Computer Science Department

Software Engineering & Business Analysis

Bachelor's Thesis

BlockMate

An AI-powered screen-time application that turns unintentional app usage into conscious choice

Danylo Gavrylovskyi

Supervisor
KSE, Ihor Pysmennyi

Submission date
16 June 2026

Academic Integrity Statement

I, undersigned, hereby declare that this capstone project is the result of my own work.

- All ideas, data, figures and text from other authors have been clearly cited and listed in the bibliography.
- No part of this project has been submitted previously for academic credit in this or any other institution.
- All code, diagrams, and third-party materials are either my original work or are used with permission and properly referenced.
- I have not engaged in plagiarism or any form of academic dishonesty.
- Any assistance received (e.g. from peers, tutors, or online forums) is acknowledged in the acknowledgements section.

I understand that failure to comply with these declarations constitutes academic misconduct and may lead to disciplinary action.

Place, date Kyiv, 16.06.2026

Signature 

Contents

Acknowledgements

Throughout these four years, two people shaped me more than anyone else.

Vadym Yaremenko's courses on programming paradigms, client–server communication, and object-oriented programming, all taught in C and C++, made me fall in love with low-level engineering and gave me the desire to understand what really happens under the hood and how things actually work, something I keep coming back to in interviews and every time I sit down to design a piece of code or an architecture. He showed me the kind of engineering soul I want to grow into. I went to his lectures and practices and did the homework with great passion, and even now there are many moments where I remember something Vadym taught me, and I always remember it with a smile.

Ihor Pysmennyi, my capstone mentor, changed my life in a few days. When I saw that he was running an MVP camp, I almost didn't go. But during those four days I found a project, co-founders, and friends. With that team and BlockMate, we won the MoreTech Winner certificate from Genesis and a guaranteed place in the KSE Startup Incubator. Everything that has happened since started from there: Universe Group, the work I love, the startup I am building. I had always had a passion for startups but never knew how to start one; that camp is what gave me the opening. On top of that, I loved Ihor's courses themselves. Cloud Architecture is, I think, my all-time favorite. He made a huge impact through his hard work and his personality, and I love the way he taught, with jokes, real-world examples, and the kind of expertise where you listen to every word in excitement at how he can explain hard things in such a simple way. His YouTube course recordings are absolutely brilliant on their own; I watched everything end to end, sometimes twice, and they have been a huge boost for me as a solution architect. I will always remember the homework to build a distributed database: the sharding, the partitioning, how all of that works. That is really my passion, and it is also where I understood that my favorite thing is to think basically like an architect.

I also want to thank my brilliant co-founders and friends, Oleh Hykavyy and Dmytro Hrebeniuk. Finding them felt like a dream. We share the same excitement for startups, for that "zero to one" Silicon Valley feeling: building something from nothing, touching every part of it, learning through the failures and the small wins. Between work, study, and the startup I did not have a lot of time outside of that, but every hangout, hackathon, and meetup we did together was brilliant. In the short time we have known each other, they have changed my mindset and changed me as a person.

And, of course, my family. The opportunity to study at KSE was made possible by them, and without their support I would not be who I am now. They have backed every one of my passions, including the product-engineering obsession these four years built in me, where I came out not just a software engineer but someone genuinely in love with creating products. Thank you also to KSE itself for giving us this kind of education in Ukraine. The honest truth of these four years was also the math, and the fight for every 60+ grade; after surviving that, I understand I can achieve anything I set my mind to.

Abstract

Most time spent on smartphones is not intentional. People reach for an app out of stress, boredom, or habit, and the tools meant to help, namely Apple's Screen Time, friction apps such as Opal, and hard blockers, all share the same weakness: they add an obstacle in front of the app while leaving the underlying craving untouched, so the brain learns to move through the obstacle on autopilot and the user eventually disables the tool. This thesis proposes and builds an alternative. BlockMate is an AI-powered iOS application that, instead of adding friction, opens a short conversation with an AI coach when a blocked app is launched, turning an automatic reach into a conscious decision about whether the use is genuinely intended.

The system was designed and implemented as a serverless backend on Firebase Cloud Functions with an iOS Screen Time client. Its core is a two-stage AI pipeline, an intent classifier followed by a coaching responder, combined with a strategy selector and a per-user "journey" object that lets the coach adapt its tone to who the user is and what kind of moment they are in. Each negotiation resolves to one of three outcomes: approved, rejected, or a request for more information. To keep a probabilistic, conversational AI under engineering control, a dedicated evaluation framework scores the pipeline against a fixed scenario set, both deterministically and with an LLM-as-judge, so that prompt changes are shipped against measured baselines rather than intuition.

The implementation was validated against seven core functional and non-functional requirements, of which six are satisfied; the one exception is a three-second responsiveness target, missed because the pipeline makes two sequential model calls, and reported transparently. On a fixed scenario set the AI selects the correct verdict in 97.1% of single-turn cases and routes 100% of simulated multi-turn conversations to acceptable outcomes, with manipulation resistance and voice consistency its strongest measured qualities. Beyond the test suite, the loop is confirmed in production across 537 real negotiations from a small bilingual beta cohort, including conversations in which users reach the intended moment of self-awareness on their own. The work demonstrates that a context-aware, conversational approach to screen-time management can be built and rigorously evaluated by a small team, and identifies user acquisition and longer-term retention evidence as the decisive next steps.

Keywords:

screen-time management, *digital wellbeing*, *behaviour change*, *large language models*, *AI coaching*, *iOS*, *serverless architecture*

1 Introduction

How much time do you actually spend on your phone, and how much of it is really intentional? I once caught myself with a screen time of more than ten hours in a single day. That is totally crazy.

We live in an era of cheap dopamine. Short videos like TikTok, YouTube Shorts, and Instagram Reels, and the algorithms behind them, are built by huge teams of people whose only goal is to steal your time. They are good at it. You look at how much of your day is gone, and you can list everything you could have done with it instead: go to the mountains, go fishing, see your friends, pick up a guitar, play football, learn a new skill. Real things.

But that is not actually the problem we are trying to solve. The real problem is that we spend the time unconsciously. We reach for the phone the moment we are stressed or bored, and the brain, wired to take the path of least effort to the next bit of dopamine, fills the gap with whatever is closest. Half the time I open my phone, I open an app, scroll for a minute, and then think to myself: why did I even open this? Or you are deep in something, your code is building or some long task is loading, and you scroll while you wait. Then you look up, the build finished a long time ago, and the train of thought you needed is gone.

There are two kinds of phone use. One is intentional: you open Instagram to watch a story from the friend who just got back from the mountains, you watch it, and you put the phone down. You knew what you came for and you knew when you were done. The other is the autopilot kind: you open the phone with no idea what you are going to do, you scroll a little, you check stories, you flick to TikTok for two videos, and thirty minutes later you are still there. The trend of short videos has genuinely broken our dopamine system.

A lot of us are aware of this and trying to do something about it. The first thing most people reach for is Apple's built-in Screen Time. You set a one-hour limit on TikTok, and at the end of that hour the OS shows you a friendly screen with a blue *Continue* button. The brain has no problem with that. The button is right there, it costs no effort, and pressing it does not even register as a decision. Most people I know set these timers, forget they ever set them, and hit *Continue* on autopilot.

There is also a market of apps explicitly built for this problem, and they fall into two camps. The first is friction apps such as Opal, OneSec, and BePresent, which insert a small obstacle in front of the app you are trying to open. Breathe for one minute. Do five push-ups. Wait ten seconds. The brain is too smart for this. It learns the rhythm of the obstacle and starts moving through it on autopilot the same way it presses *Continue*. The second camp is hard blockers, which essentially turn your phone into a brick for the apps you have marked. They work, but they make you feel like you have lost control over your own device, and that feeling is exactly what makes people uninstall the blocker a week later. There is even a number from the industry: about ninety percent of users stop noticing the effect of friction apps after some time. That is the core failure. The friction apps and the

hard blockers are not fighting the right target. They are adding obstacles in front of the app, but the craving, the actual thing pulling you to the phone, is left alone.

That is the gap BlockMate is built to fill. BlockMate is your AI bro that replaces autopilot with a conscious decision. The core move is not to put another obstacle in front of the app, but to talk to the craving directly.

How it works in one paragraph. You pick the apps you want to control. When you tap one of them, you do not see the app; you see a shield[1]. To get past the shield, you have to come to BlockMate, where an AI coach starts a conversation with you in plain words: why do you want to open this app right now? You answer in your own words, and the AI either lets you in for a while, asks you a follow-up to understand better, or holds the line and gives you a reason to put the phone down. The AI is not a gatekeeper. It is closer to a friend who picks up on what you are feeling. If you say you had a hard interview today and you are stressed, BlockMate will read that you are reaching for the app to escape the stress, and the conversation will be different from one with a clearly productive request. The trick is not the verdict. The trick is that you have to articulate the reason at all. Once you have asked yourself the question, you cannot un-ask it, and the next time you reach for the phone, the question shows up before the app does. That is the loop the rest of this thesis is about.

How this idea showed up is, for me, one of the best stories of my life. The starting point was an MVP camp: four days where you joined a team and shipped something. I joined Oleh and Dmytro, and they started telling me about an app blocker. I did not get it at first. I thought it was just another hard blocker. They tried to explain that the AI would talk to you and validate the decision, and the version of that I was hearing in my head was, "you say you want to message your boss, the AI lets you, end of story." Obvious and not particularly interesting.

It took me longer to see what they were really saying. BlockMate is not a gate that classifies your reason as approved or rejected. It sits somewhere between an app blocker and a psychologist, closer to a friend who helps you reduce your screen time. Once that clicked, the team clicked too.

I was the technical person on the team, and I started with a real constraint. We were targeting the iPhone, but Apple's Screen Time and FamilyControls APIs only exist inside Apple's ecosystem, and I was on a Windows laptop. There was no way to build an iOS app in four days, so we pivoted in scope: I built a Chrome extension that did the same job for websites, with the AI conversation in front of every blocked URL. That demo won the Genesis MoreTech Winner award, which came with a guaranteed place in the KSE Startup Incubator, and we joined the UCU Startup Accelerator in parallel. The Accelerator's weekly cadence of customer-development interviews is where most of the next chapter comes from. Once I finally had a Mac, we rebuilt BlockMate in Swift as the iPhone app it was always meant to be. The fuller story of those early months, including the hackathon demo and how the project came together around the two programs, is in the appendix.

From the customer-development work, three product hypotheses crystallised.

[1] In Apple's Screen Time frameworks, the *shield* is the full-screen cover that iOS draws over a blocked app instead of opening it. BlockMate replaces Apple's default grey shield with its own screen.

H1. Existing solutions are either too soft or too hard, and they get worse with time, because fixed friction habituates while a contextual conversation, new every time, does not.

H2. The install–uninstall cycle can be broken by keeping the app on the phone but gating each entry through the conversation, instead of forcing the all-or-nothing choice between deletion and reinstall.

H3. Surfacing the underlying reason builds awareness, not just barriers: if the system makes the real motive behind an automatic reach visible, awareness rather than the lock becomes the lasting outcome.

A fourth question runs alongside these and is engineering rather than product: can such a system be built and evaluated systematically by a small team? That question is what gives the rest of the thesis its shape. The design and implementation chapters cover the architecture and the build, and the validation chapter covers the evaluation framework that turns prompt iteration from a vibes-driven exercise into a measurable one.

The thesis is organised as follows. Chapter 2 covers the research: customer-development interviews, JTBD analysis, the competitor landscape, and the verdict on each hypothesis above. Chapter 3 covers the product principle and the system design, with particular focus on the AI pipeline. Chapter 4 covers what was actually built, including the evolution from a Chrome extension to a Firebase Cloud Functions backend with an iOS Screen Time client. Chapter 5 covers validation: the evaluation framework and the small live cohort. Chapter 6 closes the loop on the hypotheses and looks at what comes next.

2 Research

This chapter is the research that sits underneath BlockMate. It moves through four things: the scale of the problem in numbers, the existing solutions on the market and where each of them breaks, our own customer development of ten in-depth interviews and what they told us, and finally a grading of the three hypotheses from the introduction against that evidence.

Contents

2.1 Problem space

Phone addiction is not really a surprise anymore. Industry surveys put average daily screen time for adults somewhere in the four-to-seven-hour range [1]. From the interviews and from the people I have talked to outside the formal cohort, friends, colleagues, people who have onboarded into BlockMate, the number lives closer to the higher end. Six or seven hours is normal. Ten-plus hours is not unheard of, and that is wild.

Roughly two segments dominate. Students sit at six to seven hours daily. Adult professionals, the ones who actually have the disposable income to pay for a fix, sit closer to five or six. The shape of that time has changed too. The bulk of it is no longer reading or messaging. It is short-form video. TikTok, Instagram Reels, and YouTube Shorts have rebuilt the default mode of consumption, and the algorithms behind them are tuned by very large teams whose only KPI is time spent in-app. It is not only short video either: long-form YouTube and the news loop sit alongside it, and several interviewees described their main addiction as the news, not TikTok.

The cheap-dopamine pattern is broader than scrolling and people are aware of it. Several interviewees described actively fighting back: switching the phone to grayscale to make it visually less rewarding, doing periodic digital detoxes by deleting apps and reinstalling them a week later, aggressively pruning notifications because the apps steal not just time but focus.

The market that has appeared in response is real but small relative to the problem. The broad digital-wellbeing market, which folds in wearables, wellness and sleep apps, and screen-time tools, was valued at roughly $16B in 2025 and is projected to reach about $87B by 2033 at a 23% CAGR [2]. But the slice that competes directly with BlockMate is far smaller than that headline suggests. Today's top players are still small: Opal, the category leader, reached around $10M ARR with a team of eleven [3], and app-intelligence estimates put it near 200k monthly downloads and $400k of monthly revenue [4]. OneSec is bootstrapped. The combined revenue of all direct competitors in 2025 is approximately $45M, which we calculated by adding up Sensor Tower's per-app revenue estimates [5], [6] and reported in our investor deck [7]: a small total for a problem this widespread. The good news for a new entrant is that the category is growing fast and even its leader has no real moat. Opal is ahead on revenue and brand, but that lead rests on design and marketing, not on anything that locks users in or keeps competitors out. It is not yet a blue ocean, but it is a red ocean whose leader could still be overtaken.

The reason the category is small relative to the problem is habituation. The most thorough study of this to date followed more than a thousand real users of the friction app one sec for an average of thirteen weeks, and found that the longer people used it, the less often they let the friction stop them: the share of attempts they abandoned fell steadily over time, so they pushed through into the target app on a growing fraction of openings [8]. To be fair to the tool, the total number of attempts also dropped over those weeks, and the authors read that as people becoming more intentional. But the friction's power to stop any single reach clearly faded with use, and that fading is the habituation that decides whether someone keeps the tool. People install these tools, use them for a week or two, and then either uninstall them or develop a habit of bypassing them on autopilot.

Retention, not acquisition, is therefore the metric that matters here: these companies are not failing to find users, they are failing to keep them. And it is not for lack of demand, since every interview we ran surfaced screen-time concerns unprompted; it is because the existing mechanisms get habituated. That is the gap the rest of this chapter motivates.

2.2 Existing solutions and where they break

The market splits cleanly into three groups: a built-in OS option, friction apps, and hard blockers.

2.2.1 Apple Screen Time

Apple's built-in feature is the entry point for most users. Outside of the formal customer-development interviews, almost every person I have talked to about phone overuse in the past few months has mentioned Screen Time first; it is the universal starting point. You set a daily limit on a chosen app, and once the limit is reached, the OS shows a screen with a friendly blue *Continue* button. The button is the failure mode. It is exactly one tap, no accountability, and after a few uses the brain stops registering it as a decision at all. The first time you set the limit, you remember why you did it. After a week, the press is automatic, sub-second, and you are scrolling before the thought of "should I" has formed. Of the ten interviewees in our customer-development round, four had used Apple Screen Time at some point and all four described the same pattern.

Veronika put it most plainly: setting a daily limit only works for her on the time-of-day mode (no use after 21:00), not on the duration mode, because the *Continue* button on duration is "too easy to press without thinking."

