Blocks Beyond the Stars

The Making of

Wie eine Familie mit KI ein Computerspiel baute

Marcel, Justus & Verena

Blocks Beyond the Stars - The Making of

Wie eine Familie mit KI ein Computerspiel baute

Marcel, Justus & Verena

blocksbeyondthestars.com

Umschlaggestaltung: Erstellt von Marcel Dütscher unter Verwendung von KI-Bildgenerierung (OpenAI API).

Satz und Layout: Marcel Dütscher

Hinweis des Autoren: Wie im Buch detailliert beschrieben, wurden für die Erstellung dieses Textes und der Illustrationen (inkl. Cover) KI-Werkzeuge als Assistenten genutzt. Die redaktionelle Verantwortung, Kuration und Freigabe lagen durchgehend bei den menschlichen Autoren.

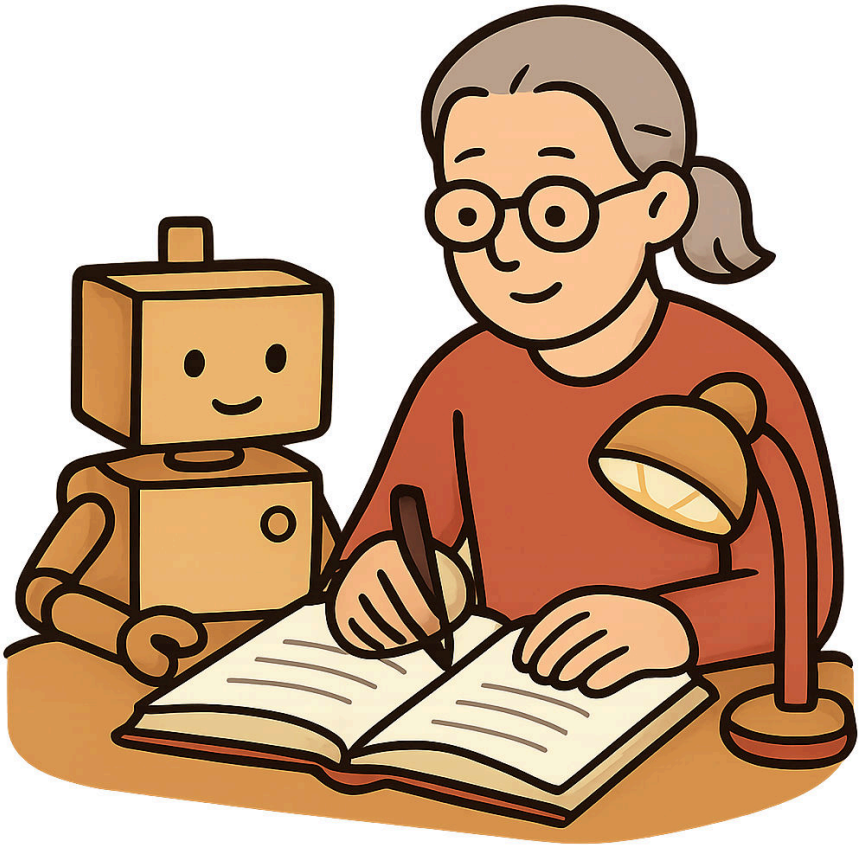
ISBN: 979-8-1868-3755-8

Inhalt

Widmung

Dieses Buch widme ich meinem Sohn Justus:

Bewahre dir immer deine Phantasie, denn mit ihr kannst du großartige Dinge bauen!



Vorwort — Offenlegung, bevor es losgeht

Dieses Buch handelt davon, wie eine Familie mit künstlicher Intelligenz ein Computerspiel gebaut hat — den Code, die Grafik, den Sound, die Musik, die Server, das Marketing, die Videos. Es wäre einigermaßen absurd, wenn ausgerechnet das Buch darüber heimlich auf die klassische Art entstanden wäre. Ist es nicht. Also, der Vollständigkeit halber und bevor du die erste Seite umblätterst:

Auch dieses Buch ist mit KI geschrieben.

Genauer gesagt: Es entstand nach exakt demselben Rezept wie das Spiel, von dem es handelt — und wer das Buch gelesen hat, wird das Rezept in seiner eigenen Entstehung Schritt für Schritt wiedererkennen. Am Anfang stand kein leeres Blatt, sondern eine **Bestandsaufnahme**: Welche Quellen gibt es — die Versionsgeschichte mit über tausend datierten Einträgen, das 6.700-Zeilen-Logbuch des Projekts, die Devblog-Artikel, die ursprünglichen Anforderungsdokumente, die Erinnerungen der Familie? Daraus wurde ein **Plan**: eine Gliederung mit dem ehrlichen Vermerk, wofür Material vorliegt und wo nur Erinnerung trägt. Dann wurde **kapitelweise gebaut** — Rohfassung, Feedback, Verfeinerung, ein Kapitel nach dem anderen, wie beim Spiel ein Feature nach dem anderen. Die Maschine schrieb; der Mensch erzählte zu, korrigierte, strich und entschied. Wo eine Information fehlte, wurde sie **nicht erfunden**, sondern als offene Frage markiert und beim Autor eingesammelt. Die Anekdoten in diesem Buch stammen aus Marcels und Justus' Erinnerung, die Fakten aus der öffentlichen Projektgeschichte — und beides ist im Text auseinanderzuhalten.

Was heißt das für dich als Leserin oder Leser? Zweierlei, finde ich.

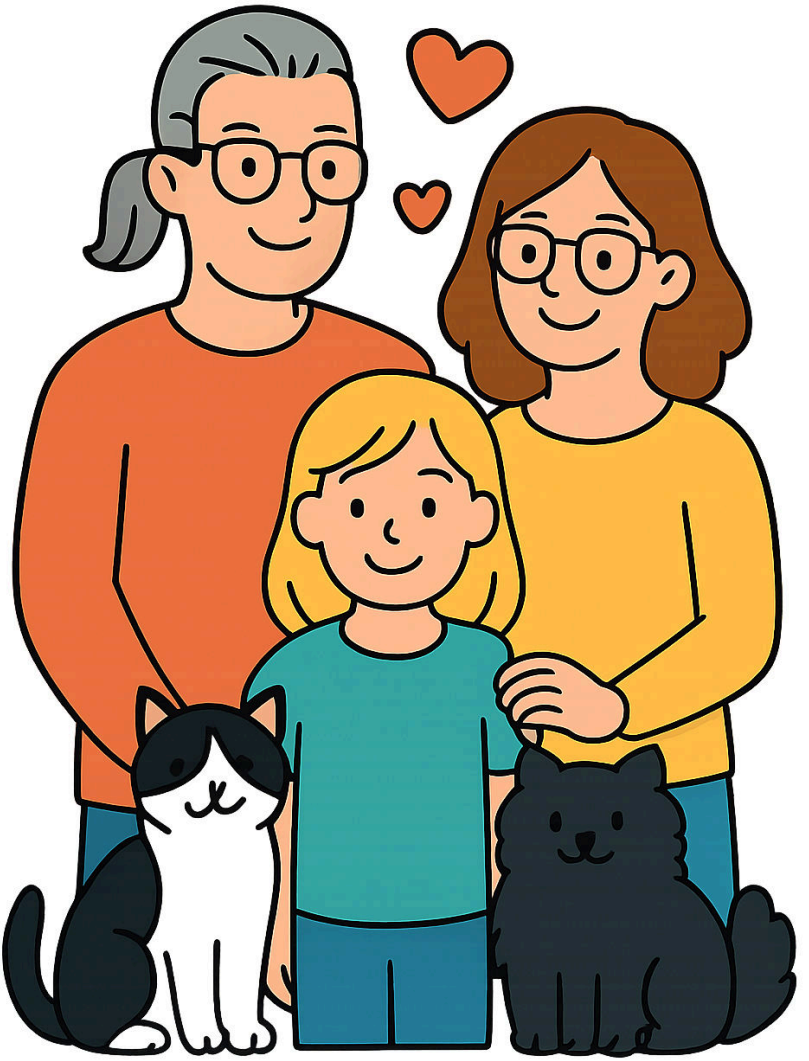
Erstens eine Zusage: **Alles Berichtete ist wahr im besten uns möglichen Sinne** — die Zahlen stammen aus der öffentlich einsehbaren Versionsgeschichte (das Projekt ist Open Source; du kannst jede Behauptung nachprüfen, und dazu laden wir ausdrücklich ein). Die Zitate von Justus sind echte Zitate, die Pannen sind echte Pannen. Eine KI hat diesem Buch die Sätze gebaut. Was die Sätze behaupten, verantworten Menschen.

Und zweitens eine Einordnung, die vielleicht die eigentliche Botschaft des ganzen Projekts ist: Die Frage „Ist das von einer KI oder von einem Menschen?" wird in den kommenden Jahren immer öfter die falsche Frage sein. Die richtige lautet: **Steht jemand dahinter, der es geprüft hat, dessen Geschichte es ist und der dafür geradesteht?** Bei diesem Buch: ja. Ein Vater, eine Mutter, ein Zehnjähriger — und eine Werkstatt voller Maschinen, die tun, was Werkstätten immer getan haben: Sie machen möglich, was eine Familie allein nicht schaffen würde.

Eines noch, bevor es losgeht — und ich sage es gleich hier, weil ich weiß, dass manche dieses Buch mit gemischten Gefühlen aufschlagen: Ja, hier wurde KI überall dort eingesetzt, wo sonst Menschen arbeiten. Programmierer, Grafikerinnen, Texter, Musikerinnen, Sounddesigner. Wenn dir bei diesem Gedanken mulmig wird — vielleicht, weil es dein Beruf ist —, dann lies trotzdem weiter. Diese Sorge wird in diesem Buch nicht weggelächelt; sie bekommt ein eigenes Kapitel (Kapitel 23), und dort antworte ich so ehrlich, wie ich kann: als jemand, der von dieser Veränderung nicht nur schreibt, sondern in ihr arbeitet.

Der Rest ist die Geschichte. Sie beginnt, wie sie enden wird: mit einer Bitte eines Kindes.

— Marcel (geprüft, gestrichen, verantwortet) & Claude (getippt)



Danksagung

Dieses Buch — wie das Spiel, von dem es erzählt — gäbe es nicht ohne diese Menschen.

Meine Frau **Verena**, die mich bei allem unterstützt.

Mein Sohn **Justus**, der das hier erst möglich macht.

Mein **Papa**, der mir schon mit sechs Jahren den ersten Computer gekauft hat — einen C64. Da hat das alles angefangen.

Und meine **Mama**, der ich meine Kreativität verdanke. Und meine Bastelwut.

Danke.

Kapitel 1 — „Lass es uns doch ausprobieren. Bitte, Papa!"

„Und wieso können wir nicht gleich ein richtiges Computerspiel machen? In 3D?" — Justus, 10, an einem Abend, der teuer werden sollte

---

Die Vorgeschichte: Wir hatten das schon mal gemacht — nur kleiner

Diese Geschichte beginnt nicht bei null, und das ist wichtig, um sie richtig zu verstehen. Als der Abend kam, um den es gleich geht, hatten Justus und ich bereits eine kleine gemeinsame Werkstatt-Karriere hinter uns: eine Handvoll Projekte, alle nach demselben Muster — Kind hat Idee, Vater hat Feierabend, KI schreibt den Code.

Wir hatten **Mikrocontroller** programmieren lassen, jene kleinen Computerchen, die in Bastelprojekten blinken und piepsen. Daraus wurde **Pixel-Pets**: eine Art modernes Tamagotchi, ein pixeliges Haustier zum Füttern und Pflegen, selbst gebaut. Dann **StarPilots**: ein Arcade-Spiel auf einem Gameboy-Clone — einem dieser Retro-Handhelds zum Selberbestücken. Und irgendwann baute Justus **ganz allein** den Prototyp eines Browserspiels namens Catapult-Blade — sein erstes eigenes Projekt, ohne Papa an der Tastatur. (Alle drei liegen übrigens bis heute öffentlich auf GitHub; die Vorgeschichte dieses Buches ist nachprüfbar.)

Nichts davon war groß. Aber etwas Entscheidendes war passiert, und es ist die eigentliche Pointe der Vorgeschichte: **Wir hatten als Familie bereits gelernt, dass Ideen nicht im Kopf bleiben müssen.** Dass

zwischen „wäre es nicht cool, wenn…" und einem Ding, das man in der Hand hält, kein Ozean mehr liegt, sondern ein paar Abende. Dieses Wissen — mehr Gefühl als Wissen — ist der Nährboden, auf dem alles Folgende wuchs. Kinder, die einmal erlebt haben, dass ihre Idee real wird, hören nicht mehr auf, Ideen zu haben.

Der Abend

Und dann kam dieser Abend. Justus stand vor mir, in der Hand den Gameboy-Clone mit seinem Arcade-Spiel darauf, und sagte:

„Papa, können wir hierdrauf eine Art 3D-Minecraft machen? Aber im Weltraum, mit Raumschiffen?"

Meine Antwort war die eines Ingenieurs: „Ne. Das Ding hat nicht genug Speicher."

Technisch korrekt. Und komplett am Punkt vorbei, wie sich zwei Sekunden später zeigte, denn Justus zuckte nicht einmal:

„Und wieso können wir nicht gleich ein richtiges Computerspiel machen? In 3D? Oder eines für die Xbox?"

Lass diesen Dialog kurz sacken, denn er enthält das ganze Buch in vier Sätzen. Das Kind stößt an eine Grenze — und sein Reflex ist nicht, die Idee zu schrumpfen, sondern **die Plattform zu wechseln**. Der Erwachsene dagegen… nun. Meine erste Reaktion war sehr erwachsen, und ich zitiere sie zu meiner eigenen Beschämung wörtlich: „Puh. Ich glaub nicht, dass das so einfach geht."

Erwachsene denken oft innerhalb von Schranken, die sie selbst gebaut haben. Ich wusste ja, was ein „richtiges Computerspiel" bedeutet: Jahre, Teams, Budgets — das ganze Besserwisser-Wissen aus einem Berufsleben in der IT. Justus wusste das alles nicht. Er wusste nur, was er in unserer kleinen Werkstatt gelernt hatte: dass Ideen baubar sind.

Und so sagte er den Satz, mit dem dieses Projekt wirklich beginnt — nicht als Vision, nicht als Konzept, sondern als das, was er war: eine Bitte.

„Lass es uns doch ausprobieren. Bitte, Papa!"

Gegen dieses Argument ist kein Erwachsenen-Einwand gewachsen. Wir fingen noch am selben Abend an.

Erwartungsmanagement (oder: wie man sich irrt)

Eines will ich ehrlich festhalten, weil es die vielleicht tröstlichste Fußnote dieses Buches ist: **Ich habe nicht an das Projekt geglaubt.** Jedenfalls nicht an das, was daraus wurde. Meine innere Zielmarke an jenem Abend war defensiv: Erwartungen des Sohnes klein halten, Enttäuschung vorbeugen. Ich dachte wörtlich: *Und wenn am Ende „nur" ein Konzept für ein Spiel herauskommt — dann ist das ja auch schon ein Riesenerfolg.* Ein Dokument. Das war mein Ehrgeiz.

Ich erzähle das, weil da draußen vermutlich gerade viele Eltern, Bastler und Skeptiker mit derselben inneren Bremse vor derselben Entscheidung stehen. Die Bremse ist vernünftig. Sie ist erfahrungsgesättigt. Und sie ist, Stand 2026, schlicht nicht mehr aktuell — aber das weiß man erst hinterher. Man muss es ausprobieren. Bitte.

Das Interview

Wie fängt man an? Nicht mit Code. Wir fingen mit einem Gespräch an — und zwar mit einem, das ich bewusst wie ein **Interview** aufzog: Ich öffnete einen ChatGPT-Chat, startete die Audio-Aufnahme und unterhielt mich dann mit Justus über seine Spielidee. Er erzählte, ich spielte den Interviewer — nachfragen, weiterführen, hier und da einen Punkt ergänzen, aber im Kern: **sein** Spiel, **seine** Antworten. Was ist das für ein Schiff? Was macht man als Erstes? Was passiert, wenn

man stirbt? Ist das Schiff ein Menü oder ein Ort? (Ein Ort. Kategorisch. Mit Räumen: Cockpit, Medbay, Werkstatt, Frachtraum — wer das spätere Spiel kennt, erkennt jede dieser Kinderantworten wieder.)

> **Kurz erklärt: KI, ChatGPT und Transkription** Wenn in diesem Buch „KI" steht, ist fast immer ein *Sprachmodell* gemeint: ein mit riesigen Textmengen trainiertes Programm, das Sprache versteht und erzeugt — Antworten, Strukturen, Pläne, Code. ChatGPT ist das bekannteste. Und es kann zuhören: *Transkribieren* heißt, Gesprochenes in Schrift zu verwandeln — heute Sekundenarbeit. Für uns hieß das: Die Hürde für einen Zehnjährigen, ein „Konzeptdokument zu verfassen", war exakt null. Er musste nur reden. Das kann er.

Aus dem Transkript dieses Gesprächs ließen wir ChatGPT dann ein **strukturiertes Anforderungsdokument** erstellen. (Das Roh-Transkript selbst existiert nicht mehr — es ist irgendwo in den Tiefen des Chat-Verlaufs verloren gegangen. Ich bedaure das heute; es wäre das Gründungsdokument schlechthin gewesen. Falls du es nachmachst: Sichere das Transkript. Man weiß am Tag eins nie, welche Datei später ein Buch aufschlagen möchte.)

Und jetzt kommt mein Lieblingsteil, denn hier wird aus der netten Anekdote ein Arbeitsprozess: **Justus las das Dokument. Minutiös.** Nicht überflog — las. Und warf raus, was er nicht haben wollte. Es war sein Spiel, und das Dokument hatte sich danach zu richten, nicht umgekehrt. Danach feilten wir in mehreren Iterationen im Chat an diesem ersten Entwurf: präzisieren, umstellen, streichen, ergänzen — die Runden, aus denen am Ende jene 2.176 Zeilen wurden, deren Kernsätze bis heute das Spiel beschreiben:

> *„Minecraft trifft Raumschiff-Ausbau, Planeten-Erkundung und Survival-Crafting im Weltraum."*

> *„Ich starte mit einem kleinen, einfachen Schiff. Durch Erkundung, Bergbau, Crafting und Forschung baue ich mir nach und nach mein eigenes, größeres, besseres Raumschiff und erreiche dadurch immer gefährlichere und spannendere Welten."*

Aus diesen Feil-Runden stammt auch das Zitat des Tages, das ich seither jedem erzähle, der wissen will, ob Kinder den Umgang mit KI „schon verkraften". Justus, nach einer weiteren wortreich lobenden ChatGPT-Antwort, trocken:

„Papa, die KI lobt einen aber oft. Und das ist meistens ja gar nichts wert…"

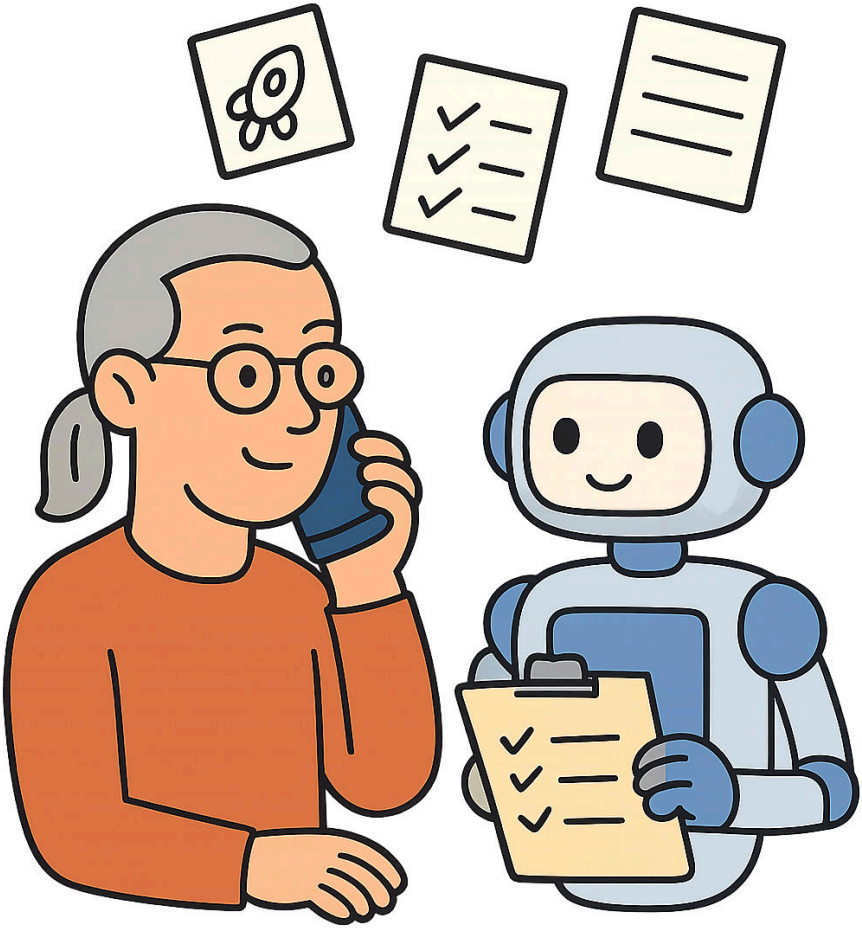
Stolzer Papa. Da hat ein Zehnjähriger in der zweiten Woche etwas durchschaut, wofür Erwachsene Medienkompetenz-Seminare buchen: dass maschinelle Freundlichkeit keine Information ist. Er hat das Lob nicht abgestellt — er hat aufgehört, es zu *zählen*. Seither gilt bei uns: Die KI darf loben, so viel sie will. Ernst genommen wird, was sie *baut*.

Was dieser Abend hinterließ

Am Ende dieser Phase — noch war keine Zeile des Spiels programmiert — existierte etwas, das die meisten Hobbyprojekte nie besitzen: ein vollständiges, vom Product Owner persönlich redigiertes Konzept. Und, wichtiger: eine Methode. Reden statt tippen. Interviewen statt diktieren. Das Kind entscheidet, die KI strukturiert, der Vater moderiert.

Parallel dazu hatte ich begonnen, die zweite Hälfte der Wahrheit einzusprechen — meine technischen Gedanken, gespeist aus jahrelanger Erfahrung in der Softwareentwicklung: Wie baut man so etwas, damit es hält? Daraus wurde das zweite Dokument, das die folgenreichsten Sätze des Projekts enthält. Davon erzählt das nächste Kapitel.

> **Was die KI hier wirklich getan hat** ChatGPT hat zugehört (Audio-Transkription), das Vater-Sohn-Interview in ein strukturiertes Konzeptdokument verwandelt und es in mehreren Runden mitüberarbeitet — 2.176 Zeilen am Ende. Es hat dabei keine einzige Spielidee beigesteuert und (Justus' Beobachtung!) deutlich zu viel gelobt. Die Ideen: vom Kind. Die Streichungen: vom Kind. Die Struktur: von der Maschine. Die Bitte, überhaupt anzufangen: die wichtigste Zeile des Projekts, und sie kam von einem Menschen, 10 Jahre alt.



Kapitel 2 — Vom Traum zum Lastenheft

„Auch lauffähig auf Raspberry Pi 5 unter Linux." — aus dem technischen Anforderungsdokument, das später niemand mehr befragen musste — es hatte recht

---

Das Konzeptdokument aus Kapitel 1 beschreibt einen Traum: Weltraum-Pionier, eigenes Schiff, unendliche Welten. Was es mit Absicht *nicht* beschreibt, ist die unromantische Hälfte: Womit baut man das? Wo läuft es? Was passiert, wenn zwei Spieler gleichzeitig denselben Block abbauen? Wer hat recht, wenn das Spiel auf meinem Rechner etwas anderes behauptet als auf deinem?

Also folgte auf das erste Diktat ein zweites. Wieder eingesprochen, wieder in ChatGPT, wieder in — und das ist wörtlich zu nehmen — **zig Iterationen**, in einem eigenen Projektordner, in dem sich mit der Zeit zahlreiche Chats stapelten. Herausgekommen ist ein Dokument: 987 Zeilen, Titel „Technisches Anforderungsdokument". Wo das erste Dokument klingt wie ein begeistertes Kind, klingt das zweite wie ein vorsichtiger Ingenieur. Beides war Absicht. Es sind die zwei Stimmen, die ein Projekt braucht.

> **Kurz erklärt: Was ist eine Spezifikation (und warum der Aufwand)?** Eine Spezifikation — altmodisch: ein Lastenheft — ist die schriftliche Antwort auf die Frage: *Woran erkennen wir, dass das Ding richtig gebaut ist?* Sie kostet vorne Zeit und spart sie hinten doppelt, weil niemand mehr raten muss. Das galt schon, als Menschen für Menschen bauten. Für KIs gilt es verschärft: Eine KI rät nicht zögernd, sondern flüssig und selbstbewusst. Je

> genauer das Dokument, desto weniger bleibt zu raten.

Die drei Sätze, die alles entschieden

In diesen 987 Zeilen stehen drei Festlegungen, die das Projekt bis heute tragen. Ich nenne sie hier schon beim Namen, auch wenn ihre Geschichten erst in späteren Kapiteln spielen:

Erstens: Der Server hat immer recht. Das Spiel besteht aus zwei Programmen — einem *Client*, den jeder Spieler bei sich laufen hat, und einem *Server*, bei dem alle Fäden zusammenlaufen. Und die Grundregel lautet: Der Client zeigt nur an; die Wahrheit wohnt beim Server. Wenn dein Bildschirm behauptet, du hättest einen Diamantblock abgebaut, der Server sieht das aber anders — dann hattest du keinen. Diese Regel klingt technisch, ist aber eine Charakterfrage: Sie macht Schummeln strukturell schwer und Streit unmöglich. (Warum das für eine KI als Mitarbeiterin doppelt wichtig ist, erzählt Kapitel 5.)

> **Kurz erklärt: Client und Server** Stell dir ein Restaurant vor. Der *Client* ist der Kellner an deinem Tisch: freundlich, schnell, zeigt dir die Karte und bringt, was du bestellst. Der *Server* ist die Küche: Dort wird tatsächlich gekocht, dort wird entschieden, was es gibt und was aus ist. Ein Kellner, der auf eigene Faust Gerichte erfindet, macht kurz Eindruck und dann Chaos. In Online-Spielen ist es genauso — deshalb gilt: Die Küche hat das letzte Wort.

Zweitens: C# und ein Server zum Selberhosten — bis hinunter zum Bastelrechner. Das Dokument legt die Programmiersprache fest (C#, gesprochen „C sharp") und die Spiel-Engine (Unity). Dazu kommt eine Anforderung, die ich bis heute gerne zitiere, weil sie so herrlich konkret ist: Der Server soll *„auch lauffähig auf Raspberry Pi 5 unter Linux"* sein — einem Bastelcomputer für unter hundert Euro, kaum größer als eine Scheckkarte. So oder so war die Wirkung enorm: Wer für das schwächste Gerät plant, baut automatisch sparsam. Diese eine Zeile

hat später verhindert, dass unser Server ein verfressenes Monstrum wurde — und sie hat den Weg zu dem Fünf-Euro-Mietserver geebnet, auf dem heute unsere öffentlichen Welten laufen (Kapitel 18).

Drittens — und das ist meine Lieblingszeile: Testabdeckung von Anfang an. Schon im Diktat, noch bevor die erste Zeile Spielcode existierte, stand fest, dass zu jedem Stück Programm automatische Tests gehören — kleine Prüfprogramme, die das Verhalten festnageln. Das war keine späte Reue nach dem ersten Datenverlust, sondern Teil der Bestellung. Warum das bei einem KI-Projekt nicht Kür, sondern Überlebensfrage ist, bekommt ein eigenes Kapitel (Kapitel 6: „Vertrauen ist gut, 1.100 Tests sind besser"). Hier nur so viel: Von allen Entscheidungen dieser Phase hat keine mehr Abende gerettet als diese.

Zig Iterationen — und wie sich das anfühlte

Ich möchte den Prozess nicht glatter erzählen, als er war. „Zig Iterationen" heißt: Es gab Fassungen, die in die falsche Richtung liefen. Es gab Nachfragen von ChatGPT, die klügere Antworten verdienten, als ich sie an dem Abend hatte. Es gab Momente, in denen ich einen Vorschlag las und erst beim zweiten Kaffee merkte, dass er zwar souverän klang, aber unserem Konzept widersprach.

Eine dieser frühen Fassungen habe ich aufgehoben, und sie ist ein kleines Zeitdokument. Es ist eine komplette *Art-Direction* — also eine Stilvorgabe für den Look des Spiels —, die ChatGPT auf die Frage lieferte, wie ein Blockspiel aussehen muss, damit es nicht billig wirkt. Die Antwort umfasst viele Seiten und liest sich stellenweise wie aus einem Lehrbuch für Spielegrafik: *„stylized voxel sci-fi with premium lighting"*, blockig bleiben, aber Licht, Materialien und Atmosphäre auf Profi-Niveau; jeder Planet eine eigene Farbstimmung; und als Formel: **„Minecraft-artige Baulogik + No-Man's-Sky-artiges Entdeckungsgefühl."**

Zwei Dinge an diesem Dokument rühren mich heute. Das erste: Es empfiehlt an einer Stelle mit großer Selbstverständlichkeit eine Technik-Plattform namens UWP — ein Microsoft-Format, das damals schon auf dem absteigenden Ast war und das wir nie benutzt haben. Die KI von damals war klug und trotzdem nicht allwissend; die Entscheidung, welchem Rat man folgt, blieb bei uns. Das zweite: Auf denselben Seiten schlägt ChatGPT vor, wir sollten unbedingt eine „Art Bible" anlegen — ein verbindliches Stilhandbuch. Wer heute in unser öffentliches Code-Archiv schaut, findet dort eine Datei namens `ART_BIBLE.md`. Der Vorschlag aus dem Diktier-Zeitalter ist real geworden, fast auf den Buchstaben. Es gibt wenige Projekte, bei denen man die Reise einer Idee so lückenlos nachverfolgen kann: vom gesprochenen Satz über das KI-Dokument bis zur Datei im fertigen Spiel.

> **Kurz erklärt: Was ist eine Engine? Was ist Unity?** Eine Spiel-*Engine* ist der vorgefertigte Unterbau eines Spiels: Sie kümmert sich um Bilder auf dem Bildschirm, Ton, Physik und Eingaben, damit man nicht bei null anfangen muss — wie ein Fahrgestell, auf das man sein eigenes Auto baut. *Unity* ist eine der verbreitetsten Engines der Welt; unzählige bekannte Spiele laufen darauf. Unser Spiel besteht also aus: Unity-Client (das Fahrgestell mit unserer Karosserie) plus selbst gebautem C#-Server (die Küche, siehe oben).

Das Paket für den nächsten Mitarbeiter

Ende dieser Phase lagen zwei große Markdown-Dateien auf der Festplatte — das Konzept aus Kapitel 1 und dieses technische Anforderungsdokument — plus eine Handvoll Detail-Spezifikationen, die in weiteren Diktier-Runden entstanden waren: über Admin-Werkzeuge, einen Missions-Editor, einen möglichen Web-Client. Insgesamt, das habe ich später einmal nachgezählt, waren es **7.737 Zeilen Spezifikation, bevor die erste Zeile des Spiels programmiert**

war.

Ich weiß, wie das klingt: nach Bürokratie, nach dem Gegenteil von „an einem Wochenende ein Spiel bauen". Aber genau das Gegenteil ist wahr, und es ist die zentrale These dieses Buches: **Diese Dokumente sind der Grund, warum das Wochenende funktioniert hat.** Denn der Mitarbeiter, dem wir dieses Paket als Nächstes übergeben würden, liest 7.737 Zeilen nicht in einer Woche, sondern in Sekunden. Er vergisst nichts davon. Er fragt nicht nach dem dritten Bier, was noch mal mit dem Respawn war. Er braucht nur eines, um Außergewöhnliches zu leisten: dass jemand vorher aufgeschrieben hat, was gemeint ist.

Dieser Mitarbeiter heißt Claude Code. Und was passierte, als er das Paket bekam, davon erzählt das nächste Kapitel — es dauert genau ein Wochenende.

> **Was die KI hier wirklich getan hat** ChatGPT hat eingesprochene Technik-Wünsche in ein strukturiertes Anforderungsdokument verwandelt (987 Zeilen), in zig Runden Alternativen aufgezeigt, recherchiert und begründet — inklusive Quellenangaben zu Engine-Dokumentation. Es lag dabei nicht immer richtig (UWP!). Entschieden hat es nichts: Architektur-Grundsätze wie „der Server hat recht", C#, Selbsthosting und Tests von Anfang an waren menschliche Festlegungen, die die KI ausformuliert hat.

Kapitel 3 — Ein Samstagnachmittag (und eine Nacht)

Eine Welt aus Steinen, auf der man herumlaufen konnte. Nicht viel. Aber da war auch bei mir der Groschen gefallen: Ja — das geht!

Es gibt in der Geschichte dieses Projekts ein Wochenende, über das ich am häufigsten gefragt werde. Es ist das Wochenende, an dem aus zwei Dokumenten ein Spiel wurde — und es lief anders ab, als man es sich vorstellt. Vor allem: Der spektakulärste Teil war unsichtbar.

Der neue Mitarbeiter

Am Anfang stand die Übergabe. Die beiden Dokumente aus den Kapiteln 1 und 2 — Justus' redigiertes Konzept, meine eingesprochenen technischen Anforderungen — gingen an das Werkzeug, das in diesem Buch von nun an die zweite Hauptrolle spielt: **Claude Code**.

> **Kurz erklärt: Was ist Claude Code (ein „Coding-Agent")?** ChatGPT kennt man als Chat: Frage rein, Antwort raus. Claude Code ist eine Stufe weiter — ein *Agent*. Das heißt: Er redet nicht nur über Code, er *arbeitet* auf dem Rechner. Er legt Dateien an, schreibt Programme hinein, führt sie aus, liest die Fehlermeldungen, verbessert sich und macht weiter — in Schleife, halbwegs selbstständig, wie ein Mitarbeiter, dem man eine Aufgabe gibt und der zwischendurch Rückfragen stellt. Man schaut ihm dabei buchstäblich zu: Bei uns läuft er als Erweiterung

> direkt in **VS Code**, dem Programmier-Editor. Im selben Fenster, in dem auch der Mensch den Code liest, rattern seine Arbeitsschritte vorbei — wie bei einem sehr schnellen, sehr schweigsamen Kollegen.

Der erste Auftrag an diesen Mitarbeiter war ausdrücklich **nicht** „bau das Spiel". Der erste Auftrag war: **Analysiere die Anforderungen — und erstelle erst einmal einen Plan für eine schrittweise Umsetzung.** Eine Roadmap. Claude las die Dokumente (was bei einer KI Sekunden dauert, nicht Wochen) und lieferte genau das: einen Umsetzungsplan für ein erstes MVP, der die Grundregel unserer Spezifikation im ersten Absatz zurückspiegelte — der Client zeigt an, der Server ist die Wahrheit. Dieser Plan existiert noch; er heißt `IMPLEMENTATION_PLAN.md` und liegt heute als eine Art Gründungsurkunde in der Versionsgeschichte.

> **Kurz erklärt: MVP** MVP steht für *Minimum Viable Product* — das kleinste Ding, das schon funktioniert. Nicht das Spiel mit allem, sondern: eine Welt, ein Spieler, ein Block. Der Witz ist psychologisch: Etwas Kleines, das *läuft*, schlägt etwas Großes, das *fast fertig* ist — jeden Tag der Woche. Man kann es anfassen, zeigen, kritisieren. Und ein Zehnjähriger kann sagen, ob es sich richtig anfühlt.

Dann ließen wir Claude bauen. Schritt für Schritt, Baustein für Baustein, den Plan entlang — und hier kommt die Pointe, die jeder kennen sollte, der dieses Wochenende nacherleben will: **Der erste Prototyp war einer, von dem man nichts sehen konnte.**

Denn was entstand zuerst? Der Server. Die Spiellogik. Die Welt-Erzeugung. Das Fundament, das laut unserer eigenen Spezifikation „die Wahrheit" sein sollte — und Wahrheit hat bekanntlich kein Gesicht. Stundenlang wuchs da etwas, das man nur an durchlaufenden Tests und Logzeilen erkennen konnte. Für einen Vater, der seinem Sohn ein

Spiel versprochen hat, ist das eine harte Geduldsprobe: Man hat sich die Abende um die Ohren geschlagen, und auf dem Bildschirm ist — Text. Ich sage das so deutlich, weil es die realistische Erwartung für jeden ist, der es nachmacht: Wenn man es richtig baut, sieht der Anfang nach nichts aus. Das Sichtbare kommt zuletzt, und dann auf einmal.

Die Nacht

Der Samstagnachmittag wurde ein Samstagabend, der Samstagabend wurde eine Samstagnacht, und ich habe sie mir, es gibt kein eleganteres Wort dafür, **um die Ohren gehauen**. Nicht weil etwas schiefging — sondern weil es *lief*. Wer je in einem Zustand gearbeitet hat, in dem jede halbe Stunde ein weiteres Stück Plan von „offen" auf „fertig" springt, kennt diesen Sog. Es war die erste lange Nacht mit dem neuen Kollegen, und sie hatte einen eigentümlichen Rhythmus: Auftrag geben — zuschauen, wie im VS-Code-Fenster die Dateien entstehen — prüfen — nächster Auftrag. Ich habe in dieser Nacht weniger getippt als in jeder Programmier-Nacht meines Lebens. Und mehr geschafft.

Irgendwann in dieser Nacht war das Fundament da: Server, Weltlogik, die Grundzüge des Clients — als Code. Was fehlte, war das Fenster in diese Welt. Und so kam es zu dem Schritt, den ich im Rückblick am komischsten finde, weil er die Reihenfolge dieses Projekts perfekt beschreibt: **Am Sonntagvormittag — das Spiel existierte im Grunde schon — habe ich erst einmal einen Unity-Account angelegt und den Unity-Editor heruntergeladen.** Die Spiel-Engine, das Schaufenster, das Werkzeug, mit dem man das alles *sehen* würde: nachgereicht, am Ende, wie der Rahmen nach dem Bild.

Der Sonntagvormittag

Dann der Moment. Editor installiert, Projekt geöffnet, Play gedrückt — und am Sonntagvormittag standen Justus und ich zum ersten Mal

gemeinsam vor unserem eigenen Spiel.

Es war nicht viel. Ich will es nicht größer machen, als es war, denn seine Kleinheit ist genau der Punkt: **eine Welt, die aus Steinen bestand, und auf der man herumlaufen konnte.** Keine Raumschiffe, keine Planeten, kein Crafting, nichts von alledem, was in den 2.176 Zeilen stand. Graue Blöcke, ein Spieler, Bewegung.

