A Case Study in Re-engineering and Porting a Legacy Codebase using LLM pair-programming: A First Look

Swen E Gaudl[1]

[1]University of Gothenburg, Gothenburg, Sweden

Abstract

Transformers have been applied to most domains and have been hailed as a tool to solve most of the problems we can describe in written language, from writing or adjusting essays, to solving mathematical problems as well as writing code. Legacy codebases are a substantial problem as some of the code is hard to read, understand or port. Specifically when the codebase has evolved over time, lots of bad practices and patterns are inherent. Code or procedural complexity often is also higher in highly adaptable or parametric software. Most current tools and agentic environments make it easy to write and port code across different languages and paradigms, but a key question is: Is this still possible once the code reaches a certain size and complexity? In this case study, porting one such human-written legacy codebase across different platforms is described, including the structured approach, some of the challenges faced and the results of a first-look.

Keywords

Re-engineering, legacy code, Generative AI, Game Porting

1. Introduction

The motivation for this work emerged from the frustration of being stuck in a specific language and ecosystem and having a highly complex parametric software at hand. Cross-platform development in games is quite common as developers want to reach a broad audience, so tools such as Unity, Unreal and Godot support a wide range of target platforms. Historically, game development has focused a lot on optimized code and tailoring games to a specific platform which in turn creates a lot of hard dependencies. As features accumulate and the codebase matures, bugs also creep in and often a legacy codebase includes a lot of implicit references and setups, making the code at some point hard to understand and follow.

Large Language Models (LLMs) are exceptionally good at producing small to medium pieces of code, making it easy to design software through descriptions in human language and receiving an implementation in even the most uncommon framework, platform, or language.

2. Problem Domain

This "first look" starts with the intention to port a mobile highly parametric game design tool "ParaVida" from Swift and SpriteKit to C# and Godot. ParaVida is the continuation of Wevva [1] which was an experimental game-design app for iOS from the MetaMakers Institute, Falmouth University, released in 2017. It was one of several "fluidic games" built on a parameterized iOS game engine designed for on-device game creation.

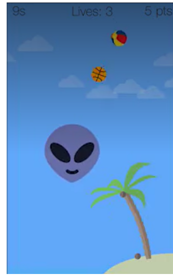
A key motivation for porting is, that the Swift environment is limiting the tool to the Apple ecosystem. This has for one implications on the development tools but secondly also on the target audience. Additionally, with a codebase written in Swift2, upgraded to Swift3 and later to Swift4 (a requirement for being able to deploy the tool), significant language and framework changes throughout the evolution of Swift occurred leaving their remnants in the codebase.

Joint Proceedings of the ACM UMAP Workshops 2026, UMAP 2026, June 8–11, 2026, Gothenburg, Sweden
swen.gaudl@gmail.com (S. E. Gaudl)
https://swen.interactions.se (S. E. Gaudl)
0000-0002-0877-7063 (S. E. Gaudl)

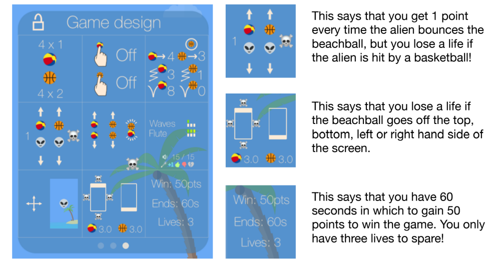
© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073
published 2026-05-28

CEUR-WS.org/Vol-4206/ParaAdapt-3.pdf



(a) The game interface of the tool, showing one of the games.



(b) The design interface and a description of three key aspects/ dimensions in the design space.

Figure 1: Legacy Game Design tool ParaVida/Wevva, showing the game and the top level design interface. In the game, the user is supposed to drag the alien head to bounce the water balls while avoiding basketballs.In the design interface, each of the grid cells allows the user to explore one of the 9 higher design dimensions.