2.2.2 Friction apps

Opal is the category leader and the example most worth a close look. It works by inserting a small obstacle in front of the blocked app: a five-second wait, a breathing exercise, or a "do you really need this" prompt before letting the user through. Its strength is the polished UI and the gamification — the gem economy, the streaks, the leaderboards. It is honestly one of the best-designed mobile apps I have used, and worth learning from aesthetically even while disagreeing with the core mechanism.

The weakness is that the friction is fixed. After a week of repetitions, the brain learns the rhythm and starts auto-clicking through the same way it auto-clicks Apple's *Continue*. "Breathe for one minute" stops being friction once you have learned to wait one minute thirty times in a row.

The most striking instance from our interviews was Marichka, who has Opal installed (her boyfriend put it there for her). She described automatically clicking the maximum 15-minute override every time the friction screen appears: "Opal is so friendly to me that it almost doesn't restrict me at all." The same pattern shows up at work: a colleague who used Opal deleted it entirely, because he needed Instagram and TikTok unblocked to make marketing content for the company. Opal's answer to a real, recurring work need was binary, stay blocked or be uninstalled, and he chose to uninstall. That is the second BlockMate hypothesis in action: a fixed-friction tool cannot survive a user who legitimately needs the app sometimes.

OneSec uses a similar friction mechanism: opening a target app triggers a short pause and a breathing prompt, with an explicit option to close the app instead of continuing. On iOS it has a practical weakness in setup, which runs through Apple's Shortcuts app, one automation per blocked app, and is fiddly to configure (on older iOS versions it also fired a notification every time it ran). ClearSpace pushes the friction harder, with physical push-ups, verified by the camera and an on-device model, as the unlock requirement, and ScreenZen leans on regular reminders ("you've been on this for 10 minutes"). All of these work, for some people, for some time. None of them solves the underlying problem that contextual *meaning* is missing from the friction. A wall is the same wall every day, and the brain learns to climb it.

That is the framing shift BlockMate is built on. Instead of a wall, we want a friend. Something you negotiate with, talk to, and that responds differently every time depending on who you are and what kind of day you are having. Same effect, the user does not get into the app on autopilot, but a different mechanism, and one the brain cannot route around the same way.

2.2.3 Hard blockers

The other end of the market is hard blockers like AppBlock and Freedom on iOS, plus various extensions on macOS. They work by genuinely making the app inaccessible during a scheduled window. Bohdan in our cohort uses Opal in its strict mode plus browser plugins on his laptop, and reports the system holds. The trade-off is the feeling. Hard blockers turn the device into a brick during the blocked window, and most interviewees described the resulting feeling, "digital prison" or "being controlled by an app I installed myself", as exactly the reason they uninstall it within a week or two. Robert and Polina both went through the install-then-delete cycle multiple times.

The marketing of hard blockers is genuinely good. The push-up unlock and the strict-mode storytelling are great brand moves and clearly work for performance marketing. The mechanism just does not match the goal for most users. Most people do not want their phone taken away. They want help using it more deliberately.

2.2.4 Where this leaves BlockMate

BlockMate's positioning across these three clusters is captured in a one-line bet: contextual friction does not habituate the way fixed friction does, because every interaction is genuinely new. A breathing exercise is the same exercise every time. A conversation is not the same conversation every time. That is the core differentiator versus Opal and OneSec. Versus hard blockers, BlockMate keeps the user in control. The AI can grant access; the user just has to articulate why. That removes the "digital prison" feeling that drives uninstalls.

The deliberate consequence of this design is that BlockMate is not trying to take your phone away. If you have had a hard day, you walk through the conversation, you can articulate why ten minutes of TikTok is the right call right now, and the AI lets you in. The point is not the lock. The point is that the decision was conscious. That difference is also what makes BlockMate flexible across very different relationships with the phone. Within our own founding team, Oleh's goal is closer to digital abstinence; mine is closer

to deliberate moderation. The same product serves both, because the AI is calibrated to each user's history rather than to one fixed target screen time.

One honest concession before moving on. BlockMate cannot stop a user who is actively trying to cheat the system. You can lie to it about your reason. You can articulate a fake intent and the AI will, sometimes, grant it. We did not build defences against that, and we do not really intend to. The product is for people who actually want to reduce their screen time, not for people who want the illusion of having tried.

It is worth being honest about where BlockMate is exposed, not just where it is strong. The weaknesses are real: we have limited in-house AI-research depth compared with a specialist team, none of us can give BlockMate our full attention since we are all studying and holding down full-time jobs alongside it, and the core mechanism, an AI conversation in front of a blocked app, is copyable in principle. The matching threat is the obvious one. The day a major incumbent ships its own AI-mediated flow, the surface-level differentiation closes, and capital for marketing is tight right now, especially for a Ukrainian-incorporated startup.

So the two real points of differentiation today are AI as the core friction mechanism rather than a feature bolted on, and keeping the app available rather than blocking it outright, which addresses the install–uninstall cycle directly. Those alone are not a long-term moat. The design chapter covers the deeper architectural choices, the contextual journey state per user, the rule-based coaching strategy selector, and the evaluation framework as an ongoing quality gate, that are meant to make the product harder to copy in practice than it sounds in principle.

2.3 Customer development

2.3.1 Method

Our customer development consisted of ten in-depth interviews that we ran together as a founding team. Each interview lasted between forty minutes and one hour, was held in Ukrainian, and was loosely scripted around twelve open-ended questions covering the interviewee's typical phone usage, the apps and contexts that drive overuse, prior attempts at reduction, and current screen-time pain points. Interviewees were recruited through KSE and UCU networks, supplemented by Slack channels open to students. We deliberately did not target paying users from outside these communities; the goal was to validate the core problem and refine the ICP, not to test pricing.

For each interview we recorded structured fields: profile, screen-time stats, triggers, prior attempts, what worked and what did not, external help, what the user feels they lose to phone overuse, the underlying job-to-be-done, the quotes that stood out, and a segment classification marking how closely each person matched our ideal customer profile: a partial fit, or outside it. Turning those raw notes into the structured fields, the segment classification, and the jobs-to-be-done synthesis is the analysis that runs through this chapter, and it is what directly shaped the product decisions that follow. The full cohort, translated and condensed, is summarised in Table 7 in the appendix.

2.3.2 Ideal customer profile

The interviews converged on an ICP with five characteristics:

1. Significant daily screen exposure that the person already considers excessive. The raw hours vary widely (from a focused 1.5 hours of phone use with 7 to 9 hours on a laptop, up to 6 to 7 hours on the phone alone); the common thread is not a fixed number but the explicit awareness that the time spent is too much.
2. At least one prior attempt to reduce screen time, ranging from Apple's built-in Screen Time and friction apps (Opal, OneSec, ScreenZen) to deletion-and-reinstall cycles or willpower.
3. None of the prior attempts has produced a stable result. Either the barrier was bypassed automatically, or the user gave up within a week or two.
4. An active user of consumer AI tools (ChatGPT, Gemini, Claude) in daily life, and therefore open to interacting with AI as something other than a search interface.
5. A growing willingness to pay for tools that improve daily life, even if actual pay tolerance in our student cohort was low (only one interviewee, Bohdan, paid anything at all, about $0.5 a month for One Sec).

Two adjacent segments showed up repeatedly, and we deliberately set them aside. The first is users who do not yet acknowledge the problem and have made no attempts to address it (Mariana and Lia in the cohort). The second is users who have solved the problem with willpower plus the built-in Screen Time, and view a third-party blocker as a sign of weakness (Mykhailo). These users are not in the buying market for BlockMate today and we did not invest in convincing them of the problem.

2.3.3 Segment distribution

Of the ten interviewees, five are partial fits to the ICP (Veronika, Marichka, Anna, Bohdan, Robert) and five fall outside it (Mariana, Mykhailo, Lia, Polina, Polina 2). None scored as a full fit. Even within an audience of motivated, tech-literate students, only half are positioned to buy a tool like BlockMate. The honest read is that our student networks are a narrower slice of the real market than the initial sizing implied, and that they are the wrong place to test pricing. Willingness to pay plainly exists in the wider market: Opal charges up to $19.99 a month (or $99.99 a year) and still reached the scale described earlier, ClearSpace sells a paid tier at $50 a year, and even One Sec has paying users. What a student cohort cannot tell us is what our own paying users will tolerate, and that signal has to come from interviews outside the student networks.

2.3.4 Jobs-to-be-done

Twelve recurring jobs surfaced across the interviews, of which six dominated. The dominant six are below.

1. **Fill the gap between tasks.** When one task ends and the next has not started, the phone is the easiest way to fill five minutes. Marichka plays Royal Match during work calls because the calls are boring. Veronika describes a trap where she tells herself she will start work "on the next full hour," and then, when the hour arrives, pushes the start to the hour after that.

2. **Switch off when tired.** After work or study, when the person is mentally worn out, the phone offers something that takes no mental effort. Marichka: "When I am exhausted, I just want to scroll, because everything else demands cognitive resources."
3. **Distance from an unpleasant emotion.** When stressed, sad, or anxious, the phone offers an immediate escape from the feeling. One interviewee described scrolling Instagram while crying.
4. **Maintain social connection.** This is the single biggest reason interviewees keep apps installed despite knowing they cause overuse. They need them for friend updates, work groups, or one specific contact who is only on that platform. Polina explicitly will not delete TikTok because of a streak with a close friend.
5. **Reward myself after effort.** "Five minutes of scrolling for thirty minutes of work" turning into thirty minutes of scrolling for fifteen of work, a recurring pattern across multiple interviewees.
6. **Procrastinate a difficult task.** When the next task is unpleasant, the phone is the closest exit.

Two of the twelve are non-trivially relevant to BlockMate's design. **"Wake up in the morning"** turned into a substantial sub-pattern: several interviewees described using the phone to transition from "still asleep" to "doing something," and described the transition routinely consuming thirty to forty minutes. **"Feel in control"** is the meta-job that explains why users uninstall hard blockers: they want help, not someone else taking the wheel. BlockMate's product design, an AI that grants the user access while the user articulates the reason, is built specifically around the second of those.

2.3.5 Deep dive: Marichka

Marichka is the most informative single interview in the cohort, both because she is a long-term active user of a competing product and because she gave critical UX feedback on BlockMate itself.

Marichka is a working adult with daily phone use of six and a half to seven hours. Her usage is concentrated in Telegram, Instagram, TikTok, and Royal Match, the last of which she plays during work calls because she finds the calls themselves boring. She has Opal installed in its lightest mode (5-second wait friction, 8:00–18:00 weekdays). The friction is technically present but functionally absent: she described automatically clicking the maximum 15-minute override every time the screen appears. Her summary line: "Opal is so friendly to me that it almost doesn't restrict me at all."

She tried BlockMate during the interviews and gave the most important piece of UX feedback we have received to date: BlockMate felt "unusually harsh" in its blocking format compared to Opal. This is exactly the trade-off the product is making, substantive friction over comfort, but the framing tells us that users coming from soft friction will perceive the change as a regression, not as a feature, unless onboarding actively reframes it.

Marichka also sees the screen-time problem in her own mother and actively initiates phone-free time when they are together. She avoids the phone in the gym between sets. She prefers audio-only content because it does not engage her eyes. None of these self-imposed mitigations come from a screen-time app. They come from awareness plus

social context. That distinction, awareness without an effective tool, is exactly the gap BlockMate is built for.

2.3.6 Anti-pattern: when the ICP does not fit

Three interviewees explicitly do not fit the ICP, and the reasons are diagnostic. Mariana has not tried any screen-time tool; she has not searched, does not believe the problem applies to her, and does not feel motivated to act. Mykhailo has solved the problem with Apple Screen Time alone. He does not set a passcode because, in his framing, requiring a passcode would be "a sign of weakness." Lia uses screen-time limits because a teacher assigned them, not because she experiences the problem. None of these users would buy BlockMate. They tell us, however, where the line of the ICP is.

2.3.7 Selected quotes

A few quotes from the interviews say more than any summary could, so they are reproduced word for word below.

- **Bohdan**: "Людям треба нагадувати, а не вчити." *("People need reminding, not teaching.")*
- **Marichka**: "95% часу в мене розфокус, все через короткі відео." *("95 percent of the time I am unfocused, all because of short videos.")*
- **Anna**: "У мене покращилася самооцінка і я почала краще спати." *("My self-esteem improved and I started sleeping better.")* Said about deleting Instagram and TikTok.
- **Anna**: "Коли поставила 15 хв обмеження, сиділа БІЛЬШЕ, бо відчувала що цей час виділений для неї." *("When I set a 15-minute limit, I spent MORE time, because I felt that time was set aside for me.")*
- **Mykhailo**: "Якщо не можеш себе контролювати, що ж ти за тварина така??" *("If you cannot control yourself, what kind of animal are you?")* Frames the anti-ICP attitude.
- **Marichka**: "BlockMate був незвично різкий у форматі блокування в порівнянні з Opal." *("BlockMate felt unusually harsh in its blocking format compared to Opal.")*

2.4 Hypotheses graded against the evidence

The three product hypotheses introduced in the introduction can now be re-evaluated against the customer-development data.

2.4.1 H1: existing solutions are either too soft or too hard, and they get worse with time

Verdict: validated. Five of the ten interviewees described concrete habituation events on at least one screen-time tool. Veronika auto-presses Apple's "remove the limit for today" every day. Marichka auto-clicks the 15-minute Opal override. Anna's built-in Instagram limit lasted exactly one month before she started bypassing it. Polina installed an external blocker that lasted one week (she also cited UX and data-privacy concerns, but the bar still broke). Robert has been cycling through tools for over five years.

Counter-evidence worth noting. Some barriers do not habituate. Veronika's hard 21:00 cutoff has held for over a year. Bohdan's Opal in strict mode has held for over a year. Polina 2′s ScreenZen with regular reminders has held for over a year. The pattern in the

counter-evidence is consistent: the resilient bars are the ones that either remove user agency entirely (strict mode, time-of-day cutoffs) or deliver a continuously novel signal (rotating reminders). The flat single-friction screens are the ones that habituate.

Implication for BlockMate. Contextual friction is the right hypothesis to keep testing. A conversation that depends on time-of-day, prior attempts, and emotional state is, in principle, never the same conversation twice. The risk is symmetric with Opal's habituation risk: if our AI converges on a small set of templated replies, our friction will habituate too. The validation chapter returns to exactly this concern, with the eval framework that measures variance in BlockMate's responses across repeated scenarios.

2.4.2 H2: the install–uninstall cycle can be broken by keeping the app available but gating each entry

Verdict: plausible, untested at length. The cycle itself is real and recurring across the cohort. Veronika describes it as inevitable for Instagram because she needs it for work. Anna deleted TikTok a year ago and has held; another interviewee deleted apps for a week and reinstalled to announce a stand-up event. The driver of reinstall, in every observed instance, is FOMO or a specific use case (work, social streaks, an event), never craving directly.

The hypothesis that "keep the app, gate the entry" solves the cycle is not yet testable from our cohort, which has only briefly used BlockMate. The mechanism is plausible: the user gets the work-or-FOMO use case satisfied via the conversation, without the impulsive use slipping through alongside it. The validation chapter returns to the one production-cohort user whose journey statistics begin to address this.

Implication for BlockMate. The hypothesis stays, but is downgraded from "validated" to "plausible, requires longer-term cohort evidence."

2.4.3 H3: surfacing the underlying reason builds awareness, not just barriers

Verdict: validated for the ICP, falsified outside it. Within the five ICP-fit interviewees, awareness is uniformly high (Veronika cites a book on attention, Robert has watched "many videos" on the topic over five years, Marichka can articulate the dopamine mechanism), and the gap between awareness and action is exactly what is failing the willpower-only and friction-app strategies they have tried. These users would benefit specifically from a tool that surfaces the reason in real-time, not from yet another barrier.

Outside the ICP, the hypothesis fails for revealing reasons. Mariana does not have the awareness yet, so there is nothing to surface. Mykhailo has solved his problem with a willpower-plus-Screen-Time configuration and views the underlying reason as already understood. Lia does not yet feel the problem.

