically identify design flaws that interrupt player immersion. Alternatively, the Player Experience of Need Satisfaction (PENS) model [61] grounds evaluation in Self-Determination Theory. Rather than surface-level engagement, it measures the satisfaction of three innate psychological needs: competence, autonomy, and relatedness, positing that long-term retention is driven by the intrinsic motivation resulting from these needs being met.

A persistent debate in the field concerns the trade-off between utilizing these standardized instruments versus ad-hoc, game-specific questionnaires. While standardized tools provide necessary benchmarking capabilities, they may fail to capture the specific design intent of unique gameplay loops. Because "fun" is subjective and multifaceted—spanning from "Hard Fun" derived from overcoming meaningful challenges to "Easy Fun" rooted in immersion and exploration—generic metrics may misinterpret intended player experiences [62]. [63] demonstrated that general usability principles frequently fail to identify game-specific problems like inconsistent mechanics or poor control mapping, necessitating the use of domain-specific heuristics. [64] highlighted that relying solely on self-reported questionnaires is often insufficient due to cognitive bias, advocating for tailored experimental setups that triangulate survey data with biometric or observational evidence to fully understand the player experience.

[65] developed the GUESS-18, a validated short-form version of the original scale. By condensing the instrument to just 18 items while retaining the core psychometric structure, the GUESS-18 allows practitioners to gather reliable subjective

data on key dimensions without causing the player fatigue associated with longer questionnaires, making it an ideal instrument for rapid prototyping cycles. [66] further demonstrated correlations between experiential factors (flow, immersion) and objective play metrics, suggesting multi-method evaluation approaches. For serious games, [67] proposed a three-tiered assessment framework evaluating usability, playability, and learning effectiveness. [68] provided a comprehensive taxonomy of evaluation methods spanning quantitative/qualitative and formative/summative approaches. [69] validated a seven-dimensional model including satisfaction, learning, effectiveness, immersion, motivation, emotion, and socialization.

To mitigate the subjectivity of self-reported data, modern frameworks increasingly rely on objective game metrics derived from telemetry. Key performance indicators (KPIs) such as completion rates, time-to-completion, and error rates (e.g., number of failed jumps in a level sector) provide "hard" evidence of usability friction that players may fail to articulate. Consequently, the industry standard has shifted toward a mixed-methods approach, where telemetry identifies what is happening (e.g., a difficulty spike), and subjective questionnaires explain why it is happening, providing a holistic view of the prototype's viability [70].

A central methodological challenge lies in establishing functional equivalence between prototypes developed through disparate implementation pipelines. Comparison cannot rely on code inspection, as different engines employ distinct architectures. Instead, the Mixed-Fidelity framework proposed by McCurdy et al. [71] is adopted. They define Functional Fidelity as the degree to which a prototype's behavior and logic match the intended system, independent of its aesthetic presentation. Under this definition, two prototypes are functionally equivalent if they possess identical "depth" (logic) and "breadth" (feature set), ensuring that the input-output state machine remains consistent across implementations.

However, objective mechanical parity does not guarantee experiential parity. [72] noted that the "micro-interactions" of a game—such as input latency, physics integration, and frame-rate smoothing—create a tactile signature that is unique to the implementation engine. A Python script and a Unity build may share the exact same logic (Functional Fidelity) yet yield distinct kinesthetic experiences. Therefore, validation requires triangulating the objective consistency of the rules with a subjective assessment of the "kinetic response".

2.4 Research Gap and Positioning

Existing research demonstrates that (1) LLMs can assist in various game development tasks, (2) rapid prototyping is essential for validating game concepts, and (3) robust methodologies exist for evaluating game experience. However, to the best of current knowledge, no prior work has systematically investigated whether vibe coding can serve as a viable methodology for creating functionally equivalent game prototypes, nor assessed LLM contributions to creative mechanical design.

Despite the growing prevalence of AI-assisted coding, the literature lacks a holistic evaluation of how these tools reshape the prototyping lifecycle. Specifically, four critical gaps have been identified:

- **Comparative Efficiency vs. Traditional Workflows:** While generation speed is a known advantage of LLMs, there is a lack of systematic, head-to-head comparisons measuring the actual time savings in a complete game prototyping loop. The trade-off between rapid initial generation and the time spent refining or debugging "vibe-based" outputs remains unquantified compared to traditional development methods.
- **From Syntax to Mechanical Fidelity:** Most benchmarks evaluate AI code based on syntactic correctness. There is a scarcity of research assessing mechanical fidelity—the ability of an LLM to understand and replicate complex, interacting systems that define "game feel", rather than merely executing isolated logic functions.
- **Formalization of Prompting Strategies:** "Vibe coding" currently exists largely as an informal community practice. There is a significant gap in defining and validating robust prompting strategies that allow developers to consistently translate high-level natural language intent into functional game architectures without reverting to technical pseudo-code.
- **The Creative Gap:** Existing works often treat the LLM as a passive code translator. There is a lack of understanding regarding the model's creative contribution: specifically, its ability to resolve ambiguity and fill in design gaps to produce a coherent experience when the human prompt provides only a vague "vibe" rather than a strict technical specification.

This research fills these gaps by providing empirical evidence on vibe coding's capabilities and limitations for game prototyping, contributing with both methodological innovations and practical guidelines for developers.

Chapter 3

Methodology

This section details the systematic approach to investigating vibe coding for game prototyping, describing the overall research design, tool selection rationale, prompting methodology development, game selection criteria, development process, and data collection procedures.

3.1 Research Design Overview

This research combines constructive research (creating game prototypes through vibe coding) with empirical evaluation (assessing their quality and comparing them to originals). The methodology proceeds in four phases:

1. **Preparation Phase:** Selection of LLM tools, selection of target games, and development of an iterative prompting strategy.

2. **Development Phase:** Creation of three game prototypes using vibe coding exclusively, through the application of specific prompting techniques and iterative refinement.

3. **Evaluation Phase:** User testing comparing vibe-coded prototypes against original versions.

4. **Analysis Phase:** Quantitative and qualitative analysis of results.

Each phase acts as a foundational step for the next, creating a cohesive research trajectory. The **Preparation Phase** defines the rigid boundaries of the experiment by locking in development tools, LLM models, and target games, ensuring that the methodology remains consistent. This framework governs the **Development Phase**, where the actual prototype generation occurs alongside the empirical measurement of progress. The **Evaluation Phase** is then critical for quantifying the real-world impact of LLMs on rapid prototyping fidelity. Finally, the **Analysis Phase** synthesizes the gathered evidence to derive concrete conclusions regarding the capabilities of vibe coding applied to game prototyping.

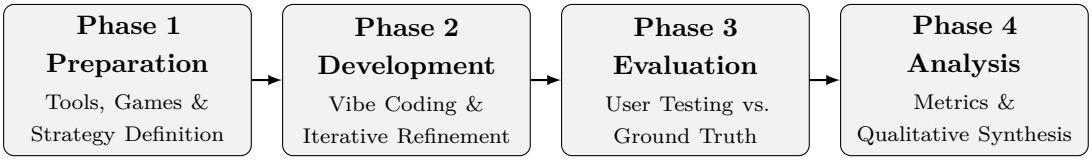

Figure 3.1: The four-phase research workflow. The output of each phase strictly informs the parameters of the next.

3.2 Tool Selection

The selection of appropriate tools is critical for vibe coding effectiveness. Several popular LLM-based development environments were evaluated based on criteria including:

- **LLM Capability:** Model sophistication, context window size, code generation quality, and community benchmarks (e.g., LMSYS Chatbot Arena).
- **Development Environment:** IDE integration, preview capabilities, debugging support, and availability via API.
- **Iteration Speed:** Responsiveness to prompts and ability to refine code.
- **Accessibility:** Cost, availability, and learning curve.

Based on these criteria, three distinct frontier models were selected to represent the current landscape of LLM capabilities:

- **GPT-4.1 (v.2025-04-14):** Selected for its high reasoning scores and widespread adoption as a baseline [73, 74].
- **Gemini 2.5 Pro:** Chosen for its extended context window capabilities, relevant for handling larger codebases [26].
- **Claude Sonnet 4 (v.20250514):** Selected for its reported high performance in creative writing and code generation tasks [75].

Regarding the development environment, while commercial AI-assisted IDEs (e.g., Cursor, Windsurf) were initially considered, they were ultimately rejected to avoid "black box" interference (such as hidden system prompts or non-transparent context management). Instead, to support the academic requirements of this study, a **custom orchestration script** was developed using the LangChain framework. This tailored environment provided specific functionalities required for rigorous tracking:

1. **Isolated Environments:** Each model operated in a unique, stateless sandbox to prevent context cross-contamination.
2. **Programmatic Logging:** All interactions and error logs were automatically stored in log files, every code output was exported into an executable file, and the scores of every iteration were exported to a structured dataset on a spreadsheet for analysis and performance tracking.
3. **State Management:** The tool features a "rollback" capability to revert conversation states and a session resumption feature to ensure experimental continuity in the event of unforeseen hardware interruptions.

Finally, through an exploratory pilot study where simple prototypes of classic games (e.g., *Pac-Man*, *Arkanoid*) were recreated, the overall feasibility of the approach was assessed. This phase did not focus on refining the prompting strategy, but rather on gaining practical familiarity with the workflow and validating that the selected models could successfully generate functional game loops, thereby confirming that the methodology was viable for the main experiment.

3.3 Game Selection Criteria

To ensure that the evaluation addresses representative game development challenges, strict criteria were established for selecting prototype games. The goal was to identify titles that serve as effective benchmarks for both mechanical fidelity and LLM reasoning capabilities:

1. **Clear Core Mechanic:** The game should center on a distinct, identifiable mechanical loop.
2. **Bounded Complexity:** The game loop and rules must be "simple enough" to be described in natural language, yet complex enough to require non-trivial logic.
3. **Traceable Development History:** While full technical documentation is rare, selected games must have sufficient public history (e.g., post-mortems, devlogs, or developer interviews) to allow for the inference of the original design intent and constraints.
4. **Reference Game Available:** The original game must be playable to serve as a direct comparative baseline.
5. **Mechanical Distinctiveness:** Each game should employ mechanics that clearly distinguish it from the others, in order to test distinct forms of computational reasoning (e.g., spatial physics, discrete logic, or systems management) and the model's adaptability across different problem domains.
6. **Game Jam Origin:** Preference is given to games developed under strict time constraints (e.g., 48-hour jams). Because these titles inherently lack secondary content bloat (such as complex meta-progression or elaborate UIs) and focus entirely on a single gameplay loop, they provide a "pure" mechanical target that serves as a fair baseline for evaluating rapid prototyping efficiency.

These criteria ensure that the selected games serve as effective benchmarks. The focus on a **Clear Core Mechanic** with **Bounded Complexity** limits the experiment to architectural logic rather than asset production. **Mechanical Distinctiveness** is crucial to verify that the prompting strategy works not just

for one type of game, but requires the LLM to switch between different modes of reasoning (e.g., calculating continuous physics for an arcade game vs. managing discrete grid states for a puzzle). Finally, the requirement for **Game Jam Origin** ensures the selected targets represent realistic prototyping scopes. By stripping away commercial bloat, jam games offer a pure distillation of a mechanic, enabling a focused evaluation of the LLM's ability to capture the core engagement loop, without penalizing the model for lacking secondary content. Combined with a **Traceable Development History**, this provides a direct comparative baseline to contextualize the time and resources saved through the vibe coding workflow.

3.4 Selected Games

Based on these criteria, three games were selected:

3.4.1 Game 1: Celeste Classic (2015)

- **Origin & Developers:** Created by Maddy Thorson and Noel Berry in 4 days for an unspecified Game Jam [76, 77].
- **Core Mechanic:** A precision platformer focusing on tight movement physics, including jumping, wall-sliding, and a multi-directional dash. The loop involves navigating single-screen rooms filled with hazards to reach an exit.
- **Why Selected:** Represents **continuous spatial physics**, **spatial reasoning**, and **input responsiveness**. It tests the LLM's ability to implement "game feel"—specifically the subtle acceleration, friction, and coyote-time logic required for satisfying platforming—rather than just static rules.
- **Development Context:** Team of 2, approx. 96 hours. Originally written in Lua (PICO-8).
- **Game source:** Playable at [78].

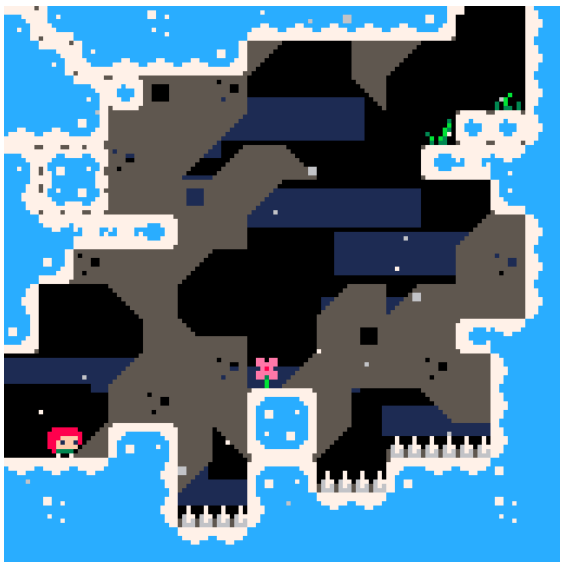

Figure 3.2: Original gameplay of Celeste Classic

3.4.2 Game 2: Mind The Gap (2013)

- **Origin & Developers:** Created by Peter and Robert Curry (Dinosaur Polo Club) for the Ludum Dare 26 game jam [79].
- **Core Mechanic:** A minimalist strategy simulation where players design a subway network. The core loop involves connecting procedurally generated stations with limited tracks and lines to efficiently transport autonomous passengers before stations become overcrowded. This requires managing passenger spawning, pathfinding, and carriage movement along dynamic tracks.
- **Why Selected:** Represents **systems management** and **graph-based logic**. This title tests the LLM's capability to implement agent-based behaviors (passenger pathfinding) and manage dynamic resource constraints within a graph structure, distinct from direct avatar control.
- **Development Context:** Team of 2, approx. 48 hours. Developed using Unity Engine.
- **Game Source:** Playable at [80].

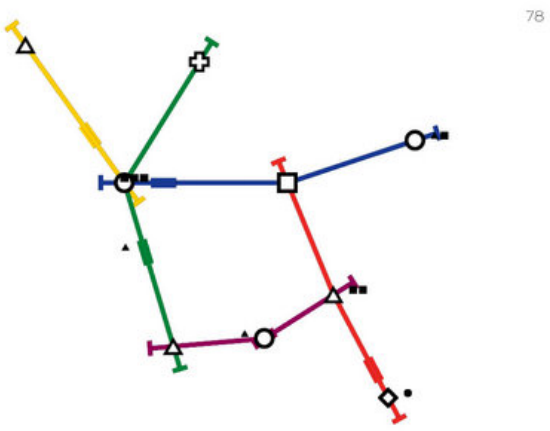

Figure 3.3: Original gameplay of Mind The Gap Classic

3.4.3 Game 3: Sacrifices Must Be Made (2018)

- **Origin & Developers:** Created by Daniel Mullins in 48 hours for Ludum Dare 43 [81].
- **Core Mechanic:** A deck-building roguelike where players sacrifice weak creatures to play stronger ones on a grid-based board. It centers on managing discrete card logic (turn phases, attack values) and resource economy.
- **Why Selected:** Represents **discrete rule systems** and **complex state management**. This title tests the LLM's capacity to implement rigid, turn-based logic structures and 2D array management, requiring precise adherence to multi-step rulesets.
- **Development Context:** Solo developer, approx. 48 hours. Developed using Unity Engine.
- **Game Source:** Playable at [82].

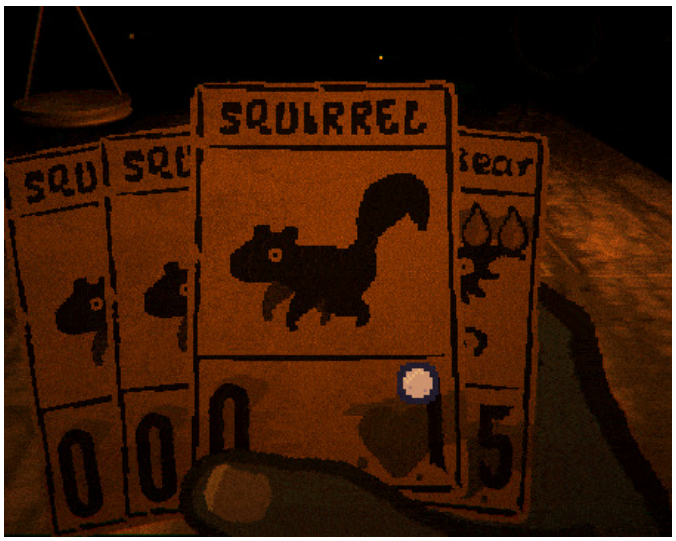

Figure 3.4: Original gameplay of Sacrifices Must Be Made

3.5 Vibe Coding Development Process

For each selected game, a systematic development process using vibe coding was followed:

3.5.1 Phase 1: Analysis

Prior to any code generation, a systematic gameplay analysis was conducted to establish the functional requirements for each prototype. This process involved playing the original titles to identify the "atomic" mechanical elements—such as player physics, control schemes, and win/loss states—that constitute the core experience. Rather than analyzing the source code, the phenomenological aspects of the game (the behavioral rules and "feel") were documented to ensure the subsequent prompts relied solely on natural language descriptions of intent.

Based on this observation, a strict Minimum Viable Product (MVP) scope was defined for each title, distinguishing between essential mechanics and secondary polish. For instance, the MVP definition for *Celeste Classic* required not only the implementation of discrete game states (three connected levels, a victory screen, and basic UI/audio) but also specific "feel" mechanics crucial to the vibe, such as precise jump/dash physics, wall interactions, and "coyote time" (i.e., a grace period allowing jumps after leaving a platform). Crucially, to be considered a valid MVP, the game was required to be fully completable from start to finish without critical logical failures or "soft-lock" states that would prevent progression. This prioritized feature list served as the ground truth for evaluating the success of the vibe coding process.

3.5.2 Phase 2: Initial Prompt Formulation

To evaluate how the semantic structure of instructions affects the LLM's output, a single initiation prompt was not relied upon. Instead, four distinct prompting strategies were established, each serving as the "seed" for a separate, parallel development branch. For all strategies, the technical target was fixed: the model was instructed to generate a complete, playable prototype in Python using the Pygame library.

For each game, four independent development threads were initiated using the following strategies:

1. **The Naïve Baseline:** A minimal, direct instruction relying entirely on the LLM's pre-trained knowledge (e.g., *"Please, generate [Game Name] in Python using Pygame"*). This strategy acts as a control group to evaluate the necessity of prompt engineering: it tests whether complex, structured prompts provide a tangible improvement over simple instructions, or if the model's inherent training is sufficient for rapid prototyping.

2. **Schell's Elemental Tetrad:** A structured prompt organized according to Jesse Schell's framework [38].

3. **Fullerton's Formal Elements:** A prompt structure based on Tracy Fullerton's design framework [83].

4. **The Hybrid System:** A synthesized approach explicitly combining Schell's and Fullerton's frameworks.

