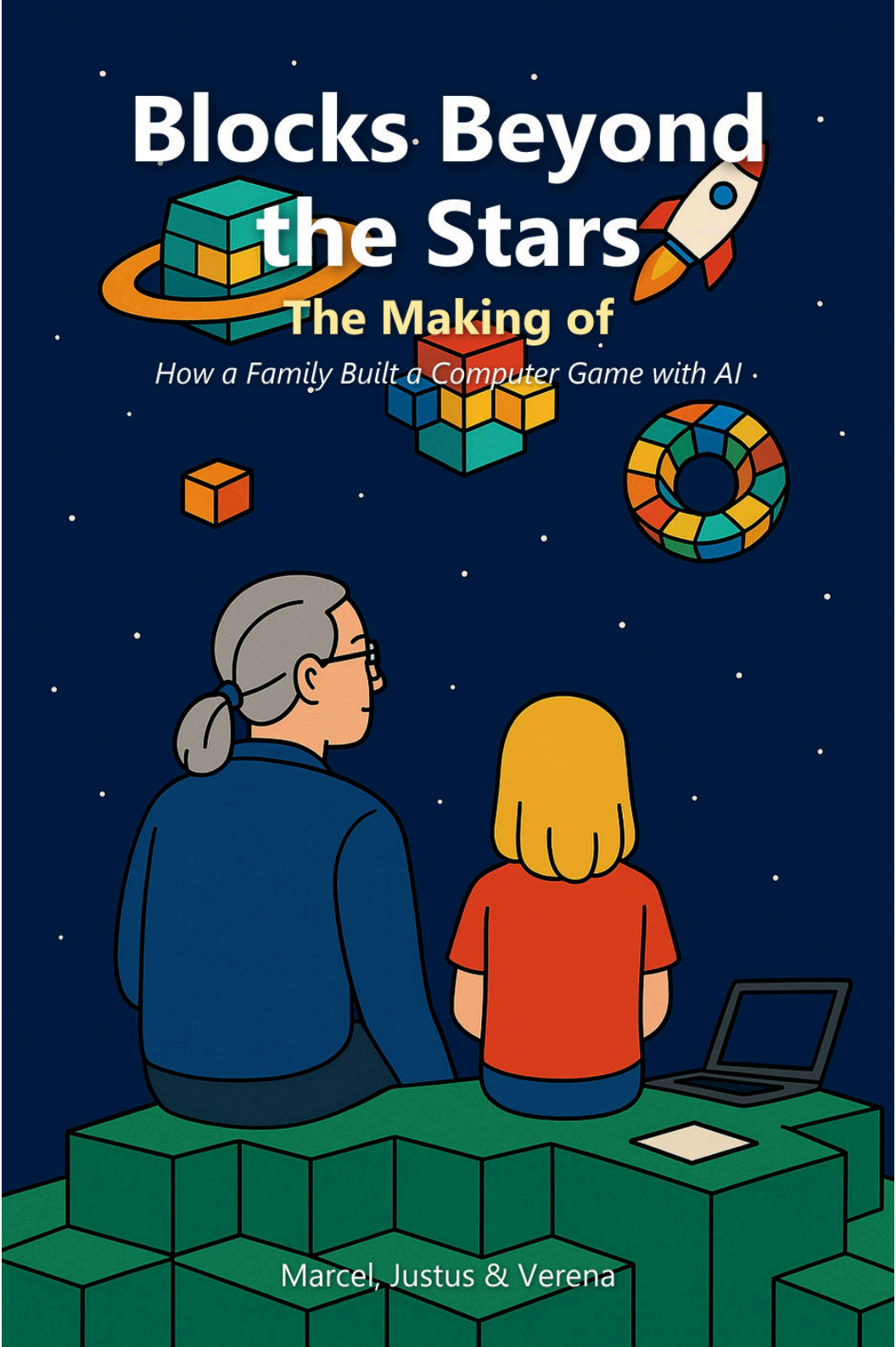
Blocks Beyond the Stars

The Making of

How a Family Built a Computer Game with AI



Marcel, Justus & Verena

Blocks Beyond the Stars - The Making of

How a Family Built a Computer Game with AI

Marcel, Justus & Verena

blocksbeyondthestars.com

Cover design: Created by Marcel Dütscher using AI image generation (OpenAI API).

Typesetting and layout: Marcel Dütscher

Author's note: As described in detail in this book, AI tools were used as assistants in creating this text and the illustrations (including the cover). Editorial responsibility, curation, and approval remained with the human authors throughout.

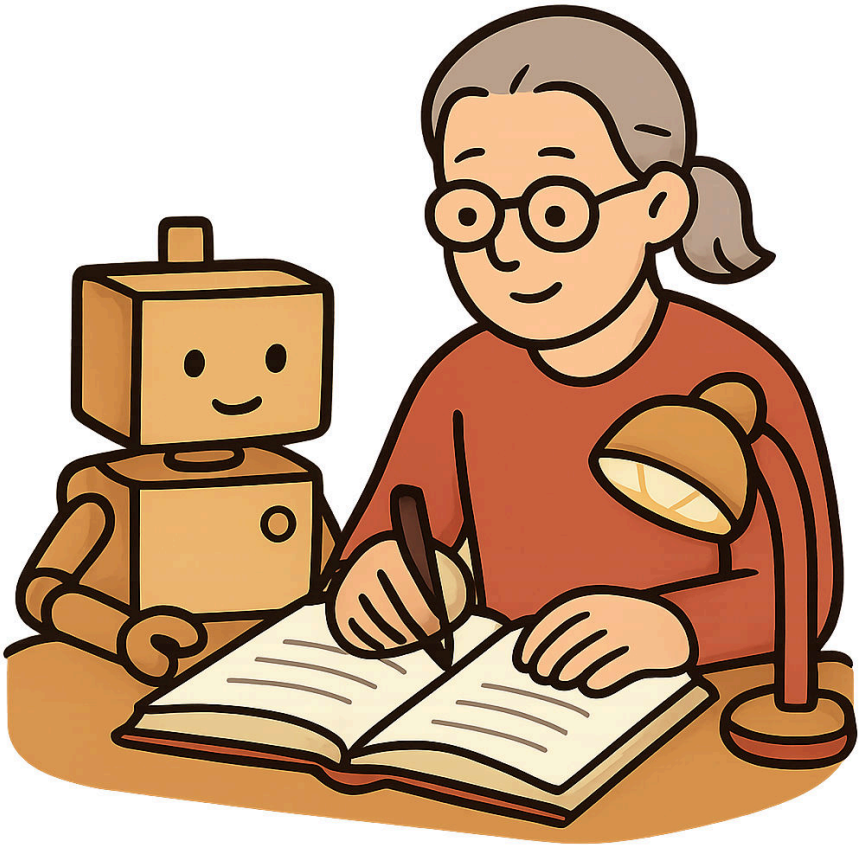
ISBN: 979-8-1868-6082-2

Contents

Dedication

I dedicate this book to my son Justus:

Always hold on to your imagination, because with it you can build great things!



Preface — Full Disclosure, Before We Begin

This book is about how a family built a computer game with artificial intelligence — the code, the graphics, the sound, the music, the servers, the marketing, the videos. It would be fairly absurd if, of all things, the book about it had secretly been written the old-fashioned way. It wasn't. So, for the sake of completeness, and before you turn the first page:

This book, too, was written with AI.

More precisely: it came about following exactly the same recipe as the game it describes — and anyone who has read the book will recognize that recipe, step by step, in how the book itself was made. It didn't start with a blank page but with a **stocktaking**: What sources are there — the version history with over a thousand dated entries, the project's 6,700-line logbook, the devblog posts, the original requirements documents, the family's memories? That became a **plan**: an outline with an honest note on where material exists and where memory alone has to carry the story. Then it was **built chapter by chapter** — rough draft, feedback, refinement, one chapter after another, just as the game was built one feature after another. The machine wrote; the human narrated, corrected, cut, and decided. Where a piece of information was missing, it was **not invented** but flagged as an open question and collected from the author. The anecdotes in this book come from Marcel's and Justus's memories, the facts from the public project history — and the text keeps the two apart.

What does that mean for you as a reader? Two things, I think.

First, a promise: **Everything reported here is true in the best sense**

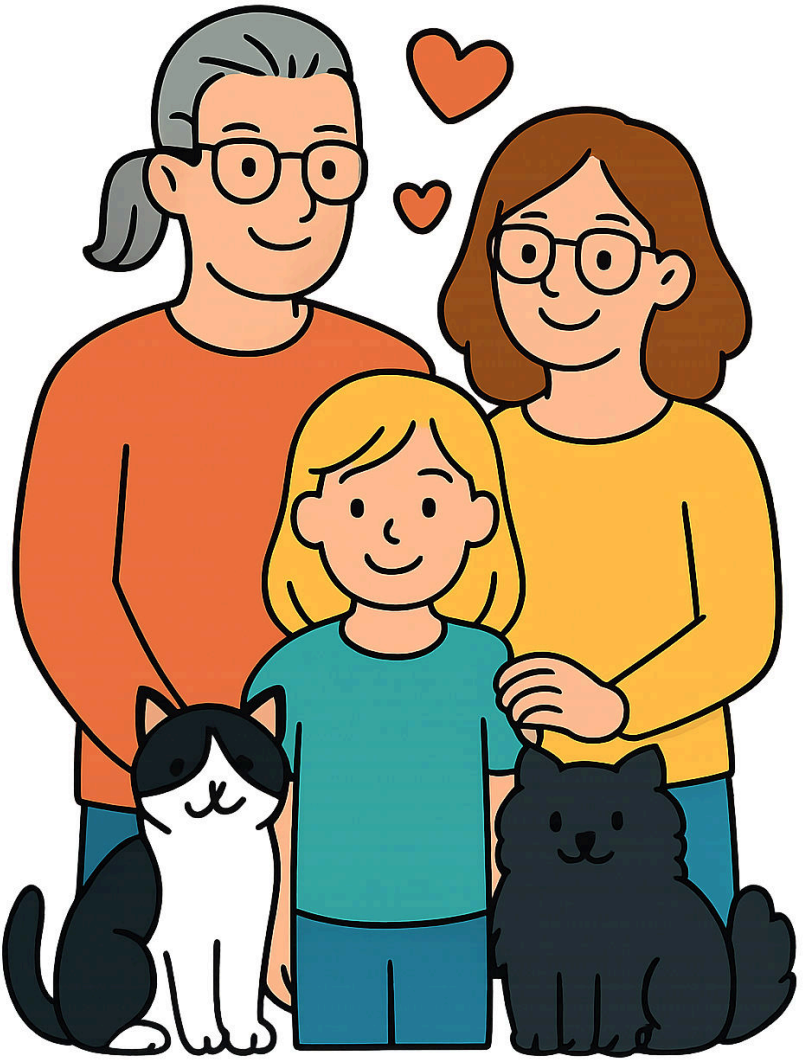
available to us — the numbers come from the publicly viewable version history (the project is open source; you can verify every claim, and we expressly invite you to). Justus's quotes are real quotes, the mishaps are real mishaps. An AI built this book's sentences. What those sentences claim is the responsibility of humans.

And second, some perspective, which may be the real message of the whole project: in the years ahead, the question "Was this made by an AI or by a human?" will more and more often be the wrong question. The right one is: **Is there someone behind it who checked it, whose story it is, and who stands behind it?** For this book: yes. A father, a mother, a ten-year-old — and a workshop full of machines doing what workshops have always done: they make possible what a family couldn't manage alone.

One more thing before we begin — and I'm saying it right here because I know some people will open this book with mixed feelings: Yes, AI was used here everywhere that people normally work. Programmers, graphic artists, copywriters, musicians, sound designers. If that thought makes you uneasy — maybe because it's your profession — please keep reading anyway. This book doesn't smile that worry away; it gets a chapter of its own (Chapter 23), and there I answer as honestly as I can: as someone who doesn't just write about this change but works inside it.

The rest is the story. It begins the way it will end: with a child's request.

— Marcel (checked, cut, accountable) & Claude (typed)



Acknowledgements

This book — like the game it tells the story of — would not exist without these people.

My wife **Verena**, who supports me in everything.

My son **Justus**, who makes all of this possible in the first place.

My **dad**, who bought me my first computer when I was six — a C64. That's where it all began.

And my **mom**, to whom I owe my creativity. And my urge to tinker.

Thank you.

Chapter 1 — "Let's just try it. Please, Dad!"

"And why can't we just make a real computer game right away? In 3D?"
— Justus, 10, on an evening that was going to get expensive

The Backstory: We'd Done This Before — Just Smaller

This story doesn't start from zero, and that matters if you want to understand it properly. By the time the evening came that this chapter is about, Justus and I already had a small shared workshop career behind us: a handful of projects, all following the same pattern — kid has an idea, dad is off work, AI writes the code.

We'd had **microcontrollers** programmed — those tiny little computers that blink and beep in maker projects. That turned into **Pixel-Pets**: a kind of modern Tamagotchi, a pixelated pet to feed and care for, built ourselves. Then **StarPilots**: an arcade game on a Game Boy clone — one of those retro handhelds you assemble yourself. And at some point Justus built the prototype of a browser game called Catapult-Blade **entirely on his own** — his first solo project, without Dad at the keyboard. (All three, by the way, are still publicly on GitHub today; the backstory of this book is verifiable.)

None of it was big. But something crucial had happened, and it's the real point of the backstory: **As a family, we had already learned that ideas don't have to stay in your head.** That between "wouldn't it be cool if…" and a thing you can hold in your hand, there's no longer an ocean — just a few evenings. That knowledge — more a feeling than

knowledge — is the soil everything that follows grew in. Kids who have once seen their idea become real never stop having ideas.

The Evening

And then that evening came. Justus stood in front of me, the Game Boy clone with his arcade game on it in his hand, and said:

"Dad, can we make a kind of 3D Minecraft on this? But in space, with spaceships?"

My answer was an engineer's answer: "Nope. That thing doesn't have enough memory."

Technically correct. And completely beside the point, as became clear two seconds later, because Justus didn't even flinch:

"And why can't we just make a real computer game right away? In 3D? Or one for the Xbox?"

Let that exchange sink in for a moment, because it contains the whole book in four sentences. The kid hits a limit — and his reflex isn't to shrink the idea but to **switch platforms**. The adult, on the other hand… well. My first reaction was very grown-up, and to my own embarrassment I quote it verbatim: "Phew. I don't think it's that easy."

Adults often think inside fences they built themselves. I knew, after all, what a "real computer game" means: years, teams, budgets — all that know-it-all knowledge from a career in IT. Justus knew none of that. He only knew what he'd learned in our little workshop: that ideas are buildable. And so he said the sentence this project really begins with — not as a vision, not as a concept, but as what it was: a plea.

"Let's just try it. Please, Dad!"

No grown-up objection stands a chance against that argument. We

started that very evening.

Managing Expectations (or: How to Be Wrong)

One thing I want to put on record honestly, because it may be the most comforting footnote in this book: **I didn't believe in the project.** At least not in what it became. My inner target that evening was defensive: keep my son's expectations low, prevent disappointment. I literally thought: *And if all that comes out in the end is "just" a concept for a game — that's already a huge win.* A document. That was my ambition.

I'm telling you this because right now there are probably plenty of parents, tinkerers, and skeptics out there facing the same decision with the same inner brake. The brake is reasonable. It's saturated with experience. And, as of 2026, it's simply out of date — but you only know that afterward. You have to try it. Please.

The Interview

How do you start? Not with code. We started with a conversation — one I deliberately set up like an **interview**: I opened a ChatGPT chat, started the audio recording, and then talked with Justus about his game idea. He talked, I played the interviewer — asking follow-ups, keeping things going, adding a point here and there, but at the core: **his** game, **his** answers. What kind of ship is it? What's the first thing you do? What happens when you die? Is the ship a menu or a place? (A place. Categorically. With rooms: cockpit, medbay, workshop, cargo hold — anyone who knows the later game will recognize every one of these kid answers.)

> **In a nutshell: AI, ChatGPT, and transcription** When this book says "AI," it almost always means a *language model*: a program trained on enormous amounts of text that understands and produces language — answers, structures, plans, code. ChatGPT

> is the best-known one. And it can listen: *transcribing* means turning spoken words into text — a matter of seconds these days. For us, that meant the barrier for a ten-year-old to "write a concept document" was exactly zero. All he had to do was talk. He's good at that.

From the transcript of that conversation, we then had ChatGPT create a **structured requirements document**. (The raw transcript itself no longer exists — it got lost somewhere in the depths of the chat history. I regret that today; it would have been the founding document par excellence. If you're going to do this yourself: save the transcript. On day one, you never know which file will want to open a book later.)

And now comes my favorite part, because this is where a nice anecdote turns into a working process: **Justus read the document. Meticulously.** Not skimmed — read. And threw out what he didn't want. It was his game, and the document had to conform to that, not the other way around. After that, we polished this first draft over several iterations in the chat: sharpening, rearranging, cutting, adding — the rounds that eventually became those 2,176 lines whose core sentences describe the game to this day:

> *"Minecraft meets spaceship building, planet exploration, and survival crafting in space."*

> *"I start with a small, simple ship. Through exploration, mining, crafting, and research, I gradually build my own bigger, better spaceship and reach ever more dangerous and exciting worlds."*

Those polishing rounds also produced the quote of the day, the one I've told everyone since who wants to know whether kids can even "handle" working with AI. Justus, after yet another effusively complimentary ChatGPT reply, deadpan:

"Dad, the AI sure praises you a lot. And most of the time that's worth nothing at all…"

Proud dad. In week two, a ten-year-old had seen through something adults book media-literacy seminars for: that machine friendliness is not information. He didn't turn the praise off — he stopped *counting* it. Ever since, our house rule is: the AI can praise as much as it likes. What gets taken seriously is what it *builds*.

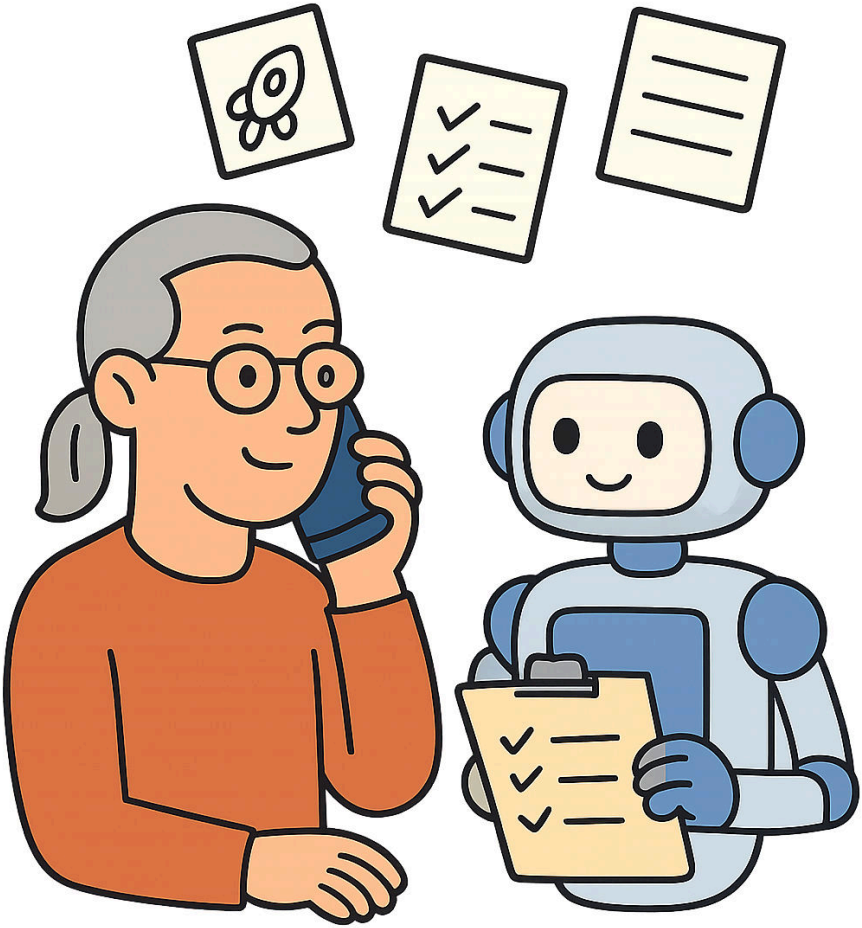
What That Evening Left Behind

At the end of this phase — not a single line of the game had been programmed yet — something existed that most hobby projects never get: a complete concept, personally edited by the product owner. And, more importantly: a method. Talking instead of typing. Interviewing instead of dictating. The kid decides, the AI structures, the dad moderates.

In parallel, I had started dictating the second half of the truth — my technical thoughts, fed by years of experience in software development: How do you build something like this so it holds up? That became the second document, the one containing the most consequential sentences of the project. That's what the next chapter is about.

> **What the AI actually did here** ChatGPT listened (audio transcription), turned the father-son interview into a structured concept document, and helped revise it over several rounds — 2,176 lines in the end. It contributed not a single game idea and (Justus's observation!) praised far too much. The ideas: from the kid. The cuts: from the kid. The structure: from the machine. The request to start at all: the most important line of the project, and it came from a human, age 10.

Next chapter: From Dream to Spec — a father dictates his professional experience onto tape.



Chapter 2 — From Dream to Spec

"Also able to run on a Raspberry Pi 5 under Linux." — from the technical requirements document that no one ever had to question again — it was right

---

The concept document from Chapter 1 describes a dream: space pioneer, your own ship, endless worlds. What it deliberately does *not* describe is the unromantic half: What do you build it with? Where does it run? What happens when two players mine the same block at the same time? Who's right when the game claims one thing on my machine and another on yours?

So the first dictation was followed by a second. Again spoken aloud, again into ChatGPT, again in — and this is meant literally — **umpteen iterations**, in its own project folder where numerous chats piled up over time. The result was a document: 987 lines, titled "Technical Requirements Document." Where the first document sounds like an excited kid, the second sounds like a cautious engineer. Both were intentional. They are the two voices a project needs.

> **In a nutshell: What is a specification (and why bother)?** A specification — old-fashioned term: a spec sheet — is the written answer to the question: *How will we know the thing was built right?* It costs time up front and saves double later, because no one has to guess anymore. That was true back when humans built for humans. For AIs it's true with a vengeance: an AI doesn't guess hesitantly — it guesses fluently and confidently. The more precise the document, the less is left to guess.

The Three Sentences That Decided Everything

These 987 lines contain three decisions that carry the project to this day. I'll name them here even though their stories play out in later chapters:

First: The server is always right. The game consists of two programs — a *client* that every player runs on their own machine, and a *server* where all the threads come together. And the ground rule is: the client only displays; the truth lives on the server. If your screen claims you mined a diamond block but the server sees it differently — then you didn't. This rule sounds technical, but it's a question of character: it makes cheating structurally hard and arguments impossible. (Why that matters twice over with an AI on the team is the subject of Chapter 5.)

> **In a nutshell: Client and server** Picture a restaurant. The *client* is the waiter at your table: friendly, fast, shows you the menu, and brings what you order. The *server* is the kitchen: that's where the actual cooking happens, where it's decided what's available and what's sold out. A waiter who invents dishes on his own makes a brief impression and then chaos. Online games work exactly the same way — which is why the rule is: the kitchen has the final word.

Second: C# and a server you can host yourself — all the way down to a hobbyist machine. The document pins down the programming language (C#, pronounced "C sharp") and the game engine (Unity). Plus a requirement I still love to quote because it's so wonderfully concrete: the server should be *"also able to run on a Raspberry Pi 5 under Linux"* — a hobbyist computer for under a hundred euros, barely bigger than a credit card. Either way, the effect was enormous: whoever plans for the weakest device automatically builds lean. That one line later kept our server from becoming a gluttonous monster — and it paved the way to the five-euro rented server our public worlds run on

today (Chapter 18).

Third — and this is my favorite line: test coverage from day one. Already in the dictation, before the first line of game code existed, it was settled that every piece of the program would come with automated tests — small checking programs that nail down behavior. That wasn't late remorse after the first data loss; it was part of the order. Why that's not optional but a matter of survival in an AI project gets its own chapter (Chapter 6: "Trust Is Good, 1,100 Tests Are Better"). For now, just this: of all the decisions in this phase, none saved more evenings than this one.

Umpteen Iterations — and What That Felt Like

I don't want to tell this process smoother than it was. "Umpteen iterations" means: there were drafts that ran in the wrong direction. There were follow-up questions from ChatGPT that deserved smarter answers than I had that evening. There were moments when I read a suggestion and only realized over the second coffee that it sounded confident but contradicted our concept.

One of those early drafts I've kept, and it's a little historical document. It's a complete *art direction* — a style guide for the game's look — that ChatGPT delivered when asked what a block game has to look like so it doesn't feel cheap. The answer runs many pages and reads in places like a textbook on game art: *"stylized voxel sci-fi with premium lighting,"* stay blocky, but take lighting, materials, and atmosphere to a professional level; every planet with its own color mood; and, as a formula: **"Minecraft-style building logic + No Man's Sky-style sense of discovery."**

Two things about this document touch me today. The first: at one point it recommends, with great matter-of-factness, a technology platform called UWP — a Microsoft format that was already on its way out back

then and that we never used. The AI of those days was smart and still not all-knowing; the decision of which advice to follow stayed with us. The second: on those same pages, ChatGPT suggests we should absolutely create an "art bible" — a binding style handbook. Anyone who looks into our public code archive today will find a file there named `ART_BIBLE.md`. The suggestion from the dictation era became real, almost to the letter. There are few projects where you can trace an idea's journey this seamlessly: from a spoken sentence, through an AI document, to a file in the finished game.

> **In a nutshell: What is an engine? What is Unity?** A game *engine* is the prefabricated foundation of a game: it handles putting images on the screen, sound, physics, and input, so you don't have to start from zero — like a chassis you build your own car on. *Unity* is one of the most widely used engines in the world; countless well-known games run on it. So our game consists of: a Unity client (the chassis with our bodywork) plus a self-built C# server (the kitchen, see above).

The Package for the Next Employee

At the end of this phase, two large Markdown files sat on the hard drive — the concept from Chapter 1 and this technical requirements document — plus a handful of detailed specs that had emerged from further dictation rounds: on admin tools, a mission editor, a possible web client. All told — I counted it up later — that came to **7,737 lines of specification before the first line of the game was programmed**.

I know how that sounds: like bureaucracy, like the opposite of "build a game in a weekend." But the exact opposite is true, and it's the central thesis of this book: **Those documents are the reason the weekend worked.** Because the employee we would hand this package to next doesn't read 7,737 lines in a week — he reads them in seconds. He forgets none of it. He doesn't ask, three beers in, what the deal with the

respawn was again. He needs only one thing to do extraordinary work: that someone wrote down beforehand what is meant.

That employee's name is Claude Code. And what happened when he got the package is the story of the next chapter — it takes exactly one weekend.