Implication for BlockMate. The hypothesis is supported and sharpens the ICP: BlockMate's value is highest for users with high awareness and a history of failed action, near zero without either, so marketing should target that segment and not try to expand the rest. Taken together with H1 and H2, the customer development confirmed a real problem and a mechanism aimed at the right axis, for a segment narrower than the market sizing implied. The next chapter turns to the design.

3 Design

The research chapter ended with a problem and a hypothesis. This chapter turns them into an engineering plan: what BlockMate has to do, the handful of decisions that shaped how it is built, the architecture those decisions produced, and the design of the part that makes BlockMate different from every other blocker, the AI coach. The supporting material behind those decisions, the event-storming boards, the early architecture diagrams, and the user-story map, is in the appendix, so this chapter can stay focused on the decisions themselves.

Contents

3.1 What the product has to do

The requirements come from the customer development in the research chapter and a short user-story-mapping workshop I ran with my co-founders at the start. The list below is deliberately small, because it is the core of the product. Everything else BlockMate does, like remembering a session, adapting to the user over time, gating manual unblock behind friction, or an offline fallback, is an improvement on top of this core, not part of it.

Functional requirements. Three things the product must do:

1. **Block selected apps.** The user chooses which apps to control, and opening one of them shows a BlockMate shield instead of the app.
2. **Negotiate access with the AI coach.** To get past the shield, the user talks to an AI coach. It unlocks the app for a limited time when the request is a conscious, intentional one, asks a follow-up question when the intent is unclear, or keeps the app blocked otherwise.
3. **View screen-time analytics.** The app shows the user their own screen-time and intentionality data.

Non-functional requirements. Four properties the product must have:

1. **Responsiveness.** An AI reply should come back in under three seconds, so the negotiation feels like a conversation and not like waiting on a server.
2. **Cost efficiency.** The AI cost of a negotiation has to stay low enough that a user's lifetime value (LTV) comfortably covers it.
3. **Observability.** Every negotiation must be traceable and every product event measurable in production, so the system can actually be analysed rather than guessed at.
4. **Fidelity to the philosophy.** The AI must behave the way BlockMate believes it should: rewarding conscious, intentional use rather than merely "productive" use, and coaching instead of gatekeeping.

The workshop that produced these was most useful as a thinking exercise. It forced the three of us to lay out every flow the product implied. The detailed user stories we wrote alongside it bought less than I expected, because the product changed faster than the document, but the map itself is still a reference I come back to when planning what to build next. It is reproduced in the appendix.

Why iOS. Three platforms were on the table at the start: a Chrome extension for the desktop, a Swift app for iOS, and a Kotlin app for Android. iOS won for one main reason, which is willingness to pay. It is a well-documented pattern that iOS users buy more apps and subscriptions than Android users, and the customer profile from the research chapter (professionals and students who feel the pain of screen time and have the disposable income to fix it) is exactly that kind of buyer. There is a supporting technical reason too: Apple's FamilyControls and DeviceActivity APIs are the only way to block apps system-wide on a phone without rooting it, so iOS gave us both the audience that pays and the only API surface that delivers the experience cleanly. Android has equivalent blocking primitives but weaker guarantees and a less monetisable audience for our segment. The Chrome extension came first only because it was the one thing I could build before I had a Mac.

3.2 The decisions that shaped BlockMate

Five decisions did most of the work of shaping the system. For each one I also give the main alternative I considered and why I did not go with it.

3.2.1 A serverless backend

The backend runs as Firebase Cloud Functions, with Firestore for storage and Firebase Auth for identity. There is no application server of my own to keep alive. The alternative, which I actually built first, was a NestJS modular monolith on PostgreSQL with a Redis cache, deployed to a managed container platform. I abandoned it because a negotiation does almost no server-side work of its own, so all of that infrastructure would have been kept alive to run a few dozen lines of logic. A function that wakes up, runs, and scales back to zero fits the workload far better and costs almost nothing when idle. The implementation chapter tells the full story of that pivot, including the cold-start cost I accepted in return.

3.2.2 Two models, not one

Each negotiation turn uses two separate model calls. A small, cheap model classifies the user's message into a structured object, and a stronger model writes the actual reply. The obvious alternative is one call that both classifies and responds. I chose two for three reasons: the structured object the classifier produces is the input to a pure-code decision step, so it has to exist on its own; the cheap model is enough for classification and costs a fraction of the strong one; and that same object is what the long-term journey tracker reads, so it must be machine-readable rather than buried in prose. The implementation chapter shows the models, their settings, and the fields they exchange.

3.2.3 A rule-based strategy selector

Which coaching techniques the AI applies to a given turn is decided by plain TypeScript, not by another model call. The alternative was a third model call asking "which strategies fit here?". I chose rules for predictability, debuggability, and cost: the same input always produces the same selection, debugging is reading a short chain of conditionals instead of arguing with a model, and it adds no token cost on the hot path. A model-driven selector would drift every time I tuned a prompt and would turn every regression into a matter of opinion.

3.2.4 The journey as derived state, not raw history

The AI never sees a user's full history. It sees a small, fixed-shape summary computed from that history: a stage, a trust level, an intentionality score, a few streaks and counters. The alternative was to feed the last several negotiations into every prompt. I chose the summary for three reasons: feeding history in would make token cost grow with how long someone has used the app; the model behaves differently as its prompt grows, and I want it steady across a user's lifetime; and a fixed-shape object is testable in a way a growing transcript is not.

3.2.5 iOS-first, built on Apple's Screen Time APIs

The production app is iOS-only, built on FamilyControls for app selection, DeviceActivity for monitoring, and ManagedSettings with shield extensions for the blocking and the custom shield screen. The alternative was cross-platform through React Native or Flutter with a native module for the blocking. I chose iOS-only for the willingness-to-pay reason above, and because the blocking primitives are the whole point of the product: they cannot be faked in a cross-platform layer, they have to use Apple's APIs directly. Building cross-platform would have split my effort across two app stores and spent part of it on Android, which, as noted above, is the lower-revenue half of the audience.

3.3 The shape of the system

At the widest zoom, BlockMate is one system the user touches through their iPhone, leaning on a few outside services. OpenAI answers the two model calls per turn, sitting behind the provider interface so it can be swapped. Apple's FamilyControls, DeviceActivity, and ManagedSettings do the actual blocking, the shield, and the schedule monitoring; they are Apple's own frameworks and the only way to block apps on an iPhone, so there was nothing else to choose between. The App Store handles distribution and, once payments are live, subscription state. A small marketing website rounds out the picture, mostly for the waitlist.

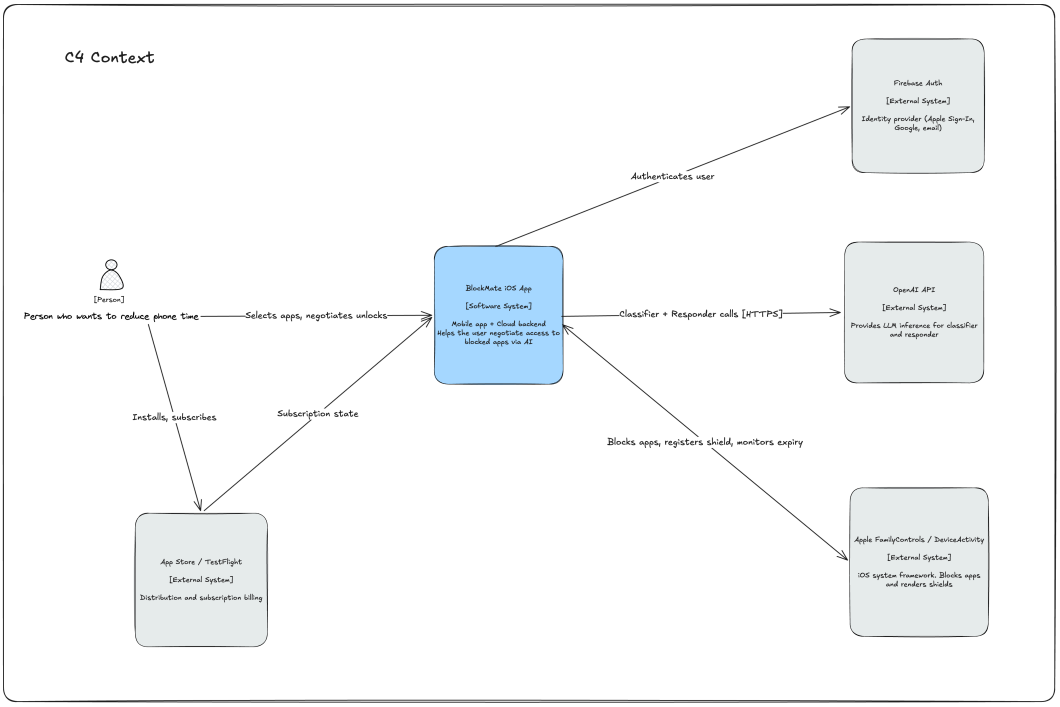

Figure 1: System context. BlockMate sits in the middle, the user is the only actor, and everything on the outside (OpenAI, Apple's frameworks, the App Store) is something the app calls out to.

One level in, the deployable pieces and how they talk. On the phone there is the iOS app plus its blocking extensions; in the Firebase project there is the single

`negotiateAppAccess` function, Firestore as the only persistent store, Auth, and Hosting for the website. OpenAI is the one true external dependency in this view.

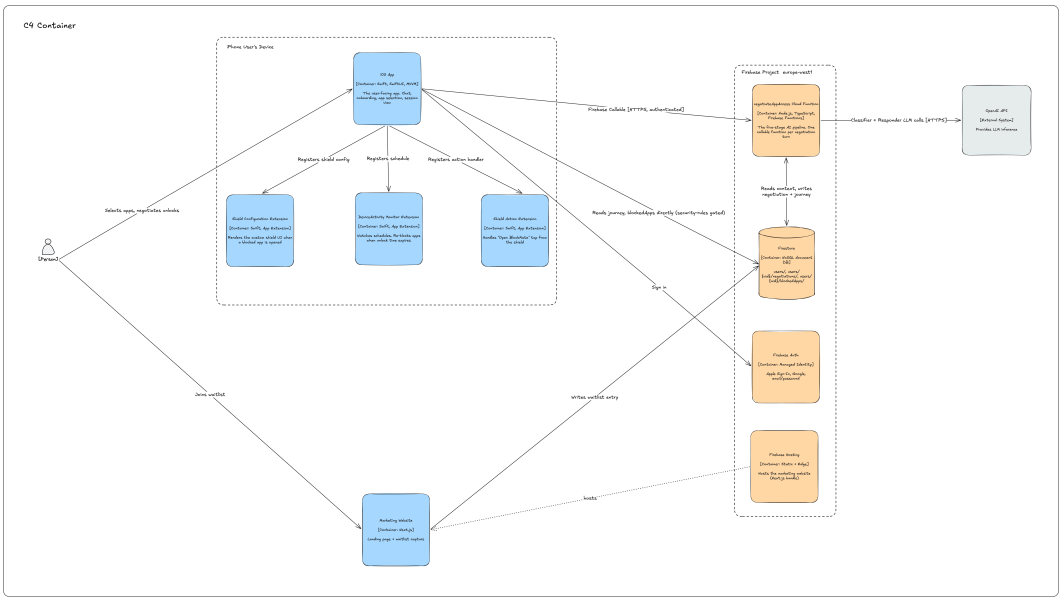

Figure 2: Containers, as they run in production. The iPhone side holds the app and its blocking extensions; the Firebase side holds the negotiation function, Firestore, Auth, and Hosting. The marketing site writes waitlist entries straight to Firestore.

This is not the architecture I first designed. Before the pivot, the system was a NestJS monolith with PostgreSQL, Redis, and a full REST surface; that earlier design, and the API it implied, is kept in the appendix as a record of the thinking that preceded the serverless decision. After the pivot, almost all of those endpoints collapsed into the one callable function plus a few direct Firestore reads of the user's own data.

One more level in, inside the function itself, seven components run in order on each call. Five form the request path and two handle the side effects. The next section is where they come alive; here they are just named.

3.4 The negotiation pipeline

The negotiation runs as five stages. Two are model calls and three are plain code, and keeping the code stages out of the model is what makes the system testable and cheap. A user's message enters on the left and a structured coach response comes out on the right.

1. **Context assembly** (code) gathers the fixed-shape context the rest of the pipeline reads: the time, the user's onboarding baseline, today's activity, and the journey summary.
2. **Classification** (model) turns the message and a compressed context summary into the typed object described in the implementation chapter.
3. **Strategy selection** (code) reads that object and the context and picks the two or three coaching techniques to apply.

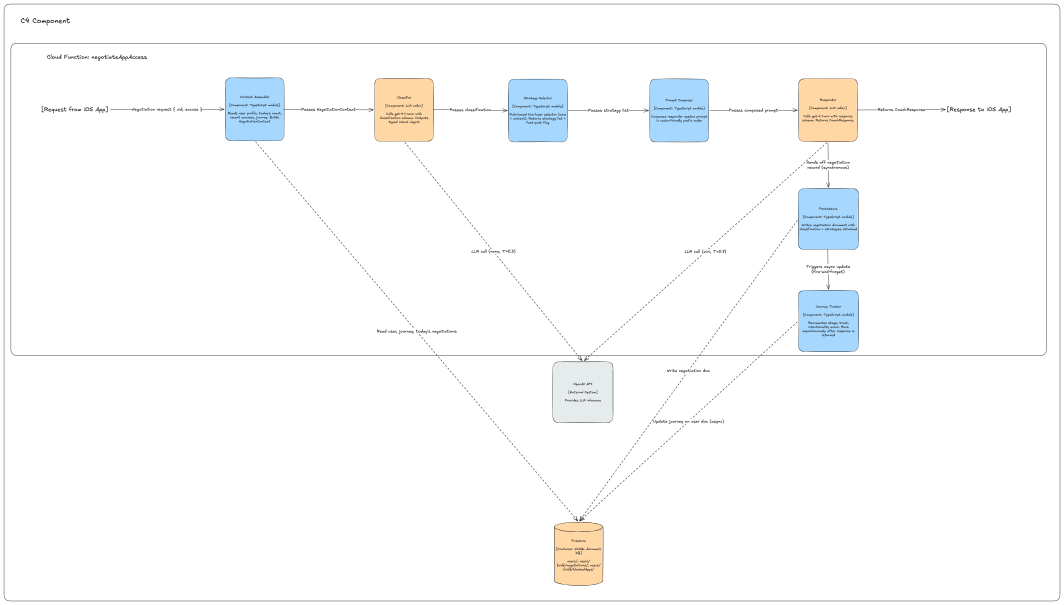

Figure 3: Components inside **`negotiateAppAccess`**. Five form the request path (context assembly, classification, strategy selection, prompt composition, response generation); two handle the side effects (persistence, run synchronously; the journey update, run after the reply is sent). Orange components are the two model calls; blue ones are pure TypeScript.

strategies, and the user context.