Strategies 2, 3, and 4 utilize a Structured Prompting technique [84] to facilitate Task Decomposition. By enforcing specific theoretical frameworks as rigid input templates, the model's output space was constrained, ensuring a holistic coverage of game system requirements [85]. This approach guides the LLM to "think" in modular design components rather than monolithic code blocks. Crucially, all initial prompts were strictly text-based; no visual references, pseudo-code, or images were provided.

3.5.3 Phase 3: Iterative Development Loop

Following the initial prompt, the development process transitioned into a standardized iterative cycle to refine the codebase. To ensure strict experimental consistency, the exact same sequence of initial prompts was replicated for every target Large Language Model. This phase utilized a "human-in-the-loop" architecture [86] where the LLM functioned as the sole implementation engine, while the human researcher acted as the supervisor and validator.

For each game, model, and prompting strategy, the development followed this strict protocol:

1. **Prompt Submission:** The text-based instruction—whether the initial Phase 2 prompt or a subsequent refinement—was transmitted to the specific LLM.

2. **Automated Injection & Execution:** Upon receiving the response, the custom Python/LangChain tool parsed the model's text output, extracted the code blocks, and injected them directly into a local `.py` file.

3. **Behavioral Analysis:** The prototype was tested against the MVP requirements defined in Phase 1.

4. **Feedback:** The applied feedback strategy depended on the type of problem encountered:

    - *Critical Runtime Failures:* If the application failed to launch or crashed, the raw Python stack trace was pasted directly back to the model without additional commentary.
    - *Logic and "Vibe" Defects:* For functional issues (e.g., incorrect physics, missing mechanics), feedback was provided purely in natural language. A "Current vs. Desired" structure was utilized, explicitly describing the observed flaw and the intended behavior (e.g., *"The player sticks to the wall indefinitely, whereas they should slide down slowly due to friction"*).

Throughout this process, strategic pivots and decision points were documented, adopting an adaptive approach that prioritized high-efficacy strategies while pruning suboptimal branches. For instance, in *Celeste Classic*, it was observed that including a concrete data structure example for level layout consistently improved spatial coherence; consequently, this pattern was standardized across subsequent iterations. Conversely, it was noted that low-temperature sampling (which forces deterministic output) often hampered the models' ability to generate creative solutions for complex logic flaws. As a result, strictly deterministic parameter branches were frequently deprecated in favor of balanced temperature settings to encourage novel problem-solving.

Notably, a **"Rethink Strategy"** was employed to resolve persistent logical dead-ends. When a model became stuck in a loop of ineffective fixes (particularly when addressing the same bug multiple times), it was explicitly instructed to discard the previous implementation entirely and re-derive the mechanic's logic from scratch rather than attempting incremental patches (e.g., *"...please discard the current logic and rethink from scratch how the pathfinding algorithm should work"*).

To standardize the evaluation of vibe coding viability, rigid termination conditions were established. The development cycle for a specific branch concluded when one of the following states was reached:

- **Success (MVP Reached):** The prototype successfully implemented all "atomic" mechanical elements and was fully completable.
- **Failure (Convergence Threshold):** The development exceeded 15 interaction turns without achieving MVP status. This limit was not merely an arbitrary efficiency constraint but a measure of *recoverability*. Preliminary tests indicated that if a model cannot converge on a stable core loop within 15 iterations, the codebase typically suffers from "context drift" and accumulated logical entropy, rendering further prompting counterproductive.

Upon the conclusion of each branch, a **Comparative Quality Assessment** was performed, recording specific performance scores for both the first executable iteration and the final output (metrics defined in the following section). These evaluations provided the necessary delta to measure the "vibe" improvement over time and helped quantify the effectiveness of the iterative interventions.

By the conclusion of the study, a total of **126 distinct game prototypes** had been generated. The distribution was weighted by complexity and methodological refinement: 63 iterations for *Celeste Classic*, 42 for *Mind the Gap*, and 21 for *Sacrifices Must Be Made*, with trials equally divided among the three target LLMs. The reduction in sample size for later titles reflects two factors: the strategic pruning of ineffective prompt branches (e.g., the low-temperature exclusion) and a documented "learning effect" where the researchers' improved prompting efficacy reduced the trial volume needed to reach a satisfying result.

3.5.4 Phase 4: Testing and Refinement

Once a prototype reached the MVP state or the iteration limit, it underwent a final validation phase. The testing protocol prioritized "phenomenological alignment"—ensuring the generated game's "feel" matched the original reference title rather than merely satisfying a functional checklist.

The validation process consisted of three distinct assessment layers:

- **Functional Verification:** Manual playtests were conducted to confirm that all atomic mechanics identified in Phase 1 were present and operative. This included stress-testing specific logic gates across the different genres: verifying that the "coyote time" physics in *Celeste Classic* actually allowed late jumps, ensuring that passenger overcrowding in *Mind the Gap* correctly triggered a Game Over state, and validating that the turn-order sequence was strictly respected in *Sacrifices Must Be Made*.
- **Comparative "Vibe" Analysis:** To assess the fidelity of the simulation, side-by-side comparisons with the original reference games were performed. This step evaluated dynamic behaviors beyond static features: Is the gravity curve similar? Do the collisions feel fair? Discrepancies were noted not as bugs, but as "fidelity gaps" that influenced the final quality score.
- **Aesthetic and Performance Constraints:** Given the study's focus on logic generation over asset creation, aesthetic refinement was strictly subordinate to functional clarity. "Programmer art" (e.g., geometric primitives representing entities) was accepted provided that the dimensions and hitboxes accurately represented the game objects. Performance benchmarks were defined by stability rather than optimization: code was deemed acceptable if it maintained a consistent framerate without perceptible stuttering or runtime crashes during standard play sessions.

3.6 Prompting Methodology

Effective vibe coding requires systematic prompting strategies. Through iterative refinement during pilot studies, a structured prompting methodology was developed,

consisting of:

3.6.1 System Prompt

To standardize the initialization state across the Schell, Fullerton, and Hybrid tracks, a Role-Prompting technique [87] was utilized to define the model's persona and technical boundaries.

The following prompt was prepended to every interaction:

> *"You are a Python game developer who specializes in creating 2D games using the Pygame library... Your goal is not to create a full playable game, but a prototype for iterating game designers' ideas..."*

Crucially, this instruction enforced specific Operational Constraints derived from early pilot failures, such as the **Rewrite Rule** (*"Each time code is requested, don't just write the modifications, but rewrite the entire script"*) to mitigate integration errors, and **Asset Abstraction** (*"Use simple placeholder graphics..."*) to prioritize logic over asset generation. (See Appendix A.1 for the full System Prompt text).

3.6.2 Structural Schemas

While the Naïve track relied on unstructured natural language, the three experimental tracks utilized rigid Schema Injection. Prompt inputs were explicitly structured using hierarchical lists to force the model's attention mechanism onto specific design domains.

The Schell Schema

Based on Jesse Schell's Elemental Tetrad [38], this structure grouped requirements into four high-level domains, offering a broad conceptual separation of concerns. It provides a structural framework that focuses on the holistic relationships between game elements, encouraging the model to view the game as a cohesive system of broad categories rather than a list of isolated functions:

- **1. Mechanics:** Rules, Player Actions, Core Loop, Win/Loss Conditions, Challenge, Progression, Entities, Resources, Additional Mechanics.

- **2. Story:** Setting, Theme, Player Role, Narrative Arc.
- **3. Aesthetics:** Visual Style, Audio Feedback, UI Elements.
- **4. Technology:** Screen Size, FPS, Input Method.

(See Appendix A.2 for an example of the Schell Prompt).

The Fullerton Schema

Derived from Tracy Fullerton's "Game Design Workshop" [83], this schema is highly granular. It forces the model to decompose the game into functional logical units rather than conceptual themes, requiring the LLM to process the design as a detailed system of interacting formal rules and elements:

- **Players:** Defining the actor(s) in play.
- **Objectives:** Defining the goal state.
- **Procedures:** Methods of play and the actions that players can take to achieve the game objectives. They are sub-divided into *Starting Action*, *Progression of Action*, and *System Procedures*.
- **Rules:** Defining game objects and allowable actions by the players. Explicitly categorized into *Rules defining objects*, *Rules restricting actions*, and *Rules determining effects*.
- **Resources:** Assets that can be used to accomplish certain goals.
- **Conflict:** Elements that do not allow players to accomplish their goals directly. They can be decomposed into *Opponents*, *Obstacles*, and *Dilemmas*.
- **Boundaries:** What separates the game from everything that is not the game.
- **Outcome:** Defining the resolution states.

(See Appendix A.3 for an example of the Fullerton Prompt).

The Hybrid Schema

This prompt structure integrates Schell's and Fullerton's approaches, aiming to provide the most comprehensive instruction set by leveraging the strengths of both design philosophies. It attempts to combine Schell's conceptual clarity with Fullerton's logical rigour. Structurally, it nests Fullerton's formal logic *inside* Schell's "Mechanics" category, while retaining Schell's "Story" and "Aesthetics" to capture the highest number of elements in the clearest way. (See Appendix A.4 for an example of the Hybrid Prompt).

3.6.3 Prompt Variations Tested

Beyond static schemas, specific engineering variables were manipulated to optimize the generation process. While full quantitative analysis is reserved for Section 5, the assessment of these variables prioritized the absolute quality scores of the **first executable iteration** and the **final output**, alongside the **Overall Quality Delta** (the magnitude of improvement between the two). Evaluating both the initial baseline viability and the prototype's macro-evolution established a clear hierarchy of efficacy among the tested variables.

- In the initial Naïve branches, **Zero-Shot** prompting was employed, relying solely on the model's internal training data. However, a **"Contextual Necessity"** for external examples in complex tasks was identified. While zero-shot sufficed for standard logic, it proved inadequate for spatial data representation. Consequently, **One-Shot** and **Few-Shot** techniques were introduced—providing concrete, paradigmatic examples of data structures (e.g., a sample tilemap array for *Celeste Classic*)—which significantly reduced hallucination by guiding the model's structural implementation.

- **Single-Step vs. Two-Step Generation:** It was observed that asking for code immediately often resulted in logic errors. Consequently, a **Two-Step Generation** protocol was implemented for the complex tracks:

    1. *Step 1:* "Please analyze the requirements and ask for clarifications for whatever element is unclear or ambiguous."

2. *Step 2:* "Now, implement this logic in Python/Pygame."

This intervention proved to be very effective. By decoupling reasoning from implementation, it induced a *Chain-of-Thought* process that allowed the model to resolve logical ambiguities before committing to syntax, consistently reducing the number of subsequent debug iterations.

- **Temperature Modulation:** As noted in the iterative methodology, the sampling temperature was manipulated to balance determinism with creativity. Contrary to standard software engineering practices which favor low temperature (e.g., $0.1 - 0.4$) for deterministic code output, the empirical data indicated that static low temperatures were detrimental to the rapid game prototyping process using vibe coding. They frequently trapped models in recursive error loops, where the agent repeatedly applied the same ineffective fix. Consequently, it was found that a medium/high temperature (e.g., 0.7) was necessary during the planning phase. This higher entropy enabled the "creative friction" required to bypass logical dead-ends and infer implicit game mechanics not strictly defined in the prompt.

3.7 Data Collection and Evaluation Metrics

To ensure a robust analysis of the workflow, a comprehensive data collection strategy was implemented. Throughout the development process, the following were systematically recorded:

- **Prompt Log:** A complete, timestamped transcript of all natural language inputs and the corresponding LLM text/code responses.
- **Code Versions:** A comprehensive version control history tracking every iteration of the codebase.
- **Development Journal:** Qualitative notes on challenges, decisions, and observations.

Complementing these artifacts, a set of **Subjective Quality Heuristics** was established to quantify the "vibe" evolution. While primarily serving as a navigational tool during development—allowing for the identification of high-potential

prompt branches and pruning ineffective ones—these metrics also provided the quantitative data needed to measure the efficacy of the iterative process. Scores were recorded at two critical snapshots for every experimental branch: immediately following the **Initial Prompt** and immediately following the **Final Iteration** (whether MVP or Failure).

Crucially, the interpretation of these metrics was context-dependent. For example, the *Gameplay Mechanics* score measured the accuracy of physics simulation in *Celeste Classic*, whereas it evaluated the rigour of discrete logic rules in *Sacrifices Must Be Made* or the efficiency of graph simulation in *Mind the Gap*.

This protocol yielded a structured dataset where each snapshot was evaluated on a 1–10 scale across eight dimensions:

- **Prompt Adherence:** Fidelity to explicit constraints and requested features.
- **Specification Interpretation:** The ability to deduce implicit requirements and handle edge cases.
- **Gameplay Mechanics:** Responsiveness of controls, accuracy of simulations, and overall playability relative to the genre.
- **Visual Consistency:** Coherence of the "programmer art" and spatial dimensions.
- **Audio Integration:** Successful implementation of sound effects and loops.
- **User Experience (UX):** Clarity of interfaces, menus, and control schemes.
- **Technical Stability:** Framerate stability and absence of runtime crashes.
- **Creative Problem-Solving:** Quality of "improvised" decisions for underspecified design aspects.

This dual approach—combining objective artifact tracking with structured qualitative scoring—enables a granular analysis of the iterative improvements, distinguishing between technical code generation and "vibe" replication.

Chapter 4

Experimental Evaluation

This section describes the experimental protocol for evaluating both the vibe-coded game prototypes and the generative workflow that produced them. Aligning with the core research questions, the evaluation addresses three primary objectives: (1) assessing whether vibe-coded prototypes achieve functional and experiential equivalence to their human-crafted originals, (2) quantifying the development efficiency and performance variances among different Large Language Models, and (3) determining the viability of LLMs as creative co-designers through user acceptance of generated mechanics.

4.1 Experimental Design

The study utilized a within-subjects experimental design, requiring all participants to play both the original and the vibe-coded versions of each game. This methodology was deliberately chosen to neutralize the confounding variable of individual gaming proficiency. By experiencing both iterations sequentially, playtesters established a direct experiential baseline, allowing them to evaluate the "feel" and responsiveness of the AI-generated mechanics against a concrete, human-crafted reference rather than an abstract standard.

4.1.1 Participant Recruitment

Data collection was conducted *in situ* during "Game Matters: The Game Jam", an industry event hosted by IGDA Milan in 2025 [88]. This setting provided access to a high-quality participant pool predominantly composed of game development students and professionals.

- **Sample Size:** A total of $N = 27$ playtesters were recruited.
- **Demographics & Expertise:** Data regarding age, professional role, and play frequency were collected. While the cohort included a diverse range of disciplines (e.g., Programmers, 2D Artists, Sound Designers), the majority were industry practitioners, ensuring a high baseline of gaming literacy. A minority of non-industry players were also included to broaden the perspective.
- **Inclusion Criteria:** Participation was voluntary and uncompensated.

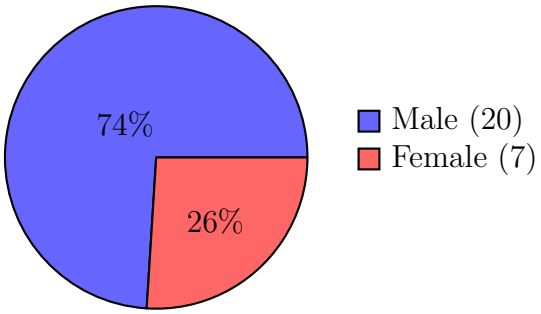

Figure 4.1: Gender Distribution ($N = 27$)

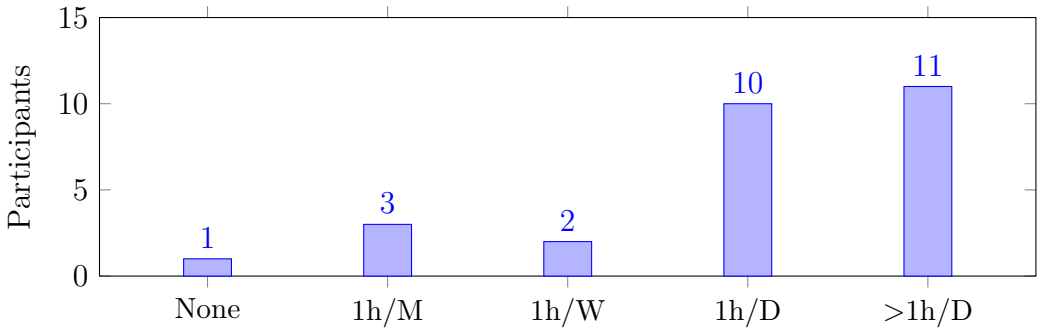

Figure 4.2: Self-reported play frequency

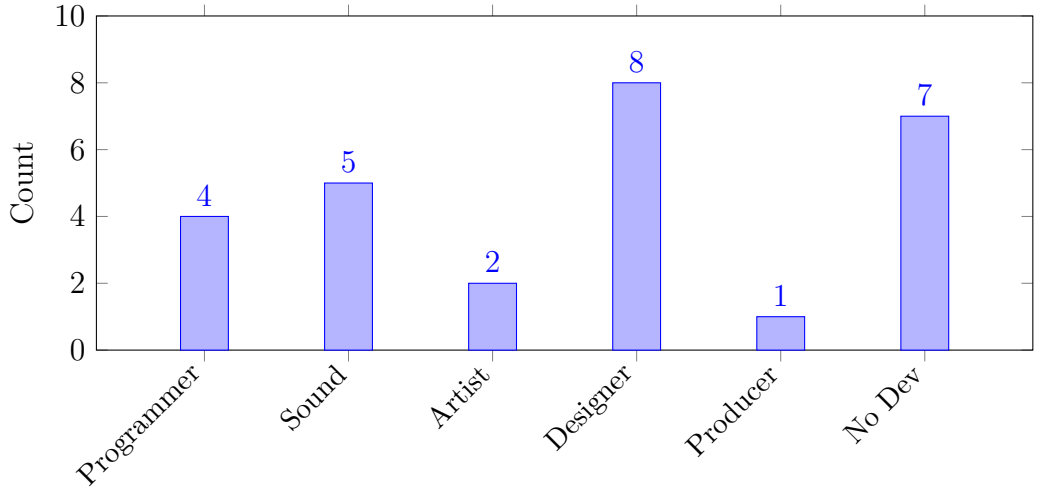

Figure 4.3: Professional role distribution

4.1.2 Experimental Protocol

To mitigate cognitive fatigue and maintain data quality, the experimental scope was restricted. While three prototypes were developed (Phase 2), the user study utilized only two: *Celeste Classic* and *Mind The Gap*. The third title, *Sacrifices Must Be Made*, was excluded from the live trials to keep the total session duration within a sustainable **30-minute window**.

The experimental protocol followed a fixed sequence for all playtesters, but the presentation order of the game versions was deliberately alternated. Specifically, the presentation sequence (Original vs. AI) was inverted between the two game blocks: participants played the Original version first for *Celeste Classic*, but encountered the AI-generated version first for *Mind The Gap*. This alternating strategy was implemented to prevent "Predictability Bias," ensuring participants could not assume a pattern (e.g., "the first version I play is always the AI one") that might skew their perception during the Blind Turing Test.

1. **Introduction:** Playtesters received a briefing on the study's scope (without revealing the generation source of specific versions), signed informed consent forms, and completed the demographic questionnaire.