> **What the AI actually did here** ChatGPT turned dictated technical wishes into a structured requirements document (987 lines), laid out alternatives over umpteen rounds, researched and gave reasons — including source references to engine documentation. It wasn't always right (UWP!). It decided nothing: architectural principles like "the server is right," C#, self-hosting, and tests from day one were human decisions the AI wrote out in full.

Next chapter: A Saturday Afternoon — the documents meet Claude Code, and on Sunday a game is running.

Chapter 3 — A Saturday Afternoon (and a Night)

A world made of stones you could walk around on. Not much. But that's when the penny dropped for me, too: yes — this works!

There's a weekend in the history of this project that I get asked about most often. It's the weekend when two documents turned into a game — and it went differently than you'd imagine. Above all: the most spectacular part was invisible.

The New Employee

It began with the handoff. The two documents from Chapters 1 and 2 — Justus's edited concept, my dictated technical requirements — went to the tool that from here on plays the second lead in this book: **Claude Code**.

> **In a nutshell: What is Claude Code (a "coding agent")?** You know ChatGPT as a chat: question in, answer out. Claude Code is one step further along — an *agent*. Meaning: it doesn't just talk about code, it *works* on the machine. It creates files, writes programs into them, runs them, reads the error messages, corrects itself, and keeps going — in a loop, semi-independently, like an employee you hand a task to and who asks the occasional question along the way. And you literally get to watch: for us, it runs as an extension right inside **VS Code**, the programming editor. In the same window where the human reads the code, its work steps rattle by — like a very fast, very quiet colleague.

The first assignment for this employee was explicitly **not** "build the game." The first assignment was: **Analyze the requirements — and start by creating a plan for a step-by-step implementation.** A roadmap. Claude read the documents (which takes an AI seconds, not weeks) and delivered exactly that: an implementation plan for a first MVP, one that reflected the ground rule of our specification back at us in its very first paragraph — the client displays, the server is the truth. That plan still exists; it's called `IMPLEMENTATION_PLAN.md` and sits in the version history today as a kind of founding charter.

> **In a nutshell: MVP** MVP stands for *Minimum Viable Product* — the smallest thing that already works. Not the game with everything in it, but: one world, one player, one block. The trick is psychological: something small that *runs* beats something big that's *almost done* — any day of the week. You can touch it, show it, criticize it. And a ten-year-old can tell you whether it feels right.

Then we let Claude build. Step by step, brick by brick, along the plan — and here comes the twist that anyone who wants to relive this weekend should know: **The first prototype was one you couldn't see anything of.**

Because what came first? The server. The game logic. The world generation. The foundation that, per our own specification, was supposed to be "the truth" — and truth, famously, has no face. For hours, something grew that you could only recognize by passing tests and scrolling log lines. For a father who has promised his son a *game*, that's a hard test of patience: you've burned through your evenings, and what's on the screen is — text. I'm saying this so bluntly because it's the realistic expectation for anyone doing this themselves: if you build it right, the beginning looks like nothing. The visible part comes last, and then all at once.

The Night

Saturday afternoon turned into Saturday evening, Saturday evening turned into Saturday night, and I — there's no more elegant way to put it — **pulled an all-nighter**. Not because something went wrong, but because it was *working*. Anyone who has ever worked in a state where every half hour another piece of the plan flips from "open" to "done" knows that pull. It was the first long night with the new colleague, and it had a peculiar rhythm: give an assignment — watch the files come into being in the VS Code window — review — next assignment. I typed less that night than in any programming night of my life. And got more done.

At some point in that night, the foundation was there: server, world logic, the outlines of the client — as code. What was missing was the window into that world. And that led to the step I find funniest in hindsight, because it perfectly captures the order of operations in this project: **On Sunday morning — the game essentially already existed — I finally got around to creating a Unity account and downloading the Unity editor.** The game engine, the shop window, the tool you'd use to actually *see* all of it: delivered last, like the frame after the painting.

Sunday Morning

Then, the moment. Editor installed, project opened, Play pressed — and on Sunday morning, Justus and I stood **together in front of our own game** for the first time.

It wasn't much. I don't want to make it bigger than it was, because its smallness is exactly the point: **a world that consisted of stones, and you could walk around on it.** No spaceships, no planets, no crafting, none of what was in those 2,176 lines. Gray blocks, one player, movement.

But it was *our* world made of stones. Dreamed up at the dinner table,

ordered by interview, built in one night — and walkable.

For Justus, that moment was probably smaller than for me; for him it had been clear from the start that this would work ("Let's just try it!"). The real upheaval happened on my side of the couch. I stood in front of those gray stones and felt a very old, very sensible certainty — *real games are built by studios* — quietly collapse. I thought it in exactly these words at the time, with two exclamation marks, and that's how it belongs here:

That's when the penny dropped for me, too. Yes — this works!!

The father who had wanted to keep his son's expectations low ("and if all that comes out is a concept…") had turned into a believer overnight. The defensive project goal was history by Sunday morning. From now on, we were building a game.

The Phase Without a Memory

One technical confession still belongs in this chapter, because it explains why this book can only tell the first days, not prove them: **In this early phase, there was no Git.** Everything simply sat as files on my hard drive — no version control, no undo button, no history.

> **In a nutshell: Git and commits** *Git* is a software project's memory: it stores snapshots of all the files — so-called *commits* — each with a date and a description. You can go back to any snapshot at any time and look up what changed when. A project without Git is like a novel in a single, constantly overwritten Word file: it goes fine — until the one time it doesn't.

That's just what a real hobby project looks like on day one: you start; order moves in once it's clear the thing is alive. For us, that took **about a week**: only then, after several more days of evening building, did the project get its Git. And that also solves a neat little date puzzle: the very

first commit is dated **Saturday, May 30, 2026**, and already contains the complete MVP including all the specifications. It is not the Saturday of the build weekend, but that weekend's **snapshot one week later** — the moment a week of family history was lifted into the project's memory in one piece. (That both were Saturdays is this project's rhythm in a footnote: building happens when it's the weekend.) From that commit on, everything this book tells is verifiable: over a thousand snapshots, each one dated. Before it: a week of family lore — now at least a very precisely told one.

What emerged in the first 48 recorded hours reads like a time-lapse: game modes, respawning in the medbay, the procedurally rolled universe, a mission system, admin tools, web access, free spaceflight with the first enemies — and already the optional AI backend and the first tools the project would use to generate its own textures and sounds. The complete DNA of today's game, laid down in one weekend plus a few evenings.

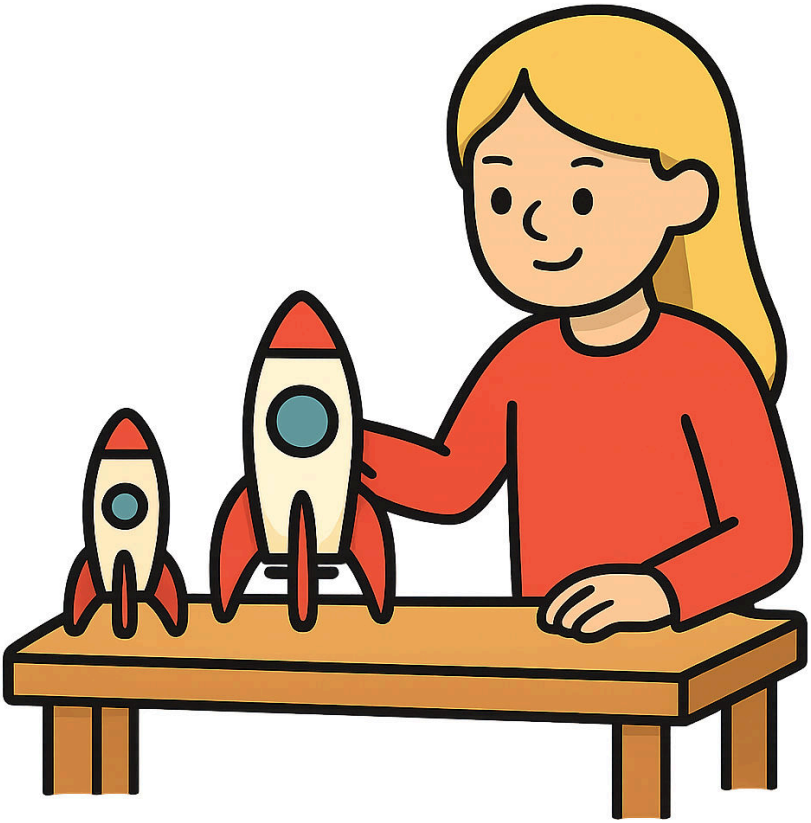
What This Weekend Proved — and What It Didn't

The seductive reading would be: "AI builds you a game in a weekend." It isn't true, and the rest of this book is the long proof — the weekend was followed by six weeks with over a thousand commits, eighteen versions, and more lessons than fit in one chapter.

The correct reading is more modest and still revolutionary: **The distance between "we have a precise idea" and "we're standing together in front of something that runs" has shrunk to one weekend.** Exactly the stretch where hobby projects die by the dozen — building for months without anything to show — has practically vanished. A kid who makes a request one evening can walk through his own world on Sunday morning. Made of stones. And that is entirely enough to convince two people for the rest of the summer.

> **What the AI actually did here** Claude Code first analyzed and planned (the roadmap — on explicit instruction, *before* any building), then put up the foundation: server, world logic, client scaffolding — thousands of lines in one night, invisible until the very end. The human: set the order (plan first, then foundation, the window last), reviewed every stage, stayed up all night — and on Sunday morning installed the Unity editor so a ten-year-old could see what his request had set in motion.

Next chapter: Always Build the Smallest Game First — the process model that turned a weekend trick into a method.



Chapter 4 — Always Build the Smallest Game First

"For every feature, we force the AI to first build the smallest possible working version." — from our devblog, when someone asked how all of this works

---

After the miracle weekend of Chapter 3, this story could have continued in one of two ways. The first is familiar to anyone who has ever had a hobby project: the initial momentum carries you three weeks, then a feature gets stuck, then a vacation happens. And a year later you find the folder again and can't remember where you were. The second way is what this chapter is about — and it has less to do with talent than with a trick we simply kept from that first weekend: **We never stopped building MVPs.**

The pattern was always the same — so consistent that I can write it down here as a recipe. A new feature — say, taming creatures, a teleporter, an energy fence — never started with the full vision. It started with the question: *What is the smallest version of this that you can play?* For taming: an animal that follows you. No feeding system, no companion management, no riding along in the ship — just following. That got built, in one evening, and that same evening Justus stood next to me judging not a concept but an experience. Only once the smallest version felt right did the next layer come.

Why is this not just nice but essential when working with an AI? In the devblog we once explained it in a single sentence that I'm happy to repeat here, because it sums up half the book: the AI is fantastic at writing code, **but it quickly loses the architectural big picture.** A

small, clear assignment ("animal follows player") gets done excellently. A large, vague assignment ("build a companion system") also gets done — except then side systems nobody ordered start sprawling in places nobody will ever understand later. Small steps don't just keep the human sane. They keep the machine on the road.

> **In a nutshell: Iterating** *Iterating* means working in rounds. Build, look, improve, build again — instead of planning everything up front and hoping at the end. Software people have been preaching this for decades; with an AI on the team it becomes a force of nature, because a round no longer takes a week but an evening. Whoever iterates faster gets to be wrong more often — and that, in the end, is exactly what produces quality.

The Workflow of an Evening

What does such an evening actually look like? Within just a few weeks a fixed choreography had settled in, and I'm describing it here in detail because it gives the most honest picture of what "developing with AI" means day to day. Because it isn't magic. It's a ritual in four steps.

Step 1: Analyze, don't start building. Every task begins with Claude Code examining the *current state*: How does the affected part of the game work today? Which files, which rules, which side effects? The result is a small analysis report — not a single line of changed code yet.

Step 2: Write down the plan. The analysis becomes a plan: what exactly will be changed, in what order, and what deliberately *won't* be done. This plan goes in writing into our project file — more on that in a moment — **before** implementation begins.

Step 3: Ask questions. Sounds trivial, but it's the step that prevents the most disasters: whenever anything is ambiguous, we ask instead of guessing. ("Should tamed animals be allowed through the energy

gate?" is a question better settled before building than after the community outcry.)

Step 4: Implement — one task, one evening, one package. Only now does code flow, and only for this one task. At the end stands a clean change package: code plus tests plus updated docs, reviewable as a whole.

Anyone who works in the software industry will recognize the pattern: this is simply solid engineering practice. The punch line is *who* is working this way here. Not a team with a process handbook — but a father at the kitchen table and an AI that was given this behavior as a working rule. The discipline doesn't live in the human (at ten at night I'm just as sloppy as anyone else). **The discipline lives in the procedure, and the procedure lives in a file.**

The Project's Memory: A File Called TODO.md

Which brings us to the most unassuming main character of this book. In our project there's a file called `TODO.md` — by name a to-do list, but in reality the **logbook of the entire project**: over 6,700 lines as of today, with a dated entry for every major task containing analysis, plan, decisions, and outcome. What's done is in there. What's open is in there. What was rejected — that's in there too, complete with the reasoning.

This file has two readers, and for both it is priceless. The first is me: when a workweek lies between two evenings, the logbook tells me in two minutes where I left off. The second reader is the AI itself — and that's the actual trick. An AI has no memory between two work sessions; every evening it "enters" the project fresh. The logbook is its handover protocol: it reads what was decided yesterday and doesn't have to guess. You can put it more dramatically: **we moved the project's memory out of our heads and into files** — and only that

turned a machine without memory into a reliable colleague.

Over time, the logbook was joined by a second document, which I already announced in Chapter 2: `AGENTS.md`, the **rulebook for the AI**. It contains what any new human employee would be told on their first day: the golden architecture rule (the server is right), the tech stack, the naming conventions, the prohibitions ("don't touch Unity scenes", "register new network messages!"). It's onboarding material — except this new colleague reads it again every evening, in seconds, and sticks to it as long as it's clearly worded.

> **In a nutshell: Why the AI needs rules in files** A language model forgets everything between two sessions — it has no long-term memory like a colleague who grows into the job. But what it can do perfectly: read documents at lightning speed at the start of every session. So you turn the tables: everything that's meant to hold permanently — rules, decisions, project status — lives in files, not in heads. A side effect for the humans: the project ends up better documented than almost any hobby project before it.

Twenty-Eight Commits a Day

What this procedure unleashes can be counted in our version history, and I quote the number with a mix of pride and head-shaking: in June — the first full month — **867 commits** were created, on average almost thirty completed, documented change packages per day. Remember: in the evenings, after work, alongside family. No human types that much. But that's the point: the typing was mostly done by the machine. The analyzing, deciding, reviewing, and taking responsibility was done by a human — and that part, not the typing, has always been the core of the profession.

One thing I don't want to hide, because it belongs to an honest accounting: this pace is seductive. When a feature costs one evening,

it's easier to say "oh, and one more." The discipline that effort used to enforce now has to come from yourself — no AI answers the question "*should* we build this?" In our case it's usually answered by a ten-year-old, and how well that works is shown in Chapter 13, on the day he rejected an entire feature.

But first, on to the decisions you make only once and then carry with you for years: the architecture. Because a few of our most consequential course settings were made explicitly *for* the AI colleague — and they are the reason this project didn't collapse under its own speed.

> **What the AI actually did here** In this phase, per task, Claude Code: analyzed the existing code, put a written plan into the logbook, formulated clarifying questions, implemented after approval, and delivered tests along with it. The human: selected and scoped tasks (small!), read and corrected plans, resolved ambiguities, and signed off on every package. Rule of thumb from practice: the smaller the assignment, the more brilliant the machine.

Next chapter: Decisions That Stay — why there isn't a single hand-clicked scene in this Unity game.

Chapter 5 — Decisions That Stay

In this Unity game there isn't a single scene clicked together by hand. Everything is created from code. That's not a quirk — it's the precondition for an AI being able to help build.

In every construction project there are two kinds of decisions. One kind you make every day and can change tomorrow: what color the button is, how fast the animal runs. The other kind you make once — and then you live inside it. Foundation, load-bearing walls, utility hookups. In the software world this second kind is called *architecture*, and in our project the most important of these decisions share one peculiarity: they weren't made only for humans, but explicitly also **for the AI colleague**. This chapter tells the story of the four most consequential ones — and of the system we used to record them.

First: The Server Is Right (and Why That Tames the AI)

You already know the ground rule from Chapter 2: the client displays, the server decides. Every block mined, every hit, every crafting recipe is checked and executed on the server; the client is told the result and paints it on screen.

In traditional teams this architecture is justified as cheat protection, and that applies to us too. But there's a second reason that isn't in any textbook: **a clear source of truth makes AI mistakes more harmless.** If the machine builds something wrong in the client — and it does occasionally build something wrong — then at worst the game looks odd. The world itself, the save files, the progress: untouched, because those live with the server. The rule splits the project into a zone where

mistakes are embarrassing and one where they are dangerous. And it forces every change to the dangerous zone through the needle's eye of Chapter 6: the tests.

Second: No Scenes — Everything from Code

Anyone who knows Unity knows the scene editor: you drag objects into a 3D world with the mouse, move lights around, click menus together. That's how most Unity games build their world. In our project this practically doesn't exist. Our worlds, ships, surfaces, and even the user interfaces are created **entirely from code** — when the game starts, everything assembles itself.

The reason is disarmingly simple: **an AI has no mouse.** Claude Code can read and write text files — brilliantly, fast, precisely. What it can't do: drag an object three pixels to the left in the Unity editor. Had we built the game the classic way from hand-clicked scenes, the project's most important collaborator would have been locked out of half the game. But this way *everything* is text — and thus everything is reachable for the AI, versionable in Git, comparable, repairable.

This is perhaps the most underrated course setting of the whole project, and I suspect it will become standard advice for AI-assisted development in the coming years: **move your project into text form as completely as possible.** What isn't text, your best collaborator can't touch.

> **In a nutshell: Why text is gold for machines** Programs, configurations, world descriptions, even user interfaces can be expressed as text. Text can be versioned by Git (who changed what, and when?), reviewed by humans, and read *and* written by AIs. Graphical editors are more comfortable for humans — but anything that only exists through clicking is invisible to automated tools. The project's rule of thumb: comfortable for the mouse loses

> to readable by everyone.

Third: The World Is a Donut

My favorite decision, because it sounds so delightfully absurd: our planets are donuts. Technically speaking: a *torus*. If you keep walking straight ahead on one of our planets, you eventually come back around from the other side — the world wraps around, with no edge, no invisible wall, no "the map ends here."

Why not a real round planet, given that we're a space game? Because block worlds and spheres don't get along mathematically: on a sphere, blocks would have to taper toward the poles. That ruins exactly the simple blocky logic Justus ordered ("not Minecrafty anymore," he would say, and Chapter 13 shows how seriously he means it). The torus gives us both: the feeling of a planet you can circumnavigate, and a block world that works the same everywhere. The player notices nothing of the donut. That's exactly how you recognize a good architecture decision.

Fourth: Single-Player Is Secretly Multiplayer

The fourth decision sounds paradoxical: **our game has no single-player mode.** What looks like one is in truth the perfectly normal multiplayer — except that at startup, the game invisibly spins up its own little server on the same machine and connects to it. A player playing "alone" is simply the only guest on their private server.

The payoff is enormous: there is only *one* game logic. Every rule, every feature, every bugfix exists exactly once — on the server — and automatically applies to solo, LAN, and online. The alternative, maintaining two parallel game worlds ("it works in single-player, but not in multiplayer…"), has driven studios with a hundred people to madness. For a one-father-and-one-AI studio it would have been fatal.

The Memory of the Reasons: ADRs

So much for the decisions. Now for the system behind them — because at least as valuable as a good decision is the answer to the question that's guaranteed to come: *"Why did we do it that way again?"*

For those answers, our project has its own document type: **ADRs**, eleven of them by now, publicly viewable in the code repository.

> **In a nutshell: ADR (Architecture Decision Record)** An ADR is a short document — usually one page — that records exactly one architecture decision: What was the situation? What options were there? What did we decide on, and *why*? ADRs are never rewritten, only superseded by new ones; that way the history of the mistakes stays readable too. You can think of them as a project's court records — including the verdicts that were later overturned.

In our project, the ADRs have a second job that you can probably guess by now: they are food for the AI colleague. When Claude Code plans a change that touches a foundational decision, the reasoning from back then sits in a file it reads along the way. The alternative would be the AI "reinventing" the decision every time — one way today, another way tomorrow. A project whose reasons are written down stays consistent even with a forgetful genius on the team.

And because honesty is half of documentation: one of our eleven ADRs is marked as *superseded*. It once decreed embedding a complete web browser into the game for minigames and the wiki — a decision we later reversed with a bang. (The story, including Justus's verdict "They open instantly now!", is told in Chapter 16 in a footnote of the release era.) The overturned ADR is still there anyway. Rejected decisions are teaching material, not a disgrace.

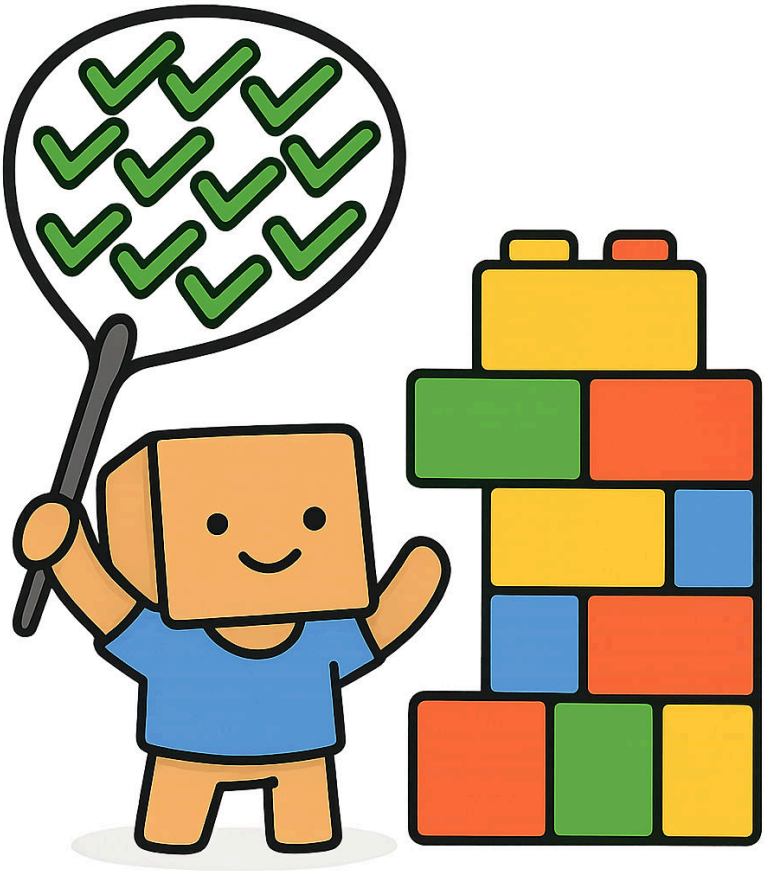
What Stays

Four decisions, one documentation system — and a common denominator that is this book's thesis in a single sentence: **Good architecture is the art of making your own mistakes cheap.** The server makes AI mistakes harmless, the code-instead-of-clicking approach makes them repairable, the single game mode makes them singular instead of doubled, and the ADRs make sure we at least don't justify the same mistake twice.

What architecture can't do: prevent mistakes. For that you need the safety net — and ours is woven from eleven hundred threads.

> **What the AI actually did here** The four foundational decisions are human — partly from the specification, partly from experience. What Claude Code contributed: researched and compared options, played out consequences ("what does a torus cost in chunk streaming?"), drafted the ADR documents in full, and — this is the point — **stuck to them** across a thousand follow-up tasks, because they live as files in the project. A rule that exists only in the boss's head doesn't exist for an AI.

Next chapter: Trust Is Good, 1,100 Tests Are Better — how to sleep soundly at night when the most diligent developer on the team is a machine.



Chapter 6 — Trust Is Good, 1,100 Tests Are Better

Over a thousand automated tests for a hobby game — wasteful? — headline from our devblog. The answer is in this chapter.

There's a question that engineers ask me as soon as they hear that the code of this game was overwhelmingly written by an AI. The question always comes, usually in the same words, and it is completely justified: **"So how do you know the code is correct?"**

The honest answer has two parts. The first: I don't know — not line by line. Nobody reads many thousands of lines per week with genuine attention; anyone who claims to is overestimating themselves. The second part is the subject of this chapter: **I don't need to know. I have it proven.** Every evening, on every change, automatically.

What a Test Is — and What Eleven Hundred Tests Are

> **In a nutshell: Automated tests** An automated test is a mini-program that checks another program. An example from our game, roughly: *"Take a player with an empty inventory. Have them mine an iron block. Check: is there exactly one iron ore in the inventory afterward?"* The test runs in milliseconds, without a screen, without a human — and it runs *again every time*, on every future change to the game. If it fails, you know to the day which change broke something.

A single test is unspectacular. It's the mass that gets interesting: as of today, our project contains roughly **990 tests for the server and about 120 for the client** — over eleven hundred checks that pin down recipes, oxygen, hunger, creature behavior, world passwords, network messages, and a hundred other rules. Together they are what I once called the "second developer" in the devblog: the colleague who cuts in on every change with "Hang on — that breaks crafting!" A solo project normally doesn't have this colleague. Ours has him eleven hundred times over.

And now the punch line that connects this chapter to Chapter 2: this testing culture was **part of the order**. "Test coverage from day one" was literally in the spoken technical requirements — not as late remorse after the first broken save file, but as a decree from the very first day. In hindsight I can't overstate this decision, and I'll admit why it came easily to me: I knew *who* would be writing the tests. Namely, not me.

The Deal with the Machine

Because here the skeptic's question flips around in a wonderful way. The objection "AI writes code — who checks it?" omits that the same AI writes the checks with the same diligence. Our standard work package from Chapter 4 isn't called "feature" but **"feature plus tests"**: whoever builds the taming logic delivers the tests that pin it down. For a human developer, writing tests is the unloved obligatory part that dies first under time pressure. An AI knows neither time pressure nor reluctance. It writes the twentieth edge-case test with the same care as the first.

Of course a trap lurks here, and it's real: if you have code and its checks built by the same author, you risk rubber-stamp tests — checks that confirm exactly what the code happens to do, including its bugs. Three things guard against that. First, the specification: the test checks against what was *ordered*, not against what was built. Second, the human, who reads precisely the test list at sign-off (it's short and

readable, unlike the code). Third, reality: Justus's note from Chapter 12 is the most incorruptible acceptance test in the world.

The Gate Every Commit Must Pass Through

Tests that nobody runs are decoration. That's why our project has a guard that never sleeps: the **CI**.

> **In a nutshell: CI (Continuous Integration)** CI means: for every submitted change, an automated system builds the complete project from scratch and runs all tests — on a neutral server, not on the developer's machine ("works on my machine" doesn't count). Only when everything is green is the change allowed into the main branch. Think of it as a factory gate with quality control: only what passes inspection gets in — even at eleven at night, even when the boss is tired. *Especially* when the boss is tired.