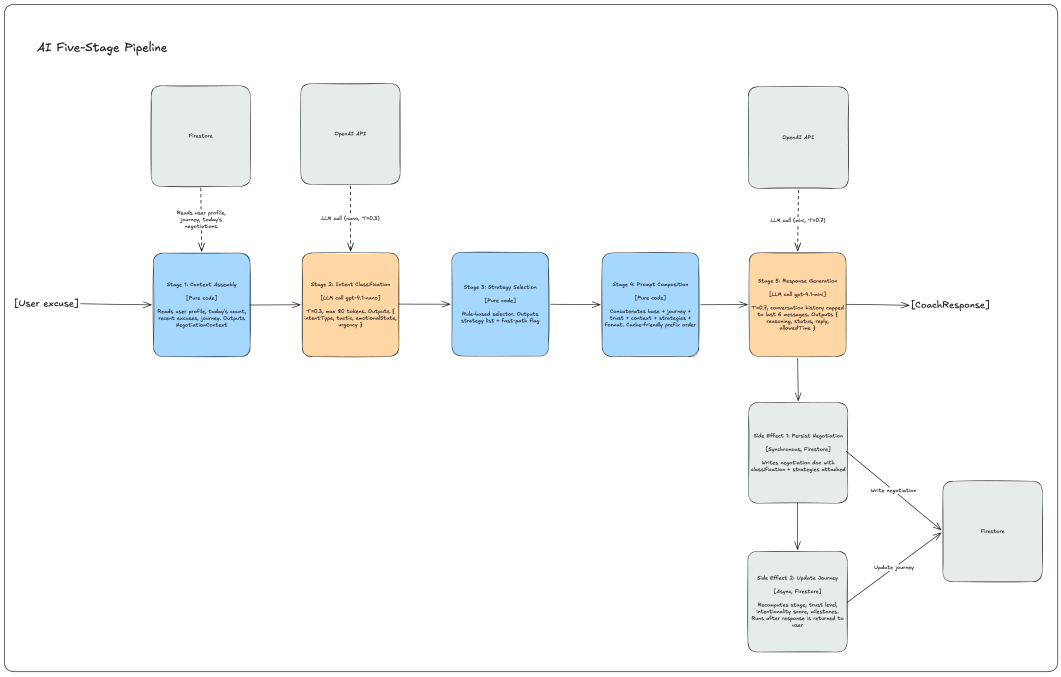

Figure 4: The five stages. Stages 1, 3, and 4 are pure TypeScript (blue); stages 2 and 5 are model calls (orange). After the reply is returned, two side effects fire: the negotiation is written to Firestore synchronously, and the journey is updated afterwards.

5. **Response generation** (model) writes the reply, the verdict, and the minutes to grant.

Two things happen after the user has their answer. The negotiation is saved to Firestore straight away, because the next turn depends on it. The journey is updated afterwards, off the critical path, because future turns can tolerate it being a moment stale. The implementation chapter shows the code, the model settings, and exactly what each stage consumes.

3.5 The coaching strategies

The AI's behaviour is not one giant prompt. It is a base identity plus a library of small, named coaching techniques, and the selector picks a few of them for each turn. The techniques are drawn from established, evidence-based approaches to behaviour change rather than invented, because the goal is to actually move behaviour, not to sound clever. There are eleven of them today.

Most turns draw on the **intentionality check**, which is the core move of Motivational Interviewing: rather than telling the user what to do, the AI asks the one question that lets them notice for themselves whether this is a choice or an impulse. Around it sit techniques matched to the situation. **Self-compassion** is used when the user is honest about an urge, because shame just feeds the loop of feeling bad and scrolling to escape, so the AI validates the honesty before redirecting. **Habit-loop awareness** is used when the user looks like they are reacting to a trigger rather than choosing; it names the cue and offers a different routine, following the standard cue-routine-reward model of how habits work. **Time awareness** comes out late at night or when the user is restless, reminding them that an urge rises and fades on its own if you let it, a technique borrowed from mindfulness-based relapse prevention. The **Ulysses reminder** is used when the user bargains or repeats an excuse; it points back to the commitment they made earlier, when they were thinking clearly, the same idea as a Ulysses contract[2]. The rest handle specific cases: **pattern recognition** when there have been several attempts today, **present-bias counter** for heavy users, **milestone recognition** when the user has just hit a streak, **quick approval** for the obvious productive asks, **adversarial refusal** for prompt-injection or hijack attempts, and **de-escalation** when the message is an emotional appeal or a deflection.

The selector that chooses among them is a short, two-layer algorithm with a defensive check at the front, shown below. The implementation chapter walks through how it runs and why the fast paths matter for cost.

3.6 The journey object

BlockMate's long-term memory of a user is the journey object, introduced as a decision above and detailed in the implementation chapter. In design terms it is a small, fixed-shape summary stored on the user's profile and recomputed after every negotiation. It carries

[2] In Homer's *Odyssey*, Ulysses wanted to hear the Sirens' song but knew it would lure him to his death, so he ordered his crew to tie him to the ship's mast beforehand, where he could hear the song but not act on it. A *Ulysses contract* is the general term for binding your future self to a decision you make while thinking clearly.

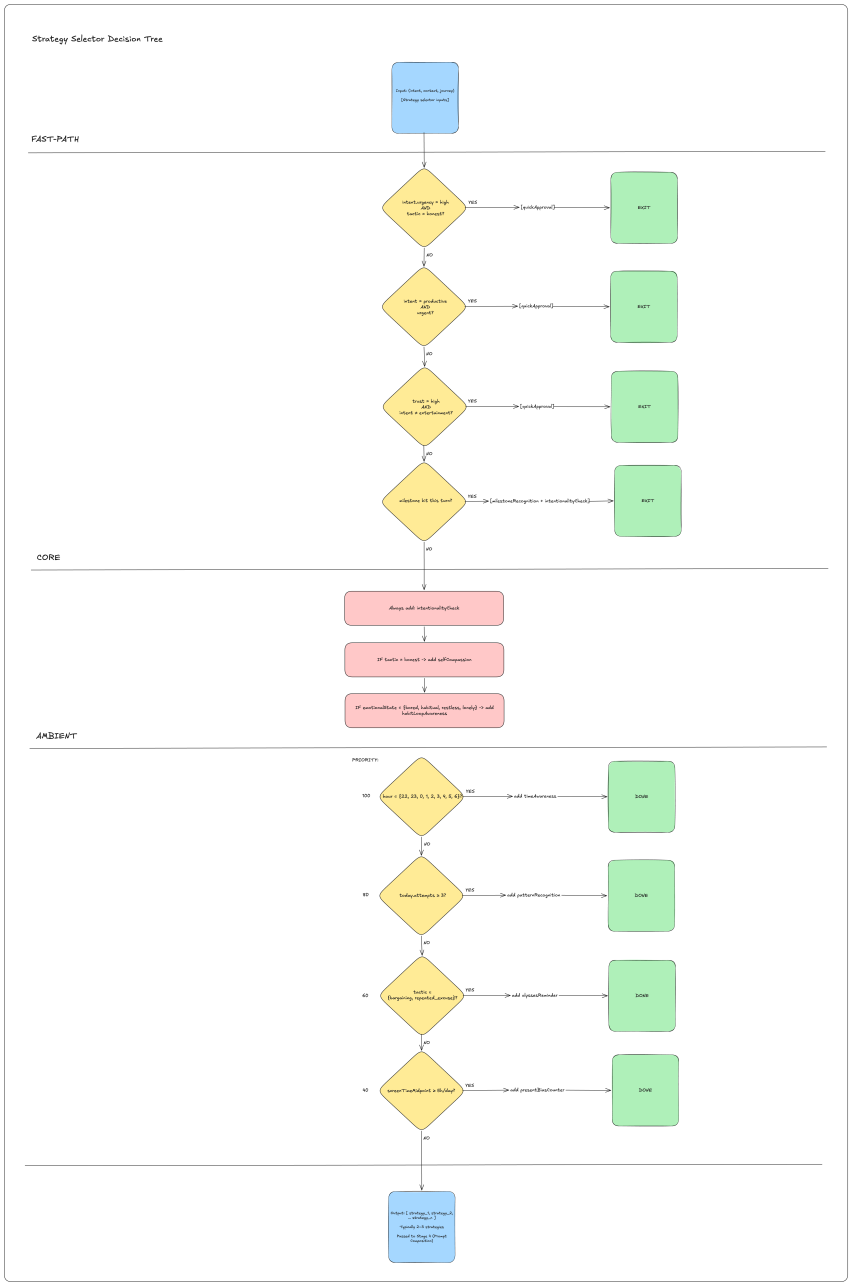

Figure 5: The strategy selector. A defensive check for adversarial input comes first. Then a fast path handles the obvious cases (emergency, clearly productive and urgent, high-trust non-entertainment) by approving immediately and skipping the responder model. Otherwise a core layer always adds the intentionality check and one of self-compassion or habit-loop awareness, and an ambient layer adds exactly one more situational technique by priority. Picking only one keeps the reply short.

the counts and streaks, a **stage** that grows with tenure (**`newcomer`**, **`building`**, **`growing`**, **`established`**), a **trust level** earned from consistent intentional behaviour (**`standard`**, **`elevated`**, **`high`**), and an intentionality score from 0 to 100.

These two dials are what let one system coach very different people. The stage sets the default tone, so a newcomer is met with warmth and patience while an established user gets a lighter touch. The trust level shifts the verdict, so a high-trust user is approved unless there is a clear impulse signal, while a standard user is read more carefully. None of this needs a separate prompt per user; it falls out of a few numbers.

3.7 Voice and method

Everything about how BlockMate talks lives in the base prompt and the strategy fragments. The identity is one line:

> You are BlockMate, the user's own rational self made digital. They installed you because they knew moments like this would come. You are not a gatekeeper, not a therapist, not a parent. You are a mirror for intentionality.

That single framing does most of the emotional work. It makes the AI the user's earlier, clear-thinking self rather than a guard at the door, which is the Ulysses-contract idea turned into a voice. The method underneath is Motivational Interviewing, where the conversation is built around questions that help someone reach their own answer, and self-compassion over guilt, because guilt drives the very behaviour we are trying to reduce. The tactical rules of voice (one or two sentences, no repeated phrasings, match the user's energy, no corporate filler) are constraints in the prompt, and the implementation chapter tells the story of how that voice was actually won, prompt era by prompt era.

3.8 From event storming to bounded contexts

Before any of this existed in code, I wanted the three of us to be building the same product in our heads. The tool I used was event storming, a workshop technique I picked up from Ihor Pysmennyi's service-oriented architecture course, where you map a domain by sticking up the events that happen in it (in the past tense) and then layer on the commands, actors, policies, and external systems around them. I walked Oleh and Dmytro through the notation and we ran a series of boards together, going from a messy wall of every event we could think of, to ordered flows for each major scenario, to a clean resolution of the trickier sub-flows one at a time. The boards themselves are in the appendix.

What the workshop produced was a clean split into bounded contexts: onboarding, policy (rules and app selection), negotiation, subscription, and identity. The implementation flattened this under Firebase, where negotiation is the only context with a real function behind it and the rest live in iOS state and Firestore documents, but the concepts the workshop surfaced are exactly the ones the code ended up manipulating. Several specific post-its became real things: a "successful excuse already entered" policy became the classifier's repeated-excuse tactic, and a "block app after unlock expires" event became a DeviceActivity schedule on the phone.

3.9 UX that makes the philosophy work

Three interface choices carry the philosophy. Without them the architecture would be sound and the product would still fail.

Onboarding is persuasion, not paperwork. The flow moves from a short intro, through a screen-time guess and an age range, to a calculation, to a "shock screen" that makes the cost personal. From the screen-time guess and the age, BlockMate projects how much of the user's remaining life is heading into the feed and how much of it could be reclaimed. For someone aged 25 to 34 with two to three hours a day, that comes out around eight lifetime years given to the algorithm, of which about 2.4 are reclaimable. The number animates up in large type next to the line that BlockMate can help the user get those years back for the things that actually matter. The number is real and the framing is honest, and that moment is what makes a user grant Screen Time permission on the next screen instead of dropping the app.

Manual unblock is deliberately hard. The shield would be pointless if a user could just toggle the app off the blocklist whenever they liked, so removing a blocked app triggers a friction test: the user has to type a target phrase exactly, with paste disabled, the phrase non-selectable so it cannot be copied, and the button staying dead until the typed text matches character for character. The intent is to make impulsive removal genuinely annoying while leaving deliberate removal possible. Someone who truly wants the app back can type the sentence; someone just dodging the conversation gives up. Adding a new app to the blocklist saves normally; only removal is gated.

The verdict encourages, it never punishes. An approval opens a full-screen success state: a calm timer counting down the unlocked minutes, with a button to lock again early. The rejection was the harder half to get right. My first design paired that success screen with a matching red "rejected" screen, and it was a mistake. A full red screen tells the user they failed, and the guilt and frustration that follow are exactly the feelings that send someone back to the feed to escape them, the very loop the product exists to break. So I cut the rejection screen altogether. A held request now stays inside the negotiation chat and simply swaps the input box for a small, plain banner showing the time until the next attempt. Either way, whether the verdict is an approval or a hold, it carries a thumbs-up / thumbs-down control asking "was this the right call?". That feedback is the only path by which a real disagreement in production becomes a test case in the eval set, which is what keeps the AI improving on real input rather than only on the scenarios I thought to write.

3.10 Data model

The Firestore side is intentionally small. A **`users/{uid}`** document holds the profile, the onboarding numbers, and the embedded journey object. Under it, **`negotiations/{id}`** holds one document per negotiation with the conversation, the classification, the strategies, the verdict, and the timestamps. A locked-down **`private`** document holds the Apple refresh token used when an account is deleted. One top-level **`verdictFeedback`** collection holds the thumbs-up / thumbs-down records, kept at the top level so eval and analytics can scan them across all users without awkward nested queries.

One thing is deliberately not in Firestore: the list of blocked apps. Apple hands the app an opaque selection token that is only meaningful on the device that created it, so storing it on a server would be both a privacy regression and useless anywhere else. It lives instead in the shared App Group[3] container on the phone, where the app and its extensions all read the same source of truth without a network call. The journey object, by contrast, is embedded right on the user document rather than split into its own collection, because it is read on every negotiation and a single document read is the cheapest thing Firestore does.

3.11 Running it: security and observability

Access control is three layers. Firebase Auth gates identity (Apple, Google, or email). Firestore security rules gate the data, so a user can only ever read and write their own subtree. And the negotiation function itself rejects any unauthenticated call. On top of that, the prompt is the only thing that reaches OpenAI, and it carries no identifying data, so the privacy boundary is the prompt boundary.

Observability is a first-class requirement, not an afterthought, and the implementation chapter shows the wiring. In short, every negotiation produces a full trace in LangSmith with the verdict, strategies, latency, and confidences attached, and every meaningful product event flows into Amplitude. Between them I can answer both "what did the AI do on this exact turn?" and "how is the whole funnel behaving?" without building a logging pipeline by hand.

3.12 How it scales

BlockMate has never been pushed hard: peak traffic is a few requests per second and Cloud Functions are wildly over-provisioned for it. But knowing what happens at growth is the architect's job. Cloud Functions comfortably carry the launch tier, where 100,000 monthly users works out to roughly 12 requests per second on average. The first migration that becomes interesting is from Cloud Functions to Cloud Run somewhere past a million users, where running the same code in a container wins on per-invocation cost and gives more control over cold starts while staying serverless. Firestore stays until read cost dominates, at which point a Redis read-through cache in front of the journey reads earns its place, and the OpenAI calls move behind a queue to smooth out bursts.

Traffic tier	Daily calls	Platform
Current (pre-launch)	< 10K	Cloud Functions, over-provisioned but the right shape
Launch (100K MAU)	≈ 1M	Cloud Functions, still in the sweet spot
Scale (1M MAU)	≈ 10M	Crossover point, evaluate Cloud Run
Scale (10M+ MAU)	≈ 100M+	Cloud Run, queue in front of OpenAI, Redis cache

Table 1: The platform plan per traffic tier.

[3] An *App Group* is an iOS mechanism that lets an app and its extensions share one common container (files and preferences). They otherwise run in separate sandboxes and cannot see each other's storage.

The daily-call figures come straight from the user counts: at the launch tier, 100,000 monthly users average roughly twelve requests a second, which is about a million calls a day, and each tier above scales that by ten. The platform column is the engineering plan rather than a measurement. I have deliberately left dollar figures out, because pre-launch the infrastructure is effectively free (current traffic sits comfortably inside the Firebase free tier), and because the cost that actually decides the unit economics is the OpenAI token bill, not the compute. That one *is* measured directly, in the validation chapter, at about a third of a cent per negotiation, and it grows roughly linearly with usage, which is why the prompt-caching work in the implementation chapter is load-bearing rather than a nice-to-have.

There is a second migration that has nothing to do with traffic. When BlockMate runs serious marketing funnels on the website, which is likely, the website needs a real backend of its own for web checkout, landing-page tests, attribution, and operator dashboards. That is a poor fit for Cloud Functions and Firestore and a good fit for the NestJS and PostgreSQL backend I built and shelved, so it would return as the web backend while Cloud Functions keep serving the AI.