2. **Block 1 (*Celeste Classic*):**

- **Instruction:** Basic tutorial on controls and objectives.
- **Session 1A (Original):** Participant plays the original game first (first three stages).
- **Assessment 1A:** Completion of reduced GUESS-18 and the Blind Turing Test.
- **Session 1B (AI-Generated):** Participant plays the entire short vibe-coded prototype.
- **Assessment 1B:** Completion of reduced GUESS-18 and the Blind Turing Test.
- **Creative Evaluation:** Rating of LLM-proposed design improvements.

3. **Block 2 (*Mind The Gap*):**

- **Instruction:** Basic tutorial on controls and objectives.
- **Session 2A (AI-Generated):** Participant plays the vibe-coded prototype first (first two "weeks" in-game).
- **Assessment 2A:** Completion of reduced GUESS-18 and the Blind Turing Test.
- **Session 2B (Original):** Participant plays the original prototype (first two "weeks" in-game).
- **Assessment 2B:** Completion of reduced GUESS-18 and the Blind Turing Test.
- **Creative Evaluation:** Rating of LLM-proposed design improvements.

4.2 Evaluation Metrics

A multi-component instrument focusing on subjective player experience, perceived authorship, and design viability was utilized to evaluate the prototypes. The protocol consisted of three distinct assessment layers. (See Appendix B for the complete text of the administered instrument).

4.2.1 Subjective Experience Metrics

To assess the quality of the player experience, the **GUESS-18** (Game User Experience Satisfaction Scale) [65] was utilized. To ensure the evaluation precisely targeted the mechanical and functional viability of the prototypes, only five of the original nine subscales were deliberately deployed. The five subscales strictly relevant to evaluating mechanical proofs-of-concept were retained, resulting in a streamlined 10-item instrument:

- **Usability/Playability:** Measuring control responsiveness and interface clarity.
- **Enjoyment:** Measuring the hedonistic value of the core loop.
- **Personal Gratification:** Measuring the sense of achievement and challenge.
- **Visual Aesthetics:** Measuring the acceptance of the "programmer art" style.
- **Creative Freedom:** Measuring the player's sense of agency within the mechanics.

Dimensions evaluating Narratives, Play Engrossment, Audio Aesthetics, and Social Connectivity were purposefully excluded to prevent construct-irrelevant noise, as these features fell entirely outside the defined scope of the established Minimum Viable Products. This targeted approach allowed for a highly focused assessment of the core gameplay loops without penalizing the prototypes for lacking secondary commercial polish.

4.2.2 Perceived Authorship (The "Blind Turing Test")

To quantify the perceptual distinction between human-tuned physics and AI-generated logic, a Blind Turing Test was administered immediately following each gameplay session.

Playtesters were asked a single binary-choice question: *"Do you believe the game you just played was designed by a human or generated by an AI?"* This metric serves as a proxy for **Mechanical Authenticity**. A high rate of "Human" attribution for an AI-generated prototype indicates that the LLM successfully replicated not

just the rules, but the subtle "micro-interactions" (e.g., coyote time, variable jump height) that characterize a polished game, effectively passing a functional Turing test for game design.

4.2.3 Generative Design Acceptance

To address RQ3 (LLM Creative Contribution), a metric for **Generative Design Acceptance** was established. During the development phase, the LLM was prompted to suggest novel mechanical improvements for each game (e.g., "Add a wind current mechanic to *Celeste Classic*").

In the post-experiment phase, playtesters were presented with these textual descriptions—decoupled from the implementation—and asked to vote (Yes/No) on whether they would be favorable to include these mechanics in the core game. This measures the **Design Viability** of the AI's creative output, distinguishing between hallucinated/irrelevant ideas and genuinely value-additive design concepts.

4.3 Data Analysis Plan

To rigorously evaluate the data collected from the development logs and the player questionnaires, a mixed-methods analysis approach was adopted. This approach combined hard statistical testing to evaluate prototype quality and efficiency, alongside empirical observations to capture the nuances of the vibe coding workflow.

4.3.1 Quantitative Analysis

The quantitative analysis focused on objective performance metrics and standardized user ratings. The following statistical methods were employed:

- **Descriptive Statistics:** Means, standard deviations, and rates were calculated to establish baselines for development time, iteration counts, and raw GUESS-18 scores. Additionally, descriptive statistics were computed on the researcher-assigned scores scores to assess perceived improvement between the first and final iterations of each prototype.

- **Categorical & Aggregated Metrics:** Binary responses (Yes/No) regarding the LLMs' creative contributions were aggregated to calculate the overall approval and preference rates for each model. Similarly, responses from the Blind Turing Test were aggregated to calculate the "Deception Rate" (i.e., the percentage of playtesters who misidentified a game's authorship).

- **Paired Comparisons:** A Wilcoxon Signed-Rank Test was utilized to compare the ordinal data from the GUESS-18 questionnaires between the paired groups (Original vs. AI-generated versions of the same game). This test was critical in determining whether the observed "Quality Gap" was statistically significant.

- **Independent Group Comparisons:** A Mann-Whitney U Test was employed to analyze differences between independent user demographics and to perform pairwise comparisons of efficiency metrics between specific LLMs.

- **Multiple Group Comparisons:** A Kruskal-Wallis H Test was used to detect overarching variances in performance and efficiency among the three tested LLMs.

- **Effect Size Calculation:** To determine not just whether a difference existed, but the magnitude of that difference, the Rank-Biserial Correlation was calculated for the non-parametric tests.

4.3.2 Qualitative Analysis

Because vibe coding relies heavily on natural language interaction, purely quantitative metrics cannot capture the full picture of the development experience. Therefore, the analysis was supplemented with qualitative empirical and observational evaluations, derived from continuous observation across the 1,000+ generated iterations. This heuristic evaluation was designed to systematically identify emerging patterns in human-AI interaction, document recurring workflow bottlenecks, and categorize the various conversational strategies employed during the iterative prompting process.

4.4 Hypotheses

To address the established Research Questions and provide a clear framework for the empirical and statistical analyses, the following specific hypotheses were formulated. These hypotheses are categorized according to the primary research vectors they investigate:

Relating to Feasibility and Prototype Quality (RQ1):

- **H1 (MVP Generation):** Large Language Models can successfully generate Minimum Viable Prototypes (MVPs) from natural language prompts without manual code intervention.
- **H2 (Sufficient Quality):** AI-generated prototypes will achieve baseline sufficiency in standardized user experience evaluations (e.g., the GUESS-18 framework).
- **H3 (Comparability):** There will be no statistically significant difference in the user experience ratings between AI-generated MVPs and their human-crafted counterparts.
- **H4 (Indistinguishability):** Players will not be able to reliably distinguish between human-crafted and AI-generated games at a rate significantly higher than random chance during blind testing.

Relating to Efficiency and Tools (RQ2):

- **H5 (Model Variance):** There are significant performance and efficiency variances between different state-of-the-art LLMs when tasked with rapid game generation.
- **H6 (Prompting Hierarchy):** The specific choice of prompting technique significantly impacts generation success and stability, establishing a measurable hierarchy of effectiveness among different strategies.
- **H7 (Efficiency Gain):** The vibe coding workflow provides a measurable acceleration in development velocity compared to traditional manual coding baselines.

Relating to Creative Agency (RQ3):

- **H8 (Creative Contribution):** LLMs can effectively act as creative partners by proposing novel game mechanics and design iterations that playtesters actively approve and adopt.

Chapter 5

Results and Discussion

This section presents the findings of the experimental evaluation, organized systematically to address the three primary Research Questions. The quantitative and qualitative data collected from the generated prototypes and the playtest participants provide a comprehensive view of the feasibility, efficiency, and creative potential of vibe coding in game pre-production.

5.1 Overview of Results

The experimental evaluation yielded robust data confirming the viability of LLM-based vibe coding as a rapid prototyping methodology, while also highlighting distinct limitations. A high-level summary of the main findings reveals the following:

- **Feasibility and Quality (RQ1):** Vibe coding proved highly capable of generating Minimum Viable Prototypes (MVPs), achieving a 96% success rate within 15 iterations. However, while AI-generated games achieved "sufficient" baseline scores in Usability, Enjoyment, and Gratification, they did not reach statistical equivalence with human-crafted originals, confirming a persistent "Quality Gap". Interestingly, the Blind Turing Test revealed a notable exception: while playtesters generally spotted AI-generated games (70.4% recognition rate), purely logic-based prototypes (e.g., *Mind the Gap*) achieved a remarkable 40.7% deception rate, approaching indistinguishability.

- **Efficiency and Tools (RQ2):** The methodology demonstrated a dramatic acceleration in development time, yielding a ∼40x velocity multiplier. The average prototype required only 8.3 iterations and 88.1 minutes to reach a playable state. Among the tested models, Gemini 2.5 Pro emerged as the most efficient and reliable "engine" for code generation, especially when paired with structured prompting techniques like the Fullerton framework.

- **Creative Agency (RQ3):** LLMs demonstrated significant capability as design collaborators, with 66% of playtesters approving and adopting their novel mechanical suggestions. Crucially, an inversion of model capabilities was observed: while GPT-4.1 lagged in raw coding efficiency (RQ2), it overwhelmingly dominated as a "creative partner" (74.1% preference rate) compared to Gemini (51.9%).

These high-level findings culminate in the concept of "Cheap Fun": a paradigm where developers accept lower initial fidelity in exchange for unprecedented iterative speed and creative augmentation. The detailed statistical breakdown supporting these conclusions is presented in the following subsections.

5.2 RQ1: Feasibility and Prototype Quality

Research Question 1: Is it possible to create working digital prototypes through vibe coding without touching a line of code, but simply through natural language instructions? Are these prototypes comparable to human-crafted ones?

To comprehensively answer this question, the analysis was broken down into three distinct areas: the technical feasibility of generating Minimum Viable Prototypes (MVPs), the evaluation of user experience to identify any potential "Quality Gap", and a Blind Turing Test to measure the distinguishability of AI-generated mechanics.

5.2.1 Technical Feasibility and MVP Generation (H1)

The fundamental premise of vibe coding relies on the LLM's ability to output functional software based purely on natural language. To test **H1 (MVP Generation)**, the "success rate"—defined as the percentage of prompt chains that successfully yielded a Minimum Viable Prototype—was tracked across 126 generation attempts. In this context, an MVP was rigorously defined as a prototype that passed a three-layer validation process: functional verification of atomic mechanics, a comparative "vibe" analysis for phenomenological alignment with the reference game, and baseline runtime stability using placeholder constraints.

The data robustly confirms H1, but highlights the necessity of human-guided iteration. Out of 126 attempts, the methodology achieved an overall **96.0% success rate** (121/126) within the maximum limit of 15 iterations.

Rather than immediate zero-shot successes, analyzing the distribution reveals a clear cumulative developmental curve (as illustrated in Figure 5.1). Only 18.0% of attempts reached MVP status in 5 or fewer iterations. This rate accelerated significantly to 75.4% by the 10-iteration mark, before finally converging at 96.0% by the 15-iteration threshold.

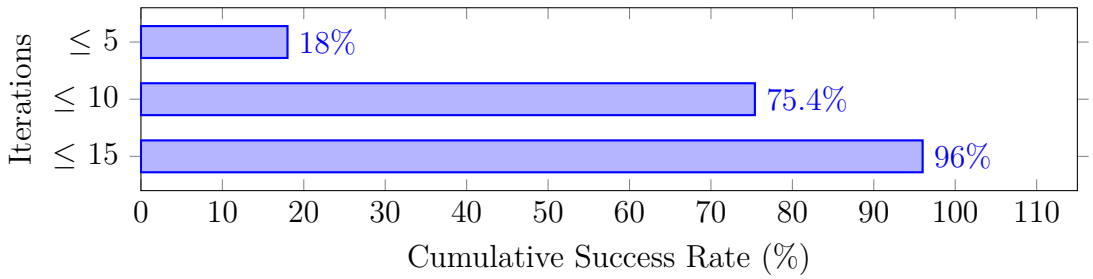

Figure 5.1: Cumulative success rate of playable MVPs across key iteration thresholds.

These convergence tiers suggest two critical operational insights for vibe coding. First, "one-shot" generation is rarely sufficient; achieving a functional and phenomenologically aligned MVP requires a baseline of conversational friction and iterative refinement. This is empirically evidenced by the dataset: no prototype achieved MVP validation on the first attempt, requiring a minimum of two iterations even in the most optimal scenarios. Second, the 15-iteration mark serves as a

strong convergence boundary. If a prototype has not reached a playable MVP state by this threshold, the context window is likely too degraded or the foundational logic too entangled, indicating that the developer should abandon the current thread and start over rather than attempting to debug further.

Answer to H1: The hypothesis is strongly supported. Large Language Models are highly capable of generating playable Minimum Viable Prototypes purely from natural language prompts without manual coding intervention. However, technical feasibility is heavily dependent on a structured, iterative conversational process rather than immediate, zero-shot generation.

5.2.2 User Experience and the "Quality Gap" (H2 & H3)

While generating a functional MVP is a significant technical milestone (H1), RQ1 also interrogates the qualitative value of these generated experiences. To evaluate this, 27 playtest participants assessed the prototypes using the standardized GUESS-18 framework, scoring the games across five core axes: Usability, Enjoyment, Creative Freedom, Gratification, and Visual Aesthetics.

Evaluating Baseline Sufficiency (H2)

To test **H2 (Sufficient Quality)**, the raw scores of the AI-generated prototypes were analyzed to determine if they provided a functional, baseline user experience. Responses were measured on a 1-to-7 Likert scale, establishing a score of 4.0 as the threshold for "sufficiency".

The data partially supports H2, with important caveats. On an individual axis level, the AI-generated MVPs successfully achieved sufficiency in Usability (4.29), Enjoyment (4.12), and Gratification (4.91), indicating that the core mechanics and gameplay loops were fundamentally engaging. However, the overall cumulative mean for AI games fell just short of the sufficiency threshold at 3.92. This was heavily dragged down by severe underperformance in Visual Aesthetics (2.94) and Creative Freedom (3.36), reflecting the limitations of placeholder graphics and rigid, generated code structures. Figure 5.2 shows the overall averages of the considered games.

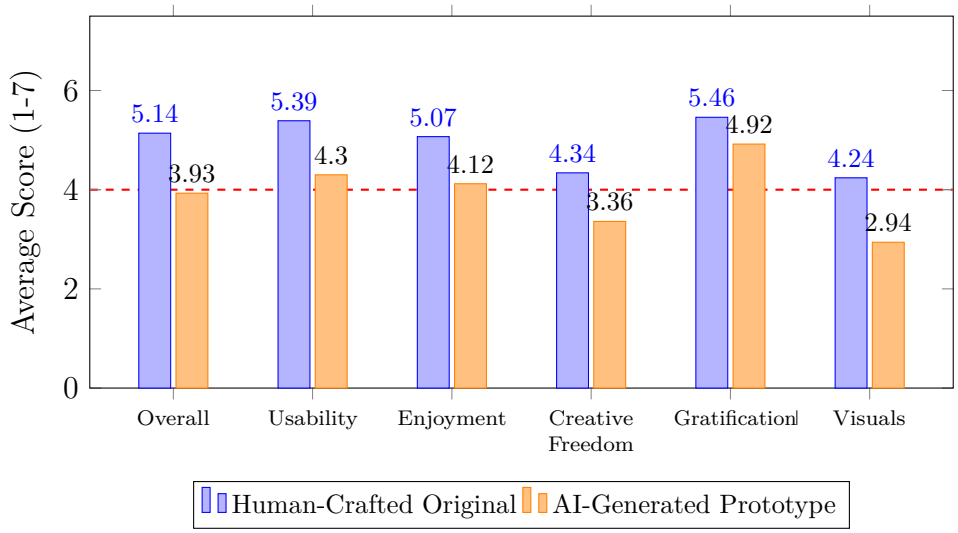

Figure 5.2: Cumulative GUESS-18 evaluation scores (Overall). The dashed red line represents the baseline sufficiency threshold (4.0).

This pattern remains remarkably consistent when isolating the evaluation by genre. As shown in Figure 5.3 (*Celeste Classic*) and Figure 5.4 (*Mind the Gap*), the AI prototypes generally follow the same contours of success and failure.

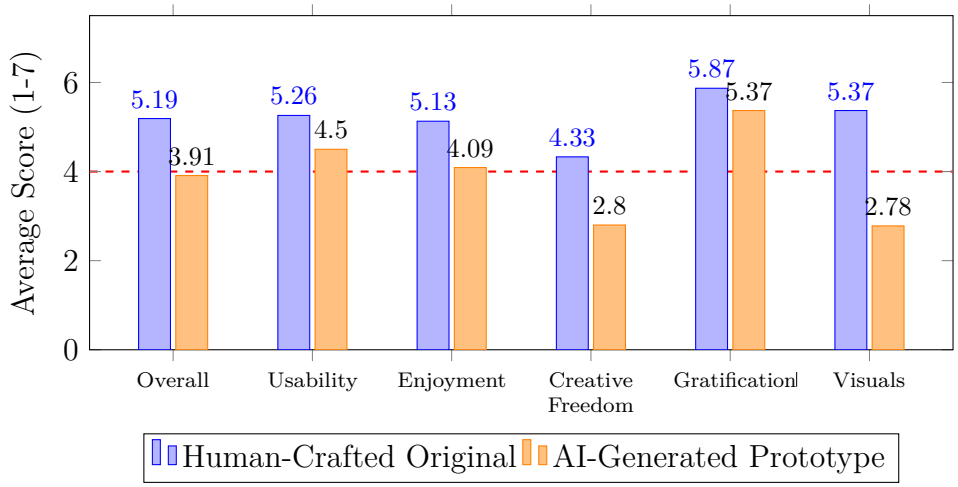

Figure 5.3: Cumulative GUESS-18 evaluation scores for *Celeste Classic*. The dashed red line represents the baseline sufficiency threshold (4.0).

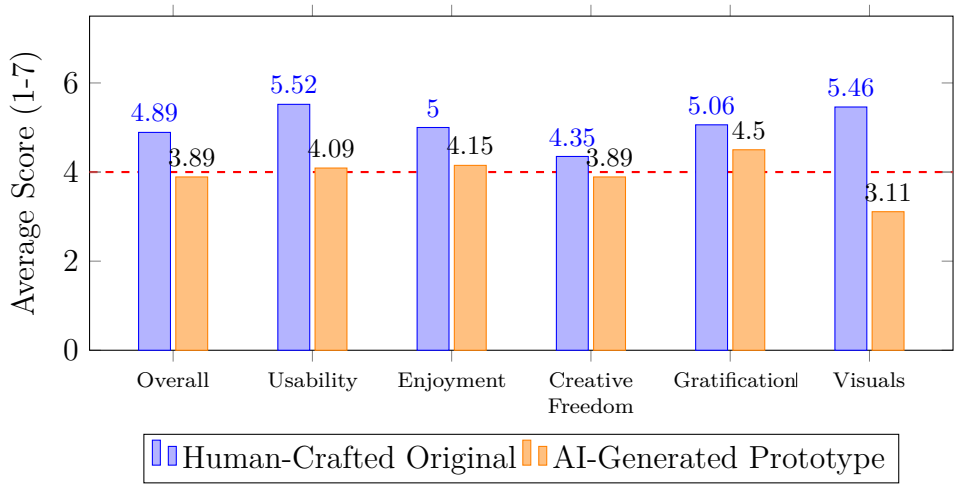

Figure 5.4: Cumulative GUESS-18 evaluation scores for *Mind the Gap*. The dashed red line represents the baseline sufficiency threshold (4.0).