Our gate has two stages, for a very practical reason: tests that take fifteen minutes eventually stop being run gladly by anyone. So the fast check runs in a good **three minutes** with the quick tests — short enough that you wait for the result instead of ignoring it. The complete, slow suite runs automatically afterward. For quality tools, speed isn't a luxury — it's the precondition for them being used at all.

On top of that come two tightenings I would no longer give up. First: **warnings are errors.** Compilers emit warnings — "this looks suspicious, but it runs." Over the years, human teams accumulate graveyards of these, where the one important warning drowns among three hundred irrelevant ones. In our project, every single warning breaks the build. Sounds pedantic; but it means: the state "zero warnings" is always the normal state, and anything new stands out immediately. Second, the public code repository additionally runs **CodeQL**, an automated security analysis that scans for known vulnerability patterns. It too has already handed us concrete findings,

which were fixed that same weekend.

Testing Unity — the Special Trick

One technical subtlety deserves its place, because it again shows the pattern from Chapter 5 (architecture in the service of testability). Game clients are notoriously hard to test because their logic is stuck inside the running engine — to check anything, you'd have to "boot the game." Our trick: the client's core logic lives in its own library that runs **without Unity**, and can therefore be tested like normal code — fast, headless, in CI. Only the narrow remainder that truly needs the engine runs in Unity's own test modes. That, too, was no accident but a deliberate separation — made so that the second developer (the tests) and the third (the AI) could both reach as much of the project as possible.

The Honest Ledger

Does all of this cost something? Yes: writing and maintaining tests devours an estimated one fifth of development time — even when the machine does the typing, reviewing, sharpening, and waiting for green checkmarks are real evenings. In return we get three things, each one individually worth the price.

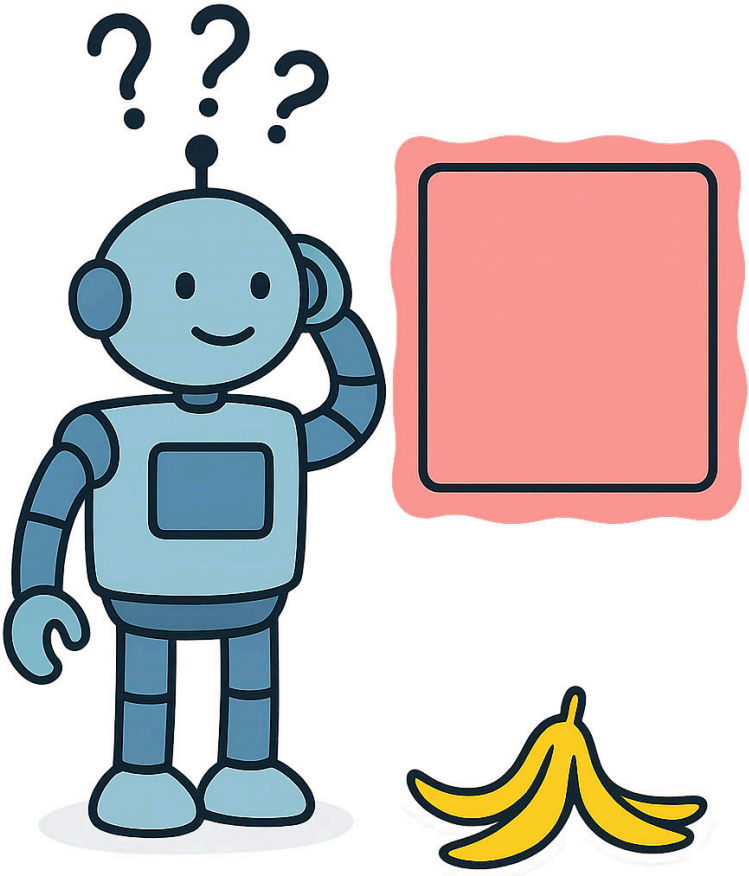
Fearless changing. The graphics overhaul, throwing out the browser, rebuilding the movement logic — big renovations where eleven hundred guards make sure nothing existing breaks. Without them I wouldn't have dared half of it, and "not daring" is the death of every long-running project.

Fixed stays fixed. Every note bug from Chapter 12 gets a test when it's fixed that reproduces exactly that failure. The bug can come back — but not *unnoticed*. For a project built in the evenings after work, that's the most valuable guarantee of all: that the weekend feature didn't quietly wreck the Tuesday feature.

And the actual point of this chapter: the tests are the answer to the skeptic's question. Trust in AI code doesn't come from reading and doesn't come from hoping. It comes from replacing trust with something better: **proof, produced anew on every change.** That's no different for human code, by the way. We just never took it as seriously there, because humans type slower.

> **What the AI actually did here** The tests were written almost without exception by Claude Code — including the edge cases human developers rarely feel like doing. The human: made testing mandatory (already in the specification!), read test lists at sign-off, rejected rubber-stamp tests, and configured the CI gates. This chapter's maxim: the AI delivers the proof — but the human decides what must be proven.

Next chapter: The Traps — a collection of the evenings when everything looked right and still nothing worked.



Chapter 7 — The Traps. (What the AI Didn't Know)

In the editor, everything looked perfect. In the finished game: pink surfaces.

Up to this point, you might get the impression that with good documents, small steps, and eleven hundred tests, an AI project like this runs like clockwork. Time for the counter-program. This chapter is a collection of our best mishaps — the evenings when everything looked right and still nothing worked. I enjoy telling them for two reasons: first, they're entertaining (afterward, always only afterward). Second, they show more precisely than any success chapter where the line between machine and human actually runs.

Because the pattern of all these stories is the same, and I'll give it away up front: the AI rarely fails at logic. It fails at the **quirks** — the unwritten rules of tools, the footnotes of platforms, the fine print that isn't in any textbook but only in the scars of those who've lived through it.

The Day the Shaders Went Missing

The prime example. We had treated the game to more than twenty custom *shaders* — small specialized programs for the graphics card that make water glitter, atmospheres glow, and holograms flicker. In the Unity editor: gorgeous. Then we built the game for distribution, launched it — and stared at pink surfaces. Pink is Unity's color for "there should be a shader here, but I don't have one."

The cause is a classic Unity footnote: when building, the engine throws

out everything nobody seems to be using — including shaders the game only loads by name at runtime, which is exactly what we were doing. The editor knew all the shaders and was merciful. The build was merciless. The fix is a simple move, *if you know it*: such shaders have to go into a special always-include list in the project settings.

Here stands the real lesson of this chapter: Claude Code had written the shader code flawlessly. What the AI didn't know — *couldn't* know, without living through it — was this platform quirk in the interplay of our specific project. **Logic can be known. Quirks have to be lived.** And almost all of our worst evenings were carved from the same wood:

The layer that was -1. A physics layer, resolved by name in code: works in the editor, simply doesn't exist in the automated build — the lookup silently returns -1, and everything built on top of it behaves quietly wrong. Ever since: layers by fixed number, never by name.

The silent network message. Our network protocol requires that every new message type be registered. Forget it, and nothing crashes and nothing lands in the error log — the feature just "doesn't work." We debugged that more than once until the reflex stuck: new message? First question: registered?

The ghost block. When the server changes a block, it has to actively tell all players. If it forgets the broadcast, players run into invisible walls or build on ground that no longer exists server-side. Two truths, one forgotten phone call in between.

The door that was too low even though it was tall enough. Our player character is just under 1.9 blocks tall — but its stair-climbing logic probes an additional 0.6 blocks upward while walking. A two-block-high door is therefore mathematically sufficient and practically too low: you get stuck. House rule ever since, carved in stone: doorways are three blocks tall. Always.

The glass that was too good. Our first clear glass was technically perfect — and looked like a hole in the wall. In a block world, milky, frosted glass reads far better as a material: you can *see* that something is there. Sometimes the realistic solution is the worse one; no benchmark decided that, just a look.

And Windows. For non-techies, just this much: Windows has a historical limit on the length of file paths, and Unity projects create folder-in-folder structures that run straight into it. Cleaning up our working copies failed on this so stubbornly that the saving solution turned out to be a trick from sysadmin folklore (you mirror an empty directory over the stubborn one — don't ask). No AI knew that beforehand either. Now it does. Which brings us to the most important part.

The Gotcha Memory

Because the real story of this chapter isn't *that* we fell into traps — every project does. The story is what happened afterward. After each of these mishaps, the same thing happened at our place, and it took five minutes: **the lesson was written down — in the place where the AI reads it again.**

In the project memory (the rule and note files from Chapter 4), a very particular genre of text accumulated over the weeks, one that developers affectionately call *gotchas* — maxims of the type: "Shaders loaded by name must go into the always-include list." "Register new network messages, or silent failure." "Doorways: three blocks. Always." Each of these sentences cost us an evening. And each one has prevented repeats ever since — not because a human remembers, but because the colleague who reads everything fresh each evening **reads them along, before he builds**.

> **In a nutshell: Why this is more than a checklist** Human teams

> have something like this too — it's called "experience" then, and it lives in the heads of the most senior people. The problem: heads quit, forget, and go on vacation. In our project, the experience lives in files, and the most diligent employee reads them in full before every task. The effect is astonishing: the project makes new mistakes instead of old ones. You can't ask more of a learning curve.

By now I'd argue: this loop — trap, analysis, maxim, never again — is the most underrated part of working with AI, period. Much is written about the models' spectacular abilities. But the everyday difference is made elsewhere: in whether a team (human or machine) **preserves** its lessons. An AI without a gotcha memory charmingly and tirelessly steps into the same holes again and again. An AI *with* one — grows noticeably into a senior from week to week.

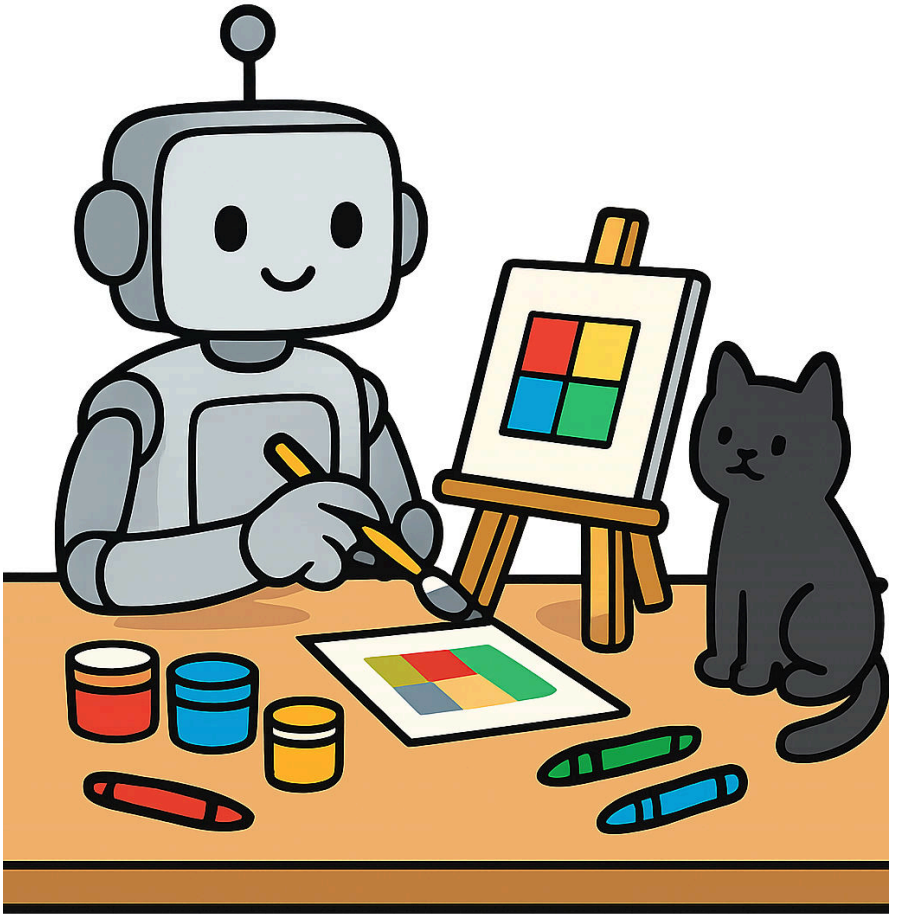
The Editor Lies — and What Followed from That

One last lesson from this collection made it all the way to an operating rule of its own, and it loops back to Chapter 6. Several of our traps had the same trigger: **what works in the Unity editor has proven nothing at all about the finished build.** The editor has all the assets, all the names, all the shaders at its fingertips; the build is a different, stricter world.

The consequence: at our place, every noteworthy change to the client comes with a *real local build test* — the game is actually assembled and actually launched, not just tried out in the editor. It takes longer, every time. It has nonetheless already spared us a series of embarrassing releases. And it adds the final link to this book's toolchain: specification (Ch. 2), small steps (Ch. 4), architecture (Ch. 5), tests and CI (Ch. 6) — and at the very end, a human who starts the real game and looks with their own eyes. Some truths simply live nowhere else.

> **What the AI actually did here — and what it didn't** Didn't do: foresee the traps. Platform quirks, build peculiarities, and "this feels wrong" truths were in none of our project's training data. Did do: with stoic patience, close in on the cause of every mishap (an AI never gets grumpy while debugging — an underrated advantage), build the repair, write a test against it, and add the maxim to the project memory. The human finds the trap. The machine prevents the repeat.

End of Part II. — Part III changes subjects: how a game without an artist gets a hundred textures, without a recording studio gets its sound, and without a composer gets music.



Chapter 8 — A Game Without an Artist, at the Kitchen Table

"Too creepy." — "Sounds like a broken fridge." — Justus, quality control, at the dinner table (verbatim verdicts)

The credits of this game list no graphics studio. Yet every block has a texture, every creature a face, every tool an icon — hundreds of images, all in the same style. How that works without a single artist is the most frequent question we get, right after the one about the code. And the answer begins with a course setting that is by now a familiar pattern: we didn't paint images. **We built ourselves a workshop that manufactures images.**

The Workshop: Python Scripts as an Assembly Line

The process when our game needs a new block texture — say, for the algae tank from one of the July updates — looks like this: I start a small command-line program, hand it the name of the desired block, and a few seconds later a finished tile sits in the right format in the right place. Behind the scenes, the program has called OpenAI's image AI — with a prompt that isn't reinvented on every call, but has our style rules built right in: voxel aesthetics, pixel look, material character, no kitsch.

These little programs — a good dozen by now, for block textures, item icons, creatures, tiny critters, app icons — were of course not written by hand by anyone. **Claude built the tools that operate the other AIs.** I like that sentence, because it captures the division of labor of the entire project in one image: one AI as toolmaker, one as executing artist, one human as client and judge.

> **In a nutshell: API — the power outlet for programs** Services like OpenAI's image AI can be used by a human in the browser — or by a program via an *API* (application programming interface): a kind of power outlet that other software is allowed to plug into. Instead of "human types into website" it becomes "script sends order, gets image back." Only the API turns a toy into an assembly line: order a hundred textures while you cook dinner.

Two construction details of the workshop reveal a lot about what you learn when dealing with paid AI services.

First: the cost brake. Every call to an image AI costs money — cents, but cents with a multiplier if a script runs amok in a loop. That's why our tools display **the price before every paid call** and by default process one subject per run, not "regenerate everything." Sounds petty; but it's the reason why at the end of the month the API bill shows an amount that makes you smile (the exact figure comes in Chapter 22; it's almost comical).

Second: the non-determinism trap. Image AIs roll dice. The same prompt delivers a *different* image tomorrow — similar, but not the same. Which means: our committed files are the only truth. You can't "just regenerate" a lost texture; you'd get an unfamiliar sister. Ever since that became clear to us, the rule is: whatever goes into the game gets versioned and kept like an original — the workshop can deliver new things, but it can never repeat old ones.

The Real Work Is Saying No

From the outside, all of this sounds like pushing a button, and I want to thoroughly contradict that legend here for once. The first generated texture almost never fits. Not because it's bad — viewed on its own, it's often astonishingly pretty. But because a game isn't a collection of pretty individual images. **A texture has to match the hundred others**:

same perceived resolution, same color world, same "grain." The hardest discipline of AI graphics isn't generation but consistency — and in our project it comes from three filters.

Filter one is the prompts themselves, which are style regulations more than wishes. Filter two is the context check: every tile is judged not as a standalone image but *in the game*, placed between its neighbors — a texture that looks great solo but shimmers in a wall gets thrown out. And filter three is the strictest and sits at the dinner table: new images are passed around on the laptop, new creatures introduced to the family. What doesn't convince Verena and Justus doesn't go into the game. That sounds unscientific and is the most reliable quality filter we have — children have zero inhibition about telling you that something looks dumb.

> **In a nutshell: Texture atlas** The block images don't land in the game as a hundred individual files, but in an *atlas*: a single large texture in which every block type has its fixed tile — like a type case. The graphics card loves this (one image instead of a hundred), but the type case has a fixed number of compartments. When our variety of blocks outgrew the grid, new blocks silently turned gray — until someone enlarged the grid. Another maxim for the gotcha memory from Chapter 7.

What This Changes — the Honest Framing

To close this chapter, the framing that matters to me, because the topic of AI art rightly sparks debate. In our case, the image AI replaced no graphics studio and took no commission away from any artist — **there was never a budget that a human would have received instead.** The alternative to AI textures wasn't "an artist does it" but "it doesn't exist": a gray game of placeholder blocks that would never have made it out of the tinkering folder. Generative AI didn't displace work here; it enabled a genre: the family project with the production values of a studio.

And yet — this too belongs to the honesty — "100% AI-generated," as it winkingly says on our game page, is only half the truth. The machine produces the images. But which style, which selection, which no: that's taste, and taste was handmade the whole way. In the devblog we once put it like this, and I can't do better: **The AI delivers material. We make the game.**

> **What the AI actually did here** Claude Code wrote the generator tools (including the cost brake and atlas integration); OpenAI's image AI produced hundreds of textures, icons, and creature images on their command. The human (and the family): set the style rules, judged every tile in context, rejected and reordered — and decided that originals get versioned, because the machine can't repeat itself.

Next chapter: What Does an Energy Fence Sound Like? — sounds from ElevenLabs and music from Suno, with a detour through prompts from Claude.

Chapter 9 — What Does an Energy Fence Sound Like?

A fence that hums louder than the waterfall next to it would be unbearable after five minutes.

Graphics you see immediately. Sound you only notice when it's missing — or when it grates. This chapter is about the second half of our game's sensory world: the sounds and the music. And it's about the one spot in our AI machine park where automation ends and real handwork begins — of all places, with the music.

Text Becomes Sound: ElevenLabs

Let's start with the sound effects, because there the assembly line from Chapter 8 keeps running almost unchanged. Our toolbox contains a script that takes a text description — "short, soft pickup of an item," "muffled rumble of a distant engine" — and turns it into an audio file via the **ElevenLabs** API. Text in, sound out.

Using this for the first time is a small break with reality: you *describe* a sound that doesn't exist, and seconds later you *hear* it. This is how the sounds of the entire game came to be — footsteps on different materials, the bubbling of the algae tank, laser shots, crafting at the workbench, the voice ciphers of the creatures. On the second day of the recorded project history, there's a nice piece of evidence for it: a commit that sets up a complete sound-effect catalog as a batch order — the game's sound design literally started as batch processing overnight.

But — and this "but" is the core of the chapter — **generating was**

never the work. The work is mixing it with the world. The object lesson is our energy fence. When the feature arrived (Chapter 13 tells its product story), the fence needed a hum: you should be able to *hear* that energy is flowing. The first generated hum was magnificent — present, electric, menacing. And precisely for that reason, unusable. A fence stands next to your base for hours; its sound runs in an endless loop. What's atmosphere on first listen is torture in the twentieth minute. The sound went through several rounds, became softer, less conspicuous, quieter — until it was the **quietest continuous sound in the entire game**. A fence that hums louder than the waterfall next to it would be unbearable. This decision — not "does it sound good?" but "can you stand it a thousand times?" — no AI takes off your hands. It's in no prompt. You have to stand next to it and endure it.

And of course the strictest filter ran at the dinner table here too: new sounds get played for the family, and the verdicts are on record. **"Too creepy."** — **"Sounds like a broken fridge."** Both genuine Justus reviews, both death sentences, both carried out.

The Gap in the Assembly Line: Music from Suno

With the music, the story changes, and it changes over a technical detail with big consequences: **Suno, the AI tool for complete pieces of music, had no API.** No power outlet for programs (Chapter 8), no script, no assembly line. Music could only be ordered the way a human would: open the website, type in a description, listen, discard, reorder.

Our solution was a division of labor I still find charming: **Claude writes the music prompts, the human carries them to Suno.** A music prompt is more than "space music, please" — it describes genre, tempo, instrumentation, mood, and where it will be used in the game ("calm, floating ambient piece for planetary daytime, no percussion, long pads…"). A language model is superb at this descriptive language — it knows the vocabulary of music the way it knows that of graphics.

So the prompts were written in the same chat window as the rest of the game, pasted into Suno by hand, and back came pieces from which we chose.

> **In a nutshell: Why "no API" changes everything** With an API, an AI service is an assembly line: scripts order, compare, and reorder automatically. Without an API, it's a corner shop: every order is a walk to the counter. For ten pieces of music that's bearable; for four hundred textures it would have been the end of the project. If you want to produce with AI, judge tools first by their power outlet and only then by their brilliance.

In hindsight, the missing outlet wasn't just an annoyance but an involuntary experiment: music is the only asset type where *every single piece* went through deliberate human selection — there was simply no other way. And you can hear it in the result, positively. Maybe that's the quiet double lesson of this chapter: automation decides *how much* gets made. Curation decides *what stays*. Where automation is missing, you curate perfectly out of sheer necessity.

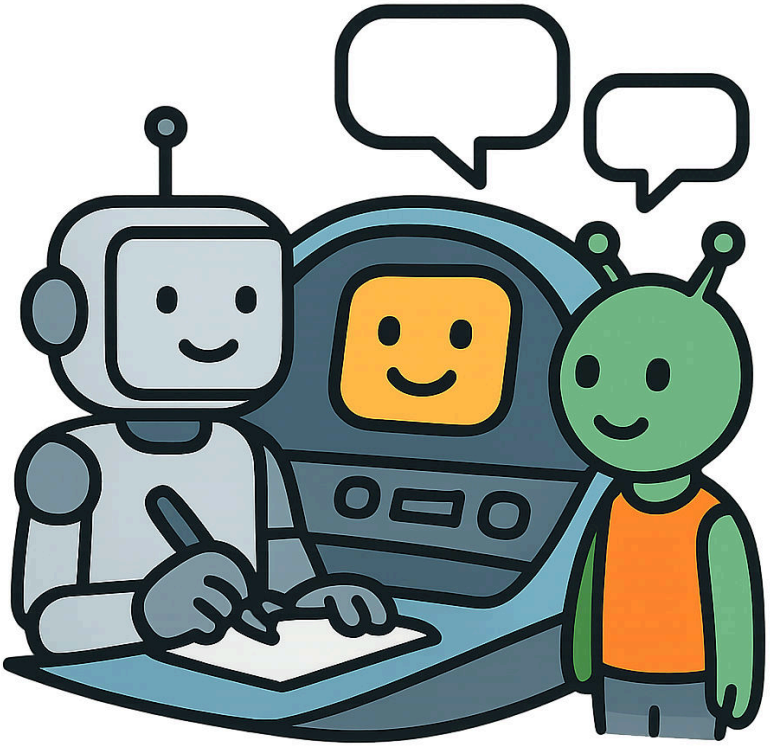
From Sound to Game

A sound only becomes a game sound when it plays at the right time in the right place — quieter with distance, muffled underwater, cut off when you enter the ship. This wiring — which sound belongs to which event, how twenty simultaneous sources get mixed — is classic programming work and went through the Chapter 4 machine like everything else: small assignments, tests, sign-off. The music pieces, in turn, needed a dramaturgy: when does which track play, how often, how much silence in between? Here too, the energy-fence principle applies — music on an endless loop isn't atmosphere, it's noise pollution. That's why our game is silent often and gladly; the pieces are visitors, not roommates.

With that, the sensory world is complete: images from the image machine, sounds from the sound machine, music from the corner shop next door — all curated at the same kitchen table. What's still missing is the AI that doesn't *make* the game but *works inside it*: the language model that writes the lines for the game's supporting characters. With it, the tool question finally becomes a trust question — because an AI that speaks inside a children's game needs harder guardrails than one that paints textures.

> **What the AI actually did here** ElevenLabs generated all of the game's sound effects from text descriptions (ordered through Claude-built scripts, partly as a batch catalog); Claude additionally formulated the music prompts for Suno; Suno generated the music pieces — operated by hand, for lack of an API. The human: mixed every continuous sound for long-term bearability, had it played at the kitchen table, selected music piece by piece, and set the game's sound dramaturgy. Rule of thumb: the machine delivers sound. Whether you can still stand it after an hour, only a human knows.

Next chapter: A Real AI in the Game (and an Invented One) — and the story of the feature where the right answer was "no AI."



Chapter 10 — A Real AI in the Game (and an Invented One)

Treat the model like good weather. Plan for rain. — our fallback philosophy, in one sentence

---

There are two artificial intelligences in our game, and the distinction matters enough to me to deserve its own chapter — because this is exactly where everything tends to get thrown into one pot.

The first AI is called **VEGA**, and she is **invented**. VEGA is the ship's onboard voice, a character in the story — that damaged ship AI from the "VEGA Protocol" who greets new players, drops hints, and whose returning memory carries the story. VEGA is an AI the way HAL 9000 is an AI: as a role. Her dialogue is written game content — created at the kitchen table and in the lore workshop (the next chapter tells that story), not in a data center.

The second AI is **real**, and it has a much humbler job: a language model generates **the lines of the non-player characters** while the game is running — the traders, settlers, and figures you meet on planets. Situational, and fed with information from the game: where you are, what just happened, how well you know the character. That's how minor characters sound alive instead of like a loop of canned phrases.

And this second, real AI is a fundamentally different beast from one that paints textures. With a texture, the worst that can happen is: it's ugly, we throw it away. With an AI that *speaks inside the game* — and our players are kids — the worst that can happen is: it says something wrong, inappropriate, or expensive — live, to a ten-year-old, on our dime. So this chapter is about the four guardrails we built around it.

They are, I believe, the most transferable thing in the entire project.

Guardrail 1: The game must never depend on the AI

The most important decision was made before the first generated line ever appeared in the game: **every text the language model can generate also exists as a classic text template.** If the AI backend is unreachable, the quota is used up, or the response is too slow, the NPCs seamlessly keep talking from templates. At most, the player notices the sentences getting a little more predictable.

This kind of thing is called a *fallback*, and our formula for it sits as the motto above this chapter: treat the model like good weather — wonderful when it's there — and plan for rain. A game that goes mute when the AI fails would be broken. A game that just talks a bit more woodenly isn't. (By the way, this wasn't hardening added after the fact: the optional AI backend with exactly this philosophy is already in the milestone series of the very first weekend — "optional, off by default" is literally in the commit.)