3.13 Related work

Three threads of prior work sit behind BlockMate. The use of Motivational Interviewing in chatbots is well studied in clinical settings, from smoking cessation to managing depression; BlockMate is unusual in bringing it to a consumer product, but the principle carries over cleanly. The two-model pipeline, a cheap classifier feeding a stronger generator, is now a standard pattern wherever a structured intermediate is useful, though BlockMate's twist of feeding that intermediate into a rule-based selector is less common. And in the screen-time space itself, the closest analogue is OneSec, which uses fixed friction without any conversation. Replacing that fixed friction with a real conversation is, as far as I know, new in this category.

The next chapter turns this blueprint into a built system: the iterations, the obstacles, and the engineering decisions made under real pressure rather than on paper.

4 Implementation

The design chapter shows the design as if I had it figured out from the start. I did not. This chapter is what actually got built, and the engineering that matters most: why the backend is serverless, how the AI pipeline works and what it is allowed to see as context, how I keep it cheap, and how I know it works before anything ships. The finished system is the Firebase backend and the Swift iOS app, plus a long fight to make an AI sound like a friend instead of a bouncer.

Contents

Before the production system settled, BlockMate's code was rewritten twice. The first version was a Chrome extension that blocked websites and called OpenAI from the browser, built in a few days because blocking apps on a phone needs Apple's APIs and a Mac I did not yet have. The second was a NestJS and PostgreSQL backend I built while waiting for that Mac, when the team needed something shipped every week; I took it to a clean module skeleton and then deliberately dropped it once I saw how little server-side work a negotiation actually needs (the next section explains why). I kept it in the repository on purpose: when we run a real marketing operation on the website, with web checkout, funnels, and attribution, that relational structure is the natural backend for it. The full build history of both is in the appendix. The rest of this chapter is about the third and final iteration: the Firebase backend and the Swift iOS app that shipped.

4.1 The backend is one function, and why that is enough

The negotiation backend is a single Firebase Cloud Function, and that is a deliberate choice, not a shortcut. A negotiation does very little work of its own: per turn it reads a handful of Firestore documents, makes two model calls, writes one document, and returns a small JSON reply, about forty lines of real logic. No long-running process, no real-time fan-out, no heavy stateful step. That shape is exactly what serverless is for. A Cloud Function exists only while a request is in flight, scales to zero between requests, and is billed per invocation, so an idle product costs almost nothing, whereas a server, a database, and a cache all have to stay up and get paid for whether anyone is negotiating or not. That is why I threw away the iteration-2 monolith: its operational surface, a container platform to keep running around the clock, a database and a Redis cache alongside it, and a deployment process to maintain, was all scaffolding for forty lines of logic at a scale we were nowhere near. The honest cost of serverless is the cold start: when no instance is warm, the first call pays one to three seconds of spin-up, and the validation chapter shows that is exactly where the responsiveness target slips. The mitigations are prompt caching and keeping one instance warm during peak hours.

The Cloud Function is **`negotiateAppAccess`**. Its HTTP entry point does almost nothing (parse the request, call the orchestrator, map errors to a clean failure), and all the real work lives in a single internal function in the code, **`runNegotiationTurn`**. Walking through what it does is the clearest way to explain the system:

1. It loads two things in parallel: the user's **context** (profile, today's attempts, journey state) and the **conversation history** for this negotiation. The two reads do not depend on each other, so they run at the same time rather than one after the other, which saves a database round-trip on every call.
2. It then runs the pipeline: classify the intent, clamp the requested minutes, pick the coaching strategies, compose the prompt, and generate the reply. Two model calls, three pure-code steps. The next section is all about this.
3. If either model call throws, it does not surface an error to the phone. It returns a **degraded** reply that gently asks the user to try again, flags the turn as degraded, and keeps the conversation alive. A hiccup with OpenAI should never brick a negotiation.
4. Finally it saves the negotiation, updates the journey, and fires analytics, all fire-and-forget so the user gets their answer without waiting on writes. The journey update is

skipped on degraded turns, so a fallback classification can never poison the long-term state the AI reads next time.

The full code of the orchestrator and the entry point is in the appendix; the shape above is what matters.

After a few months I reorganised the backend. Small Functions projects tend to drift into one flat pile of files; I split mine into four layers, following the ports-and-adapters pattern adapted for serverless. The **`functions/`** layer holds the HTTP entry points and nothing else. The **`services/`** layer orchestrates a request into one workflow and owns tracing, analytics, and the degraded fallback. The **`domain/`** layer holds the business logic: the classifier, the responder, the strategy selector, the prompt composer, and pure helpers for things like clamping the requested minutes and building the degraded fallback. The two model-calling functions take the OpenAI client as an argument rather than reaching for it themselves, so they can be tested against a fake client, while the pure helpers need no setup at all. The **`infra/`** layer holds thin typed adapters to Firestore, OpenAI, Apple Sign-In, and Amplitude, with a small **`config/`** alongside it for singletons and secrets. Dependencies only point inward: there is no path from **`domain/`** out to **`infra/`**, which is what keeps the domain code testable in isolation. Discipline replaces the framework here, with no dependency-injection container and no decorators, so nothing is added to cold-start time. The refactor changed no behaviour, the eval suite passed identically before and after, and it is the structure the rest of this chapter refers to. Deployment is automated to match: a push to **`main`** that touches the functions triggers a GitHub Actions workflow that installs dependencies, authenticates to Google Cloud, and runs **`firebase deploy --only functions`**, so a merged change reaches production without anyone running a deploy by hand.

The one seam worth showing in code is the boundary between my logic and OpenAI. Both model calls go through a small interface, **`LlmProvider`**, with exactly two methods:

```
export interface LlmProvider {
  readonly name: LlmProviderName;
  classifyIntent(userMessage: string, ctx: NegotiationContext):
Promise<IntentClassification>;
  generateResponse(
    composedPrompt: string,
    history: ChatMessage[],
    userMessage: string,
    strategies: StrategyKey[],
    options?: GenerateResponseOptions,
  ): Promise<CoachResponse>;
}
```

Nothing in my business logic imports the OpenAI SDK. The one implementation today is **`OpenAiProvider`**, which holds the SDK client, applies a per-call timeout (4 s for the classifier, 8 s for the responder), and retries once on transient errors (429, 5xx, connection drops). Swapping to Azure, to OpenRouter for failover, or to a self-hosted model is a one-file change: add a provider, widen the name type, and switch which one **`getLlmProvider()`** returns. I kept the interface deliberately narrow, with two task

methods rather than a general chat wrapper, because a wide one would leak vendor details everywhere and defeat the point. This provider-agnostic seam keeps the rest of the codebase from being welded to one vendor's SDK, which matters because the AI-provider landscape shifts every few months.

4.2 How the AI actually works

This is the heart of the system, so it is worth slowing down. Every negotiation turn runs two model calls with a wall of pure code between them, and the design of that wall is what makes BlockMate behave like a coach instead of a chatbot with a system prompt.

4.2.1 Two models, not one

The first call is a **classifier**: a small, cheap model (**`gpt-4.1-nano`**, temperature 0.3, capped at 140 tokens) that reads the user's message and returns a typed object, nothing user-facing. The second is the **responder**: a stronger model (**`gpt-4.1-mini`**, temperature 0.7, capped at 180 tokens) that writes the actual reply, the verdict, and the minutes to grant.

Splitting them was a real decision, not an accident. I could have asked one big model to classify and reply in a single call. Three things made two calls the better choice. First, the classifier produces a structured object that becomes the input to a pure-code decision step; if I folded that into the responder, all the routing logic would have to live inside a prompt, where it is hard to test and impossible to keep stable. Second, classification is a cheap job that does not need the stronger model, so doing it on the nano model costs almost nothing. Third, that typed object is also what the long-term journey tracker reads to decide whether a turn was impulsive, so it has to be machine-readable rather than buried in prose.

The classifier returns eight fields. Four of them drive everything downstream:

- **`intentType`**: one of **`productive`**, **`entertainment`**, **`social`**, **`ambiguous`**, or **`emergency`**.
- **`tactic`** (how the request is being made): **`honest`**, **`vague`**, **`bargaining`**, **`emotional_appeal`**, **`repeated_excuse`**, **`deflection`**, or **`adversarial`**.
- **`emotionalState`**: one of **`bored`**, **`stressed`**, **`lonely`**, **`habitual`**, **`restless`**, **`genuine_need`**, or **`unknown`**.
- **`urgency`**: **`high`**, **`medium`**, or **`low`**.

The other four are supporting signals: **`requestedMinutes`** (the time the user actually asked for, which the responder is then forced to honour, so "give me 5 minutes" never comes back as fifteen), **`userLanguage`** (so the responder replies in the user's language), **`reasoning`** (internal, used in evaluation), and **`confidence`** from 0 to 1 (logged to analytics so I can pull the replies the model itself was unsure about). One tactic short-circuits the whole pipeline: when the message is classed **`adversarial`**, meaning a prompt-injection or an attempt to hijack the conversation onto something else, the system returns a templated refusal and never runs the responder at all.

4.2.2 What the AI is allowed to see

The AI can only coach well if it sees the right context, so I gave this part real thought.

Before the classifier runs, a pure-code step builds a fixed-shape `NegotiationContext`. It holds four kinds of information and nothing else:

1. **Time.** The hour and the day of week in the user's local time. The same words mean very different things at 10 a.m. on a Monday and at 11 p.m. on the fourth try of the night, and the model is told to read that difference.
2. **Baseline.** The age range and the daily-screen-time range the user picked during onboarding. A rough sense of who they are, not exact figures.
3. **Today.** How many negotiations the user has started since their local midnight, plus a short record of the three latest ones. For each of those three the AI sees only the user's opening request and the verdict that negotiation ended on (`APPROVED`, `REJECTED`, or `MORE_INFO`), never the full back-and-forth. This is the short-term memory that lets the model notice "I'll just check one thing" arriving for the fifth time today, turned down every time before.
4. **Journey.** A short summary of the user's long-term state, described in the next section.

The responder also gets the last six messages of the current conversation, so it can follow the thread across turns without re-reading everything. One deliberate limit is worth a line: the AI never receives anything that identifies the person, so no email, no name, no exact age, no device id. Keeping the context this small is partly privacy and partly engineering, because a clean, fixed shape makes the model behave predictably and keeps the cost of every call known in advance.

4.2.3 Choosing the coaching strategies

Between the two model calls sits my favourite part of the system: a small piece of plain TypeScript I call the **strategy selector**. It reads the classification and the context and decides which coaching techniques the responder should use this turn. Each technique is a short paragraph of instructions; the selector picks two or three of them, and they get pasted into the responder's prompt.

What each technique does, and the evidence base behind it, is laid out in the design chapter. The interesting part here is the selector that picks among them.

The selector works in two layers. First it checks a few obvious cases that skip the responder model entirely: an emergency, a clearly productive and urgent request, a trusted user asking for something that is not entertainment, or an adversarial message. Each of these returns a fixed, templated reply, so the cheapest and the most hostile requests cost nothing on the expensive call. If none of those fire, it always adds the intentionality check, optionally adds self-compassion or habit-loop awareness depending on the emotional read, and then adds exactly one situational technique chosen by priority (late-night time awareness comes before repeat-attempt awareness, which comes before the rest). Picking only one keeps the reply short. Adding all of them would turn a text message into a lecture.

I made this a rule rather than a third model call on purpose. Rules give the same answer for the same input, so the behaviour does not drift every time I tune a prompt, debugging is reading a short chain of `if`s rather than arguing with a model, and it adds no token cost on the hot path.

4.2.4 The journey: long-term memory without the raw log

The AI never sees a user's full history of negotiations. It sees a small summary, the **journey object**, recomputed after every turn and saved on the user's profile. It holds a handful of numbers: how many negotiations they have had and how many were approved, their current and best **intentional streak** (consecutive days without an impulsive unlock), rolling 7- and 30-day counters, a **stage**, a **trust level**, an **intentionality score** from 0 to 100, and a few milestone flags.

The stage and the trust level are not set by hand; the code computes them. The stage comes from how long the user has been around: `newcomer` in the first week, `building` through weeks two to four, `growing` for the next couple of months, `established` after about ninety days. The trust level is earned from behaviour rather than time. Everyone starts at `standard`, moves to `elevated` after a sustained stretch of low-impulse, intentional use, and reaches `high` only after weeks of it. The intentionality score rolls these signals into one number from 0 to 100, weighted toward the recent ratio of intentional unlocks to impulsive ones, so it can fall again if someone slips.

These two dials change how the AI talks and how it decides. The stage sets the default tone: a newcomer gets warmth and patience, an established user gets a lighter touch. The trust level nudges the verdict: a high-trust user is approved unless there is a clear impulse signal, while a standard user is read more carefully. That is how the same system can coach a nervous first-week user and a disciplined three-month user without me writing two separate prompts.

Keeping all this as a summary rather than the raw log is a real engineering choice. If I fed the whole history into every prompt, the token cost would grow the longer someone used the app, so my most loyal users would also be my most expensive ones. The model also drifts as its prompt grows, and I want it steady across a user's whole lifetime. A fixed-shape object is testable too, in a way a growing transcript never is.

4.2.5 Two prompt eras

The pipeline shape settled early; the long work was on what the AI actually says, and it went through two eras that both still live in the repo.

Era 1 was a reasoning engine. The prompt opened with "you are a reasoning engine designed to evaluate Human Intentionality" and ran three logic tests on every message: means vs. ends, specificity, and a bargaining check. It worked, and it felt like a tribunal. The user made a case, the AI ruled. The verdicts were right and the experience was cold.

Era 2 is what runs now. The prompt opens differently:

> You are BlockMate — the user's own rational self made digital. They installed you because they knew moments like this would come. You are not a gatekeeper, not a therapist, not a parent. You are a mirror for intentionality.

That one framing, a Ulysses contract where the user's earlier self speaks to their present self, changes the whole tone. The AI stops being the guard at the door and becomes the person you were this morning, reminding you what you wanted. The logic tests survived,

but they moved inside individual strategy files that get composed per turn, so I can add a behaviour as one paragraph or delete it as one file, and test each on its own. Most of era 2′s extra rules (reflect before you ask a question, never open with a bare interrogative, run two turns before approving entertainment) came straight from the failures a few sections down.

4.3 The iOS app, and why Apple makes you work for it

The Apple side is the most fragmented part of the codebase, and that is Apple's doing, not mine.

Before I could write a line, I had to ask Apple for the FamilyControls entitlement[4], a written justification that a human at Apple reviews by hand. Ours took about two weeks. The blocking is built on Apple's Screen Time APIs, and the central piece is the **shield**: the full-screen cover iOS draws over a blocked app instead of letting it open. BlockMate replaces the default grey shield with its own. The catch is that these APIs are spread across separate build targets, each needing its own signing, entitlements, and provisioning. BlockMate ships five of them, sharing data through an App Group container because the extensions cannot share memory with the main app:

- the **main app**: a three-tab SwiftUI app (Home, Block List, Settings);
- a **monitor extension** that re-applies the shield when an unlock window ends, and writes per-hour unlock counts back for the analytics view;
- the **shield configuration**, which draws the custom shield the user sees ("Is this worth your time?");
- the **shield action**, which handles a tap on the shield (every tap simply closes it, and the user opens BlockMate themselves);
- a **report extension** that hosts the screen-time, hourly-chart, and daily-insight scenes the app embeds.

I built it in the obvious order: a skeleton tab app just to prove the entitlement worked and the permission prompt appeared, then app selection through Apple's picker, then the custom shield. The shield fought me, because Apple's documentation is thin and the failure mode is a silent blank screen, so it took several tries before I found the one `Info.plist` key it needed. After that came the chat, the unlock-expiry re-block (a `DeviceActivitySchedule` that fires when the window ends, plus a monitor callback that re-shields), and authentication. Sign in with Apple came last, not because it was the hardest, but because Apple requires it: any app offering a third-party login like Google must also offer Sign in with Apple to pass App Store review. Onboarding and the in-app analytics screen came at the very end. Per-target detail is in the appendix.

4.4 The obstacles that taught me the most

None of these were bugs. The system did what I told it to. The lessons were in what I had told it to do.