When holistically evaluating the prototypes across all genres, approximately 90% of the human-crafted original games achieved an average sufficiency score (≥ 4.0). By contrast, the AI-generated prototypes achieved this sufficiency benchmark in approximately 50% of cases. Interestingly, despite the statistical dominance of human-crafted games, a minority of playtesters actively preferred the AI-generated experiences over the originals: 18.5% of playtesters (5 out of 27) favored the AI version of *Celeste Classic*, and 14.8% (4 out of 27) preferred the AI version of *Mind the Gap*.

Testing for Comparability (H3)

However, establishing baseline sufficiency does not imply parity with human developers. To test **H3 (Comparability)**, a Wilcoxon Signed-Rank Test was conducted on the paired ordinal data to compare the player evaluations of the human-crafted reference games against their AI-generated counterparts. To understand the magnitude of these differences, the Rank-Biserial Correlation (r_{rb}) was calculated as a measure of effect size, where a value closer to 1.0 indicates a more profound divergence between the two versions.

The overall statistical analysis (detailed in Table 5.1) comprehensively rejects equivalence. Every single p-value falls below the 0.05 threshold of significance,

with the vast majority falling below 0.001. Therefore, **H3 is firmly rejected**; no AI-generated prototype achieved statistical comparability with its human-authored reference.

However, the Rank-Biserial Correlation reveals crucial nuances about the nature of this "Quality Gap". The data indicates that the gap is not uniform across the player experience:

- **The Widest Gaps:** The most profound differences consistently appear in Visual Aesthetics ($r_{rb} = 0.906$) and the Overall impression ($r_{rb} = 0.847$). This confirms that the AI's reliance on placeholder graphics and rigid layouts is the most recognizable deficiency to players.
- **The Narrowest Gaps:** Conversely, the AI approaches closer parity in Enjoyment ($r_{rb} = 0.698$) and Gratification ($r_{rb} = 0.746$). While still statistically distinct, these lower effect sizes suggest the AI is much better at capturing the intangible "fun factor" of a game than it is at matching human-level visual polish.

Table 5.1: Overall Wilcoxon Signed-Rank Test Results (Original vs. AI-Generated). $p < 0.05$ indicates a significant difference; Rank-Biserial Correlation (r_{rb}) indicates effect size.

GUESS-18 Axis	p**-value**	**Correlation** (r_{rb})
Overall	< 0.001	0.847
Usability	< 0.001	0.796
Enjoyment	0.002	0.698
Creative Freedom	< 0.001	0.793
Gratification	0.002	0.746
Visual Aesthetics	< 0.001	0.906

This overarching trend remains remarkably consistent when analyzing the specific game genres. As shown in Table 5.2 (*Celeste Classic*) and Table 5.3 (*Mind the Gap*), the AI games are definitively outperformed across all axes ($p < 0.05$).

However, isolating the genres highlights where specific mechanical frameworks help or hinder the AI. Most notably, the puzzle-driven *Mind the Gap* showed a

significantly narrower gap in Creative Freedom ($r_{rb} = 0.574, p = 0.029$) compared to the action-driven *Celeste Classic* ($r_{rb} = 0.843, p < 0.001$). This suggests that the AI struggles less with design constraints when generating highly structured, logic-based games compared to open, physics-heavy platformers.

Table 5.2: Wilcoxon Results for *Celeste Classic* (Original vs. AI-Generated).

GUESS-18 Axis	***p*-value**	**Correlation (r_{rb})**
Overall	< 0.001	0.852
Usability	0.004	0.663
Enjoyment	0.006	0.672
Creative Freedom	< 0.001	0.843
Gratification	0.011	0.663
Visual Aesthetics	< 0.001	0.906

Table 5.3: Wilcoxon Results for *Mind the Gap* (Original vs. AI-Generated).

GUESS-18 Axis	***p*-value**	**Correlation (r_{rb})**
Overall	< 0.001	0.831
Usability	< 0.001	0.800
Enjoyment	0.004	0.738
Creative Freedom	0.029	0.574
Gratification	0.012	0.628
Visual Aesthetics	< 0.001	0.914

Answers to H2 and H3: The data presents a clear duality in vibe coding capabilities. **H2 is partially supported**: LLMs can reliably generate gameplay loops of sufficient Usability and Enjoyment, achieving overall sufficiency in roughly half of all attempts, though they are heavily constrained by aesthetic limitations. Conversely, **H3 is rejected**: these generated prototypes are not statistically comparable to human-crafted games. However, effect size metrics reveal that while the aesthetic gap remains massive, the AI is remarkably adept at narrowing the gap in intangible player enjoyment. This dynamic formally establishes the "Quality Gap" while highlighting the core value proposition of AI prototyping.

5.2.3 Authorship Recognition and the Blind Turing Test (H4)

The final metric for evaluating prototype quality was a Blind Turing Test designed to address **H4 (Indistinguishability)**. Playtesters were asked to guess whether the prototype they just played was crafted by a human or generated by an AI, testing if players could truly distinguish the "vibe" from the manual craft.

Overall Recognition Rates and False Positives

When holistically evaluating the entire dataset, playtesters demonstrated a general capacity to identify the true authorship of the prototypes, making the correct guess in 75.9% of all cases. However, the fact that nearly one-quarter of guesses (24.1%) were incorrect indicates a persistent degree of confusion rather than absolute clarity.

When playing AI-generated prototypes specifically, playtesters correctly identified the AI's authorship 70.4% of the time, resulting in an overall "Deception Rate" of 29.6%. Because players could distinguish the games at a rate significantly higher than random chance, **H4 is generally rejected**. Figure 5.5 illustrates this overall perception.

Notably, the overarching data also reveals a consistent false positive rate regarding human work: across all human-crafted original games, 18.5% of playtesters mistakenly guessed the game was made by AI. This indicates that human authorship is not always self-evident to the player, and a minority will misattribute human development to generative AI.

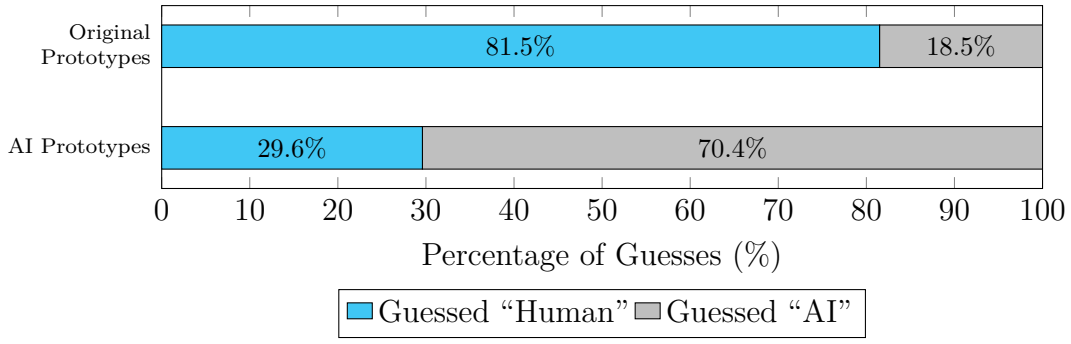

Figure 5.5: Overall Blind Turing Test results representing perceived authorship.

The Genre Exception

Visualizing the results across specific genres (Figure 5.6) reveals how different mechanical frameworks influence this detection. While visual fidelity and physics remained a dead giveaway in action games (the *Celeste Classic* AI prototype only deceived 18.5% of players), the logic-based *Mind the Gap* AI prototype achieved a remarkable **40.7% deception rate**. In highly structured, system-driven genres, the AI's output approaches effective indistinguishability.

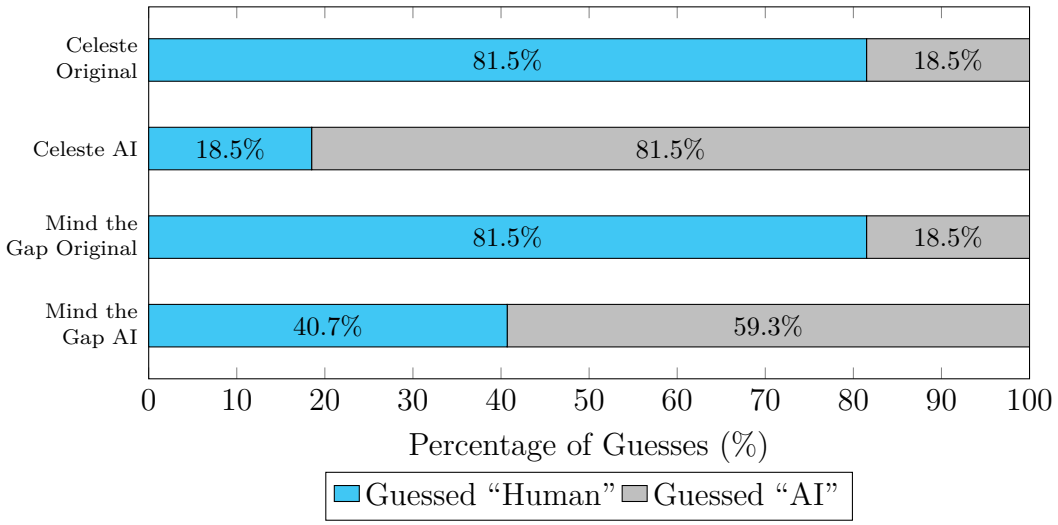

Figure 5.6: Blind Turing Test results separated by game genre.

The "Holistic Tell" vs. Aesthetic Bias

A common assumption in evaluating these results is that players simply recognized the AI due to its reliance on basic placeholder graphics. To mathematically test this "Aesthetic Bias", a Mann-Whitney U test was conducted to determine if there was a statistically significant difference in the Visual Aesthetics scores given by players who guessed "Human" versus those who guessed "AI".

Crucially, the test revealed no statistically significant correlation. When evaluating the AI-generated games, players who assigned low Visual scores were not significantly more likely to guess the game was AI-generated (*Celeste AI Visuals*: $U = 72.00, p = 0.294$; *Mind the Gap AI Visuals*: $U = 109.50, p = 0.296$). Because $p > 0.05$ across the board, the data proves that poor graphics alone do not trigger

AI detection. Instead, players sense the AI's authorship holistically—through the pacing, the friction of the mechanics, and the overall phenomenological "vibe" of the gameplay loop.

Answer to H4: The hypothesis is rejected, though with notable caveats. While players can reliably distinguish between human-crafted and AI-generated games (achieving a 75.9% overall accuracy rate), a minor degree of confusion persists, evidenced by an 18.5% false positive rate when identifying original games. Furthermore, a significant exception exists for purely logic-based, system-driven prototypes, where AI generation can achieve a deception rate (40.7%) that approaches practical indistinguishability.

5.2.4 Answer to RQ1

In conclusion of RQ1, vibe coding is highly feasible for rapid digital prototyping, reliably generating functional MVPs through iterative natural language prompting with a 96.0% success rate within 15 iterations. While these AI-generated prototypes achieve baseline sufficiency in Usability, Enjoyment, and Gratification—and reach general overall sufficiency in approximately half of all attempts—they are not yet statistically comparable to human-crafted games. This confirms a persistent "Quality Gap" driven largely by limitations in visual aesthetics and creative freedom. Furthermore, while players can reliably identify AI authorship through a holistic assessment of the gameplay experience (achieving a 75.9% overall recognition rate), this boundary blurs significantly when prototyping purely logic-based puzzle games.

5.3 RQ2: Tool Effectiveness Comparison

Research Question 2: Which LLMs and prompting techniques among the currently used ones are best suited for rapid game prototyping tasks, and what is the quantitative gain in performance?

To comprehensively answer this question, the prototyping process was analyzed across three distinct dimensions: the variance in capability between different LLM models (H5), the hierarchy of effectiveness among various prompting strategies (H6), and the ultimate quantitative gain in development velocity and "fun" generation

(H7).

5.3.1 Model Variance and Reliability (H5)

To test **H5 (Model Variance)**, three state-of-the-art Large Language Models were evaluated: GPT-4.1 (gpt-4.1-2025-04-14), Gemini 2.5 Pro (gemini-2.5-pro), and Claude Sonnet 4 (claude-sonnet-4-20250514). The models were assessed based on three specific vectors: iteration speed, self-assigned prototype quality, and technical stability.

Iteration Speed and Time Estimation

Across all successful prototype generations, reaching a functional Minimum Viable Prototype required a strictly measured average of 8.3 iterations. Figure 5.7 illustrates the general distribution of these iterations across the entire dataset, highlighting a bell-like curve centering around the 6-to-11 iteration mark.

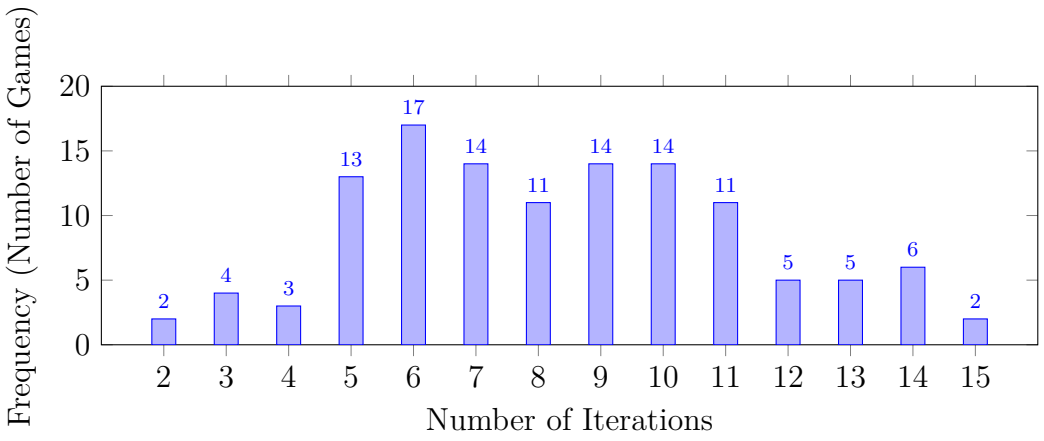

Figure 5.7: Overall distribution of iterations required to reach a functional prototype state.

To translate this raw iteration count into a tangible metric of development velocity, an empirical time baseline was established based on continuous observation of the workflow during the study. The generation process was modeled using an estimated constant of 30 minutes for the initial conceptualization and prompt engineering phase, plus an estimated 7 minutes for each subsequent evaluation,

debugging, and re-prompting cycle. This yields the following formula to calculate the Average Time-to-Prototype:

$$\text{Estimated Time (minutes)} = 30 + (7 \times \#\text{ of Iterations}) \tag{5.1}$$

Applying this formula, the 8.3 average iterations translates to an estimated development time of 88.1 minutes per game.

Iteration Speed by Model

However, a significant variance in efficiency emerged when isolating this data by the specific LLM employed. As shown in Figure 5.8, each model exhibits a distinct distribution curve, heavily impacting their respective average generation times.

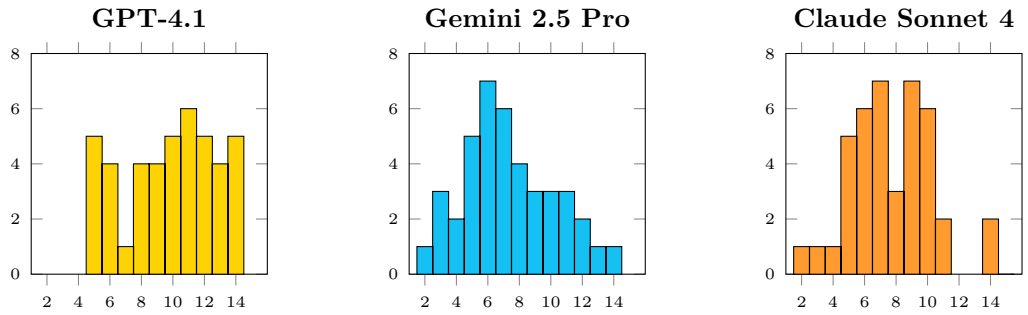

Figure 5.8: Distribution of iterations required, separated by LLM model.

The descriptive statistics (detailed in Table 5.4) reveal that Gemini 2.5 Pro required the fewest iterations on average (7.34), achieving the fastest estimated time-to-prototype at roughly 81.4 minutes. Claude Sonnet 4 followed closely behind with a mean of 7.75 iterations ($\approx$ 84.3 minutes), while GPT-4.1 trailed significantly, requiring an average of nearly 10 iterations ($\approx$ 99.6 minutes) to reach the same functional baseline.

Table 5.4: Descriptive statistics for prototype iterations and estimated generation times across models.

Metric	GPT-4.1	Gemini 2.5 Pro	Claude Sonnet 4
Valid Attempts	39 / 42	41 / 42	41 / 42
Mean Iterations	9.949	7.341	7.756
Std. Deviation	2.819	2.912	2.727
Avg. Time	$\approx$ 99.6 min	$\approx$ 81.4 min	$\approx$ 84.3 min

Testing for Equivalence

To determine if these observed differences in iteration speed were statistically significant, a Kruskal-Wallis test was conducted on the overall dataset. The test confirmed a highly significant variance across the three models ($p < 0.001$).

To further interrogate the specific hierarchy of these tools, pairwise Mann-Whitney U tests were deployed. The statistical analysis confirmed that GPT is definitely slower than its competitors, with both Gemini ($p < 0.001, r_{rb} = 0.472$) and Claude ($p = 0.001, r_{rb} = 0.431$) significantly outperforming it. However, when comparing Claude against Gemini, the Mann-Whitney test yielded a p-value of 0.557 ($r_{rb} = -0.091$). Because this value sits well above the 0.05 threshold, it can be mathematically concluded that Claude and Gemini are statistically equivalent regarding iteration speed.

Prototype Quality (Self-Assigned Scores)

To evaluate the qualitative output of the models transparently, the games were graded using the researcher-assigned quality scores during the generation process. This evaluation was split into two phases: the initial capability (1st Iteration Score) and the finalized output (Last Iteration Score).

When evaluating the quality of the prototype after a single prompt, the models achieved broad statistical equivalence. A Kruskal-Wallis test on the 1st Iteration Scores yielded a p-value of 0.201, indicating no significant variance. Despite this mathematical equivalence, Gemini led slightly (mean 49.34), Claude was close behind (47.76), and GPT trailed slightly (46.79).

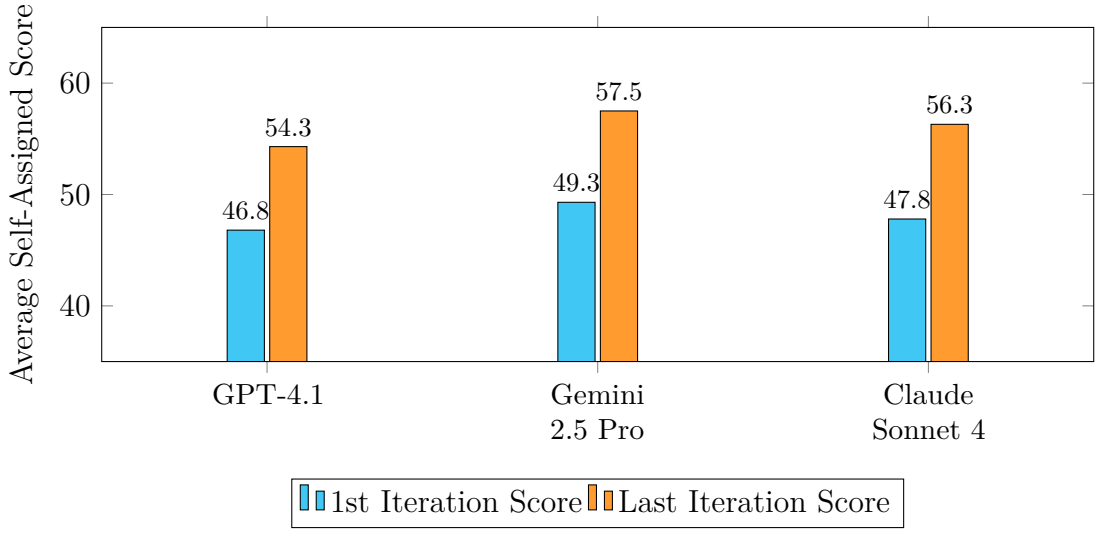

Figure 5.9: Comparison of initial and final self-assigned prototype quality scores across LLMs.