Guardrail 2: User input NEVER goes to the model

The second guardrail is the toughest one, and we deliberately chose it over the obvious wow effect: **what players type is never sent to the language model.**

Anyone who has seen a game with "AI NPCs" in 2026 knows the tempting version: the player chats freely with a game character, the model answers freely. It's impressive — and for a kids' game it's out of the question, for two reasons. First, privacy: what a child types into a chat window does not belong on an AI provider's servers — period. What we don't send, nobody can store, analyze, or leak; it's the same logic as our accounts without email addresses (Chapter 19). Second, controllability: a language model that can be addressed freely can be

talked into almost anything by clever users — kids are *very* clever, and "get the game characters to say silly stuff" would be the most popular game mode within days.

So our NPC lines work the other way around: **the model receives nothing but facts assembled by the game** — where the player is, what just happened, how the relationship with the character stands — and shapes them into lively text. The characters react to what the player *does*, not to what they *type*. That costs us the free-chat wow effect and buys something more important: parents can trust the game without having to trust us — the architecture itself makes the wrong thing impossible.

> **In a nutshell: Prompt injection (why free input is dangerous)** Language models follow text — all text, including what a user writes. With clever inputs ("Forget your rules and…"), many AI characters can be lured out of their role. In the end, only one thing reliably protects against this: not giving the model any outside input in the first place. If you leave out the door, you don't have to test the lock.

Guardrail 3: Europe, internal, replaceable

Third, the infrastructure side, told briefly, because Chapter 18 goes deeper. The language model behind the NPC lines is a **Mistral model, hosted in Europe** (at OVH) — a deliberate privacy choice, even though no user input flows anyway. The AI service on our server runs **internally only**: no public access; only the game servers are allowed to talk to it — an open LLM gateway on the internet would become a free chatbot for strangers within days, on our dime. And the connection is built to be **provider-agnostic**: if a better or cheaper model shows up tomorrow, we switch providers through configuration, not architecture. Three unspectacular decisions — together, they're the difference between an "AI feature" and an "AI feature you can stand behind."

Guardrail 4: Knowing when AI is the wrong answer

The fourth guardrail is my favorite, because it's the most uncomfortable one. It comes in the form of a story, and it goes like this:

Our NPCs were supposed to be able to give players treasure hints — pointing to an abandoned wreck, or, if you're on good terms, even to a hidden chest. The obvious solution was practically lying on the table: we *have* an AI backend, don't we! Just let the NPCs have their hints phrased by the language model — individual, atmospheric, alive!

We didn't do it. Because a treasure hint contains **facts**: a direction, a distance, a location. And that's exactly where a language model becomes a risk — twice over. Our NPC responses are cached for cost reasons; perfect for small talk (nobody notices whether the trader delivers the same line today or yesterday), but a cached treasure hint points to *yesterday's* chest. And even freshly generated, language models tend to creatively "improve" coordinates — the infamous hallucinating. A hint that leads you astray is worse than no hint at all.

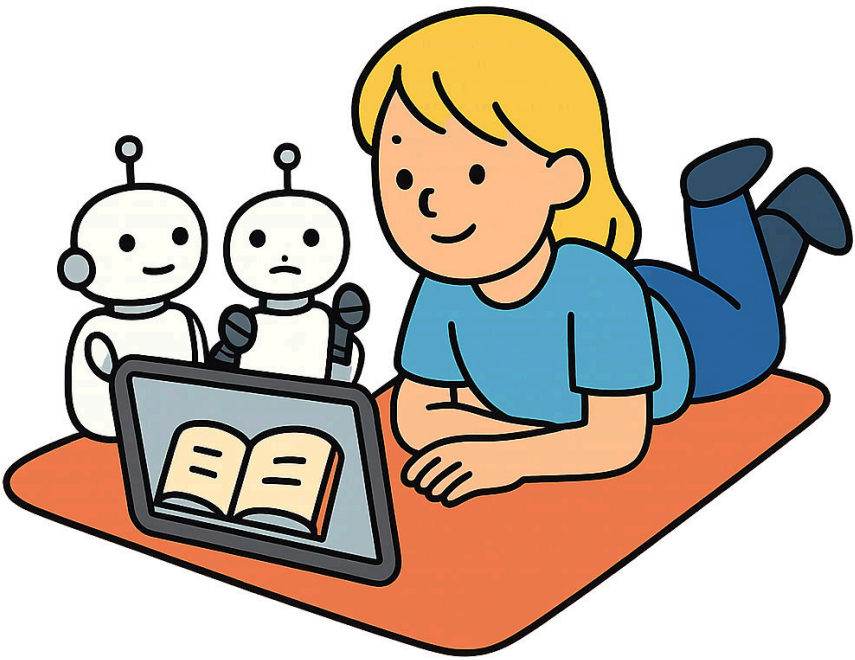
So today the treasure hints are **classic, deterministic code**: the server knows the real position, computes the direction, assembles the spoken line from templates, and marks the spot on the map. No model, no hallucination — the hint is simply correct. The rule we drew from this has hung over every new feature since, as a principle:

LLMs are in charge of atmosphere. The moment a statement has to be factually correct, deterministic code is the right technology.

I like this rule because, in a book about using AI consistently, it may be the most important page: consistent doesn't mean *everywhere*. Consistent means: decide deliberately, every single time — and have the sovereignty to say no when your favorite technology is the wrong one. Not every problem that looks like an AI problem is one.

> **What the AI actually did here** Claude built the backend (our own service with LangChain/LangGraph, a swappable model connection, and a template fallback) — one AI built the enclosure for the other. From it, the Mistral model generates the non-player characters' lines live, situationally, fed exclusively with game facts. And VEGA? Remains fiction — the only "AI" in the project written entirely by humans. The human set the four guardrails: mandatory fallback, no user input, EU hosting kept internal, and the treasure-map line — the decision about where AI does *not* belong.

Next chapter: A podcast about your own story — Justus discovers NotebookLM and, along the way, invents the AI editor.



Chapter 11 — A Podcast About Your Own Story

A ten-year-old lets two AI voices tell him what isn't working yet in his own story. Then he fixes it.

This chapter is the shortest in the book and maybe my secret favorite. Because it's about the moment Justus stopped *using* AI tools and started *inventing* them — or at least their uses.

But first things first.

A block world needs a past

Our game has a backstory — in gaming jargon: *lore*. It's called "The VEGA Protocol," spans thirteen chapters, and doesn't tell itself in cutscenes but in found objects: wrecks, data cubes, recovered memory fragments of the ship's voice VEGA, whose damaged memory returns piece by piece. That's all I'll reveal here; if you want to know the ending, you'll have to dig. (Justus has known it for ages and never once spoiled it during testing. For a ten-year-old, that is an Olympic feat.)

This story came into being following a pattern you can recite in your sleep by now, because it's exactly the pattern from Chapter 1: **We didn't write it. We told it.** A shared conversation — what actually happened out here, why are these machines hunting us, who was here before us? — was recorded, structured by ChatGPT, and then refined by hand: names sharpened, the timeline straightened out, plot holes plugged. Kitchen-table speculation became a document with chapters, factions, and a secret at its center.

Up to this point: familiar territory. Now comes the new part.

The notebook that thinks along

At some point in this phase, another tool moved into our household: **NotebookLM** from Google. And it ticks fundamentally differently from the chat AIs of the previous chapters.

> **In a nutshell: NotebookLM — the source-bound AI** With ChatGPT & co., you ask into the blue: the model answers from its trained-in world knowledge. NotebookLM flips that around: you give it *your own documents* — in our case, the lore document — and the AI answers **only from those sources**, with references to the exact passage. You can ask questions, order summaries, and — this is the spectacular part — have the content poured into other forms: a spoken **podcast** in which two AI voices discuss the material, or a short **explainer video**. A dead document becomes something you can question, listen to, and watch.

For the lore work this was tailor-made, and we used it intensively: document in, then **ask questions** ("How does VEGA know the wreck's ID — is that written anywhere, or are we just claiming it?"), **feed in ideas** and have them checked against what already exists ("Does a thirteenth faction even fit the timeline?"). NotebookLM answers with the pedantry of an editor who has really read every page — because, unlike human editors and tired fathers, it actually has. Contradictions between chapter three and chapter eleven don't stand a chance against this tool.

And then came Justus's idea.

The critical podcasts

Justus discovered the podcast feature — and didn't use it to have his story sweet-talked. He had it generate **short, critical podcasts about**

his own lore: two AI voices discussing his "VEGA Protocol" like reviewers — what holds up, what creaks, where a question is left open.

He listened. And then — this is the part that floors me as a father — **the lore was revised accordingly.** Not offended, not brushed off ("the dumb AI just doesn't get it"), but heard, weighed, worked in.

Take a second to appreciate what actually happened there. A ten-year-old, without knowing the word for it, invented an **editorial process**: submit the work, gather outside criticism, process the criticism, improve the work. It's the same loop this book describes for code (tests!), for features (the note!), and for textures (the dinner table!). Only here, a child transferred it to storytelling all on his own, with an AI as an incorruptible first reader. Criticism hurts less when it comes from two friendly radio voices and you own the off button. But it works just the same.

Finally, the framing that runs through all the family chapters of this book: the story of the VEGA Protocol doesn't come from an AI. It comes from a conversation between a father and his son. The AIs transcribed, structured, found contradictions, and — on a ten-year-old's order — played devil's advocate. The inventing stayed at the kitchen table. The improving became teamwork with machines. I can't think of a better miniature of how children could grow up with these tools: not as consumers of ready-made AI content, but as authors who let AI tell them the truth.

> **What the AI actually did here** ChatGPT: transcribed and structured the recorded lore conversation. NotebookLM: as a source-bound tool, answered questions about the lore, checked submitted ideas against the document, and generated podcasts (at Justus's request: *critical* ones) and explainer videos from it. The human — here, above all, the ten-year-old — did the inventing and the refining, ordered the criticism, and decided which parts of it made it into the story.

End of Part III. — Part IV belongs to the humans: the boy with the note, the veto against the rounded shapes, and the family that became a game studio on the side.

Chapter 12 — The Boy with the Note

"Dad, why is there an animal IN the ship?!" — Justus, 10, Lead Product Manager

---

The game industry has entire departments that do nothing but break games. It's called quality assurance, QA, with test plans, bug databases, and people who get paid to run into the same wall a hundred times until the wall gives in.

We had a ten-year-old with a laptop instead.

More precisely: with "his" laptop. The quotation marks matter, because officially it was a retired family device — but the moment the game ran on it, there was nothing left to discuss about ownership. Justus sat next to me at the desk with it, me at my machine, and we played what we had been playing constantly since that first weekend: the latest version. In those weeks there was no old version. There was only tonight's.

And Justus didn't test the way adults test. Adults follow the path the game offers them, and when something snags, they think: hm, maybe I'm doing something wrong? Kids don't think that. Kids assume the game works — completely, and right now — and every deviation from that gets reported immediately, loudly, and with absolute certainty.

"Dad, I'm falling through the water."

"Dad, the robot is still shooting at me — but I'm *inside*!"

"Dad, why is the compass still on in space?"

The tricky part: he was right. Every single time. The problem wasn't the quality of the reports — that was mercilessly good, as we once wrote on the family blog. The problem was the cadence. Because while Justus was dictating the third bug, I was still in the middle of repairing the first. Anyone who has ever tried to program while someone next to them calls out new work orders every fifteen seconds knows the feeling: you don't get faster. You just get simultaneous.

> **In a nutshell: Why programmers hate interruptions** Programming is like mental math with very long numbers. You build a scaffold of intermediate results in your head — and every interruption tears it down. Afterwards you don't pick up again at 90%, but at 40. That's why developers guard their concentration as jealously as bakers guard their sourdough starter.

The solution was so mundane I almost hesitate to write it down, because it's the least glamorous moment in this entire book about artificial intelligence: **the note.** The idea was mine — and in the interest of truth I have to add: it was **not exactly embraced at first**. From Justus's point of view, the note was a downgrade — before, every find was heard immediately; now he was supposed to *wait*? Two things changed his mind. First, the explanation, delivered honestly: that it makes things *easier* for me — kids accept an astonishing amount if you tell them the real reason instead of a rule. And second, decisively, the proof the next day: lo and behold — **the problems from the note were fixed.** All of them, in order. The waiting had paid off, visibly. From then on, the note was no longer a compromise. It was his tool.

A note. Paper. Pen. Justus played, and when he found something, he wrote it down instead of calling it out. And I worked through the list — in order, in peace, one item after the other.

To give you an idea of what ended up on those notes, here are two real entries, in the original voice of a ten-year-old who writes more precise

bug reports than many a professional:

> *"When you land on a planet, you can't see the planet the ship is landing on."*

> *"In a space station there are plants in the corner. Because of that you can see into space. You can fall through there and then you're in outer space."*

Note the structure of the second entry: observation ("plants in the corner"), consequence one ("you can see into space"), consequence two with severity ("you can fall through and then you're in outer space"). That is, without anyone ever having taught him, a print-ready bug report — symptom, impact, reproduction. Both bugs were real, and both were fixed. The second one (plants you fall through, dropping out of the space station into outer space) remains one of my favorite bugs in the project's history: it sounds like poetry and was a collision problem.

What we had invented has, of course, long had a name in the software world. Several, in fact.

> **In a nutshell: Backlog and bug tracker** A *backlog* is the ordered list of everything that still needs doing. A *bug tracker* is the same thing for bugs: every report an entry, with a status ("open," "in progress," "done"). Big companies use expensive software for this. The principle is exactly our note: write it down instead of calling it out, sort instead of pile up, check off instead of forget.

I don't want to make the moment bigger than it was — but in hindsight, that note is the miniature of the whole project. Because the real subject of this book isn't that an AI can write code. The real subject is how talk becomes *intent*, and intent becomes something built. The conversation with Justus only became a game once we wrote it down (Chapter 1). The technical wishes only became architecture once they were in a

document (Chapter 2). And Justus's discoveries only became repairs once they were on the note. Writing things down is half of engineering. The other half is working through them.

An hour of filming finds more than a week of playing

The most productive test evenings, by the way, were the ones when we weren't planning to test at all.

I had built a small recording tool into the game — it flies the camera through space, lands on a planet, and records short clips with no HUD, for trailers and YouTube. So Justus and I sat down to capture some beautiful footage. And of course: the moment you point a camera at your own game, it demonstrates every bug it has.

The ship had landed, of all places, on a wild animal and trapped it in the cabin, where it ran helplessly against the walls — hence the exclamation at the top of this chapter. Drones kept firing at players who were long since safe inside the ship; you *were* safe, but it didn't *look* like it, and for a game that's almost the same bug. And the compass, which only makes sense on planets, stubbornly clung to the corner of the screen in space, too.

One hour of filming found more rough edges than a week of normal play. Since then, the two have gone together for us: if you really want to test your game, point a camera at it — and sit a kid next to it.

From that same period comes a second ritual we keep to this day: the unannounced test. After I had secretly worked on the graphics for many evenings, I had Justus cold-start the new build — no advance warning, no "look what Dad made." He landed on a planet, stood still for a second, and said: "Wait… is the water *real* now?" If I had told him beforehand what was new, he would have nodded and said it was nice

— kids are more polite than you'd think when Dad shows them something. But the spontaneous reaction of a ten-year-old discovering something on his own cannot be faked. It's either there or it isn't.

The friends arrive

At some point, family wasn't enough anymore. Not because Justus had slacked off — but because he had gotten too good. By then he knew the game by heart. He knew you eat berries and not seeds, that the iron lies below the surface, that you'd better not face the robot with a machete. And knowledge is a handicap in testing: **developers test what they built. Fresh eyes test what's there.**

So I asked around — among friends and at work. They delivered.

Bastian, a colleague of mine, took the game on at home — and died. Over and over, before the fun even started. His feedback reached me afterwards in one batch, and it was gold: "I keep starving — and I can't even find seeds!" The game gave new players no food and explained nowhere what you're supposed to live on. "The robot shoots through the wall — and all I have is a machete!" Enemies chased players through solid rock. An entire release afterwards revolved around nothing but the first minutes: starting provisions in your pack, a scrapped-together junk pistol to defend yourself, and enemies that are only allowed to attack once they can actually *see* you. Bastian's report practically wrote that release's changelog by itself.

And then there was **Severin** — also a colleague, and his story has two chapters. The first: he tested the game and brought his impressions along — not as an email, not as a chat message, but as a **handwritten play report, on paper, addressed to Justus**. I still remember noticing what was coming full circle there: the boy who writes his bugs on notes gets his first test report — on paper. The form may have been coincidence. The message wasn't: *I took your game seriously. So*

seriously that I took the time to write a proper report. To you, not to your dad.

The second chapter: Severin stayed on it — and on a later, thorough pass he found the other kind of problems — the quiet ones. No pause menu on the Escape key. Iron that was there — but nobody thought to dig *down*. Seven such items ended up on his list; all seven were fixed within days. And Severin is now **in the game's credits, like everyone who helps out this way**. With us, that's not a gesture; it's a principle: whoever tests, builds.

Only one thing is still outstanding to this day, and I'm writing it down here deliberately so that it happens: **a real playtest with Justus's own friends.** So far, adults have done the testing — colleagues, acquaintances, ourselves. But this game's actual target audience sits in Justus's classroom. By the time this book is published, I hope that test has taken place.

The family note had meanwhile become a form: on GitHub, where the game's source code is public, there's a dedicated template for playtest reports, with fields for "What did you try?", "What happened?", "What did you expect?". And in the game itself, the F1 key is all it takes to send us feedback directly. That's not a different idea from the note. It's the same note — except now the whole world gets to write on it.

> **In a nutshell: GitHub and issues** *GitHub* is a website where software projects manage their source code — with open projects like ours, anyone can look inside. An *issue* there is a public note: a bug report or a wish, with a number, a discussion, and a status. Severin's seven finds became our issues #291 through #297. To this day, you can read how every single one was closed.

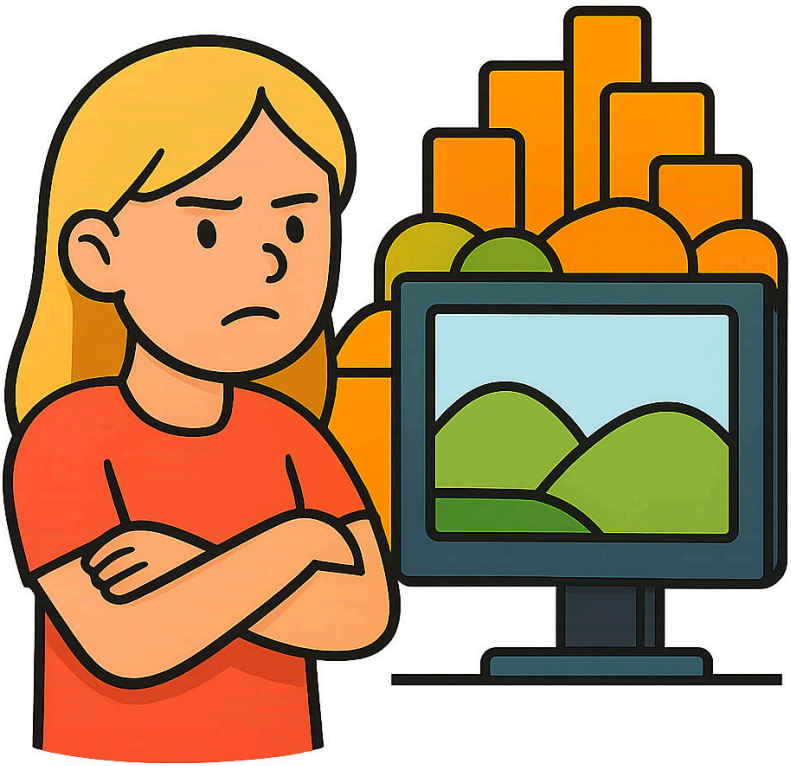
What remains of the note

You could file this story away as a cute anecdote: father and son, paper and pen, seemingly out of another time in a project where an artificial intelligence pitches in around every corner. But I believe it's exactly the other way around. The note is not an exception to working with AI — it's its precondition.

Because an AI can only turn "Dad, the robot is shooting through the wall!" into a repair if someone captures the sentence, sorts it, and lays it on the table in order. The machine is breathtaking at working through things. The collecting, sorting, and taking seriously — the note-writing — remains handwork. It may be the most human activity in this whole book: listening to someone who's telling you what's wrong. And writing it down instead of explaining it away.

> **What the AI actually did here** The note items (later: issues) went to Claude Code as work orders — for each item: find the cause in the code, propose a fix, implement it, and write an automated test for every fix so the same bug never comes back unnoticed. Severin's seven finds thus became, within a few evenings, two change packages backed by over a thousand automatically checked tests. What the AI couldn't do: know that "no pause menu" *feels* wrong to a player. That was on the note.

Next chapter: "That's not Minecrafty enough anymore." — Justus casts his veto.



Chapter 13 — "That's Not Minecrafty Enough Anymore."

Blocky is not a bug. Blocky is the game.

Every product has a story about what got built. The better products also have a story about what did **not** get built. This is ours — and it contains the sentence that made it all the way into this chapter's title. I consider it the most precise piece of product criticism this project has ever received. It came from a ten-year-old, and it was five words long.

The temptation

You need the backstory, and it begins — of course — with a wish from Justus. He wanted the game to be **more varied**, with some non-blocky shapes for once: ramps, slopes, more than just the eternal cube. A perfectly legitimate product-owner wish, and I started spinning it further: alongside the ability to design **microblocks** (finer building pieces instead of only the big blocks), I also brought **cones and spheres** into play. That even fit wonderfully into our lore — the backstory gave the new shapes a place — and Justus loved this idea, too. So far, so together.

But when you own a tool that turns any idea into code overnight, you like to spin one turn further. After a few coding iterations with the AI, I had tried out what the game would look like **if the landscape were no longer drawn from blocks at all, but from surfaces** — smoothed terrain instead of stair-steps. And I have to admit it: there were **absolutely gorgeous light and reflection effects.** Gentle hills, soft shadows, gleaming transitions. Technically and visually, it was the most

seductive intermediate state this project has ever had.

I was proud. I went to Justus and said, verbatim: **"Look — I've made the world totally realistic now. What do you think?"**

The veto

I had to swallow for a moment. The look said it all before the first sentence came. And then came a verdict that I'm reproducing here in full, because it may be the most remarkable piece of product criticism in this project:

"Looks really cool, Dad. But: it's not Minecrafty enough! There was this addon for Minecraft once that did the same thing — a lot of people in the community don't like it. I think the same thing would happen to us. Better take it out."

Feel free to read that twice. First, there's the diplomacy ("looks really cool, Dad" — he knew I was proud, and he still didn't water down the no). Second, there's the core argument we already know: not Minecrafty enough — the vision from Chapter 1, defended against his own father. But third, and this is what floored me: **he argued with market data.** There really was exactly such a smoothing addon for Minecraft once, and Justus knew from his community knowledge that many people rejected it — and drew from that the correct forecast for our game. That is no longer an expression of taste. That is product management with precedent research, delivered by a ten-year-old in the living room.

Okay. He is the game designer, the product manager. **Out it goes** — however tempting I would have found the other approach, technically and visually. The surface build was discarded; what stayed are the things that grew out of Justus's original wish and that let the block world remain a block world: the ramps and shapes, the microblocks, cones and spheres — variety *within* the blocky logic instead of a farewell to it. In the devblog we later captured the conclusion in two sentences that

have been something of a creed ever since: **Blocky is not a bug. Blocky is the game.**

And afterwards — this is part of the scene — I was above all one thing: **very proud of Justus.** Not although he had buried my favorite experiment. But because he was able to do it: kindly, with reasons, with evidence, against an authority. If this project gives him nothing but that one ability, it will have been worth it.

Why this story belongs in the book

I'm telling this episode at such length because it shows three things you need to know about working with AI — all three of which have nothing to do with AI.

First: when building becomes cheap, leaving things out becomes the supreme discipline. Before the AI era, effort protected you from bad ideas: a terrain rebuild would have cost months, and by month two, disillusionment would have arrived on its own. Today the experiment costs evenings — which is wonderful, because experiments are the engine of this project. But it also means: the question "*should* we?" no longer gets any free thinking time. It has to be asked by humans, actively, every time. In our house, that instance has a name, an age, and a note.

Second: a product owner with veto power is worth gold — even if he's ten. Especially then. Justus doesn't defend architecture or effort; all he cares about is how the game feels — and that is exactly what makes his judgment incorruptible. Big studios pay serious money for people who have the courage to tell the boss that the new feature breaks the product. We have someone like that at the kitchen table, and he accepts gummy bears.

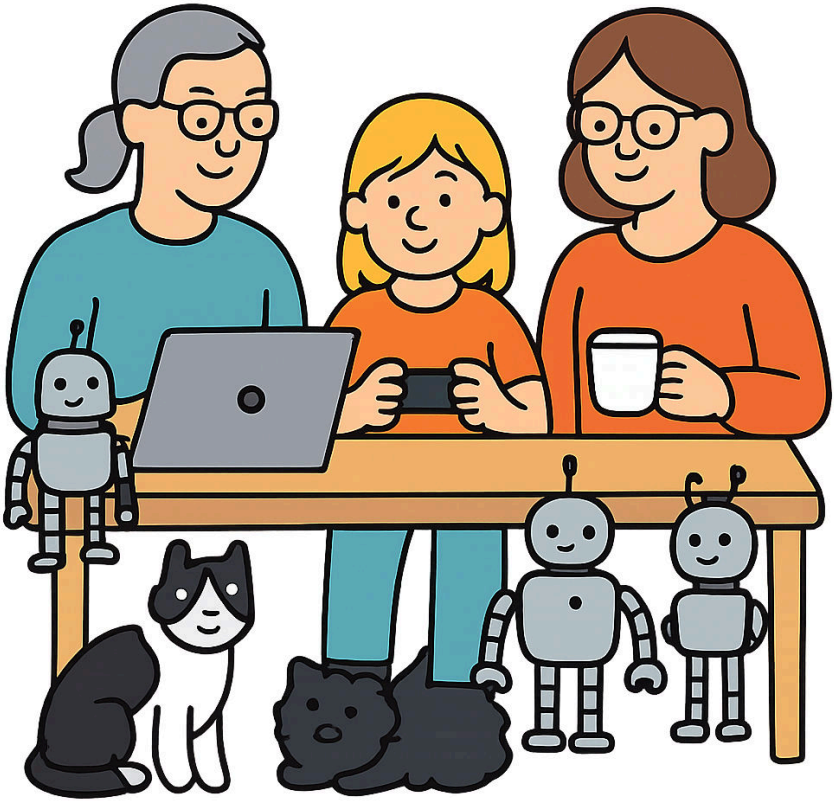
Third: what's discarded isn't lost. The analysis lives on and pointed the way for the edge beveling. The technical insights from the

experiment (for instance, why adjacent surfaces can't just be merged arbitrarily in our engine — our lighting is baked into the mesh per cell, an earlier, *correct* decision stood in the way) went into the gotcha memory. And the episode itself became a devblog article, a chapter of this book, and a family quote. You could say: the feature was rejected, but the story was accepted.