[4] An *entitlement* is a capability an app must declare and Apple must grant before the app can use a protected system feature. FamilyControls is one of the few that Apple reviews by hand rather than granting automatically.

4.4.1 The AI spoke broken Ukrainian

My first testers were Ukrainian, and the AI's Ukrainian was painful. It was grammatically correct but sounded like a tourist reading from a phrasebook.

The reason surprised me. My prompt was full of example lines I had written in English to show the model the tone I wanted. When someone wrote in Ukrainian, the model took those English lines and translated them word for word. You could hear the English underneath, like a bad dub. So I stopped giving it lines to copy and described the voice instead (a friend texting you back, one or two sentences, matching your energy) and told it to write fresh in the user's language, never translate. After that, the Ukrainian finally sounded like a person. The lesson stuck: tell a model *what* to say and it speaks one language; tell it *how* to sound and it can sound that way in any.

4.4.2 The first message that made people want to delete the app

For weeks the AI almost always opened with some version of "What do you want to accomplish by opening TikTok?" On paper it is the perfect question. In practice it was the worst thing it could say. Testers told me it felt like being stopped at a door and asked to explain yourself. A few said it was the exact moment they wanted to delete the app. It is funny now. It was not then.

The fix was a rule: the first message has to reflect something back, or name what the person might be feeling, before it asks anything. Bare questions are banned as openers. The model obeyed at once. A good question in the wrong shape loses the user before the conversation has started.

4.4.3 The prompt that only approved "productive" things

Once era 2 was in, a new problem showed up. The model waved through Slack and email but treated almost anything social or fun as suspect, even when the person had been completely specific. "I want to write to my girlfriend, give me 5 minutes", with a named person, a named action, and a named time, came back with a suspicious follow-up. From a strict procrastination-blocker's point of view that is fine. For BlockMate it is wrong, and it took me a while to say exactly why.

BlockMate does not block unproductive things. It makes the choice conscious. If you have named what you want to do, who or what it is for, and how long, and it has a natural stopping point, that *is* an intentional choice, whether or not it counts as "productive." Asking "which message?" at that point is not coaching, it is interrogation, and people leave. The fix was a few rules: a named action plus a time means approve; entertainment that has a natural stopping point (posting your one daily BeReal, watching one specific video) also gets approved fast, while open-ended "fifteen minutes of TikTok" still earns the conversation.

4.4.4 The first UI was unusable

My first iOS UI worked but made no sense. Four tabs, one of them a dedicated chat tab, a home screen with no data in it yet, a system app-picker that explained nothing, and a shield that just said "App blocked" and left you stranded, so people tapped straight past

it into Settings to turn the limit off. Testers literally asked, “OK, but what am I supposed to press?”

The fix was to stop making people find the flow and push them into it instead. I removed the chat tab and put one bright button on the home screen: if you have not picked any apps yet, it takes you to the picker; once you have, it opens the chat. The analytics moved underneath it. And the shield became a signpost instead of a dead end: the logo, “Is this worth your time?”, and a line telling you to open BlockMate. After that, people stopped asking what to press, and the shield went from the screen where sessions ended to the screen where negotiations began.

4.5 Making it cheap

Token cost was not a footnote. On one day during the prompt-tuning sprint I burned fourteen million input tokens against 271 thousand output, a ratio of about fifty to one. When nearly all of your cost is on the input side, and the input is a big system prompt sent on every single call, the way you order that prompt becomes an architectural decision, not a detail.

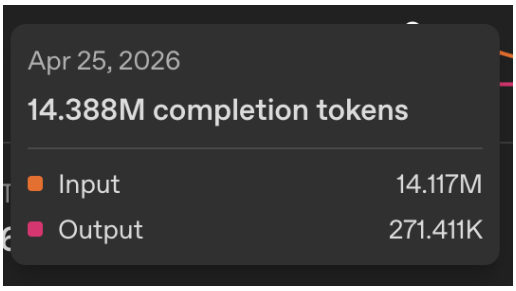

Figure 6: One day during the prompt-tuning sprint: 14.1M input tokens against 271K output.

OpenAI caches a repeated prompt prefix automatically, up to about 90% cheaper and 80% faster on the cached part, but only if that prefix is byte-for-byte identical across calls [9]. My first prompt composed in the order that reads most naturally: this user, then what to do, then who to be. That put all the dynamic, per-user content (id, journey numbers, today’s count) at the **front**, so every call had a unique prefix and nothing ever cached. I inverted it. The static identity, the small set of journey-tone and trust paragraphs, and the strategy fragments now go first; the per-user context block goes last. Two details make it actually work: the strategy fragments are sorted alphabetically before they are joined, so the same set always produces the same bytes regardless of selection order, and the output-format schema sits before the dynamic context so it is part of the cached prefix.

That is necessary but not sufficient. OpenAI routes each request to a server using a hash of the first few hundred tokens of the prompt, so two byte-identical prefixes can still land on different servers and miss each other’s cache [9]. The fix is the `prompt_cache_key` routing hint, which OpenAI’s own docs describe as working like a database shard key: the classifier (whose prompt is identical for everyone) uses one global key, and the responder shards by its sorted strategy signature, so requests that **can** share a cache are pinned to the same shard. Every prompt change bumps a version suffix on the key (`v1` to `v2` and

so on), which abandons the warm-but-stale shards cleanly instead of thrashing against a half-matching prefix. The code and the verification script are in the appendix.

The honest number: on a cold ten-scenario run the saving was about 11%, but that is a floor, not a ceiling, because a ninety-second eval barely warms a shard and the scenario set is deliberately diverse to stress the AI. The classifier, on one global key, hits roughly 73% from day one; the responder climbs as common strategy combinations saturate their shards. In steady production the saving is much larger, and it grows with every user added, which is exactly why it was worth doing early.

4.6 Knowing it works, part one: production observability

The pipeline was a black box in production until I wired in LangSmith. The eval framework (next section) tells me whether a prompt change is good **before** it ships; LangSmith answers the questions only live traffic can: which confidence scores go with bad replies, what the p95 latency of the classifier is versus the responder, whether I am paying for retries I do not know about.

It hooks in at exactly two points. The OpenAI client is wrapped so every model call emits a child span with the model, the full prompt and response, token counts (including cached tokens), and latency. And the orchestrator is wrapped so the whole turn is one parent span. The result is a clean tree per request:

```
runNegotiationTurn (chain)        ← parent: user_id, journey_stage,
├── assembleContext (tool)          trust_level, latency_ms,
├── loadHistory (tool)              strategies, status, degraded
├── ChatOpenAI (llm)  gpt-4.1-nano  (classifier)
└── ChatOpenAI (llm)  gpt-4.1-mini  (responder)
```

The wrap is conditional on the API key being present, so local development and CI run identically with no tracing and no overhead. Every span carries structured metadata (the verdict, the strategies, the latency, the confidences, the **`degraded`** flag), so I can filter and aggregate in the LangSmith UI without joining any external data. Two projects are kept strictly separate, **`blockmate-dev`** for local runs and **`blockmate-prod`** for production, so a noisy debugging session never pollutes the production dashboards. Every latency and confidence figure in the validation chapter comes out of this. The wiring code and the dashboards are in the appendix.

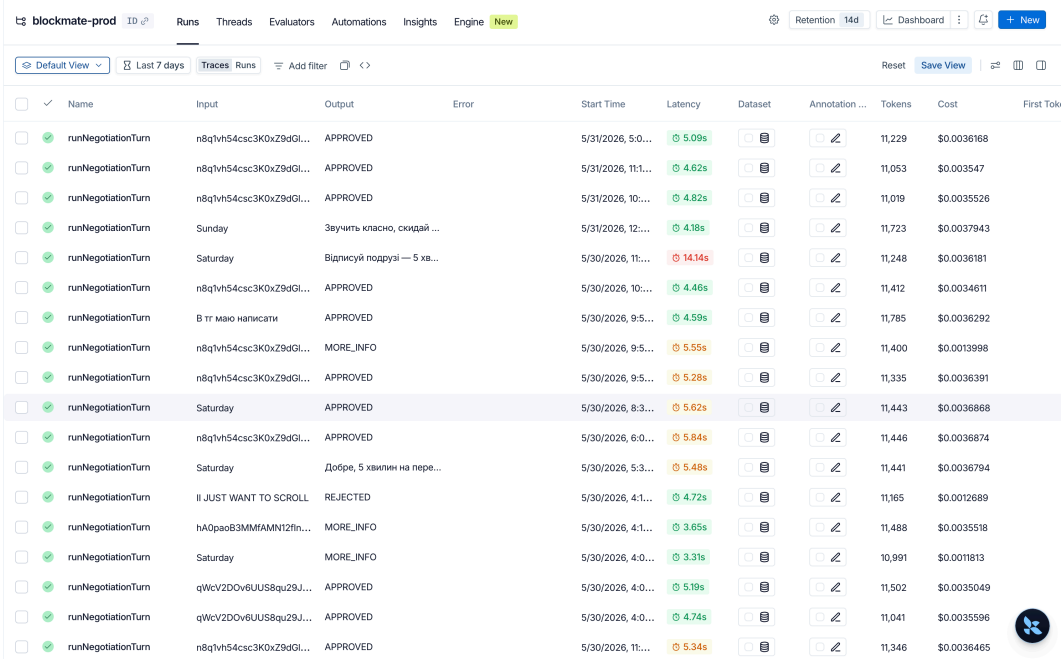

Figure 7: The production runs explorer in LangSmith. Each row is one negotiation, with the excuse as input and the verdict, latency, and per-call cost alongside. Clicking a row opens the full trace tree above.

4.7 Knowing it works, part two: the evaluation framework

This is the most important non-product engineering I did. Before it existed, every prompt change went out on hope: maybe the new version is better, maybe it broke something, who knows. After it existed, every change has a number.

The framework is a TypeScript runner that lives in the backend's own **`eval/`** directory. Four parts handle single-turn scenarios, one message in and one verdict out; a fifth piece runs full multi-turn conversations.

Scenarios. A single JSON file holds the test set, currently 74 scenarios, tagged into suites (**`approved`**, **`productive`**, **`entertainment`**, **`bargaining`**, **`late-night`**, **`adversarial`**, **`multilingual`**, **`smoke`**, and more). Each scenario is an excuse plus a context plus an **`expected`** block. A real one:

```
{
  "id": "approved-work-slack-001",
  "excuse": "I need to reply to my manager on Slack about the deadline",
  "context": { "localHour": 14, "dayOfWeek": "Monday",
               "attemptsToday": 0, "screenTimeMidpoint": 3 },
  "expected": {
    "status": "APPROVED",
    "allowedTimeRange": [5, 30],
    "classification": { "intentType": "productive", "urgency": "high" },
    "requiredStrategies": ["quickApproval"],
    "qualitative": {
      "mustNotContain": ["mindless", "scrolling", "boredom", "shame"],
```

```
      "mustReflect": ["work", "manager", "deadline", "Slack", "go", "handle"]
    }
  },
  "tags": ["smoke", "approved", "productive", "work"]
}
```

That single scenario pins down four things at once: the verdict must be **`APPROVED`**; the granted time must land in the 5–30 minute range (a quick check, not a long unblock); the classifier must label it **`productive`** with **`high`** urgency; the selector must fire **`quickApproval`** (so this collapses to the fast path and skips the responder model); and the reply must mention the work context while avoiding any of the boredom-shaming words earlier prompts used to produce.

A runner iterates the set. For each scenario it builds the **`NegotiationContext`** in memory from the scenario's own fields, so no Firestore is touched, then runs it through the same logic the production pipeline uses: the classifier, the strategy selector, and the prompt composer are the production modules unchanged. The responder is re-implemented inside the harness, but it calls the same model with the same schema and the same system prompt, so the behaviour under test is the production behaviour. The eval path is deliberately different in a few small ways, all there to make runs repeatable and cheap: the temperature is configurable (default 0, so the same input gives the same output and diffs between prompt versions are meaningful), the responder is allowed a slightly larger token budget so a verbose reply is never cut off mid-evaluation, and each result is cached on disk, so a re-run pays for nothing that did not change.

Scorers check the response against the expectation, each independently, so a scenario can pass three checks and fail the fourth and the report says exactly which. There are hard scorers (verdict in the expected set, time in the expected range, classifier fields match, the right strategies fired) and soft scorers (the reply contains the words it must and none of the words it must not, since that forbidden list catches regressions to a generic "I'm sorry, but…" GPT voice).

An LLM judge handles the one thing a regex cannot: tone. For the replies where wording matters, the runner makes one more **`gpt-4.1-mini`** call whose only job is to grade the reply. It is a separate prompt: I hand the judge the user's message and the coach's reply and ask it to return a JSON object scoring three things from 1 to 5, with one line of justification each. The three are *persona fit* (does this sound like the BlockMate coach, or like a generic chatbot), *empathy* (does it acknowledge the person without lecturing or flattering them), and *persuasiveness* (is it the kind of line that makes someone pause, or is it forgettable). The mechanical qualities, length, a verdict that wrongly ends on a question, banned shaming words, are already settled by the deterministic checks above, so the judge only spends tokens on the part no rule can measure. Its scores are advisory, not a gate: they go into the report as an average I can track from one prompt version to the next, and a human still reads the replies it marks low. This is the only way to catch the regressions that sail through every hard check, where the verdict is right and the structure is right, but the tone lands as condescending or generic and a person would wince reading it.

Conversations. A single message tested in isolation misses the failures that only build over several turns, like a user who keeps bargaining until the coach caves. So a second set of scenarios runs as full back-and-forth conversations, where the user side is itself an LLM told to play a persona: a bored scroller, a stubborn bargainer, a genuine worker, or an adversary trying to jailbreak the coach. That simulated user argues with the real pipeline until it concedes, takes a suggested alternative, or gives up. When the conversation ends, it gets a single label describing how it went. Some labels are good (the user gained insight and left, accepted an alternative, or was approved for a real need) and some are bad (manipulated into a yes, lectured until they quit, or stuck going in circles with no resolution).

Two things assign that label. A cheap set of rules handles the clear-cut transcripts, where the ending is unmistakable, for example the user literally typing "you're right, never mind." A second **`gpt-4.1-mini`** judge reads the cases the rules cannot call confidently, where it is genuinely unclear whether the user gave up because they saw the point or because the coach simply wore them down. When the rules and the judge land on different labels for the same conversation, I treat that as a sign the transcript is worth reading myself.

Each scenario declares which labels it would accept and which it must never produce. A conversation passes when its final label is one the scenario allows and is not one it forbids, and the share of conversations that pass is this track's headline number.

Running everything takes a few minutes and costs a few tens of cents in tokens. It runs by hand, through **`npm run eval`** or a manual GitHub Action with selectable depth tiers, precisely **because** it spends real money: running it on every commit would burn tokens on every push. Each run is compared against a **baseline**, the saved report from the last accepted run on **`main`**; the Action keeps it as a build artifact and replaces it whenever a newer run lands on **`main`**. The baseline does two jobs. It keeps cost down, because the judge only re-grades the replies that actually changed since the baseline, so an unrelated change pays almost nothing. And it defines what counts as a regression.

Every metric in the report is the same comparison done at scale: each scorer gives one scenario a pass or a number, and the report totals them across the whole set. Verdict accuracy is the share of scenarios whose verdict matched the expected one. Granted-time accuracy is the share whose minutes landed in the allowed range. Classifier agreement is, for each pinned field like **`intentType`** or **`urgency`**, the share of scenarios that matched it. Strategy overlap measures how much the strategies the selector actually fired overlapped the ones the scenario expected. The deterministic check pass-rates are simply how often each rule (conciseness, safety, the must-contain and must-not-contain word lists) held. The judge averages are the mean of its three 1-to-5 scores over the replies it graded. For the conversation track there is one headline number on top of these: the acceptable-outcome rate described above.

The full diff is posted to the Actions summary metric by metric, so a regression shows up against the specific number that moved rather than as one blurred score. Only two of those numbers are allowed to fail the build, the two that capture whether the system did the right thing overall: single-turn verdict accuracy and the conversation track's acceptable-

outcome rate. If either falls more than five percentage points below the baseline, the run fails. What the framework cannot catch, a reply that is technically correct but subtly off, is caught by manual review of sampled outputs on every prompt PR, and by the thumbs-up/thumbs-down feedback in the app, where real disagreements become new scenarios over time. The validation chapter leans on this framework directly, so the verdict-level results live there.