However, by the final iteration, this hierarchy had solidified, and the model variance re-emerged as statistically significant (Kruskal-Wallis, $p = 0.016$).

To interrogate this final variance, pairwise Mann-Whitney U tests were conducted. Gemini consolidated its lead as the most consistently capable engine (final score 57.46) and statistically outperformed GPT ($p = 0.004, r_{rb} = -0.21$). Claude occupied the middle ground (56.32), effectively acting as a bridge between the models, remaining statistically equivalent to both Gemini ($p = 0.198$) and GPT ($p = 0.110$).

Interestingly, an analysis of the distribution extremes (visible in the Standard Deviation and Maximum scores in Table 5.5) reveals a critical nuance regarding GPT-4.1. While GPT averaged the lowest final scores overall and possessed the lowest statistical reliability, it exhibited high-variance unpredictability. The four absolute highest-rated individual games in the entire study (scoring 64) were all generated by GPT. Thus, while Gemini is the most reliable engine for consistent baseline quality, GPT remains a high-risk, high-reward alternative capable of generating superior outliers.

Table 5.5: Descriptive statistics for self-assigned quality scores (1st and Last Iterations).

Metric	GPT-4.1	Gemini 2.5 Pro	Claude Sonnet 4
1st Iteration Scores			
Mean Score	46.79	49.34	47.76
Std. Deviation	7.804	5.620	5.791
Minimum Score	30	32	32
Maximum Score	62	60	58
Last Iteration Scores			
Mean Score	54.28	57.46	56.32
Std. Deviation	5.876	3.668	3.791
Minimum Score	39	49	46
Maximum Score	64	63	63

Additional Considerations: Technical Stability and Sound

While initial scores suggested statistical equivalence, technical stability at the first iteration revealed severe reliability issues across all models. Overall, only 57.8% of games were technically functional on the first attempt. GPT proved to be the least reliable, producing "broken" prototypes on the first iteration over half the time (only 48.7% valid). Claude achieved 60.9% validity, while Gemini was the most technically stable at 63.4%.

Furthermore, sound implementation proved universally problematic. On the first iteration, only 48.7% of prototypes successfully generated adequate audio. However, this issue was largely resolved through the iterative workflow, with 79.3% of prototypes achieving functional sound by their final iteration.

Answer to H5: The hypothesis is fully supported. There is a statistically significant variance in both performance and efficiency across the tested models. Gemini 2.5 Pro proved to be the most efficient and reliable engine, requiring the fewest iterations and demonstrating the highest initial technical stability. Claude Sonnet 4 performed firmly in the upper tier alongside Gemini, achieving statistical equivalence in rapid iteration speed and closely matching its initial technical reliability. Conversely, GPT-4.1 operated as a high-variance engine—requiring

significantly more iterations and struggling with initial stability, yet ultimately producing some of the highest-rated individual prototypes by the final iteration.

5.3.2 Prompting Hierarchy (H6)

Beyond the underlying model, the specific methodology used by the human operator drastically impacted the outcome. Testing **H6 (Prompting Hierarchy)** revealed a clear stratification of effective techniques.

Note: Because multiple prompting techniques have been frequently combined (e.g., using a Fullerton framework alongside Few-Shot examples), the following metrics reflect the isolated marginal impact of each variable across the dataset.

- **Structured vs. Unstructured:** The data definitively proves that structured prompts outperform unstructured (Naïve) approaches across all metrics. Unstructured prompts averaged more iterations (8.97) and significantly lower final scores (52.67). By contrast, structured approaches required fewer iterations (8.04) and yielded higher final quality (57.48).
- **Frameworks:** When isolating the specific structured architectures (detailed in Table 5.6), the "Fullerton" framework delivered the best overall quality, achieving the highest initial (51.07) and final (59.00) scores. The "Schell" framework converged slightly faster (7.84 iterations) but peaked slightly lower. Attempting to combine them into a "Hybrid" approach proved counterproductive, increasing iterations (8.60) while lowering final scores compared to pure Fullerton.
- **Context and Shots:** Providing the LLM with context examples drastically improved performance. Few-shot techniques delivered the best overall results (7.50 average iterations; 58.60 final score). One-shot approaches came close in quality (57.21) but were slower (8.47 iterations), while Zero-shot methods were the least effective (53.71 final score).
- **Temperature and Parameters:** Utilizing a standard/normal LLM temperature yielded vastly superior results (56.74 final score) compared to low-temperature settings, which produced significantly slower (9.47 iterations) and unsatisfactory outputs (51.82 final score).

- **Advanced Techniques:** Utilizing System Prompts and 2-Step Generation processes slightly increased the overall time-to-completion (averaging 8.42 and 8.60 iterations, respectively), but consistently delivered higher quality final results (57.06 and 58.00) than single-prompt baselines.

Table 5.6: Performance Comparison of Core Prompting Frameworks.

Metric	Naïve	Schell	Fullerton	Hybrid
Avg. Iterations	8.97	7.84	8.00	8.60
1st Shot Score	42.06	50.78	51.07	49.35
Last Shot Score	52.67	57.58	59.00	56.10

Answer to H6: The hypothesis is fully supported. There is a measurable hierarchy of effectiveness among prompting techniques. The optimal methodology consists of a structured "Fullerton" framework, augmented by Few-Shot examples, standard temperature settings, and multi-step system prompting. Conversely, unstructured, Zero-Shot, and low-temperature prompts actively degrade generation success and stability.

5.3.3 Efficiency Gain and "Cheap Fun" (H7)

A primary advantage of vibe coding lies in its rapid generation velocity. To test **H7 (Efficiency Gain)**, the generation time of AI prototypes was compared against traditional human development metrics. This comparison assumed a constraint of 12 man-hours per developer, per day, reflecting the frenetic and highly intensive nature of game jam events.

Time Gain and Velocity Multipliers

The baseline average for generating a playable AI game was calculated at exactly 88.1 minutes per game (incorporating an average of 30 minutes for initial prompt writing and 7 minutes per subsequent iteration, across an average of 8.3 iterations).

When comparing these per-game AI metrics to estimated human development times, the acceleration is profound. Generalized across the 88.1-minute average, vibe

coding operates with an estimated 49x Velocity Multiplier compared to traditional manual coding. Looking strictly at the specific test cases evaluated in the playtest, the AI demonstrated a 43x Velocity Multiplier overall. Table 5.7 details these temporal differences, separating the data by genre and including the cumulative man-hours required to generate the evaluated games.

Table 5.7: Performance and Efficiency gains comparing AI Generation against traditional Man-Hours.

Metric	Overall	Celeste Classic	Mind the Gap
Human Man-Hours	144	96	48
AI Man-Hours	3.3	1.2	2.1
Velocity Multiplier	43x	80x	22x

Generating "Cheap Fun"

To contextualize this extreme speed within the actual player experience, a "Fun Efficiency" metric was derived by dividing the prototype's GUESS-18 score by the time required to build it. Holistically, the LLM creates "fun" at an approximate 38x multiplier compared to traditional manual coding.

As established in RQ1, AI-generated games face a persistent Quality Gap and struggle to match the peak fidelity of human-crafted experiences. However, the data from RQ2 demonstrates that the AI generates baseline engagement at a massively accelerated rate.

This dynamic introduces the quantitative concept of "Cheap Fun". Because the LLM can rapidly output functional mechanics, it produces baseline gameplay experiences at an industrial scale. While there is a qualitative ceiling to the engagement an LLM can generate, this high-velocity production allows designers to rapidly validate core loops, test mechanical viability, and "fail faster" without the sunk costs of manual coding.

Answer to H7: The hypothesis is fully supported. Vibe coding provides a massive, measurable acceleration in development velocity compared to traditional manual coding baselines. Demonstrating an average 49x Velocity Multiplier, an experimentally measured Velocity Multiplier of 43x, and a 38x Fun Score Multiplier,

the methodology validates itself as a highly efficient engine for generating rapid, baseline prototype engagement, even if it cannot yet replicate the peak qualitative fidelity of human-crafted games.

5.3.4 Qualitative Tool Assessment and Empirical Best Practices

While the quantitative data presented in the sections relative to H5, H6, and H7 establishes a clear hierarchy of models and frameworks, the experimental process also revealed critical empirical considerations regarding human-AI collaboration. The raw capability of an LLM is heavily bound by the operator's approach to problem-solving.

Three primary qualitative best practices emerged during testing:

- **The Learning Effect:** The most effective overarching technique is simply to prompt frequently and learn. Engaging in high-frequency iteration and adapting to how the model interprets specific vocabulary yields far better results than attempting to engineer a flawless zero-shot prompt.

- **Domain Knowledge as a Catalyst:** While vibe coding theoretically allows non-programmers to generate code, having foundational knowledge in the relevant field proves incredibly helpful. For example, explicitly knowing which pathfinding algorithm to suggest can significantly improve the output, as it frees the language model from certain logical challenges and reduces the cognitive load of the prompt.

- **Iterative "Think From Scratch" Strategies:** When a model becomes trapped in a loop of logical errors or broken code, employing an iterative strategy can be highly beneficial. Specifically, utilizing a "think from scratch" approach encourages a thorough exploration of ideas before settling on a final response, forcing the model to re-evaluate its logic and break out of hallucination loops.

5.3.5 Answer to RQ2

In response to RQ2, the experimental data identifies a clear and optimal path for vibe coding in game prototyping. Gemini 2.5 Pro emerges as the most effective overall engine, offering a superior balance of iteration velocity and technical stability. Claude Sonnet 4 occupies a highly reliable upper-tier position alongside it, while GPT-4.1, despite having lower performance reliability, remains a high-variance alternative capable of producing higher qualitative peaks. Furthermore, the methodology proves as critical as the model: the application of Structured Prompting (specifically, Few-Shot approaches utilizing the Fullerton Framework) yields statistically significant improvements in code quality over naïve approaches. Ultimately, vibe coding demonstrates up to a $\sim$40x Velocity Multiplier compared to traditional development. While the resulting prototypes possess a persistent fidelity gap, the ability to generate "Cheap Fun" at such extreme speeds validates the methodology as a transformative pre-production tool, effectively shifting the developer's role from manual implementation to high-level curation.

5.4 RQ3: LLM Creative Contribution

Research Question 3: Is an LLM capable of and effective in proposing novel game mechanic iterations that improve upon the initially proposed version?

To answer RQ3, the study evaluated **H8 (Creative Partnership)**, measuring whether playtesters actively approved and adopted the novel mechanical variations and design iterations proposed by the LLMs during the prototyping process.

5.4.1 Overall Appreciation and Adoption Rates

The data confirms that LLMs are highly effective at proposing novel game mechanics that improve upon a user's initial design. During the evaluation phase, playtesters were presented with various AI-generated features and mechanical variations, strictly judging them on their potential to enhance the gameplay experience. Under these conditions, participants voluntarily adopted the AI's suggestions in 66% of cases. The fact that exactly two out of three playtesters appreciated these ideas firmly

validates the LLM's role as a competent source of design innovation. By successfully generating mechanics that human evaluators deemed additive and engaging, the AI demonstrates its utility beyond mere technical execution. It proves that within the rapid prototyping pipeline, the LLM functions not just as a passive code generator, but as a viable co-designer capable of resolving creative bottlenecks.

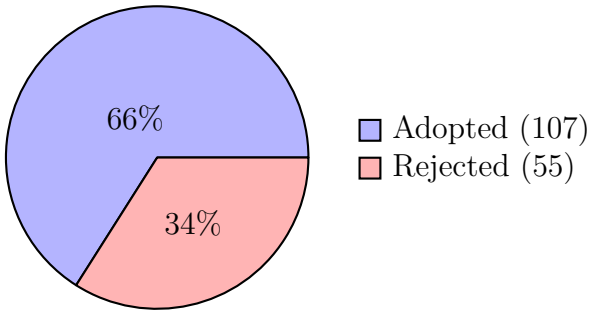

Figure 5.10: Overall playtester adoption rate of LLM-suggested game mechanics.

5.4.2 The Creative Leaderboard: Model Specialization

Crucially, breaking down these adoption rates by the specific AI model reveals a "Creative Leaderboard" that entirely flips the results established in the previous Research Question.

While Gemini 2.5 Pro was previously identified as the most reliable engine for technical execution, it scored the lowest in creative contribution, with playtesters adopting its ideas only 51.9% of the time. Conversely, GPT-4.1 proved to be the superior creative partner. Its design suggestions achieved a remarkable 74.1% approval rate, significantly outperforming Gemini. Claude Sonnet 4 once again occupied a strong middle ground, achieving a 68.5% approval rate for its creative input.

This data identifies a distinct and highly valuable specialization within the vibe coding workflow. While GPT-4.1 may exhibit high variance and lower reliability when generating the actual game code, its ideas are highly valued by playtesters. It shines primarily as a creative partner in game design tasks.

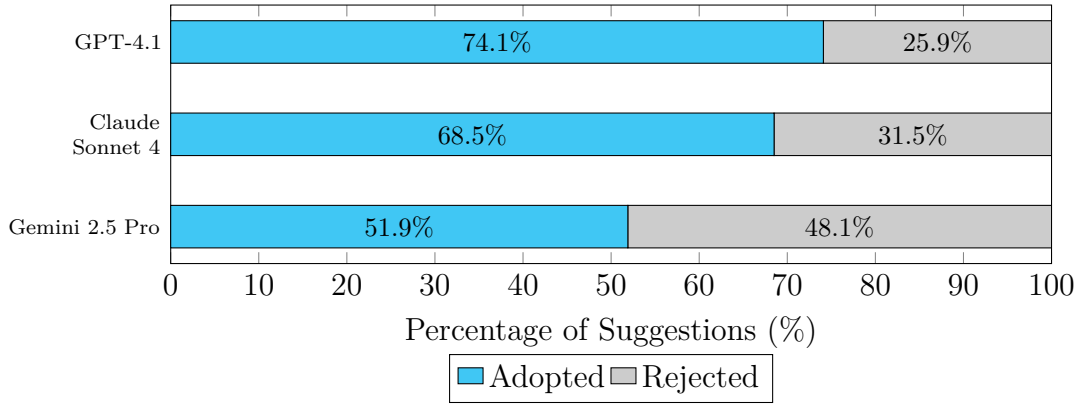

Figure 5.11: Creative approval and rejection rates separated by the proposing LLM.

5.4.3 Answer to RQ3

H8 is fully supported. In answer to RQ3, LLMs are highly capable of and effective in proposing novel game mechanic iterations, with playtesters actively adopting AI-suggested improvements in 66% of cases. Furthermore, the data reveals a powerful hybrid workflow opportunity for developers: different models handle different stages of the production pipeline with varying degrees of success. While Gemini is the preferred "engine" for reliable code execution, GPT-4.1 serves as the optimal companion for creative ideation and design partnership.

5.5 Game-Specific Qualitative Findings

While the quantitative data establishes broad trends in performance and efficiency, examining the generation process across the three distinct prototype genres reveals crucial qualitative nuances regarding how Large Language Models handle different types of game design logic.

Action/Platformer Games (*Celeste Classic*)

In the action-platformer genre, the LLMs demonstrated a stark dichotomy between mechanical execution and spatial reasoning. Core player mechanics, such as jump arcs and movement physics, were often well-implemented with minimal prompting.

However, the models consistently struggled with spatial level design. Autonomous attempts to generate platforming levels frequently resulted in structurally unsound, uncompletable, or physically disconnected environments (see Figure 5.12). While utilizing One-Shot and Few-Shot prompting techniques with existing level arrays mitigated this issue by giving the AI a structural template, it did not fundamentally solve the LLM's lack of true spatial and architectural awareness.

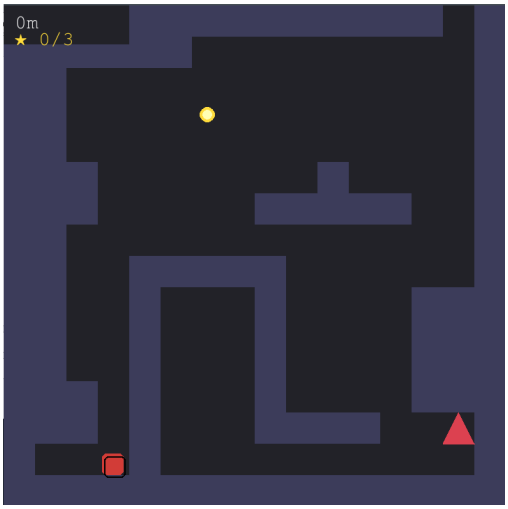

Figure 5.12: *Celeste Classic* generated by Gemini 2.5 Pro.

Real-Time Simulation (*Mind the Gap*)

For the real-time simulation prototype, the primary technical hurdle was developing a functional passenger routing system. When prompted with generalized instructions (e.g., "make sure the passengers can make transfers to reach their destination"), the models struggled to produce optimized or functional movement logic. However, when the human operator explicitly stated the necessity of a specific pathfinding algorithm (such as A* or Dijkstra's), the generation quality improved drastically, often resolving the issue within a single iteration (Figure 5.13). This underscores a critical reality of vibe coding: domain expertise remains fundamental. Generating a complex game is not as simple as asking an AI to "make it work"; the operator must possess enough technical vocabulary to know exactly *what* architectural solutions to request.

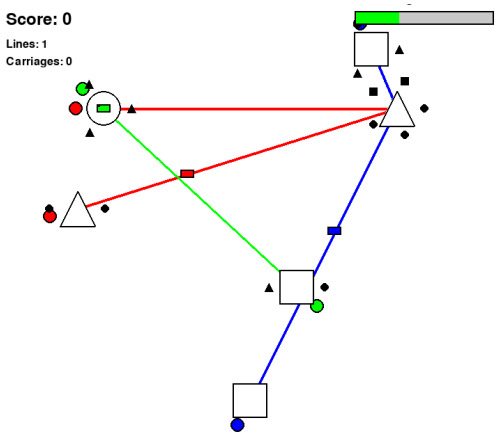

Figure 5.13: *Mind the Gap* generated by Gemini 2.5 Pro.

Rule-Based Mechanics (*Sacrifices Must Be Made*)

Conversely, the digital card game prototype proved to be the easiest and most seamless to produce (Figure 5.14). Because this genre relies inherently on discrete state-machines, mathematical variable tracking, and strict turn-based rules rather than continuous spatial physics or complex real-time system simulation, the abstract nature of its gameplay translates natively into the sequential, code-driven logic that LLMs excel at generating. Once the core game flow, the deck mechanics, and the conditional rules of the cards were explicitly defined in the prompt, the generation of these consequential, interconnected systems occurred naturally. As a result, this prototype required significantly less iterative debugging and architectural troubleshooting than its action or puzzle counterparts.