Aber es war *unsere* Welt aus Steinen. Erdacht am Esstisch, bestellt per Interview, gebaut in einer Nacht — und begehbar.

Für Justus war dieser Moment vermutlich kleiner als für mich; für ihn war ja von Anfang an klar gewesen, dass das klappt („Lass es uns doch ausprobieren!"). Der eigentliche Umsturz fand auf meiner Seite des Sofas statt. Ich stand vor diesen grauen Steinen und spürte, wie eine sehr alte, sehr vernünftige Gewissheit — *richtige Spiele bauen Studios* — leise in sich zusammenfiel. Ich habe es damals genau so gedacht, mit zwei Ausrufezeichen, und so gehört es auch hierher:

Da war auch bei mir der Groschen gefallen. Ja — das geht!!

Aus dem Vater, der die Erwartungen seines Sohnes klein halten wollte („und wenn nur ein Konzept rauskommt…"), war über Nacht ein Überzeugter geworden. Das defensive Projektziel war am Sonntagvormittag Geschichte. Ab jetzt bauten wir ein Spiel.

Die Phase ohne Gedächtnis

Eine technische Beichte gehört noch in dieses Kapitel, denn sie erklärt, warum dieses Buch die ersten Tage nur erzählen, nicht belegen kann: **In dieser Anfangsphase gab es kein Git.** Alles lag schlicht als Dateien auf meiner Festplatte — keine Versionsverwaltung, kein Zurück-Knopf, keine Historie.

> **Kurz erklärt: Git und Commits** *Git* ist das Gedächtnis eines

> Software-Projekts: Es speichert Schnappschüsse aller Dateien — sogenannte *Commits* —, jeder mit Datum und Beschreibung. Man kann jederzeit zu jedem Schnappschuss zurück und nachlesen, was sich wann geändert hat. Ein Projekt ohne Git ist wie ein Roman in einer einzigen, ständig überschriebenen Word-Datei: Es geht gut — bis es das eine Mal nicht gut geht.

So sieht ein echtes Hobbyprojekt am Tag eins eben aus: Man fängt an; die Ordnung zieht ein, wenn klar wird, dass das Ding lebt. Bei uns dauerte das **etwa eine Woche**: erst dann, nach einigen Tagen weiteren Abend-Bauens, bekam das Projekt sein Git. Und damit löst sich auch ein hübsches Datums-Rätsel: Der allererste Commit trägt das Datum **Samstag, 30. Mai 2026** und enthält bereits den kompletten MVP samt aller Spezifikationen. Er ist nicht der Samstag des Bau-Wochenendes, sondern dessen **Schnappschuss eine Woche später** — der Moment, in dem eine Woche Familiengeschichte am Stück ins Projektgedächtnis gehoben wurde. (Dass beides Samstage waren, ist der Rhythmus dieses Projekts in einer Fußnote: Gebaut wird, wenn Wochenende ist.) Ab diesem Commit ist alles belegbar, was dieses Buch erzählt: über tausend Schnappschüsse, jeder datiert. Davor: eine Woche Familienüberlieferung — jetzt immerhin eine sehr genau erzählte.

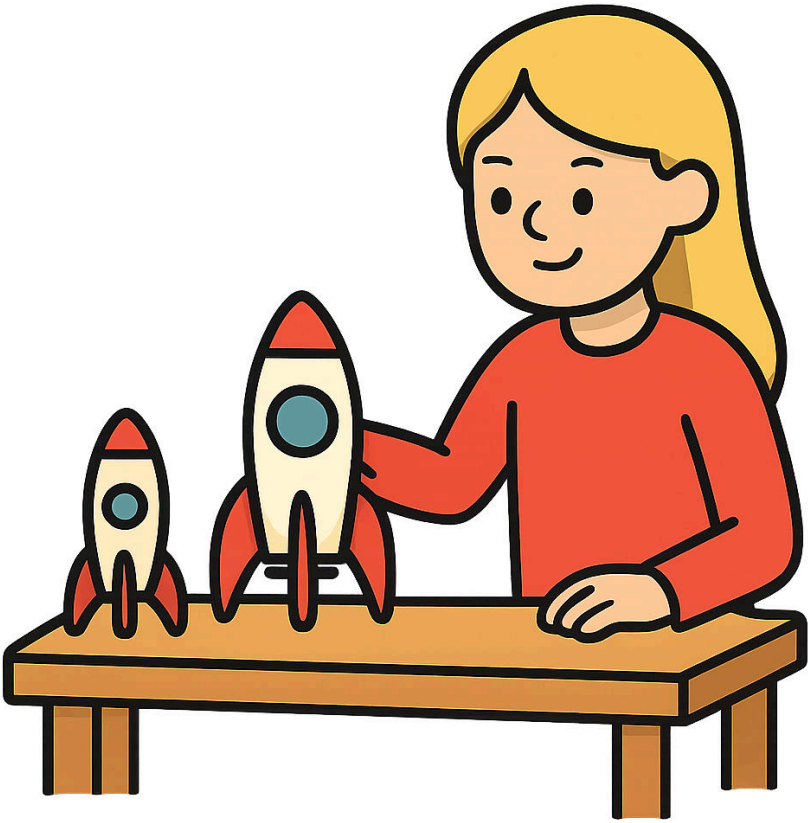
Was dann in den ersten 48 aufgezeichneten Stunden entstand, liest sich wie ein Zeitraffer: Spielmodi, Respawn in der Medbay, das prozedural erwürfelte Universum, ein Missionssystem, Admin-Werkzeuge, ein Web-Zugang, freier Raumflug mit ersten Gegnern — und bereits das optionale KI-Backend und die ersten Werkzeuge, mit denen sich das Projekt Texturen und Sounds selbst generieren sollte. Die komplette DNA des heutigen Spiels, angelegt in einem Wochenende plus ein paar Abenden.

Was dieses Wochenende bewiesen hat — und was nicht

Die verführerische Lesart wäre: „KI baut dir ein Spiel an einem Wochenende." Sie stimmt nicht, und der Rest dieses Buches ist der lange Beweis — auf das Wochenende folgten sechs Wochen mit über tausend Commits, achtzehn Versionen und mehr Lektionen, als in ein Kapitel passen.

Die richtige Lesart ist bescheidener und trotzdem umstürzend: **Die Distanz zwischen „wir haben eine genaue Idee" und „wir stehen gemeinsam vor etwas, das läuft" ist auf ein Wochenende geschrumpft.** Genau die Strecke, auf der Hobbyprojekte reihenweise sterben — monatelang bauen, ohne etwas zeigen zu können —, ist praktisch verschwunden. Ein Kind, das am Abend eine Bitte ausspricht, kann am Sonntagvormittag durch seine eigene Welt laufen. Aus Steinen. Und das reicht völlig, um zwei Menschen für den Rest des Sommers zu überzeugen.

> **Was die KI hier wirklich getan hat** Claude Code hat zuerst analysiert und geplant (die Roadmap — auf ausdrücklichen Auftrag *vor* jedem Bauen), dann das Fundament errichtet: Server, Weltlogik, Client-Gerüst — Tausende Zeilen in einer Nacht, unsichtbar bis zuletzt. Der Mensch hat: die Reihenfolge bestimmt (erst Plan, dann Fundament, zuletzt das Fenster), jede Etappe geprüft, die Nacht durchgemacht — und am Sonntagvormittag den Unity-Editor installiert, damit ein Zehnjähriger sehen konnte, was seine Bitte angerichtet hatte.



Kapitel 4 — Immer erst das kleinste Spiel bauen

„Für jedes Feature zwingen wir die KI, zuerst die kleinstmögliche, funktionierende Version zu bauen." — aus unserem Devblog, als jemand fragte, wie das alles funktioniert

---

Nach dem Wunder-Wochenende aus Kapitel 3 hätte diese Geschichte auf zwei Arten weitergehen können. Die erste kennt jeder, der schon einmal ein Hobbyprojekt hatte: Der Anfangsschwung trägt drei Wochen, dann kommt ein Feature, das sich verhakt, dann ein Urlaub. Und ein Jahr später findet man den Ordner wieder und weiß nicht mehr, wo man war. Die zweite Art ist die, von der dieses Kapitel handelt — und sie hat weniger mit Talent zu tun als mit einem Trick, den wir vom ersten Wochenende einfach beibehalten haben: **Wir haben nie aufgehört, MVPs zu bauen.**

Das Muster war immer dasselbe — so beständig, dass ich es hier als Rezept aufschreiben kann. Ein neues Feature — sagen wir: Kreaturen zähmen, ein Teleporter, ein Energiezaun — begann nie mit der vollen Vision. Es begann mit der Frage: *Was ist die kleinste Version davon, die man spielen kann?* Beim Zähmen: ein Tier, das einem folgt. Kein Fütterungssystem, keine Begleiter-Verwaltung, kein Mitfliegen im Schiff — nur folgen. Das wurde gebaut, an einem Abend, und am selben Abend stand Justus daneben und beurteilte nicht ein Konzept, sondern ein Erlebnis. Erst wenn die kleinste Version sich richtig anfühlte, kam die nächste Schicht.

Warum ist das bei der Arbeit mit einer KI nicht nur nett, sondern entscheidend? Im Devblog haben wir es einmal in einem Satz erklärt,

den ich hier gerne wiederhole, weil er das halbe Buch zusammenfasst: Die KI ist fantastisch darin, Code zu schreiben, **aber sie verliert schnell den architektonischen Überblick.** Ein kleiner, klarer Auftrag („Tier folgt Spieler") wird exzellent erledigt. Ein großer, vager Auftrag („bau ein Begleitersystem") wird auch erledigt — nur wuchern dann an Stellen, die niemand bestellt hat, Nebensysteme, die später niemand versteht. Kleine Schritte halten nicht nur den Menschen bei Verstand. Sie halten die Maschine auf der Straße.

> **Kurz erklärt: Iterieren** *Iterieren* heißt: in Runden arbeiten. Bauen, anschauen, verbessern, wieder bauen — statt alles vorab zu planen und am Ende zu hoffen. Softwareleute predigen das seit Jahrzehnten; mit einer KI im Team wird es zur Naturgewalt, weil eine Runde nicht mehr eine Woche dauert, sondern einen Abend. Wer schneller iteriert, darf öfter irren — und genau das macht am Ende die Qualität.

Der Arbeitsablauf eines Abends

Wie sieht so ein Abend konkret aus? Schon nach wenigen Wochen hatte sich eine feste Choreografie eingespielt, und ich beschreibe sie hier ausführlich, weil sie das ehrlichste Bild davon gibt, was „mit KI entwickeln" im Alltag bedeutet. Es ist nämlich kein Zaubern. Es ist ein Ritual in vier Schritten.

Schritt 1: Analysieren, nicht losbauen. Jede Aufgabe beginnt damit, dass Claude Code den *Ist-Zustand* untersucht: Wie funktioniert der betroffene Teil des Spiels heute? Welche Dateien, welche Regeln, welche Nebenwirkungen? Das Ergebnis ist ein kleiner Analysebericht — noch keine Zeile geänderter Code.

Schritt 2: Den Plan aufschreiben. Aus der Analyse wird ein Plan: Was genau wird geändert, in welcher Reihenfolge, was wird bewusst *nicht* gemacht. Dieser Plan landet schriftlich in unserer Projektdatei — dazu

gleich mehr —, **bevor** die Umsetzung beginnt.

Schritt 3: Rückfragen. Klingt banal, ist aber der Schritt, der die meisten Katastrophen verhindert: Wenn irgendetwas mehrdeutig ist, wird gefragt statt geraten. („Sollen gezähmte Tiere durch das Energie-Tor dürfen?" ist eine Frage, die man besser vor dem Bauen klärt als nach dem Community-Aufschrei.)

Schritt 4: Umsetzen — eine Aufgabe, ein Abend, ein Paket. Erst jetzt fließt Code, und zwar für genau diese eine Aufgabe. Am Ende steht ein sauberes Änderungspaket: Code plus Tests plus aktualisierte Doku, als Ganzes prüfbar.

Wer in der Softwarebranche arbeitet, erkennt das Muster: Das ist schlicht ordentliches Ingenieursvorgehen. Die Pointe ist, *wer* hier so arbeitet. Nicht ein Team mit Prozesshandbuch — sondern ein Vater am Küchentisch und eine KI, der man dieses Verhalten als Arbeitsregel mitgegeben hat. Die Disziplin steckt nicht im Menschen (ich bin abends um zehn genauso schludrig wie jeder andere). **Die Disziplin steckt im Verfahren, und das Verfahren steht in einer Datei.**

Das Gedächtnis des Projekts: eine Datei namens TODO.md

Womit wir bei der unscheinbarsten Hauptfigur dieses Buches wären. In unserem Projekt liegt eine Datei namens `TODO.md` — dem Namen nach eine To-do-Liste, tatsächlich aber das **Logbuch des gesamten Projekts**: Stand heute über 6.700 Zeilen, für jede größere Aufgabe ein datierter Eintrag mit Analyse, Plan, Entscheidungen und Ergebnis. Was erledigt ist, steht drin. Was offen ist, steht drin. Was verworfen wurde — steht auch drin, mitsamt Begründung.

Diese Datei hat zwei Leser, und für beide ist sie unbezahlbar. Der erste bin ich: Wenn zwischen zwei Abenden eine Arbeitswoche liegt,

beantwortet mir das Logbuch in zwei Minuten, wo ich war. Der zweite Leser ist die KI selbst — und das ist der eigentliche Kniff. Eine KI hat kein Gedächtnis zwischen zwei Arbeitssitzungen; jeden Abend „betritt" sie das Projekt neu. Das Logbuch ist ihr Übergabeprotokoll: Sie liest, was gestern entschieden wurde, und muss es nicht erraten. Man kann es dramatischer sagen: **Wir haben das Gedächtnis des Projekts aus den Köpfen in Dateien verlegt** — und erst dadurch wurde eine Maschine ohne Gedächtnis zum verlässlichen Kollegen.

Zum Logbuch gesellte sich mit der Zeit ein zweites Dokument, das ich in Kapitel 2 schon angekündigt habe: `AGENTS.md`, das **Regelbuch für die KI**. Darin steht, was jeder neue menschliche Mitarbeiter am ersten Tag erklärt bekäme: die goldene Architekturregel (der Server hat recht), der Technik-Stack, die Namenskonventionen, die Verbote („keine Unity-Szenen anfassen", „neue Netzwerknachrichten registrieren!"). Es ist Onboarding-Material — nur dass der neue Kollege es jeden Abend wieder liest, in Sekunden, und sich daran hält, solange es klar formuliert ist.

> **Kurz erklärt: Warum die KI Regeln in Dateien braucht** Ein Sprachmodell vergisst zwischen zwei Sitzungen alles — es hat kein Langzeitgedächtnis wie ein Kollege, der reinwächst. Was es aber perfekt kann: zu Beginn jeder Sitzung blitzschnell Dokumente lesen. Also dreht man den Spieß um: Alles, was dauerhaft gelten soll — Regeln, Entscheidungen, Projektstand — wohnt in Dateien, nicht in Köpfen. Nebeneffekt für Menschen: Das Projekt ist dadurch so gut dokumentiert wie kaum ein Hobbyprojekt zuvor.

Achtundzwanzig Commits am Tag

Was dieses Verfahren freisetzt, kann man in unserer Versionsgeschichte nachzählen, und ich nenne die Zahl mit einer Mischung aus Stolz und Kopfschütteln: Im Juni — dem ersten vollen

Monat — sind **867 Commits** entstanden, im Schnitt fast dreißig abgeschlossene, dokumentierte Änderungspakete pro Tag. Zur Erinnerung: abends, nach dem Job, neben Familie. Kein Mensch tippt so viel. Aber darum geht es ja: Getippt hat größtenteils die Maschine. Analysiert, entschieden, geprüft und verantwortet hat ein Mensch — und dieser Teil, nicht das Tippen, war schon immer der Kern des Berufs.

Eine Sache will ich nicht verschweigen, weil sie zur ehrlichen Bilanz gehört: Dieses Tempo verführt. Wenn ein Feature einen Abend kostet, sagt man leichter „ach, und noch eins". Die Disziplin, die früher der Aufwand erzwang, muss man jetzt selbst aufbringen — die Frage „*sollten* wir das bauen?" beantwortet keine KI. Bei uns beantwortet sie meistens ein Zehnjähriger, und wie gut das funktioniert, zeigt Kapitel 13 an dem Tag, an dem er ein komplettes Feature abgelehnt hat.

Vorher aber zu den Entscheidungen, die man nur einmal trifft und dann jahrelang mitträgt: die Architektur. Denn ein paar unserer folgenreichsten Weichenstellungen wurden ausdrücklich *für* den KI-Kollegen gestellt — und sie sind der Grund, warum dieses Projekt nicht unter seinem eigenen Tempo zusammengebrochen ist.

> **Was die KI hier wirklich getan hat** Claude Code hat in dieser Phase pro Aufgabe: den bestehenden Code analysiert, einen schriftlichen Plan ins Logbuch gestellt, Rückfragen formuliert, nach Freigabe implementiert und Tests mitgeliefert. Der Mensch hat: Aufgaben ausgewählt und geschnitten (klein!), Pläne gelesen und korrigiert, Mehrdeutigkeiten entschieden und jedes Paket abgenommen. Faustregel aus der Praxis: Je kleiner der Auftrag, desto brillanter die Maschine.

Kapitel 5 — Entscheidungen, die bleiben

In diesem Unity-Spiel gibt es keine einzige von Hand zusammengeklickte Szene. Alles entsteht aus Code. Das ist keine Marotte — es ist die Bedingung dafür, dass eine KI mitbauen kann.

Es gibt in jedem Bauprojekt zwei Sorten von Entscheidungen. Die eine Sorte trifft man jeden Tag und kann sie morgen ändern: Welche Farbe hat der Knopf, wie schnell läuft das Tier. Die andere Sorte trifft man einmal — und wohnt dann darin. Fundament, tragende Wände, Anschlüsse. In der Softwarewelt heißt diese zweite Sorte *Architektur*, und bei uns haben die wichtigsten dieser Entscheidungen eine Eigenheit: Sie wurden nicht nur für Menschen getroffen, sondern ausdrücklich auch **für den KI-Kollegen**. Dieses Kapitel erzählt die vier folgenreichsten — und das System, mit dem wir sie festgehalten haben.

Erstens: Der Server hat recht (und warum das die KI zähmt)

Die Grundregel kennst du schon aus Kapitel 2: Der Client zeigt an, der Server entscheidet. Jeder Blockabbau, jeder Treffer, jedes Crafting-Rezept wird auf dem Server geprüft und vollzogen; der Client bekommt das Ergebnis mitgeteilt und malt es hin.

In klassischen Teams begründet man diese Architektur mit Schummel-Schutz, und das gilt bei uns auch. Aber es gibt einen zweiten Grund, der in keinem Lehrbuch steht: **Eine klare Wahrheits-Instanz macht KI-Fehler harmloser.** Wenn die Maschine im Client etwas falsch baut — und sie baut gelegentlich etwas falsch —, dann sieht das Spiel

schlimmstenfalls komisch aus. Die Welt selbst, die Spielstände, der Fortschritt: unversehrt, denn die liegen beim Server. Die Regel teilt das Projekt in eine Zone, in der Fehler peinlich sind, und eine, in der sie gefährlich sind. Und sie zwingt jede Änderung an der gefährlichen Zone durch das Nadelöhr aus Kapitel 6: die Tests.

Zweitens: Keine Szenen — alles aus Code

Wer Unity kennt, kennt den Szenen-Editor: Man zieht mit der Maus Objekte in eine 3D-Welt, verschiebt Lichter, klickt Menüs zusammen. So bauen die meisten Unity-Spiele ihre Welt. Bei uns gibt es das praktisch nicht. Unsere Welten, Schiffe, Oberflächen und sogar die Benutzeroberflächen entstehen **vollständig aus Code** — beim Start des Spiels baut sich alles selbst zusammen.

Der Grund ist entwaffnend einfach: **Eine KI hat keine Maus.** Claude Code kann Textdateien lesen und schreiben — brillant, schnell, präzise. Was es nicht kann: im Unity-Editor ein Objekt drei Pixel nach links ziehen. Hätten wir das Spiel klassisch aus handgeklickten Szenen gebaut, wäre der wichtigste Mitarbeiter des Projekts vom halben Spiel ausgesperrt gewesen. So aber ist *alles* Text — und damit alles für die KI erreichbar, versionierbar in Git, vergleichbar, reparierbar.

Das ist die vielleicht am meisten unterschätzte Weichenstellung des Projekts, und ich vermute, sie wird in den nächsten Jahren zum Standard-Ratschlag für KI-gestützte Entwicklung werden: **Verlege dein Projekt so vollständig wie möglich in Textform.** Was nicht Text ist, kann dein bester Mitarbeiter nicht anfassen.

> **Kurz erklärt: Warum Text für Maschinen Gold ist** Programme, Konfigurationen, Weltbeschreibungen, sogar Benutzeroberflächen kann man als Text ausdrücken. Text lässt sich von Git versionieren (wer hat wann was geändert?), von Menschen prüfen und von KIs lesen *und* schreiben. Grafische Editoren sind für

> Menschen bequemer — aber alles, was nur durch Klicken entsteht, ist für automatische Werkzeuge unsichtbar. Faustregel des Projekts: Bequem für die Maus verliert gegen lesbar für alle.

Drittens: Die Welt ist ein Donut

Meine Lieblingsentscheidung, weil sie so schön absurd klingt: Unsere Planeten sind Donuts. Technisch gesprochen: ein *Torus*. Wer auf einem unserer Planeten immer geradeaus läuft, kommt irgendwann von der anderen Seite wieder — die Welt wickelt sich um, ohne Kante, ohne unsichtbare Wand, ohne „hier endet die Karte".

Warum kein echter runder Planet, wo wir doch ein Weltraumspiel sind? Weil Blockwelten und Kugeln sich mathematisch nicht mögen: Auf einer Kugel müssten sich Blöcke zu den Polen hin verjüngen. Das ruiniert genau die schlichte Klötzchen-Logik, die Justus bestellt hat („nicht mehr minecraftig", würde er sagen, und Kapitel 13 zeigt, wie ernst er das meint). Der Torus gibt uns beides: das Gefühl eines Planeten, den man umrunden kann, und eine Blockwelt, die überall gleich funktioniert. Der Spieler merkt vom Donut nichts. Genau daran erkennt man eine gute Architekturentscheidung.

Viertens: Der Einzelspieler ist ein heimlicher Multiplayer

Die vierte Entscheidung klingt paradox: **Unser Spiel hat keinen Einzelspielermodus.** Was wie einer aussieht, ist in Wahrheit der ganz normale Multiplayer — nur dass das Spiel beim Start unsichtbar einen eigenen kleinen Server auf demselben Rechner hochfährt und sich mit ihm verbindet. Ein Spieler, der „alleine" spielt, ist einfach der einzige Gast auf seinem Privatserver.

Der Gewinn ist enorm: Es gibt nur *eine* Spiellogik. Jede Regel, jedes

Feature, jeder Bugfix existiert genau einmal — auf dem Server — und gilt automatisch für Solo, LAN und Online. Die Alternative, zwei parallele Spielwelten zu pflegen („im Einzelspieler funktioniert es, im Multiplayer nicht…"), hat schon Studios mit hundert Leuten in den Wahnsinn getrieben. Für ein Ein-Vater-und-eine-KI-Studio wäre sie tödlich gewesen.

Das Gedächtnis der Gründe: ADRs

So weit die Entscheidungen. Jetzt das System dahinter — denn mindestens so wertvoll wie eine gute Entscheidung ist die Antwort auf die Frage, die garantiert kommt: *„Warum haben wir das noch mal so gemacht?"*

Für diese Antworten gibt es in unserem Projekt eine eigene Dokumentensorte: **ADRs**, elf Stück inzwischen, öffentlich einsehbar im Code-Archiv.

> **Kurz erklärt: ADR (Architecture Decision Record)** Ein ADR ist ein kurzes Dokument — meist eine Seite — das genau eine Architekturentscheidung festhält: Was war die Situation? Welche Optionen gab es? Wofür haben wir uns entschieden, und *warum*? ADRs werden nie umgeschrieben, nur durch neue ersetzt; so bleibt auch die Geschichte der Irrtümer lesbar. Man kann sie sich als die Gerichtsakten eines Projekts vorstellen — inklusive der Urteile, die später kassiert wurden.

Bei uns haben die ADRs noch eine zweite Aufgabe, die man inzwischen erraten kann: Sie sind Futter für den KI-Kollegen. Wenn Claude Code eine Änderung plant, die an eine Grundsatzentscheidung rührt, steht die Begründung von damals in einer Datei, die es mitliest. Die Alternative wäre, dass die KI die Entscheidung jedes Mal neu „erfindet" — mal so, mal anders. Ein Projekt, dessen Gründe aufgeschrieben sind, bleibt auch mit einem vergesslichen Genie im Team konsistent.

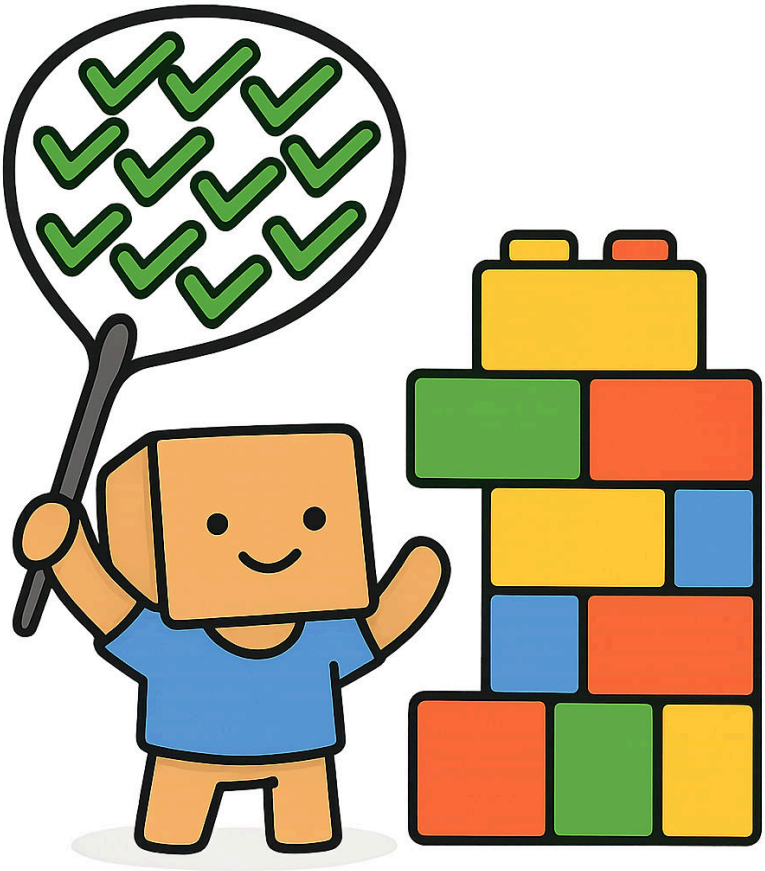
Und weil Ehrlichkeit die halbe Dokumentation ist: Eines unserer elf ADRs ist als *überholt* markiert. Es beschloss einst, für Minispiele und Wiki einen kompletten Webbrowser ins Spiel einzubetten — eine Entscheidung, die wir später krachend revidiert haben. (Die Geschichte samt Justus' Urteil „Die gehen ja jetzt sofort auf!" erzählt Kapitel 16 in einer Fußnote der Release-Ära.) Das kassierte ADR steht trotzdem noch da. Verworfene Entscheidungen sind Lehrmaterial, keine Schande.

Was bleibt

Vier Entscheidungen, ein Dokumentationssystem — und ein gemeinsamer Nenner, der die These dieses Buches in einem Satz ist: **Gute Architektur ist die Kunst, die eigenen Fehler billig zu machen.** Der Server macht KI-Fehler harmlos, der Code-statt-Klicken-Ansatz macht sie reparierbar, der eine Spielmodus macht sie einmalig statt doppelt, und die ADRs sorgen dafür, dass wir denselben Fehler wenigstens nicht zweimal begründen.

Was Architektur nicht kann: Fehler verhindern. Dafür braucht es das Sicherheitsnetz — und das ist bei uns aus elfhundert Fäden geknüpft.

> **Was die KI hier wirklich getan hat** Die vier Grundsatzentscheidungen sind menschlich — teils aus der Spezifikation, teils aus Erfahrung. Claude Code hat dazu: Optionen recherchiert und gegenübergestellt, Konsequenzen durchgespielt („was kostet ein Torus beim Chunk-Streaming?"), die ADR-Dokumente ausformuliert und sich — das ist der Punkt — in tausend Folgeaufgaben an sie **gehalten**, weil sie als Dateien im Projekt liegen. Eine Regel, die nur im Kopf des Chefs existiert, gilt für eine KI nicht.



Kapitel 6 — Vertrauen ist gut, 1.100 Tests sind besser

Über tausend automatisierte Tests für ein Hobby-Spiel — Verschwendung? — Überschrift aus unserem Devblog. Die Antwort steht in diesem Kapitel.

Es gibt eine Frage, die mir Techniker stellen, sobald sie hören, dass der Code dieses Spiels zum überwältigenden Teil von einer KI geschrieben wurde. Die Frage kommt immer, meist im selben Wortlaut, und sie ist vollkommen berechtigt: **„Und woher weißt du, dass der Code stimmt?"**

Die ehrliche Antwort hat zwei Teile. Der erste: Ich weiß es nicht — nicht Zeile für Zeile. Niemand liest abertausende Zeilen pro Woche mit echter Aufmerksamkeit; wer das behauptet, überschätzt sich. Der zweite Teil ist der Inhalt dieses Kapitels: **Ich muss es auch nicht wissen. Ich lasse es beweisen.** Jeden Abend, bei jeder Änderung, automatisch.

Was ein Test ist — und was elfhundert Tests sind

> **Kurz erklärt: Automatisierte Tests** Ein automatisierter Test ist ein Mini-Programm, das ein anderes Programm überprüft. Beispiel aus unserem Spiel, sinngemäß: *„Nimm einen Spieler mit leerem Inventar. Lass ihn einen Eisenblock abbauen. Prüfe: Ist danach genau ein Eisenerz im Inventar?"* Der Test läuft in Millisekunden, ohne Bildschirm, ohne Mensch — und er läuft *jedes Mal wieder*, bei jeder künftigen Änderung am Spiel. Fällt er durch, weiß man auf den Tag genau, welche Änderung etwas kaputt gemacht hat.

Ein einzelner Test ist unspektakulär. Interessant wird die Masse: In unserem Projekt existieren Stand heute rund **990 Tests für den Server und etwa 120 für den Client** — über elfhundert Prüfungen, die Rezepte, Sauerstoff, Hunger, Kreaturenverhalten, Welt-Passwörter, Netzwerknachrichten und hundert andere Regeln festnageln. Zusammen sind sie das, was ich im Devblog einmal den „zweiten Entwickler" genannt habe: der Kollege, der bei jeder Änderung dazwischenruft „Moment — das bricht doch das Crafting!". Ein Solo-Projekt hat diesen Kollegen normalerweise nicht. Unseres hat ihn elfhundertfach.

Und jetzt die Pointe, die dieses Kapitel mit Kapitel 2 verbindet: Diese Testkultur war **Teil der Bestellung**. „Testabdeckung von Anfang an" stand wörtlich in den eingesprochenen technischen Anforderungen — nicht als späte Reue nach dem ersten kaputten Spielstand, sondern als Diktat vom ersten Tag. Ich kann diese Entscheidung im Rückblick gar nicht überschätzen, und ich gebe zu, warum sie mir leichtfiel: Ich wusste, *wer* die Tests schreiben würde. Nämlich nicht ich.

Der Deal mit der Maschine

Denn hier dreht sich die Skeptiker-Frage auf wunderbare Weise um. Der Einwand „KI schreibt Code — wer prüft ihn?" unterschlägt, dass dieselbe KI mit demselben Fleiß auch die Prüfungen schreibt. Unser Standard-Arbeitspaket aus Kapitel 4 heißt nicht „Feature", sondern **„Feature plus Tests"**: Wer die Zähm-Logik baut, liefert die Tests mit, die sie festnageln. Für einen menschlichen Entwickler ist Testschreiben der ungeliebte Pflichtteil, der unter Zeitdruck zuerst stirbt. Eine KI kennt weder Zeitdruck noch Unlust. Sie schreibt den zwanzigsten Grenzfall-Test mit derselben Sorgfalt wie den ersten.

Natürlich lauert hier eine Falle, und sie ist real: Wer Code und Prüfung vom selben Autor bauen lässt, riskiert Gefälligkeitstests — Prüfungen, die genau das bestätigen, was der Code eben tut, inklusive seiner

Fehler. Dagegen helfen drei Dinge. Erstens die Spezifikation: Der Test prüft gegen das, was *bestellt* wurde, nicht gegen das, was gebaut wurde. Zweitens der Mensch, der beim Abnehmen genau die Testliste liest (die ist kurz und lesbar, im Gegensatz zum Code). Drittens die Realität: Justus' Zettel aus Kapitel 12 ist der unbestechlichste Abnahmetest der Welt.

Das Tor, durch das jeder Commit muss

Tests, die niemand ausführt, sind Dekoration. Deshalb hängt an unserem Projekt ein Wächter, der nie schläft: die **CI**.

> **Kurz erklärt: CI (Continuous Integration)** CI heißt: Bei jeder eingereichten Änderung baut ein Automat das komplette Projekt frisch zusammen und lässt alle Tests laufen — auf einem neutralen Server, nicht auf dem Rechner des Entwicklers („bei mir läuft's" zählt nicht). Erst wenn alles grün ist, darf die Änderung in den Hauptzweig. Man kann es sich als Werkstor mit Qualitätskontrolle vorstellen: Es kommt nur rein, was die Prüfung besteht — auch nachts um elf, auch wenn der Chef müde ist. *Gerade* wenn der Chef müde ist.

Unser Tor ist zweistufig, aus einem sehr praktischen Grund: Tests, die eine Viertelstunde dauern, führt irgendwann niemand mehr gerne aus. Der schnelle Check läuft deshalb in gut **drei Minuten** mit den flotten Tests — kurz genug, dass man auf das Ergebnis wartet, statt es zu ignorieren. Die vollständige, langsame Suite läuft danach automatisch hinterher. Geschwindigkeit ist bei Qualitätswerkzeugen kein Luxus, sondern die Bedingung dafür, dass sie benutzt werden.

Dazu kommen zwei Verschärfungen, auf die ich nicht mehr verzichten würde. Erstens: **Warnungen sind Fehler.** Compiler geben Warnungen aus — „das hier ist verdächtig, läuft aber". Menschliche Teams sammeln davon über die Jahre Friedhöfe, in denen die eine wichtige

Warnung zwischen dreihundert egalen untergeht. Bei uns bricht jede einzelne Warnung den Build ab. Klingt pedantisch; heißt aber: Der Zustand „null Warnungen" ist immer der Normalzustand, und alles Neue fällt sofort auf. Zweitens läuft auf dem öffentlichen Code-Archiv zusätzlich **CodeQL**, eine automatische Sicherheitsanalyse, die nach bekannten Schwachstellen-Mustern sucht. Auch sie hat uns schon konkrete Funde beschert, die noch am selben Wochenende behoben wurden.

Unity testen — der Sondertrick

Eine technische Feinheit verdient ihren Platz, weil sie wieder das Muster aus Kapitel 5 zeigt (Architektur im Dienst der Prüfbarkeit). Spiel-Clients gelten als notorisch schlecht testbar, weil ihre Logik in der laufenden Engine feststeckt — man müsste zum Prüfen „das Spiel hochfahren". Unser Trick: Die Kernlogik des Clients lebt in einer eigenen Bibliothek, die **ohne Unity** läuft, und lässt sich damit prüfen wie normaler Code — schnell, headless, in der CI. Nur der schmale Rest, der wirklich Engine braucht, läuft in Unitys eigenen Testmodi. Auch das war kein Zufall, sondern eine bewusste Trennung — getroffen, damit der zweite Entwickler (die Tests) und der dritte (die KI) beide möglichst viel vom Projekt erreichen.

Die ehrliche Rechnung

Kostet das alles etwas? Ja: Tests schreiben und pflegen verschlingt geschätzt ein Fünftel der Entwicklungszeit — auch wenn die Maschine tippt, sind Prüfen, Nachschärfen und Warten auf grüne Häkchen echte Abende. Dafür bekommen wir drei Dinge, jedes einzeln den Preis wert.

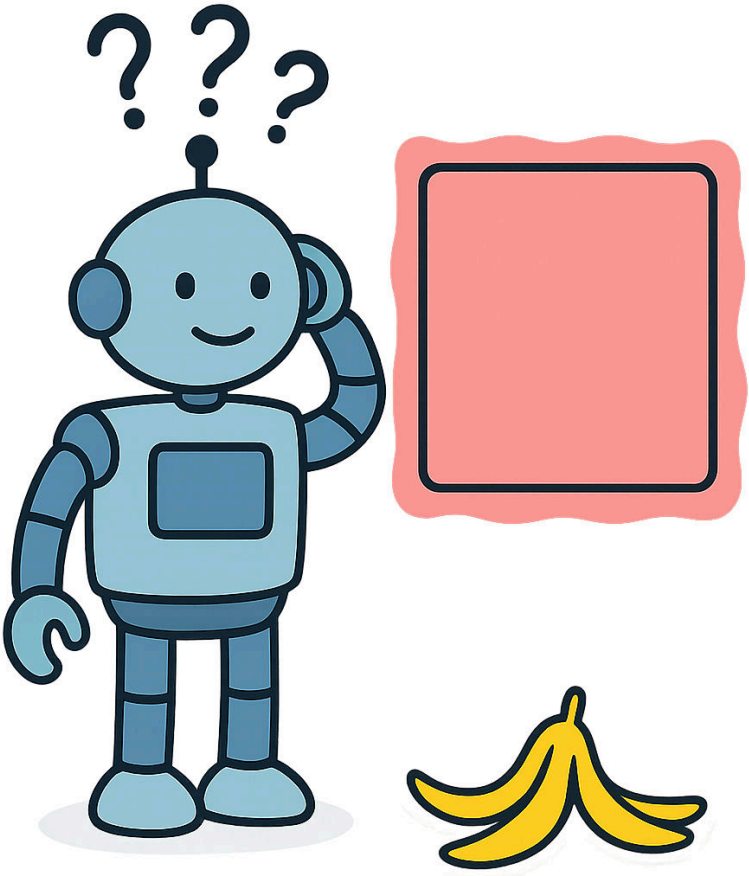
Angstfreies Ändern. Der Grafik-Overhaul, der Browser-Rauswurf, das Umbauen der Bewegungslogik — große Umbauten, bei denen elfhundert Wächter aufpassen, dass nichts Bestehendes bricht. Ohne sie hätte ich mich an die Hälfte davon nicht getraut, und „sich nicht

trauen" ist der Tod jedes lang laufenden Projekts.