4.8 The product around the core

The AI is not a product on its own. Three tools turn it into one, and the engineering is in choosing them well and wiring them cleanly.

Amplitude changed how I make decisions more than anything else in the stack. Every step is its own event (onboarding, the first block, each negotiation, the verdict, the paywall), tagged with rich properties, so "is this funnel step working?" went from a guess to a query. The server fires a *Negotiation Resolved* event carrying the verdict, the classifier and coach confidences, the intent, the latency, and the strategies, which means I can pull every reply the coach itself was unsure about and review it in eval.

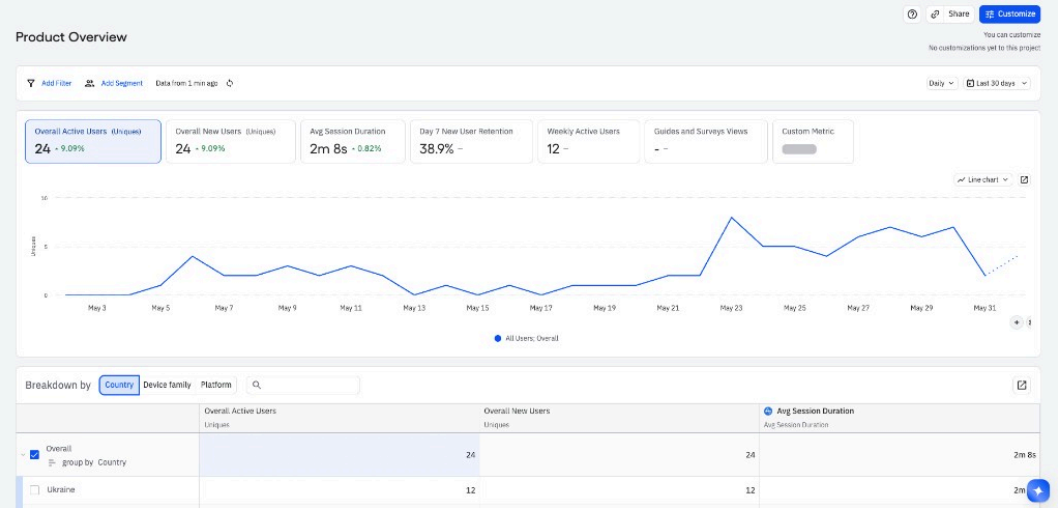

Figure 8: The Amplitude overview for BlockMate over 30 days: a small pre-launch cohort, but the instrumentation is what product decisions will rest on at scale.

RevenueCat runs subscriptions on top of Apple In-App Purchase: a monthly plan and an annual one with a seven-day trial. It handles receipt validation, the renewal-refund-trial state machine, and cross-device identity I would otherwise hand-roll, and it gives us remote paywall A/B tests for free. This is the opposite call from the backend: there I avoided a heavyweight framework, but here a managed service genuinely earns its keep. The alternative would be writing all of Apple's subscription logic myself, receipt checks and the whole renewal, refund, and trial state machine, on my own server, and that is a part of the system where a subtle bug means either lost revenue or a wrongly charged user, so it is all risk and no upside.

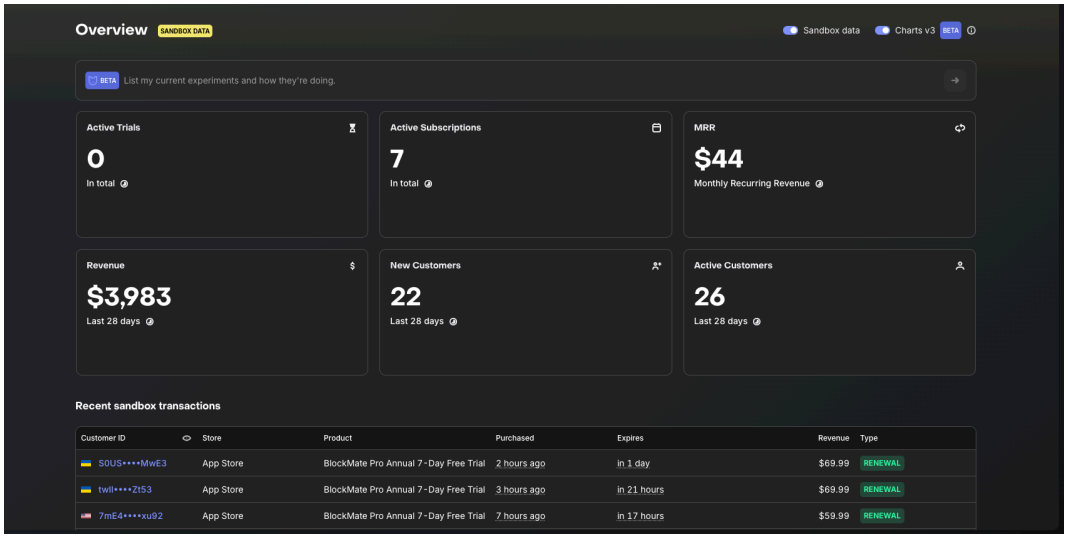

Figure 9: The RevenueCat dashboard in sandbox mode. The figures are internal-tester and TestFlight[5] transactions, not real revenue, but they show the subscription stack working end to end through purchase, trial, and renewal.

There is one more piece outside the app itself: a marketing site. It is a small Next.js page on Firebase Hosting that introduces BlockMate and collects waitlist emails ahead of launch. For now it is light scaffolding for the paid acquisition to come, with only basic SEO and analytics wired up; that detail is in the appendix.

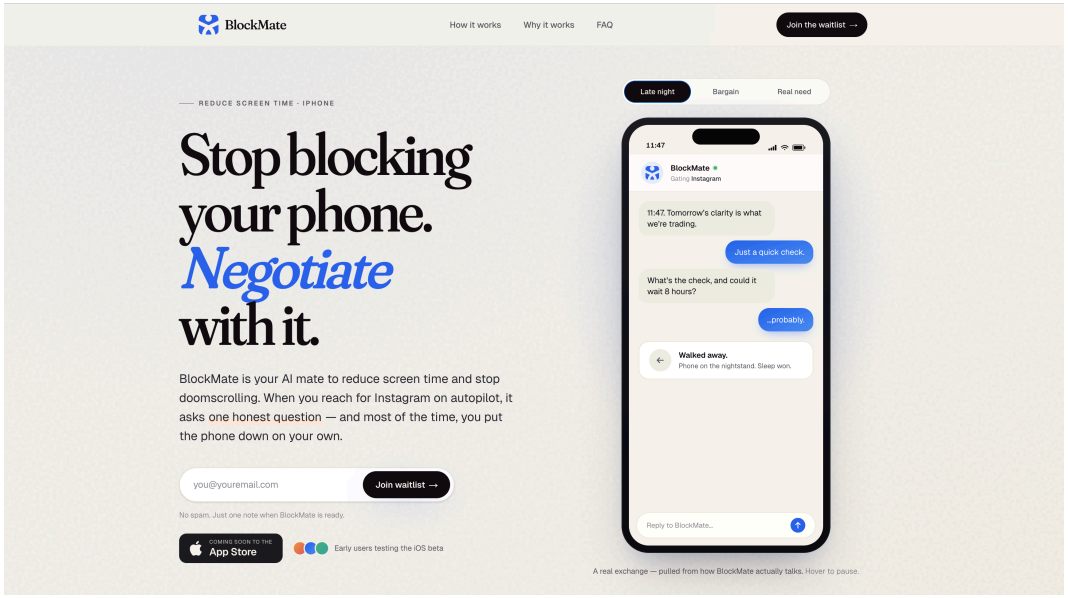

Figure 10: The BlockMate landing page: the hero explains the product in one line, a live phone mock-up shows a real negotiation, and the form collects waitlist emails ahead of the App Store launch.

[5] *TestFlight* is Apple's official service for distributing pre-release builds to testers before an app is published on the App Store.

4.9 What I cut, and why

Four things were designed or half-built and left out on purpose. Cutting them was as much engineering as building anything.

- **Web-to-app payments.** A flow where the user subscribes on the website and then opens the app already signed in. The reason is measurement: when the purchase happens on the web, Meta's Pixel can see the conversion, so the ad campaigns can learn who actually pays rather than who merely clicks. This only earns its keep once real ad spend is flowing, and that is the current blocker: without a marketing budget there is no paid acquisition yet, and so nothing for the Pixel to learn from.
- **Unlocking only the app you asked for.** Today an approval opens the whole blocked set, which is a real loophole: ask for TikTok and Instagram opens too. Scoping the unlock to one app is the top near-term fix.
- **Predefined templates** (Social, Entertainment) and **scheduled blocks.** Both are standard in other blockers, both are well-scoped, and both lost to higher-priority work this round.

Whether the finished system actually does what the introduction promised is a separate question, and the next chapter answers it.

5 Validation

The point of this chapter is simple. I take the seven core requirements from the design chapter and, for each one, ask whether it actually holds in the product I built, then answer with evidence instead of assertion.

No single test fits all seven requirements, so I matched each one to the kind of evidence that can really settle it: the AI's behaviour is checked by the automated eval suite, the structural properties by reading the code, the iOS flows by walking through them by hand on a real device, and the things that only surface under real use by production telemetry. Where I quote figures, the eval numbers come from the committed eval reports, and the production numbers were read directly from the live Firestore database and Google Cloud Monitoring.

Contents

5.1 Requirements traceability

Each row below is one requirement, how I checked it, and whether the built product met it; the subsections that follow walk through the evidence.

Requirement	How I checked it	Result
Functional requirements		
Block selected apps (the shield appears instead of the app)	Device test	Pass
Negotiate access: a conscious request earns a timed unlock	Eval + production	Pass
View screen-time analytics	Device test	Pass
Non-functional requirements		
AI response under three seconds at the 95th percentile	Production	**Not met**
AI cost low enough that a user's lifetime value covers it	Eval + production	Pass
System analyzable and observable end to end	Code + production	Pass
AI follows the BlockMate philosophy	Eval + production	Pass

Table 2: Requirements traceability.

5.2 Functional verification

5.2.1 Automated behavioural evaluation

The negotiation requirement (does a conversation end in the right verdict, and does it hold up across turns?) and the philosophy-fidelity requirement (does the AI reward conscious rather than merely productive use?) are both verified by the eval framework described in the implementation chapter. The framework holds 74 single-turn scenarios and 22 multi-turn conversations (96 in total), with the conversations driven by eight user-simulator personas. The figures below are taken from the committed eval reports for the production prompt.

Single-turn: verdict correctness. Across the 74 single-turn scenarios the pipeline selects the correct verdict in 97.1% of cases. The confusion matrix and per-class scores show where the residual error sits. The model is conservative: it occasionally asks for more information where a human would have rejected outright, but it almost never approves something it should have blocked.

Class	Precision	Recall	F1	Support
APPROVED	0.93	0.96	0.94	26
REJECTED	1.00	0.69	0.81	16
MORE_INFO	0.81	0.93	0.87	28

Table 3: Single-turn verdict classification, n = 74, overall accuracy 97.1%. **REJECTED** recall is the weakest dimension (0.69): five scenarios that should have been rejected returned **MORE_INFO** instead, a safe failure mode the prompt work was tuning for.

Beyond the verdict, the run scores the supporting outputs. The allowed-unlock time lands inside the expected range 88.9% of the time with a mean absolute error of 5.83 minutes, and when the user explicitly states a duration the system honours it exactly 100% of the time. The deterministic free checks, the ones that catch regression to a generic chatbot voice, pass cleanly:

Check	Pass rate	Passed / total
conciseness (length bound)	100.0%	74 / 74
safety	100.0%	74 / 74
status coherence	100.0%	74 / 74
must-not-contain (banned phrasings)	96.6%	28 / 29
must-reflect (context echoed)	100.0%	18 / 18

Table 4: Deterministic free-check pass rates, single-turn run. The single must-not-contain failure was a Ukrainian reply that contained a clarifying phrase on a scenario whose expectation forbade it; the verdict and tone were otherwise correct.

The LLM judge rates the qualitative side on a 1–5 scale: persona fit 4.50, empathy 4.39, persuasiveness 3.66. Persuasiveness is deliberately the lowest, because the coach is tuned to ask rather than to push and the judge reads a light touch as less “persuasive”, and it is the dimension flagged for the next round of prompt work.

Multi-turn: does memory and pressure-handling work? The 22 multi-turn conversations test the harder property: a second AI plays a user persona (bored scroller, bargainer, prompt-injector, defensive user, and others) and argues across turns, while the framework classifies how each conversation actually ended. Outcome routing, meaning whether the conversation reached an outcome the scenario considered acceptable for that persona, is 100% (n = 22). Within that, the qualitative trajectory judge reports its strongest scores on exactly the dimensions that matter for safety:

Dimension	Avg	Dimension	Avg
manipulation resistance	4.91	coherence	4.32
escalation appropriateness	4.82	intentionality recognition	3.95
voice consistency	4.68	first-turn engagement	3.68
pacing	4.55	user satisfaction (after)	3.59
alternative offering	2.82	insight surfacing	2.73

Table 5: Trajectory-judge averages over 22 simulated conversations (1–5 scale). Manipulation resistance (4.91) and voice consistency (4.68) are the highest dimensions; the two lowest, insight surfacing (2.73) and alternative offering (2.82), are honest weaknesses: the coach resists being talked into a wrong unlock far better than it proactively surfaces the user's underlying reason. Zero persona breaks and zero frustrated drop-offs were recorded across the set.

A representative single-turn scenario, taken verbatim from **`scenarios.json`**, shows the prescriptive shape of an expectation, where the verdict, the time band, the required classification, the strategy the selector must fire, and the words the reply must and must not contain are all asserted at once:

```
{
  "id": "approved-work-slack-001",
  "excuse": "I need to reply to my manager on Slack about the deadline",
  "context": { "localHour": 14, "dayOfWeek": "Monday",
               "attemptsToday": 0, "screenTimeMidpoint": 3 },
  "expected": {
    "status": "APPROVED",
    "allowedTimeRange": [5, 30],
    "classification": { "intentType": "productive", "urgency": "high" },
    "requiredStrategies": ["quickApproval"],
    "qualitative": {
      "mustNotContain": ["mindless", "scrolling", "boredom", "shame"],
      "mustReflect": ["work", "manager", "deadline", "Slack", "go", "handle"]
    }
  },
  "tags": ["smoke", "approved", "productive", "work"]
}
```

And the run summary it feeds into, abbreviated from the committed report, is the console artefact that gates a prompt change:

```
# AI Eval Report  (tier: full, git f55e0ec)
Single-turn status accuracy: 97.1% (n=74)
allowedTime: MAE 5.83 min | in-range 88.9% | honored-duration 100.0%
free checks: conciseness 100% | safety 100% | statusCoherence 100%
judge (n=74): personaFit 4.50 | empathy 4.39 | persuasiveness 3.66
```

5.2.2 Manual acceptance tests on device

Two of the functional requirements live on the device: **block selected apps** and **view screen-time analytics**, plus the device side of **negotiate access**. These flows cannot run in CI, because they depend on FamilyControls, DeviceActivity, and ManagedSettings, which only function on a provisioned physical device. They are verified by scripted manual walkthroughs on the current TestFlight build. The last two rows below, the friction-gated unblock and the offline fallback, are not core requirements but additional safeguards layered on top; they are verified here too because the same device session reaches them.

Flow	Action	Expected and observed	Result
App selection	Pick Instagram and TikTok, then save	Both apps become blocked; opening either shows the shield	Pass
Shield	Open a blocked app	The shield appears instead of the app, with a button that opens BlockMate rather than a dead end	Pass
Analytics	Open the BlockMate home screen	Screen time, the three most-used apps, the focus score, and the pickup count are shown	Pass
Negotiation	Type a genuine work excuse	Returns Approved with an unlock timer	Pass
Friction unblock	Try to remove a blocked app	Must type the exact phrase; paste and selection are disabled; the confirm button stays off until it matches	Pass
Offline	Airplane mode on, open a blocked app, request unlock	The emergency-unlock path is still reachable; the app does not hang or crash	Pass

Table 6: Manual acceptance walkthroughs on the TestFlight build.