3. LLM-Based Porting Approach

The following section describes the porting strategy as well as the porting approach. Before this approach was used, several local LLMs (*Llama*3, *Qwen*3-*coder*, *GLM*-4.6 with a large context window of 100K) were tried. I also briefly explored the Agentic environment opencode[2] and direct repository and code access using both local models but also hosted models (*KimiK*2.5, *BigPickle* and *Nemotron*3*SuperFree*), but this failed due to the complexity of the codebase and the usage of free and smaller models. After a few interactions, models got stuck, started hallucinating, forgot structures, or could not consistently pursue the port but switched between different parts in the codebase leading to endless loops. The reported approach is based on using *ChatGPT*-5.2 in thinking mode. I split the approach into 3 distinct sessions, which are on different abstraction levels of the needed tasks. I first asked about creating a structured porting approach, splitting the task into manageable subtasks; those could also be carried out by agents. The second session used those instructions to carry out the port. Instead of providing access to the codebase through GitHub, I uploaded a zipped version of the repository. Through earlier tests, this improved the quality of the answers in contrast to providing access to the repository, which was not always read in in its entirety. Using a single session or mixing tasks on different abstractions levels was tried multiple times always leading to a bad or fast degrading performance. Thus, the split into the sessions and levels of tasks were done to avoid similar issues, where a model went into a state that was hard to recover from, as it lost focus and started breaking large parts of the codebase.

3.1. Pair programming an LLM-mediated re-engineering task

The work was split into three very different phases/sessions. When the entire task turned from a re-engineering task by human-human pair-programming and structural analysis with re-engineering tools into an LLM-mediated activity[3] capable of going beyond what a single developer could have done in the same amount of time. A paragraph is everything from a short prompts saying ”continue” to detailed instructions without an empty line forming a prompt.

- **SwiftToGodot** is the short framing phase: 128 non-empty paragraphs, 937 words in total, and one logged ”think” span of 1m32s. It is mostly architectural and process-setting, with very little code. It establishes the spec-first migration strategy, the thin vertical slice, the SpriteKit⟶Godot mapping, and the notion of a parametric runtime rather than a line-by-line port.
- **Game Spec Migration Guide** is the main implementation phase: 7,566 non-empty paragraphs, 241,772 words, 115 logged ”think” spans totaling about 44m31s, and very high code density. This is where the work moves from migration theory into loaders, adapters, runtime construction, controller dragging, collisions, clustering, UI, chromosomes, and many compile/runtime fixes. It is the core engineering log.

- **3D Game Icon Creation** is smaller in size but surprisingly expensive in iteration: 861 non-empty paragraphs, about 14,774 words, 30 logged think spans totaling about 68m42s, with one "think" span reaching 16m50s. Although it begins as an asset task, it repeatedly spills back into runtime loading, adapter emission of icon metadata, controller-image behaviour, and spec/adapter alignment.

The high-level project trajectory was:

- Strategic abstraction: define the migration model before coding.
- Executable runtime: build schema, loaders, adapters, spawners, entity factory, and a minimal shell.
- Language and binding stabilization: move from GDScript to C#, then spend substantial effort on Godot C# dictionary/Variant semantics.
- Mechanics parity chase: controller drag, bounded movement, collisions, cluster logic, scoring fidelity, chromosome structure.
- Asset/runtime integration: icons, controller imagery, spec-side symbolic fields, and the adapter paths needed to make them actually appear in-game.

3.2. Observations of the Process: Correctives, frustration, and redirection

Below are the analysed observations. These are based on an analysis of the session transcripts and the interactions of the user with the system. The strongest pattern from the logs is: The user's tone began strategically, then became very corrective once the code was expected to compile and the behaviour was expected to match the legacy app. The corrections were not random; they were highly diagnostic focusing on structure first and then detailed elements. The user intervened most often in following five situations.

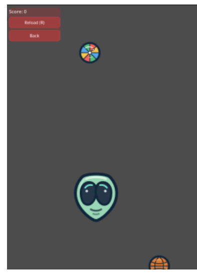
When abstraction drifted away from the actual codebase: The user repeatedly redirected the discussion away from generic patterns and toward exact file-level integration by asking which files needed modification, requesting drop-in replacements, and asking not just for "the idea", but for code that matched the current game components such as the main game loop, entity factories and the interaction system.