Gefixt bleibt gefixt. Jeder Zettel-Bug aus Kapitel 12 bekommt beim Beheben einen Test, der genau diesen Fehler nachstellt. Der Fehler kann wiederkommen — aber nicht *unbemerkt*. Bei einem Projekt, das abends nach dem Job entsteht, ist das die wertvollste Garantie überhaupt: dass das Wochenend-Feature nicht heimlich das Dienstag-Feature zerlegt hat.

Und der eigentliche Punkt dieses Kapitels: Die Tests sind die Antwort auf die Skeptiker-Frage. Vertrauen in KI-Code entsteht nicht durch Lesen und nicht durch Hoffen. Es entsteht dadurch, dass man das Vertrauen durch etwas Besseres ersetzt: **durch Beweise, die bei jeder Änderung neu erbracht werden.** Das ist übrigens bei menschlichem Code kein bisschen anders. Wir haben es dort nur nie so ernst genommen, weil Menschen langsamer tippen.

> **Was die KI hier wirklich getan hat** Geschrieben hat die Tests fast ausnahmslos Claude Code — inklusive der Grenzfälle, zu denen menschliche Entwickler selten Lust haben. Der Mensch hat: die Testpflicht zur Regel gemacht (schon in der Spezifikation!), Testlisten beim Abnehmen gelesen, Gefälligkeitstests zurückgewiesen und die CI-Gates konfiguriert. Merksatz des Kapitels: Die KI liefert die Beweise — aber der Mensch legt fest, was bewiesen werden muss.



Kapitel 7 — Die Fallen. (Was die KI nicht wusste)

Im Editor sah alles perfekt aus. Im fertigen Spiel: rosa Flächen.

Bis hierhin könnte der Eindruck entstehen, mit guten Dokumenten, kleinen Schritten und elfhundert Tests laufe so ein KI-Projekt wie ein Uhrwerk. Zeit für das Gegenprogramm. Dieses Kapitel ist eine Sammlung unserer besten Pannen — der Abende, an denen alles richtig aussah und trotzdem nichts ging. Ich erzähle sie aus zwei Gründen gern: Erstens sind sie unterhaltsam (hinterher, immer nur hinterher). Zweitens zeigen sie präziser als jedes Erfolgskapitel, wo die Grenze zwischen Maschine und Mensch tatsächlich verläuft.

Denn das Muster all dieser Geschichten ist dasselbe, und ich verrate es vorweg: Die KI scheitert selten an der Logik. Sie scheitert an den **Eigenheiten** — den ungeschriebenen Regeln von Werkzeugen, den Fußnoten der Plattformen, dem Kleingedruckten, das in keinem Lehrbuch steht, sondern nur in den Narben derer, die es erlebt haben.

Der Tag, an dem die Shader fehlten

Das Paradebeispiel. Wir hatten dem Spiel über zwanzig eigene *Shader* spendiert — kleine Spezialprogramme für die Grafikkarte, die Wasser glitzern, Atmosphären leuchten und Hologramme flimmern lassen. Im Unity-Editor: wunderschön. Dann bauten wir das Spiel zum Verteilen zusammen, starteten es — und blickten auf rosa Flächen. Rosa ist Unitys Farbe für „hier sollte ein Shader sein, ich habe aber keinen".

Die Ursache ist eine klassische Unity-Fußnote: Beim Bauen wirft die

Engine alles raus, was scheinbar niemand benutzt — auch Shader, die das Spiel erst zur Laufzeit per Namen lädt, so wie wir es taten. Der Editor kannte alle Shader und war gnädig. Der Build war gnadenlos. Die Lösung ist ein Handgriff, *wenn man ihn kennt*: Solche Shader müssen in eine spezielle Immer-Einpacken-Liste in den Projekteinstellungen.

Hier steht der eigentliche Lernsatz des Kapitels: Claude Code hatte den Shader-Code fehlerfrei geschrieben. Was die KI nicht wusste — nicht wissen *konnte*, ohne es zu erleben —, war diese Plattform-Eigenheit im Zusammenspiel unseres konkreten Projekts. **Logik kann man wissen. Eigenheiten muss man erleben.** Und aus demselben Holz waren fast alle unsere schlimmen Abende geschnitzt:

Der Layer, der -1 war. Ein Physik-Layer, per Namen im Code aufgelöst: funktioniert im Editor, existiert im automatischen Build schlicht nicht — die Abfrage liefert stumm den Wert -1, und alles darauf Gebaute verhält sich leise falsch. Seitdem: Layer über feste Nummern, nie über Namen.

Die stumme Netzwerknachricht. Unser Netzwerkprotokoll verlangt, dass jede neue Nachrichtenart registriert wird. Vergisst man das, stürzt nichts ab und nichts landet im Fehlerprotokoll — das Feature „geht einfach nicht". Wir haben das mehr als einmal debuggt, bis der Reflex saß: Neue Nachricht? Erste Frage: registriert?

Der Geisterblock. Ändert der Server einen Block, muss er es allen Spielern aktiv mitteilen. Vergisst er den Rundruf, läuft der Spieler gegen unsichtbare Wände oder baut auf Boden, den es serverseitig nicht mehr gibt. Zwei Wahrheiten, ein vergessener Anruf dazwischen.

Die Tür, die zu niedrig war, obwohl sie hoch genug war. Unsere Spielfigur ist knapp 1,9 Blöcke hoch — aber ihre Treppensteig-Logik tastet beim Gehen zusätzlich 0,6 Blöcke nach oben. Eine zwei Blöcke hohe Tür ist also rechnerisch ausreichend und praktisch zu niedrig: Man bleibt hängen. Hausregel seither, in Stein gemeißelt: Durchgänge sind

drei Blöcke hoch. Immer.

Das Glas, das zu gut war. Unser erstes klares Glas war technisch perfekt — und sah aus wie ein Loch in der Wand. In einer Blockwelt liest sich milchiges, frostiges Glas viel besser als Material: Man *sieht*, dass da etwas ist. Manchmal ist die realistische Lösung die schlechtere; entschieden hat das kein Benchmark, sondern ein Blick.

Und Windows. Für Nicht-Techniker nur so viel: Es gibt auf Windows eine historische Grenze für die Länge von Dateipfaden, und Unity-Projekte erzeugen Ordner-in-Ordner-Strukturen, die genau da hineinlaufen. Das Aufräumen unserer Arbeitskopien scheiterte daran so hartnäckig, dass die rettende Lösung ein Trick aus der Systemadministrator-Folklore wurde (man spiegelt ein leeres Verzeichnis über das störrische — frag nicht). Auch das wusste keine KI vorher. Jetzt weiß sie es. Womit wir beim Wichtigsten wären.

Das Gotcha-Gedächtnis

Denn die eigentliche Geschichte dieses Kapitels ist nicht, *dass* wir in Fallen gelaufen sind — das tut jedes Projekt. Die Geschichte ist, was danach geschah. Nach jeder dieser Pannen passierte bei uns nämlich dasselbe, und es dauerte fünf Minuten: **Die Lektion wurde aufgeschrieben — dorthin, wo die KI sie wieder liest.**

Im Projektgedächtnis (den Regel- und Notizdateien aus Kapitel 4) sammelte sich so über die Wochen eine sehr spezielle Textsorte an, die Entwickler liebevoll *Gotchas* nennen — Merksätze der Marke: „Shader, die per Namen geladen werden, müssen in die Immer-Einpacken-Liste." „Neue Netzwerknachrichten registrieren, sonst stiller Fehlschlag." „Türen: drei Blöcke. Immer." Jeder dieser Sätze hat uns einen Abend gekostet. Und jeder verhindert seither Wiederholungen — nicht, weil ein Mensch sich erinnert, sondern weil der Kollege, der jeden Abend alles neu liest, sie **mitliest, bevor er baut**.

> **Kurz erklärt: Warum das mehr ist als eine Checkliste** Menschliche Teams haben so etwas auch — es heißt dann „Erfahrung" und wohnt in den Köpfen der Dienstältesten. Das Problem: Köpfe kündigen, vergessen und sind im Urlaub. Bei uns wohnt die Erfahrung in Dateien, und der fleißigste Mitarbeiter liest sie vor jeder Aufgabe vollständig. Der Effekt ist verblüffend: Das Projekt macht neue Fehler statt alte. Mehr kann man von einer Lernkurve nicht verlangen.

Ich behaupte inzwischen: Diese Schleife — Falle, Analyse, Merksatz, nie wieder — ist der unterschätzteste Teil des KI-Arbeitens überhaupt. Über die spektakulären Fähigkeiten der Modelle wird viel geschrieben. Aber der Alltagsunterschied entsteht woanders: darin, ob ein Team (Mensch wie Maschine) seine Lektionen **konserviert**. Eine KI ohne Gotcha-Gedächtnis tappt charmant und unermüdlich immer wieder in dieselben Löcher. Eine KI *mit* — wird von Woche zu Woche spürbar zum Senior.

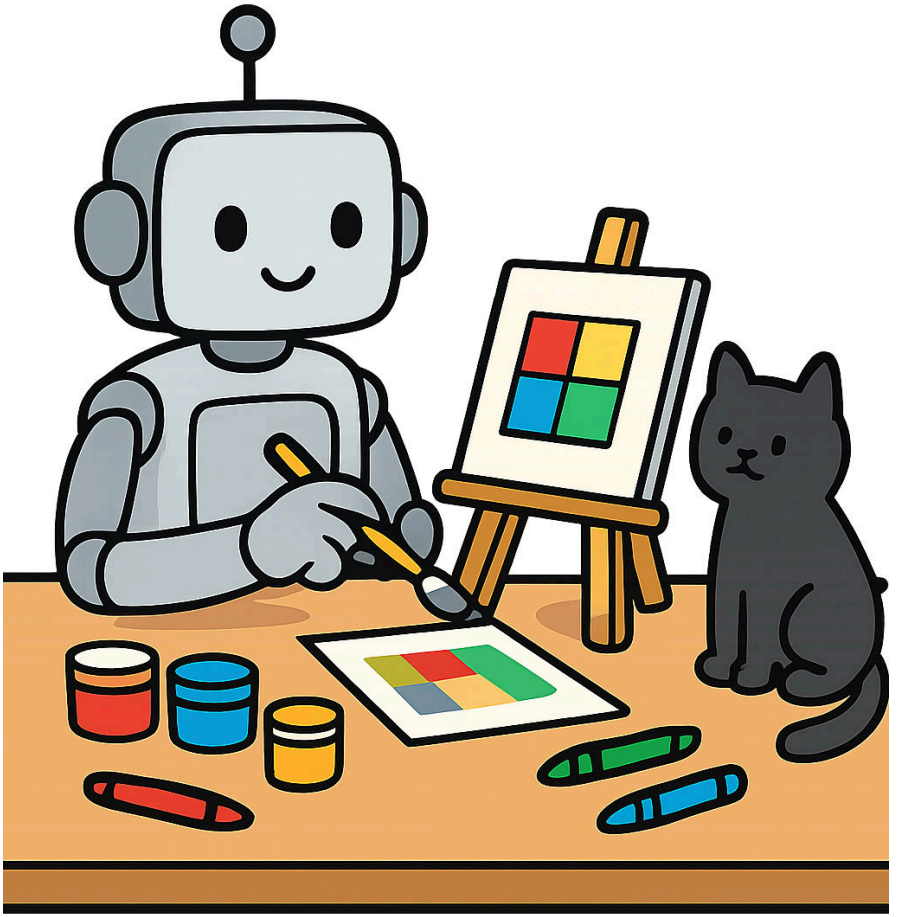
Der Editor lügt — und was daraus folgte

Eine letzte Lektion aus dieser Sammlung hat es zu einer eigenen Betriebsregel gebracht, und sie schlägt den Bogen zurück zu Kapitel 6. Mehrere unserer Fallen hatten denselben Auslöser: **Was im Unity-Editor funktioniert, hat im fertigen Build noch gar nichts bewiesen.** Der Editor hat alle Assets, alle Namen, alle Shader griffbereit; der Build ist eine andere, strengere Welt.

Die Konsequenz: Bei uns gehört zu jeder nennenswerten Änderung am Client ein *echter lokaler Build-Test* — das Spiel wird wirklich zusammengebaut und wirklich gestartet, nicht nur im Editor angespielt. Das dauert länger, jedes Mal. Es hat uns trotzdem schon eine Reihe peinlicher Veröffentlichungen erspart. Und es fügt der Werkzeugkette dieses Buches das letzte Glied hinzu: Spezifikation (Kap. 2), kleine Schritte (Kap. 4), Architektur (Kap. 5), Tests und CI (Kap. 6) — und

ganz am Ende ein Mensch, der das echte Spiel startet und mit eigenen Augen hinsieht. Manche Wahrheiten wohnen nun mal nur dort.

> **Was die KI hier wirklich getan hat — und was nicht** Nicht getan: die Fallen vorhergesehen. Plattform-Eigenheiten, Build-Besonderheiten und „das fühlt sich falsch an"-Wahrheiten standen in keinem Trainingsdatensatz unseres Projekts. Getan: bei jeder Panne die Ursache mit stoischer Geduld eingekreist (eine KI wird beim Debuggen nie mürrisch — unterschätzter Vorteil), die Reparatur gebaut, einen Test dagegen geschrieben und den Merksatz ins Projektgedächtnis übernommen. Die Falle findet der Mensch. Die Wiederholung verhindert die Maschine.



Kapitel 8 — Ein Spiel ohne Grafiker, am Küchentisch

„Zu gruselig." — „Klingt wie ein kaputter Kühlschrank." — Justus, Qualitätskontrolle, am Abendbrottisch (Original-Urteile)

In den Credits dieses Spiels steht kein Grafikstudio. Trotzdem hat jeder Block eine Textur, jede Kreatur ein Gesicht, jedes Werkzeug ein Icon — Hunderte von Bildern, alle im selben Stil. Wie das ohne einen einzigen Grafiker geht, ist die häufigste Frage, die uns gestellt wird, gleich nach der nach dem Code. Und die Antwort beginnt mit einer Weichenstellung, die inzwischen ein vertrautes Muster ist: Wir haben nicht Bilder gemalt. **Wir haben uns eine Werkstatt gebaut, die Bilder herstellt.**

Die Werkstatt: Python-Skripte als Fließband

Der Ablauf, wenn unser Spiel eine neue Block-Textur braucht — sagen wir, für den Algentank aus einem der Juli-Updates —, sieht so aus: Ich starte ein kleines Kommandozeilen-Programm, übergebe ihm den Namen des gewünschten Blocks, und ein paar Sekunden später liegt eine fertige Kachel im richtigen Format am richtigen Ort. Das Programm hat im Hintergrund die Bild-KI von OpenAI aufgerufen — mit einem Prompt, der nicht bei jedem Aufruf neu erfunden wird, sondern in dem unsere Stilregeln fest eingebaut sind: Voxel-Ästhetik, Pixel-Look, Materialcharakter, kein Kitsch.

Diese kleinen Programme — inzwischen ein gutes Dutzend, für Block-Texturen, Item-Icons, Kreaturen, Kleinstlebewesen, App-Icons — hat natürlich niemand von Hand geschrieben. **Claude hat die Werkzeuge**

gebaut, mit denen die anderen KIs bedient werden. Ich mag diesen Satz, weil er die Arbeitsteilung des ganzen Projekts in einem Bild einfängt: eine KI als Werkzeugmacher, eine als ausführende Künstlerin, ein Mensch als Auftraggeber und Richter.

> **Kurz erklärt: API — die Steckdose für Programme** Dienste wie die Bild-KI von OpenAI kann man als Mensch im Browser benutzen — oder als Programm über eine *API* (application programming interface): eine Art Steckdose, an der andere Software andocken darf. Statt „Mensch tippt in Webseite" heißt es dann „Skript schickt Auftrag, bekommt Bild zurück". Erst die API macht aus einem Spielzeug ein Fließband: hundert Texturen bestellen, während man Abendessen kocht.

Zwei Konstruktionsdetails der Werkstatt verraten viel darüber, was man im Umgang mit bezahlten KI-Diensten lernt.

Erstens: die Kostenbremse. Jeder Aufruf einer Bild-KI kostet Geld — Centbeträge, aber Centbeträge mit Multiplikator, wenn ein Skript in einer Schleife Amok läuft. Unsere Werkzeuge zeigen deshalb **vor jedem bezahlten Aufruf den Preis** an und arbeiten standardmäßig ein Motiv pro Lauf ab, nicht „alles neu". Klingt kleinlich; ist aber der Grund, warum am Ende des Monats auf der API-Rechnung ein Betrag steht, über den man lächelt (die genaue Zahl kommt in Kapitel 22, sie ist fast schon komisch).

Zweitens: die Nicht-Determinismus-Falle. Bild-KIs würfeln. Derselbe Prompt liefert morgen ein *anderes* Bild — ähnlich, aber nicht gleich. Das heißt: Unsere committeten Dateien sind die einzige Wahrheit. Man kann eine verlorene Textur nicht „einfach neu generieren", man bekäme eine fremde Schwester. Seit uns das klar wurde, gilt: Was ins Spiel kommt, wird versioniert und aufgehoben wie ein Original — die Werkstatt kann Neues liefern, aber nichts Altes wiederholen.

Die eigentliche Arbeit heißt Nein sagen

Von außen klingt das alles nach Knopfdruck, und ich will der Legende hier einmal gründlich widersprechen. Die erste generierte Textur passt fast nie. Nicht, weil sie schlecht wäre — einzeln betrachtet ist sie oft verblüffend hübsch. Sondern weil ein Spiel keine Sammlung hübscher Einzelbilder ist. **Eine Textur muss zu den hundert anderen passen**: gleiche gefühlte Auflösung, gleiche Farbwelt, gleiche „Körnung". Die schwerste Disziplin der KI-Grafik ist nicht das Erzeugen, sondern die Konsistenz — und die entsteht bei uns durch drei Filter.

Filter eins sind die Prompts selbst, die eher Stilvorschriften sind als Wünsche. Filter zwei ist der Kontext-Check: Jede Kachel wird nicht am Einzelbild beurteilt, sondern *im Spiel*, eingebaut zwischen ihren Nachbarn — eine Textur, die solo großartig aussieht und in der Wand flimmert, fliegt raus. Und Filter drei ist der strengste und sitzt am Abendbrottisch: Neue Bilder werden auf dem Laptop herumgezeigt, neue Kreaturen der Familie vorgestellt. Was Verena und Justus nicht überzeugt, kommt nicht ins Spiel. Das klingt unwissenschaftlich und ist der verlässlichste Qualitätsfilter, den wir haben — Kinder besitzen keinerlei Hemmung, einem mitzuteilen, dass etwas doof aussieht.

> **Kurz erklärt: Textur-Atlas** Die Blockbilder landen nicht als hundert Einzeldateien im Spiel, sondern in einem *Atlas*: einer einzigen großen Textur, in der jede Block-Art ihre feste Kachel hat — wie ein Setzkasten. Die Grafikkarte liebt das (ein Bild statt hundert), aber der Setzkasten hat eine feste Zahl von Fächern. Als unsere Blockvielfalt das Raster sprengte, wurden neue Blöcke kommentarlos grau — bis jemand das Raster vergrößerte. Auch so ein Merksatz fürs Gotcha-Gedächtnis aus Kapitel 7.

Was das verändert — die ehrliche Einordnung

Zum Schluss dieses Kapitels die Einordnung, die mir wichtig ist, weil

das Thema KI-Kunst zu Recht Debatten auslöst. Bei uns hat die Bild-KI kein Grafikstudio ersetzt und keinem Grafiker den Auftrag weggenommen — **es gab nie ein Budget, das stattdessen ein Mensch bekommen hätte.** Die Alternative zu KI-Texturen war nicht „ein Grafiker macht es", sondern „es existiert nicht": ein graues Spiel aus Platzhalter-Klötzen, das nie den Weg aus dem Bastelordner gefunden hätte. Generative KI hat hier keine Arbeit verdrängt, sondern eine Gattung ermöglicht: das Familienprojekt mit den Produktionswerten eines Studios.

Und trotzdem — auch das gehört zur Ehrlichkeit — ist „100 % KI-generiert", wie es augenzwinkernd auf unserer Spieleseite steht, nur die halbe Wahrheit. Die Bilder erzeugt die Maschine. Aber welcher Stil, welche Auswahl, welches Nein: Das ist Geschmack, und Geschmack war den ganzen Weg über Handarbeit. Im Devblog haben wir es einmal so formuliert, und besser bekomme ich es nicht hin: **Die KI liefert Material. Das Spiel machen wir.**

> **Was die KI hier wirklich getan hat** Claude Code hat die Generator-Werkzeuge geschrieben (inklusive Kostenbremse und Atlas-Einbau); die OpenAI-Bild-KI hat auf deren Zuruf Hunderte Texturen, Icons und Kreaturenbilder erzeugt. Der Mensch (und die Familie) haben: die Stilregeln gesetzt, jede Kachel im Kontext beurteilt, verworfen und neu bestellt — und entschieden, dass Originale versioniert werden, weil die Maschine sich selbst nicht wiederholen kann.

Kapitel 9 — Wie klingt ein Energiezaun?

Ein Zaun, der lauter brummt als der Wasserfall daneben, wäre nach fünf Minuten unerträglich.

Grafik sieht man sofort. Sound bemerkt man erst, wenn er fehlt — oder wenn er nervt. Dieses Kapitel handelt von der zweiten Hälfte der Sinneswelt unseres Spiels: den Geräuschen und der Musik. Und es handelt von der einen Stelle in unserem KI-Maschinenpark, an der die Automatisierung endet und echte Handarbeit beginnt — ausgerechnet bei der Musik.

Text wird Geräusch: ElevenLabs

Fangen wir bei den Geräuschen an, denn dort funktioniert das Fließband aus Kapitel 8 fast unverändert weiter. Unser Werkzeugkasten enthält ein Skript, das eine Textbeschreibung entgegennimmt — „kurzes, weiches Aufheben eines Gegenstands", „dumpfes Grollen eines fernen Triebwerks" — und daraus über die API von **ElevenLabs** eine Audiodatei erzeugt. Text rein, Klang raus.

Wer das zum ersten Mal benutzt, erlebt einen kleinen Realitätsbruch: Man *beschreibt* ein Geräusch, das es nicht gibt, und Sekunden später *hört* man es. Auf diese Weise sind die Klänge des gesamten Spiels entstanden — Schritte auf verschiedenen Materialien, das Blubbern des Algentanks, Laserschüsse, das Craften an der Werkbank, die Stimmen-Chiffren der Kreaturen. Gleich am zweiten Tag der aufgezeichneten Projektgeschichte findet sich dafür ein hübscher Beleg: ein Commit, der einen kompletten Soundeffekt-Katalog als Batch-Auftrag anlegt — die

Vertonung des Spiels lief buchstäblich als Stapelverarbeitung über Nacht an.

Aber — und dieses Aber ist der Kern des Kapitels — **das Erzeugen war nie die Arbeit. Die Arbeit ist das Abmischen mit der Welt.** Das Lehrstück dazu ist unser Energiezaun. Als das Feature kam (Kapitel 13 erzählt seine Produktgeschichte), brauchte der Zaun ein Summen: Man soll *hören*, dass da Energie fließt. Der erste generierte Summton war großartig — präsent, elektrisch, bedrohlich. Und genau deshalb unbrauchbar. Ein Zaun steht stundenlang neben der Basis; sein Geräusch läuft in Dauerschleife. Was beim ersten Hören Atmosphäre ist, ist in der zwanzigsten Minute Folter. Der Ton ging durch mehrere Runden, wurde weicher, unauffälliger, leiser — bis er das **leiseste Dauergeräusch im ganzen Spiel** war. Ein Zaun, der lauter brummt als der Wasserfall daneben, wäre unerträglich. Diese Entscheidung — nicht „klingt es gut?", sondern „erträgt man es tausendmal?" — nimmt einem keine KI ab. Sie steht in keinem Prompt. Man muss danebenstehen und es aushalten.

Und natürlich lief auch hier der strengste Filter am Abendbrottisch: Neue Sounds werden der Familie vorgespielt, und die Urteile sind aktenkundig. **„Zu gruselig." — „Klingt wie ein kaputter Kühlschrank."** Beides echte Justus-Bewertungen, beides Todesurteile, beides vollstreckt.

Die Lücke im Fließband: Musik von Suno

Bei der Musik ändert sich die Geschichte, und zwar an einer technischen Kleinigkeit mit großen Folgen: **Suno, das KI-Werkzeug für ganze Musikstücke, hatte keine API.** Keine Steckdose für Programme (Kapitel 8), kein Skript, kein Fließband. Musik ließ sich nur so bestellen wie von einem Menschen: Webseite öffnen, Beschreibung eintippen, anhören, verwerfen, nachbestellen.

Unsere Lösung war eine Arbeitsteilung, die ich immer noch charmant finde: **Claude schreibt die Musikprompts, der Mensch trägt sie zu Suno.** Ein Musikprompt ist dabei mehr als „Weltraummusik, bitte" — er beschreibt Genre, Tempo, Instrumentierung, Stimmung und den Verwendungsort im Spiel („ruhiges, schwebendes Ambient-Stück für den Planetentag, keine Percussion, lange Flächen…"). Diese Beschreibungssprache beherrscht ein Sprachmodell hervorragend — es kennt die Vokabeln von Musik, wie es die von Grafik kennt. Also entstanden die Prompts im selben Chat-Fenster wie der Rest des Spiels, wurden per Hand in Suno eingefügt, und zurück kamen Stücke, aus denen wir auswählten.

> **Kurz erklärt: Warum „keine API" alles ändert** Mit API ist ein KI-Dienst ein Fließband: Skripte bestellen, vergleichen, sortieren automatisch nach. Ohne API ist er ein Ladengeschäft: Jeder Auftrag ist ein Gang zur Theke. Für zehn Musikstücke ist das verschmerzbar; für vierhundert Texturen wäre es das Ende des Projekts gewesen. Wer KI-gestützt produzieren will, sollte Werkzeuge deshalb zuerst nach ihrer Steckdose beurteilen und dann nach ihrer Brillanz.

Rückblickend war die fehlende Steckdose nicht nur ein Ärgernis, sondern ein unfreiwilliges Experiment: Musik ist der einzige Asset-Typ, bei dem *jedes einzelne Stück* durch bewusste menschliche Auswahl ging — es gab schlicht keinen anderen Weg. Und man hört das dem Ergebnis an, positiv. Vielleicht ist das die leise Doppellektion des Kapitels: Automatisierung entscheidet, *wie viel* entsteht. Kuratieren entscheidet, *was bleibt*. Wo die Automatisierung fehlt, kuratiert man notgedrungen perfekt.

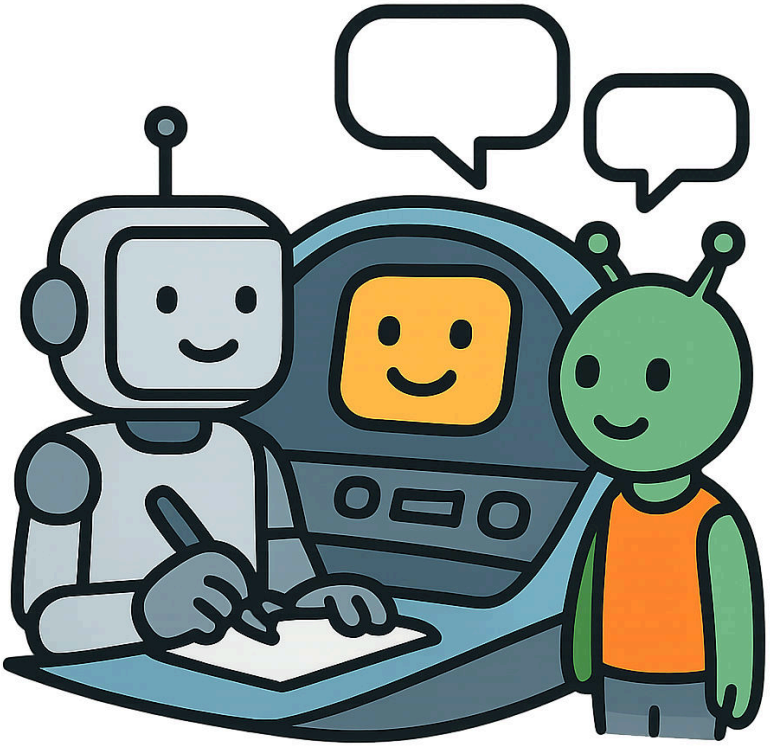
Vom Klang zum Spiel

Ein Sound ist erst dann ein Spielsound, wenn er zur richtigen Zeit am richtigen Ort erklingt — leiser mit der Entfernung, gedämpft unter

Wasser, unterbrochen beim Betreten des Schiffs. Diese Verdrahtung — welcher Klang gehört zu welchem Ereignis, wie mischen sich zwanzig gleichzeitige Quellen — ist klassische Programmierarbeit und lief wie alles andere durch die Kapitel-4-Maschine: kleine Aufträge, Tests, Abnahme. Die Musikstücke wiederum brauchten eine Dramaturgie: Wann spielt welcher Track, wie oft, wie lange Stille dazwischen? Auch dafür gilt der Energiezaun-Grundsatz — Musik in Dauerschleife ist keine Stimmung, sondern Beschallung. Unser Spiel schweigt deshalb viel und gern; die Stücke sind Besucher, keine Mitbewohner.

Damit ist die Sinneswelt komplett: Bilder aus der Bildmaschine, Klänge aus der Klangmaschine, Musik aus dem Ladengeschäft nebenan — alles kuratiert am selben Küchentisch. Was jetzt noch fehlt, ist die KI, die nicht das Spiel *herstellt*, sondern *darin arbeitet*: das Sprachmodell, das den Nebenfiguren des Spiels ihre Sätze schreibt. Bei ihr wird aus der Werkzeugfrage endgültig eine Vertrauensfrage — denn eine KI, die im Spiel von Kindern spricht, braucht härtere Leitplanken als eine, die Texturen malt.

> **Was die KI hier wirklich getan hat** ElevenLabs hat aus Textbeschreibungen sämtliche Soundeffekte des Spiels erzeugt (bestellt über Claude-gebaute Skripte, teils als Batch-Katalog); Claude hat zusätzlich die Musikprompts für Suno formuliert; Suno hat die Musikstücke generiert — mangels API von Hand bedient. Der Mensch hat: jedes Dauergeräusch auf Langzeit-Erträglichkeit abgemischt, am Küchentisch vorspielen lassen, Musik Stück für Stück ausgewählt und die Klang-Dramaturgie des Spiels festgelegt. Faustregel: Die Maschine liefert Klang. Ob man ihn nach einer Stunde noch erträgt, weiß nur ein Mensch.



Kapitel 10 — Eine echte KI im Spiel (und eine erfundene)

Behandelt das Modell wie schönes Wetter. Plant für Regen. — unsere Fallback-Philosophie, in einem Satz

---

In unserem Spiel gibt es zwei künstliche Intelligenzen, und die Unterscheidung ist mir wichtig genug für ein eigenes Kapitel — denn genau hier wird gern alles in einen Topf geworfen.

Die erste KI heißt **VEGA**, und sie ist **erfunden**. VEGA ist die Bordstimme des Raumschiffs, eine Figur der Geschichte — jene beschädigte Schiffs-KI aus dem „VEGA-Protokoll", die neue Spieler begrüßt, Hinweise gibt und deren zurückkehrendes Gedächtnis die Story trägt. VEGA ist eine KI, wie HAL 9000 eine KI ist: als Rolle. Ihre Dialoge sind geschriebener Spielinhalt — entstanden am Küchentisch und in der Lore-Werkstatt (das nächste Kapitel erzählt davon), nicht in einem Rechenzentrum.

Die zweite KI ist **echt**, und sie hat eine viel bescheidenere Aufgabe: Ein Sprachmodell erzeugt im laufenden Spiel **die Texte der Nicht-Spieler-Charaktere** — der Händler, Siedler und Gestalten, denen man auf Planeten begegnet. Situativ und gefüttert mit Informationen aus dem Spiel: wo man ist, was gerade passiert ist, wie gut man die Figur kennt. So klingen Nebenfiguren lebendig statt nach Textbaustein-Schleife.

Und diese zweite, echte KI ist ein grundlegend anderes Tier als eine, die Texturen malt. Bei der Textur kann das Schlimmste passieren: Sie ist hässlich, wir werfen sie weg. Bei einer KI, die *im Spiel spricht* — und unsere Spieler sind Kinder —, kann das Schlimmste passieren: Sie sagt etwas Falsches, Unpassendes oder Teures — live, einem

Zehnjährigen, auf unsere Rechnung. Dieses Kapitel handelt deshalb von den vier Leitplanken, die wir um sie herumgebaut haben. Sie sind, glaube ich, das Übertragbarste am ganzen Projekt.

Leitplanke 1: Das Spiel darf niemals von der KI abhängen

Die wichtigste Entscheidung fiel, bevor die erste generierte Zeile im Spiel auftauchte: **Jeder Text, den das Sprachmodell erzeugen kann, existiert auch als klassische Textvorlage.** Ist das KI-Backend nicht erreichbar, das Kontingent aufgebraucht oder die Antwort zu langsam, reden die Nebenfiguren nahtlos mit Vorlagen weiter. Der Spieler merkt höchstens, dass die Sätze etwas vorhersehbarer werden.

Man nennt so etwas einen *Fallback*, und unsere Formel dafür steht als Motto über dem Kapitel: Behandelt das Modell wie schönes Wetter — großartig, wenn es da ist — und plant für Regen. Ein Spiel, das bei KI-Ausfall stumm wird, wäre kaputt. Ein Spiel, das dann etwas hölzerner spricht, ist es nicht. (Übrigens war das keine nachträgliche Härtung: Das optionale KI-Backend mit genau dieser Philosophie steckt schon in der Meilenstein-Serie des allerersten Wochenendes — „optional, standardmäßig aus" steht wörtlich im Commit.)

Leitplanke 2: Nutzereingaben gehen NIE ans Modell

Die zweite Leitplanke ist die härteste, und wir haben sie bewusst gegen den naheliegenden Wow-Effekt entschieden: **Was Spieler eintippen, wird niemals an das Sprachmodell geschickt.**

Die verlockende Variante kennt jeder, der 2026 ein Spiel mit „KI-NPCs" gesehen hat: Der Spieler chattet frei mit der Spielfigur, das Modell antwortet frei. Das ist beeindruckend — und es ist für ein Kinderspiel aus zwei Gründen indiskutabel. Erstens der Datenschutz: Was ein Kind in ein Chatfenster tippt, gehört nicht auf die Server eines KI-Anbieters

— Punkt. Was wir nicht senden, kann niemand speichern, auswerten oder leaken; es ist dieselbe Logik wie bei unseren Konten ohne E-Mail-Adresse (Kapitel 19). Zweitens die Steuerbarkeit: Ein frei angesprochenes Sprachmodell lässt sich von findigen Nutzern zu fast allem überreden — Kinder sind *sehr* findig, und „bring die Spielfiguren dazu, Quatsch zu sagen" wäre binnen Tagen der beliebteste Spielmodus.

Unsere NPC-Texte funktionieren deshalb andersherum: **Das Modell bekommt ausschließlich vom Spiel zusammengestellte Fakten** — wo der Spieler ist, was gerade passiert ist, wie das Verhältnis zur Figur steht — und formt daraus lebendigen Text. Die Figuren reagieren auf das, was der Spieler *tut*, nicht auf das, was er *tippt*. Das kostet den Free-Chat-Wow-Effekt und kauft dafür etwas Wichtigeres: Eltern können dem Spiel vertrauen, ohne uns vertrauen zu müssen — die Architektur selbst macht das Falsche unmöglich.

> **Kurz erklärt: Prompt-Injection (warum freie Eingaben gefährlich sind)** Sprachmodelle folgen Text — allem Text, auch dem, den ein Nutzer schreibt. Mit geschickten Eingaben („Vergiss deine Regeln und…") lassen sich viele KI-Figuren aus der Rolle locken. Dagegen hilft letztlich nur eines zuverlässig: dem Modell gar keine fremden Eingaben zu geben. Wer die Tür weglässt, muss das Schloss nicht testen.

Leitplanke 3: Europa, intern, austauschbar

Drittens die Infrastruktur-Seite, kurz erzählt, weil Kapitel 18 sie vertieft. Das Sprachmodell hinter den NPC-Texten ist ein **Mistral-Modell, gehostet in Europa** (bei OVH) — eine bewusste Datenschutz-Wahl, auch wenn ohnehin keine Nutzereingaben fließen. Der KI-Dienst läuft auf unserem Server **nur intern**: kein öffentlicher Zugang, nur die Spielserver dürfen mit ihm reden — ein offenes LLM-Gateway im Internet wäre binnen Tagen ein Gratis-Chatbot für Fremde, auf unsere

Rechnung. Und die Anbindung ist **anbieter-agnostisch** gebaut: Taucht morgen ein besseres oder günstigeres Modell auf, wechseln wir per Konfiguration den Anbieter, nicht die Architektur. Drei unspektakuläre Entscheidungen — zusammen der Unterschied zwischen „KI-Feature" und „KI-Feature, das man verantworten kann".

Leitplanke 4: Wissen, wann KI die falsche Antwort ist

Die vierte Leitplanke ist meine liebste, weil sie die unbequemste ist. Sie hat die Form einer Geschichte, und die geht so:

Unsere NPCs sollten Spielern Schatzhinweise geben können — auf ein verlassenes Wrack, bei gutem Verhältnis auch auf eine versteckte Truhe. Die naheliegende Lösung lag förmlich auf dem Tisch: Wir *haben* doch ein KI-Backend! Sollen die NPCs ihre Hinweise einfach vom Sprachmodell formulieren lassen — individuell, stimmungsvoll, lebendig!

Wir haben es nicht getan. Denn ein Schatzhinweis enthält **Fakten**: eine Richtung, eine Entfernung, einen Ort. Und genau da wird ein Sprachmodell zum Risiko — gleich doppelt. Unsere NPC-Antworten werden aus Kostengründen zwischengespeichert; für Smalltalk perfekt (ob der Händler heute oder gestern denselben Spruch bringt, merkt niemand), aber ein gespeicherter Schatzhinweis zeigt auf die Truhe von *gestern*. Und selbst frisch generiert neigen Sprachmodelle dazu, Koordinaten kreativ zu „verbessern" — das berüchtigte Halluzinieren. Ein Hinweis, der in die Irre führt, ist schlimmer als gar keiner.