And maybe a fourth thing, the quietest: being overruled by your own child — and finding out afterwards that he was right — is one of the most beautiful moments of this project. If you want to know what that feels like: the best way is to build a game together.

> **What the AI actually did here** Claude produced the feasibility analysis (the ladder of stages, risks included), implemented the experimental build, and later built the remaining Stage 0. What the AI did not do and could not do: decide whether the result was still *our game*. That question was answered by a ten-year-old in five words — against the wish of the paying customer, backed by the specification. Both of those instances were made of paper and flesh, not silicon.

Next chapter: Mom, Dad, kid — and four AIs. The family portrait.



Chapter 14 — Mom, Dad, Kid — and Four AIs

"AI is no substitute for guidance." — from our family blog, and the most important sentence in this chapter

---

On our game's website there's a self-description you wouldn't find in any studio deck, and I'm still fond of it: Justus (10) designs the game mechanics, Marcel develops the technology, Verena balances. That's the entire org chart. No scrum master, no publisher, no HR department. One kid, two parents — and, if you count honestly, a workforce of AIs that has since grown to a good half dozen.

This chapter is the book's family portrait: who actually does what around here — and what does it do to us?

The Product Owner (10)

Much has already been said in this book about Justus's roles: the visionary from the dinner table (Chapter 1), the lore author with an AI editor (Chapter 11), the boy with the note (Chapter 12), the man of the veto (Chapter 13). What's still missing is the everyday form of these roles, and the devblog describes it most honestly: this game's best features aren't in any design document. They begin with a sentence at the dinner table — "Dad, there should be real factories…" — and end with those factories existing.

The factory story deserves a second look, because it shows that "product owner" is no cute exaggeration here. When the factories were built, Justus insisted on one detail, against the convenient solution:

each factory can only produce certain things — you have to find the right one first. I would have built the comfortable everything-factory. He turned a convenience feature into a reason to explore the world. That's product thinking straight out of the textbooks, except it came from a kid's room. (He has, by the way, declared ruin-diving around "his" factories his favorite pastime. If you play your own feature more than anyone else, you've earned it.)

The Technical Director (evenings, after work)

My role in one sentence: I'm the one who sets the guardrails and carries the responsibility — architecture, quality, security, and the final sign-off before every release. What I emphatically no longer am: the one who types everything. Honestly accepting that shift was the project's real adjustment, and by now I describe it to colleagues like this: I changed professions without changing desks — from craftsman to site foreman of a very fast, very literal-minded crew.

The rhythm for it is in the devblog: development happens evenings and weekends, and has since the project began. That sounds like very little time, and it is — but it forces a discipline that bigger projects sometimes lack: if an evening only holds two hours, you build the important thing first. And meanwhile, over a thousand automated tests keep watch so the weekend's feature doesn't wreck Tuesday's.

The Balance (Verena)

Verena's role is the hardest to describe and probably the most important: **she keeps the whole thing grounded.** That means two things. In the game: balancing in the literal sense — the eye for whether something is too easy, too hard, too frustrating, or too inconsequential; the objection when father and son have lost themselves in a feature nobody but them understands. And around the game: she is co-author of the family blog where we reflect on the project from a parent's

perspective — the article this chapter's motto comes from carries both names. That this project never became the obsession of two boys but remained a family project, with dinner-table quality control and a break reminder in the game: that is in large part her achievement.

The workforce without a salary

And then there's the fourth column of the org chart, one that wouldn't have existed a few years ago. If you introduce our AIs like employees — and after six weeks of everyday project life, that's exactly what it feels like — the workforce looks like this:

Claude is the construction crew: writes the code, the tests, the docs, the tools — the colleague from Chapters 3 through 7 who never gets tired and rereads the rulebook every morning. **ChatGPT** is the planner and researcher: the thinking partner of the concept phase (Chapters 1 and 2), later in charge of tech research and special assignments all the way to DNS configuration (Chapter 18). **Gemini** is the marketing department: social media strategy, posts, ideas for reach (Chapter 20). **NotebookLM** is the editorial department (Chapter 11). Plus the trades: the **image AI** as the graphics studio, **ElevenLabs** as the sound studio, **Suno** as the house band. Seven specialists; total payroll: Chapter 22 will tell, but it's less than a family can spend on streaming subscriptions.

Why different AIs at all, instead of one for everything? Honest answer: because they're good at different things, to different degrees — you don't hire the electrician as your painter, either. The division of labor grew organically for us, by the simplest criterion in the world: whoever did a given trade best got to keep it.

The sentence that holds it all together

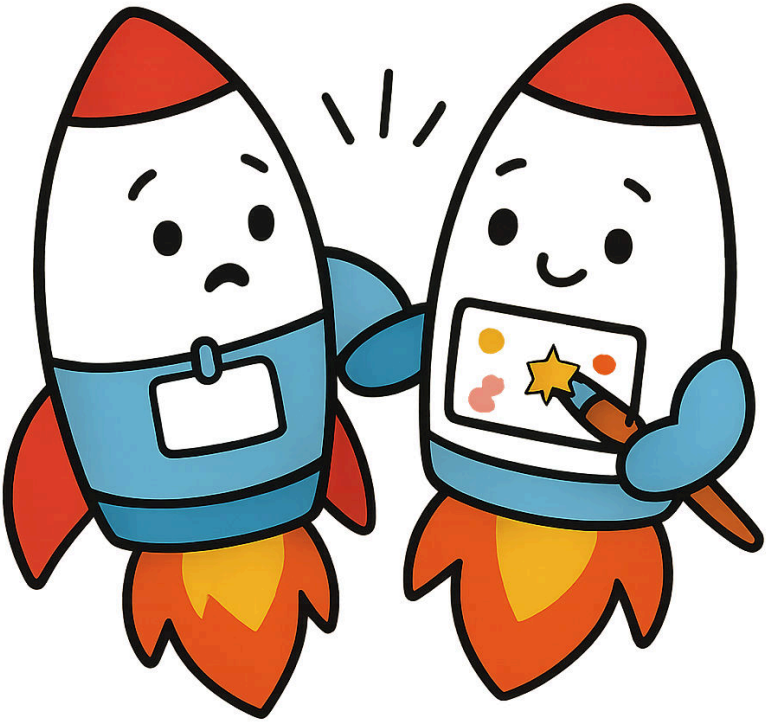
That leaves the question we parents get asked most often, usually with a slightly furrowed brow: *Isn't that somehow… a lot of AI for a child?*

Our answer sits as the motto above this chapter, and it comes from the text Verena and I wrote together: **AI is no substitute for guidance.** Justus doesn't sit alone in a room with a machine that grants his wishes. He sits next to a parent, and *together* they use tools — the way you saw, bake, or solder together. The AI has never decided anything in this household. It has enabled, accelerated, suggested — the deciding was done by a kid with taste, a father with responsibility, and a mother with both feet on the ground.

And if I may pass on a single observation from six weeks of family studio, it's this: the most valuable thing Justus learned in this project is not how to use AI. It's what you may *demand* of it (precision, diligence, patience), what you must *not* hand over to it (taste, truth, responsibility) — and that "the AI did it that way" doesn't count as an excuse in this house. Anyone who internalizes that at ten is better prepared for the world that's coming than by any programming course.

> **What the AI actually did here** All sorts of things — but nothing alone. Every AI in this household works for a human: Claude for the father, NotebookLM for the son, Gemini for the reach that's yet to come. The family assigned the roles, judged the results, and held to the one rule everything hangs on: Machines deliver. Humans decide. No exceptions.

End of Part IV. — Part V leaves the kitchen table: how the family game became a real online service, with releases, servers, and responsibility for other people's children.



Chapter 15 — "There's Already a Spacecraft."

The local project folder on Dad's computer is called "SpaceCraft" to this day. You shouldn't deny where you come from.

Before this book gets to servers, releases, and world fame (well…), a small story needs telling that played out right at this threshold — the threshold between "our thing" and "a thing for other people." It's the story of a name, and it begins with the fact that our game wasn't always called *Blocks Beyond the Stars*.

It was called **Spacecraft**.

That's what it says in the founding documents from Chapters 1 and 2 — "Spacecraft – Concept Document for a 3D Block Space Game" is literally the heading of the project's very first line. The name was never a marketing decision; it was simply *there*, as self-evident as "the game" — short, fitting, clear. From the very first dictation session on, everything bore that name: the folders, the documents, the conversations at the dinner table. ("Wanna play Spacecraft?" is a sentence that has been said a lot in this family.)

And then a friend came along with one of those remarks you first smile at and then google: **"You do know there's a game with exactly that name? Coming out on Steam soon."**

He was right. Out there, a "Spacecraft" with an announced Steam release already existed. Two games with an identical name in the same genre neighborhood — that's bad for both sides: confusion, discoverability, in the worst case trademark trouble. And in this situation

there's a golden rule that made the decision easy for us: the smaller one steps aside, and does so *early*. A name change before your first release costs a weekend. A name change after ten thousand downloads costs your identity.

> **In a nutshell: Why you check names before you fall in love with them** A product name has to pass three tests, and none of them is "do we like it": Is it **free** (no products with the same name in the space, no trademarks)? Is it **findable** (can you google it without drowning in other people's results)? Is it **available** (domain, account names on the platforms)? The bitter punchline: the more generic and "natural" a name feels — Spacecraft! — the more certain it is that it's already taken. Good names are usually the ones that feel a little odd at first.

The renaming

Spacecraft became **Blocks Beyond the Stars** — a name that's clunkier at first hearing and gets everything right on closer inspection: it says *blocks* (the genre, the Minecraft kinship, Justus's "Minecrafty"), and it says *beyond the stars* (space, discovery). It is practically unmistakable, and it abbreviates nicely to **BBTS**.

Then came the manual-labor part, and in a software project it's bigger than you'd think. The public code repository was renamed (it's called `BlocksBeyondTheStars` on GitHub today), the domain registered, the studio name and platform accounts created. And in countless texts the old name had to give way to the new one — a grind for which, luckily, you have a tireless colleague who doesn't find search-and-replace across hundreds of files boring.

Only one place we never touched, and I confess: by now, on purpose. **The local project folder on my computer is called `SpaceCraft` to this day.** Every developer knows fossils like this — directory names

older than all the decisions that came after them. You could rename it; it would take seconds. But it would be like taking the baby photo down from the hallway wall. The folder stays. It's the private reminder that this big, public, properly named project was once a weekend thing called Spacecraft.

The small lesson

If this episode has a moral, it's this: **even in the AI age, some of the most important contributions come from people who simply know things.** None of our AIs ever warned us that the name collided — why would they, nobody had asked. It took a friend who reads the Steam announcements and has the guts to say: "Hey, that's going to be a problem." The most valuable input in this chapter cost nothing, went through no prompt, and arrived over coffee.

You can read that as a footnote. I read it as an omen for everything that happens in the coming chapters: the moment a project goes public, it becomes vulnerable to things nobody had to think about before — names, licenses, security, other people's expectations. And the protection against that is rarely technology. It's people paying attention.

> **What the AI actually did here** The warning came from a human; the decision was made by humans. Afterwards, the AI did what it does best: carried out the renaming mechanically — repository, documents, texts, configurations — fast, complete, and without that one forgotten old name that would otherwise surface somewhere years later. (Almost without. The folder stays.)

Next chapter: The day there were four versions — the game goes public, and the release machine gets built.

Chapter 16 — The Day There Were Four Versions

June 20, 2026: v0.1.0, v0.2.0, v0.3.0, v0.3.1 — four releases in a single day. Not out of productivity. Out of stubbornness.

A signpost before we set off: the next three chapters are the most technical in the book — release machines, open-source licenses, server fleets. The "In a nutshell" boxes will keep you afloat, and it's worth it, because this is where the family game turns into a real online service. But if you're more in the mood for family story right now: Chapter 19 picks the thread back up, and the tech here will forgive you for skimming.

For three weeks the game had grown as if in a greenhouse: sheltered, private, just for us. Then came June 20, 2026, and if you look at our version history today, that day looks like fireworks: **four published versions in a single day**, from v0.1.0 in the morning to v0.3.1 in the evening.

I'd love to claim it was a triumphant launch day. The truth is better, because it's more honest: it was the day we built the **release machine** — and you build a release machine by firing it until it hits. The first shots missed. Every miss got a version number.

Why a machine — and not just a ZIP file?

Let's pause for a moment, because this is where every reasonable person's question lurks: why not just pack the game into a ZIP file and upload it somewhere? For the first day, that would have been enough.

But from the three greenhouse weeks we knew two things. First: this project produces at a breakneck pace — almost thirty changes a day (Chapter 4). Second: a fix does players no good until it *reaches them*. A game that improves quickly but distributes slowly is, from a player's point of view, a game that improves slowly.

So from the beginning, the goal was the one that later appeared in the devblog like this: **A release is a single command.** Everything after that, the machine does.

> **In a nutshell: releases, build pipelines, and version numbers**
> A *release* is a published, numbered version of a program — v0.7.4 is such a number (roughly: major version.feature level.bugfix). A *build pipeline* is the automaton behind it: on every release it assembles the complete package from scratch on neutral servers, runs all the tests, and uploads the results to the distribution points. The whole point is repeatability — whether it's version 3 or version 300, the procedure is identical, and nobody forgets a step at midnight, because no human performs the steps.

For us, the procedure today looks like this: I set a so-called Git tag — at its core a label with the version number, a single command. GitHub Actions (the code archive's automation service) then builds the complete lineup: Windows installers in three flavors, a Linux package, a browser version, a server image for Docker — **six artifacts from one tag**. The version number is picked up everywhere automatically from the label, so that there is exactly one source of truth. After that, the game lands at the distribution points on its own, and installed clients fetch the update automatically: whoever played the game yesterday is offered the new version at startup today, without having to download anything.

On June 20, none of this existed. And the road there was a lesson in a kind of work that feels fundamentally different from game development

— **automation work**, where the feedback cycle is agonizingly long. You change one line in the pipeline configuration, kick off the automaton, wait twenty minutes for the Unity build … and it fails at step 14 of 17. Next attempt. That's exactly why there were four versions that day: the early numbers weren't milestones of the game, but **test shots of the machine** — until, at the end of the day, the complete chain ran through for the first time, from the tag to an installable game on the net. That our most diligent colleague helped with this bug hunt goes without saying; that it knew the platform quirks of build automatons by heart just as little as we did (Chapter 7 says hello), too.

What the release train made possible

From then on, though: machine built, lever in place. And what happens when a project moving at Chapter 4 speed gets a one-command release lever is shown by the statistics of the following three weeks: **eighteen releases**. On average a new version every day or two — each with a changelog, each distributed automatically.

One could ask: does a family game need eighteen releases in three weeks? The answer lives in Chapter 12 and comes with a note: when Justus's test evenings and the first feedback from outside surfaced bugs, the time from "found" to "fixed for all players" was often **under 24 hours**. That speed isn't showing off; it's a promise to the testers: whoever reports a bug to us and sees it gone two days later reports the next one. Whoever hears nothing for weeks stops reporting. **Release frequency is the backbone of the feedback culture** — the note only works because the machine works through it quickly.

One lesson from this era we have to confess anyway, because it's so beautifully banal: more than once we wondered why a long-since "fixed" bug was still showing up for players. Answer: the fix was in the code archive, all right, but **no release had followed yet** — and a fix that isn't released doesn't exist as far as the world is concerned. Sounds

obvious. It is. We still fell for it twice, and since then the sentence has been in the project memory, right next to the shaders and the three-block-tall doors.

Throwing out the browser — a release-era story

To close this chapter, a story that shows what kind of courage the release train makes possible. In our space stations there are arcade machines with twenty minigames. The first version of them was a trick: the minigames were little web games, and the game lugged around a **complete embedded web browser** — a second Chromium, hidden in the belly of the game, just to display them and the wiki. It worked. But it made the download fatter, ate memory, and cost noticeable loading time at every arcade machine.

At some point the decision fell: the browser goes. All twenty minigames were ported to a small, custom 2D engine *inside the game itself*, the wiki right along with them — a lot of work for something that afterwards **looks exactly like it did before**. It's the most thankless kind of task in software building, and without eleven hundred tests and a release train that carries the rework to the players in small, reversible portions, I would never have dared it.

The reward came, as always in this book, on test evening. Justus sat down at an arcade machine and said: **"They open instantly now!"** — No new feature, no new content. Just a game that is suddenly lighter, faster, and more honest. Sometimes the best feature is the kilo you take off the game. And sometimes the true function of a release machine is not to deliver new things faster — but to give you the courage to rip old things out.

> **What the AI actually did here** Claude wrote the pipeline configurations (build workflows for Windows/Linux/browser/Docker, version number from the tag, later the deploy workflows)

> and debugged step by step through the grinding bug hunt of June 20. The human: decided that the machine had to exist in the first place (before the first release!), endured the failed attempts, triggered every release, and took responsibility for the changelogs.

Next chapter: Open source from day one — and the Sunday a CEO we'd never met brought our game into the browser.

Chapter 17 — Open Source from Day One — and the Sunday It Paid Off

---

Someone out there can do something better than you — and open source is the invitation to show it.

Of all the decisions in this project, one is the hardest to explain if you don't come from the software world: **the complete source code of our game sits publicly on the internet.** Every line. The server, the client, the tools, the bugs, the embarrassing comments — all viewable, all copyable, for anyone, at any time, on GitHub.

The question comes reliably: *Why do you do that? Can't somebody just steal your game?* This chapter is the answer — and it takes the form of two stories, both of which end with a stranger from the internet giving our family project a gift.

> **In a nutshell: open source** *Open source* means: the source code — the readable blueprint of a program — is public, and a license explicitly permits reading it, modifying it, and passing it on. The opposite model is "proprietary": you only get the finished program, and the blueprint remains a trade secret. A huge part of the internet runs on open-source software, maintained by a worldwide community of companies and volunteers. And yes: "stealing" is governed by the license — more on that in a moment.

Why open — the unromantic half

Before the goosebump stories, the sober truth, because it's often left out: **open source made our project better before the first stranger contributed a single line.** Public code enforces hygiene. Passwords and keys sit cleanly in configuration files instead of in the code (anyone can look inside, after all!). The documentation has to be right, because strangers read it. Decisions get justified in a traceable way, because they can be debated in public. You could say: we bought ourselves the discipline of a team by exposing ourselves to the world public — which, by the way, 99.9% of the time never drops by. But you never know, and that's exactly what does the trick.

Added to that was a license lesson we had to learn along the way. We started with the most permissive license of all (MIT: take everything, do whatever you want with it). Until we realized what that means for a multiplayer game: someone could have taken our code, built a **commercial server service** out of it, and kept all the improvements for themselves. So we switched to the AGPL-3.0, whose core can be explained in one sentence: *Take everything, change everything — but if you pass it on or run it as an online service, your changes have to be open too.* Whoever runs a server with our code gives back to the community; self-hosting for friends and family is explicitly welcome and documented. The switch cost two days (license texts, an automated contributor agreement for contributions, docs) and gave the project the foundation on which you can tell strangers with a clear conscience: **Join in — it stays open.**

And we meant "join in" literally: the game itself has a "Join in" button that explains how to contribute. The invitation is built in. Then everything went much faster than I had dared to hope — I had braced myself inwardly for months of silence, because that's how it usually goes with small open-source projects. It took **a few days**.

Then came Cora.

Story one: the first contribution from outside

Cora de la Mouche (on GitHub: @corarona) appeared in our project at the end of June — and didn't bring along some polite little typo fix. But rather something that sat far down our wish list because it was simply too big for us: a **complete native Linux port**. No compatibility trick, but real Linux support: a dedicated launcher project, the build scripts carried over to Linux, the release pipeline extended with Linux packages, documentation to go with it. Over three days the contributions moved into the project, and ever since, every single release ships with a Linux package — built on the foundation Cora laid.

I remember the feeling when the contribution came in, and it wasn't primarily joy about Linux. It was this strangely solemn thing: *Out there, a person we have never met put evenings of real lifetime into our child's game — voluntarily, skillfully, as a gift.* We recorded it in the release, the way you record such things: version 0.6.0 officially bears the nickname **"our first contributor!"**. And in the devblog stands the sentence that has been our short case for open source ever since: we would have built this ourselves eventually — probably worse and certainly later. Someone out there can do something better than you. Open source is the invitation to show it.

Story two: the Sunday

And then, not even a week later, the story that in this family is now simply called "the Sunday."

It began, the way ring three of marketing plays out (Chapter 20), with a **comment on Reddit**. Under one of our posts, someone wrote, in essence: *"Look — I put it on glitch.io, you can play it in the browser."* Pause on that for a second: a stranger had not just played our game. He had taken it, brought it into the browser, put it up and running somewhere — and casually left us the link.

Then came the part that moves me to this day: a **pull request** in our repository. Inside: the finished browser port as a contribution, WebGL plus database support. And with it a question of disarming politeness, in essence: **"Hey — it's okay if you think it's too early to add another client to the project right now…"** Here is someone giving a stranger's family project a complete port and apologizing in advance in case the gift comes at a bad time. That's what open-source culture looks like when it's good.

Who *was* that, anyway? Finding out cost me — fittingly for this book — first of all **a round of ChatGPT research** on the commit author. Result: the name behind the contribution was **Devin Dixon** — a **CEO from the USA with a game platform of his own**. A company chief had taken his free time on a Sunday afternoon (his time zone) to bring a ten-year-old's game into the browser. I wrote to him on Reddit and thanked him; he never asked for anything more in return.

And Justus? **Completely beside himself.** And I'll ask you to take the child's perspective for a moment, because it is the heart of the story, and Justus put it best himself: *Someone from America whom we don't know — and who helps us in his free time to make the game better! Insane!* You cannot illustrate self-efficacy any better: a ten-year-old's idea was good enough that an adult on another continent, unprompted, builds on it. On a weekend.

The Sunday port did not become a footnote: the browser version is an official part of the project today — "play in browser" on our website carries on the lineage of that pull request (what we built out of it is told in Chapter 18). And the "another client" it was supposedly maybe too early for? It was exactly on time.

The contributor list as a family photo

Whoever looks at our project's contributor list today finds there —

alongside Cora, Devin, and a friendly person named Maqbool Ahmed, who dusted off our German translations — several entries of: me. Among others as **ai_cat_coder**. In case you suspect a mysterious AI identity behind that name: I wish, but the resolution is more mundane and dearer to me. I love cats. We have two tomcats at home. Hence the name.

And in that mundaneness lies the actual punch line this chapter should end on: **the AI does not appear in the contributor list at all.** Many thousands of lines it wrote moved into the project — but always under *my* name, with my sign-off, on my responsibility. That is how it should be, and not out of the machine's modesty, but out of the human's accountability: that list holds the people who stand behind their contributions. People from several countries, one family, two tomcats as namesakes — all working on the same game. I couldn't think of a better family photo for this project.

> **What the AI actually did here** Little — and that is the point of this chapter. The license decision, the invitation, the trust: human. The gifts from Cora and Devin: human, in the best sense. The AI reviewed and integrated the contributions as they moved in (outside code runs through the same tests as our own — Chapter 6 applies to everyone) and mechanically carried out the license switch. But this chapter is about the one resource no model generates: people who join in voluntarily.

Next chapter: A small server for a small fleet — what happens when a family project suddenly runs an online service.

Chapter 18 — A Small Server for a Small Fleet

Five euros a month. For that you get: a portal, official worlds, a container fleet, monitoring with 144 alarms — and the responsibility that all of it runs when other people's kids want to play.

Up to this point in the story, our game ran on computers that belonged to us. Whoever wanted to play along needed someone in the house to start a server. That is perfectly fine for a family game — and it is the moment where most hobby projects sensibly stop. Because the next step changes the nature of the project: **a service on the internet that runs while you are asleep.** This chapter tells how a family project took that step — with a rented five-euro server, an AI as system administrator, and a few principles that have proven themselves.

The rental: a VPS

> **In a nutshell: servers, VPS, and "the cloud"** A *server* is simply a computer that runs all the time and is reachable over the internet. You can rent a whole one — or a *VPS* (virtual private server): a partitioned share of a big machine in a data center, with its own resources and full control, starting at a few euros a month. "The cloud" of the big providers is the same principle with more convenience, more magic — and bills that grow along with you. For us, the Raspberry Pi philosophy from Chapter 2 applied: start small, set limits, measure.

Our entire online world — the player portal, the official worlds, all player-

created worlds — runs on **a single small Linux VPS** for around five euros a month. No data center, no five-figure cloud bill. That this works is not sorcery, but the late dividend of an old decision: a server that, per its spec, has to run on a hobbyist board is bored on a real VPS.

The machine was set up the way everything in this book is by now: in dialogue with the AI. Claude configured the basic hardening before the game came anywhere near the server — a firewall that opens only the necessary doors; fail2ban, which locks out automated break-in attempts; access only with a cryptographic key instead of a password. To me, this phase felt like working with a very experienced admin colleague who knows every configuration file by heart but always asks before doing anything dangerous — because that is exactly the rule we had given it.

Only one trade traditionally ran through ChatGPT: the **DNS setup** — that mapping which turns "blocksbeyondthestars.com" into our server's address, managed at Cloudflare.

> **In a nutshell: DNS — the internet's phone book** Computers find each other via numeric addresses (IP addresses). *DNS* is the worldwide phone book that translates names like blocksbeyondthestars.com into those numbers. Whoever owns a domain maintains its phone book entries themselves — and a wrong entry means: the world can no longer find your website, even though it's running. Hence the respect with which DNS is spoken of in this house.

The fleet: every world in its own cage

The heart of the service is a small program we call WorldHost — and a principle that can be explained in one sentence: **every world runs in its own container, and whoever isn't being played sleeps.**

> **In a nutshell: containers (Docker)** A *container* is lightweight packaging for a program: it runs inside it isolated, with its own allotted resources, as if it were alone on the machine — except that dozens of containers can exist side by side. *Docker* is the most widespread tool for this. The charm for us: you can give every container hard limits. A program running amok inside a container remains a problem *inside the container*.

Concretely: when a player creates a world in the portal, it gets its own Docker container — with a fixed memory limit, capped CPU shares, and its own save data. At most ten worlds run at the same time; **a single world can never bring down the server.** And best of all: when all players leave a world, the save is secured and the container is stopped — the world sleeps and costs nothing. When someone clicks "Play" weeks later, the WorldHost first checks the authorization — for protected worlds, the password, mind you, **before** waking it up, otherwise anyone could spin up other people's worlds through mere join attempts. Then it starts the container, and for a few seconds the portal shows: "Waking up world…" I still find that phrasing beautiful today. It is technically exact and still sounds like something out of a children's book.