When Godot C# binding details were mishandled: Throughout the process, dictionary and Variant issues emerging from missing access to Godot's specific syntax were the points, where the user's frustration was most explicit. Statements such as "This is not the right syntax for a Godot dictionary" and "you again brought in CS1061..." showed a low tolerance for answers that were structurally plausible but not binding-correct for Godot .NET.

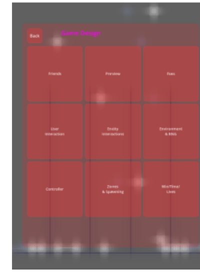
When a previous stable behaviour regressed: The statement, "Something is completely wrong now. The UI is off the screen...," was not just a complaint about a bug; it showed that preserving already working behaviour mattered as much as adding new features. This reflected a classic re-engineering concern: regression control during incremental replacement. Using an agentic approach directly on the repository, unit tests could have provided additional safeguards and reduced this friction.

When visual output failed the desired style or fidelity: The icon thread showed a similar pattern on the asset side: "the snowflake is still looking bad," along with earlier feedback about other visual entities. This was a different type of correction, but it had the same structure: the user was steering toward concrete quality criteria rather than accepting something that was merely close.

When the user wanted more detail: There was a clear escalation path in what the user asked for. At first, the user accepted planning and process. Then, the user wanted more executable steps leading to exact code inquiries. After having executable code, exact file ordering was next, enforcing also minimal modifications of the codebase, and when refactorings were needed they opted for drop-ins. Finally, once the project became tangled, the user wanted explanations at the level of "how and where do the model update a specific file" or "what did the model do with old classes or removed code calls"



(a) The new game interface



(b) The new design interface

Figure 2: The Ported Godot version, a) showing a game that is loaded from the data of the original tool and b) showing the top level design interface with a game running in the background.

4. Conclusion

To conclude this ”first look”, the re-engineering progressed and resulted in a working port of the legacy software which seemed to time-consuming without the support this process offered, but the progress was uneven and not frictionless.

What worked well: The project found the right conceptual center early: spec-first migration, thin slice, and adapter-based normalization. The logs show a move from plan to runtime: loaders, world-building, spawners, controller geometry, icon resolution, and chromosome structures were all introduced. The user kept forcing the work toward semantic faithfulness whenever it drifted into convenience patches.

What went poorly: The implementation phase accumulated too many compile-time and binding-level misfires, especially around Godot C# and language-specific implementations. Several code drop-ins appear to have been produced without strong compile verification; one icon-session answer explicitly says the .NET SDK was unavailable, so compile checking could not be completed. That is exactly the condition under which “looks right” code created long correction loops. The process repeatedly mixed three layers at once: legacy schema interpretation, runtime architecture, and UI/editor behaviour. That made debugging harder because regressions were often introduced while solving a problem on different layer.

The most important practical conclusions: The hardest part of the port was not the Swift⟶Godot conceptual mapping. It became contract discipline between legacy spec, adapter, runtime, and UI. It also showed haw far you can push LLM pair-programming when only relying on an EDU account and not having to consider additional costs.

Declaration on Generative AI: During this work, the author used ChatGPT and other LLMs for code generation and as analytical tool for text interpretation in combination with traditional AI approaches. The author reviewed and edited the content as needed and takes full responsibility for the publication’s content.

References

[1] E. Powley, M. Nelson, S. Gaudl, S. Colton, B. P. Ferrer, R. Saunders, P. Ivey, M. Cook, Wevva: Democratising game design, in: Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, volume 13, 2017, pp. 273–275.

[2] opencode, https://opencode.ai, 2025. Last Accessed: 2026-05-07.

[3] M. Rost, Co-disclosing the computer: Llm-mediated computing through reflective conversation, in: Proceedings of the 2026 CHI Conference on Human Factors in Computing Systems, 2026, pp. 1–13.