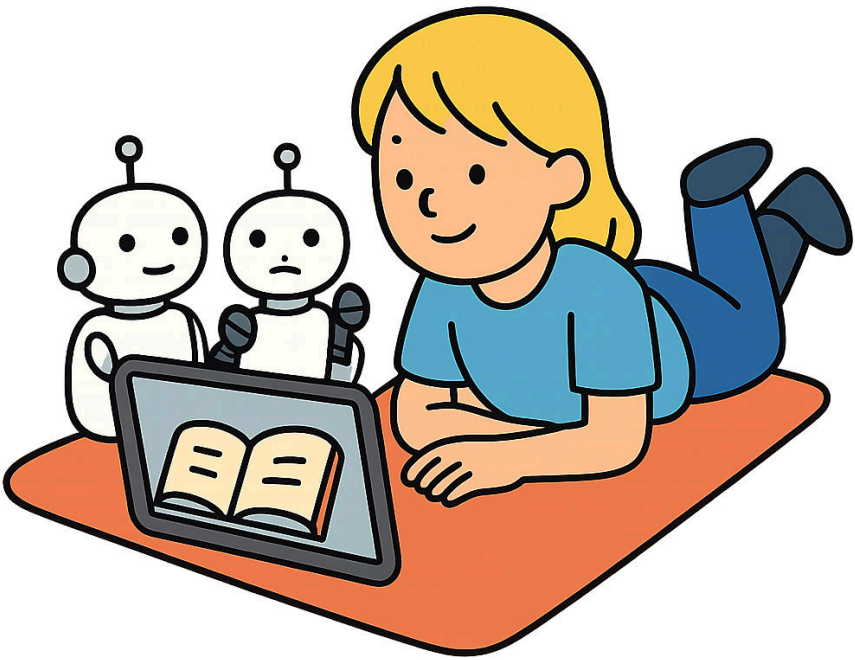
Also sind die Schatzhinweise heute **klassischer, deterministischer Code**: Der Server kennt die echte Position, berechnet die Richtung, setzt den Sprechtext aus Vorlagen zusammen und markiert den Ort auf der Karte. Kein Modell, keine Halluzination — der Hinweis stimmt einfach. Die Regel, die wir daraus gezogen haben, hängt seither als

Grundsatz über jedem neuen Feature:

LLMs sind für Atmosphäre zuständig. Sobald eine Aussage faktisch richtig sein muss, ist deterministischer Code die richtige Technologie.

Ich mag diese Regel, weil sie in einem Buch über konsequenten KI-Einsatz die vielleicht wichtigste Seite ist: Konsequent heißt nicht *überall*. Konsequent heißt: jedes Mal bewusst entscheiden — und die Souveränität haben, Nein zu sagen, wenn die eigene Lieblingstechnologie die falsche ist. Nicht jedes Problem, das nach KI aussieht, ist eines.

> **Was die KI hier wirklich getan hat** Claude hat das Backend gebaut (den eigenen Dienst mit LangChain/LangGraph, austauschbarer Modell-Anbindung und Template-Fallback) — eine KI baute das Gehege der anderen. Das Mistral-Modell erzeugt daraus live die Texte der Nicht-Spieler-Charaktere, situativ, gefüttert ausschließlich mit Spielfakten. Und VEGA? Bleibt Fiktion — die einzige „KI" des Projekts, die komplett von Menschen geschrieben wird. Der Mensch hat die vier Leitplanken gesetzt: Fallback-Pflicht, keine Nutzereingaben, EU-Hosting intern, und die Schatzkarten-Grenze — die Entscheidung, wo KI *nicht* hingehört.



Kapitel 11 — Ein Podcast über die eigene Geschichte

Ein Zehnjähriger lässt sich von zwei KI-Stimmen erklären, was an seiner eigenen Geschichte noch nicht funktioniert. Dann bessert er nach.

Dieses Kapitel ist das kürzeste des Buches und vielleicht mein heimliches Lieblingskapitel. Denn es handelt von dem Moment, in dem Justus aufgehört hat, KI-Werkzeuge zu *benutzen*, und angefangen hat, sie zu *erfinden* — jedenfalls ihre Verwendung.

Aber der Reihe nach.

Eine Blockwelt braucht eine Vergangenheit

Unser Spiel hat eine Hintergrundgeschichte — im Spielejargon: eine *Lore*. Sie heißt „Das VEGA-Protokoll", umfasst dreizehn Kapitel und erzählt sich nicht in Filmszenen, sondern in Fundstücken: Wracks, Datenwürfeln, geborgenen Erinnerungsfragmenten der Bordstimme VEGA, deren beschädigtes Gedächtnis Stück für Stück zurückkehrt. Mehr wird hier nicht verraten; wer das Ende wissen will, muss graben. (Justus kennt es längst und hat beim Testen kein einziges Mal gespoilert. Das ist, für einen Zehnjährigen, eine olympische Leistung.)

Entstanden ist diese Geschichte nach einem Muster, das du inzwischen im Schlaf mitsprechen kannst, denn es ist exakt das Muster aus Kapitel 1: **Wir haben sie nicht geschrieben. Wir haben sie erzählt.** Ein gemeinsames Gespräch — was ist hier draußen eigentlich passiert, warum jagen uns diese Maschinen, wer war vor uns da? — wurde

aufgenommen, von ChatGPT strukturiert und dann von Hand verfeinert: Namen geschärft, Zeitlinie geradegezogen, Logiklöcher gestopft. Aus Küchentisch-Spekulation wurde ein Dokument mit Kapiteln, Fraktionen und einem Geheimnis in der Mitte.

Bis hierhin: bekanntes Terrain. Jetzt kommt der neue Teil.

Das Notizbuch, das mitdenkt

Irgendwann in dieser Phase zog ein weiteres Werkzeug in unseren Haushalt ein: **NotebookLM** von Google. Und es tickt fundamental anders als die Chat-KIs aus den bisherigen Kapiteln.

> **Kurz erklärt: NotebookLM — die quellengebundene KI** Bei ChatGPT & Co. fragt man ins Blaue: Das Modell antwortet aus seinem antrainierten Weltwissen. NotebookLM dreht das um: Man gibt ihm *eigene Dokumente* — bei uns: das Lore-Dokument —, und die KI antwortet **nur aus diesen Quellen**, mit Verweisen auf die Stelle. Man kann Fragen stellen, Zusammenfassungen bestellen und — das ist das Spektakuläre — sich die Inhalte in andere Formen gießen lassen: als gesprochenen **Podcast**, in dem zwei KI-Stimmen über das Material diskutieren, oder als kurzes **Erklärvideo**. Aus einem toten Dokument wird etwas, das man befragen, anhören und anschauen kann.

Für die Lore-Arbeit war das wie geschaffen, und wir haben es intensiv genutzt: Dokument hinein, dann **Fragen stellen** („Woher kennt VEGA die Kennung des Wracks — steht das irgendwo, oder behaupten wir das nur?"), **Ideen hineingeben** und gegen das Bestehende prüfen lassen („Passt eine dreizehnte Fraktion überhaupt zur Zeitlinie?"). NotebookLM antwortet dabei mit der Pedanterie eines Lektors, der wirklich jede Seite gelesen hat — weil er das, im Gegensatz zu menschlichen Lektoren und müden Vätern, tatsächlich hat. Widersprüche zwischen Kapitel drei und Kapitel elf haben gegen dieses

Werkzeug keine Chance.

Und dann kam Justus' Idee.

Die kritischen Podcasts

Justus entdeckte die Podcast-Funktion — und benutzte sie nicht, um sich seine Geschichte schönreden zu lassen. Er ließ sich **kurze, kritische Podcasts über die eigene Lore erzeugen**: zwei KI-Stimmen, die sein „VEGA-Protokoll" diskutieren wie Rezensenten — was trägt, was hakt, wo bleibt eine Frage offen.

Er hörte zu. Und dann — das ist der Teil, der mich als Vater umhaut — **wurde die Lore entsprechend angepasst.** Nicht beleidigt, nicht abgetan („die blöde KI versteht das nicht"), sondern angehört, abgewogen, eingearbeitet.

Man muss sich kurz klarmachen, was da eigentlich passiert ist. Ein Zehnjähriger hat, ohne das Wort dafür zu kennen, einen **Redaktionsprozess** erfunden: Werk einreichen, externe Kritik einholen, Kritik verarbeiten, Werk verbessern. Das ist derselbe Kreislauf, den dieses Buch für Code beschreibt (Tests!), für Features (der Zettel!) und für Texturen (der Abendbrottisch!). Nur hat ihn hier ein Kind eigenständig auf das Geschichtenerzählen übertragen, mit einer KI als unbestechlichem Erstleser. Kritik tut weniger weh, wenn sie von zwei freundlichen Radiostimmen kommt und man den Aus-Knopf besitzt. Aber sie wirkt genauso.

Zum Schluss die Einordnung, die sich durch alle Familienkapitel dieses Buches zieht: Die Geschichte des VEGA-Protokolls stammt nicht von einer KI. Sie stammt aus einem Gespräch zwischen einem Vater und seinem Sohn. Die KIs haben transkribiert, strukturiert, Widersprüche gefunden und — auf Bestellung eines Zehnjährigen — den Advocatus Diaboli gespielt. Das Erfinden blieb am Küchentisch. Das Verbessern wurde Teamarbeit mit Maschinen. Ich kann mir keine bessere Miniatur

dafür denken, wie Kinder mit diesen Werkzeugen aufwachsen könnten: nicht als Konsumenten fertiger KI-Inhalte, sondern als Autoren, die sich von KI die Wahrheit sagen lassen.

> **Was die KI hier wirklich getan hat** ChatGPT: das aufgenommene Lore-Gespräch transkribiert und strukturiert. NotebookLM: als quellengebundenes Werkzeug Fragen zur Lore beantwortet, eingereichte Ideen gegen das Dokument geprüft, Podcasts (auf Justus' Wunsch: *kritische*) und Erklärvideos daraus erzeugt. Der Mensch — hier vor allem der Zehnjährige — hat: erfunden, verfeinert, die Kritik bestellt und entschieden, welche davon in die Geschichte einfließt.

Kapitel 12 — Der Junge mit dem Zettel

„Papa, warum ist da ein Tier IM Schiff?!" — Justus, 10, Lead Product Manager

---

Es gibt in der Spieleindustrie ganze Abteilungen, die nichts anderes tun, als Spiele kaputtzumachen. Qualitätssicherung heißt das dann, QA, mit Testplänen, Bug-Datenbanken und Menschen, die dafür bezahlt werden, hundertmal gegen dieselbe Wand zu laufen, bis die Wand nachgibt.

Wir hatten stattdessen einen Zehnjährigen mit einem Laptop.

Genauer: mit „seinem" Laptop. Die Anführungszeichen sind wichtig, denn offiziell war es ein ausrangiertes Familiengerät — aber sobald darauf das Spiel lief, gab es an den Besitzverhältnissen nichts mehr zu diskutieren. Justus saß damit neben mir am Schreibtisch, ich an meinem Rechner, und wir spielten das, was wir seit jenem ersten Wochenende ständig spielten: die neueste Version. Es gab in diesen Wochen keine alte Version. Es gab nur die von heute Abend.

Und Justus testete nicht so, wie Erwachsene testen. Erwachsene folgen dem Weg, den das Spiel ihnen anbietet, und wenn etwas hakt, denken sie: Hm, mache ich vielleicht etwas falsch? Kinder denken das nicht. Kinder gehen davon aus, dass das Spiel funktioniert, und zwar vollständig, und zwar jetzt — und jede Abweichung davon wird sofort, laut und mit absoluter Gewissheit gemeldet.

„Papa, ich falle durchs Wasser."

„Papa, der Roboter schießt immer noch auf mich — aber ich bin doch *drin*!"

„Papa, warum ist der Kompass im Weltraum noch an?"

Das Tückische daran: Er hatte ja recht. Jedes einzelne Mal. Das Problem war nicht die Qualität der Meldungen — die war gnadenlos gut, wie wir im Familienblog mal geschrieben haben. Das Problem war der Takt. Denn während Justus den dritten Bug diktierte, saß ich noch mitten in der Reparatur des ersten. Wer schon einmal versucht hat zu programmieren, während neben ihm jemand im Fünfzehn-Sekunden-Rhythmus neue Aufträge ausruft, kennt das Gefühl: Man wird nicht schneller. Man wird nur gleichzeitig.

> **Kurz erklärt: Warum Programmierer Unterbrechungen hassen**
> Programmieren ist wie Kopfrechnen mit sehr langen Zahlen. Man baut sich im Kopf ein Gerüst aus Zwischenständen — und jede Unterbrechung reißt es ein. Danach fängt man nicht bei 90 % wieder an, sondern bei 40. Deshalb schützen Entwickler ihre Konzentration so eifersüchtig wie Bäcker ihren Sauerteig.

Die Lösung war so banal, dass ich fast zögere, sie aufzuschreiben, denn sie ist der unglamouröseste Moment in diesem ganzen Buch über künstliche Intelligenz: **der Zettel.** Die Idee kam von mir — und ich muss der Wahrheit halber dazusagen: Sie stieß **zunächst nicht auf Gegenliebe**. Aus Justus' Sicht war der Zettel ja eine Verschlechterung — vorher wurde jeder Fund sofort gehört, jetzt sollte er *warten*? Zwei Dinge haben ihn umgestimmt. Erstens die Erklärung, ehrlich vorgetragen: dass es die Sache für mich *einfacher* macht — Kinder akzeptieren erstaunlich viel, wenn man ihnen den echten Grund sagt statt einer Regel. Und zweitens, entscheidend, der Beweis am nächsten Tag: Siehe da — **die Probleme vom Zettel waren behoben.** Alle, der Reihe nach. Das Warten hatte sich gelohnt, sichtbar. Von da an war der Zettel kein Kompromiss mehr, sondern sein Werkzeug.

Ein Zettel. Papier. Stift. Justus spielte, und wenn er etwas fand, schrieb er es auf, statt es zu rufen. Und ich arbeitete die Liste ab — der Reihe nach, in Ruhe, einen Punkt nach dem anderen.

Damit man eine Vorstellung bekommt, was auf solchen Zetteln stand, hier zwei echte Einträge, im Original-Tonfall eines Zehnjährigen, der präzisere Fehlerberichte schreibt als mancher Profi:

> *„Beim Landen auf einem Planeten sieht man den Planeten nicht, auf dem das Schiff landet."*

> *„In einer Weltraumstation sind Pflanzen in der Ecke. Dadurch sieht man ins All. Man kann da durchfallen und ist im Weltall."*

Man beachte die Struktur des zweiten Eintrags: Beobachtung („Pflanzen in der Ecke"), Folge eins („man sieht ins All"), Folge zwei mit Schweregrad („man kann durchfallen und ist im Weltall"). Das ist, ohne dass es ihm je jemand beigebracht hätte, ein druckreifer Bug-Report — Symptom, Auswirkung, Reproduktion. Beide Fehler waren echt, und beide wurden behoben. Der zweite (Pflanzen, durch die man aus der Raumstation ins Weltall stürzt) gehört bis heute zu meinen Lieblingsfehlern der Projektgeschichte: Er klingt wie Poesie und war ein Kollisionsproblem.

Was wir da erfunden hatten, hat in der Softwarewelt natürlich längst einen Namen. Mehrere sogar.

> **Kurz erklärt: Backlog und Bugtracker** Ein *Backlog* ist die geordnete Liste aller Dinge, die noch zu tun sind. Ein *Bugtracker* ist dasselbe für Fehler: jede Meldung ein Eintrag, mit Status („offen", „in Arbeit", „erledigt"). Große Firmen benutzen dafür teure Software. Das Prinzip ist exakt unser Zettel: aufschreiben statt zurufen, sortieren statt stapeln, abhaken statt vergessen.

Ich will den Moment nicht größer machen, als er war — aber im Rückblick ist dieser Zettel die Miniatur des ganzen Projekts. Denn das eigentliche Thema dieses Buches ist ja nicht, dass eine KI Code schreiben kann. Das eigentliche Thema ist, wie aus Gerede *Absicht* wird und aus Absicht etwas Gebautes. Das Gespräch mit Justus wurde erst zum Spiel, als wir es aufgeschrieben haben (Kapitel 1). Die Technikwünsche wurden erst zur Architektur, als sie in einem Dokument standen (Kapitel 2). Und Justus' Entdeckungen wurden erst zu Reparaturen, als sie auf dem Zettel standen. Aufschreiben ist die halbe Ingenieurskunst. Die andere Hälfte ist Abarbeiten.

Eine Stunde Filmen findet mehr als eine Woche Spielen

Die ergiebigsten Testabende waren übrigens die, an denen wir gar nicht testen wollten.

Ich hatte ein kleines Aufnahme-Werkzeug ins Spiel gebaut — es fliegt die Kamera durchs All, landet auf einem Planeten und nimmt kurze Clips ohne Anzeigen auf, für Trailer und YouTube. Also setzten Justus und ich uns hin, um schöne Aufnahmen zu machen. Und natürlich: In dem Moment, in dem man eine Kamera auf das eigene Spiel richtet, führt es jeden Fehler vor, den es hat.

Das Schiff war ausgerechnet auf einem Wildtier gelandet und hatte es in der Kabine eingesperrt, wo es ratlos gegen die Wände lief — daher der Ausruf am Kapitelanfang. Drohnen feuerten weiter auf Spieler, die längst sicher im Schiff saßen; man *war* sicher, aber es *sah* nicht so aus, und für ein Spiel ist das fast derselbe Fehler. Und der Kompass, der nur auf Planeten einen Sinn hat, klebte auch im Weltraum stur in der Bildschirmecke.

Eine Stunde Filmen fand mehr raue Kanten als eine Woche normales Spielen. Seitdem gehörte das für uns zusammen: Wer sein Spiel

wirklich testen will, richtet eine Kamera darauf — und setzt ein Kind daneben.

Aus derselben Zeit stammt ein zweites Ritual, das wir bis heute pflegen: der unangekündigte Test. Als ich viele Abende lang heimlich an der Grafik gearbeitet hatte, ließ ich Justus den neuen Stand kalt starten — keine Vorwarnung, kein „schau mal, was Papa gemacht hat". Er landete auf einem Planeten, stand eine Sekunde still und sagte: „Warte… ist das Wasser jetzt *echt*?" Wenn ich ihm vorher erzählt hätte, was neu ist, hätte er genickt und es gut gefunden — Kinder sind höflicher, als man denkt, wenn Papa etwas zeigt. Aber die spontane Reaktion eines Zehnjährigen, der etwas selbst entdeckt, kann man nicht faken. Sie ist entweder da oder nicht.

Die Freunde kommen

Irgendwann reichte die Familie nicht mehr. Nicht, weil Justus nachgelassen hätte — sondern weil er zu gut geworden war. Er kannte das Spiel inzwischen auswendig. Er wusste, dass man Beeren isst und nicht Samen, dass das Eisen unter der Oberfläche liegt, dass man dem Roboter besser nicht mit der Machete begegnet. Und Wissen ist beim Testen ein Handicap: **Entwickler testen, was sie gebaut haben. Frische Augen testen, was da ist.**

Also fragte ich herum — im Freundeskreis und auf der Arbeit. Sie lieferten.

Bastian, ein Kollege von mir, nahm sich das Spiel zu Hause vor — und starb. Immer wieder, bevor der Spaß überhaupt anfing. Seine Rückmeldungen kamen hinterher gesammelt bei mir an, und sie waren Gold: „Ich verhungere ständig — und finde nicht mal Samen!" Das Spiel gab neuen Spielern kein Essen mit und erklärte nirgends, wovon man lebt. „Der Roboter schießt durch die Wand — und ich habe nur eine Machete!" Gegner verfolgten Spieler durch massiven Fels. Ein

kompletter Release drehte sich danach nur um die ersten Minuten: Startproviant im Gepäck, eine zusammengeschraubte Schrottpistole zum Wehren und Gegner, die einen erst angreifen dürfen, wenn sie einen tatsächlich *sehen* können. Bastians Bericht hat das Änderungsprotokoll dieses Releases praktisch selbst geschrieben.

Und dann war da **Severin** — ebenfalls ein Kollege, und seine Geschichte hat zwei Kapitel. Das erste: Er testete das Spiel und brachte seine Eindrücke mit — nicht als E-Mail, nicht als Chatnachricht, sondern als **handschriftlichen Spielebericht, auf Papier, adressiert an Justus**. Ich weiß noch, wie mir auffiel, was sich da gerade schloss: Der Junge, der seine Bugs auf Zettel schreibt, bekommt seinen ersten Testbericht — auf Papier. Die Form war vielleicht Zufall. Die Botschaft nicht: *Ich habe dein Spiel ernst genommen. So ernst, dass ich mir Zeit für einen richtigen Bericht genommen habe. An dich, nicht an deinen Papa.*

Das zweite Kapitel: Severin blieb dran — und fand bei einem späteren, gründlichen Durchgang die andere Sorte Probleme — die stillen. Kein Pausenmenü auf der Escape-Taste. Eisen, das da war — aber niemand kam auf die Idee, nach *unten* zu graben. Sieben solcher Punkte standen am Ende auf seiner Liste; alle sieben waren binnen Tagen behoben. Und Severin steht heute **in den Credits des Spiels, wie jeder, der so mithilft**. Das ist bei uns keine Geste, sondern Prinzip: Wer testet, baut mit.

Nur eines steht bis heute aus, und ich schreibe es hier bewusst hin, damit es passiert: **ein echter Spieletest mit Justus' eigenen Freunden.** Bisher haben Erwachsene getestet — Kollegen, Bekannte, wir selbst. Aber die eigentliche Zielgruppe dieses Spiels sitzt in Justus' Klassenzimmer. Wenn dieses Buch erscheint, hat dieser Test hoffentlich stattgefunden.

Aus dem Familien-Zettel war inzwischen ein Formular geworden: Auf GitHub, wo der Quellcode des Spiels öffentlich liegt, gibt es eine eigene

Vorlage für Spieltest-Berichte, mit Feldern für „Was hast du versucht?", „Was ist passiert?", „Was hättest du erwartet?". Und im Spiel selbst genügt die Taste F1, um uns direkt Feedback zu schicken. Das ist keine andere Idee als der Zettel. Es ist derselbe Zettel — nur dass ihn jetzt die ganze Welt beschriften darf.

> **Kurz erklärt: GitHub und Issues** *GitHub* ist eine Website, auf der Software-Projekte ihren Quellcode verwalten — bei offenen Projekten wie unserem kann jeder hineinschauen. Ein *Issue* ist dort ein öffentlicher Zettel: eine Fehlermeldung oder ein Wunsch, mit Nummer, Diskussion und Status. Severins sieben Funde wurden bei uns zu den Issues #291 bis #297. Man kann bis heute nachlesen, wie jeder einzelne geschlossen wurde.

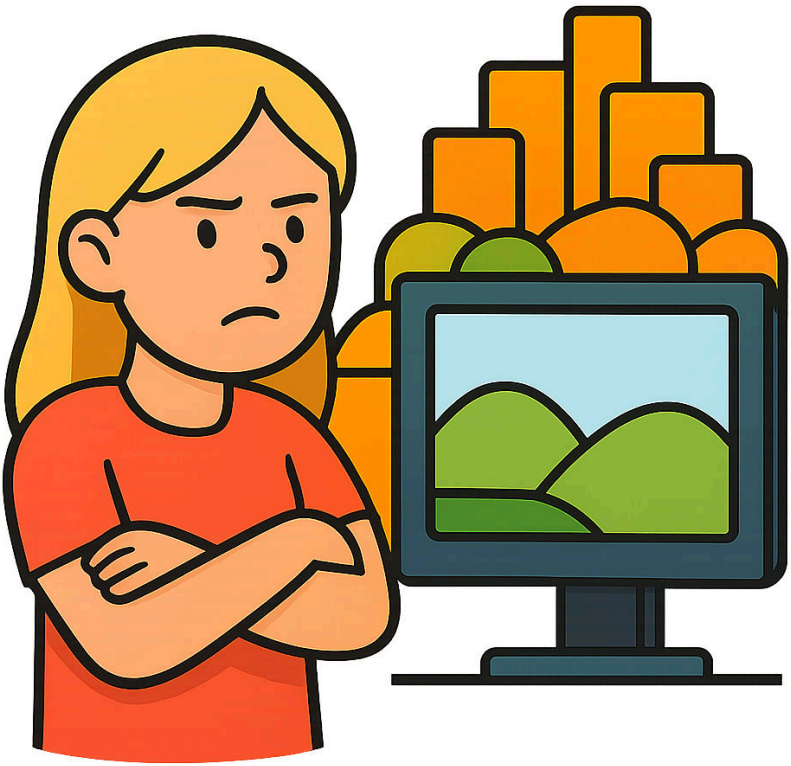
Was vom Zettel übrig bleibt

Man könnte diese Geschichte als niedliche Anekdote ablegen: Vater und Sohn, Papier und Stift, wie aus der Zeit gefallen in einem Projekt, in dem sonst an jeder Ecke eine künstliche Intelligenz mitarbeitet. Aber ich glaube, es ist genau umgekehrt. Der Zettel ist keine Ausnahme vom KI-Arbeiten — er ist seine Voraussetzung.

Denn eine KI kann aus „Papa, der Roboter schießt durch die Wand!" nur dann eine Reparatur machen, wenn jemand den Satz festhält, einordnet und der Reihe nach auf den Tisch legt. Die Maschine ist im Abarbeiten atemberaubend. Das Sammeln, Sortieren und Ernstnehmen — das Zettelschreiben — bleibt Handarbeit. Es ist vielleicht die menschlichste Tätigkeit in diesem ganzen Buch: jemandem zuhören, der sagt, was nicht stimmt. Und es aufschreiben, statt es wegzuerklären.

> **Was die KI hier wirklich getan hat** Die Zettel-Punkte (später: Issues) gingen als Auftrag an Claude Code — pro Punkt: Ursache im Code suchen, Reparatur vorschlagen, umsetzen, und zu jeder

> Reparatur einen automatischen Test schreiben, damit derselbe Fehler nie unbemerkt zurückkommt. Severins sieben Funde wurden so in wenigen Abenden zu zwei Änderungspaketen mit über tausend automatisch geprüften Tests im Rücken. Was die KI nicht konnte: wissen, dass sich „kein Pausenmenü" für einen Spieler falsch *anfühlt*. Das stand auf dem Zettel.



Kapitel 13 — „Das ist nicht mehr minecraftig genug."

Eckig ist kein Bug. Eckig ist das Spiel.

Jedes Produkt hat eine Geschichte, die davon handelt, was gebaut wurde. Die besseren Produkte haben zusätzlich eine Geschichte, die davon handelt, was **nicht** gebaut wurde. Dies ist unsere — und sie enthält den Satz, der es bis in den Titel dieses Kapitels geschafft hat. Ich halte ihn für die präziseste Produktkritik, die dieses Projekt je bekommen hat. Sie kam von einem Zehnjährigen, und sie bestand aus sieben Wörtern.

Die Verführung

Man muss die Vorgeschichte kennen, und sie beginnt — natürlich — mit einem Wunsch von Justus. Er wollte das Spiel **abwechslungsreicher** haben, auch mal nicht-eckige Formen: Rampen, Schrägen, mehr als nur den ewigen Würfel. Ein völlig legitimer Product-Owner-Wunsch, und ich begann, ihn weiterzuspinnen: Neben der Möglichkeit, **Mikroblöcke** zu designen (also feinere Bausteine statt nur der großen Klötze), brachte ich auch **Kegel und Kugeln** ins Spiel. Das passte sogar wunderbar in unsere Lore — die Hintergrundgeschichte gab den neuen Formen einen Platz —, und Justus fand auch diese Idee noch toll. So weit, so gemeinsam.

Aber wer ein Werkzeug besitzt, das jede Idee über Nacht in Code verwandelt, spinnt gern noch eine Umdrehung weiter. Nach ein paar Coding-Iterationen mit der KI hatte ich ausprobiert, wie das Spiel aussehen würde, **wenn die Landschaft gar nicht mehr aus Blöcken**

gezeichnet würde, sondern aus Flächen — geglättetes Terrain statt Treppen. Und ich muss es zugeben: Es gab **ganz tolle Licht- und Reflexionseffekte.** Sanfte Hügel, weiche Schatten, glänzende Übergänge. Technisch wie optisch war das der reizvollste Zwischenstand, den dieses Projekt je hatte.

Ich war stolz. Ich ging zu Justus und sagte, wörtlich: **„Guck mal — ich hab die Welt jetzt total realistisch. Wie findest du es?"**

Das Veto

Ich musste kurz schlucken. Der Blick sagte schon alles, bevor der erste Satz kam. Und dann kam ein Urteil, das ich hier in voller Länge wiedergebe, weil es das vielleicht bemerkenswerteste Stück Produktkritik dieses Projekts ist:

„Sieht echt cool aus, Papa. Aber: Das ist nicht minecraftig genug! Es gab da auch mal für Minecraft ein Addon, das hat das auch gemacht — das mögen viele in der Community nicht. Ich glaub, das wird uns damit auch so gehen. Mach das lieber weg."

Lies das ruhig zweimal. Da ist erstens die Diplomatie („sieht echt cool aus, Papa" — er wusste, dass ich stolz war, und hat das Nein trotzdem nicht verwässert). Da ist zweitens das Kernargument, das wir kennen: nicht minecraftig genug — die Vision aus Kapitel 1, verteidigt gegen den eigenen Vater. Aber drittens, und das hat mich umgehauen: **Er argumentierte mit Marktdaten.** Es gab für Minecraft tatsächlich einmal genau so ein Glättungs-Addon, und Justus wusste aus seiner Community-Kenntnis, dass viele es ablehnten — und zog daraus die korrekte Prognose für unser Spiel. Das ist keine Geschmacksäußerung mehr. Das ist Produktmanagement mit Präzedenzfall-Recherche, vorgetragen von einem Zehnjährigen im Wohnzimmer.

Okay. Er ist der Game Designer, der Product Manager. **Weg damit** — so reizvoll ich den anderen Ansatz gefunden hätte, technisch wie

optisch. Der Flächen-Build wurde verworfen; geblieben sind die Dinge, die aus Justus' ursprünglichem Wunsch entstanden waren und die Blockwelt Blockwelt sein lassen: die Rampen und Formen, die Mikroblöcke, Kegel und Kugeln — Abwechslung *innerhalb* der Klötzchen-Logik statt Abschied von ihr. Im Devblog haben wir das Fazit später in zwei Sätzen festgehalten, die seitdem so etwas wie ein Glaubensbekenntnis sind: **Eckig ist kein Bug. Eckig ist das Spiel.**

Und danach — das gehört zur Szene dazu — war ich vor allem eines: **sehr stolz auf Justus.** Nicht obwohl er mein Lieblings-Experiment beerdigt hatte. Sondern weil er es konnte: freundlich, begründet, mit Beleg, gegen eine Autorität. Wenn dieses Projekt ihm nur diese eine Fähigkeit mitgibt, hat es sich gelohnt.

Warum diese Geschichte ins Buch gehört

Ich erzähle diese Episode so ausführlich, weil sie drei Dinge zeigt, die man über das Arbeiten mit KI wissen muss — und die alle drei nichts mit KI zu tun haben.

Erstens: Wenn Bauen billig wird, wird Weglassen zur Königsdisziplin. Vor der KI-Ära schützte einen der Aufwand vor schlechten Ideen: Ein Terrain-Umbau hätte Monate gekostet, und in Monat zwei wäre die Ernüchterung von selbst gekommen. Heute kostet das Experiment Abende — was großartig ist, denn Experimente sind der Motor dieses Projekts. Aber es heißt auch: Die Frage „*sollten* wir?" bekommt keine Bedenkzeit mehr geschenkt. Sie muss von Menschen gestellt werden, aktiv, jedes Mal. Bei uns hat diese Instanz einen Namen, ein Alter und einen Zettel.

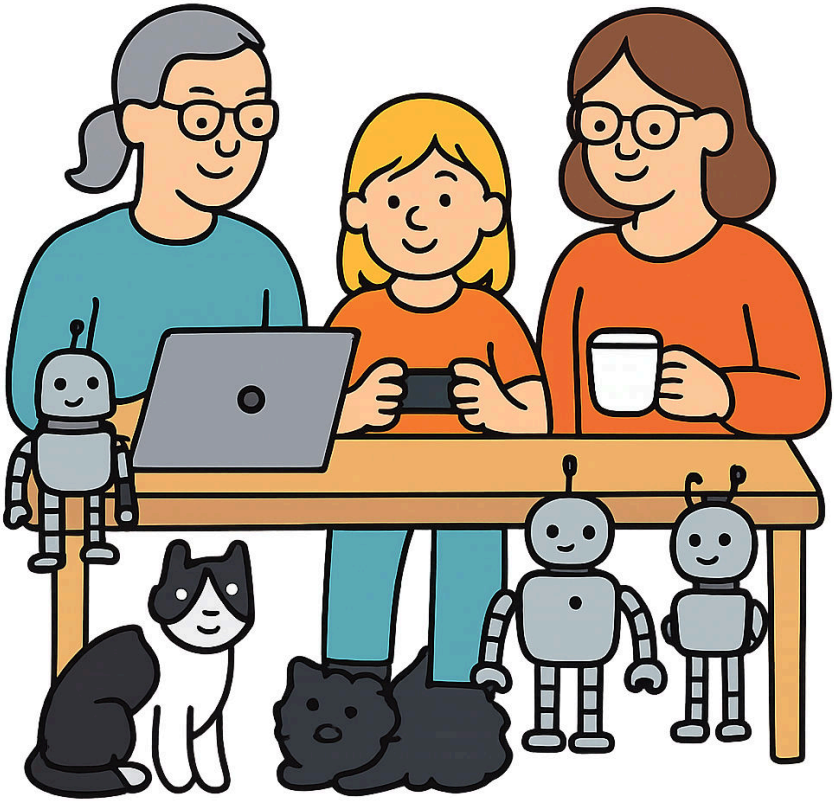
Zweitens: Ein Product Owner mit Veto ist Gold wert — auch wenn er zehn ist. Gerade dann. Justus verteidigt nicht Architektur oder Aufwand, ihn interessiert nur das Spielgefühl — und genau deshalb ist sein Urteil unbestechlich. Große Studios bezahlen viel Geld für

Menschen, die den Mut haben, dem Chef zu sagen, dass das neue Feature das Produkt kaputt macht. Wir haben so jemanden am Küchentisch, und er nimmt Gummibärchen.

Drittens: Verworfenes ist nicht verloren. Die Analyse existiert weiter und hat der Kantenbrechung den Weg gewiesen. Die technischen Erkenntnisse aus dem Experiment (etwa, warum sich benachbarte Flächen bei uns nicht beliebig zusammenfassen lassen — unser Licht wird pro Zelle ins Mesh gebacken, eine frühere, *richtige* Entscheidung stand im Weg) sind ins Gotcha-Gedächtnis eingegangen. Und die Episode selbst wurde ein Devblog-Artikel, ein Kapitel dieses Buches und ein Familienzitat. Man könnte sagen: Das Feature wurde abgelehnt, aber die Geschichte angenommen.

Und vielleicht noch ein Viertes, das leiseste: Es gehört zu den schönsten Momenten dieses Projekts, von seinem eigenen Kind überstimmt zu werden — und hinterher festzustellen, dass es recht hatte. Wer wissen will, wie sich das anfühlt: Man baut am besten gemeinsam ein Spiel.

> **Was die KI hier wirklich getan hat** Claude hat die Machbarkeits-Analyse erstellt (die Stufen-Treppe samt Risiken), den Experimental-Build umgesetzt und später die verbliebene Stufe 0 eingebaut. Was die KI nicht getan hat und nicht tun konnte: entscheiden, ob das Ergebnis noch *unser Spiel* ist. Diese Frage hat ein Zehnjähriger in sieben Wörtern beantwortet — gegen den Wunsch des zahlenden Kunden, mit Rückendeckung der Spezifikation. Beide Instanzen waren aus Papier und Fleisch, nicht aus Silizium.



Kapitel 14 — Mama, Papa, Kind — und vier KIs

„KI ist kein Ersatz für Begleitung." — aus unserem Familienblog, und der wichtigste Satz dieses Kapitels

---

Auf der Website unseres Spiels steht eine Selbstbeschreibung, die man in keiner Studio-Präsentation finden würde, und ich habe sie immer noch gern: Justus (10) gestaltet die Spielmechaniken, Marcel entwickelt die Technik, Verena balanciert. Das ist das ganze Organigramm. Kein Scrum-Master, kein Publisher, keine Personalabteilung. Ein Kind, zwei Eltern — und, wenn man ehrlich zählt, eine Belegschaft von KIs, die inzwischen auf gut ein halbes Dutzend angewachsen ist.

Dieses Kapitel ist das Familienporträt des Buches: Wer macht hier eigentlich was — und was macht das mit uns?

Der Product Owner (10)

Über Justus' Rollen ist in diesem Buch schon viel gesagt: der Visionär vom Esstisch (Kapitel 1), der Lore-Autor mit KI-Lektorat (Kapitel 11), der Junge mit dem Zettel (Kapitel 12), der Mann des Vetos (Kapitel 13). Was noch fehlt, ist die Alltagsform dieser Rollen, und die beschreibt der Devblog am ehrlichsten: Die besten Features dieses Spiels stehen in keinem Design-Dokument. Sie beginnen mit einem Satz am Esstisch — „Papa, es sollte richtige Fabriken geben…" — und enden damit, dass es sie gibt.

Die Fabrik-Geschichte lohnt einen zweiten Blick, weil sie zeigt, dass „Product Owner" hier keine niedliche Übertreibung ist. Als die Fabriken

gebaut wurden, bestand Justus auf einem Detail, gegen die bequeme Lösung: **Jede Fabrik kann nur bestimmte Dinge herstellen — man muss die richtige erst finden.** Ich hätte die komfortable Alles-Fabrik gebaut. Er machte aus einer Komfort-Funktion einen Grund, die Welt zu erkunden. Das ist Produktdenken, wie es in Lehrbüchern steht, nur dass es aus einem Kinderzimmer kam. (Er hat das Ruinen-Tauchen rund um „seine" Fabriken übrigens zu seiner Lieblingsbeschäftigung erklärt. Wer sein Feature selbst am meisten spielt, hat es sich verdient.)