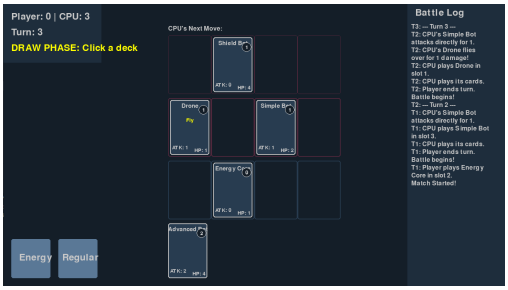

Figure 5.14: *Sacrifices Must Be Made* generated by Gemini 2.5 Pro.

5.6 Synthesis and Implications

5.6.1 Vibe Coding as a Prototyping Methodology

Based on the experimental data and qualitative observations, vibe coding proves to be a highly viable methodology for the "quick and dirty" prototyping of isolated game mechanics, rather than a standalone tool for full-scale game development. It excels at accelerating brainstorming sessions and delivering immediate, playable proofs-of-concept. However, the methodology is not currently suited for generating complete games, nor does it produce the level of polish required to showcase a vertical slice to a potential investor. The optimal workflow implies a strict hand-off: vibe coding should be used exclusively to rapidly validate a core mechanical loop, after which the project should transition to a primarily human-led development cycle for the actual production phase.

The methodology's efficiency is heavily dictated by genre and mechanical scope. Vibe coding shines brightest when applied to static, rule-based systems (such as card games or discrete state-machines), where the game logic translates seamlessly into the sequential text processing of an LLM. Conversely, as a prototype introduces complex continuous mechanisms—such as physics interactions or real-time simulations—the language models remain capable, but the generation process becomes significantly more complex and requires rigorous human steering. Most notably, the methodology cannot currently be relied upon for intricate creative-technical tasks, such as spatial reasoning or abstract level design, which still fundamentally require a human architectural understanding to remain functional.

5.6.2 Comparison to Traditional Prototyping

When contextualized within broader game development practices, the most striking difference between vibe coding and human-led prototyping is the extreme trade-off between velocity and final polish. As established by the efficiency metrics, vibe coding offers an immediate and massive acceleration in development time. However, within the scope of rapid prototyping, this speed is accompanied by observable fidelity constraints. While LLMs excel at generating functional baseline engagement—the aforementioned "Cheap Fun"—they consistently struggle to

autonomously replicate the nuanced game feel, aesthetic cohesion, and bespoke friction that characterize a meticulously human-crafted experience.

Furthermore, this paradigm shift fundamentally alters the skill requirements of the prototyping phase. On one level, vibe coding democratizes the pipeline: anyone can benefit from it, allowing non-programmers—such as 2D artists, writers, or producers—to rapidly instantiate and test their ideas without writing a single line of code. However, the methodology disproportionately amplifies the capabilities of existing experts. Programmers benefit immensely because they possess the technical vocabulary to explicitly direct the AI's architecture, bypassing its problem-solving limitations. Similarly, experienced game designers can leverage precise industry terminology to accurately translate abstract concepts into functional mechanics.

Ultimately, the decision of when to employ each approach is dictated by the project's current phase. Vibe coding is the superior choice for the absolute beginning of a project's lifecycle, where the goal is to "fail faster", test mechanical viability, and explore a breadth of ideas. Human-led prototyping becomes strictly necessary once a core mechanic is validated and the focus shifts to deepening the player experience, refining spatial design, and polishing the game for public or commercial presentation.

5.6.3 The Role of LLMs in Creative Design

In evaluating the implications of RQ3, the experimental data support the conceptualization of LLMs as highly valid, albeit specialized, creative partners. While the study does not assert that AI can entirely substitute human game designers, it definitively proves their efficacy as collaborative agents within a brainstorming ecosystem. Rather than replacing human inventiveness, the LLM functions as a supplementary team member—acting as a catalyst to sustain creative momentum and serving as a rapid sounding board for iterating on ideas. The fact that playtesters voluntarily adopted AI-suggested mechanical variations in 66% of cases indicates that LLMs are capable of proposing genuinely useful iterations during the prototyping phase.

Furthermore, this dynamic hints at a shifting workflow for human developers. The interaction is inherently iterative: the designer inputs a constraint, the AI proposes a mechanical solution, and the human curates, rejects, or refines that

output. Ultimately, while it remains to be seen whether the industry will eventually produce commercially successful games entirely generated by AI, the data confirms the immediate utility of LLMs as collaborative tools that actively contribute to human-led design.

Crucially, the "Creative Leaderboard" established that not all LLMs possess the same capabilities in terms of creative ideation, with different models exhibiting distinct inclinations. This research demonstrates that a mixed-tool pipeline is the ideal approach for vibe coding: leveraging GPT-4.1 as the optimal companion for game design and brainstorming, while deploying Gemini as the primary engine for reliable code realization and execution.

Chapter 6

Conclusion

This research investigated "vibe coding" as a methodology for the rapid digital prototyping of game mechanics. Through the systematic recreation of three distinct game jam prototypes and a structured evaluation of human-AI collaboration, this study sought to quantify the capabilities, limitations, and creative potential of Large Language Models within the pre-production pipeline in the context of a rapid prototyping methodology.

6.1 Summary of Contributions

This research demonstrates that vibe coding serves as a transformative tool for the initial stages of game design. While AI-generated prototypes cannot currently replace human-led development, they represent a powerful accelerator for pre-production and a valid collaborative partner for maintaining creative momentum. By systematically addressing the core Research Questions, this study establishes the following primary conclusions:

- **RQ1 (Feasibility and Quality):** While LLMs are highly capable of generating functional game prototypes, these AI-authored games are not flawless, nor are they statistically comparable to polished, human-crafted titles. This distinction was further evidenced by a Blind Turing Test: while players can generally identify AI authorship by sensing the holistic "vibe" and mechanical

friction of action-oriented games, purely logic-driven AI prototypes can approach true indistinguishability. Ultimately, despite this general quality gap, vibe coding represents a highly viable and effective tool for validating core mechanics and gameplay loops during the pre-production phase.

- **RQ2 (Efficiency and Tools):** Generative workflows offer a highly significant boost to developer productivity and prototyping velocity. The study demonstrates that this efficiency is heavily dependent on the chosen model and instructional strategy. At present, utilizing structured prompting frameworks—among several other valid prompting techniques that yielded positive results—in tandem with models like Gemini 2.5 Pro provides the most reliable and efficient engine for technical execution.

- **RQ3 (Creative Agency):** LLMs serve as highly effective creative partners in the design process. Playtesters generally validated and approved the novel mechanics suggested by the models. Furthermore, the data reveals that different LLMs possess distinctly different creative capabilities; just as Gemini excels in code generation, GPT-4.1 currently stands out as the superior model for ideation and creative design tasks, strongly advocating for a multi-model approach.

Synthesizing these findings, this research makes four distinct contributions to game development research and practice:

1. **Empirical Evaluation of the Vibe Coding Methodology:** This research establishes a methodological baseline that can be utilized and built upon in future studies to test newer models and more complex genres. Foundational metrics are provided for evaluating the capabilities of LLMs applied to rapid game prototyping.

2. **Conceptualization of "Cheap Fun":** Based on empirical evidence, the paradigm of "Cheap Fun" is formalized to concisely define the current landscape of AI game prototyping—characterized by the high-velocity production of functional, baseline engagement that is ultimately bounded by observable constraints in final polish.

3. **The Prompting Hierarchy:** A data-backed framework for optimal AI interaction is established, proving that structured, domain-specific prompting drastically outperforms unstructured approaches in both generation speed and code stability.

4. **The Mixed-Tool Pipeline:** It is identified that the future of AI-assisted game development relies on model specialization. The best results are achieved by utilizing different LLMs for their specific strengths (e.g., GPT-4.1 for design and ideation; Gemini 2.5 Pro for logic and execution) rather than relying on a single monolithic system.

6.2 Practical Implications

Based on empirical observations, distinct sets of practical guidelines are proposed, tailored to both individual developers executing the code and studio leads managing the broader production pipeline.

6.2.1 Guidelines for Game Designers and Developers

For practitioners seeking to rapidly improve their generation results and optimize their day-to-day workflow, the following micro-level strategies are recommended:

- **Choice of Tools and the 15-Iteration Rule:** To maximize the benefits of rapid prototyping, developers must actively test and alternate between different AI models. While this research identifies a generally optimal mixed-tool pipeline, every game project presents unique logical hurdles, and testing remains fundamental to assess which tool best suits a specific developmental need. Furthermore, based on empirical observations, adhering to a strict "15-iteration rule" is recommended. If a functional baseline for a core mechanic is not achieved within 15 iterations, the generation is statistically unlikely to converge. Rather than wasting time attempting to brute-force a broken prototype, developers should simply start over—wiping the context window, changing the model, or rewriting the initial prompt from scratch.

- **Prioritize Structured Prompting:** The methodology used to communicate with the AI is as important as the AI itself. While techniques like few-shot examples, 2-step generation, and role prompting are valuable, utilizing structured prompting techniques yields the most consistent baseline improvements. When combined with human domain expertise, the operator's ability to steer the model is drastically enhanced.
- **Diagnostic and Architectural Specificity:** When iterating on a generated prototype, developers must abandon vague feedback in favor of hyper-specific diagnostic instructions across three axes:
  - *Debugging:* Instead of stating "it doesn't work, fix it," explicitly state the environment and error (e.g., "I received a NullReferenceException on line 232 when the player collides with the wall").
  - *Game Feel:* Instead of subjective complaints (e.g., "the jump is horrible"), use mechanical descriptors (e.g., "the jump is too low, lacks power, and feels sticky on departure; I need a higher, snappier movement with higher gravity on the descent").
  - *Logic Implementation:* Instead of broad goals (e.g., "make enemies chase the player"), dictate the architecture (e.g., "implement an A* pathfinding algorithm for the enemies with a 0.5-second calculation delay to simulate a relaxed chase style").

6.2.2 Guidelines for Studio Leads and Production

For directors and studios looking to integrate AI into their operational pipelines, the findings suggest the following macro-level best practices:

- **Enforce a Strict "Hand-Off" Workflow:** Vibe coding should be leveraged exclusively to "fail faster" during early pre-production. Leadership should encourage teams to use AI to rapidly test mechanical viability and discard bad ideas without incurring the sunk costs of human engineering hours. However, once a core loop is validated, development must strictly transition to human-led engineering to achieve the necessary visual polish and nuanced game feel that AI currently cannot provide.

- **Empower Cross-Disciplinary Prototyping:** Because vibe coding lowers the technical barrier to entry, studios no longer need to bottleneck their engineering departments during the ideation phase. Leadership should empower narrative designers, 2D/3D artists, and producers to independently generate and test their own "Cheap Fun" proofs-of-concept. This democratizes the brainstorming process and saves expensive senior programmer hours for actual production tasks.
- **Shift Training from Syntax to Systems Architecture:** Generating complex games with AI is not as simple as asking a model to "make a game". The human operator must possess the technical vocabulary to guide the AI. Therefore, studio training and hiring practices should pivot: rather than strictly evaluating junior designers on low-level coding syntax (which the AI can write), studios should train them in high-level computer science concepts, systems architecture, and algorithmic logic so they can effectively act as algorithmic directors.

6.3 Theoretical Implications

Beyond immediate practical application, this study highlights several broader implications for the future of game design:

- **The Shift from Implementation to Curation:** Vibe coding fundamentally alters the developer's role during prototyping. As AI assumes the burden of manual code implementation, the human designer elevates to the role of a Creative Director—managing algorithmic tools, curating outputs, and enforcing a cohesive architectural vision.
- **The Nature of AI Creativity:** This research contextualizes the LLM not as an autonomous artist, but as a catalyst for human creativity. It functions as an untiring sounding board that prevents "blank page syndrome". However, human "taste" remains the critical factor in curating these outputs and producing genuinely novel experiences.
- **Democratization vs. Amplification:** While vibe coding lowers the barrier

to entry—allowing non-programmers to instantiate playable concepts—it disproportionately amplifies the output of existing experts. The tools democratize the *attempt* at game creation, but they heavily reward those who already understand the deep structural language of game design.

- **Risk Mitigation and Creative Freedom:** As LLM-driven rapid prototyping methodologies mature, they offer the industry a mechanism to test unconventional mechanics and concepts with a significantly higher degree of freedom. By drastically reducing the sunk costs of time and budget typically associated with early development, studios can afford to explore riskier, more innovative ideas, ultimately paving the way for more creative and experimental game design.

Chapter 7

Limitations and Future Work

7.1 Limitations

7.1.1 Methodological Limitations

- **Sample Size of Evaluated Games:** The experimental scope was restricted to the recreation of exactly three game jam prototypes. While chosen to represent distinct genres, this small sample size cannot represent the entirety of game design paradigms.

- **Temporal LLM Dependencies:** This research captures a specific snapshot in time. The capabilities of Large Language Models are evolving at an unprecedented rate; the performance metrics recorded for the analyzed models may become obsolete as newer, more robust models and architectures are deployed.

- **Systematization of Iterative Techniques:** While this research systematically evaluated initial prompt construction and overall generation performance, the specific techniques used during the iterative refinement phase were primarily observed empirically. Past the initial generation, there was no systematic quantitative coding of the exact follow-up prompting strategies (e.g., how to

optimally communicate a bug fix vs. a design change). Gathering structured data on these micro-interactions would provide deeper insights into human-AI communication workflows.

7.1.2 Scope Limitations

- **Prototype-Level Complexity:** The evaluated games were strictly constrained to the prototype phase, focusing exclusively on core mechanical loops. The study did not test the LLMs' ability to generate or maintain the complex architectures required for full-scale production games, such as save-state systems, deep UI hierarchies, or memory optimization.
- **Absence of 3D and Spatial Computing:** The methodologies were tested entirely within 2D environments. The current study does not account for the significantly higher mathematical and spatial complexities involved in 3D game engines.
- **Engine Specificity (Pygame Only):** In all prompts, the LLMs were explicitly instructed to generate games utilizing the Python-based Pygame library. While this successfully demonstrated the models' reasoning and coding capabilities within this specific framework, generation has not been tested across other popular industry engines (e.g., Unity, Godot) or languages (e.g., C#, C++), which possess different architectural rules and might yield varied results based on the models' specific training data.

7.1.3 Evaluation Limitations

- **Subjective Evaluation Bias:** During the iterative generation process, the evaluation of the prototype's functional success, code quality, and iteration convergence relied entirely on internal assessment. While standardized prompting frameworks and strict testing criteria were employed to mitigate this, the absence of a broader evaluation panel inherently introduces a degree of subjective bias that external, independent code reviews could have minimized.
- **Short-Term Play Sessions:** The user evaluation relied on the GUESS-18 framework, which excellently measures immediate qualities of the playtested

games. However, because the playtests were inherently short, the study lacks longitudinal data regarding long-term player retention and whether the AI's core loops remain engaging over extended periods.

- **Evaluation Metric Breadth:** Although the gathered data provides a solid foundational picture of the current landscape, incorporating a wider diversity of metrics would yield deeper insights. Expanding both quantitative tracking (e.g., in-game telemetry, more evaluation axes in playtests) and qualitative methods (e.g., conducting in-depth, open-ended post-playtest interviews) would help capture more nuanced dimensions of the player experience.

- **Sample Size Constraints:** The quantitative findings derived from the user experience surveys are inherently limited by the number of participants, meaning broader generalizations regarding global player preferences should be approached with caution.

7.2 Future Research Directions

The findings of this study establish a foundational baseline for vibe coding, opening several critical avenues for future academic and industry research.

7.2.1 Expanding Scope

- **Complex Genres and Mechanics:** While this study evaluated action, simulation, and card mechanics, future research should test the methodology against significantly more complex genres (e.g., deep RPGs, physics-based simulations, or grand strategy games) to determine how the "Quality Gap" scales as systemic complexity increases.

- **3D and Spatial Computing:** Future studies must test the limits of LLM generation within 3D physics engines and complex spatial environments to determine if the models' current lack of architectural awareness persists outside of 2D constraints.

7.2.2 Methodological Extensions

- **A/B Comparative Team Studies:** To definitively quantify the trade-offs of the methodology, future controlled experiments should compare two distinct development teams—one utilizing traditional manual programming and the other utilizing vibe coding—tasked with building the exact same game specification. This would provide direct comparative metrics on development time, code maintainability, and final subjective quality.

- **AI-Assisted Hybrid Pipelines:** Moving beyond pure "zero-code" vibe coding, future research should evaluate realistic human-AI cooperation. Studies should measure workflows where an AI generates foundational artifacts, but experienced human programmers actively iterate, refactor, and modify the codebase, assessing how this collaborative pipeline impacts overall project stability and developer satisfaction.

- **Natural Language Engine Control via MCP:** Future research should investigate the integration of the Model Context Protocol (MCP) to bridge the gap between Large Language Models and industry-standard game engines, such as Unity or Unreal Engine. By securely exposing engine-specific APIs and editor functions through MCP, developers could theoretically control these complex environments entirely through natural language. Evaluating an AI's ability to autonomously manipulate 3D scenes, configure components, and manage assets directly within the editor would represent a massive paradigm shift from the current conversational copy-paste methodology.

- **Agentic IDE Integration and Debugging:** Alongside engine control, future tools should explore true agentic development workflows. Research should evaluate frameworks where the AI has direct, autonomous access to a project's local environment—allowing it to independently read project hierarchies, monitor real-time error logs, and execute terminal commands without requiring the human operator to manually transcribe errors.

7.2.3 Creative and Educational Applications

- **Game Design Education:** Vibe coding presents a transformative opportunity for academia. Future research should explore using LLMs in introductory game design courses, allowing students to learn systems architecture and game flow without being entirely blocked by syntax errors.
- **Interdisciplinary Democratization:** Further studies should focus on how narrative designers, audio engineers, and visual artists can leverage these tools to build interactive portfolios without relying on dedicated engineering teams.

7.3 Closing Thoughts

As Large Language Models continue to advance, vibe coding represents more than just a transient productivity hack—it fundamentally reimagines the relationship between a designer's intent and technical implementation. While the findings reveal a persistent "Quality Gap" and strict boundaries regarding spatial reasoning, the methodology's ability to mass-produce "Cheap Fun" is undeniable. By removing the immediate friction of syntax, vibe coding allows developers to test riskier ideas, fail faster, and iterate at an unprecedented velocity.

However, "Cheap Fun" for rapid generation is a double-edged sword. While it acts as a powerful catalyst for exploration, high-volume output without rigorous human curation quickly flattens into shallow, derivative experiences. This is precisely why vibe coding is currently most valuable when utilized primarily as a pre-production tool. While it is impossible to predict the ultimate capabilities of future AI architectures, today's models are best understood as instruments that empower capable crafters rather than replace them. Ultimately, the trajectory of this research suggests that natural language interfaces will become a critical, democratizing pillar of the game development pipeline—permanently elevating the designer from a mechanical implementer to a creative curator.

Appendix A

Example Prompts

This appendix details the foundational system prompts and specific framework structures utilized during the generation process to guide the Large Language Models.