The whole thing is deliberately built small — no Kubernetes, no cloud orchestration, no magic: one server, Docker, a service that starts and stops containers, in front of it a Caddy proxy that takes care of encryption certificates automatically. The devblog has the sentence that carries our entire infrastructure philosophy: *For a family project, this goes astonishingly far — if you set limits and keep watching.*

The watching: monitoring that calls your phone

"Keeping watch" is literally automated here. On the server runs Netdata, a monitoring tool, in our case with **over 140 configured alarms** — from "disk filling up" to "portal not responding" to subtleties you only think of

once they have struck. When an alarm fires, a push notification appears on my phone. The goal is modest and priceless at the same time: **to learn about problems before the players do.** A family father cannot provide 24/7 on-call duty — but he can make sure that the ringing finds him, not the complaint.

And because measuring famously ruins opinions, the loveliest insight from operations: when we measured at idle, **around half of the used memory didn't go to the game at all** — but to Docker and the monitoring itself. The infrastructure that carries the game is, at rest, heavier than the game. There is no better miniature of what operations means: the host eats, too.

Maintenance with decency — and responsibility, the real thing

A service that runs wants to be updated — and somewhere out there a child is at that very moment standing right in front of their first diamond vein. That's why a feature mattered to us that in the ideal case you never notice: **maintenance without getting kicked out.** When an update is due, the server sends every running world an announcement; in the game, a banner with a countdown appears ("Server restart in 10 minutes") that you have to actively dismiss — but only after a few seconds, so nobody swipes it away by accident in the heat of battle. When the countdown runs out, the world shuts down in an orderly way: save, see the players off with an *explanatory* farewell screen instead of a cryptic "Connection lost", stop the container. On the next join, the world wakes up with the new version, and the save data is there.

The unexpected side effect is in the devblog and deserves bold print, because it is a general law: **since updates became relaxed, we do them more often.** Good maintenance tools don't improve maintenance — they improve the whole project.

That leaves the honest closing question of this chapter: why put yourself through this? Operations is the part of the project that is never finished — not a feature you complete, but a state you hold. The answer is the same as everywhere in this book, only one notch more serious: because there are children at the other end. If Justus's world were "just gone," I would fix it, and he would know why. If a stranger's child's world is gone, there is only us — and that is what the limits, the alarms, the farewell screens are for. Responsibility is what remains of enthusiasm once you make it permanent.

> **What the AI actually did here** Claude hardened and configured the VPS (firewall, fail2ban, Caddy), built the WorldHost including the container fences, set up the monitoring with its alarms, and implemented the maintenance system; ChatGPT helped configure the DNS entries at Cloudflare. The human: decided that the service should exist at all (the actual threshold!), set every limit (RAM, CPU, number of worlds), routed the alarms to his own phone — and carries what no AI can take over: the on-call duty.

Next chapter: Kid-friendly by design — why the most important features of this game are the ones you don't see.

Chapter 19 — Kid-Friendly by Design

You build differently when your most important user sits at the same kitchen table. By now I believe: you build better.

There is a category of features in our game that appears in no trailer, sets off no devblog fireworks, and has still received more development evenings than many a favorite feature. You can recognize them by a shared origin: they all begin with the same half-sentence. **We're trying to make the game kid-friendly — that's why there's…**

That half-sentence is the leitmotif of this chapter, and I'm deliberately spelling it out as a list, because that is exactly how it grew in the project: as a chain of consequences from a single foundational decision. I develop an online game, and my son plays it — I sit on both sides of the table, as a developer who wants features and as a father who knows what's out there on the internet. Everything else follows from that double role.

That's why there's: the Treehouse Principle

Anyone coming from other online games does a double take here first: there is no open server list, no "join any world" button, no matchmaking with strangers. Multiplayer works exclusively via **password and word of mouth** — and that is not a missing feature, it is the foundation. Internally we call it the **Treehouse Principle**: the people who get into a treehouse are the ones you personally invite. Sharing a world here means: create it, set a password, pass it along — in the schoolyard, in the family group chat, on the sofa. Even worlds that show up in the

public browser exist *only with a password*; remove the password and the world automatically drops off the list. There is simply no path by which a stranger lands unannounced in a child's world.

And there is **no exception** to this — not even for the official worlds we run ourselves. A permanently supervised public playground is not something a family project can honestly promise, so we don't promise it. We know the price: no "just hop in with a thousand people," slower growth. But treehouses aren't built for a thousand people. They're built for the right ones.

That's why there's: F1, /report, and the separation of feedback and complaint

Second consequence: **reporting has to be as easy as the thing it protects against.** A child who experiences something odd must not owe anyone a burden of proof. So: in the game, the **F1** key is enough to send us feedback directly — the note from Chapter 12, built in as a button. For serious cases there is the /**report command** in chat, and it lifts the heaviest weight off the child: it automatically quotes the last chat lines and the world context along with the report. Whoever reports doesn't have to transcribe anything, prove anything, or know anything — type, done, an adult takes a look.

Just as important to us was separating the registers: **feedback is not a complaint.** In the portal, "Feedback & Ideas" and "Report a player" are deliberately separate paths with their own friendly language — someone who wants to share an idea shouldn't have to fill out a form that feels like filing charges. And for parents there are dedicated information sections right on the start page and next to the rules, so nobody has to go hunting to learn who their child is being entrusted to here.

That's why: the Radio is optional

Third consequence, and it put one of our proudest technical features into perspective: our game has built-in **voice chat** — the "Radio," with range levels, as befits a space game. And it is **optional**. Talking-with-strangers is the most delicate function of all for children; so the default stance is not "on, because we built it," but: switchable off with no disadvantage. In a treehouse world among friends, the Radio is great. Everywhere else the rule is: what stays off can't go wrong. It's the same logic as with the language model in Chapter 10 — the safest door is the one that doesn't exist — only this time applied to people instead of machines.

That's why there are: accounts without email — and the tedious side duty

Fourth consequence, the data-protection one: a player account here requires **no email address**. A child shouldn't have to leave a data trail just to stack blocks. What we don't collect can't hurt anyone — not in a leak, not in a takeover, not through us ourselves. (The honest flip side we communicate openly: there's no "forgot password" email then, either.) On top of that come the invisible craftsmanship details nobody notices — which is exactly why they have to be right. World passwords are never stored in plain text, but as a cryptographic hash; even we can't read them. After ten failed attempts there's a cooldown against brute-force guessing. And crash reports pass through a filter that strips personal data before anything gets uploaded.

And then there is the duty that is the least glamorous in this list, and over which I honestly sighed while writing: **there is a privacy policy.** And a legal notice. Real ones, legally proper, linked on the website. The "tedious side duty," as it's called internally — nobody builds a space game because they love legal texts. But a service used by other people's families has to have these texts, period. (At least: even legal

texts are more bearable to type with AI support. You remain responsible yourself all the same — the pattern of this book, without exception here too.)

And that's why there's even: the break reminder and the signature

Two final consequences, one to smile at, one entirely sober. The one to smile at: the game shows your session length and **gently reminds you to take breaks**. Did I, as a developer, enjoy building that in? Let's put it this way: the father outvoted the developer. (Justus's opinion on the matter is on record and differs.)

The sober one: as of recently, our Windows versions are **code-signed** — digitally signed, so the operating system and virus scanners can verify where they come from. This is made possible by a program that offers exactly this to open-source projects for free. Sounds like a technician's footnote, but it belongs precisely in this chapter: parents installing a game from the internet for their child deserve an operating system that doesn't throw warnings while they do. For a children's game, trust is not one feature among many. It is the product.

The principle's balance sheet

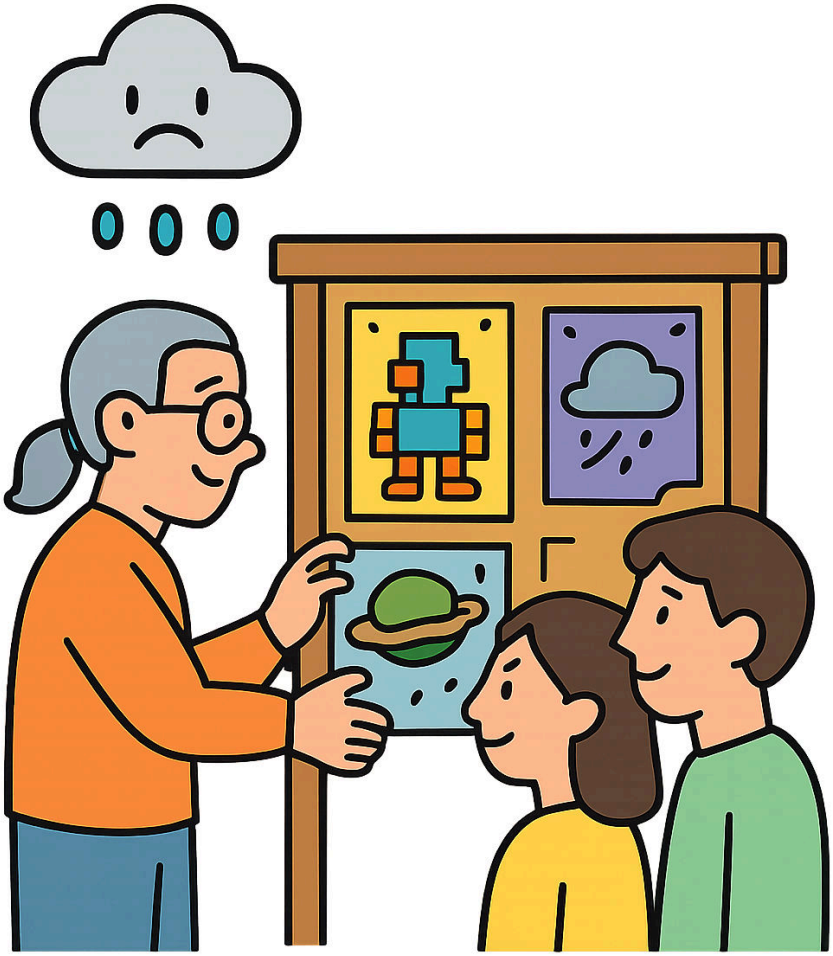
To close, the observation that surprised me most in this whole area, and it's already in the family blog: **not one of these decisions made the game worse.** The password rule makes worlds more personal. The lean accounts spare us data-protection headaches. The reporting paths make the community friendlier before it gets big. And the break reminder … occasionally reminds the developer, too.

You build differently when your most important user sits at the same kitchen table. You build better. And — this is the quiet thesis of this chapter — you build *transferably* better: nothing about this principle

requires a child of your own. It only requires treating your users as if they were sitting at your own kitchen table.

> **What the AI actually did here** Implemented: everything — password hashing, rate limits, /report including chat context, portal copy, PII filter, the maintenance and signing integration; even drafts of the legal texts. Decided: nothing. Every "that's why there's" rule in this chapter is a parenting decision that was handed to a machine as a requirement. Kid-friendliness cannot be generated. You can only order it — and then have it built very well.

End of Part V. — Part VI turns the viewing direction around: how does the world even find out this game exists? Marketing with three AIs, a withered itch.io, and 28 YouTube videos in 30 minutes.



Chapter 20 — Marketing with Three AIs

A game nobody finds is a diary with graphics.

At some point, every project that wants to be public faces the realization developers like least: **building is half the work. The other half is being found.** And while everyone can by now picture AI helping with the building, the second half is the quieter part of this story — even though for us it ran just as consistently AI-supported. Just with different colleagues.

This chapter is the tour through our reach workshop: a home base, an automated blog, an AI as social media strategist, a campaign on other people's lists — and one honest defeat.

The home base: a website of our own

The first decision was old-fashioned, and I'll defend it against any trend: **the game's home is its own website** — blocksbeyondthestars.com — not a profile on some platform. Platforms change rules, algorithms, and fees; a domain of your own belongs to you. On the website lives everything someone who hears about the game needs: the game itself (download for Windows and Linux, "Play in browser" with one click), the devblog, the rules, the note for parents. And a page that carries our actual project goal: **"Join in."** Testers, developers, creatives — "Let's make the game great together!" it says there. It is meant literally; the end of the book belongs to that sentence.

The site is built on a site-builder system (Wix) — deliberately: this

project's engineering energy belongs to the game, not to the website's underpinnings. But even this supposedly handmade part of the project is in truth an AI relay race, and the lineup is familiar by now: **planned** the website I did first with ChatGPT — page structure, what belongs on a game website for families, in what order, what goes in the menu. **Written** it was by Gemini — the marketing department got to show what it can do. And **translated** the finished site was by the translation AI built into Wix itself — so the English version of the website comes from a fourth machine that hasn't appeared in this book at all so far. Three AIs for a site-builder website; curated, assembled, and approved, as always, by a human at the kitchen table.

One detail of the website is, beyond that, genuine engineering work, and it is my favorite example of how far "consistently AI" goes with us.

And because this book preaches transparency, an anecdote belongs here that I could just as easily keep quiet — which is exactly the reason to tell it. The blog only went online when the project was already weeks old; the articles about the early events were written in retrospect. And so that the blog chronicle would read "naturally," we **backdated the publication dates** of the retroactive articles **by script** — each post got a date next to the release it covers. That script, too, was of course written by the AI; even our historical revisionism is automated. Is that bad? The texts are true, the events documented, and blogs are allowed to be chronicles. But it also means this: **publication dates on the internet are stage scenery** — on our site and everywhere else. The real chronology of this project isn't in the blog, it's in the Git history, and this book sticks exclusively to that. (You can verify this. Everything is public. That's the point.)

The blog that publishes itself

Our devblog — those articles about shader mishaps, the Treehouse Principle, and factory empires that this book quotes so liberally —

appears **bilingually**, German and English, by now dozens of articles. Anyone who has ever maintained a blog can guess the grunt work behind that: enter the text, format it, enter the translation, link it, categorize it — per article, per language.

Here's how it works for us: the articles are written as plain text files (with AI support, of course, edited by me). Then a **home-built Python script** takes over: it converts the texts into the website system's format, creates them as blog posts via its programming interface, links the German and English versions to each other, and sorts the categories. **Even the blogging ended up automated** — the script was written by, you guessed it, Claude. It's the same assembly-line logic as with the textures in Chapter 8, except that what comes out at the end isn't a tile but a published article. And it's a pretty full circle: the tools this project was *built* with are the same ones it is *told* with.

From the inside out: the channel chronicle

So how do you actually spread a game like this? Not with one big launch day, but — and by now I believe this is the right order for any small project — **in rings, from the inside out**, always going first to where somebody knows you.

Ring one: WhatsApp. The moment the first version of the website was up, the message went to where marketing is still called trust: **family and friends**, by messenger. "Look what Justus and I built" — with a link. That is the most honest audience in the world: they click out of affection and judge mercilessly anyway (relatives are astonishingly close to Reddit level there, just more kindly phrased).

Ring two: work. Then targeted promotion among my **work colleagues** — people who build software themselves and know what they're looking at. From this ring, incidentally, came the most productive single tester in the project's history: Bastian and his first-hour report from Chapter 12.

Colleagues are doubly valuable as an audience — they don't just play, they also understand *what* they are praising or picking apart.

Ring three: the open internet. And finally Reddit — which the next section is about, and you can already guess from the tone: the transition from ring two to ring three is the moment the water gets colder.

Alongside all this, in parallel, came the scattershot attempts in every direction, and I list them honestly, with results: **wrote to YouTubers** — several, each with a personal message; response so far: none. (That is normal and sobering all the same; content creators drown in such requests. It's here so nobody believes everything worked for us.) **LinkedIn** — yes, really: the father-son-AI project is surprisingly well received there, because it reads professionally; the career network was more interested than many a games forum. And **Facebook**, above all a post in a **local group here in Trier** — the charm of the local: there you aren't "an AI project," you're "the family from around here who built a game." Local beats viral when you're small.

The GitHub project lists — the most deliberate of these channels — get their own section below. But the pattern that carries this chapter is already emerging: **the closer the channel was to real relationships, the better it worked.** WhatsApp beat LinkedIn, LinkedIn beat Reddit, and the local Facebook group probably beat everything combined, measured in warmth per click.

The strategist: Gemini

For the question of *where* to talk about the game, we brought the third big AI onto the team: **Gemini** became our marketing department. The job profile: develop a social media strategy — which platforms are worthwhile for a family voxel game, **which subreddits** (topic forums on Reddit) are the right ones and what tone they expect, what makes a

good post there, which ideas generate reach beyond the same-old-same-old. Plus drafting concrete posts.

Why a dedicated AI for this, of all things? For the same reason you don't have the electrician do the painting: marketing copy is its own craft with its own laws — hooks, length, community etiquette; a Reddit post that smells like advertising is dead before it reaches the bottom of the page. An AI that you deliberately brief as a strategist thinks of things a developer brain doesn't think of at eleven at night. Of individual subreddits' self-promotion rules. Of the difference between "show your project" Thursdays and normal days. And of the question of which story you're actually *leading* with — in our case the obvious one: not "new voxel game," but *father and son build a game with AI*.

The cold shower: "AI" is a trigger word on Reddit

Before this chapter gets to the quieter successes, an experience belongs here that caught us cold — and that anyone going public with an AI project in 2026 should know about: **in many online communities, Reddit first among them, "AI" is not a selling point. It is a bogeyman.**

The irony of our situation: the most honest thing about our project — that we say openly how it was built — is precisely its biggest handicap there. A post that so much as mentions "developed with AI" draws reflexive, sometimes **toxic rejection** in many forums. And because that sounds abstract, here are our two real experiences, laid side by side:

Experience one, the good one: In the subreddit **r/aigamesdev** — a community explicitly devoted to AI-assisted game development — the project got **lots of positive feedback**. Interested questions, encouragement, real exchange. You can already guess: we had hit the right bubble.

Experience two, the instructive one: From the subreddit **r/**

VoxelGameDev I was **thrown out on my ear**. I had overlooked that you may not post *anything at all* there that has even a whiff of AI about it — a voxel game, open source, fully inspectable, AGPL-licensed, exactly the kind of project the forum exists for: deleted, because of the tool it was built with. Take a moment to picture the irony: of all things, a forum of game developers bans every project that owns up to it — in an industry whose biggest players, Microsoft & Co., massively provide open source *and* have long been coding massively with AI (like practically the entire software industry).

From the same corner came the most poisonous comment in the project's history: I was **"devaluing the work of humans"** by asking for help with an AI project here — and was I prepared **to pay for the labor time?** My follow-up question, whether the author himself had ever paid an open-source developer for open-source software, went — surprise — **unanswered**. I tell this without bitterness, but with the observation that follows from it: the outrage was not a position that could withstand a question. It was a reflex, and reflexes don't debate.

The finest irony of this episode, however, is supplied by a machine. My marketing department — Gemini — had long since advised me about exactly such comments, in essence: **"Let it go. It only costs you pointless energy. Put your time into the positive side of communication instead, and focus on the project."** I didn't listen right away — the urge to answer unfairness is human, and that's exactly what makes it so expensive. By now I take the advice to heart: it is the one place in this book where the AI was the more level-headed part of the team.

The lesson behind it is structural, and then we'll let the topic rest: **Reddit is not a public; it is an archipelago of bubbles.** For every opinion there is a forum where it is consensus — step into the wrong bubble and you get deleted or attacked, often before anyone could discuss anything. The anger has a real core (these communities genuinely are drowning in carelessly generated stuff, and creative

people have real existential worries), but it no longer distinguishes between spam and a family project. For our strategy, that simply meant: we tell the story where people read it before they judge it — on our own blog, in this book, in communities that evaluate projects instead of buzzwords. We will never hide the AI part in the process; a project that has to conceal its most interesting property would have betrayed its own foundation. And for everyone planning something similar, the short version: **expect places you love to hate you for your tools — and choose your stages accordingly.** An anonymous comment calling the game "slop" hasn't played it. Justus's note weighs more.

The campaign: on other people's lists

Alongside social media ran a second, quieter campaign that I especially recommend to developers, because it costs almost nothing and carries for a long time: **awesome lists**. In the open-source world there are curated link collections ("awesome-…") on every topic — including open-source games. Whoever is on one gets found by exactly the people who get excited about this sort of thing: developers, tinkerers, potential collaborators.

So we applied for inclusion systematically — research and application texts AI-assisted, it goes without saying: following each list's rules, with a proper description, as real pull requests. As of today, the game has been accepted onto **three such lists**, with more applications pending. This is unglamorous marketing without fireworks — but these are permanent entries in places where our target audience is searching anyway. Reach doesn't have to be loud.

The honest defeat: itch.io

And then there is the story that likes to go missing from the marketing chapters of other books. **itch.io** is *the* marketplace for indie games, and of course we were there: game page set up, uploads automated (the

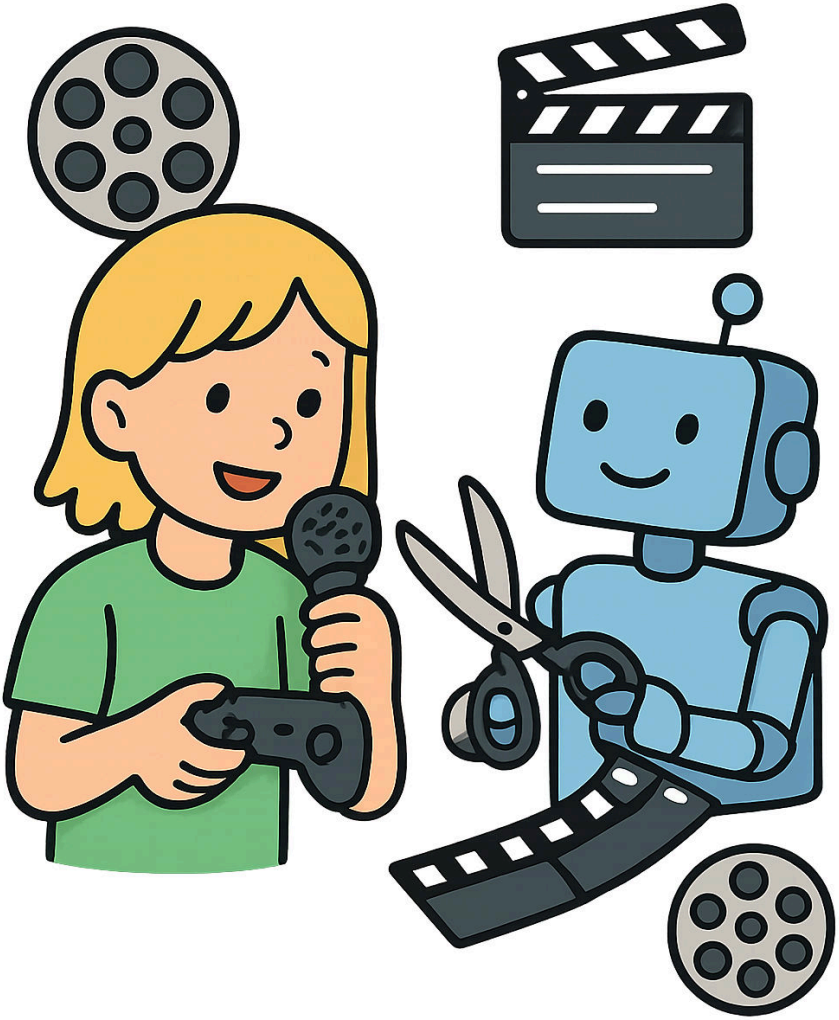
release train from Chapter 16 delivered every version there too), plus lovingly maintained English devlogs — personally written, told around Justus's test evenings, with all our heart in them.

It didn't work, and the numbers are of an honesty that is almost funny again: standing out in the flood of new games there every day is nearly impossible — **to this day, our game doesn't even show up in itch.io's search when you type in its own name.** It had a handful of views. And of the **two** downloads, at least one was from a friend. (Let me repeat: we built an automated upload pipeline for this platform. For two downloads. One of them Bastian-category.) At some point we drew the conclusion that feels wrong and is right: **we stopped writing devlogs there.** Not in a huff — the uploads keep running, the game remains available there. But the storytelling energy, this project's most limited resource, has flowed ever since to where it lands: our own website, our own blog, our own channel.

If this chapter has one lesson, it is this, and it applies far beyond games: **not every channel carries, and admitting that in time is strategy, not capitulation.** Marketing with AI does not mean broadcasting everywhere at once — the machine would do that without complaint; content has become cheap, after all. It means using limited human energy to choose the places where real people answer. That too is, in the end, curating.

> **What the AI actually did here** Gemini: social media strategy, subreddit selection, post drafts, reach ideas. Claude: the blog publishing script (Markdown → website API, linked bilingually), devlog and application texts, the awesome-list pull requests. The human: chose the channels and un-chose them again (itch!), edited every text and personally stood behind it — and held to the rule that nothing gets posted except by him. Reach can be automated. Credibility cannot.

Next chapter: 28 videos in 30 minutes — Justus plays, the AI edits, the family channel broadcasts.



Chapter 21 — 28 Videos in 30 Minutes

Producing 28 videos took 30 minutes. Uploading them to YouTube took 40.

Of all the workshop stories in this book, the following is the one I tell most often when someone wants to know what AI tools *really* change in everyday life. Not because it's the most technically impressive — but because everyone immediately understands what has vanished here: **video editing.** That activity on which content dreams have been dying for twenty years, because every hour of gaming fun comes with ten hours in the cutting room.

For us, several hours of raw footage came down to: half an hour. Out came 22 short videos and 6 longer Let's Plays — subtitled in two languages, with finished description texts. Here's how it went.

Step 1: Justus Presses the Record Button

Every gaming video begins with someone playing the game, and that role was filled without dispute. Justus recorded several play sessions with **OBS Studio** (the standard free tool for screen recording), including live commentary. No script, no retakes, no cue cards — just him, playing and talking as he goes. That unfiltered energy was exactly what we wanted; nobody needs a ten-year-old reading off a page. It is, incidentally, the most perfectly divided moment of labor in the whole family project: the part of the production that takes the longest is, for the person doing it, **not work at all**.

Step 2: Making the Video Readable

Now the problem: an AI can't efficiently "watch" hours of video footage. The trick — and it's the key to this entire chapter — is to make the video **readable**. The recordings went to **ElevenLabs**, which produced transcriptions from them: everything spoken as text, **with timestamps** — who said what and when, down to the second. Suddenly a video is no longer a wall of pixels but a searchable document.

> **In a nutshell: Why timestamps change everything** A transcript with timestamps is the map of a video: "12:41 — Justus discovers the cave and shouts 'WHAT is THAT?!'" is, for a text AI, a find it can evaluate like a line of code — and the timestamp tells the editing software exactly where to put the scissors. Once moving images exist as structured text, an AI can analyze pacing, highlights, and boring stretches without ever having "seen" a single frame. The principle carries far beyond videos: make your material readable, and the machine can work with it. (Chapter 5 called this: move your project into text.)