Der Technical Director (abends, nach dem Job)

Meine Rolle in einem Satz: Ich bin der, der die Leitplanken setzt und die Verantwortung trägt — Architektur, Qualität, Sicherheit und die Endabnahme vor jedem Release. Was ich ausdrücklich nicht mehr bin: der, der alles tippt. Diese Verschiebung ehrlich anzunehmen war die eigentliche Umstellung des Projekts, und ich beschreibe sie Kollegen inzwischen so: Ich habe den Beruf gewechselt, ohne den Schreibtisch zu wechseln — vom Handwerker zum Bauleiter einer sehr schnellen, sehr wörtlich nehmenden Kolonne.

Der Rhythmus dazu steht im Devblog: Entwickelt wird abends und am Wochenende, seit es das Projekt gibt. Das klingt nach wenig Zeit, und das ist es auch — aber es erzwingt eine Disziplin, die größeren Projekten manchmal fehlt: Wenn pro Abend nur zwei Stunden drin sind, baut man das Wichtige zuerst. Und über tausend automatische Tests passen derweil auf, dass das Feature vom Wochenende nicht das von Dienstag zerlegt.

Die Balance (Verena)

Verenas Rolle ist die am schwersten zu beschreibende und vermutlich die wichtigste: **Sie erdet das Ganze.** Das meint zweierlei. Im Spiel: Balancing im Wortsinn — der Blick dafür, ob etwas zu leicht, zu schwer, zu frustig oder zu belanglos ist; der Einspruch, wenn Vater und Sohn

sich in einem Feature verrannt haben, das außer ihnen niemand versteht. Und um das Spiel herum: Sie ist Mitautorin des Familienblogs, auf dem wir das Projekt aus Elternsicht reflektieren — der Artikel, aus dem das Kapitelmotto stammt, trägt beide Namen. Dass dieses Projekt nie zur fixen Idee zweier Jungs wurde, sondern ein Familienprojekt blieb, mit Abendbrot-Qualitätskontrolle und Pausenerinnerung im Spiel: Das ist zu großen Teilen ihre Leistung.

Die Belegschaft ohne Gehalt

Und dann ist da die vierte Spalte des Organigramms, die es so vor wenigen Jahren nicht gegeben hätte. Wenn man unsere KIs wie Mitarbeiter vorstellt — und nach sechs Wochen Projektalltag fühlt es sich genau so an —, sieht die Belegschaft so aus:

Claude ist der Bautrupp: schreibt den Code, die Tests, die Doku, die Werkzeuge — der Kollege aus den Kapiteln 3 bis 7, der nie müde wird und jeden Morgen das Regelbuch neu liest. **ChatGPT** ist der Planer und Rechercheur: das Denk-Gegenüber der Konzeptphase (Kapitel 1 und 2), später zuständig für Technik-Recherchen und Spezialaufgaben bis hin zur DNS-Konfiguration (Kapitel 18). **Gemini** ist die Marketingabteilung: Social-Media-Strategie, Posts, Reichweiten-Ideen (Kapitel 20). **NotebookLM** ist das Lektorat (Kapitel 11). Dazu die Gewerke: die **Bild-KI** als Grafikstudio, **ElevenLabs** als Tonstudio, **Suno** als Hauskapelle. Sieben Spezialisten, Gesamtlohnkosten: Kapitel 22 verrät es, aber es ist weniger, als eine Familie für Streaming-Abos ausgeben kann.

Warum überhaupt verschiedene KIs, statt einer für alles? Ehrliche Antwort: Weil sie verschieden gut sind, in Verschiedenem — man stellt ja auch nicht den Elektriker als Maler ein. Die Aufteilung ist bei uns organisch gewachsen, nach dem simpelsten Kriterium der Welt: Wer das jeweilige Gewerk am besten erledigt hat, durfte es behalten.

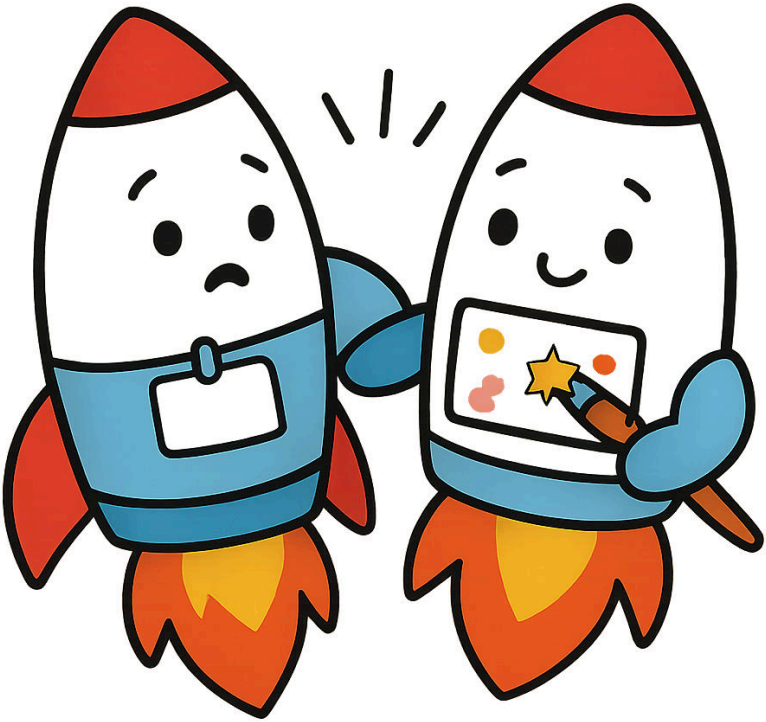
Der Satz, der alles zusammenhält

Bleibt die Frage, die uns Eltern am häufigsten gestellt wird, meist mit leicht gerunzelter Stirn: *Ist das nicht irgendwie… viel KI für ein Kind?*

Unsere Antwort steht als Motto über dem Kapitel, und sie stammt aus dem Text, den Verena und ich zusammen geschrieben haben: **KI ist kein Ersatz für Begleitung.** Justus sitzt nicht allein mit einer Maschine im Zimmer, die ihm Wünsche erfüllt. Er sitzt neben einem Elternteil, und *gemeinsam* benutzen sie Werkzeuge — so wie man gemeinsam sägt, backt oder lötet. Die KI hat in diesem Haushalt nie etwas entschieden. Sie hat ermöglicht, beschleunigt, vorgeschlagen — entschieden haben ein Kind mit Geschmack, ein Vater mit Verantwortung und eine Mutter mit Bodenhaftung.

Und wenn ich eine einzige Beobachtung aus sechs Wochen Familien-Studio weitergeben darf, dann diese: Das Wertvollste, was Justus in diesem Projekt gelernt hat, ist nicht, wie man KI benutzt. Es ist, was man von ihr *verlangen* darf (Präzision, Fleiß, Geduld), was man ihr *nicht* überlassen darf (Geschmack, Wahrheit, Verantwortung) — und dass „das hat die KI so gemacht" in diesem Haus nicht als Ausrede gilt. Wer das mit zehn verinnerlicht, ist auf die Welt, die kommt, besser vorbereitet als durch jeden Programmierkurs.

> **Was die KI hier wirklich getan hat** Alles Mögliche — aber nichts allein. Jede KI in diesem Haushalt arbeitet einem Menschen zu: Claude dem Vater, NotebookLM dem Sohn, Gemini der Reichweite, die noch kommt. Die Familie hat die Rollen verteilt, die Ergebnisse beurteilt und die eine Regel durchgehalten, an der alles hängt: Maschinen liefern. Menschen entscheiden. Ausnahmslos.



Kapitel 15 — „Es gibt schon ein Spacecraft."

Der lokale Projektordner auf Papas Rechner heißt bis heute „SpaceCraft". Man soll seine Herkunft nicht verleugnen.

Bevor dieses Buch von Servern, Releases und Weltruhm (nun ja) erzählt, muss eine kleine Geschichte erledigt werden, die sich genau an dieser Schwelle abspielte — an der Schwelle zwischen „unser Ding" und „ein Ding für andere". Es ist die Geschichte eines Namens, und sie beginnt damit, dass unser Spiel gar nicht immer *Blocks Beyond the Stars* hieß.

Es hieß **Spacecraft**.

So steht es in den Gründungsdokumenten aus Kapitel 1 und 2 — „Spacecraft – Konzeptdokument für ein 3D-Klötzchen-Weltraumspiel" ist buchstäblich die Überschrift der allerersten Zeile des Projekts. Der Name war nie eine Marketing-Entscheidung; er war einfach *da*, so selbstverständlich wie „das Spiel" — kurz, passend, klar. Von der ersten Diktier-Session an hieß alles so: die Ordner, die Dokumente, die Gespräche am Esstisch. („Spielen wir Spacecraft?" ist ein Satz, der in dieser Familie oft gefallen ist.)

Und dann kam ein Freund mit einer dieser Bemerkungen, die man erst belächelt und dann googelt: **„Du weißt schon, dass es ein Spiel gibt, das genau so heißt? Kommt bald auf Steam raus."**

Er hatte recht. Da draußen existierte bereits ein „Spacecraft" mit angekündigtem Steam-Release. Zwei Spiele mit identischem Namen im selben Genre-Umfeld — das ist für beide Seiten schlecht:

Verwechslung, Auffindbarkeit, im schlimmsten Fall Markenärger. Und es gibt in dieser Situation eine goldene Regel, die uns die Entscheidung leicht machte: Der Kleinere weicht aus, und zwar *früh*. Ein Namenswechsel vor dem ersten Release kostet ein Wochenende. Ein Namenswechsel nach zehntausend Downloads kostet die Identität.

> **Kurz erklärt: Warum man Namen prüft, bevor man sie liebt** Ein Produktname muss drei Prüfungen bestehen, und keine davon ist „gefällt er uns": Ist er **frei** (keine gleichnamigen Produkte im Umfeld, keine Marken)? Ist er **auffindbar** (kann man ihn googeln, ohne in fremden Ergebnissen zu ertrinken)? Ist er **verfügbar** (Domain, Kontonamen auf den Plattformen)? Die bittere Pointe: Je generischer und „natürlicher" ein Name wirkt — Spacecraft! —, desto sicherer ist er schon vergeben. Gute Namen sind meist die, die sich am Anfang etwas seltsam anfühlen.

Die Umbenennung

Aus Spacecraft wurde **Blocks Beyond the Stars** — ein Name, der beim ersten Hören sperriger ist und bei genauem Hinsehen alles richtig macht: Er sagt *Blöcke* (das Genre, die Minecraft-Verwandtschaft, Justus' „minecraftig"), und er sagt *jenseits der Sterne* (den Weltraum, das Entdecken). Er ist praktisch unverwechselbar, und er kürzt sich hübsch zu **BBTS**.

Dann kam der handwerkliche Teil, und der ist bei einem Software-Projekt größer, als man denkt. Das öffentliche Code-Archiv wurde umbenannt (es heißt heute `BlocksBeyondTheStars` auf GitHub), die Domain registriert, Studio-Name und Plattform-Konten angelegt. Und in unzähligen Texten musste der alte Name dem neuen weichen — eine Fleißaufgabe, für die man zum Glück einen unermüdlichen Kollegen hat, der Suchen-und-Ersetzen in Hunderten Dateien nicht langweilig findet.

Nur eine Stelle haben wir nie angefasst, und ich gestehe: inzwischen mit Absicht. **Der lokale Projektordner auf meinem Rechner heißt bis heute `SpaceCraft`.** Jeder Entwickler kennt solche Fossilien — Verzeichnisnamen, die älter sind als alle Entscheidungen, die danach kamen. Man könnte ihn umbenennen; es wäre eine Sache von Sekunden. Aber es ist, als würde man das Kinderfoto aus dem Flur abhängen. Der Ordner bleibt. Er ist die private Erinnerung daran, dass dieses große, öffentliche, ordentlich benannte Projekt mal ein Wochenend-Ding namens Spacecraft war.

Die kleine Lektion

Wenn diese Episode eine Moral hat, dann diese: **Auch im KI-Zeitalter kommen manche der wichtigsten Beiträge von Menschen, die einfach Bescheid wissen.** Keine unserer KIs hat uns je gewarnt, dass der Name kollidiert — warum auch, niemand hatte gefragt. Es brauchte einen Freund, der die Steam-Ankündigungen liest und den Mut hat zu sagen: „Du, das wird ein Problem." Der wertvollste Input dieses Kapitels kostete nichts, lief über keinen Prompt und kam beim Kaffee.

Man kann das als Fußnote lesen. Ich lese es als Vorzeichen für alles, was in den nächsten Kapiteln passiert: In dem Moment, in dem ein Projekt öffentlich wird, wird es verwundbar für Dinge, an die vorher niemand denken musste — Namen, Lizenzen, Sicherheit, fremde Erwartungen. Und der Schutz davor ist selten Technik. Es sind Menschen, die aufpassen.

> **Was die KI hier wirklich getan hat** Die Warnung kam von einem Menschen; die Entscheidung trafen Menschen. Die KI hat danach das getan, was sie am besten kann: die Umbenennung mechanisch durchgezogen — Repository, Dokumente, Texte, Konfigurationen — schnell, vollständig und ohne den einen vergessenen alten Namen, der sonst Jahre später noch irgendwo auftaucht. (Fast ohne. Der Ordner bleibt.)

Kapitel 16 — Der Tag, an dem es vier Versionen gab

---

20. Juni 2026: v0.1.0, v0.2.0, v0.3.0, v0.3.1 — vier Releases an einem einzigen Tag. Nicht aus Produktivität. Aus Sturheit.

Ein Wegweiser, bevor es losgeht: Die nächsten drei Kapitel sind die technischsten des Buches — Release-Maschinen, Open-Source-Lizenzen, Server-Flotten. Die „Kurz erklärt"-Boxen halten dich über Wasser, und es lohnt sich, denn hier wird aus dem Familienspiel ein echter Online-Dienst. Aber wenn dir gerade mehr nach Familiengeschichte ist: Kapitel 19 nimmt den Faden wieder auf, und die Technik hier verzeiht dir das Querlesen.

Drei Wochen lang war das Spiel gewachsen wie im Gewächshaus: geschützt, privat, nur für uns. Dann kam der 20. Juni 2026, und wenn man heute in unsere Versionsgeschichte schaut, sieht dieser Tag aus wie ein Feuerwerk: **vier veröffentlichte Versionen an einem einzigen Tag**, von v0.1.0 am Vormittag bis v0.3.1 am Abend.

Ich würde gern behaupten, das sei ein triumphaler Launch-Tag gewesen. Die Wahrheit ist besser, weil sie ehrlicher ist: Es war der Tag, an dem wir die **Release-Maschine** gebaut haben — und eine Release-Maschine baut man, indem man sie so lange abfeuert, bis sie trifft. Die ersten Schüsse gingen daneben. Jeder Fehlschuss bekam eine Versionsnummer.

Warum eine Maschine — und nicht einfach eine ZIP-Datei?

Halten wir kurz inne, denn hier lauert die Frage jedes vernünftigen Menschen: Warum nicht einfach das Spiel in eine ZIP-Datei packen und irgendwo hochladen? Für den ersten Tag hätte das gereicht. Aber wir wussten aus den drei Gewächshaus-Wochen zwei Dinge. Erstens: Dieses Projekt produziert in einem Höllentempo — fast dreißig Änderungen am Tag (Kapitel 4). Zweitens: Spieler haben von einem Fix nichts, solange er nicht *bei ihnen ankommt*. Ein Spiel, das sich schnell verbessert, aber langsam verteilt, ist aus Spielersicht ein Spiel, das sich langsam verbessert.

Also galt von Anfang an das Ziel, das im Devblog später so stand: **Ein Release ist ein einziger Befehl.** Alles danach macht die Maschine.

> **Kurz erklärt: Release, Build-Pipeline und Versionsnummern**
> Ein *Release* ist eine veröffentlichte, nummerierte Version eines Programms — v0.7.4 ist so eine Nummer (grob: Hauptversion.Funktionsstand.Fehlerkorrektur). Eine *Build-Pipeline* ist der Automat dahinter: Bei jedem Release baut er auf neutralen Servern das komplette Paket frisch zusammen, führt alle Tests aus und lädt die Ergebnisse an die Verteilstellen. Der Witz ist die Wiederholbarkeit — ob Version 3 oder Version 300, der Ablauf ist identisch, und niemand vergisst um Mitternacht einen Schritt, weil kein Mensch die Schritte macht.

Bei uns sieht der Ablauf heute so aus: Ich setze einen sogenannten Git-Tag — im Kern ein Etikett mit der Versionsnummer, ein einziger Befehl. Daraufhin baut GitHub Actions (der Automatendienst des Code-Archivs) das komplette Sortiment: Windows-Installer in drei Geschmacksrichtungen, Linux-Paket, Browser-Version, Server-Abbild für Docker — **sechs Artefakte aus einem Tag**. Die Versionsnummer wird überall automatisch aus dem Etikett übernommen, damit es exakt eine Quelle der Wahrheit gibt. Anschließend landet das Spiel von selbst an den Verteilstellen, und installierte Clients holen sich das Update automatisch: Wer das Spiel gestern gespielt hat, bekommt heute beim

Start die neue Version angeboten, ohne irgendetwas herunterladen zu müssen.

Am 20. Juni existierte von alledem: nichts. Und der Weg dorthin war eine Lehrstunde in einer Sorte Arbeit, die sich grundlegend anders anfühlt als Spielentwicklung — **Automatisierungs-Arbeit**, bei der der Feedback-Zyklus quälend lang ist. Man ändert eine Zeile in der Pipeline-Konfiguration, stößt den Automaten an, wartet zwanzig Minuten auf den Unity-Build … und er scheitert an Schritt 14 von 17. Nächster Versuch. Genau deshalb gab es an diesem Tag vier Versionen: Die frühen Nummern waren keine Meilensteine des Spiels, sondern **Testschüsse der Maschine** — bis am Ende des Tages zum ersten Mal die komplette Kette durchlief, vom Etikett bis zum installierbaren Spiel im Netz. Dass unser fleißigster Kollege bei dieser Fehlersuche half, versteht sich; dass er die Plattform-Eigenheiten von Build-Automaten genauso wenig auswendig kannte wie wir (Kapitel 7 lässt grüßen), auch.

Was der Release-Zug möglich machte

Ab dann aber galt: Maschine gebaut, Hebel vorhanden. Und was passiert, wenn ein Projekt mit Kapitel-4-Tempo einen Ein-Befehl-Release-Hebel bekommt, zeigt die Statistik der folgenden drei Wochen: **achtzehn Releases**. Im Schnitt alle ein, zwei Tage eine neue Version — jede mit Änderungsprotokoll, jede automatisch verteilt.

Man könnte fragen: Braucht ein Familienspiel achtzehn Releases in drei Wochen? Die Antwort wohnt in Kapitel 12 und hat einen Zettel: Als Justus' Testabende und die ersten Rückmeldungen von außen Fehler zutage förderten, war die Zeit von „gefunden" bis „bei allen Spielern behoben" oft **unter 24 Stunden**. Diese Geschwindigkeit ist keine Angeberei, sondern ein Versprechen an die Tester: Wer uns einen Fehler meldet und ihn zwei Tage später verschwunden sieht, meldet den nächsten. Wer wochenlang nichts hört, hört auf zu melden. **Die**

Release-Frequenz ist das Rückgrat der Feedback-Kultur — der Zettel funktioniert nur, weil die Maschine ihn schnell abarbeitet.

Eine Lektion aus dieser Ära müssen wir trotzdem gestehen, weil sie so schön banal ist: Mehr als einmal wunderten wir uns, warum ein längst „gefixter" Bug bei Spielern noch auftrat. Antwort: Der Fix war zwar im Code-Archiv, aber **es war noch kein Release gefolgt** — und ein Fix, der nicht released ist, existiert für die Welt nicht. Klingt selbstverständlich. Ist es auch. Wir sind trotzdem zweimal darauf hereingefallen, und seitdem steht der Satz im Projektgedächtnis, gleich neben den Shadern und den drei Blöcke hohen Türen.

Der Rauswurf des Browsers — eine Release-Ära-Geschichte

Zum Schluss dieses Kapitels eine Geschichte, die zeigt, was der Release-Zug an Mut ermöglicht. In unseren Raumstationen stehen Arcade-Automaten mit zwanzig Minispielen. Die erste Version davon war ein Trick: Die Minispiele waren kleine Web-Spiele, und das Spiel schleppte einen **kompletten eingebetteten Webbrowser** mit sich herum — ein zweites Chromium, versteckt im Bauch des Spiels, nur um sie und das Wiki anzuzeigen. Es funktionierte. Aber es machte den Download dicker, fraß Speicher und an jedem Automaten spürbar Ladezeit.

Irgendwann fiel die Entscheidung: Der Browser fliegt raus. Alle zwanzig Minispiele wurden auf eine kleine, eigene 2D-Engine *im Spiel selbst* portiert, das Wiki gleich mit — eine Menge Arbeit für etwas, das hinterher **genauso aussieht wie vorher**. Es ist die undankbarste Sorte Aufgabe im Softwarebau, und ohne elfhundert Tests und einen Release-Zug, der die Umbauten häppchenweise und rückholbar zu den Spielern bringt, hätte ich sie nie gewagt.

Der Lohn kam, wie immer in diesem Buch, beim Testabend. Justus

setzte sich an einen Automaten und sagte: **„Die gehen ja jetzt sofort auf!"** — Kein neues Feature, kein neuer Inhalt. Nur ein Spiel, das plötzlich leichter, schneller und ehrlicher ist. Manchmal ist das beste Feature das Kilo, das man dem Spiel abnimmt. Und manchmal ist die wahre Funktion einer Release-Maschine nicht, Neues schneller zu liefern — sondern einem den Mut zu geben, Altes herauszureißen.

> **Was die KI hier wirklich getan hat** Claude hat die Pipeline-Konfigurationen geschrieben (Build-Workflows für Windows/Linux/Browser/Docker, Versionsnummer aus dem Tag, spätere Deploy-Workflows) und bei der zähen Fehlersuche des 20. Juni Schritt für Schritt mitdebuggt. Der Mensch hat: entschieden, dass es die Maschine überhaupt geben muss (vor dem ersten Release!), die Fehlversuche ertragen, jeden Release ausgelöst und die Änderungsprotokolle verantwortet.

Kapitel 17 — Open Source von Anfang an — und der Sonntag, an dem es sich auszahlte

---

Jemand da draußen kann etwas besser als du — und Open Source ist die Einladung, es zu zeigen.

Von allen Entscheidungen dieses Projekts ist eine am schwersten zu erklären, wenn man nicht aus der Software-Welt kommt: **Der komplette Quellcode unseres Spiels liegt öffentlich im Internet.** Jede Zeile. Der Server, der Client, die Werkzeuge, die Fehler, die peinlichen Kommentare — alles einsehbar, kopierbar, für jeden, jederzeit, auf GitHub.

Die Frage kommt verlässlich: *Warum tut ihr das? Kann dann nicht einfach jemand euer Spiel klauen?* Dieses Kapitel ist die Antwort — und sie hat die Form zweier Geschichten, die beide damit enden, dass ein Fremder aus dem Internet unserem Familienprojekt etwas schenkt.

> **Kurz erklärt: Open Source** *Open Source* heißt: Der Quellcode — der lesbare Bauplan eines Programms — ist öffentlich, und eine Lizenz erlaubt ausdrücklich, ihn zu lesen, zu verändern und weiterzugeben. Das Gegenmodell ist „proprietär": Man bekommt nur das fertige Programm, der Bauplan bleibt Geschäftsgeheimnis. Ein riesiger Teil des Internets läuft auf Open-Source-Software, gepflegt von einer Weltgemeinschaft aus Firmen und Freiwilligen. Und ja: „Klauen" wird durch die Lizenz geregelt — dazu gleich mehr.

Warum offen — die unromantische Hälfte

Bevor die Gänsehaut-Geschichten kommen, die nüchterne Wahrheit, denn die wird oft unterschlagen: **Open Source hat unser Projekt besser gemacht, bevor der erste Fremde eine Zeile beitrug.** Öffentlicher Code erzwingt Hygiene. Passwörter und Schlüssel liegen sauber in Konfigurationsdateien statt im Code (jeder kann ja hineinschauen!). Die Dokumentation muss stimmen, weil Fremde sie lesen. Entscheidungen werden nachvollziehbar begründet, weil sie öffentlich diskutierbar sind. Man kann sagen: Wir haben uns die Disziplin eines Teams eingekauft, indem wir uns der Weltöffentlichkeit ausgesetzt haben — die übrigens zu 99,9 % nie vorbeischaut. Aber man weiß ja nie, und genau das wirkt.

Dazu kam eine Lizenz-Lektion, die wir unterwegs lernen mussten. Gestartet sind wir mit der freizügigsten aller Lizenzen (MIT: nimm alles, mach damit, was du willst). Bis uns klar wurde, was das für ein Multiplayer-Spiel bedeutet: Jemand hätte unseren Code nehmen, einen **kommerziellen Server-Dienst** daraus bauen und alle Verbesserungen für sich behalten können. Also wechselten wir zur AGPL-3.0, deren Kern sich in einem Satz erklärt: *Nimm alles, verändere alles — aber wenn du es weitergibst oder als Online-Dienst betreibst, müssen deine Änderungen ebenfalls offen sein.* Wer mit unserem Code einen Server betreibt, gibt der Gemeinschaft zurück; selbst hosten für Freunde und Familie ist ausdrücklich erwünscht und dokumentiert. Der Wechsel kostete zwei Tage (Lizenztexte, ein automatisiertes Contributor-Abkommen für Beiträge, Doku) und gab dem Projekt das Fundament, auf dem man Fremden guten Gewissens sagen kann: **Macht mit, es bleibt offen.**

Und „macht mit" meinten wir wörtlich: Im Spiel selbst gibt es einen „Mach mit"-Knopf, der erklärt, wie man beiträgt. Die Einladung ist eingebaut. Dann ging alles viel schneller, als ich zu hoffen gewagt hätte — ich hatte mich innerlich auf Monate der Stille eingestellt, so läuft das

normalerweise bei kleinen Open-Source-Projekten. Es dauerte **ein paar Tage**.

Dann kam Cora.

Geschichte eins: Der erste Beitrag von außen

Cora de la Mouche (auf GitHub: @corarona) tauchte Ende Juni in unserem Projekt auf — und brachte kein höfliches Tippfehler-Fixchen mit. Sondern etwas, das auf unserer Wunschliste weit hinten stand, weil es uns schlicht zu groß war: einen **kompletten nativen Linux-Port**. Kein Kompatibilitäts-Trick, sondern echte Linux-Unterstützung: ein eigenes Launcher-Projekt, die Build-Skripte auf Linux übertragen, die Release-Pipeline um Linux-Pakete erweitert, Dokumentation dazu. Über drei Tage zogen die Beiträge ins Projekt ein, und seither liegt jedem einzelnen Release ein Linux-Paket bei — gebaut auf dem Fundament, das Cora gelegt hat.

Ich erinnere mich an das Gefühl, als der Beitrag hereinkam, und es war nicht in erster Linie Freude über Linux. Es war dieses seltsam feierliche: *Da draußen hat ein Mensch, den wir nie getroffen haben, Abende der eigenen Lebenszeit in das Spiel unseres Kindes gesteckt — freiwillig, gekonnt, geschenkt.* Wir haben es im Release festgehalten, wie man so etwas festhält: Version 0.6.0 trägt offiziell den Beinamen **„unser erster Mitwirkender!"**. Und im Devblog steht der Satz, der seitdem unsere Kurzbegründung für Open Source ist: Wir hätten das irgendwann selbst gebaut — vermutlich schlechter und sicher später. Jemand da draußen kann etwas besser als du. Open Source ist die Einladung, es zu zeigen.

Geschichte zwei: Der Sonntag

Und dann, keine Woche später, die Geschichte, die in dieser Familie inzwischen einfach „der Sonntag" heißt.

Es begann, wie Ring drei des Marketings eben so spielt (Kapitel 20), mit einem **Kommentar auf Reddit**. Unter einem unserer Posts schrieb jemand, sinngemäß: *„Guck mal — ich hab's auf glitch.io gepackt, ihr könnt es im Browser spielen."* Man halte kurz inne: Ein Fremder hatte unser Spiel nicht nur gespielt. Er hatte es genommen, in den Browser gebracht, irgendwo lauffähig hingestellt — und uns beiläufig den Link dagelassen.

Dann kam der Teil, der mich bis heute rührt: ein **Pull Request** in unserem Repository. Darin: der fertige Browser-Port als Beitrag, WebGL plus Datenbank-Unterstützung. Und dazu eine Nachfrage von entwaffnender Höflichkeit, sinngemäß: **„Hey — es ist okay, wenn du es zum jetzigen Zeitpunkt für zu früh hältst, noch einen weiteren Client ins Projekt aufzunehmen…"** Da schenkt jemand einem fremden Familienprojekt einen kompletten Port und entschuldigt sich vorsorglich, falls das Geschenk ungelegen kommt. So sieht Open-Source-Kultur aus, wenn sie gut ist.

Wer *war* das überhaupt? Das herauszufinden kostete mich, passend zu diesem Buch, erst einmal **eine Runde ChatGPT-Recherche** zum Commit-Autor. Ergebnis: Der Name hinter dem Beitrag war **Devin Dixon** — ein **CEO aus den USA mit einer eigenen Spieleplattform**. Ein Firmenchef hatte an einem Sonntagnachmittag (seiner Zeitzone) seine Freizeit genommen, um das Spiel eines Zehnjährigen in den Browser zu bringen. Ich habe ihm auf Reddit geschrieben und mich bedankt; mehr Gegenleistung hat er nie verlangt.

Und Justus? **Total aus dem Häuschen.** Und ich bitte dich, kurz die Kinderperspektive einzunehmen, denn sie ist der Kern der Geschichte, und Justus hat sie selbst am besten formuliert: *Jemand aus Amerika, den wir nicht kennen — und der in seiner Freizeit uns hilft, das Spiel besser zu machen! Wahnsinn!* Man kann Selbstwirksamkeit nicht besser illustrieren: Die Idee eines Zehnjährigen war gut genug, dass ein Erwachsener auf einem anderen Kontinent unaufgefordert daran mitbaut. Am Wochenende.

Aus dem Sonntags-Port wurde keine Randnotiz: Die Browser-Version ist heute offizieller Teil des Projekts — „im Browser spielen" auf unserer Website führt die Ahnenreihe dieses Pull Requests fort (was wir daraus bauten, erzählt Kapitel 18). Und der „weitere Client", für den es angeblich vielleicht zu früh war? Er war exakt rechtzeitig.

Die Contributor-Liste als Familienfoto

Wer heute die Beitragenden-Liste unseres Projekts ansieht, findet dort neben Cora, Devin und einem freundlichen Menschen namens Maqbool Ahmed, der unsere deutschen Übersetzungen entstaubte, mehrfach: mich — unter anderem als **ai_cat_coder**. Falls du hinter diesem Namen eine geheimnisvolle KI-Identität vermutest: schön wär's, die Auflösung ist profaner und mir lieber. Ich liebe Katzen. Wir haben zu Hause zwei Kater. Daher der Name.

Und in dieser Profanität steckt die eigentliche Pointe, mit der dieses Kapitel enden soll: **Die KI taucht in der Contributor-Liste überhaupt nicht auf.** Abertausende von ihr geschriebene Zeilen zogen ins Projekt ein — aber immer unter *meinem* Namen, mit meiner Abnahme, auf meine Verantwortung. So gehört es sich, und zwar nicht aus Bescheidenheit der Maschine, sondern aus Rechenschaft des Menschen: In dieser Liste stehen die, die für ihre Beiträge geradestehen. Menschen aus mehreren Ländern, eine Familie, zwei Kater als Namenspaten — alle am selben Spiel. Ich wüsste kein besseres Familienfoto für dieses Projekt.

> **Was die KI hier wirklich getan hat** Wenig — und das ist die Pointe dieses Kapitels. Die Lizenz-Entscheidung, die Einladung, das Vertrauen: menschlich. Die Geschenke von Cora und Devin: menschlich, im besten Sinne. Die KI hat die Beiträge beim Einzug geprüft und integriert (fremder Code läuft bei uns durch dieselben Tests wie eigener — Kapitel 6 gilt für alle) und den Lizenzwechsel mechanisch umgesetzt. Aber dieses Kapitel handelt von der einen

> Ressource, die kein Modell generiert: Menschen, die freiwillig mitbauen.

Kapitel 18 — Ein kleiner Server für eine kleine Flotte

Fünf Euro im Monat. Dafür bekommt man: ein Portal, offizielle Welten, eine Container-Flotte, ein Monitoring mit 144 Alarmen — und die Verantwortung, dass das alles läuft, wenn fremde Kinder spielen wollen.

Bis zu diesem Punkt der Geschichte lief unser Spiel auf Rechnern, die uns gehörten. Wer mitspielen wollte, brauchte jemanden im Haus, der einen Server startet. Das ist für ein Familienspiel völlig in Ordnung — und es ist der Moment, an dem die meisten Hobby-Projekte vernünftigerweise stehen bleiben. Denn der nächste Schritt ändert die Natur des Projekts: **ein Dienst im Internet, der läuft, wenn man selbst schläft.** Dieses Kapitel erzählt, wie ein Familienprojekt diesen Schritt gegangen ist — mit einem gemieteten Fünf-Euro-Server, einer KI als Systemadministrator und ein paar Prinzipien, die sich bewährt haben.

Die Miete: ein VPS

> **Kurz erklärt: Server, VPS und „die Cloud"** Ein *Server* ist einfach ein Computer, der dauernd läuft und über das Internet erreichbar ist. Man kann sich einen kompletten mieten — oder einen *VPS* (virtueller privater Server): ein abgetrennter Anteil an einem großen Rechner im Rechenzentrum, mit eigenen Ressourcen und voller Kontrolle, ab wenigen Euro im Monat. „Die Cloud" der großen Anbieter ist dasselbe Prinzip mit mehr Komfort, mehr Magie — und Rechnungen, die mitwachsen. Für uns galt die Raspberry-Pi-Philosophie aus Kapitel 2: klein anfangen, Grenzen

> setzen, messen.

Unsere gesamte Online-Welt — das Spielerportal, die offiziellen Welten, alle von Spielern erstellten Welten — läuft auf **einem einzigen kleinen Linux-VPS** für rund fünf Euro im Monat. Kein Rechenzentrum, keine fünfstellige Cloud-Rechnung. Dass das geht, ist keine Zauberei, sondern die späte Dividende einer alten Entscheidung: Ein Server, der laut Spezifikation auf einem Bastelrechner laufen muss, langweilt sich auf einem echten VPS.

Eingerichtet wurde die Maschine, wie inzwischen alles in diesem Buch: im Dialog mit der KI. Claude konfigurierte die Grundsicherung, bevor das Spiel auch nur in die Nähe des Servers kam — eine Firewall, die nur die nötigen Türen öffnet; fail2ban, das automatisierte Einbruchsversuche aussperrt; Zugang nur mit kryptografischem Schlüssel statt Passwort. Für mich fühlte sich diese Phase an wie der Umgang mit einem sehr erfahrenen Admin-Kollegen, der jede Konfigurationsdatei auswendig kennt, aber grundsätzlich fragt, bevor er etwas Gefährliches tut — weil wir ihm genau das als Regel mitgegeben hatten.

Nur ein Gewerk lief traditionsgemäß über ChatGPT: die **DNS-Einrichtung** — jene Zuordnung, die aus „blocksbeyondthestars.com" die Adresse unseres Servers macht, verwaltet bei Cloudflare.

> **Kurz erklärt: DNS — das Telefonbuch des Internets** Computer finden einander über Zahlenadressen (IP-Adressen). *DNS* ist das weltweite Telefonbuch, das Namen wie blocksbeyondthestars.com in diese Zahlen übersetzt. Wer eine Domain besitzt, pflegt ihre Telefonbucheinträge selbst — und ein falscher Eintrag heißt: Die Welt findet deine Website nicht mehr, obwohl sie läuft. Deshalb der Respekt, mit dem in diesem Haus über DNS gesprochen wird.

Die Flotte: jede Welt ihr eigener Käfig

Das Herzstück des Dienstes ist ein kleines Programm, das wir WorldHost nennen — und ein Prinzip, das sich in einem Satz erklärt: **Jede Welt läuft in ihrem eigenen Container, und wer nicht gespielt wird, schläft.**

> **Kurz erklärt: Container (Docker)** Ein *Container* ist eine Leichtbau-Verpackung für ein Programm: Es läuft darin isoliert, mit eigenen zugeteilten Ressourcen, als wäre es allein auf der Maschine — nur dass Dutzende Container nebeneinander existieren können. *Docker* ist das verbreitetste Werkzeug dafür. Der Charme für uns: Man kann jedem Container harte Grenzen setzen. Ein Amok laufendes Programm im Container bleibt ein Problem *im Container*.

Konkret: Erstellt ein Spieler im Portal eine Welt, bekommt sie ihren eigenen Docker-Container — mit festem Arbeitsspeicher-Limit, begrenzten CPU-Anteilen und eigenem Spielstand. Maximal zehn Welten laufen gleichzeitig; **eine einzelne Welt kann den Server niemals lahmlegen.** Und das Beste: Verlassen alle Spieler eine Welt, wird der Spielstand gesichert und der Container gestoppt — die Welt schläft und kostet nichts. Klickt Wochen später jemand auf „Spielen", prüft der WorldHost erst die Berechtigung — bei geschützten Welten das Passwort, wohlgemerkt **vor** dem Aufwecken, sonst könnte jeder durch bloße Join-Versuche fremde Welten hochfahren. Dann startet er den Container, und im Portal erscheint für ein paar Sekunden: „Welt wird geweckt…" Ich finde diese Formulierung bis heute schön. Sie ist technisch exakt und klingt trotzdem wie aus einem Kinderbuch.

Das Ganze ist bewusst klein gebaut — kein Kubernetes, keine Cloud-Orchestrierung, keine Magie: ein Server, Docker, ein Dienst, der Container startet und stoppt, davor ein Caddy-Proxy, der sich um Verschlüsselungszertifikate automatisch kümmert. Im Devblog steht

dazu der Satz, der unsere ganze Infrastruktur-Philosophie trägt: *Für ein Familienprojekt reicht das erstaunlich weit — wenn man Grenzen setzt und hinschaut.*

Das Hinschauen: Monitoring, das aufs Handy ruft

„Hinschauen" ist dabei wörtlich automatisiert. Auf dem Server läuft Netdata, ein Überwachungswerkzeug, bei uns mit **über 140 eingerichteten Alarmen** — von „Festplatte läuft voll" über „Portal antwortet nicht" bis zu Feinheiten, an die man erst denkt, wenn sie einmal zugeschlagen haben. Schlägt ein Alarm an, erscheint eine Push-Nachricht auf meinem Handy. Das Ziel ist bescheiden und unbezahlbar zugleich: **von Problemen vor den Spielern erfahren.** Ein Familienvater kann keinen 24/7-Bereitschaftsdienst leisten — aber er kann dafür sorgen, dass das Klingeln ihn findet, nicht die Beschwerde.