A.1 System Prompt

Listing A.1: Base System Prompt Example

```
You are a Python game developer who specializes in creating 2D games using the
    Pygame library. Your job is to generate clean, readable, and runnable
    Pygame code that brings the users prototype ideas to life. The goal is to
    provide a working prototype that allows a game designer to quickly test and
     iterate on their ideas, not to produce a full game.

We'll follow a mandatory 2-steps generation pipeline, where:
- After player gives you the game design brief, you stop and ask for further
    details and instructions. Do not ask for parameter values as they're very
    specific on the implementation that will be generated and cannot possibly
    be known by the game designer (use reasonable placeholders that will be
    tweaked in the following iterations)
- When every doubt is clear, you can proceed to iterate over game code

Follow these principles:
- Write complete, self-contained, runnable scripts. Include imports,
    initialization, main loop, and a clean exit.
```

```
- Each time code is requested, always rewrite the full script rather than applying only incremental changes.
- Keep the code simple and modular. Use object-oriented programming only when necessary.
- Also game should be easily modifiable by game designers using proper parameters at the beginning of the code.
- Comment the code for clarity. Explain game logic, key sections, and any non-obvious behavior.
- Always double-check for consistency (especially in level design)
- Use simple placeholder graphics like rectangles, circles, or text. Do not rely on external assets, dependencies, or configuration files unless explicitly provided.
- Ask for clarification if the user request is ambiguous or underspecified.
```

A.2 Schell Prompt

Listing A.2: Schell Framework Prompt Example

```
Please design this game using the following brief

Game Design Brief
1. Mechanics  What are the rules and actions?
- Player actions:
        - Move left and right
        - Jump
        - Wall-jump (jump while touching a wall to jump off it)
        - After a wall-jump, apply a strong horizontal velocity away from the wall and temporarily disable horizontal input toward that wall to prevent reattaching or reversing direction mid-jump.
        - Player has to press jump button for each wall-jump (no automatic wall-jump on button pressed)
        - Dash
            - Dash can be done in 8 directions (e.g. if player dashes while Up+Left arrow are pressed, player will dash in the up-left direction)
            - Dashing is recharged only when player touches the ground (not after a wall-jump)
- Core gameplay loop: Navigate challenging platforming levels by jumping, wall-jumping, and dashing to reach the summit.
```

```
- Objectives / Win condition: Reach next level by going up above the level
  border. Complete 3 levels in these way. After 3 levels player sees a
  victory screen.
- Failure / Lose conditions: Falling into spikes or pits causes the player to
  respawn at the beginning of a level.
- Challenge & Progression: Increasingly difficult platforming challenges
  requiring precise timing and movement combinations and a variety of
  techniques (e.g. wall-jump, dash).
- Environment:
      - Single-screen platforming levels with solid platforms and walls in
  large blocks
      - Player cannot go past over lateral borders of a level, nor wall-jump
  on them
      - Spikes that kill the player on contact
      - Collectibles for optional challenge/scoring
- Resources:
      - Dash charges: Player can dash once in midair, recharges when touching
   ground (not walls)
      - Collectibles: Optional collectibles for score/completion percentage
- Entities:
      - Player: player character
      - Collectibles: collectible items
      - Spikes: hazardous terrain
- Level Design:
    - Levels should be defined using a simple matrix-based ASCII layout,
    editable by the game designer directly. Each level is a list of strings or
    characters, where each character represents a tile or entity.
    - We'll reference at the matrix with 0-based coordinates, where row 0 is
    the highest row and column 0 is the leftmost column
    - Generated levels should be as diverse as possible, trying to force the
    player to use the wider possible range of abilities (e.g. wall-jumps,
    dashes)
    - It is not necessary to have lots of elements as long as there is a nice
    and clear level design idea behind the level.
    - Collectibles should be placed in an alternate path that is harder to
    reach than the easiest one used just to pass to the next level
    - Verify that player doesn't spawn on spikes or is not stuck inside a wall,
     that all collectibles are reachable, and that it is possible to use every
    platform
    - Tile legend:
```

```
        - '.' (space): Empty air
        - '#': Wall/Platform
        - 'P': Player spawn position
        - '^': Spikes (kills player on touch)
        - 'C': Collectible
    - Levels must be small enough to fit the screen (typically 16x16 grid max, scaled to fit 720x720 resolution).
    - Designer should be able to add more levels by appending them to a list
    - Try to simulate a player traversing the level to double-check level design, keeping in mind player specific movement parameters
    - Example levels (DO NOT REUSE THESE DESIGNS, USE THEM AS A REFERENCE TO CREATE SOMETHING NEW)
        - Example 1(This level is intended to be traversed by a series of wall-jumps on the left half, followed by a jump into the right
        [
            "#############...#",
            "######....##...#",
            "####......##...#",
            "###.......##...#",
            "###.......#....#",
            "###....^..#....#",
            "#......#......##",
            "####...#......##",
            "####...#......##",
            "####...#......##",
            "####...#......##",
            "####...#.....###",
            "..##...#.......#",
            "...#...#^^^.....",
            ".P.....####.....",
            "############....."
        ]
        - Example 1 Step-by-step traversal
            1.Player starts at .P.....####..... near the bottom-left.
            2.Player walks right toward the vertical column made of # blocks ( column 7).
            3.Player should use a combination of jumps and wall-jumps up the narrow shaft (from around row 14 to row 6); The narrowness ensures you must  alternate wall jumps between the left and right surfaces.
```

```
            4.Player must avoid the spike trap in row 5: "###....^..#....#" (Theres a spike (^) right in the center, so youll need to dash or wall-jump carefully to avoid it while climbing)
            5.Likely, player will cling to the left wall just below the spike, then dash up-right to the next safe wall section (up above # block in (6,14) or (11,13))
            6.Player can jump up and reach the next level through the opening in the ceiling in row 0 (above the #...# border in row 0)
- Additional Mechanics: Coyote time (brief window to jump after leaving a platform), jump buffering (pressing jump slightly before landing registers the jump)

2. Story  What is the narrative or setting?
- Setting / Theme: A mountain climbing adventure with minimalist pixel art aesthetic.
- Player role/character: A small climber ascending a treacherous mountain.
- Narrative arc (if any): Progressive difficulty as the player climbs higher up the mountain through multiple screens/levels.

3. Aesthetics  What does the game look, sound, and feel like?
- Visual style: Clean pixel art with a limited color palette, simple geometric shapes for platforms and hazards.
- UI elements: Meters counter in corner (each climbed level adds 100m to the counter), collectibles count under meters counter.
- Audio / Feedback: Simple sound effects for jumping, dashing, collecting items, and death. Looping minimalist chiptune background music.
- Graphical assets:
  - Player: Small square that is red when can dash, blue otherwise.
  - Platforms: Solid colored rectangles.
  - Spikes: Triangular hazards pointing upward.
  - Collectibles: Small yellow sprites.

4. Technology  What platform or medium?
- Screen size: 720x720 pixels.
- Frames per second: 60
- Input method: Keyboard arrow keys for movement, Z key for jump, X key for dash.
- Sound assets: Use Pygame and numpy for generating sounds (include them inside the script) for jump, dash, collect, death sounds and background music.
```

A.3 Fullerton Prompt

Listing A.3: Fullerton Framework Prompt Example

```
Please design a game following these formal elements using a pygame script

- Fullerton Formal Elements
    - Players: 1 character, controller by the player,
        - Player interaction patterns: Player vs. Game
    - Objectives
        - Player's goal is to reach the top of the level by using a variety of platform skills
        - A level ends when player exits the top boundary
        - Collecting optional collectibles (for score or challenge) adding a secondary exploration objective
    - Procedures
        - Starting Action: Player spawns at a predefined position ("P" on the level matrix)
        - Progression of Action:
            - Navigate the environment using movement (Arrow Keys) and platform mechanics
            - Reach the exit level or fal into a spike/pit to respawn
        - Special Actions
            - Jump (Z button)
                - Jump has coyote time and jump buffering
            - Wall-Jump (Z button towards a wall)
                - Player is forced to jump in the opposite direction, backing off the wall with a firm jump, forcing him to alternate between walls. Think of it as a soft dash, where player is uncontrollable after the jump until he doesn't touch the opposite wall or the ground.
            - Dash in 8 directions (X Button + Arrow Key)
            - Collect item
        - System procedures: Physics update loop (gravity, collision), input handling, level transition, win condition detection
    - Rules
        - Rules defining objects and concepts
            - Level: Defined via an ASCII matrix (16x16) where
                - "." is empty space
                - "#" is a wall/platform
                - "^" is a spike
```

```
            - "P" is the player spawn
            - "C" is a collectible
            - Example of a level (DO NOT COPY THIS LEVEL BUT USE IT AS A REFERENCE TO CREATE SOMETHING NEW):
                [
                    "############...#",
                    "######....##...#",
                    "####......##...#",
                    "###.......##...#",
                    "###.......#....#",
                    "###....^..#....#",
                    "#......#......##",
                    "####...#......##",
                    "####...#......##",
                    "####...#......##",
                    "####...#......##",
                    "####...#.....###",
                    "..##...#.......#",
                    "...#...#^^^.....",
                    ".P.....####.....",
                    "############....."
                ]
    - Rules restricting actions
        - Dash is limited to 1 per air time, recharged only by touching the ground (not after a wall jump)
        - Wall-jump cannot attach to level borders, only walls
    - Rules determining effects
        - On player contact with spikes, player dies and then respawns
        - On player falling in a pit, player dies and then respawns
- Resources
    - Lives: Infinite lives
    - Collectibles: Optional items placed in harder paths
    - Health: 1-hit death
    - Actions: Movement, jump, wall-jump, dash
    - Special Terrain: Spikes, walls
    - Score: Measured in climbed meters. Starts at 0. Each new level adds 100m to the score
- Conflict
    - Obstacles: Platform, gaps, spikes
    - Opponents: Environment and gravity
```

```
66          - Dilemmas: What sequence of moves should I do in order to pass this
    level?
67      - Boundaries
68          - Defined by screen level (16x16 tiles, scaled up to 720x720 window
    size)
69          - Player cannot go outside screen borders, except the one on top for
    reaching next level
70      - Outcome
71          - Victory: After 3 levels are cleared, a victory screen is displayed
72          - Loss: If player dies, he respawns at initial spawn point. There's no
    permanent failure
73      - Other elements
74          - Background music and SFX for jump, dash, collect, death
75          - Player is red when can dash, blue otherwise
76          - UI displays meters counter in the top left corner, along collectibles
     counter
```

A.4 Hybrid Prompt

Listing A.4: Hybrid Framework Prompt Example

```
1  Please design this game using the following brief
2
3  Game Design Brief
4  1. Mechanics  What are the rules and actions?
5      - Players: 1 character, controller by the player,
6          - Player interaction patterns: Player vs. Game
7      - Objectives
8          - Player's goal is to reach the top of the level by using a variety of
    platform skills
9          - A level ends when player exits the top boundary
10         - Collecting optional collectibles (for score or challenge) adding a
    secondary exploration objective
11     - Procedures
12         - Starting Action: Player spawns at a predefined position ("P" on the
    level matrix)
13         - Progression of Action:
```

```
        - Navigate the environment using movement and platform mechanics
        - Reach the exit level or fal into a spike/pit to respawn
    - Special Actions
        - Jump
            - Jump has coyote time and jump buffering
        - Wall-Jump
            - Player is forced to jump in the opposite direction, backing off the wall with a firm jump, forcing him to alternate between walls. Think of it as a soft dash, where player is uncontrollable after the jump until he doesn't touch the opposite wall or the ground.
        - Dash in 8 directions
        - Collect item
    - System procedures: Physics update loop (gravity, collision), input handling, level transition, win condition detection
    - Rules
    - Rules defining objects and concepts
        - Level: A 16x16 matrix of ASCII characters that must represent a connected walkable space, meaning all non-wall tiles (., ^, O, P) must be reachable from the player spawn point P by walking through empty space. This ensures the level is both playable and logically coherent. In a level, we have:
            - "." is empty space
            - "#" is a wall/platform
            - "^" is a spike
            - "P" is the player spawn
            - "C" is a collectible
            - Example of a level (DO NOT COPY THIS LEVEL BUT USE IT AS A REFERENCE TO CREATE SOMETHING NEW):
                [
                    "#############...#",
                    "######....##...#",
                    "####......##...#",
                    "###.......##...#",
                    "###.......#....#",
                    "###....^..#....#",
                    "#......#......##",
                    "####...#......##",
                    "####...#......##",
                    "####...#......##",
                    "####...#......##",
```

```
                    "####...#.....###",
                    "..##...#.......#",
                    "...#...#^^^.....",
                    ".P.....####.....",
                    "###########....."
                ]
    - Rules restricting actions
        - Dash is limited to 1 per air time, recharged only by touching the ground (not after a wall jump)
        - Wall-jump cannot attach to level borders, only walls
    - Rules determining effects
        - On player contact with spikes, player dies and then respawns
        - On player falling in a pit, player dies and then respawns
- Resources:
    - Lives: Infinite lives
    - Collectibles: Optional items placed in harder paths
    - Health: 1-hit death
    - Actions: Movement, jump, wall-jump, dash
    - Special Terrain: Spikes, walls
- Entities:
        - Player: player character
        - Collectibles: collectible items
        - Spikes: hazardous terrain
    - Conflict
    - Obstacles: Platform, gaps, spikes
    - Opponents: Environment and gravity
    - Challenge: Increasingly difficult platforming challenges requiring precise timing and movement combinations and a variety of techniques (e.g. wall-jump, dash).
    - Dilemmas: What sequence of moves should I do in order to pass this level?
- Boundaries
    - Defined by screen level
    - Player cannot go outside screen borders, except the one on top for reaching next level
- Outcome
    - Victory: After 3 levels are cleared, a victory screen is displayed
    - Loss: If player dies, he respawns at initial spawn point. There's no permanent failure

```

```
2. Story  What is the narrative or setting?
    - Setting / Theme: A mountain climbing adventure with minimalist pixel art
    aesthetic.
    - Player role/character: A small climber ascending a treacherous mountain.
    - Narrative arc (if any): Progressive difficulty as the player climbs
    higher up the mountain through multiple screens/levels.

3. Aesthetics  What does the game look, sound, and feel like?
    - Visual style: Clean pixel art with a limited color palette, simple
    geometric shapes for platforms and hazards.
    - UI elements: Meters counter in corner (each climbed level adds 100m to
    the counter), collectibles count under meters counter.
    - Audio / Feedback: Simple sound effects for jumping, dashing, collecting
    items, and death. Looping minimalist chiptune background music.
    - Graphical assets:
        - Player: Small square that is red when can dash, blue otherwise.
        - Platforms: Solid colored rectangles.
        - Spikes: Triangular hazards pointing upward.
        - Collectibles: Small yellow sprites.

4. Technology  What platform or medium?
    - Screen size: 720x720 pixels.
    - Frames per second: 60
    - Input method: Keyboard arrow keys for movement, Z key for jump, X key for
     dash.
    - Sound assets: Use Pygame and numpy for generating sounds (include them
    inside the script) for jump, dash, collect, death sounds and background
    music.
```

Appendix B

Questionnaire

This appendix contains the survey instrument administered to playtest participants during the user evaluation phase. The questionnaire was structured into distinct sections, with the game evaluation and authorship sections being repeated four times (once for each evaluated game version).

B.1 Demographics

1. Age?
2. Gender?
3. How often do you play video games?
4. Are you a game developer? If yes, in what role?

B.2 GUESS-18 Evaluation

Note: Playtesters rated the following statements on a 1-to-7 Likert scale (1 = Strongly Disagree to 7 = Strongly Agree). This section was repeated for each of the four game versions played.

1. I find the controls of the game to be straightforward.

2. I think the game is fun.
3. I find the game's interface to be easy to navigate.
4. I feel bored while playing the game.
5. I enjoy the game's graphics.
6. I want to do as well as possible during the game.
7. I feel the game allows me to be imaginative.
8. I am very focused on my own performance while playing the game.
9. I think the game is visually appealing.
10. I feel creative while playing the game.

B.3 Blind Turing Test

Note: This question was asked immediately following the GUESS-18 evaluation for each game version.

1. Was the game you just played generated by an AI? (Yes / No)

B.4 LLM Creative Contribution Proposals

Note: Participants were presented with the following AI-generated mechanical variations and asked to evaluate their validity as design improvements, without knowing they were proposed by an LLM.

***Celeste Classic* Proposals:**

1. Invisible wind currents that push players when they come into contact with them. — *Proposed by GPT-4.1*

2. An invulnerability dash that can be used once per level. — *Proposed by Gemini 2.5 Pro*

3. The ability to use a grappling hook to attach to poles within the level. — *Proposed by Claude Sonnet 4*

***Mind The Gap* Proposals:**

1. Random events affecting the line (e.g., rain → trains are 30% slower; maintenance → a specific station becomes unusable for a week). — *Proposed by GPT-4.1*

2. Using the score to purchase new upgrades (e.g., 50 passengers → New Train; 200 passengers → New Line). — *Proposed by Gemini 2.5 Pro*

3. Differentiated passenger types (e.g., VIP Passengers with lower patience leading to a game over if not promptly served by a train, or Tourists who want to make multiple stops at multiple stations on the same line). — *Proposed by Claude Sonnet 4*

Bibliography

[1] Pfeuffer Müller et al. «SpatialProto: Exploring Real-World Motion Captures for Rapid Prototyping of Interactive Mixed Reality». In: *CHI '21: Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems* (2021) (cit. on p. 1).

[2] Mandeville Crossley et al. «Engineering a Collaborative Framework for Applied Game Development». In: *Conference: Meaningful Play 2016* (2016) (cit. on p. 1).

[3] Andrew Webster. «Neal Stephenson's swordfighting game is officially dead». In: *The Verge* (2014) (cit. on p. 1).

[4] Heather Maxwell Chandler. *The Game Production Handbook*. 3rd ed. Jones Bartlett Learning, 2013 (cit. on pp. 1, 10).

[5] Farbod Azsan. «Indie Game Development Cost Guide 2026». In: *Polydin* (2026) (cit. on pp. 1, 10).

[6] Kultima A. «Defining Game Jam». In: *Conference: 10th International Conference on the Foundations of Digital Games (FDG 2015)* (2015) (cit. on p. 2).

[7] Jon A. Preston. «Game Jams: Community, Motivations, and Learning among Jammers». In: *Int. J. Game Based Learn.* 2 (2012), pp. 51–70. URL: `https://api.semanticscholar.org/CorpusID:36272785` (cit. on p. 2).

[8] M. Meriläinen, R. Aurava, A. Kultima, and J. Stenros. «Game Jams for Learning and Teaching». In: *International Journal of Game-Based Learning* 10 (2 2020), pp. 54–71. DOI: `10.4018/ijgbl.2020040104` (cit. on p. 2).

[9] Ali Arya, Jeff Chastine, Jon Preston, and Allan Fowler. «An International Study on Learning and Process Choices in the Global Game Jam». In: *International Journal of Game-Based Learning* 3 (Oct. 2015), pp. 27–46. DOI: 10.4018/ijgbl.2013100103 (cit. on p. 2).