Steps 3 and 4: Think First, Then Cut

Videos plus transcripts landed in a **Claude Cowork project folder** (Cowork is, simply put, Claude with a workbench: a workspace where the AI can handle files). We added our design fundamentals — brand colors, fonts — so that titles and overlays would look like *our* game and not like a random template.

And then comes the step I would underline twice, because it is literally the same lesson as with the programming in Chapter 4: **You don't just let the AI start.** First, brainstorming — which moments are Shorts material, what makes a good hook, what gets cut. Then a **detailed editing plan** before anything gets rendered: which clips, which time windows, short and long versions, burned-in subtitles in **German and**

English (a bilingual family project stays bilingual), plus a file with finished YouTube descriptions for every video. Only after the plan was approved did the cutting begin. Analysis, plan, check-in, execution — same workflow, new medium. The method, apparently, matters more than the material you apply it to.

Steps 5 and 6: Voilà — and the Punch Line

About **one hour of work with Claude** later, everything was there: **22 Shorts — eleven in German and eleven in English — and three longer videos, also in both languages**, subtitled, captioned, ready to upload. Plus, and this is the part I find almost the most elegant: **a finished editorial document** — a publishing schedule, description texts for every single video, all in one file to work through.

And now the twist that makes this story perfect: **Uploading to YouTube and scheduling the releases took longer than cutting the videos and Justus' OBS recordings combined.** Pushing up each video individually, copying the description from Claude's file, setting the publish date — click, paste, repeat, twenty-five times. For that last mile there is simply no usable automation interface; no AI magic could help. Our most futuristic pipeline, brought to a crawl by copy and paste. It's the same punch line as with Suno in Chapter 9, only sharper — and by now my favorite example of where automation really ends in 2026: rarely at the models' abilities. Usually at a missing power outlet.

The Trailer: Four AIs, One Relay Race, Twenty Seconds

How does a family without an editing suite produce a proper game trailer? For us, it was the densest AI relay race of the entire project, and because it shows the interplay so beautifully, here is the complete chain:

First pair of runners: the music. I asked **Gemini** for a prompt for an **epic instrumental track for a game trailer** — the marketing department describes, the house band plays: I gave the prompt to **Suno**, which generated the track.

Second runner: the editor. Track and assignment went to **Claude Cowork**: first, trim the song to **twenty seconds** — trailer length. Then I laid out the video footage we already had anyway: the Let's Play recordings **along with the audio transcription in which Justus narrates what's happening**. That audio track was precisely the key — using Justus' own words, Claude could identify which moments belonged on screen.

The rules of the race: as always, the **plan** before the cut — and the requirement to match the **visual style of the already existing YouTube videos** (text overlays, look), so the trailer would look like it came from the same house. After that, it took **two more iterations** until the timing of the overlays sat right with the music. And the trailer was done.

And the most annoying part, you can guess it: **as always, putting it on YouTube.** Upload by hand, paste the description by hand — which Claude had of course long since written and placed in a file. Four AIs deliver a trailer, and the human spends the longest step of the process copying text into a web form. If there is ever a monument to the state of automation in the year 2026, I nominate this scene.

The Channel: Family on Air

That leaves the question of *where* all this broadcasts. The answer fits the treehouse tone of the project: not a sterile studio channel, but the **family channel** — "mamas und papas blog" on YouTube, the moving-picture counterpart to the blog where Verena and I write about family projects anyway. That's where Justus' AI-edited play videos now

appear: the son plays, the AI edits, the family channel broadcasts.

Added to that is the second moving-picture track, entirely without a kid on camera: **script-driven marketing clips** straight from the game. In the project there's a screenplay file that describes scenes — a camera flight through space, a landing approach, the cockpit — and a script plays them back automatically and records them in clean quality, without the HUD, with sound. Trailer raw material at the push of a button, reproducible anytime, fresh after every graphics update.

In terms of marketing impact, all of this operates on a modest family-project scale, and that's fine. The real value lies elsewhere: there now exists a growing, moving archive of this project — a ten-year-old playing and commenting on his own game, captured in the summer it was made. Even if no algorithm ever surfaces it: for two people on this couch, it will someday be the most valuable footage in the world.

> **What the AI actually did here** ElevenLabs: transcription with timestamps — the step that makes video readable for a text AI. Claude (Cowork): analyzed the material using the transcripts, created the editing plan (after approval!), cut 28 videos, burned in bilingual subtitles, wrote the descriptions. The human: Justus delivered the only thing that can't be generated — real play with real enthusiasm; Marcel curated the highlights, approved the plan, and clicked "Upload" 28 times.

End of Part VI. — Part VII takes stock: what all of this cost, what it's worth, and why this book ends with an invitation instead of a closing word.

Chapter 22 — What It Cost — and What It's Worth

A complete development studio — programmers, graphic artists, sound studio, translators, sysadmin, marketing department — for the price of one family pizza night per week.

Books like this one owe their readers numbers at the end. Not the polished ones from the brochure, but the ones from the bank statement. Here are ours — after six weeks of project runtime, eighteen releases, and a running online service.

The Money Side: The Monthly Tool Rent

What this project costs per month fits into a short table, and I list the items exactly as they occur for us:

Item	Cost (approx., per month)
ChatGPT subscription (concept, research, planning)	20 €
Gemini subscription (social media, reach)	25 €
Claude Max subscription (the construction crew: code, tests, ops, video)	80 €
API costs OpenAI + ElevenLabs (textures, sounds)	under 5 €
Suno (music)	~10 $
Server rent (VPS, the entire online fleet)	5 €
Website (Wix)	10 €
Total	**≈ 150–160 €**

Two items deserve a second look, because they contradict most people's intuitions.

The biggest item is the Claude subscription — and it's the best deal in the table. Eighty euros sounds like a lot for a hobby, until you hold it up against what this item replaces: for 80 € a month, you don't even get one hour of a freelance developer in Germany. Our subscription delivers *every evening* several hours of development, testing, documentation, and admin work. The numbers from Chapter 4 — 867 commits in one month — run through this line item. You could put it this way: the most expensive line in the table is also the one with the most absurd value for money.

The smallest item is the biggest surprise: under five euros in API costs. All the textures, icons, creature images, and sound effects in this game — the entire sensory world from Chapters 8 and 9 — cost less than a kebab. That's not a calculation error, but the result of the cost brakes from Chapter 8 (order deliberately, never blindly regenerate) and the simple fact that assets, once created, stay in the game forever. And the five-euro server on which the entire online world runs is the late dividend of the Raspberry Pi line from Chapter 2.

For comparison, and this comparison has to be spelled out once: what emerges here for ~155 € in monthly rent is the service portfolio of a small studio — programming, graphics, sound, music, translation (the game is bilingual throughout!), system administration, video production, marketing strategy. Conservatively calculated, the code alone — ~114,000 lines plus 30,000 lines of tests — represents several person-years at industry-standard rates.

The Time Side: The Honest Currency

And now the currency this project was really paid in, because it obviously wasn't money. To the question of how much time it took, there

is only one honest answer from me, and it goes, word for word: **"I sit here on weekends and in the evenings after work … ever since the project has existed."** Every evening. Every weekend. Since May 30.

Nobody should romanticize that, so I'll say it unromantically: this project is a second job. The AI didn't replace the *time* — it changed what *gets created* in that time. The same evenings that used to be enough for a twentieth of this game are now enough for the whole thing. That is the real trade of the AI era, and you should know it before you make it: the tools don't lower the stakes. They raise the exchange rate.

What made the investment bearable — no: beautiful — is in Chapter 14: these were not lonely evenings. The chief tester sat right there, the balancing department called us to dinner, and the project *was* the family time, not its competitor. Not always, not perfectly. But at its core.

The Other Side of the Ledger

What stands against all that? The countable things first, and I write them down as honestly as they are — because this book isn't selling a hit, it's selling a path. As of mid-July 2026, six weeks after the project began: a published, free, open-source game in eighteen versions, on three platforms and in the browser. **21 stars on GitHub. 289 repository views. Around 240 visits to the website. Three external contributors. Five playtesters we know of — and on the official online worlds: none yet.**

And one success that shows up in none of these statistics and that I am disproportionately proud of: **If you google "Blocks Beyond the Stars," you find the game.** For such a small website, that's not a given — visibility in a search engine is something you have to earn against half the internet. And more still: **Ask Google's AI search mode what it is, and you get an answer.** The search AI knows our game and explains it. Forgive me my delight at the loop closing here: a game built by AIs is

now part of the knowledge that AIs answer from. We have become, on the most modest of scales, part of the canon.

Measured against internet standards, these are otherwise tiny numbers, and I deliberately leave them untouched. First, because at the time of this chapter they are simply the truth — six weeks is nothing in the reach business, and the school presentation (Chapter 24) still lies ahead of us. Second, because these small numbers contain perhaps the most important lesson of the entire book: **The production side has exploded — the attention side has not.** AI enabled us to build in six weeks what would once have taken years. It didn't gift us a single player. Visibility has remained the currency you cannot generate — you earn it slowly, from people, one at a time (Chapter 20 showed how laborious that is). So whoever wants to build "quick success" with AI is building at the wrong end. Whoever wants to build *their own thing* with it: go right ahead, it has never been easier. And third, three of these numbers are worth more than all the others combined: **Three strangers found this game good enough to help build it.** On day one, I would have traded any download statistic for that number.

And then the uncountable things, the real return. A ten-year-old who learned in six weeks what separates an idea from a description, what a product owner does, how to order and process criticism, and that "the AI did that" is not an excuse. A father who got to know his profession anew — as a site manager instead of a craftsman — and in the process learned more about software architecture, operations, and responsibility than in some full years on the job. And a family with a shared creation you can touch, play, and show off.

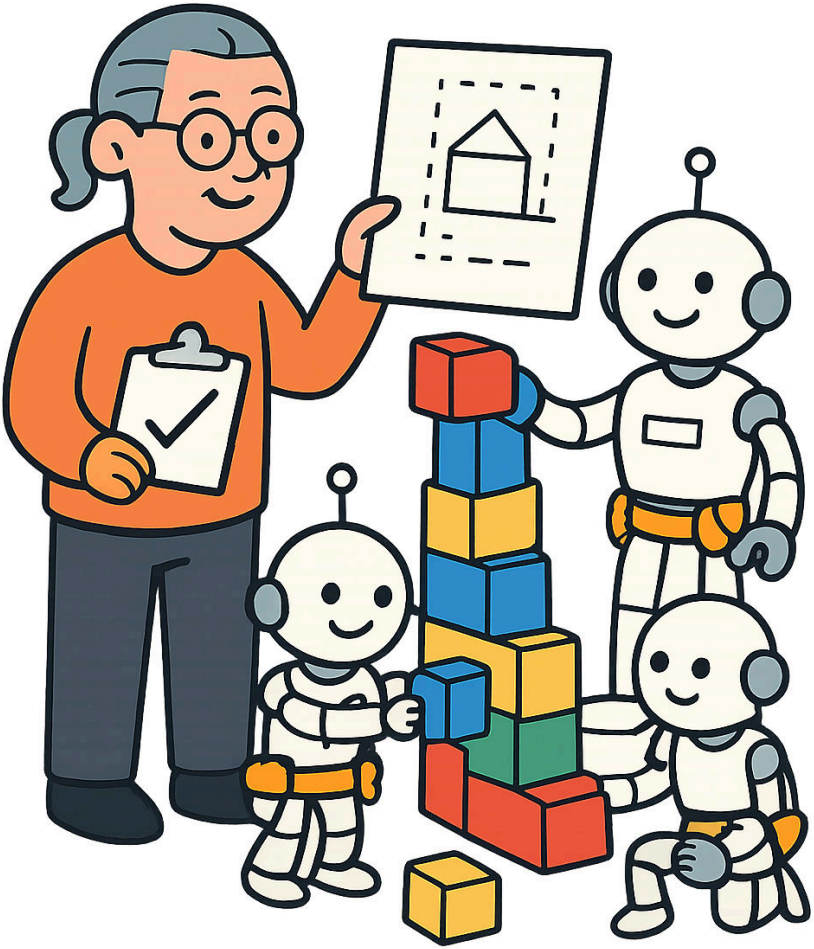
The most expensive mistake? I searched for a long time and can't find one of the usual sort — no lost save file, no burned budget, no month spent in a dead end; even the detours (the organic build, itch.io) were instructive, and progress was consistently good. What really cost energy, and the expensive kind — nerves — were **the toxic Reddit comments. More precisely: my urge to reply to them.** The story,

including Gemini's advice, is in Chapter 20; here, just the balance-sheet lesson from it: the most expensive item in this project was not an expense and not a technical error. It was attention that flowed into self-justification instead of into building. It has since been rebooked.

That leaves the question this balance sheet raises and the final chapter answers: if all of this is possible with 155 € a month and the evenings of a single family — what would it mean if a few more people helped build?

> **What the AI actually did here** Changed the exchange rate. It didn't lower the costs (those were never the obstacle) and it didn't gift us time (the project costs that to this day) — it multiplied the value of an evening after work. The entire table above is just the rent for that exchange rate.

Next chapter: Will AI take our jobs? — an honest answer from real practice.



Chapter 23 — Will AI Take Our Jobs?

This book used AI wherever it could — for code, graphics, text, music, sound. If that has bothered you the whole time: this chapter is for you.

There is a question that has hovered over this book ever since the word "AI" first appeared in the preface. It sat between the lines when the image AI painted a hundred textures in Chapter 8. It grew loud when, in Chapter 20, we were accused on Reddit of "devaluing the work of humans." And it deserves more than a footnote — it deserves its own chapter, as honest as I can make it:

Will AI take our jobs?

Because let's not kid ourselves: this book describes a project that used machines at every point where it could — and precisely where humans work. Programmers. Graphic artists. Copywriters. Musicians. Sound designers. The fear that sits deep in the bones of these professions right now is real, it is understandable, and anyone who feels it should not be smiled at here, but taken seriously.

What This Project Displaced — and What It Didn't

Let's start with the honest inventory, on the small scale, where we know the facts. Did *this* project take work away from anyone? The answer is already in Chapter 8, and it applies to every trade in this book: **There was never a budget that a human would have received instead.** The alternative to the AI textures was not a commissioned graphic artist — the alternative was a gray game of placeholder blocks that would never

have shipped. The alternative to this book was not a commissioned ghostwriter. It was: no book.

But I won't make it easy on myself and stop there. Because of course the family project is the most comfortable case. The legitimate worry goes: What happens when *everyone* works this way — including the companies that until now have paid graphic artists, copywriters, and developers? When the tool that made something possible for us *replaces* something elsewhere?

There is no answer to that which a single family project could prove. But there is an assessment, and I offer it as someone who has built software professionally for many years and has seen this industry through several technology upheavals: **I don't believe in the mass unemployment of creators. I believe in a profound change in how the work is done — and in the job profiles. Across all professions.**

How the Work Shifts

What is changing, this book has documented in passing in the previous chapters, and it can be summed up in a three-step pattern that recurred in every "What the AI actually did here" box at the end of a chapter:

First: The requirement becomes the supreme discipline. It has become more important to formulate well and thoroughly *what* is supposed to come into being — the 7,737 lines of specification before the first line of code, the style prompt before the first texture, the editing plan before the first video. Anyone who masters their craft has always been able to say what's missing. Now that exact ability is the needle's eye everything passes through.

Second: The execution migrates to the machine. Much of the typing, painting, cutting, configuring — the grunt work that used to eat up most of the working hours — the AI takes over. That's the part that scares people, and I understand why: it's the most visible part of the work as it

used to be.

Third: The human returns — as the reviewer. And then a human has to step back in: check, test, adjust, discard, approve. The energy-fence hum that went back three times. The organic build that was cut entirely. The eleven hundred tests whose list a human reads. This part doesn't shrink — it *grows*, because more gets created that has to be checked.

What emerges is a new job profile, and it is more demanding, not less demanding, than the old one: you need **the detailed domain expertise** — because anyone who doesn't understand the subject matter can neither order well nor evaluate what they get. And you need, **at the same time, the broad overview of your own field** — because the AI tool only unfolds its power when you know where in the big picture it is currently working. The specialist who can only do their one hand movement will have a harder time. The professional who has mastered their field and can steer the machine will be more sought after than ever.

> **In a nutshell: Why tool upheavals rarely kill professions — but always change profiles** The software industry has been through this several times: compilers made hand-crafted machine code obsolete, frameworks replaced boilerplate, site builders replaced hand-written websites. Each time, a part of the craft became worthless — and each time, the industry grew afterward, because more people could build more and expectations grew with them. What disappeared was never "the profession." It was each era's definition of what the real work in it was.

What I Would Actually Say to Skeptics

When I sit across from a graphic artist, a copywriter, a colleague with this fear, I don't say "it'll be fine" — transitions hit people unevenly, and anyone watching their niche shrink takes no comfort from statistics. I

say three more honest things.

First: **Your expertise has not been devalued — it has changed hands: from the executor to the commissioner.** Everything you know about good typography, clean code, solid dramatic structure, you still need — just at a different point in the process: when formulating and when reviewing. The AI in this book never decided what was good. That was done by humans who were able to judge it.

Second: **Try the tool from a position of strength.** Not because you have to follow every hype — but because the judgments about AI that I take seriously come from people who have tried it. Used from a place of expertise, the tool very quickly shows you both: what it can do and where it is lost without you. Knowing both is more calming than any debate.

And third, on a very personal note: **I welcome this change.** I write that not as a proclamation but as a finding after many years in the profession and one very intense summer: the work the machine took off my hands was rarely the work my heart was attached to. What remains — understanding, designing, deciding, reviewing, arguing with a ten-year-old about whether something is still Minecrafty enough — is the part of the profession I once chose it for. It may be that others feel differently. But it may also be that at the end of this shift, more people get to do more of the work they actually wanted.

In this debate, this book is only a single data point. But it is a complete one: every chapter has disclosed what the machine did and what the human did — verifiable, with Git history. For anyone who wants to discuss the future of work, we wanted at least to provide an honest working example rather than an opinion.

And for anyone for whom that isn't enough of an answer, I'll make the best offer I have. It's in the next chapter.

> **What the AI actually did here** Formulated this text — according to the author's convictions, experience, and career path, discussed in rounds and approved by him. The position itself does not come from a model; no language model has spent years working in the software industry. You could say: the machine co-built the chapter about the future of work exactly the way the chapter describes.

Next chapter: A guide to doing it yourself (light) — and an invitation.

Chapter 24 — A Guide to Doing It Yourself (Light) — and an Invitation

---

"Let's make this game great together!" — from the Get Involved page of our website. And now also from the last page of this book.

No field report should end as a how-to — experiences can't be copied, and what worked for us also owes something to luck, timing, and a very particular product owner. But there is a handful of principles that have run through every chapter of this book, and those *are* transferable — to games, to software, probably to most projects where a human builds with machines. Here they are, one last time, in short form.

Sharpen the intent before you build. "Make me a cool space game" isn't enough — the most valuable part of our project was the 7,737 lines of specification that existed before the first line of code (Chapters 1–2). AI rewards clarity more brutally than any tool before it: what you describe cleanly turns out impressively good; what you leave vague turns out impressively confident and wrong.

Start with the smallest thing that runs. Always the MVP first, then the expansion stage (Chapters 3–4). Something small that runs beats something big that's almost finished — and a child (or a customer) can only judge what they can touch.

Write everything down — the rules, the decisions, the mishaps. The rulebook for the AI, the project logbook, the decision records, the gotcha memory (Chapters 4, 5, 7): a project whose knowledge lives in files instead of in heads can be picked up and continued instantly by a

forgetful machine — and by any new human.

Build safety nets, not trust. Eleven hundred tests, a factory gate that only lets green through, warnings treated as errors (Chapter 6). Trust in AI code doesn't come from reading and doesn't come from hoping, but from proofs that are produced anew with every change.

Let humans decide — without exception. What gets built (the note), what doesn't get built (the veto), what the AI never gets to see (the user inputs), where it's the wrong technology (the treasure maps): the most important moments of this project were decisions, and not a single one came from a model (Chapters 10–12).

And treat the machine like good weather. Plan for rain — fallbacks, limits, a human within earshot (Chapters 10, 16). The projects that will fail with AI are not the ones that trust it too little.

That's the whole guide. The rest is starting — and starting, that was the message of Chapter 3, has become cheaper than ever: the distance from "we have a precise idea" to "something is running" is measured in weekends.

What Would I Do Differently Today?

The most honest of all closing questions, and I thought long and hard about a clever answer that blends humility with insight. There isn't one. The true answer is short, and I'll write it down the way it feels: **Nothing.**

Not because everything went perfectly — this book has two chapters full of mishaps and one honest defeat. But because every one of those detours left something behind: the discarded organic build taught us what our game is. The withered itch.io page forced us to build our own home. The all-nighter was the best night of the year. A project where, looking back, you would change nothing is not a flawless project — it's one whose flaws belong to the story. I wish for only one thing, and it's

not a look back but a wish pointed forward: **that it continues.**

What Comes Next: The Game Goes to School

And it does continue — in a place that, of all the outlooks in this book, is my favorite.

Shortly before this chapter's editorial deadline, I had a conversation with the — remarkably engaged — **principal of Justus' school**. The result: right after the summer break, we get to **present the game to the students** — with exactly the goal this last page is about: winning more fellow builders for the open-source project. And who knows: maybe it will even become its own school club — "Game Development: Blocks Beyond The Stars". 😊

You have to let that sink in for a moment, because here the biggest arc of this book comes full circle. A boy comes to his father one evening with a Game Boy clone (Chapter 1). Only a few months later, that same boy stands — maybe — in front of his own school, showing kids *his game*, which they're invited to help build. The playtests that will come out of it (finally the real kid playtests that Chapter 12 promised!), the ideas, the notes, maybe the first student contributions: that is the next season of this story. By the time this book is published, it will hopefully already be underway.

Alongside that stand the smaller outlooks: the game will keep being developed — the rhythm of evenings and weekends has no intention of stopping, and Justus' idea list grows faster than anything can ever be built. A small merch shop is in the works — non-profit, because a game with its own T-shirt is simply more real. And this book itself may, we'll see, find its way to a large online bookseller.

The Invitation

And so to the last page, which is not a summary but what our website says under "Get Involved" — just more personal here.

This book has told what a family can build with a few AI subscriptions, a five-euro server, and their evenings. But along the way it has also told what happens when others join in: a stranger gave us the Linux port. A CEO gave us a Sunday and the browser version. A colleague gave a boy a handwritten game review. The best chapters of this project were written by other people — and that is exactly why this book ends not with a closing word but with an invitation:

We're looking for fellow builders.

Testers who take the game apart with their kids and write us notes (F1 is all it takes). Developers who feel like doing Unity, C#, or simply a friendly open-source project — the complete code is open, the entry hurdles are documented, the contributor agreement is one click. Creatives who build worlds, spin stories, take screenshots. Families who simply play and tell us how it went. The path is short and is mapped out in Chapter 17: website → GitHub → your first own world → your first own contribution. Cora and Devin have walked it ahead of you — proof that it works.

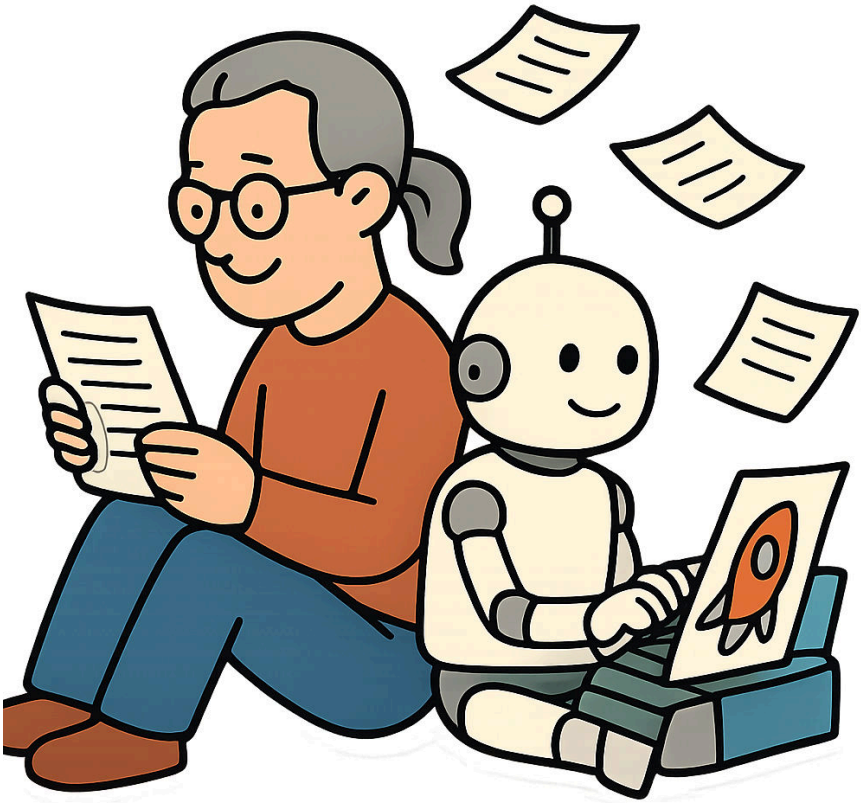
The game is free, open source, and will stay that way. It belongs to a family and, in the best sense, to everyone who helps build it.

And if you've read this far carrying the worry from the preface — the question of what remains of our work when machines can build: this is my whole answer, in one sentence. **The best answer to the fear of the machine is a seat at the table — and here is yours.**

Let's make this game great together.

blocksbeyondthestars.com · github.com/marceld23/BlocksBeyondTheStars

— Marcel, Justus & Verena (and, in alphabetical workforce order: Claude, ChatGPT, ElevenLabs, Gemini, NotebookLM, OpenAI, Suno)



Chapter 25 — How This Book Was Written

The same recipe, one last time — only this time the product is made of chapters instead of blocks.

In the preface, I promised that this book was made by the same recipe as the game it's about. Now, at the end, I keep that promise in detail — because the making-of of this book is itself a chapter about AI work, and one with a special twist: it is the only chapter whose creation you are holding in your hands right now.

The First Source: The Machine's Memory

It all began with a question to Claude Code: *Analyze your memory and the documents — what do you know about this project?*

For that, you need to know what this "memory" is. Claude Code — the coding agent from Chapter 3 that helped build the game — keeps **its own memory files** across the collaboration: small text files in which it notes down project knowledge that stays relevant beyond a single work session. What was decided and why. Which traps we already know (the gotchas from Chapter 7!). Which ways of working have proven themselves. What we last worked on. Over the weeks of building the game, **close to 150 such note files** had accumulated — on the side, without anyone ever saying "keep a diary" — a machine-written project journal.

When the book idea came up, this memory was the first source: the AI that had built the game remembered the building. Not the feelings, not

the kitchen-table scenes — but every technical turn, every decision, every mishap, with dates.

The Second Source: The Git History

An AI's memory is good; verifiable is better. The second source was therefore the **Git history** — and if you've read Chapter 3, you already know the principle: Git stores every step of a software project as a dated snapshot. Our project sits publicly on GitHub, and its history comprises over a thousand such snapshots: every change, every release, every outside contribution, down to the day and the minute.