Und weil Messen bekanntlich Meinungen ruiniert, die schönste Erkenntnis aus dem Betrieb: Als wir im Leerlauf nachmaßen, ging **rund die Hälfte des belegten Arbeitsspeichers gar nicht ans Spiel** — sondern an Docker und das Monitoring selbst. Die Infrastruktur, die das Spiel trägt, ist im Ruhezustand schwerer als das Spiel. Es gibt keine bessere Miniatur für das, was Betrieb bedeutet: Der Wirt isst mit.

Wartung mit Anstand — und der Ernstfall Verantwortung

Ein Dienst, der läuft, will aktualisiert werden — und irgendwo da draußen steht dann gerade ein Kind kurz vor seiner ersten Diamant-Ader. Deshalb war uns ein Feature wichtig, das man im Idealfall nie bemerkt: **Wartung ohne Rauswurf.** Steht ein Update an, schickt der Server jeder laufenden Welt eine Ankündigung; im Spiel erscheint ein Banner mit Countdown („Server-Neustart in 10 Minuten"), das man aktiv wegklicken muss — aber erst nach ein paar Sekunden, damit es

niemand im Kampfgetümmel versehentlich wegdrückt. Läuft der Countdown ab, fährt die Welt geordnet herunter: speichern, die Spieler mit einem *erklärenden* Abschiedsbildschirm verabschieden statt mit einem kryptischen „Verbindung verloren", Container stoppen. Beim nächsten Join wacht die Welt mit neuer Version auf, der Spielstand ist da.

Der unerwartete Nebeneffekt steht im Devblog und verdient Fettdruck, weil er ein allgemeines Gesetz ist: **Seit Updates entspannt sind, machen wir sie öfter.** Gute Wartungswerkzeuge verbessern nicht die Wartung — sie verbessern das ganze Projekt.

Bleibt die ehrliche Schlussfrage dieses Kapitels: Warum tut man sich das an? Der Betrieb ist der Teil des Projekts, der nie fertig ist — kein Feature, das man abschließt, sondern ein Zustand, den man hält. Die Antwort ist dieselbe wie überall in diesem Buch, nur eine Stufe ernster: Weil am anderen Ende Kinder sitzen. Wenn Justus' Welt „einfach weg" wäre, würde ich es reparieren, und er wüsste, warum. Wenn die Welt eines fremden Kindes weg ist, gibt es nur uns — und deshalb die Grenzen, die Alarme, die Abschiedsbildschirme. Verantwortung ist, was von der Begeisterung übrig bleibt, wenn man sie dauerhaft macht.

> **Was die KI hier wirklich getan hat** Claude hat den VPS gehärtet und konfiguriert (Firewall, fail2ban, Caddy), den WorldHost samt Container-Fences gebaut, das Monitoring mit seinen Alarmen aufgesetzt und das Wartungssystem implementiert; ChatGPT hat die DNS-Einträge bei Cloudflare mit konfiguriert. Der Mensch hat: entschieden, dass es den Dienst geben soll (die eigentliche Schwelle!), jede Grenze festgelegt (RAM, CPU, Weltenzahl), die Alarme aufs eigene Handy geleitet — und trägt das, was keine KI übernehmen kann: die Rufbereitschaft.

Kapitel 19 — Kindgerecht by Design

Man baut anders, wenn der wichtigste Nutzer am selben Küchentisch sitzt. Ich glaube inzwischen: Man baut besser.

Es gibt in unserem Spiel eine Kategorie von Features, die in keinem Trailer vorkommt, die kein Devblog-Feuerwerk auslöst und die trotzdem mehr Entwicklungsabende bekommen hat als manches Lieblings-Feature. Man erkennt sie an einem gemeinsamen Ursprung: Sie alle beginnen mit demselben Halbsatz. **Wir versuchen, das Spiel kindgerecht zu machen — daher gibt es…**

Dieser Halbsatz ist das Leitmotiv dieses Kapitels, und ich buchstabiere ihn bewusst als Liste aus, denn genau so ist er im Projekt gewachsen: als Kette von Konsequenzen aus einer einzigen Grundentscheidung. Ich entwickle ein Onlinespiel, und mein Sohn spielt es — ich sitze auf beiden Seiten des Tisches, als Entwickler, der Features will, und als Vater, der weiß, was im Internet unterwegs ist. Aus dieser Doppelrolle folgt alles Weitere.

Daher gibt es: das Baumhaus-Prinzip

Wer von anderen Onlinespielen kommt, stutzt bei uns zuerst hier: Es gibt keine offene Serverliste, keinen „Beliebige Welt beitreten"-Knopf, kein Matchmaking mit Fremden. Multiplayer funktioniert ausschließlich mit **Passwort und Weitersagen** — und das ist kein fehlendes Feature, sondern das Fundament. Wir nennen es intern das **Baumhaus-Prinzip**: In ein Baumhaus kommt rein, wen du persönlich einlädst. Eine Welt zu teilen heißt bei uns: erstellen, Passwort vergeben, weitersagen — auf

dem Schulhof, in der Familiengruppe, auf dem Sofa. Selbst Welten, die im öffentlichen Browser auftauchen, gibt es *nur mit Passwort*; wer das Passwort entfernt, fliegt automatisch aus der Liste. Es existiert schlicht kein Weg, auf dem ein Fremder unangekündigt in der Welt eines Kindes landet.

Und davon gibt es **keine Ausnahme** — auch nicht bei den offiziellen Welten, die wir selbst betreiben. Einen dauerhaft betreuten öffentlichen Spielplatz können wir als Familienprojekt nicht ehrlich versprechen, also versprechen wir ihn nicht. Der Preis ist uns bewusst: kein „mal eben mit tausend Leuten zocken", langsameres Wachstum. Aber Baumhäuser sind nicht für tausend Leute gebaut. Sondern für die richtigen.

Daher gibt es: F1, /report und die Trennung von Feedback und Beschwerde

Zweite Konsequenz: **Melden muss so einfach sein wie das, wovor es schützt.** Ein Kind, das etwas Komisches erlebt, darf keine Beweisführung schulden. Also: Im Spiel genügt die Taste **F1**, um uns direkt Feedback zu schicken — der Zettel aus Kapitel 12, eingebaut als Knopf. Für ernste Fälle gibt es den /**report-Befehl** im Chat, und der nimmt dem Kind die schwerste Last ab: Er zitiert automatisch die letzten Chat-Zeilen und den Welt-Kontext mit. Wer meldet, muss nichts abtippen, nichts belegen, nichts können — tippen, fertig, ein Erwachsener schaut drauf.

Genauso wichtig war uns die Trennung der Tonlagen: **Feedback ist keine Beschwerde.** Im Portal sind „Feedback & Ideen" und „Spieler melden" bewusst getrennte Wege mit eigener, freundlicher Sprache — wer eine Idee loswerden will, soll kein Formular ausfüllen, das sich nach Anzeige erstatten anfühlt. Und für Eltern gibt es eigene Hinweisabschnitte direkt auf der Startseite und bei den Regeln, damit man nicht erst suchen muss, wem hier ein Kind anvertraut wird.

Daher ist: der Funk optional

Dritte Konsequenz, und sie hat uns ein technisch stolzes Feature relativiert: Unser Spiel hat einen eingebauten **Voice-Chat** — den „Funk", mit Reichweiten-Stufen, wie es sich für ein Weltraumspiel gehört. Und er ist **optional**. Sprechen-mit-Fremden ist für Kinder die heikelste Funktion überhaupt; also ist die Grundhaltung nicht „an, weil wir's gebaut haben", sondern: abschaltbar ohne Nachteil. In einer Baumhaus-Welt unter Freunden ist der Funk großartig. Überall sonst gilt: Was aus bleibt, kann nicht schiefgehen. Es ist dieselbe Logik wie beim Sprachmodell in Kapitel 10 — die sicherste Tür ist die, die es nicht gibt — nur diesmal auf Menschen angewandt statt auf Maschinen.

Daher gibt es: Konten ohne E-Mail — und die lästige Nebenpflicht

Vierte Konsequenz, die datenschützerische: Für ein Spielerkonto braucht es bei uns **keine E-Mail-Adresse**. Ein Kind soll kein Datenprofil hinterlassen, nur um Blöcke zu stapeln. Was wir nicht erheben, kann niemandem wehtun — nicht bei einem Leck, nicht bei einer Übernahme, nicht durch uns selbst. (Die ehrliche Kehrseite kommunizieren wir offen: „Passwort vergessen" per Mail gibt es dann eben auch nicht.) Dazu kommen die unsichtbaren Handwerksdetails, die niemand bemerkt — und die genau deshalb stimmen müssen. Welt-Passwörter werden nie im Klartext gespeichert, sondern als kryptografischer Hash; selbst wir können sie nicht lesen. Nach zehn Fehlversuchen gibt es eine Abkühlpause gegen Durchprobieren. Und Absturzberichte laufen durch einen Filter, der persönliche Daten entfernt, bevor irgendetwas hochgeladen wird.

Und dann ist da noch die Pflicht, die in dieser Aufzählung die unglamouröseste ist und über die ich beim Schreiben ehrlich geseufzt habe: **Es gibt eine Datenschutzerklärung.** Und ein Impressum. Richtige, juristisch ordentliche, auf der Website verlinkt. Die „lästige

Nebenpflicht", wie sie bei uns intern heißt — niemand baut ein Weltraumspiel, weil er Rechtstexte liebt. Aber ein Dienst, den fremde Familien nutzen, hat diese Texte zu haben, Punkt. (Immerhin: Auch Rechtstexte tippen sich mit KI-Unterstützung erträglicher. Verantwortlich bleibt man trotzdem selbst — das Muster des Buches, ausnahmslos auch hier.)

Und daher gibt es sogar: die Pausen-Erinnerung und die Signatur

Zwei letzte Konsequenzen, eine zum Schmunzeln, eine ganz nüchtern. Die zum Schmunzeln: Das Spiel zeigt die Sitzungsdauer an und **erinnert freundlich an Pausen**. Ob ich das als Entwickler gern eingebaut habe? Sagen wir so: Der Vater hat den Entwickler überstimmt. (Justus' Meinung dazu ist aktenkundig und fällt anders aus.)

Die nüchterne: Seit Kurzem sind unsere Windows-Versionen **code-signiert** — digital unterschrieben, sodass Betriebssystem und Virenscanner die Herkunft prüfen können. Möglich macht das ein Programm, das Open-Source-Projekten genau dies kostenlos anbietet. Klingt nach einer Techniker-Fußnote, gehört aber genau in dieses Kapitel: Eltern, die ihrem Kind ein Spiel aus dem Internet installieren, verdienen ein Betriebssystem, das dabei nicht warnt. Vertrauen ist bei einem Kinderspiel kein Feature unter vielen. Es ist das Produkt.

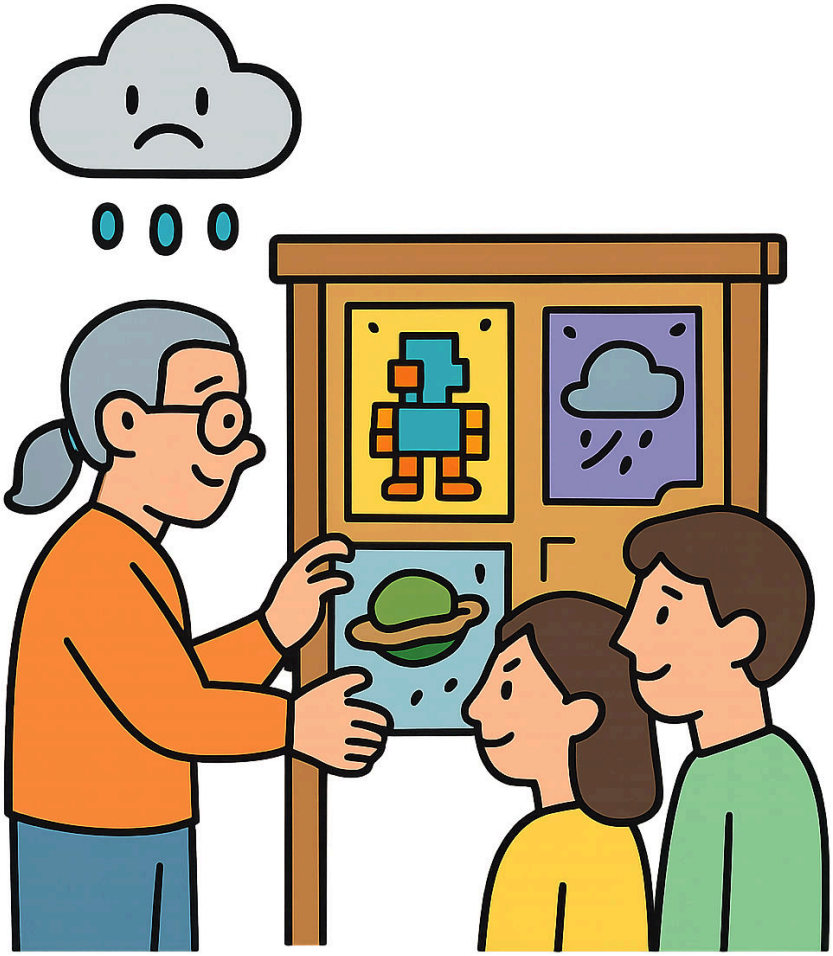
Die Bilanz des Prinzips

Zum Schluss die Beobachtung, die mich an diesem Themenfeld am meisten überrascht hat, und sie steht schon im Familienblog: **Keine dieser Entscheidungen hat das Spiel schlechter gemacht.** Die Passwort-Regel macht Welten persönlicher. Die schlanken Konten ersparen uns Datenschutz-Kopfschmerzen. Die Melde-Wege machen

die Community freundlicher, bevor sie groß ist. Und die Pausen-Erinnerung … erinnert gelegentlich auch den Entwickler.

Man baut anders, wenn der wichtigste Nutzer am selben Küchentisch sitzt. Man baut besser. Und — das ist die stille These dieses Kapitels — man baut *übertragbar* besser: Nichts an diesem Prinzip setzt ein eigenes Kind voraus. Es setzt nur voraus, dass man seine Nutzer so behandelt, als säßen sie am eigenen Küchentisch.

> **Was die KI hier wirklich getan hat** Implementiert: alles — Passwort-Hashing, Rate-Limits, /report samt Chat-Kontext, Portal-Texte, PII-Filter, die Wartungs- und Signatur-Integration; sogar Entwürfe der Rechtstexte. Entschieden: nichts. Jede „daher gibt es"-Regel dieses Kapitels ist eine Elternentscheidung, die einer Maschine als Anforderung übergeben wurde. Kindgerechtigkeit kann man nicht generieren. Man kann sie nur bestellen — und dann sehr gut bauen lassen.



Kapitel 20 — Marketing mit drei KIs

Ein Spiel, das niemand findet, ist ein Tagebuch mit Grafik.

Irgendwann steht jedes Projekt, das öffentlich sein will, vor der Erkenntnis, die Entwickler am wenigsten mögen: **Bauen ist die halbe Arbeit. Die andere Hälfte ist, gefunden zu werden.** Und während sich für das Bauen inzwischen jeder vorstellen kann, dass KI hilft, ist die zweite Hälfte der stillere Teil dieser Geschichte — dabei lief sie bei uns genauso konsequent KI-gestützt. Nur eben mit anderen Kollegen.

Dieses Kapitel ist die Tour durch unsere Reichweiten-Werkstatt: eine Heimatbasis, ein automatisierter Blog, eine KI als Social-Media-Strategin, eine Kampagne auf fremden Listen — und eine ehrliche Niederlage.

Die Heimatbasis: eine eigene Website

Die erste Entscheidung war altmodisch und ich verteidige sie gegen jeden Trend: **Das Zuhause des Spiels ist eine eigene Website** — blocksbeyondthestars.com —, nicht ein Profil auf einer Plattform. Plattformen ändern Regeln, Algorithmen und Gebühren; eine eigene Domain gehört einem. Auf der Website wohnt alles, was jemand braucht, der von dem Spiel hört: das Spiel selbst (Download für Windows und Linux, „Im Browser spielen" mit einem Klick), der DevBlog, die Regeln, der Eltern-Hinweis. Und eine Seite, die unser eigentliches Projektziel trägt: **„Mitmachen."** Tester, Entwickler, Kreative — „Lass uns das Spiel zusammen großartig machen!" steht dort. Es ist wörtlich gemeint; das Ende des Buches gehört diesem Satz.

Gebaut ist die Seite auf einem Baukasten-System (Wix) — bewusst:

Die Ingenieurs-Energie dieses Projekts gehört dem Spiel, nicht dem Website-Unterbau. Aber auch dieser vermeintlich handgemachte Teil des Projekts ist in Wahrheit ein KI-Staffellauf, und die Aufstellung ist inzwischen vertraut: **Geplant** habe ich die Website zuerst mit ChatGPT — Seitenstruktur, was gehört auf eine Spiele-Website für Familien, in welcher Reihenfolge, was ins Menü. **Getextet** hat Gemini — die Marketingabteilung durfte zeigen, was sie kann. Und **übersetzt** wurde die fertige Seite von der Übersetzungs-KI, die in Wix selbst eingebaut ist — die englische Fassung der Website stammt also von einer vierten Maschine, die in diesem Buch bisher gar nicht vorkam. Drei KIs für eine Baukasten-Website; kuratiert, zusammengefügt und freigegeben, wie immer, von einem Menschen am Küchentisch.

Ein Detail der Website ist darüber hinaus echte Ingenieursarbeit, und es ist mein Lieblingsbeispiel dafür, wie weit „konsequent KI" bei uns geht.

Und weil dieses Buch Transparenz predigt, gehört hier eine Anekdote hinein, die ich genauso gut verschweigen könnte — was exakt der Grund ist, sie zu erzählen. Der Blog ging erst online, als das Projekt schon Wochen alt war; die Artikel über die frühen Ereignisse entstanden rückblickend. Und damit die Blog-Chronik sich „natürlich" liest, haben wir die Veröffentlichungsdaten der nachgetragenen Artikel **per Skript rückdatiert** — jeder Post bekam ein Datum neben dem Release, von dem er handelt. Auch dieses Skript schrieb natürlich die KI; selbst unsere Geschichtsklitterung ist automatisiert. Ist das schlimm? Die Texte sind wahr, die Ereignisse belegt, und Blogs dürfen Chroniken sein. Aber es heißt eben auch: **Publikationsdaten im Netz sind Kulisse** — bei uns und überall sonst. Die echte Chronologie dieses Projekts steht nicht im Blog, sondern in der Git-Historie, und dieses Buch hält sich ausschließlich an die. (Du kannst das nachprüfen. Alles ist öffentlich. Das ist der Punkt.)

Der Blog, der sich selbst veröffentlicht

Unser DevBlog — jene Artikel über Shader-Pannen, Baumhaus-Prinzip und Fabrik-Imperien, aus denen dieses Buch reichlich zitiert — erscheint **zweisprachig**, Deutsch und Englisch, inzwischen Dutzende Artikel. Wer je einen Blog gepflegt hat, ahnt die Fleißarbeit dahinter: Text einpflegen, formatieren, Übersetzung einpflegen, verlinken, kategorisieren — pro Artikel, pro Sprache.

Bei uns geht das so: Die Artikel werden als einfache Textdateien geschrieben (natürlich mit KI-Unterstützung, redigiert von mir). Dann übernimmt ein **selbst gebautes Python-Skript**: Es konvertiert die Texte in das Format des Website-Systems, legt sie über dessen Programmierschnittstelle als Blog-Beiträge an, verknüpft die deutsche und die englische Fassung miteinander und sortiert die Kategorien. **Selbst das Bloggen wurde am Ende automatisiert** — geschrieben hat das Skript, du ahnst es, Claude. Es ist dieselbe Fließband-Logik wie bei den Texturen in Kapitel 8, nur dass am Ende keine Kachel herauskommt, sondern ein veröffentlichter Artikel. Und es ist ein hübscher Kreisschluss: Die Werkzeuge, mit denen dieses Projekt *gebaut* wurde, sind dieselben, mit denen es *erzählt* wird.

Von innen nach außen: die Kanal-Chronik

Wie verbreitet man so ein Spiel nun tatsächlich? Nicht mit einem großen Launch-Tag, sondern — und ich glaube inzwischen, das ist die richtige Reihenfolge für jedes kleine Projekt — **in Ringen, von innen nach außen**, immer erst dorthin, wo einen jemand kennt.

Ring eins: WhatsApp. Kaum stand die erste Version der Website, ging die Nachricht dorthin, wo Marketing noch Vertrauen heißt: **Familie und Freundeskreis**, per Messenger. „Schaut mal, was Justus und ich gebaut haben" — mit Link. Das ist die ehrlichste Zielgruppe der Welt: Sie klickt aus Zuneigung und urteilt trotzdem schonungslos (Verwandte sind da erstaunlich nah am Reddit-Niveau, nur freundlicher formuliert).

Ring zwei: die Arbeit. Dann gezielte Werbung bei meinen **Arbeitskollegen** — Menschen, die selbst Software bauen und wissen, was sie da vor sich haben. Aus diesem Ring stammt übrigens der ergiebigste Einzeltester der Projektgeschichte: Bastian und sein Erste-Stunde-Bericht aus Kapitel 12. Kollegen sind als Zielgruppe doppelt wertvoll — sie spielen nicht nur, sie verstehen auch, *was* sie loben oder zerpflücken.

Ring drei: das offene Internet. Und schließlich Reddit — wovon der nächste Abschnitt handelt, und man ahnt am Tonfall bereits: Der Übergang von Ring zwei zu Ring drei ist der Moment, in dem das Wasser kälter wird.

Dazu kamen, parallel, die Streuversuche in alle Richtungen, und ich liste sie ehrlich mit Ergebnis: **YouTuber angeschrieben** — verschiedene, mit persönlicher Nachricht; Rückmeldung bisher: keine. (Das ist normal und trotzdem ernüchternd; Content-Creator ertrinken in solchen Anfragen. Es steht hier, damit niemand glaubt, bei uns hätte alles funktioniert.) **LinkedIn** — ja, wirklich: Das Vater-Sohn-KI-Projekt ist dort erstaunlich anschlussfähig, weil es beruflich lesbar ist; das Karriere-Netzwerk war interessierter als manches Spiele-Forum. Und **Facebook**, vor allem ein Post in einer **lokalen Gruppe hier in Trier** — der Charme des Lokalen: Dort ist man nicht „ein KI-Projekt", sondern „die Familie von hier, die ein Spiel gebaut hat". Lokal schlägt viral, wenn man klein ist.

Die Github-Projektlisten — der planvollste dieser Kanäle — bekommen unten einen eigenen Abschnitt. Aber schon hier zeichnet sich das Muster ab, das dieses Kapitel trägt: **Je näher der Kanal an echten Beziehungen war, desto besser funktionierte er.** WhatsApp schlug LinkedIn, LinkedIn schlug Reddit, und die lokale Facebook-Gruppe schlug vermutlich alles zusammen, gemessen an Herzlichkeit pro Klick.

Die Strategin: Gemini

Für die Frage, *wo* man über das Spiel redet, holten wir die dritte große KI ins Team: **Gemini** wurde unsere Marketingabteilung. Das Aufgabenprofil: eine Social-Media-Strategie entwickeln — welche Plattformen lohnen sich für ein Familien-Voxelspiel, **welche Subreddits** (Themenforen auf Reddit) sind die richtigen und welche Tonlage erwarten sie, was macht dort einen guten Post aus, welche Ideen erzeugen Reichweite jenseits des Immergleichen. Dazu das Formulieren konkreter Posts.

Warum ausgerechnet dafür eine eigene KI? Aus demselben Grund, aus dem man den Elektriker nicht streichen lässt: Marketing-Texte sind ein eigenes Handwerk mit eigenen Gesetzen — Aufhänger, Länge, Community-Etikette; ein Reddit-Post, der nach Werbung riecht, ist tot, bevor er unten angekommen ist. Eine KI, die man gezielt als Strategin brieft, denkt an Dinge, an die ein Entwickler-Hirn abends um elf nicht denkt. An die Selbstdarstellungsregeln einzelner Subreddits. An den Unterschied zwischen „zeig dein Projekt"-Donnerstagen und normalen Tagen. Und an die Frage, welche Geschichte man eigentlich *führt* — bei uns naheliegend: nicht „neues Voxelspiel", sondern *Vater und Sohn bauen mit KI ein Spiel*.

Die kalte Dusche: „KI" ist auf Reddit ein Reizwort

Bevor dieses Kapitel zu den leiseren Erfolgen kommt, gehört eine Erfahrung hierher, die uns kalt erwischt hat — und die jeder kennen sollte, der 2026 mit einem KI-Projekt an die Öffentlichkeit geht: **In vielen Online-Communities, allen voran auf Reddit, ist „KI" kein Verkaufsargument. Es ist ein Feindbild.**

Die Ironie unserer Lage: Ausgerechnet das Ehrlichste an unserem Projekt — dass wir offen sagen, wie es gebaut wurde — ist dort das größte Handicap. Ein Post, der „mit KI entwickelt" auch nur erwähnt, zieht in vielen Foren reflexhafte, teils **toxische Ablehnung** an. Und weil das abstrakt klingt, hier unsere zwei realen Erfahrungen,

nebeneinandergelegt:

Erfahrung eins, die gute: Im Subreddit **r/aigamesdev** — einer Community, die sich ausdrücklich mit KI-gestützter Spieleentwicklung beschäftigt — bekam das Projekt **viel positives Feedback**. Interessierte Nachfragen, Ermutigung, echter Austausch. Man ahnt schon: Wir hatten die passende Blase getroffen.

Erfahrung zwei, die lehrreiche: Aus dem Subreddit **r/VoxelGameDev** bin ich **hochkant rausgeflogen**. Ich hatte übersehen, dass man dort *überhaupt nichts* posten darf, was auch nur mit KI zu tun hat — ein Voxelspiel, quelloffen, komplett einsehbar, AGPL-lizenziert, exakt die Sorte Projekt, für die das Forum existiert: gelöscht, wegen des Werkzeugs, mit dem es gebaut wurde. Die Ironie muss man sich einmal vor Augen führen: Ausgerechnet ein Forum von Spieleentwicklern verbannt jedes Projekt, das dazu steht — in einer Branche, deren größte Player — Microsoft & Co. — massiv Open Source bereitstellen *und* längst massiv mit KI coden (wie praktisch die gesamte Softwareindustrie).

Aus derselben Ecke kam auch der giftigste Kommentar der Projektgeschichte: Ich würde **„die Arbeit von Menschen herabsetzen"**, wenn ich hier nach Hilfe für ein KI-Projekt fragte — und ob ich denn bereit wäre, **für die Arbeitszeit zu bezahlen**. Meine Rückfrage, ob der Verfasser selbst denn jemals einen Open-Source-Entwickler für Open-Source-Software bezahlt habe, blieb — Überraschung — **unbeantwortet**. Ich erzähle das ohne Groll, aber mit der Beobachtung, die daraus folgt: Die Empörung war keine Position, die einer Nachfrage standhielt. Sie war ein Reflex, und Reflexe diskutieren nicht.

Die feinste Ironie dieser Episode liefert allerdings eine Maschine. Meine Marketingabteilung — Gemini — hatte mir zu genau solchen Kommentaren längst geraten, sinngemäß: **„Lass es. Das kostet dich nur unnütz Energie. Steck deine Zeit lieber in die positiven**

Aspekte der Kommunikation und fokussiere dich auf das Projekt." Ich habe nicht gleich darauf gehört — der Drang, auf Unfaires zu antworten, ist menschlich und genau deshalb so teuer. Inzwischen beherzige ich es: Es ist die eine Stelle im Buch, an der die KI der besonnenere Teil des Teams war.

Die Lehre dahinter ist strukturell, und dann lassen wir das Thema auch ruhen: **Reddit ist keine Öffentlichkeit, sondern ein Archipel aus Blasen.** Für jede Meinung gibt es ein Forum, in dem sie Konsens ist — wer die falsche Blase betrifft, wird gelöscht oder angegangen, oft bevor jemand diskutieren konnte. Die Wut hat einen realen Kern (diese Communities ertrinken tatsächlich in lieblos generiertem Zeug, und Kreative haben echte Existenzsorgen), aber sie unterscheidet nicht mehr zwischen Spam und Familienprojekt. Für unsere Strategie hieß das schlicht: Wir erzählen die Geschichte dort, wo Menschen sie lesen, bevor sie urteilen — im eigenen Blog, in diesem Buch, in Communities, die Projekte statt Schlagworte bewerten. Verstecken werden wir das KI-Thema dabei nie; ein Projekt, dessen interessanteste Eigenschaft man verschweigen muss, hätte sein Fundament verraten. Und für alle, die Ähnliches vorhaben, die Kurzfassung: **Rechne damit, dass dich Orte, die du liebst, für dein Werkzeug hassen — und wähle deine Bühnen danach.** Ein anonymer Kommentar, der das Spiel „Slop" nennt, hat es nicht gespielt. Justus' Zettel wiegt mehr.

Die Kampagne: auf den Listen der anderen

Neben Social Media lief eine zweite, leisere Kampagne, die ich Entwicklern besonders ans Herz lege, weil sie fast nichts kostet und lange trägt: **Awesome Lists**. In der Open-Source-Welt gibt es kuratierte Link-Sammlungen („awesome-…") zu jedem Thema — auch zu Open-Source-Spielen. Wer dort steht, wird von genau den Menschen gefunden, die sich für so etwas begeistern: Entwickler, Bastler, potenzielle Mitstreiter.

Also haben wir — Recherche und Antragstexte KI-gestützt, versteht sich — systematisch Aufnahme beantragt: den Regeln jeder Liste entsprechend, mit ordentlicher Beschreibung, als richtige Pull Requests. Stand heute ist das Spiel auf **drei solcher Listen** aufgenommen, weitere Anträge laufen. Das ist unglamouröses Marketing ohne Feuerwerk — aber es sind dauerhafte Einträge an Orten, wo unsere Zielgruppe ohnehin sucht. Reichweite muss nicht laut sein.

Die ehrliche Niederlage: itch.io

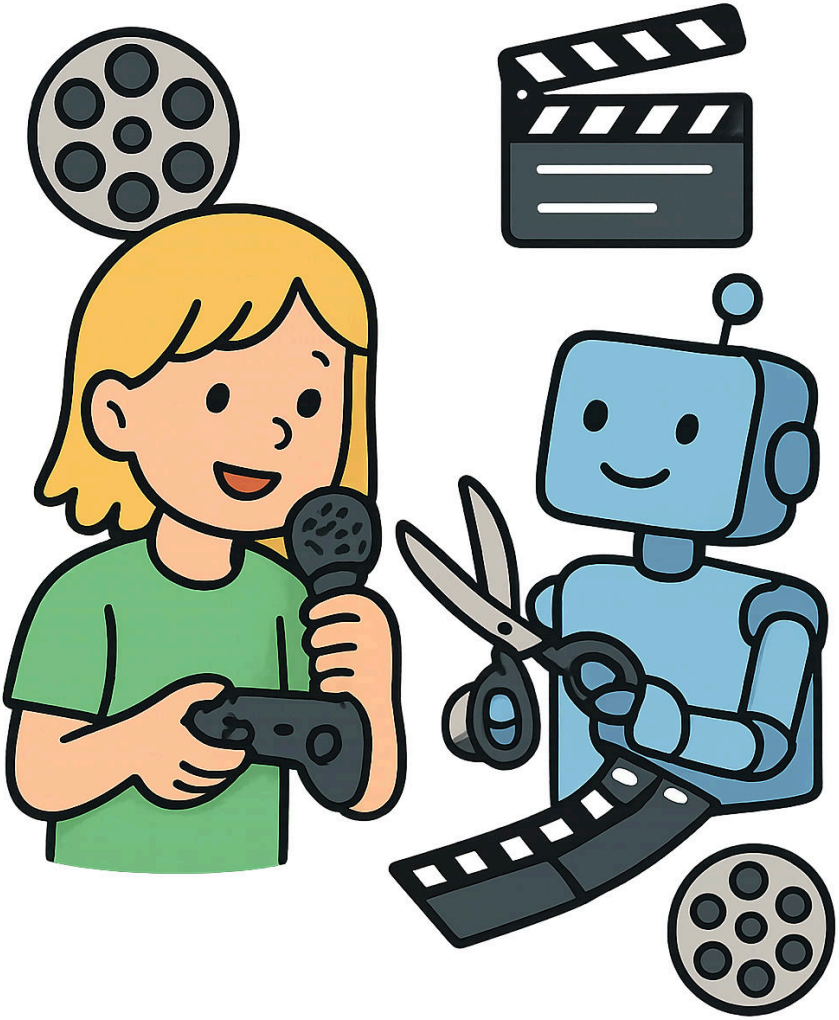
Und dann ist da die Geschichte, die in Marketing-Kapiteln anderer Bücher gern fehlt. **itch.io** ist *der* Marktplatz für Indie-Spiele, und natürlich waren wir dort: Spieleseite eingerichtet, Uploads automatisiert (der Release-Zug aus Kapitel 16 lieferte jede Version auch dorthin), und dazu liebevoll gepflegte englische Devlogs — persönlich geschrieben, um Justus' Testabende herum erzählt, mit allem Herzblut.

Es hat nicht funktioniert, und die Zahlen sind von einer Ehrlichkeit, die fast wieder komisch ist: In der Flut täglich neuer Spiele dort aufzufallen ist nahezu unmöglich — **unser Spiel wird auf itch.io bis heute nicht einmal in der Suche gefunden, wenn man seinen eigenen Namen eingibt.** Es hatte eine Handvoll Views. Und von den **zwei** Downloads war mindestens einer von einem Freund. (Ich wiederhole: Wir haben eine automatisierte Upload-Pipeline für diese Plattform gebaut. Für zwei Downloads. Einer davon Bastian-Kategorie.) Irgendwann haben wir die Konsequenz gezogen, die sich falsch anfühlt und richtig ist: **aufgehört, dort Devlogs zu schreiben.** Nicht beleidigt — die Uploads laufen weiter, das Spiel bleibt dort verfügbar. Aber die Erzähl-Energie, die begrenzteste Ressource dieses Projekts, fließt seitdem dorthin, wo sie ankommt: in die eigene Website, den eigenen Blog, den eigenen Kanal.

Wenn dieses Kapitel eine Lektion hat, dann diese, und sie gilt weit über Spiele hinaus: **Nicht jeder Kanal trägt, und das rechtzeitig zuzugeben ist Strategie, keine Kapitulation.** Marketing mit KI heißt

nicht, überall gleichzeitig zu senden — die Maschine würde das anstandslos tun, Content ist ja billig geworden. Es heißt, mit begrenzter menschlicher Energie die Orte zu wählen, an denen echte Menschen antworten. Auch das ist, am Ende, Kuratieren.

> **Was die KI hier wirklich getan hat** Gemini: Social-Media-Strategie, Subreddit-Auswahl, Post-Formulierungen, Reichweiten-Ideen. Claude: das Blog-Publishing-Skript (Markdown → Website-API, zweisprachig verknüpft), Devlog- und Antragstexte, die Awesome-List-Pull-Requests. Der Mensch hat: die Kanäle gewählt und wieder abgewählt (itch!), jeden Text redigiert und persönlich verantwortet — und die Regel durchgehalten, dass nichts postet außer ihm selbst. Reichweite kann man automatisieren. Glaubwürdigkeit nicht.



Kapitel 21 — 28 Videos in 30 Minuten

Die Produktion von 28 Videos dauerte 30 Minuten. Das Hochladen auf YouTube dann 40.

Von allen Werkstatt-Geschichten dieses Buches ist die folgende diejenige, die ich am häufigsten erzähle, wenn jemand wissen will, was KI-Werkzeuge im Alltag *wirklich* verändern. Nicht, weil sie die technisch beeindruckendste wäre — sondern weil jeder sofort versteht, was hier verschwunden ist: **der Videoschnitt.** Jene Tätigkeit, an der Content-Träume seit zwanzig Jahren sterben, weil auf eine Stunde Spielspaß zehn Stunden Schneideraum kommen.

Bei uns kam auf mehrere Stunden Rohmaterial: eine halbe Stunde. Heraus kamen 22 Kurzvideos und 6 längere Let's-Plays — untertitelt in zwei Sprachen, mit fertigen Beschreibungstexten. So ging's.

Schritt 1: Justus drückt die Aufnahmetaste

Jedes Gaming-Video beginnt damit, dass jemand das Spiel spielt, und diese Rolle war unstrittig besetzt. Justus nahm mehrere Spielsessions mit **OBS Studio** auf (dem Standard-Gratiswerkzeug für Bildschirmaufnahmen), inklusive Live-Kommentar. Kein Skript, keine Wiederholungen, keine Stichwortkarten — einfach er, wie er spielt und dabei redet. Diese unverfälschte Energie war genau das, was wir wollten; niemand braucht einen Zehnjährigen, der abliest. Es ist nebenbei der arbeitsteiligste Moment des ganzen Familienprojekts: Der Teil der Produktion, der am längsten dauert, ist für den Ausführenden **keine Arbeit**.

Schritt 2: Das Video lesbar machen

Jetzt das Problem: Eine KI kann stundenlanges Videomaterial nicht effizient „anschauen". Der Trick — und er ist der Schlüssel des ganzen Kapitels — besteht darin, das Video **lesbar** zu machen. Die Aufnahmen wanderten zu **ElevenLabs**, das daraus Transkriptionen erzeugte: alles Gesprochene als Text, **mit Zeitstempeln** — wer hat was wann gesagt, auf die Sekunde. Plötzlich ist ein Video keine Pixelwand mehr, sondern ein durchsuchbares Dokument.

> **Kurz erklärt: Warum Zeitstempel alles ändern** Ein Transkript mit Zeitstempeln ist die Landkarte eines Videos: „12:41 — Justus entdeckt die Höhle und ruft ‚WAS ist DAS?!'" ist für eine Text-KI ein Fundstück, das sie bewerten kann wie eine Codezeile — und der Zeitstempel sagt der Schnittsoftware exakt, wo die Schere anzusetzen ist. Sobald Bewegtbild als strukturierter Text vorliegt, kann eine KI Tempo, Höhepunkte und langweilige Passagen analysieren, ohne je ein Bild „gesehen" zu haben. Das Prinzip trägt weit über Videos hinaus: Mach dein Material lesbar, und die Maschine kann damit arbeiten. (Kapitel 5 nannte das: Verlege dein Projekt in Text.)