[10] Andrej Karpathy. "There's a new kind of coding I call "vibe coding", where you fully give in to the vibes, embrace exponentials, and forget that the code even exists. It's possible because the LLMs (e.g. Cursor Composer w Sonnet) are getting too good. Also I just talk to Composer with SuperWhisper so I barely even touch the keyboard. I ask for the dumbest things like "decrease the padding on the sidebar by half" because I'm too lazy to find it. I "Accept All" always, I don't read the diffs anymore. When I get error messages I just copy paste them in with no comment, usually that fixes it. The code grows beyond my usual comprehension, I'd have to really read through it for a while. Sometimes the LLMs can't fix a bug so I just work around it or ask for random changes until it goes away. It's not too bad for throwaway weekend projects, but still quite amusing. I'm building a project or webapp, but it's not really coding - I just see stuff, say stuff, run stuff, and copy paste stuff, and it mostly works." Feb. 1, 2025. URL: https://x.com/karpathy/status/1886192184808149383 (cit. on pp. 2, 7).

[11] *What Is Vibe Coding?* Aug. 15, 2025. URL: https://github.com/resources/articles/what-is-vibe-coding (cit. on pp. 2, 7).

[12] NVIDIA. *Evolving AI-Powered Game Development with Retrieval-Augmented Generation*. NVIDIA Technical Blog. Discusses RAG for rapid prototyping and AI-assisted game development. 2024 (cit. on p. 2).

[13] Vinay Bamil. *Vibe Coding: Toward an AI-Native Paradigm for Semantic and Intent-Driven Programming*. 2025. arXiv: 2510.17842 [cs.SE]. URL: https://arxiv.org/abs/2510.17842 (cit. on pp. 2, 3).

[14] Marko Horvat. «What is Vibe coding and when should you use it (or not)?» In: *TechRxiv* (2025) (cit. on p. 3).

[15] Amy Copadis. *Vibe Marketing: Hype, Reality, and Real Case Studies*. 2025-10-15. URL: https://backlinko.com/vibe-marketing (cit. on p. 3).

[16] Shilpa Shastri. *From Idea to Prototype in Hours: The Rise of Vibe Coding*. 2025-09-09. URL: https://cloudtweaks.com/2025/09/idea-to-prototype-hours-vibe-coding (cit. on pp. 3, 8).

[17] *The State of AI in Game Development*. Naavik. Industry report on AI tools for game prototyping and development. 2025 (cit. on p. 3).

[18] Danrui Li et al. «Cardiverse: Harnessing LLMs for Novel Card Game Prototyping». In: *arXiv preprint arXiv:2502.07128* (2025). Addresses automated card game prototyping using LLMs for novel game mechanics (cit. on pp. 3, 9).

[19] Sam Earle, Samyak Parajuli, and Andrzej Banburski-Fahey. «DreamGarden: A Designer Assistant for Growing Games from a Single Prompt». In: (2025). arXiv: 2410.01791 [cs.HC]. URL: https://arxiv.org/abs/2410.01791 (cit. on pp. 3, 9).

[20] Michael Cook, Simon Colton, and Jeremy Gow. «The ANGELINA Videogame Design System—Part I». In: *IEEE Transactions on Computational Intelligence and AI in Games* 9.2 (2017), pp. 192–203. DOI: 10.1109/TCIAIG.2016.2520256 (cit. on p. 4).

[21] Georgios N. Yannakakis, Antonios Liapis, and Constantine Alexopoulos. «Mixed-initiative co-creativity». In: *International Conference on Foundations of Digital Games*. 2014. URL: https://api.semanticscholar.org/CorpusID:7551405 (cit. on p. 4).

[22] Jim Springfield. *IntelliSense History, Part 1*. Dec. 18, 2007. URL: https://devblogs.microsoft.com/cppblog/intellisense-history-part-1 (cit. on p. 7).

[23] Amanda Silver. *Introducing Visual Studio IntelliCode*. May 7, 2018. URL: https://devblogs.microsoft.com/visualstudio/introducing-visual-studio-intellicode (cit. on p. 8).

[24] Ofer Mendelevitch. *Large Language Models for Code Generation - Part 1*. May 16, 2023. URL: https://www.vectara.com/blog/large-language-models-llms-for-code-generation-part-1 (cit. on p. 8).

[25] Ajay Bandi, Bhavani Kongari, Roshini Naguru, Sahitya Pasnoor, and Sri Vidya Vilipala. «The Rise of Agentic AI: A Review of Definitions, Frameworks, Architectures, Applications, Evaluation Metrics, and Challenges». In: *Future Internet* 17.9 (2025). ISSN: 1999-5903. DOI: 10.3390/fi17090404. URL: https://www.mdpi.com/1999-5903/17/9/404 (cit. on p. 8).

[26] Gemini Team. *Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context*. 2024. arXiv: 2403.05530 [cs.CL]. URL: https://arxiv.org/abs/2403.05530 (cit. on pp. 8, 19).

[27] North et al. «Beyond Syntax: How Do LLMs Understand Code?» In: ICSE-NIER '25. Ottawa, Ontario, Canada: IEEE Press, 2025, pp. 86–90. ISBN: 9798331537111. DOI: 10.1109/ICSE-NIER66352.2025.00023. URL: https://doi.org/10.1109/ICSE-NIER66352.2025.00023 (cit. on p. 8).

[28] Shunyu Yao et al. *ReAct: Synergizing Reasoning and Acting in Language Models*. 2023. arXiv: 2210.03629 [cs.CL]. URL: https://arxiv.org/abs/2210.03629 (cit. on p. 8).

[29] Ahmed Fawzy, Amjed Tahir, and Kelly Blincoe. «Vibe Coding in Practice: Motivations, Challenges, and a Future Outlook - a Grey Literature Review». In: (Sept. 2025). DOI: 10.48550/arXiv.2510.00328 (cit. on p. 8).

[30] Lily Hay Newman. *Vibe Coding Is the New Open Source—in the Worst Way Possible*. 2025-10-06. URL: https://www.wired.com/story/vibe-coding-is-the-new-open-source (cit. on p. 8).

[31] Gabriele Romagnoli. *What professionals really think about "Vibe Coding"*. 2025-06-02. URL: https://uxdesign.cc/what-professionals-really-think-about-vibe-coding-e209345e56cd (cit. on p. 8).

[32] Stack Overflow. *2025 Developer Survey: AI Trends and Developer Sentiment*. Percentages have been normalized based on the number of effective respondents. 2025. URL: https://survey.stackoverflow.co/2025 (cit. on p. 9).

[33] Arthur Juliani. «One Trillion and One Nights: An experiment using LLMs to procedurally generate browser-based JRPGs». In: *Medium* (2025). Explores

LLM integration in game development for procedural content generation (cit. on p. 9).

[34] «PORTAL: Policy Optimization and Reasoning for Tactical Artificial Learning». In: *arXiv preprint arXiv:2503.13356v1* (2025). Framework for developing AI agents capable of playing thousands of 3D video games through language-guided policy generation (cit. on p. 9).

[35] «GVGAI-LLM: Evaluating Large Language Model Agents with Infinite Games». In: *arXiv preprint arXiv:2508.08501* (2025). Benchmark for evaluating LLM performance in game playing and design (cit. on p. 9).

[36] «LIGS: Developing an LLM-infused Game System for Emergent Narrative». In: *Proceedings of the Extended Abstracts of the CHI Conference on Human Factors in Computing Systems*. Explores LLM applications in game narratives and identifies challenges. 2024 (cit. on p. 9).

[37] «Large Language Models and Games: A Survey and Roadmap». In: *arXiv preprint arXiv:2402.18659v1* (2024). Comprehensive survey of LLM roles in games including design assistance and player roles (cit. on p. 9).

[38] Jesse Schell. *The Art of Game Design: A Book of Lenses*. 3rd. CRC Press, 2019 (cit. on pp. 10, 26, 30).

[39] Eric Ries. *Minimum Viable Product: a guide*. Aug. 3, 2009. URL: `https://www.startuplessonslearned.com/2009/08/minimum-viable-product-guide.html` (cit. on p. 10).

[40] Sami et al. Hyrynsalmi. «What is a Minimum Viable (Video) Game?» In: *Challenges and Opportunities in the Digital Era*. Cham: Springer International Publishing, 2018, pp. 217–231 (cit. on p. 10).

[41] Extra Credits. *Making Your First Game: Minimum Viable Product - Scope Small, Start Right*. YouTube video. Jan. 28, 2015. URL: `https://www.youtube.com/watch?v=UvCri1tqIxQ` (cit. on p. 10).

[42] Gianty. *What Is a Vertical Slice? Exploring Key Concepts and Benefits*. Aug. 12, 2024. URL: `https://www.gianty.com/vertical-slice-game-development` (cit. on p. 10).

[43] Vikki Blake. *Why devs need to fail fast, and how to do so*. June 20, 2024. URL: https://www.gamesindustry.biz/why-devs-need-to-fail-fast-and-how-to-do-so-gi-sprint (cit. on p. 11).

[44] Ethan Levy. *Failing fast and hard*. Sept. 9, 2013. URL: https://www.gamedeveloper.com/business/failing-fast-and-hard (cit. on p. 11).

[45] J. Musil et al. «Synthesized Essence: What Game Jams Teach About Prototyping of New Software Products». In: *Proceedings of Software Development Conference*. Evaluates game jam concept for rapid prototyping and new product development. 2010 (cit. on p. 11).

[46] «Evaluating Serious Slow Game Jams as a Mechanism for Co-Designing Serious Games». In: *Games: Research and Practice* (2024). Methodology for game jams focusing on learning outcomes and prototyping (cit. on p. 11).

[47] Maxim Kiselev. *Real reasons (not) to build custom game engines in 2024*. Sept. 19, 2024. URL: https://www.gamedeveloper.com/programming/real-reasons-not-to-build-custom-game-engines-in-2024 (cit. on p. 11).

[48] Gemma Ellison. *The Siren Song of Custom Game Engines: Why Indie Developers Should Resist*. Apr. 13, 2025. URL: https://www.wayline.io/blog/siren-song-custom-game-engines-indie-developers (cit. on p. 11).

[49] Alex Antra. *Game Engines: Part 3 - Troubles of Commercial Engines*. May 1, 2024. URL: https://reconnectrecap.substack.com/p/game-engines-part-3-troubles-of-commercial (cit. on p. 11).

[50] Playablemaker Blog. *The Role of No-Code Technology in Modern Game Design*. May 12, 2025. URL: https://playablemaker.com/the-role-of-no-code-technology-in-modern-game-design (cit. on p. 11).

[51] David Weintrop and Uri Wilensky. «Transitioning from introductory block-based and text-based environments to professional programming languages in high school computer science classrooms». In: *Computers Education* 142 (2019), p. 103646. ISSN: 0360-1315. DOI: https://doi.org/10.1016/j.compedu.2019.103646. URL: https://www.sciencedirect.com/science/article/pii/S036013151930199X (cit. on p. 11).

[52] «Live Game Design: Prototyping at the Speed of Play». In: *Proceedings of the 20th International Conference on the Foundations of Digital Games*. Introduces Live Game Design approach for rapid game prototyping. 2025 (cit. on p. 11).

[53] Shyam Sudhakaran, Miguel González-Duque, Claire Glanois, Matthias Freiberger, Elias Najarro, and Sebastian Risi. *MarioGPT: Open-Ended Text2Level Generation through Large Language Models*. 2023. URL: https://arxiv.org/abs/2302.05981 (cit. on p. 12).

[54] Oleg Chumakov. *Generating prototypes from game design document with Cursor, Zed and LÖVE*. July 9, 2025. URL: https://blog.luden.io/generating-prototypes-from-game-design-document-with-cursor-zed-and-l%C3%B6ve-7b8d932194d7 (cit. on p. 12).

[55] Fernanda De La Torre, Cathy Mengying Fang, Han Huang, Andrzej Banburski-Fahey, Judith Amores Fernandez, and Jaron Lanier. *LLMR: Real-time Prompting of Interactive Worlds using Large Language Models*. 2024. URL: https://arxiv.org/abs/2309.12276 (cit. on p. 12).

[56] Danrui Li, Sen Zhang, Sam S. Sohn, Kaidong Hu, Muhammad Usman, and Mubbasir Kapadia. *Cardiverse: Harnessing LLMs for Novel Card Game Prototyping*. 2025. URL: https://arxiv.org/abs/2502.07128 (cit. on p. 12).

[57] Jussi Kemppainen. *Echoes of Somewhere: Generative AI-assisted experimental game research project*. Devlog. 2024. URL: https://www.echoesofsomewhere.com/ (cit. on p. 12).

[58] W. A. IJsselsteijn, K. Poels, and Y. A. W. de Kort. «The Game Experience Questionnaire: Development of a self-report measure to assess player experiences of digital games». In: *FUGA Technical Report, TU Eindhoven* (2008). Widely used questionnaire for evaluating game experience across multiple dimensions (cit. on p. 13).

[59] Mikki H. Phan, Joseph R. Keebler, and Barbara S. Chaparro. «The Development and Validation of the Game User Experience Satisfaction Scale (GUESS)». In: *Human Factors* (2016). Psychometrically validated comprehensive gaming scale for playtesting (cit. on p. 13).

[60] Penelope Sweetser and Peta Wyeth. «GameFlow: a model for evaluating player enjoyment in games». In: 3.3 (2005). URL: https://doi.org/10.1145/1077246.1077253 (cit. on p. 13).

[61] Richard Ryan, C. Rigby, and Andrew Przybylski. «The Motivational Pull of Video Games: A Self-Determination Theory Approach». In: *Motivation and Emotion* 30 (Dec. 2006), pp. 344–360 (cit. on p. 13).

[62] Nicole Lazzaro. «Why we Play Games: Four Keys to More Emotion without Story». In: *Game Dev Conf* (Jan. 2004). URL: https://www.researchgate.net/publication/248446107_Why_we_Play_Games_Four_Keys_to_More_Emotion_without_Story (cit. on p. 13).

[63] David Pinelle, Nelson Wong, and Tadeusz Stach. «Heuristic Evaluation for Games: Usability Principles for Video Game Design». In: Sept. 2008 (cit. on p. 13).

[64] Davide Aloi, Sara Bouzit, and Jie Li. «Biometric Methods for User Research: Three Case Studies». In: *Extended Abstracts of the CHI Conference on Human Factors in Computing Systems*. CHI EA '24. 2024. URL: https://doi.org/10.1145/3613905.3637132 (cit. on p. 13).

[65] Joseph R. Keebler, William J. Shelstad, Dustin C. Smith, Barbara S. Chaparro, and Mikki H. Phan. «Validation of the GUESS-18: a short version of the game user experience satisfaction scale (GUESS)». In: *J. Usability Studies* 16.1 (Nov. 2020), pp. 49–62 (cit. on pp. 13, 39).

[66] L. E. Nacke et al. «Gameplay experience testing with playability and usability surveys: An experimental pilot study». In: (2010). Methodology for correlating game experience with quality measures (cit. on p. 14).

[67] T. Olsen, K. Procci, and C. Bowers. «Serious Games Usability Testing: How to Ensure Proper Usability, Playability, and Effectiveness». In: *Design, User Experience, and Usability. DUXU 2011*. Three-tiered approach to game usability assessment. 2011 (cit. on p. 14).

[68] S. Kirginas. «User Experience Evaluation Methods for Games in Serious Contexts». In: Comprehensive review of quantitative and qualitative UX evaluation methods. 2023, pp. 19–42 (cit. on p. 14).

[69] D.F. do Nascimento et al. «Player Experience Evaluation in Digital Games: A Multidimensional Model». In: (2026). Seven-dimensional questionnaire for evaluating player experience (cit. on p. 14).

[70] Alessandro Canossa Magy Seif El-Nasr Anders Drachen. *Game Analytics: Maximizing the Value of Player Data*. 1st ed. Springer, 2013. ISBN: 1447147685; 9781447147688; 9781447147695; 1447147693 (cit. on p. 14).

[71] Michael McCurdy, Christopher Connors, Guy Pyrzak, Bob Kanefsky, and Alonso Vera. «Breaking the fidelity barrier: an examination of our current characterization of prototypes and an example of a mixed-fidelity success». In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '06. Association for Computing Machinery, 2006, pp. 1233–1242. ISBN: 1595933727. URL: https://doi.org/10.1145/1124772.1124959 (cit. on p. 14).

[72] Steve Swink. *Game feel: a game designer's guide to virtual sensation*. CRC press, 2008 (cit. on p. 14).

[73] Wei-Lin Chiang et al. *Chatbot Arena: An Open Platform for Evaluating LLMs by Human Preference*. 2024. arXiv: 2403.04132 [cs.AI] (cit. on p. 19).

[74] OpenAI. *GPT-4 Technical Report*. Tech. rep. OpenAI, 2024. URL: https://arxiv.org/abs/2303.08774 (cit. on p. 19).

[75] Anthropic. *The Claude 3 Model Family: Opus, Sonnet, Haiku*. Tech. rep. Anthropic, 2024. URL: https://www-cdn.anthropic.com/de8ba9b01c9ab7cbabf5c33b80b7bbc618857627/Model_Card_Claude_3.pdf (cit. on p. 19).

[76] *Celeste*. July 21, 2015. URL: https://www.lexaloffle.com/bbs/?tid=2145 (cit. on p. 22).

[77] *Celeste's 8-Bit Classic Progenitor Gets Sequel Made In Three Days*. Jan. 25, 2021. URL: https://screenrant.com/celeste-classic-8-bit-sequel-three-days (cit. on p. 22).

[78] *Celeste Classic - Game*. Jan. 25, 2021. URL: https://maddymakesgamesinc.itch.io/celesteclassic (cit. on p. 22).

[79] *Mind The Gap*. Oct. 18, 2013. URL: https://dinopoloclub.com/2013/10/18/mind-the-gap (cit. on p. 23).

[80] *Mind The Gap - Game*. Apr. 29, 2013. URL: https://web.archive.org/web/20130811164201/http://prototypes.codepoint.co.nz/ld26 (cit. on p. 23).

[81] *Sacrifices Must Be Made - Ludum Dare 43 Page*. Dec. 4, 2018. URL: https://ldjam.com/events/ludum-dare/43/sacrifices-must-be-made-2 (cit. on p. 24).

[82] *Sacrifices Must Be Made - Game*. Dec. 4, 2018. URL: https://dmullinsgames.itch.io/sacrifices-must-be-made (cit. on p. 24).

[83] Tracy Fullerton. *Game Design Workshop: A Playcentric Approach to Creating Innovative Games*. 4th ed. CRC Press, 2018 (cit. on pp. 26, 31).

[84] Craig Van Slyke. *Prompting 101: Open versus Structured Prompts*. Mar. 18, 2025. URL: https://aigoestocollege.substack.com/p/prompting-101-open-versus-structured (cit. on p. 26).

[85] Aali Asad et al. *Structured Prompting Enables More Robust Evaluation of Language Models*. 2025. arXiv: 2511.20836 [cs.CL]. URL: https://arxiv.org/abs/2511.20836 (cit. on p. 26).

[86] Cole Stryker. *What is human-in-the-loop?* July 8, 2025. URL: https://www.ibm.com/think/topics/human-in-the-loop (cit. on p. 26).

[87] Valeriia Kuka. *Role Prompting*. Sept. 27, 2024. URL: https://learnprompting.org/docs/advanced/zero_shot/role_prompting (cit. on p. 30).

[88] *Game Matters: The Game Jam - IGDA Milan @ SAE Institute*. URL: https://itch.io/jam/game-matters-the-game-jam-igda-milan-sae-institute (cit. on p. 36).