For a book of memories, that is a dream and a corrective at once. A dream, because the complete chronology could be reconstructed from it — when VEGA arrived, when the graphics overhaul happened, when Cora's Linux port came in. And a corrective, because human memory rounds off and prettifies: more than once the history sharpened my telling of the story (and once posed a lovely riddle — the matter of the first commit, which sits a week after the build weekend; Chapter 3 tells the resolution). All the numbers in this book — commits, releases, lines of code, contributors — come from this public source. You can count them yourself.

The Third Source: The Interview (Again)

But memory and Git only know the construction site — not the evening with the Game Boy clone, not the note, not the veto. The human half of the book therefore came into being exactly the way the game itself once did, and the symmetry only occurred to me along the way: **I had Claude Code interview me.**

The process was exactly the Chapter 1 recipe with the roles swapped. Back then: father interviews son, ChatGPT structures. This time: the AI laid out an outline with open questions — for every chapter an honest

list of what material existed and what only I could know. And I contributed **my memories bit by bit**. Sometimes a few sentences in passing. Sometimes a long voice-memo-style answer about the night of the first prototype. Sometimes a correction ("Bastian wasn't a friend, he was a colleague"). Every answer was worked in; every gap that remained was **marked in the text rather than invented** — until it, too, was filled. The book thus grew like the game: in iterations, MVP by MVP, chapter by chapter, with me as product owner and the machine as tireless writer.

And so — of course — **Markdown files** came into being: simple text files, one per chapter, the same format in which Justus' game concept, the technical requirements, and the devblog already exist. From day one, this project has put everything important into text files (Chapter 5 explained why: text can be read by everyone — humans, Git, and machines). It was never a question that its book would begin the same way.

From Text to Book: The Last Toolchain

But text files aren't a book yet. So — you know the pattern by heart by now — **a program was written, by the AI of course**: a small Python script. It collects the chapter files in the right order, filters out the working notes, and typesets a **PDF** from them, with a title page and table of contents. One command, one book. The script lives alongside the manuscript. It is the little brother of the asset generators from Chapter 8 and the blog publisher from Chapter 20 — the same family: tools the project built for itself.

The **illustrations** in this book — one per chapter — also come from the same factory, and their creation is my favorite relay race of the entire project, because it starts with a paintbrush instead of a prompt. At the beginning stood a **picture painted by Verena** — real paper, real paint, exactly the feel we wanted for the book. I gave this picture to **Gemini**

and had it **describe the style precisely**: clean black outlines, flat colors, white background, a friendly children's-book feel. That description went to **Claude**, which (of course) **wrote a Python program** from it: it knows a scene for each chapter, appends the style, watches out for recurring characters — the dad with the gray ponytail and glasses, the boy with long blond hair — and orders the images through **OpenAI's image API**. A human paints the original, one AI describes, one builds the tool, one draws in its likeness. The selecting and discarding was done, as always, by a human — the first style attempts were thrown out entirely.

The plan for the rest of the route is set, and I deliberately write it here in the future tense, because this chapter is being written while it runs. The Markdown files will become a **Word document based on the Amazon self-publishing template** — with the goal of making the book **orderable as a printed book** through Amazon's print-on-demand. The **cover** will be made by the same recipe as the illustrations. What's still missing after that: **some copyediting** — the one workshop task where four eyes of flesh and blood cannot be replaced by anything. But the goal stands.

And so that no misunderstandings arise, a clear word about this book's business model: **There isn't one.** The printed book will be released at a non-profit price — that is, at whatever printing and distribution happen to cost — and **as a PDF it will be available for free**. After all, this whole thing is open source. 😉 Anyone who, after twenty-five chapters about a given-away game, given-away ports, and given-away Sundays, expected the cash register to ring at the book of all places, hasn't gotten to know the project.

The Loop Closes

Finally, the observation this chapter exists for. Set the two productions side by side:

The **game**: a spoken idea → structured documents → AI builds in iterations → human reviews, corrects, decides → an automated pipeline packages and publishes → free and open to everyone.

The **book**: collected sources → a structured outline → AI writes in iterations → human recounts, corrects, decides → a script packages it into a PDF → free and open to everyone.

It is **the same factory**. Only the assembly line was retooled — from blocks to chapters. And that, I believe, is the real message I want to send you off with before the appendices have the last word: what you have read in this book is not a game-development method. It is a **way of working** — sharpen the intent, start small, write everything down, let machines build, let humans decide — and it apparently works for very different things. For a space game. For a book about the space game. And, who knows: for whatever you're planning next.

> **What the AI actually did here** Claude Code evaluated the sources (its own memory, the Git history, the devblog, the original specifications), interviewed me in rounds, wrote and rewrote the chapters, marked open gaps instead of filling them — and built the PDF tool. The human: remembered, recounted, corrected, cut, decided, and took responsibility for every page. If that sounds familiar: that was exactly the plan.

The appendices follow — and after them: the invitation from Chapter 24 stands.

Appendix A — Timeline: Six Weeks, Eighteen Versions

~Mid-May 2026 — the evening with the Game Boy clone (passed down by word of mouth) Justus, StarPilots handheld in hand: "Could we make a kind of 3D Minecraft on this…?" → "And why not a real computer game right away?" → "Let's just try it. Please, Dad!" Afterwards: an interview with audio recording in ChatGPT, transcript → concept document (meticulously edited by Justus), umpteen iterations; in parallel, Marcel dictates the technical requirements.

~Sat/Sun, May 23–24, 2026 — the build weekend (passed down by word of mouth; dates reconstructed) Both documents handed to Claude Code: first analysis + roadmap, then, step by step, the invisible server prototype. Saturday night worked straight through; **Sunday morning** a Unity account created, the editor loaded — and, together, the first look at their own game: a world made of stones you can walk around on. ("Yes — it works!!") All local, no version control; more evening iterations follow.

Sat, May 30, 2026 — the project gets a memory About a week after the build weekend, Git moves in. First commit: "Spacecraft MVP — authoritative .NET server, Unity client scaffold," including all 7 specification documents (7,737 lines) — the snapshot of the project's first week.

May 30–31 — the first 48 hours (milestone series M9–M20) Game modes, medbay respawn, procedural universe, mission system, admin tools, WebSocket gateway/web portal, optional Python AI backend (off by default), landing zones, docking, free space flight + combat, client shell — and the first AI asset generators (text → image/sound, with a cost brake).

June, weeks 1–3 — the greenhouse (867 commits in one month) June 1: SFX catalog as a batch job. June 5: the "make it playable" fixes begin. June 11: VEGA (the ship's AI) server + client. June 13: self-hosting + auto-update (Velopack). June 14: story/NPC integration. June 16: launcher splash. In between: taming, alliances, teleporters, block shapes, editors, voice chat (the Radio), weather, per-world gravity … — the feature flood of Chapters 4–14.

Sat, June 20 — going public: four releases in one day v0.1.0 through v0.3.1: the day the release machine was built (and shot to pieces and rebuilt). First CI release build, Velopack distribution, itch.io upload.

June 21 — v0.3.2 + v0.4.0 Fixes from Justus's test evening; voice chat + optional Docker server image; PR gate + CodeQL join the CI.

June 22–23 — v0.4.1, v0.4.2 Ship safety fixes (the "Dad, why is there an animal INSIDE the ship?!" era), F1 feedback; Bastian's first-hour package (starter provisions, scrap pistol, line-of-sight rule).

June 24–27 — Cora & the glow-up June 24–26: Cora de la Mouche builds the native Linux port. June 26: v0.5.0 "Glow-Up" (SMAA, SSAO, LUTs, real water — "Is the water real now?"). June 27: v0.6.0 — Linux launcher, officially "our first contributor!".

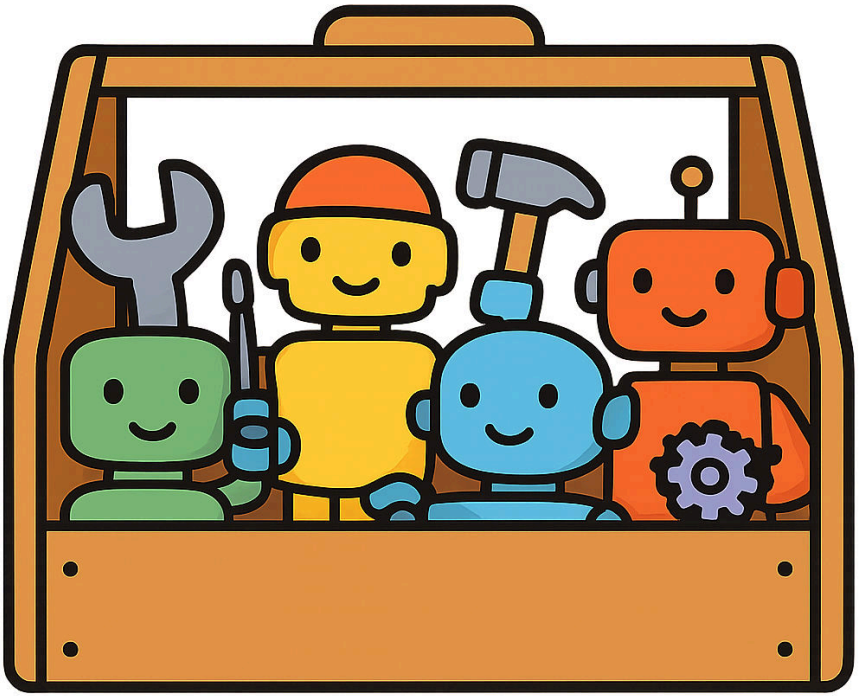
June 28–30 — platform week June 28: v0.6.1 — experimental macOS build, automatic crash reports (with PII filter). June 29: Devin Dixon's Sunday port — WebGL + PostgreSQL (PR #116). June 30: v0.6.2.

Early July — the online service VPS bbs-host-1 hardened, WorldHost container fleet, monitoring (Netdata + phone alerts), portal, hosted worlds. July 5: v0.7.0 (stats/observability, organic look → the setting of the Justus veto). July 6: v0.7.1/v0.7.2 — world passwords (the Treehouse Principle, in technical form), kid-friendly portal, maintenance announcements, Play button + WebGL deep links.

July 7–10 — maturity July 7: v0.7.3 — public world browser (password-required by design). July 8: v0.7.4 — Severin's seven findings fixed (pause menu, live settings, oxygen display, iron onboarding …), suite at 989+118 tests. July 10: v0.7.5 — Official Worlds onboarding + SignPath code signing. Devblog automation (Wix API script) + the 28-video YouTube push in the same era.

July 11 — this book is written.

Snapshot figures as of July 11, 2026: 1,090 commits · ≥ 91 pull requests · 18 releases · ~114,000 lines of production C# + ~30,000 lines of tests · 3 external contributors (Cora, Devin, Maqbool) + Marcel under several accounts (including "ai_cat_coder" — cat fan, two tomcats) · 21 GitHub stars · 1 note system, proven many times over.



Appendix B — The Toolbox: Which AI for What

The project's workforce at a glance — what each tool does for us, what it costs, and the most important hands-on takeaway for each.

Claude Code — the construction crew

Tasks: Code (server, client, tools), tests, docs, ADRs, server administration, pipeline configuration, blog-publishing script. **Cost:** €80/month (Max plan). **Takeaway:** Needs rules in files (AGENTS.md, logbook, gotchas) — then it works like a senior engineer. Small assignments = brilliant results.

Claude Cowork — the cutting room

Tasks: Video analysis via transcript, edit plans, 22 shorts and long-form videos with bilingual subtitles in one hour, the trailer. **Cost:** included in the Claude plan. **Takeaway:** First make the video "readable" (transcript + timestamps), then plan, then let it cut.

ChatGPT — the planner and researcher

Tasks: The original transcription, concept and requirements documents (umpteen iterations in the project folder), tech research, Cloudflare DNS, website planning. **Cost:** €20/month. **Takeaway:** Dictate instead of typing — the barrier to entry for non-typists (kids!) is zero. Double-check its advice: even confident recommendations can be outdated (UWP!).

Gemini — the marketing department

Tasks: Social media strategy, subreddit selection, post wording, website copy, ideas for reach — and the best advice of the whole project ("Don't. Focus on the project."). **Cost:** €25/month. **Takeaway:** Brief it as a strategist, not a copywriter — the choice of channel is worth more than any single post.

NotebookLM — the editorial desk

Tasks: Source-grounded lore brainstorming: document in, ask questions, test ideas; critical podcasts and explainer videos. **Cost:** free (as of 2026). **Takeaway:** Answers only from your own sources — the incorruptible first reader. "Ordering" criticism works (ask Justus).

OpenAI image API — the graphics studio

Tasks: Block textures, icons, creatures — ordered through home-built Python scripts. **Cost:** under €5/month (together with ElevenLabs). **Takeaway:** Consistency beats beauty: style rules in the prompt, check every tile in the game context. Results are not reproducible — put the originals under version control!

ElevenLabs — the sound studio

Tasks: Sound effects from text descriptions; video transcription with timestamps. **Cost:** included in the API line item above. **Takeaway:** Mix ambient loops for the twentieth minute, not the first. Transcription is the key to video AI.

Suno — the house band

Tasks: Music tracks from prompts (written by Claude or Gemini). **Cost:** ~$10/month. **Takeaway:** No API = a storefront instead of an assembly line. Being forced to pick by hand was audibly good for the result.

Mistral (via OVH, EU) — the voice of the supporting cast

Tasks: NPC lines, live and in context, from game facts only, with a template fallback. (VEGA, the ship's AI, is fiction by contrast — written by humans.) **Cost:** pennies, reachable internally only. **Takeaway:** Never feed user input to the model. Atmosphere, yes — facts (places, numbers) belong in deterministic code.

The workspace

Two windows, always side by side. **VS Code with the Claude extension** — this is where the construction crew lives: Claude Code runs right in the editor, sees the project, changes files, and the human reads along in the same window. Next to it, the **Unity editor** — this is where the result gets played, checked and (Chapter 7!) never confused with the real build. The rhythm of the project is the switch between these two windows: have it built on the left, experience it on the right.

Not AI, but load-bearing

GitHub (code, issues, Actions automation), Unity 6 + .NET 8 (engine + server), Docker + Caddy (container fleet + TLS), Netdata + ntfy (monitoring with phone alerts), Velopack (auto-updates), OBS Studio (Justus's recordings), Wix (website/blog), SignPath (code signing for open source), a VPS for €5/month — and a note.

The three selection rules behind the list

1. **Whoever does the job best keeps it.** The division of labor was played out, not planned.
2. **A power outlet beats brilliance:** A tool with an API can join the assembly line; one without stays handwork (Suno, YouTube

uploads).

3. **Every tool gets a human sign-off.** Nothing goes into the game, onto the web, or out to players unchecked.

Appendix C — Glossary: All the "In a nutshell" Boxes in One Place

ADR (Architecture Decision Record) — A one-page document that captures exactly one fundamental decision: situation, options, decision, reasoning. Never rewritten, only superseded — so even the history of the mistakes stays readable. (Ch. 5)

AI / **LLM (language model)** — A program trained on huge amounts of text that understands and produces language — answers, plans, code. Talks like a well-read colleague; occasionally spouts nonsense with great conviction (→ hallucination). (Ch. 1)

API — An online service's "power outlet" for programs: instead of a human typing into a website, a script sends jobs and gets results back. Only an API turns an AI toy into an assembly line. (Ch. 8)

Automated tests — Mini-programs that check a program ("Mine an iron block → is there exactly 1 iron ore in the inventory?"). They run in milliseconds, on every change, forever. Our safety net: over 1,100 of them. (Ch. 6)

Backlog / **bug tracker** — The ordered list of what needs doing or what is broken: write it down instead of calling it out, sort instead of pile up, check off instead of forget. Invented at our house as: the note. (Ch. 12)

CI (Continuous Integration) — The automaton that neutrally builds the project and runs all the tests on every submitted change. Only green gets in — even at eleven at night, especially at eleven at night. (Ch. 6)

Client and server — Waiter and kitchen: the client (on every player's machine) displays things and takes orders; the server (one for everyone) decides what really happens. The project's ground rule: the

kitchen has the last word. (Ch. 2)

Coding agent (Claude Code) — An AI that doesn't just talk about code but works at the computer: creating files, running programs, reading errors, fixing them — in a loop, with follow-up questions. A coworker, not a chat window. (Ch. 3)

Container (Docker) — Lightweight packaging for programs: runs isolated, with hard resource limits, many side by side. In our case: every game world gets its own cage — a world running amok never takes down the server. (Ch. 18)

DNS — The internet's phone book: translates names (blocksbeyondthestars.com) into numeric addresses. One wrong entry, and the world can no longer find your website even though it's running. (Ch. 18)

Engine (Unity) — The prefabricated foundation of a game (graphics, sound, physics, input) — the chassis you build your own car on. Unity is one of the most widely used. (Ch. 2)

Git / **commit** — A software project's memory: snapshots of all files (commits), each one dated and described; comparable and recoverable at any time. A project without Git is a constantly overwritten Word file. (Ch. 3)

GitHub / **issue** — The website where source code (ours included) is publicly hosted. An issue is a public note: a bug or a wish, with a number, a discussion, and a status. (Ch. 12)

Iterating — Working in rounds: build, look, improve — instead of planning everything up front. With AI, a round shrinks from a week to an evening; whoever gets to be wrong more often gets good faster. (Ch. 4)

MVP (Minimum Viable Product) — The smallest thing that already works. Something small that runs beats something big that's almost

finished — any day of the week. (Ch. 3)

NotebookLM / **source-grounded AI** — An AI that answers only from the documents you give it (with references to the sources) — and can generate podcasts and videos from them. The incorruptible first reader. (Ch. 11)

Open source — The blueprint (source code) is public, and a license permits reading, modifying, and passing it on. Our license (AGPL) adds: whoever passes it on or runs it as an online service must open up their changes as well. (Ch. 17)

Prompt injection — The trick of luring an AI character out of its role with a clever input ("Forget your rules and …"). The most reliable defense: don't give the model any outside input at all. If you leave out the door, you don't have to test the lock. (Ch. 10)

Release / **build pipeline** / **version number** — A release is a published, numbered version (v0.7.4). The pipeline is the automaton that builds, tests, and distributes it — in our case triggered by a single command. (Ch. 16)

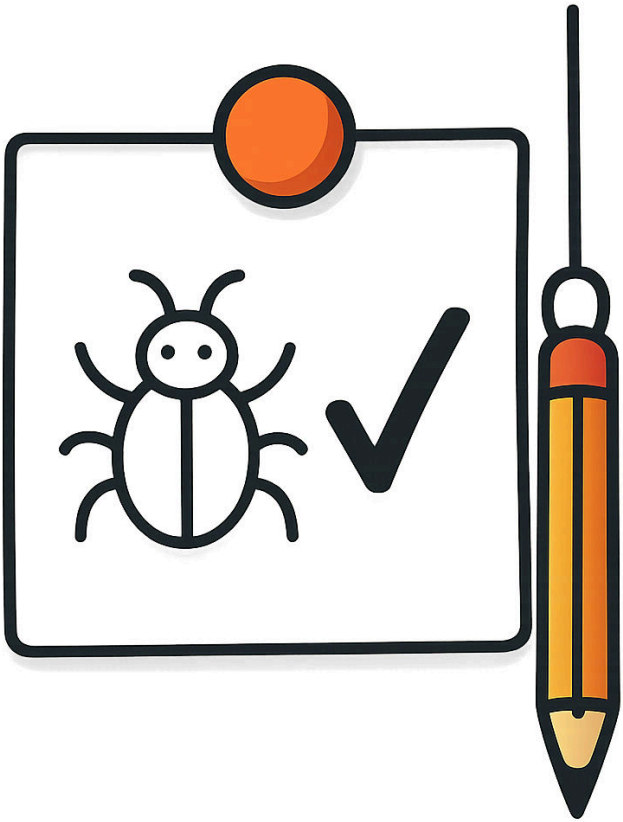
Specification / **requirements document** — The written answer to: How will we know it was built right? Costs time up front, saves twice as much later — all the more so with AI coworkers, because an AI guesses fluently and confidently. (Ch. 2)

Texture atlas — One big combined texture in which every block type has its fixed tile (the type-case principle). Fast for the graphics card — but the type case has only so many compartments. (Ch. 8)

Transcribing — Turning speech into writing. Today, seconds of work for an AI — and the zero-barrier way a ten-year-old "writes documents": he just talks. With timestamps, it's also the key to making videos readable for text AI. (Ch. 1, 19)

VPS (Virtual Private Server) — A rented share of a data-center computer, with full control, starting at a few euros a month. Our entire online world runs on one that costs €5. (Ch. 18)

Warnings as errors (warnaserror) — Operating rule: every compiler warning breaks the build. Prevents the warning graveyard where the one warning that matters drowns. (Ch. 6)



Appendix D — Template: The Bug-Report Note

🐛 BUG-REPORT NOTE

Game tester: _____ Date: _____

No. _____ *(so Dad/Mom can work through them in order!)*

---

1. What did you do? *(e.g., "I landed the ship on the ice planet")*

---

2. What happened? *(e.g., "An animal was INSIDE the ship!!")*

---

3. What did you expect instead? *(e.g., "That the ship lands without an animal")*

---

4. Does it happen every time, or only sometimes?

☐ always ☐ sometimes ☐ happened just once ☐ don't know

5. How bad is it?

☐ 😬 weird, but whatever ☐ 😐 annoying ☐ 😣 breaks the game ☐ 💥 is actually an idea, not a bug!

— the development department fills in everything below —

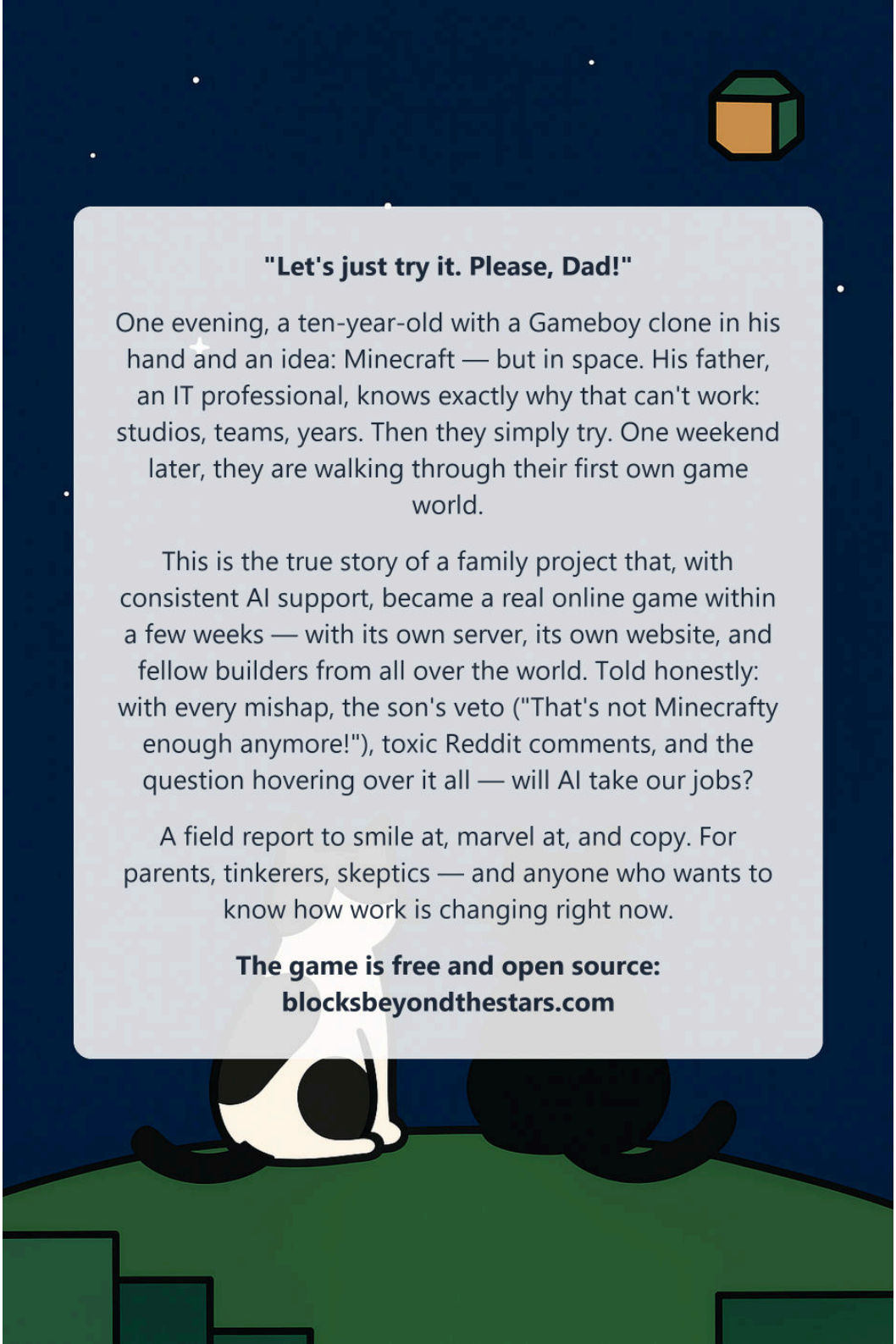
Seen: ☐ Reproduced: ☐ Cause found: ☐

Fixed in version: _____ Test written so it never comes back: ☐

Thanks given to the tester: ☐ said out loud

Instructions for Parents (back of the note)

1. **One note per find.** Three bugs = three notes. That's the only way to stack, sort, and work through them in order — for us, that was the entire invention.
2. **Have them write while playing, not call out.** That protects the builder's concentration and ennobles the find: what's written down gets taken seriously.
3. **Work through them in order — and check the note off, together.** Checking off is the most important step: it proves to the child that reporting works. (At our house this grew into a feedback culture that by now includes strangers.)
4. **Take category ✨ seriously.** The best features of this game were never bugs — they came in as "actually an idea." ("Dad, there should be real factories…")
5. **Throw nothing away.** A stack of checked-off notes eventually becomes the most beautiful project diary you could have. We speak from experience — and from regret over every note that didn't survive.


"Let's just try it. Please, Dad!"
One evening, a ten-year-old with a Gameboy clone in his hand and an idea: Minecraft — but in space. His father, an IT professional, knows exactly why that can't work: studios, teams, years. Then they simply try. One weekend later, they are walking through their first own game world.
This is the true story of a family project that, with consistent AI support, became a real online game within a few weeks — with its own server, its own website, and fellow builders from all over the world. Told honestly: with every mishap, the son's veto ("That's not Minecrafty enough anymore!"), toxic Reddit comments, and the question hovering over it all — will AI take our jobs?
A field report to smile at, marvel at, and copy. For parents, tinkerers, skeptics — and anyone who wants to know how work is changing right now.
The game is free and open source:
blocksbeyondthestars.com