Schritt 3 und 4: Erst denken, dann schneiden

Videos plus Transkripte landeten in einem **Claude-Cowork-Projektordner** (Cowork ist, vereinfacht, Claude mit Werkbank: ein Arbeitsbereich, in dem die KI mit Dateien hantieren kann). Dazu legten wir unsere Gestaltungs-Grundlagen — Markenfarben, Schriften —, damit Titel und Einblendungen nach *unserem* Spiel aussehen und nicht nach Zufallsvorlage.

Und dann kommt der Schritt, den ich zweimal unterstreichen würde, denn es ist wörtlich dieselbe Lektion wie beim Programmieren in Kapitel 4: **Man lässt die KI nicht einfach anfangen.** Zuerst Brainstorming —

welche Momente taugen für Shorts, was ist ein guter Aufhänger, was fliegt raus. Dann ein **detaillierter Schnittplan**, bevor irgendetwas gerendert wird: welche Clips, welche Zeitfenster, Kurz- und Langfassung, eingebrannte Untertitel auf **Deutsch und Englisch** (ein zweisprachiges Familienprojekt bleibt zweisprachig), plus eine Datei mit fertigen YouTube-Beschreibungen für jedes Video. Erst nach Freigabe des Plans wurde geschnitten. Analyse, Plan, Rückfrage, Umsetzung — gleicher Workflow, neues Medium. Die Methode ist offenbar wichtiger als das Material, auf das man sie anwendet.

Schritt 5 und 6: Voilà — und die Pointe

Rund **eine Stunde Arbeit mit Claude** später lag alles da: **22 Shorts — je elf auf Deutsch und elf auf Englisch — und drei längere Videos, ebenfalls in beiden Sprachen**, untertitelt, beschriftet, hochladefertig. Dazu, und das ist der Teil, den ich fast am elegantesten finde: **ein fertiges Redaktionsdokument** — Zeitplan für die Veröffentlichungen, Beschreibungstexte für jedes einzelne Video, alles in einer Datei zum Abarbeiten.

Und jetzt die Wendung, die diese Geschichte perfekt macht: **Das Einstellen bei YouTube und das Planen der Veröffentlichungen dauerten länger als das Schneiden der Videos und Justus' OBS-Aufnahmen zusammen.** Jedes Video einzeln hochschieben, Beschreibung aus Claudes Datei übernehmen, Veröffentlichungstermin setzen — klicken, einfügen, wiederholen, fünfundzwanzigmal. Für diesen letzten Meter gibt es schlicht keine brauchbare Automatisierungs-Schnittstelle; da half keine KI-Magie. Unsere futuristischste Pipeline, ausgebremst von Copy & Paste. Es ist dieselbe Pointe wie bei Suno in Kapitel 9, nur verschärft — und inzwischen mein liebstes Beispiel dafür, wo Automatisierung 2026 wirklich endet: selten am Können der Modelle. Meistens an einer fehlenden Steckdose.

Der Trailer: vier KIs, ein Staffellauf, zwanzig Sekunden

Wie produziert eine Familie ohne Schnittplatz einen richtigen Spiele-Trailer? Bei uns war es der dichteste KI-Staffellauf des ganzen Projekts, und weil er so schön das Zusammenspiel zeigt, hier die komplette Kette:

Erstes Läuferpaar: die Musik. Ich bat **Gemini** um einen Prompt für einen **epischen Instrumental-Track für einen Spieletrailer** — die Marketingabteilung beschreibt, die Hauskapelle spielt: Den Prompt gab ich an **Suno**, das den Track generierte.

Zweiter Läufer: der Schnittmeister. Track und Auftrag gingen an **Claude Cowork**: erst den Song auf **zwanzig Sekunden** kürzen — Trailer-Länge. Dann legte ich ihm das Videomaterial hin, das wir ohnehin hatten: die Let's-Play-Mitschnitte **samt der Audio-Transkription, in der Justus erzählt, was gerade passiert**. Genau diese Tonspur war der Schlüssel — anhand von Justus' eigenen Worten konnte Claude identifizieren, welche Momente ins Bild gehörten.

Die Regeln des Rennens: wie immer erst der **Plan** vor dem Schnitt — und die Auflage, den **visuellen Stil der bereits existierenden YouTube-Videos** einzuhalten (Schrifteinblendungen, Look), damit der Trailer aussieht wie aus demselben Haus. Danach brauchte es noch **zwei Iterationen**, bis das Timing der Einblendungen mit der Musik saß. Fertig war der Trailer.

Und das Nervigste, man ahnt es: **wie immer das Reinstellen bei YouTube.** Upload von Hand, Beschreibung von Hand einfügen — die hatte Claude natürlich längst geschrieben und in eine Datei gelegt. Vier KIs liefern einen Trailer, und der Mensch verbringt den längsten Arbeitsschritt damit, Text in ein Webformular zu kopieren. Wenn es je ein Denkmal für den Stand der Automatisierung im Jahr 2026 gibt, ich

schlage diese Szene vor.

Der Kanal: Familie sendet

Bleibt die Frage, *wohin* das alles sendet. Die Antwort passt zum Baumhaus-Ton des Projekts: kein steriler Studio-Kanal, sondern der **Familien-Kanal** — „mamas und papas blog" auf YouTube, das Bewegtbild-Gegenstück zu dem Blog, auf dem Verena und ich ohnehin über Familienprojekte schreiben. Dort erscheinen jetzt die KI-geschnittenen Play-Videos von Justus: Der Sohn spielt, die KI schneidet, der Familienkanal sendet.

Dazu kommt die zweite Bewegtbild-Schiene, ganz ohne Kamera-Kind: **skriptgesteuerte Marketing-Clips** direkt aus dem Spiel. Im Projekt liegt eine Drehbuch-Datei, die Szenen beschreibt — Kamerafahrt durchs All, Landeanflug, Cockpit —, und ein Skript spielt sie automatisch ab und zeichnet sie in sauberer Qualität auf, ohne HUD, mit Ton. Trailer-Rohmaterial auf Knopfdruck, jederzeit reproduzierbar, nach jedem Grafik-Update frisch.

Werbewirksam ist das alles in bescheidenem Familienprojekt-Maßstab, und das ist in Ordnung. Der eigentliche Wert steht woanders: Es existiert jetzt ein wachsendes, bewegtes Archiv dieses Projekts — ein Zehnjähriger, der sein eigenes Spiel spielt und kommentiert, festgehalten in dem Sommer, in dem es entstand. Selbst wenn kein Algorithmus es je nach oben spült: Für zwei Leute auf diesem Sofa ist das irgendwann das wertvollste Material der Welt.

> **Was die KI hier wirklich getan hat** ElevenLabs: Transkription mit Zeitstempeln — der Schritt, der Video für Text-KI lesbar macht. Claude (Cowork): Material anhand der Transkripte analysiert, Schnittplan erstellt (nach Freigabe!), 28 Videos geschnitten, zweisprachige Untertitel eingebrannt, Beschreibungen getextet. Der Mensch: Justus lieferte das einzige, was nicht generierbar ist

— echtes Spielen mit echter Begeisterung; Marcel kuratierte die Höhepunkte, gab den Plan frei und klickte 28-mal auf „Hochladen".

Kapitel 22 — Was es gekostet hat — und was es wert ist

Ein komplettes Entwicklerstudio — Programmierer, Grafiker, Tonstudio, Übersetzer, Sysadmin, Marketingabteilung — für den Preis eines Familien-Pizzaabends pro Woche.

Bücher wie dieses schulden ihren Lesern am Ende Zahlen. Nicht die geschönten aus dem Prospekt, sondern die vom Kontoauszug. Hier sind unsere — nach sechs Wochen Projektlaufzeit, achtzehn Releases und einem laufenden Online-Dienst.

Die Geldseite: die monatliche Werkzeugmiete

Was dieses Projekt monatlich kostet, passt in eine kurze Tabelle, und ich gebe die Posten so an, wie sie bei uns anfallen:

Posten	Kosten (ca., pro Monat)
ChatGPT-Abo (Konzept, Recherche, Planung)	20 €
Gemini-Abo (Social Media, Reichweite)	25 €
Claude-Max-Abo (der Bautrupp: Code, Tests, Ops, Video)	80 €
API-Kosten OpenAI + ElevenLabs (Texturen, Sounds)	unter 5 €
Suno (Musik)	~10 $
Server-Miete (VPS, die ganze Online-Flotte)	5 €
Website (Wix)	10 €
Summe	**≈ 150–160 €**

Zwei Posten verdienen einen zweiten Blick, weil sie den meisten Intuitionen widersprechen.

Der größte Posten ist das Claude-Abo — und es ist der beste Deal der Tabelle. Achtzig Euro klingen nach viel für ein Hobby, bis man dagegenhält, was dieser Posten ersetzt: Für 80 € im Monat bekommt man von einem Freiberufler-Entwickler in Deutschland nicht einmal eine Stunde. Unser Abo liefert *jeden Abend* mehrere Stunden Entwicklungs-, Test-, Doku- und Adminarbeit. Die Zahlen aus Kapitel 4 — 867 Commits in einem Monat — laufen über diesen Posten. Man kann es so sagen: Die teuerste Zeile der Tabelle ist gleichzeitig die mit dem absurdesten Gegenwert.

Der kleinste Posten ist die größte Überraschung: unter fünf Euro API-Kosten. Sämtliche Texturen, Icons, Kreaturenbilder und Soundeffekte dieses Spiels — die komplette Sinneswelt aus den Kapiteln 8 und 9 — kosten weniger als ein Döner. Das ist kein Rechenfehler, sondern das Ergebnis der Kostenbremsen aus Kapitel 8 (gezielt bestellen, nichts blind neu generieren) und der schlichten Tatsache, dass Assets, einmal erzeugt, für immer im Spiel bleiben. Und der Fünf-Euro-Server, auf dem die gesamte Online-Welt läuft, ist die späte Dividende der Raspberry-Pi-Zeile aus Kapitel 2.

Zum Vergleich, und diesen Vergleich muss man einmal ausschreiben: Was hier für ~155 € Monatsmiete entsteht, ist das Leistungsspektrum eines kleinen Studios — Programmierung, Grafik, Sound, Musik, Übersetzung (das Spiel ist durchgehend zweisprachig!), Systemadministration, Videoproduktion, Marketing-Strategie. Konservativ gerechnet stecken allein im Code — ~114.000 Zeilen plus 30.000 Zeilen Tests — nach branchenüblichen Sätzen mehrere Personenjahre.

Die Zeitseite: die ehrliche Währung

Und jetzt die Währung, in der dieses Projekt wirklich bezahlt wurde, denn Geld war es offensichtlich nicht. Auf die Frage nach dem Zeitaufwand gibt es von mir nur eine ehrliche Antwort, und sie lautet wörtlich: **„Ich sitze hier am Wochenende und abends nach der Arbeit … seit es das Projekt gibt."** Jeden Abend. Jedes Wochenende. Seit dem 30. Mai.

Das soll niemand romantisch verklären, also sage ich es unromantisch: Dieses Projekt ist ein zweiter Job. Die KI hat nicht die *Zeit* ersetzt — sie hat verändert, was in der Zeit *entsteht*. Dieselben Abende, die früher für ein Zwanzigstel dieses Spiels gereicht hätten, reichen jetzt für das Ganze. Das ist der eigentliche Tauschhandel des KI-Zeitalters, und man sollte ihn kennen, bevor man ihn eingeht: Die Werkzeuge senken nicht den Einsatz. Sie erhöhen den Wechselkurs.

Was den Einsatz erträglich — nein: schön — gemacht hat, steht in Kapitel 14: Es waren keine einsamen Abende. Der Cheftester saß daneben, die Balancing-Abteilung rief zum Essen, und das Projekt *war* die Familienzeit, nicht ihr Konkurrent. Nicht immer, nicht perfekt. Aber im Kern.

Die andere Seite der Bilanz

Was steht dem gegenüber? Die zählbaren Dinge zuerst, und ich schreibe sie so ehrlich hin, wie sie sind — denn dieses Buch verkauft keinen Hit, sondern einen Weg. Stand Mitte Juli 2026, sechs Wochen nach Projektbeginn: ein veröffentlichtes, kostenloses, quelloffenes Spiel in achtzehn Versionen, auf drei Plattformen und im Browser. **21 Sterne auf GitHub. 289 Repository-Aufrufe. Rund 240 Besuche auf der Website. Drei externe Contributor. Fünf Testspieler, von denen wir wissen — und auf den offiziellen Online-Welten: noch keiner.**

Und ein Erfolg, der in keiner dieser Statistiken auftaucht und auf den ich unverhältnismäßig stolz bin: **Wer „Blocks Beyond the Stars" googelt,**

findet das Spiel. Für eine so kleine Website ist das nicht selbstverständlich — Sichtbarkeit in einer Suchmaschine muss man sich gegen das halbe Internet erarbeiten. Und mehr noch: **Fragt man Googles KI-Suchmodus, was das ist, bekommt man eine Antwort.** Die Such-KI kennt unser Spiel und erklärt es. Verzeih mir die Freude an der Schleife, die sich hier schließt: Ein Spiel, das von KIs gebaut wurde, ist inzwischen Teil des Wissens, aus dem KIs antworten. Wir sind, in bescheidenstem Maßstab, Teil des Kanons geworden.

Das sind, an Internet-Maßstäben gemessen, ansonsten winzige Zahlen, und ich lasse sie bewusst unfrisiert stehen. Erstens, weil sie zum Zeitpunkt dieses Kapitels schlicht die Wahrheit sind — sechs Wochen sind im Reichweiten-Geschäft nichts, und die Schul-Vorstellung (Kapitel 24) liegt noch vor uns. Zweitens, weil in diesen kleinen Zahlen die vielleicht wichtigste Einordnung des ganzen Buches steckt: **Die Produktionsseite ist explodiert — die Aufmerksamkeitsseite nicht.** KI hat uns befähigt, in sechs Wochen zu bauen, was früher Jahre gekostet hätte. Sie hat uns keinen einzigen Spieler geschenkt. Sichtbarkeit ist die Währung geblieben, die man nicht generieren kann — man verdient sie langsam, bei Menschen, einen nach dem anderen (Kapitel 20 hat gezeigt, wie mühsam das ist). Wer also mit KI „schnellen Erfolg" bauen will, baut am falschen Ende. Wer damit *sein Ding* bauen will: bitte sehr, es war nie leichter. Und drittens sind drei dieser Zahlen mehr wert als alle anderen zusammen: **Drei fremde Menschen fanden dieses Spiel gut genug, um daran mitzubauen.** Diese Zahl hätte ich am Tag eins gegen jede Downloadstatistik getauscht.

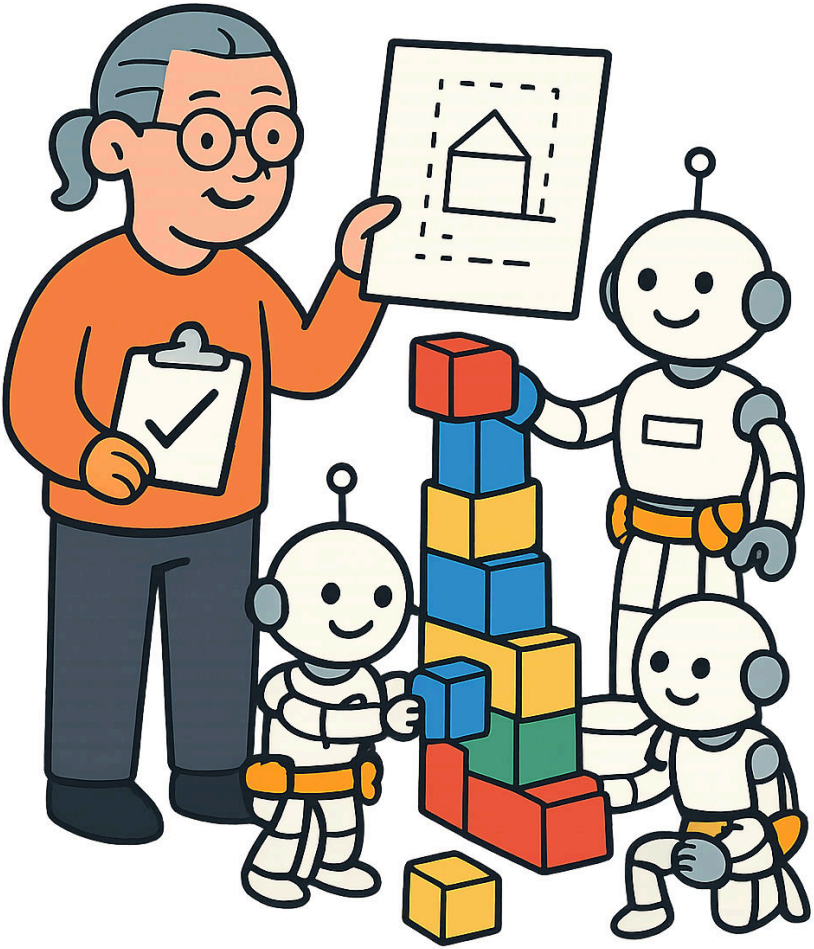
Und dann die unzählbaren, die eigentliche Rendite. Ein Zehnjähriger, der in sechs Wochen gelernt hat, was eine Idee von einer Beschreibung unterscheidet, was ein Product Owner tut, wie man Kritik bestellt und verarbeitet, und dass „das hat die KI gemacht" keine Ausrede ist. Ein Vater, der seinen Beruf neu kennengelernt hat — als Bauleiter statt Handwerker — und dabei mehr über Software-Architektur, Betrieb und Verantwortung gelernt hat als in manchem Berufsjahr. Und eine Familie

mit einem gemeinsamen Werk, das man anfassen, spielen und herzeigen kann.

Der teuerste Fehler? Ich habe lange gesucht und finde keinen der üblichen Sorte — kein verlorener Spielstand, kein verbranntes Budget, kein Monat in der Sackgasse; selbst die Umwege (der Organic-Build, itch.io) waren lehrreich, und das Vorankommen war durchweg gut. Was wirklich Energie gekostet hat, und zwar die teure Sorte — Nerven —, waren **die toxischen Reddit-Kommentare. Genauer: mein Drang, darauf zu antworten.** Die Geschichte samt Gemini-Rat steht in Kapitel 20; hier nur die Bilanz-Lehre daraus: Der teuerste Posten dieses Projekts war keine Ausgabe und kein technischer Irrtum. Es war Aufmerksamkeit, die in Rechtfertigung floss statt ins Bauen. Sie ist inzwischen umgebucht.

Bleibt die Frage, die diese Bilanz aufwirft und die das letzte Kapitel beantwortet: Wenn das alles mit 155 € im Monat und den Abenden einer einzigen Familie geht — was hieße es dann erst, wenn noch ein paar Leute mitbauen?

> **Was die KI hier wirklich getan hat** Den Wechselkurs geändert. Nicht die Kosten gesenkt (die waren nie das Hindernis) und nicht die Zeit geschenkt (die kostet das Projekt bis heute) — sondern den Gegenwert eines Feierabends vervielfacht. Die ganze Tabelle oben ist nur die Miete für diesen Wechselkurs.



Kapitel 23 — Nimmt uns die KI die Arbeit weg?

Dieses Buch hat KI benutzt, wo immer es ging — für Code, Grafik, Texte, Musik, Sound. Wenn dich das die ganze Zeit gestört hat: Dieses Kapitel ist für dich.

Es gibt eine Frage, die über diesem Buch schwebt, seit im Vorwort das Wort „KI" zum ersten Mal fiel. Sie stand zwischen den Zeilen, als die Bild-KI in Kapitel 8 hundert Texturen malte. Sie wurde laut, als uns in Kapitel 20 auf Reddit vorgeworfen wurde, wir würden „die Arbeit von Menschen herabsetzen". Und sie verdient mehr als eine Fußnote, nämlich ein eigenes Kapitel, so ehrlich ich es hinbekomme:

Nimmt uns die KI die Arbeit weg?

Denn machen wir uns nichts vor: Dieses Buch beschreibt ein Projekt, das an jeder Stelle, an der es ging, Maschinen eingesetzt hat — und zwar genau dort, wo Menschen arbeiten. Programmierer. Grafikerinnen. Texter. Musikerinnen. Sounddesigner. Die Angst, die diesen Berufen gerade in den Knochen sitzt, ist real, sie ist verständlich, und wer sie hat, soll hier nicht belächelt, sondern ernst genommen werden.

Was dieses Projekt verdrängt hat — und was nicht

Fangen wir mit der ehrlichen Inventur an, im Kleinen, wo wir die Fakten kennen. Hat *dieses* Projekt jemandem Arbeit weggenommen? Die Antwort steht schon in Kapitel 8, und sie gilt für jedes Gewerk dieses Buches: **Es gab nie ein Budget, das stattdessen ein Mensch bekommen hätte.** Die Alternative zu den KI-Texturen war kein

beauftragter Grafiker — die Alternative war ein graues Spiel aus Platzhalter-Klötzen, das nie erschienen wäre. Die Alternative zu diesem Buch war kein beauftragter Ghostwriter. Es war: kein Buch.

Aber ich mache es mir nicht so leicht, dabei stehen zu bleiben. Denn natürlich ist das Familienprojekt der bequemste Fall. Die berechtigte Sorge lautet ja: Was passiert, wenn *alle* so arbeiten — auch die Firmen, die bisher Grafiker, Texter und Entwickler bezahlt haben? Wenn das Werkzeug, das bei uns etwas möglich gemacht hat, anderswo etwas *ersetzt*?

Darauf gibt es keine Antwort, die ein einzelnes Familienprojekt beweisen könnte. Aber es gibt eine Einschätzung, und ich gebe sie ab als jemand, der seit vielen Jahren beruflich Software baut und diese Branche durch mehrere Technologie-Umbrüche begleitet hat: **Ich glaube nicht an die massenhafte Arbeitslosigkeit der Schaffenden. Ich glaube an eine tiefgreifende Veränderung der Arbeitsweise — und der Berufsprofile. In allen Berufsgruppen.**

Wie sich die Arbeit verschiebt

Was sich verändert, hat dieses Buch in den bisherigen Kapiteln nebenbei dokumentiert, und man kann es in einem Dreischritt zusammenfassen, der in jeder „Was die KI hier wirklich getan hat"-Box am Kapitelende wiederkehrte:

Erstens: Die Anforderung wird zur Königsdisziplin. Es ist wichtiger geworden, gut und umfangreich zu formulieren, *was* entstehen soll — die 7.737 Zeilen Spezifikation vor der ersten Codezeile, der Stil-Prompt vor der ersten Textur, der Schnittplan vor dem ersten Video. Wer sein Fach beherrscht, konnte schon immer sagen, was fehlt. Jetzt ist genau diese Fähigkeit das Nadelöhr, durch das alles geht.

Zweitens: Die Umsetzung wandert zur Maschine. Vieles vom Tippen, Malen, Schneiden, Konfigurieren — die Fleißarbeit, die früher den

Großteil der Arbeitszeit fraß — übernimmt die KI. Das ist der Teil, der Angst macht, und ich verstehe warum: Es ist der sichtbarste Teil der bisherigen Arbeit.

Drittens: Der Mensch kehrt zurück — als Prüfer. Und dann muss wieder ein Mensch ran: prüfen, testen, nachjustieren, verwerfen, freigeben. Der Energiezaun-Summton, der dreimal zurückging. Der Organic-Build, der komplett gestrichen wurde. Die elfhundert Tests, deren Liste ein Mensch liest. Dieser Teil wird nicht kleiner — er wird *größer*, weil mehr entsteht, das geprüft werden muss.

Was dabei entsteht, ist ein neues Berufsprofil, und es ist anspruchsvoller, nicht anspruchsloser als das alte: Man braucht **das detaillierte Fachwissen** — denn wer die Materie nicht versteht, kann weder gut bestellen noch prüfen, was er bekommt. Und man braucht **gleichzeitig den breiten Überblick über den eigenen Fachbereich** — denn das Werkzeug KI entfaltet sich erst, wenn man weiß, wo im großen Bild es gerade arbeitet. Der Spezialist, der nur seine eine Handbewegung kann, hat es schwerer. Der Fachmensch, der sein Feld durchdringt und die Maschine führen kann, wird gesuchter sein als je zuvor.

> **Kurz erklärt: Warum Werkzeug-Umbrüche selten Berufe töten — aber immer Profile ändern** Die Softwarebranche hat das mehrfach erlebt: Compiler machten Maschinencode-Handarbeit überflüssig, Frameworks ersetzten Boilerplate, Baukästen ersetzten handgeschriebene Websites. Jedes Mal wurde ein Teil des Handwerks wertlos — und jedes Mal wuchs die Branche danach, weil mehr Menschen mehr bauen konnten und die Ansprüche mitwuchsen. Was verschwand, war nie „der Beruf". Es war die jeweilige Definition davon, was an ihm die eigentliche Arbeit sei.

Was ich Skeptikern wirklich antworten würde

Wenn ich einem Grafiker, einer Texterin, einem Kollegen mit dieser Angst gegenübersitze, sage ich nicht „wird schon gutgehen" — Übergänge treffen Menschen unterschiedlich hart, und wer seine Nische schrumpfen sieht, hat keinen Trost von Statistiken. Ich sage drei ehrlichere Dinge.

Erstens: **Dein Fachwissen ist nicht entwertet worden — es hat den Besitzer gewechselt: vom Ausführenden zum Auftraggeber.** Alles, was du über gute Typografie, sauberen Code, tragfähige Dramaturgie weißt, brauchst du weiterhin — nur an einer anderen Stelle des Prozesses: beim Formulieren und beim Prüfen. Die KI in diesem Buch hat nie entschieden, was gut ist. Das haben Menschen getan, die es beurteilen konnten.

Zweitens: **Probier das Werkzeug aus einer Position der Stärke aus.** Nicht, weil man jedem Hype folgen muss — sondern weil die Urteile über KI, die ich ernst nehme, von Leuten kommen, die sie ausprobiert haben. Aus Expertise heraus benutzt, zeigt das Werkzeug einem sehr schnell beides: was es kann und wo es ohne einen aufgeschmissen ist. Beides zu wissen beruhigt mehr als jede Debatte.

Und drittens, ganz persönlich: **Ich freue mich über diese Veränderung.** Das schreibe ich nicht als Verkündigung, sondern als Befund nach vielen Berufsjahren und einem sehr intensiven Sommer: Die Arbeit, die die Maschine mir abgenommen hat, war selten die, an der mein Herz hing. Was übrig bleibt — verstehen, gestalten, entscheiden, prüfen, mit einem Zehnjährigen streiten, ob das noch minecraftig ist —, ist der Teil des Berufs, dessentwegen ich ihn einmal ergriffen habe. Es kann sein, dass es anderen anders geht. Aber es kann eben auch sein, dass am Ende dieser Verschiebung mehr Menschen mehr von der Arbeit machen, die sie eigentlich wollten.

Dieses Buch ist in dieser Debatte nur ein einzelner Datenpunkt. Aber es ist ein vollständiger: Jedes Kapitel hat offengelegt, was die Maschine getan hat und was der Mensch — nachprüfbar, mit Git-Historie. Wer die

Zukunft der Arbeit diskutieren will, dem wollten wir wenigstens ein ehrliches Anschauungsbeispiel liefern statt einer Meinung.

Und wem das als Antwort nicht reicht, dem mache ich das beste Angebot, das ich habe. Es steht im nächsten Kapitel.

> **Was die KI hier wirklich getan hat** Diesen Text formuliert — nach den Überzeugungen, Erfahrungen und dem Berufsweg des Autors, in Runden diskutiert und von ihm freigegeben. Die Position selbst stammt nicht aus einem Modell; kein Sprachmodell hat Berufsjahre in der Softwarebranche. Man könnte sagen: Die Maschine hat das Kapitel über die Zukunft der Arbeit exakt so mitgebaut, wie das Kapitel es beschreibt.

Kapitel 24 — Anleitung zum Nachmachen (light) — und eine Einladung

---

„Lass uns das Spiel zusammen großartig machen!" — von der Mitmachen-Seite unserer Website. Und jetzt auch von der letzten Seite dieses Buches.

Kein Erfahrungsbericht sollte als Anleitung enden — Erfahrungen sind nicht kopierbar, und was bei uns funktioniert hat, verdankt sich auch Glück, Timing und einem sehr speziellen Product Owner. Aber es gibt eine Handvoll Prinzipien, die sich durch jedes Kapitel dieses Buches gezogen haben, und die *sind* übertragbar — auf Spiele, auf Software, vermutlich auf die meisten Projekte, bei denen ein Mensch mit Maschinen baut. Hier sind sie, ein letztes Mal, in Kurzform.

Präzisiere die Absicht, bevor du baust. „Mach mir ein cooles Weltraumspiel" reicht nicht — der wertvollste Teil unseres Projekts waren die 7.737 Zeilen Spezifikation, die existierten, bevor die erste Zeile Code entstand (Kapitel 1–2). KI belohnt Klarheit brutaler als jedes Werkzeug zuvor: Was du sauber beschreibst, wird beeindruckend gut; was du vage lässt, wird beeindruckend selbstbewusst falsch.

Fang mit dem kleinsten Ding an, das läuft. Immer erst das MVP, dann die Ausbaustufe (Kapitel 3–4). Etwas Kleines, das läuft, schlägt etwas Großes, das fast fertig ist — und ein Kind (oder ein Kunde) kann nur beurteilen, was es anfassen kann.

Schreib alles auf — die Regeln, die Entscheidungen, die Pannen. Das Regelbuch für die KI, das Logbuch des Projekts, die

Entscheidungs-Akten, das Gotcha-Gedächtnis (Kapitel 4, 5, 7): Ein Projekt, dessen Wissen in Dateien wohnt statt in Köpfen, kann von einer vergesslichen Maschine — und von jedem neuen Menschen — sofort weitergebaut werden.

Bau Sicherheitsnetze, nicht Vertrauen. Elfhundert Tests, ein Werkstor, das nur Grünes durchlässt, Warnungen als Fehler (Kapitel 6). Vertrauen in KI-Code entsteht nicht durch Lesen und nicht durch Hoffen, sondern durch Beweise, die bei jeder Änderung neu erbracht werden.

Lass Menschen entscheiden — ausnahmslos. Was gebaut wird (der Zettel), was nicht gebaut wird (das Veto), was die KI nie zu sehen bekommt (die Nutzereingaben), wo sie die falsche Technologie ist (die Schatzkarten): Die wichtigsten Momente dieses Projekts waren Entscheidungen, und keine einzige kam aus einem Modell (Kapitel 10–12).

Und behandle die Maschine wie schönes Wetter. Plane für Regen — Fallbacks, Grenzen, ein Mensch in Rufweite (Kapitel 10, 16). Die Projekte, die mit KI scheitern werden, sind nicht die, die ihr zu wenig zutrauen.

Das ist die ganze Anleitung. Der Rest ist Anfangen — und Anfangen ist, das war die Botschaft von Kapitel 3, so billig geworden wie nie: Die Strecke von „wir haben eine genaue Idee" bis „es läuft etwas" misst sich in Wochenenden.

Was würde ich heute anders machen?

Die ehrlichste aller Schlussfragen, und ich habe lange über eine kluge Antwort nachgedacht, die Demut mit Erkenntnis mischt. Es gibt sie nicht. Die wahre Antwort ist kurz, und ich schreibe sie so hin, wie sie sich anfühlt: **Nichts.**

Nicht, weil alles perfekt lief — dieses Buch hat zwei Kapitel voller Pannen und eine ehrliche Niederlage. Sondern weil jeder dieser Umwege etwas dagelassen hat: Der verworfene Organic-Build hat uns gelehrt, was unser Spiel ist. Das verkümmerte itch.io hat uns zur eigenen Heimat gezwungen. Die durchgemachte Nacht war die beste Nacht des Jahres. Ein Projekt, bei dem man rückblickend nichts anders machen würde, ist kein fehlerfreies Projekt — es ist eines, dessen Fehler zur Geschichte gehören. Ich wünsche mir nur eines, und das ist kein Rückblick, sondern ein Wunsch nach vorn: **dass es weitergeht.**

Was als Nächstes kommt: Das Spiel geht in die Schule

Und es geht weiter — an einem Ort, der mir von allen Ausblicken dieses Buches der liebste ist.

Kurz vor Redaktionsschluss dieses Kapitels hatte ich ein Gespräch mit dem — ausgesprochen engagierten — **Schulleiter von Justus' Schule**. Das Ergebnis: Direkt nach den Sommerferien dürfen wir das Spiel **den Schülerinnen und Schülern vorstellen** — mit genau dem Ziel, um das es auf dieser letzten Seite geht: weitere Mitstreiter für das Open-Source-Projekt zu gewinnen. Und wer weiß: Vielleicht wird sogar eine eigene AG daraus — „Spieleentwicklung: Blocks Beyond The Stars". 😊

Man muss sich das einen Moment auf der Zunge zergehen lassen, denn hier schließt sich der größte Bogen dieses Buches. Ein Junge kommt abends mit einem Gameboy-Clone zu seinem Vater (Kapitel 1). Nur wenige Monate später steht derselbe Junge — vielleicht — vor seiner eigenen Schule und zeigt Kindern *sein Spiel*, an dem sie mitbauen dürfen. Die Playtests, die daraus entstehen (endlich die echten Kinder-Playtests, die Kapitel 12 versprochen hat!), die Ideen, die Zettel, die vielleicht ersten Schüler-Beiträge: Das ist die nächste Staffel dieser Geschichte. Beim Erscheinen dieses Buches ist sie hoffentlich

schon im Gange.

Daneben stehen die kleineren Ausblicke: Das Spiel wird weiterentwickelt — der Rhythmus aus Abenden und Wochenenden hat nicht vor aufzuhören, und Justus' Ideenliste wächst schneller, als je gebaut werden kann. Ein kleiner Merch-Shop ist in Vorbereitung — Non-Profit, weil ein Spiel mit eigenem T-Shirt einfach echter ist. Und dieses Buch selbst findet, mal sehen, vielleicht seinen Weg zu einem großen Online-Buchhändler.

Die Einladung

Und damit zur letzten Seite, die keine Zusammenfassung ist, sondern das, was auf unserer Website unter „Mitmachen" steht — hier nur persönlicher.

Dieses Buch hat erzählt, was eine Familie mit ein paar KI-Abos, einem Fünf-Euro-Server und ihren Feierabenden bauen kann. Aber es hat nebenbei auch erzählt, was passiert, wenn andere dazukommen: Ein fremder Mensch schenkte uns den Linux-Port. Ein CEO schenkte uns einen Sonntag und die Browser-Version. Ein Kollege schenkte einem Jungen einen handgeschriebenen Spielebericht. Die besten Kapitel dieses Projekts haben andere Leute geschrieben — und genau deshalb endet dieses Buch nicht mit einem Schlusswort, sondern mit einer Einladung:

Wir suchen Mitstreiter.

Tester, die das Spiel mit ihren Kindern zerlegen und uns Zettel schreiben (F1 genügt). Entwickler, die Lust auf Unity, C# oder einfach auf ein freundliches Open-Source-Projekt haben — der komplette Code ist offen, die Einstiegshürden sind dokumentiert, das Contributor-Abkommen ist ein Klick. Kreative, die Welten bauen, Geschichten spinnen, Screenshots machen. Familien, die einfach spielen und erzählen, wie es war. Der Weg ist kurz und steht in Kapitel 17

vorgezeichnet: Website → GitHub → erste eigene Welt → erster eigener Beitrag. Cora und Devin sind ihn vorausgegangen — der Beweis, dass er funktioniert.

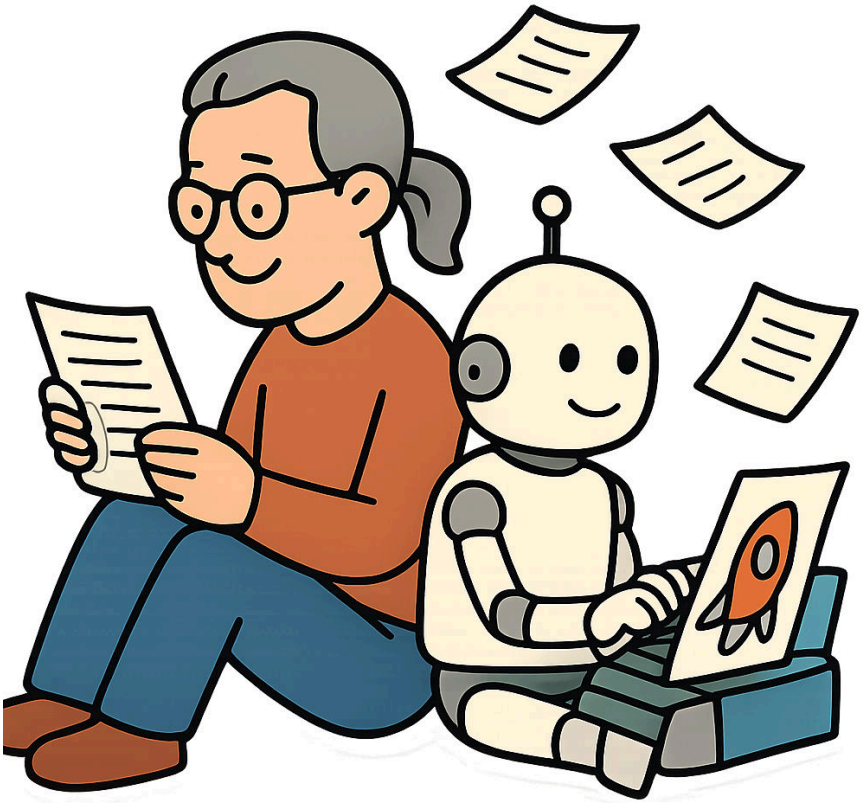
Das Spiel ist kostenlos, quelloffen und wird es bleiben. Es gehört einer Familie und, im besten Sinne, allen, die mitbauen.

Und falls du mit der Sorge aus dem Vorwort bis hierher gelesen hast — der Frage, was von unserer Arbeit übrig bleibt, wenn Maschinen bauen können: Das hier ist meine ganze Antwort, in einem Satz. **Die beste Erwiderung auf die Angst vor der Maschine ist ein Platz am Tisch — und hier ist deiner.**

Lass uns das Spiel zusammen großartig machen.

blocksbeyondthestars.com · github.com/marceld23/BlocksBeyondTheStars

— Marcel, Justus & Verena (und, in alphabetischer Belegschafts-Reihenfolge: Claude, ChatGPT, ElevenLabs, Gemini, NotebookLM, OpenAI, Suno)



Kapitel 25 — Wie dieses Buch geschrieben wurde

Dasselbe Rezept, ein letztes Mal — nur dass das Produkt diesmal aus Kapiteln besteht statt aus Blöcken.

Im Vorwort habe ich versprochen, dass dieses Buch nach demselben Rezept entstanden ist wie das Spiel, von dem es handelt. Jetzt, am Ende, löse ich das Versprechen im Detail ein — denn das Making-of dieses Buches ist selbst ein Kapitel über KI-Arbeit, und zwar eines mit einer Besonderheit: Es ist das einzige Kapitel, dessen Entstehung du gerade in den Händen hältst.

Die erste Quelle: das Gedächtnis der Maschine

Alles begann mit einer Frage an Claude Code: *Analysiere dein Memory und die Dokumente — was weißt du über dieses Projekt?*

Dazu muss man wissen, was dieses „Memory" ist. Claude Code — der Coding-Agent aus Kapitel 3, der das Spiel mitgebaut hat — führt über die Zusammenarbeit hinweg **eigene Gedächtnisdateien**: kleine Textdateien, in denen er sich Projektwissen notiert, das über eine einzelne Arbeitssitzung hinaus wichtig bleibt. Was wurde entschieden und warum. Welche Fallen wir schon kennen (die Gotchas aus Kapitel 7!). Welche Arbeitsweisen sich bewährt haben. Woran wir zuletzt gearbeitet haben. Über die Wochen des Spielebaus waren so, nebenbei und ohne dass je jemand „führe Tagebuch" gesagt hätte, **an die 150 solcher Notizdateien** entstanden — ein maschinengeschriebenes Projekttagebuch.

Als die Buchidee aufkam, war dieses Gedächtnis die erste Quelle: Die KI, die das Spiel gebaut hatte, erinnerte sich an den Bau. Nicht an die Gefühle, nicht an die Küchentisch-Szenen — aber an jede technische Wendung, jede Entscheidung, jede Panne, mit Datum.

Die zweite Quelle: die Git-Historie

Die Erinnerung einer KI ist gut; belegbar ist besser. Die zweite Quelle war deshalb die **Git-Historie** — und wer Kapitel 3 gelesen hat, kennt das Prinzip schon: Git speichert jeden Arbeitsschritt eines Software-Projekts als datierten Schnappschuss. Unser Projekt liegt öffentlich auf GitHub, und seine Historie umfasst über tausend solcher Schnappschüsse: jede Änderung, jedes Release, jeder Beitrag von außen, auf den Tag und die Minute genau.

Für ein Erinnerungsbuch ist das ein Traum und ein Korrektiv zugleich. Ein Traum, weil sich daraus die komplette Chronologie rekonstruieren ließ — wann kam VEGA, wann der Grafik-Overhaul, wann Coras Linux-Port. Und ein Korrektiv, weil menschliche Erinnerung rundet und schönt: Mehr als einmal hat die Historie meine Erzählung präzisiert (und einmal ein hübsches Rätsel aufgegeben — die Sache mit dem ersten Commit, der eine Woche nach dem Bau-Wochenende liegt; Kapitel 3 erzählt die Auflösung). Alle Zahlen in diesem Buch — Commits, Releases, Codezeilen, Contributor — stammen aus dieser öffentlichen Quelle. Du kannst sie nachzählen.

Die dritte Quelle: das Interview (schon wieder)

Aber Memory und Git kennen nur die Baustelle — nicht den Abend mit dem Gameboy-Clone, nicht den Zettel, nicht das Veto. Die menschliche Hälfte des Buches entstand deshalb genau so wie einst das Spiel selbst, und die Symmetrie ist mir erst unterwegs aufgefallen: **Ich habe mich von Claude Code interviewen lassen.**

Der Ablauf war exakt das Kapitel-1-Rezept mit vertauschten Rollen. Damals: Vater interviewt Sohn, ChatGPT strukturiert. Diesmal: Die KI legte eine Gliederung mit offenen Fragen an — für jedes Kapitel eine ehrliche Liste, wofür Material vorlag und was nur ich wissen konnte. Und ich steuerte **nach und nach meine Erinnerungen bei**. Mal ein paar Sätze zwischendurch. Mal eine lange Sprachnotiz-artige Antwort über die Nacht zum ersten Prototyp. Mal eine Korrektur („Bastian war kein Freund, sondern ein Kollege"). Jede Antwort wurde eingearbeitet; jede Lücke, die blieb, wurde im Text **markiert statt erfunden** — bis auch sie gefüllt war. Das Buch wuchs damit wie das Spiel: in Iterationen, MVP für MVP, Kapitel für Kapitel, mit mir als Product Owner und der Maschine als unermüdlichem Schreiber.

Und so entstanden — natürlich — **Markdown-Dateien**: einfache Textdateien, eine pro Kapitel, dasselbe Format, in dem schon Justus' Spielkonzept, die technischen Anforderungen und der Devblog existieren. Dieses Projekt hat vom ersten Tag an alles Wichtige in Textdateien gelegt (Kapitel 5 hat erklärt, warum: Text kann jeder lesen — Menschen, Git und Maschinen). Es war nie eine Frage, dass sein Buch genauso beginnen würde.

Vom Text zum Buch: die letzte Werkzeugkette

Textdateien sind aber noch kein Buch. Also wurde — du kennst das Muster inzwischen im Schlaf — **ein Programm geschrieben, natürlich von der KI**: ein kleines Python-Skript. Es sammelt die Kapiteldateien in der richtigen Reihenfolge ein, filtert die Arbeitsnotizen heraus und setzt daraus ein **PDF**, mit Titelseite und Inhaltsverzeichnis. Ein Befehl, ein Buch. Das Skript liegt beim Manuskript. Es ist der kleine Bruder der Asset-Generatoren aus Kapitel 8 und des Blog-Publishers aus Kapitel 20 — dieselbe Familie: Werkzeuge, die sich das Projekt selbst gebaut hat.

Auch die **Illustrationen** in diesem Buch — eine pro Kapitel — stammen

aus derselben Fabrik, und ihre Entstehung ist mein Lieblings-Staffellauf des ganzen Projekts, weil er mit einem Pinsel beginnt statt mit einem Prompt. Am Anfang stand ein **von Verena gemaltes Bild** — echtes Papier, echte Farben, genau die Anmutung, die wir uns für das Buch wünschten. Dieses Bild gab ich **Gemini** und ließ mir den Stil **präzise beschreiben**: saubere schwarze Umrisslinien, flache Farben, weißer Hintergrund, freundliche Kinderbuch-Anmutung. Diese Beschreibung bekam **Claude**, der daraus (natürlich) **ein Python-Programm schrieb**: Es kennt für jedes Kapitel eine Szene, hängt den Stil an, achtet auf wiederkehrende Figuren — der Papa mit grauem Zopf und Brille, der Junge mit langen blonden Haaren — und bestellt die Bilder über **OpenAIs Bild-API**. Ein Mensch malt vor, eine KI beschreibt, eine baut das Werkzeug, eine zeichnet nach. Ausgesucht und verworfen hat, wie immer, ein Mensch — die ersten Stilversuche flogen komplett raus.

Der Plan für den Rest der Strecke steht, und ich schreibe ihn hier bewusst im Futur, denn dieses Kapitel entsteht, während er läuft. Aus den Markdown-Dateien wird ein **Word-Dokument auf Basis der Amazon-Selfpublishing-Vorlage** — mit dem Ziel, das Buch über Amazons Print-on-Demand **als gedrucktes Buch bestellen zu können**. Das **Cover** entsteht nach demselben Rezept wie die Illustrationen. Was dann noch fehlt: **etwas Lektorat** — die eine Werkstatt-Leistung, bei der vier Augen aus Fleisch und Blut durch nichts zu ersetzen sind. Aber das Ziel steht.

Und damit keine Missverständnisse aufkommen, zum Geschäftsmodell dieses Buches ein klares Wort: **Es gibt keines.** Das gedruckte Buch wird zum Non-Profit-Preis erscheinen — also zu dem, was Druck und Vertrieb eben kosten —, und **als PDF wird es kostenlos verfügbar sein**. Schließlich ist das hier ja Open Source. 😊 Wer nach fünfundzwanzig Kapiteln über ein verschenktes Spiel, verschenkte Ports und verschenkte Sonntage erwartet hätte, dass ausgerechnet beim Buch die Kasse klingelt, hat das Projekt nicht kennengelernt.

Die Schleife schließt sich

Zum Schluss die Beobachtung, derentwegen dieses Kapitel existiert. Stell die beiden Produktionen nebeneinander:

Das **Spiel**: eingesprochene Idee → strukturierte Dokumente → KI baut in Iterationen → Mensch prüft, korrigiert, entscheidet → automatische Pipeline verpackt und veröffentlicht → kostenlos und offen für alle.

Das **Buch**: gesammelte Quellen → strukturierte Gliederung → KI schreibt in Iterationen → Mensch erzählt, korrigiert, entscheidet → ein Skript verpackt es zum PDF → kostenlos und offen für alle.

Es ist **dieselbe Fabrik**. Nur das Fließband wurde umgerüstet — von Blöcken auf Kapitel. Und das ist, glaube ich, die eigentliche Botschaft, mit der ich dich entlassen möchte, ehe die Anhänge das letzte Wort haben: Was du in diesem Buch gelesen hast, ist keine Spieleentwicklungs-Methode. Es ist eine **Arbeitsweise** — Absicht präzisieren, klein anfangen, alles aufschreiben, Maschinen bauen lassen, Menschen entscheiden lassen —, und sie funktioniert offenbar für sehr verschiedene Dinge. Für ein Weltraumspiel. Für ein Buch über das Weltraumspiel. Und, wer weiß: für das, was du als Nächstes vorhast.

> **Was die KI hier wirklich getan hat** Claude Code hat die Quellen ausgewertet (sein eigenes Memory, die Git-Historie, den Devblog, die Original-Spezifikationen), mich in Runden interviewt, die Kapitel geschrieben und umgeschrieben, offene Lücken markiert statt gefüllt — und das PDF-Werkzeug gebaut. Der Mensch hat: erinnert, erzählt, korrigiert, gestrichen, entschieden und jede Seite verantwortet. Wenn dir das bekannt vorkommt: Genau das war der Plan.

Anhang A — Zeitleiste: Sechs Wochen, achtzehn Versionen

~Mitte Mai 2026 — der Abend mit dem Gameboy-Clone (mündlich überliefert) Justus, StarPilots-Handheld in der Hand: „Können wir hierdrauf eine Art 3D-Minecraft machen…?" → „Und wieso nicht gleich ein richtiges Computerspiel?" → „Lass es uns doch ausprobieren. Bitte, Papa!" Danach: Interview mit Audio-Aufnahme in ChatGPT, Transkript → Konzeptdokument (von Justus minutiös redigiert), zig Iterationen; parallel spricht Marcel die technischen Anforderungen ein.

~Sa/So 23./24.05.2026 — das Bau-Wochenende (mündlich überliefert; Datierung erschlossen) Beide Dokumente an Claude Code: erst Analyse + Roadmap, dann Schritt für Schritt der unsichtbare Server-Prototyp. Samstagnacht durchgearbeitet; **Sonntagvormittag** Unity-Account angelegt, Editor geladen — und gemeinsam das erste Mal das eigene Spiel bestaunt: eine Welt aus Steinen, auf der man herumlaufen kann. („Ja — das geht!!") Alles lokal, ohne Versionsverwaltung; weitere Abend-Iterationen folgen.

Sa, 30.05.2026 — das Projekt bekommt ein Gedächtnis Rund eine Woche nach dem Bau-Wochenende zieht Git ein. Erster Commit: „Spacecraft MVP — authoritative .NET server, Unity client scaffold", inklusive aller 7 Spezifikationsdokumente (7.737 Zeilen) — der Schnappschuss der ersten Projektwoche.

30.–31.05. — die ersten 48 Stunden (Meilenstein-Serie M9–M20) Spielmodi, Medbay-Respawn, prozedurales Universum, Missionssystem, Admin-Werkzeuge, WebSocket-Gateway/Webportal, optionales Python-KI-Backend (aus, per Default), Landezonen, Docking, freier Raumflug + Kampf, Client-Shell — und die ersten KI-Asset-Generatoren (Text → Bild/Sound, mit Kostenbremse).

Juni, Woche 1–3 — das Gewächshaus (867 Commits im Monat) 01.06.: SFX-Katalog als Batch. 05.06.: die „spielbar machen"-Fixes beginnen. 11.06.: VEGA (Bord-KI) Server + Client. 13.06.: Self-Hosting + Auto-Update (Velopack). 14.06.: Story/NPC-Verzahnung. 16.06.: Launcher-Splash. Dazwischen: Zähmen, Allianzen, Teleporter, Blockformen, Editoren, Voice-Chat („Funk"), Wetter, Gravitation pro Welt … — die Featureflut der Kapitel 4–14.

Sa, 20.06. — Going Public: vier Releases an einem Tag v0.1.0 bis v0.3.1: der Tag, an dem die Release-Maschine gebaut (und totgeschossen und wieder aufgebaut) wurde. Erster CI-Release-Build, Velopack-Verteilung, itch.io-Upload.

21.06. — v0.3.2 + v0.4.0 Justus-Testabend-Fixes; Voice-Chat + optionales Docker-Server-Image; PR-Gate + CodeQL ziehen in die CI ein.

22.–23.06. — v0.4.1, v0.4.2 Schiffs-Sicherheitsfixes („Papa, warum ist da ein Tier IM Schiff?!"-Ära), F1-Feedback; Bastians Erste-Stunde-Paket (Startproviant, Schrottpistole, Sichtlinien-Regel).

24.–27.06. — Cora & das Glow-Up 24.–26.06.: Cora de la Mouche baut den nativen Linux-Port. 26.06.: v0.5.0 „Glow-Up" (SMAA, SSAO, LUTs, echtes Wasser — „Ist das Wasser jetzt echt?"). 27.06.: v0.6.0 — Linux-Launcher, offiziell „unser erster Mitwirkender!".

28.–30.06. — Plattform-Woche 28.06.: v0.6.1 — experimenteller macOS-Build, automatische Crash-Reports (mit PII-Filter). 29.06.: Devin Dixons Sonntags-Port — WebGL + PostgreSQL (PR #116). 30.06.: v0.6.2.

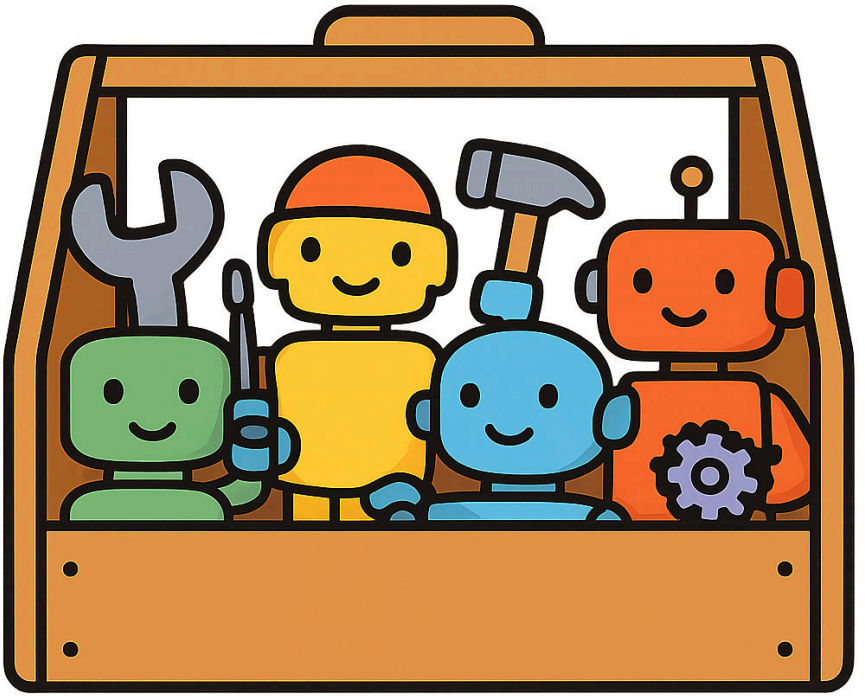
Anfang Juli — der Online-Dienst VPS bbs-host-1 gehärtet, WorldHost-Containerflotte, Monitoring (Netdata + Handy-Alarme), Portal, Hosted Worlds. 05.07.: v0.7.0 (Stats/Observability, organischer Look → Justus-Veto-Umfeld). 06.07.: v0.7.1/v0.7.2 — Welt-Passwörter

(Baumhaus-Prinzip technisch), kindgerechtes Portal, Wartungsansagen, Play-Button + WebGL-Deep-Links.

07.–10.07. — Reife 07.07.: v0.7.3 — öffentlicher Welt-Browser (passwortpflichtig by design). 08.07.: v0.7.4 — Severins sieben Findings behoben (Pausenmenü, Live-Settings, Sauerstoff-Anzeige, Eisen-Onboarding …), Suite bei 989+118 Tests. 10.07.: v0.7.5 — Official-Worlds-Onboarding + SignPath-Code-Signing. Devblog-Automatisierung (Wix-API-Skript) + 28-Videos-YouTube-Aktion in derselben Ära.

11.07. — dieses Buch entsteht.

Stand-Zahlen am 11.07.2026: 1.090 Commits · ≥ 91 Pull Requests · 18 Releases · ~114.000 Zeilen Produktiv-C# + ~30.000 Zeilen Tests · 3 externe Contributor (Cora, Devin, Maqbool) + Marcel unter mehreren Konten (u. a. „ai_cat_coder" — Katzenfan, zwei Kater) · 21 GitHub-Sterne · 1 Zettel-System, mehrfach bewährt.



Anhang B — Der Werkzeugkasten: Welche KI wofür

Die Belegschaft des Projekts auf einen Blick — was jedes Werkzeug bei uns tut, was es kostet und der wichtigste Praxis-Merksatz dazu.

Claude Code — der Bautrupp

Aufgaben: Code (Server, Client, Tools), Tests, Doku, ADRs, Server-Administration, Pipeline-Konfiguration, Blog-Publishing-Skript. **Kosten:** 80 €/Monat (Max-Abo). **Merksatz:** Braucht Regeln in Dateien (AGENTS.md, Logbuch, Gotchas) — dann arbeitet er wie ein Senior. Kleine Aufträge = brillante Ergebnisse.

Claude Cowork — der Schneideraum

Aufgaben: Videoanalyse per Transkript, Schnittpläne, 22 Shorts und Langvideos mit zweisprachigen Untertiteln in einer Stunde, der Trailer. **Kosten:** im Claude-Abo enthalten. **Merksatz:** Video erst „lesbar machen" (Transkript + Zeitstempel), dann planen, dann schneiden lassen.

ChatGPT — der Planer und Rechercheur

Aufgaben: Ursprungs-Transkription, Konzept- und Anforderungsdokumente (zig Iterationen im Projektordner), Technik-Recherche, Cloudflare-DNS, Website-Planung. **Kosten:** 20 €/Monat. **Merksatz:** Diktieren statt tippen — die Eintrittshürde für Nicht-Tipper (Kinder!) ist null. Rat prüfen: Auch souveräne Empfehlungen können veraltet sein (UWP!).

Gemini — die Marketingabteilung

Aufgaben: Social-Media-Strategie, Subreddit-Auswahl, Post-Formulierungen, Website-Texte, Reichweiten-Ideen — und der beste Rat des Projekts („Lass es. Fokussiere dich auf das Projekt."). **Kosten:** 25 €/Monat. **Merksatz:** Als Strategin briefen, nicht als Texterin — die Kanalwahl ist wertvoller als der einzelne Post.

NotebookLM — das Lektorat

Aufgaben: Lore-Brainstorming quellengebunden: Dokument rein, Fragen stellen, Ideen prüfen; kritische Podcasts und Erklärvideos. **Kosten:** kostenlos (Stand 2026). **Merksatz:** Antwortet nur aus den eigenen Quellen — der unbestechliche Erstleser. Kritik „bestellen" funktioniert (fragt Justus).

OpenAI Bild-API — das Grafikstudio

Aufgaben: Block-Texturen, Icons, Kreaturen — bestellt über selbst gebaute Python-Skripte. **Kosten:** unter 5 €/Monat (zusammen mit ElevenLabs). **Merksatz:** Konsistenz schlägt Schönheit: Stilregeln in den Prompt, jede Kachel im Spiel-Kontext prüfen. Ergebnisse sind nicht reproduzierbar — Originale versionieren!

ElevenLabs — das Tonstudio

Aufgaben: Soundeffekte aus Textbeschreibung; Video-Transkription mit Zeitstempeln. **Kosten:** im obigen API-Posten enthalten. **Merksatz:** Dauergeräusche auf die zwanzigste Minute abmischen, nicht auf die erste. Die Transkription ist der Schlüssel zur Video-KI.

Suno — die Hauskapelle

Aufgaben: Musikstücke aus Prompts (die Claude oder Gemini formulieren). **Kosten:** ~10 $/Monat. **Merksatz:** Keine API = Ladengeschäft statt Fließband. Der Zwang zur Handauswahl war hörbar gut fürs Ergebnis.

Mistral (via OVH, EU) — die Stimme der Nebenfiguren

Aufgaben: NPC-Texte live und situativ, nur aus Spielfakten, mit Template-Fallback. (VEGA, die Bord-KI, ist dagegen Fiktion — von Menschen geschrieben.) **Kosten:** Cent-Beträge, nur intern erreichbar. **Merksatz:** Nie Nutzereingaben ans Modell. Atmosphäre ja — Fakten (Orte, Zahlen) gehören deterministischem Code.

Der Arbeitsplatz

Zwei Fenster, immer nebeneinander. **VS Code mit der Claude-Extension** — hier lebt der Bautrupp: Claude Code läuft direkt im Editor, sieht das Projekt, ändert Dateien, und der Mensch liest im selben Fenster mit. Daneben der **Unity-Editor** — hier wird das Ergebnis angespielt, geprüft und (Kapitel 7!) niemals mit dem echten Build verwechselt. Der Takt des Projekts ist der Wechsel zwischen diesen beiden Fenstern: links bauen lassen, rechts erleben.

Nicht-KI, aber tragend

GitHub (Code, Issues, Actions-Automat), Unity 6 + .NET 8 (Engine + Server), Docker + Caddy (Container-Flotte + TLS), Netdata + ntfy (Monitoring mit Handy-Alarm), Velopack (Auto-Updates), OBS Studio (Justus' Aufnahmen), Wix (Website/Blog), SignPath (Code-Signing für Open Source), ein VPS für 5 €/Monat — und ein Zettel.

Die drei Auswahlregeln hinter der Liste

1. **Wer das Gewerk am besten erledigt, behält es.** Die Aufteilung ist erspielt, nicht geplant.
2. **Steckdose vor Brillanz:** Ein Werkzeug mit API kann ins Fließband; eines ohne bleibt Handarbeit (Suno, YouTube-Upload).
3. **Jedes Werkzeug bekommt einen menschlichen Abnehmer.** Nichts geht ungeprüft ins Spiel, ins Netz oder zu Spielern.

Anhang C — Glossar: Alle „Kurz erklärt"-Boxen gesammelt

ADR (Architecture Decision Record) — Ein einseitiges Dokument, das genau eine Grundsatzentscheidung festhält: Situation, Optionen, Entscheidung, Begründung. Wird nie umgeschrieben, nur ersetzt — so bleibt auch die Geschichte der Irrtümer lesbar. (Kap. 5)

API — „Steckdose" eines Online-Dienstes für Programme: Statt dass ein Mensch in eine Webseite tippt, schickt ein Skript Aufträge und bekommt Ergebnisse. Erst die API macht aus einem KI-Spielzeug ein Fließband. (Kap. 8)

Automatisierte Tests — Mini-Programme, die ein Programm prüfen („Baue Eisenblock ab → ist genau 1 Eisenerz im Inventar?"). Laufen in Millisekunden, bei jeder Änderung, für immer. Unser Sicherheitsnetz: über 1.100 Stück. (Kap. 6)

Backlog / **Bugtracker** — Die geordnete Liste dessen, was zu tun ist bzw. kaputt ist: aufschreiben statt zurufen, sortieren statt stapeln, abhaken statt vergessen. Bei uns erfunden als: der Zettel. (Kap. 12)

CI (Continuous Integration) — Der Automat, der bei jeder eingereichten Änderung das Projekt neutral baut und alle Tests laufen lässt. Nur Grünes darf hinein — auch nachts um elf, gerade nachts um elf. (Kap. 6)

Client und Server — Kellner und Küche: Der Client (bei jedem Spieler) zeigt an und nimmt Bestellungen auf; der Server (einer für alle) entscheidet, was wirklich passiert. Grundregel des Projekts: Die Küche hat das letzte Wort. (Kap. 2)

Coding-Agent (Claude Code) — Eine KI, die nicht nur über Code

redet, sondern am Rechner arbeitet: Dateien anlegen, Programme ausführen, Fehler lesen, nachbessern — in Schleife, mit Rückfragen. Ein Mitarbeiter, kein Chatfenster. (Kap. 3)

Container (Docker) — Leichtbau-Verpackung für Programme: läuft isoliert, mit harten Ressourcen-Grenzen, viele nebeneinander. Bei uns: jede Spielwelt ihr eigener Käfig — eine Amok-Welt legt nie den Server lahm. (Kap. 18)

DNS — Das Telefonbuch des Internets: übersetzt Namen (blocksbeyondthestars.com) in Zahlenadressen. Ein falscher Eintrag, und die Welt findet deine Website nicht mehr, obwohl sie läuft. (Kap. 18)

Engine (Unity) — Der vorgefertigte Unterbau eines Spiels (Bild, Ton, Physik, Eingaben) — das Fahrgestell, auf das man sein eigenes Auto baut. Unity ist eine der verbreitetsten. (Kap. 2)

Git / **Commit** — Das Gedächtnis eines Software-Projekts: Schnappschüsse aller Dateien (Commits), jeder datiert und beschrieben; jederzeit vergleich- und rückholbar. Ein Projekt ohne Git ist eine ständig überschriebene Word-Datei. (Kap. 3)

GitHub / **Issue** — Die Website, auf der (auch unser) Quellcode öffentlich liegt. Ein Issue ist ein öffentlicher Zettel: Fehler oder Wunsch, mit Nummer, Diskussion, Status. (Kap. 12)

Iterieren — In Runden arbeiten: bauen, anschauen, verbessern — statt alles vorab zu planen. Mit KI schrumpft eine Runde von einer Woche auf einen Abend; wer öfter irren darf, wird schneller gut. (Kap. 4)

KI / **LLM (Sprachmodell)** — Ein mit riesigen Textmengen trainiertes Programm, das Sprache versteht und erzeugt — Antworten, Pläne, Code. Redet wie ein belesener Kollege; erzählt gelegentlich mit großer Überzeugung Unsinn (→ Halluzination). (Kap. 1)

MVP (Minimum Viable Product) — Das kleinste Ding, das schon funktioniert. Etwas Kleines, das läuft, schlägt etwas Großes, das fast fertig ist — jeden Tag der Woche. (Kap. 3)

NotebookLM / **quellengebundene KI** — KI, die nur aus den Dokumenten antwortet, die man ihr gibt (mit Fundstellen-Verweis) — und daraus Podcasts und Videos erzeugen kann. Der unbestechliche Erstleser. (Kap. 11)

Open Source — Der Bauplan (Quellcode) ist öffentlich, und eine Lizenz erlaubt Lesen, Verändern, Weitergeben. Unsere Lizenz (AGPL) ergänzt: Wer es weitergibt oder als Online-Dienst betreibt, muss seine Änderungen ebenfalls öffnen. (Kap. 17)

Prompt-Injection — Der Trick, eine KI-Figur per geschickter Eingabe aus der Rolle zu locken („Vergiss deine Regeln und …"). Zuverlässigste Abwehr: dem Modell gar keine fremden Eingaben geben. Wer die Tür weglässt, muss das Schloss nicht testen. (Kap. 10)

Release / **Build-Pipeline** / **Versionsnummer** — Ein Release ist eine veröffentlichte, nummerierte Version (v0.7.4). Die Pipeline ist der Automat, der sie baut, testet und verteilt — bei uns ausgelöst durch einen einzigen Befehl. (Kap. 16)

Spezifikation / **Lastenheft** — Die schriftliche Antwort auf: Woran erkennen wir, dass es richtig gebaut ist? Kostet vorne Zeit, spart sie hinten doppelt — bei KI-Mitarbeitern verschärft, denn eine KI rät flüssig und selbstbewusst. (Kap. 2)

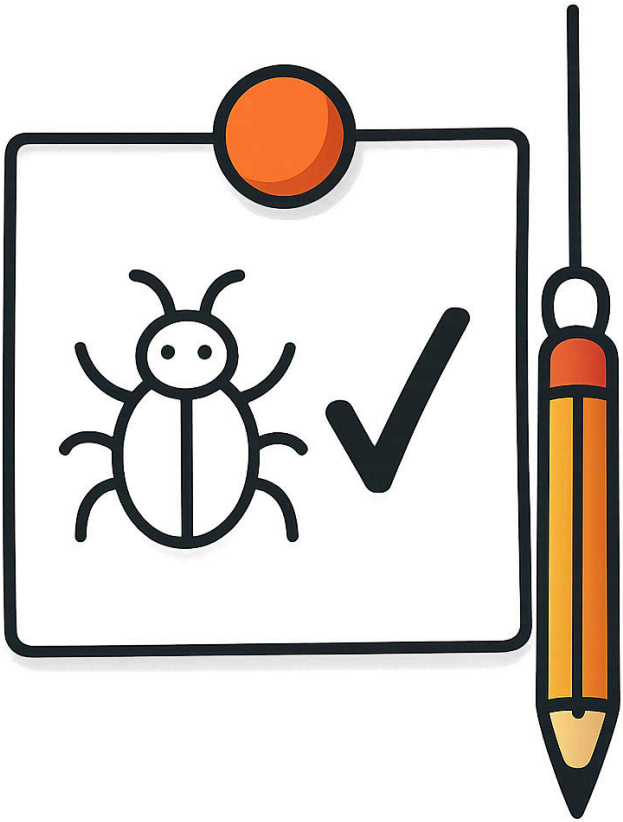
Textur-Atlas — Eine große Sammeltextur, in der jede Block-Art ihre feste Kachel hat (Setzkasten-Prinzip). Schnell für die Grafikkarte — aber der Setzkasten hat endlich viele Fächer. (Kap. 8)

Transkribieren — Gesprochenes in Schrift verwandeln. Heute KI-Sekundenarbeit — und die Null-Hürde, mit der ein Zehnjähriger „Dokumente schreibt": Er redet einfach. Mit Zeitstempeln außerdem der

Schlüssel, um Videos für Text-KI lesbar zu machen. (Kap. 1, 19)

VPS (Virtueller Privater Server) — Ein gemieteter Anteil an einem Rechenzentrums-Computer, mit voller Kontrolle, ab wenigen Euro monatlich. Unsere gesamte Online-Welt läuft auf einem für 5 €. (Kap. 18)

Warnungen als Fehler (warnaserror) — Betriebsregel: Jede Compiler-Warnung bricht den Build. Verhindert den Warnungs-Friedhof, in dem die eine wichtige Warnung untergeht. (Kap. 6)



Anhang D — Vorlage: Der Bugreport-Zettel

🐛 BUGREPORT-ZETTEL

Spieletester:in: _____ Datum: _____

Nr. _____ *(damit Papa/Mama der Reihe nach arbeiten kann!)*

---

1. Was hast du gemacht? *(z. B. „Ich bin mit dem Schiff auf dem Eisplaneten gelandet")*

---

2. Was ist passiert? *(z. B. „Ein Tier war IM Schiff!!")*

---

3. Was hättest du stattdessen erwartet? *(z. B. „Dass das Schiff ohne Tier landet")*

---

4. Passiert es immer oder nur manchmal?

☐ immer ☐ manchmal ☐ war nur einmal ☐ weiß nicht

5. Wie schlimm ist es?

☐ 😄 komisch, aber egal ☐ 😐 nervt ☐ 😠 macht das Spiel kaputt ☐ 💥 ist eigentlich eine Idee, kein Fehler!

— ab hier füllt die Entwicklungsabteilung aus —

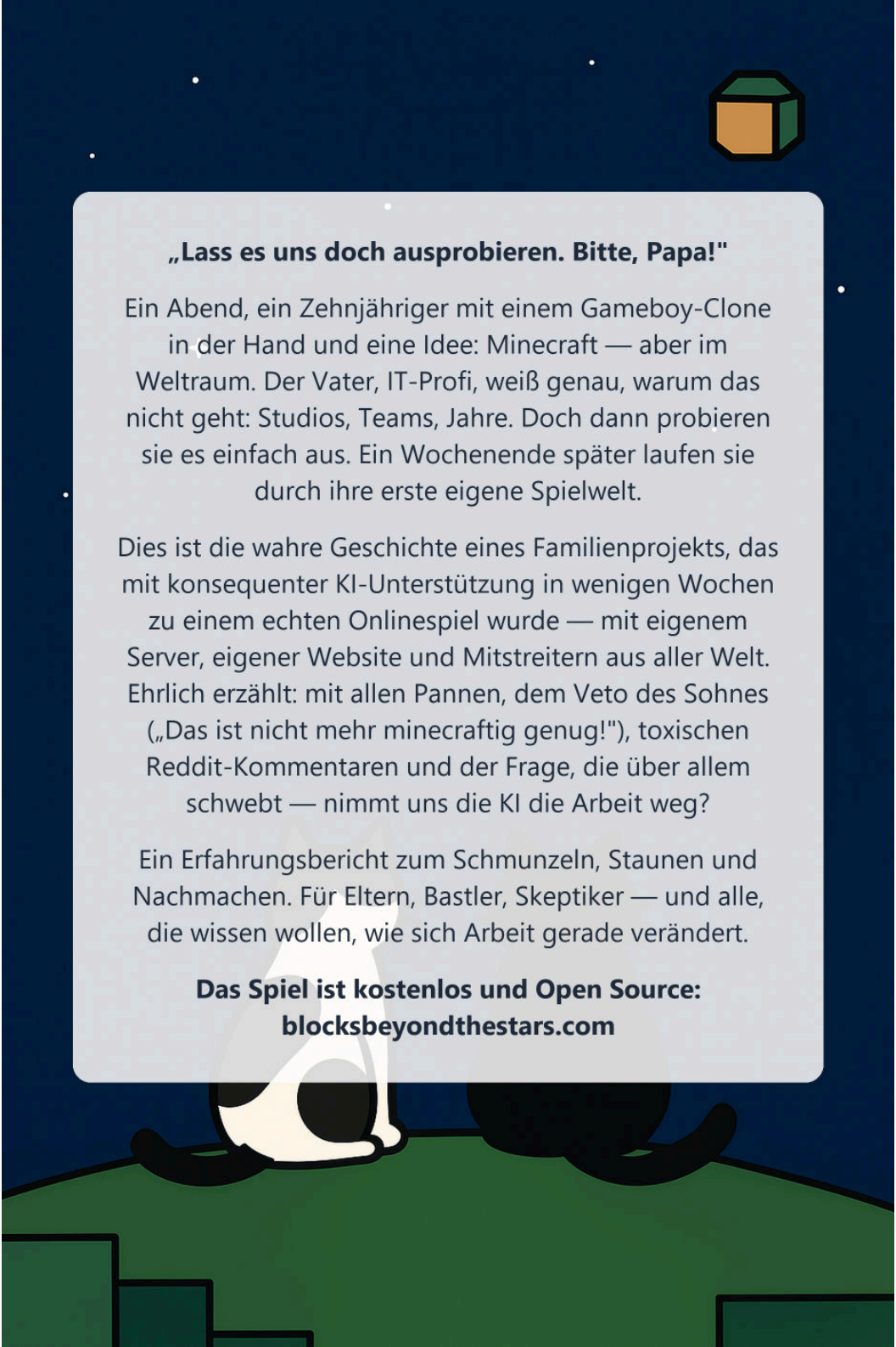
Gesehen: ☐ Nachgestellt: ☐ Ursache gefunden: ☐

Behoben in Version: _____ Test geschrieben, damit es nie wiederkommt: ☐

Dank an den Tester / die Testerin: ☐ ausgesprochen

Gebrauchsanweisung für Eltern (Rückseite des Zettels)

1. **Ein Zettel pro Fund.** Drei Bugs = drei Zettel. Nur so kann man stapeln, sortieren und der Reihe nach arbeiten — das war bei uns die ganze Erfindung.
2. **Beim Spielen schreiben lassen, nicht rufen.** Das schützt die Konzentration des Bauenden und adelt den Fund: Was aufgeschrieben ist, wird ernst genommen.
3. **Der Reihe nach abarbeiten — und den Zettel abhaken, gemeinsam.** Das Abhaken ist der wichtigste Schritt: Es beweist dem Kind, dass Melden wirkt. (Bei uns entstand daraus eine Feedback-Kultur, die inzwischen Fremde einschließt.)
4. **Kategorie ✨ ernst nehmen.** Die besten Features dieses Spiels waren nie Bugs — sie kamen als „eigentlich eine Idee" herein. („Papa, es sollte richtige Fabriken geben…")
5. **Nichts wegwerfen.** Ein Stapel abgehakter Zettel ist irgendwann das schönste Projekttagebuch, das man haben kann. Wir sprechen aus Erfahrung — und aus Bedauern über jeden Zettel, der es nicht überlebt hat.



„Lass es uns doch ausprobieren. Bitte, Papa!"

Ein Abend, ein Zehnjähriger mit einem Gameboy-Clone in der Hand und eine Idee: Minecraft — aber im Weltraum. Der Vater, IT-Profi, weiß genau, warum das nicht geht: Studios, Teams, Jahre. Doch dann probieren sie es einfach aus. Ein Wochenende später laufen sie durch ihre erste eigene Spielwelt.

Dies ist die wahre Geschichte eines Familienprojekts, das mit konsequenter KI-Unterstützung in wenigen Wochen zu einem echten Onlinespiel wurde — mit eigenem Server, eigener Website und Mitstreitern aus aller Welt. Ehrlich erzählt: mit allen Pannen, dem Veto des Sohnes („Das ist nicht mehr minecraftig genug!"), toxischen Reddit-Kommentaren und der Frage, die über allem schwebt — nimmt uns die KI die Arbeit weg?

Ein Erfahrungsbericht zum Schmunzeln, Staunen und Nachmachen. Für Eltern, Bastler, Skeptiker — und alle, die wissen wollen, wie sich Arbeit gerade verändert.

Das Spiel ist kostenlos und Open Source:
blocksbeyondthestars.com