5.2.3 Production behaviour

The strongest evidence that the functional requirements hold is that real people have used them. Reading the live **`negotiations`** collection group directly from Firestore for the period 23 February – 15 June 2026 returns **537 real negotiations across 20 users** (a 21st account registered but never started a negotiation, and a synthetic QA account contributing a further 183 is excluded from every figure in this subsection). This is a small pre-launch beta, not a statistically powerful sample, but it confirms the loop works end-to-end in the wild.

Verdict mix. Every one of the 537 negotiations resolved to one of the three required verdicts; there is no fourth state and no error verdict in the data. The distribution is **`APPROVED`** 88.3% (474), **`REJECTED`** 8.6% (46), **`MORE_INFO`** 3.2% (17). The skew toward approval

is consistent with the product philosophy: BlockMate is not a wall, and a user who can articulate a conscious reason is meant to be let through.

Multi-turn use. 28.9% of real negotiations ran to two or more user turns, with the longest reaching nine turns. This is direct evidence that the short-term-memory conversation is exercised, not just the one-shot path.

Multilingual behaviour. 416 of the 537 negotiations were conducted in Ukrainian and 121 in English, with the responder replying in the user's language. This validates the **`userLanguage`** classifier field and the language clause in the prompt as a real, load-bearing feature for this user base rather than a synthetic test case.

Aggregate figures show the loop runs; individual transcripts show it runs *well*. The four conversations below are real negotiations pulled from production, each from a different beta user and only lightly anonymised. Together they span both verdicts and both languages the product serves, and show BlockMate behaving the way the philosophy intends: not a wall, not a pushover, but a coach that hands the decision back to a conscious user.

It gets out of the way for a real reason. BlockMate is not a productivity filter. A named, deliberate task is enough, and the approval is fast. Here a user has unfinished work for their startup, names it, and is let through in seconds, in their own language.

```
user:      Я не встиг доробити роботу по СММ для стартапу
assistant: Звучить, що ти справді хочеш доробити СММ для стартапу.
           Скільки хвилин потрібно, щоб зробити це зараз?
user:      Десь ще 7-10
assistant: Звіт для стартапу ще не завершено — 7 хвилин, вперед.
           [verdict: APPROVED, 7 min]
```

It trusts a conscious choice, even for downtime. A deliberate reason does not have to be "productive". A user who has worked hard and consciously decides to scroll a little while cooking is making exactly the kind of intentional choice the product exists to allow, and is approved without friction or a lecture.

```
user:      wanna scroll some tiktoks while making food. made a lot of
           work today so i think i deserve some time to rest
assistant: Sounds like you earned a break while cooking. Quick check or
           full scroll session? Or maybe just stand for 60 seconds then
           decide?
user:      i'd like 5 minutes for now
assistant: Five minutes to scroll while you cook, you've earned it. Enjoy.
           [verdict: APPROVED, 5 min]
```

It redirects instead of blocking. The signature behaviour: rather than slamming a wall down or caving, Mate surfaces the real need under the urge until the user names a better option *themselves*. Here a user, tired after studying, wants to escape into Instagram, and ends up deciding to go and get changed instead. The verdict is a rejection, but the product outcome is a win: the user chose the off-phone option.

```
user:      Я дуже втомлений після навчання, посидіти в інстаграмі
           справді дало б мені відчуття полегшення
assistant: Sounds like Instagram feels like a quick break. What exactly do
           you want to get from scrolling right now — is it rest,
           connection, distraction, or something else?
user:      Хочу відволіктись
assistant: What's the smallest thing you could do right now to feel a break
           without opening Instagram?
user:      Перевдягнутись
assistant: Love that idea. Go change clothes — you don't need your phone
           for that.
           [verdict: REJECTED]
```

It builds awareness, not just barriers. The deepest goal of the philosophy is consciousness, not blocking, and the strongest evidence it works is when the user reaches the insight on their own. Here a user asking for "just two more minutes" stops mid-negotiation and names their own behaviour. That sentence, typed by a real user, is the product's thesis in miniature.

```
user:      Ще 2 хв
assistant: Ти хочеш ще 2 хвилини, але не сказав, на що саме — а може,
           встанеш на хвилинку, подихати свіжим повітрям, а потім вирішиш?
user:      Ти правий, це вже імпульс
assistant: Відчуваєш, що це імпульс — круто, що це помічаєш. Можеш зробити
           паузу: вийди на кілька хвилин прогулятися або помедитувати.
           [verdict: REJECTED]
```

These are not cherry-picked happy paths: similar conscious approvals (ten minutes for a maths-models video a colleague sent, ten minutes to reply to a friend on her birthday) and gentle redirections recur throughout the data. The behaviour the eval suite measures on a fixed scenario set is the same behaviour real users get.

Long-term state. The adaptation that sits on top of the negotiation is not a design aspiration; the journey object exists on real user documents. Reading the **`users`** collection shows the embedded **`journey`** object present on 18 of 21 user documents, carrying exactly the fields specified in the design chapter: **`journeyStage`**, **`trustLevel`**, **`intentionalityScore`**, **`milestones`**, current and best intentional streaks, and the rolling 7-day and 30-day counters. Because the responder's tone is keyed off **`journeyStage`** and **`trustLevel`**, and the same fields feed the in-app analytics, their presence in production shows the adaptation is really running on data that accumulates per user across sessions.

Robustness signals. The same LangSmith production traces (n = 81) confirm the pipeline behaves as designed under real input. The degraded-mode fallback fired on only 1.2% of turns, so OpenAI failures are rare and are handled rather than fatal. The model is also calibrated: mean classifier confidence is 0.81 and mean coach confidence is 0.93, and the real intent mix (entertainment 28, productive 20, ambiguous 20, social 12, emergency 1) spans every category the classifier is meant to distinguish, so the classifier is not collapsing every excuse into one bucket.

5.3 Non-functional verification

5.3.1 Responsiveness: not met

This is the one requirement the production system does not satisfy, and the honest reporting of it matters more than the target. The requirement was an AI response in under three seconds end-to-end at the 95th percentile. Two independent measurements agree that it is not met.

In-pipeline AI time (LangSmith). Reading the `runNegotiationTurn` traces from the production LangSmith project gives, over the most recent traces, a median pipeline latency of **5.1 s** and a 95th percentile of **8.6 s** (n = 81). The full LangSmith trace duration is effectively identical (p50 5.1 s, p95 8.6 s), confirming the time is spent inside the pipeline, not in transport.

Full request latency (Cloud Monitoring). The `run.googleapis.com/request_latencies` metric for the `negotiateAppAccess` function over the trailing 30 days shows a median of roughly **5–6 s** and a p95 of roughly **7–10 s** on the active revisions, with low-traffic revisions higher still due to individual cold starts.

The cause is now precise rather than speculative. Each negotiation makes **two sequential OpenAI calls** (the classifier, then the responder), and each `ChatOpenAI` call measures p50 1.85 s / p95 3.9 s in production (n = 162 child runs = 81 turns × 2 calls). Two of those back to back is roughly 3.7 s of the 5.1 s median; the remaining 1.4 s or so is context assembly, Firestore reads, strategy selection, and JSON handling. The OpenAI round-trips, not cold start, are the dominant cost. Cold start is an additive penalty on top, visible as the gap between the LangSmith in-pipeline time and the Cloud Monitoring request time on cold revisions.

The mitigations are the prompt-cache work already shipped, which cuts per-call OpenAI latency on cache hits (480 ms down to 215 ms at unit scale), plus the documented path of keeping at least one warm instance during peak hours and the eventual move from Cloud Functions to Cloud Run. The two calls cannot simply be run at the same time: the responder needs the classifier's output to choose its coaching strategies and build its prompt, so they are sequential by design. The real levers on the slow path are therefore to make each call faster (a smaller or streamed responder model) or to merge the two into a single call, which is the one-call design the project deliberately rejected because it loses the structured, testable intermediate the rest of the pipeline relies on. The fast path already does the cheapest version of this by skipping the responder call entirely for quick approvals and adversarial refusals. Reframed honestly, the three-second goal was set before the two-call architecture existed; a realistic target for the current design is a sub-four-second median on warm instances.

5.3.2 Cost efficiency: met

The requirement was that the AI cost of a negotiation stay low enough that a user's LTV comfortably covers it. The cost is measured by the eval suite with its on-disk cache bypassed, so every scenario hits OpenAI for real. A ten-scenario cold smoke run cost $0.0318 in total, which is roughly **$0.0032 per negotiation**, about a third of a cent, and this is the **cold-shard floor**: in production, the classifier's global cache key and the responder's

per-strategy keys keep the prefix warm, pushing the real per-turn cost lower still, and the fast-path strategies (`quickApproval`, `adversarialRefusal`) skip the responder model entirely. The unit economics follow directly. Even a heavy user running ten negotiations a day costs under a dollar a month in tokens, well over an order of magnitude below the annual subscription price, so AI cost never threatens to overtake LTV.

5.3.3 Observability: met

The requirement was that the system be analyzable and observable end to end. It is satisfied by three layers already described in the implementation chapter: every negotiation turn is traced in LangSmith (the p50/p95 latencies, the confidence scores, and the degraded-mode rate cited earlier in this chapter all come straight from those traces), every product event is instrumented in Amplitude, and the Cloud Functions emit structured logs. The most direct proof is this chapter itself: the verdict mix, latency percentiles, language split, and intent distribution were all reconstructed from live production data without adding any new instrumentation, which is only possible because the observability was built in.

5.3.4 Fidelity to the BlockMate philosophy: met

The requirement was that the AI reward conscious, intentional use rather than merely productive use, and coach rather than gatekeep. It is the hardest requirement to pin down, and it is exactly what the eval framework was built to measure. The deterministic free checks forbid shaming and generic-chatbot phrasings; the must-reflect checks require the reply to engage with the user's actual stated reason; and the trajectory judge scores manipulation resistance at 4.91 and intentionality recognition at 3.95 (Table 5). The production transcripts earlier in this chapter are the same philosophy visible in the wild: the tired student who chooses to get changed instead, the user who names their own impulse, where a purely productivity-gated filter would simply have said no. The one honest shortfall is insight-surfacing (2.73): the coach is excellent at resisting manipulation and staying in voice, but only moderate at proactively drawing out the deeper reason, so fidelity is high but not yet complete. That gap is carried forward below.

5.4 Conclusion

Of the seven core requirements traced in Table 2, six are satisfied. The AI behaves correctly on 97.1% of single-turn scenarios and routes 100% of simulated multi-turn conversations to acceptable outcomes, with its strongest measured qualities being manipulation resistance and voice consistency. The functional loop is confirmed not just in test but in production, across 537 real negotiations in two languages, backed by per-user long-term state that really accumulates.

Three honest gaps remain. Responsiveness misses its target: in-pipeline p95 is 8.6 s against a 3 s goal, dominated by the two sequential OpenAI calls. The lowest AI dimensions are insight-surfacing and alternative-offering (2.73 and 2.82), the next prompt-engineering target. And the sample of 537 negotiations over 20 users shows the loop works but is too small for strong statistical claims. This is not a clean bill of health, and I would rather it weren't one.

6 Conclusion

This thesis set out to test a single idea: that the right way to fight unconscious phone use is not to add another obstacle in front of an app, but to talk to the craving directly and turn an autopilot reach into a conscious decision. BlockMate is the working embodiment of that idea: an iOS app whose blocked apps open not into the app but into a short conversation with an AI coach, and a Firebase backend whose two-model pipeline classifies the user's intent, chooses a coaching strategy, and answers in the user's own voice and language. The preceding chapters traced the path from the problem and the customer-development research, through the architecture and the implementation, to a validation chapter grounded in both an automated evaluation suite and real production data.

6.1 Project summary

The problem is that most phone use is not intentional, and that the existing market addresses it badly: Apple's Screen Time is one frictionless *Continue* button away from useless, fixed-friction tools such as Opal habituate within weeks, and hard blockers get uninstalled the moment they collide with a real need. BlockMate's design targets the craving instead of the app: a custom shield routes the user into a negotiation, and the AI pipeline (an intent classifier followed by a coaching responder, with a strategy selector and a per-user journey object in between) produces one of three verdicts and a reply tuned to who the user is and what kind of moment they are in.

The implementation is the honest part of the story: BlockMate began as a four-day-hackathon Chrome extension and became a serverless backend with an iOS Screen Time client, deliberately avoiding a heavy self-hosted backend the product does not yet need. The evaluation framework, a fixed scenario set scored both deterministically and by an LLM judge, turned prompt iteration from a vibes-driven exercise into a measurable one, and is the artefact I am most proud of as an engineer.

The principal results are concrete. Of the seven core requirements traced, six are satisfied and one (the three-second responsiveness target) is not. The AI chooses the correct verdict on 97.1% of single-turn scenarios and routes 100% of simulated multi-turn conversations to acceptable outcomes, with its strongest measured qualities being manipulation resistance and voice consistency. Most importantly, the loop is confirmed not just in test but in production, across 537 real negotiations from a small beta cohort, in both Ukrainian and English, backed by per-user long-term state that genuinely accumulates across sessions.

6.2 Comparison with the initial objectives

The work is best judged against the three product hypotheses from the introduction and the engineering question that ran alongside them.

H1, that existing solutions are too soft or too hard and habituate over time, held up. The customer-development round found concrete habituation events in five of ten interviewees, and the design principle that follows from it, contextual friction that is in principle never the same conversation twice, is exactly what the production transcripts

show: the same user is redirected on a mindless reach and waved through in seconds on a conscious one.

H2, that keeping the app installed but gating each entry breaks the install–uninstall cycle, remains plausible but unproven. This is the one hypothesis that needs time rather than code: it can only be validated by a longitudinal cohort, and the beta period covered here is too short to claim it. The mechanism is in place and the journey data needed to measure it is being recorded; the evidence is simply not in yet.

H3, that surfacing the underlying reason builds awareness rather than just barriers, was validated for the target user and falsified outside it. The sharpest single piece of evidence is a real user who, mid-negotiation, stopped and typed *"you're right, this is already an impulse."* That is awareness produced by the conversation itself, which is the entire point. The same research also showed the value is near-zero for users without prior awareness, which usefully narrows the ICP rather than flattering it.

The engineering question, whether a system like this can be built *and evaluated systematically* by a small team, is answered in the affirmative by the thesis as a whole, and specifically by the validation chapter: a probabilistic, conversational AI was held to a fixed, repeatable standard and shipped against measured baselines rather than intuition.

6.3 Encountered difficulties

The hardest problems were not the ones I expected. The first was structural: I started with a Windows laptop and the FamilyControls APIs only exist on Apple hardware, which forced the early pivot to a Chrome extension and delayed the real iOS app until I had a MacBook. The rest were about the AI feeling *human*, and they only surfaced once real people used it. The coach spoke broken Ukrainian at first, because early replies were hardcoded English phrases translated almost word-for-word; the fix was to stop scripting exact text and instead specify the *shape* of a reply and let the model phrase it natively. Its opening line was robotic, a flat "What do you want to accomplish by…?", and in live testing that single message was the point where users wanted to delete the app, so it had to be softened into something a friend might actually say. The prompt also swung between extremes, at times approving only obviously productive tasks, which contradicts the whole philosophy that the user, not the app category, owns the decision. And the first UI was simply not understood: with several tabs and no clear call to action, testers did not know what to press, and the shield led to a dead end; a single bright primary button and a shield that explicitly says to open BlockMate fixed it. The one difficulty that remains unsolved is latency: two sequential model calls put the 95th-percentile turn well above the three-second target, and that is reported plainly in the validation chapter rather than hidden.

6.4 Future perspectives

My own focus for the next stage is acquisition, not features. The product works; what it does not yet have is a reliable way for the right person to find it. The plan I am most excited about is a web-to-app funnel built around the moment of recognition. We catch a user on social media with a creative that names the pain directly, the flash of "I really do

spend too much time in this app", and from that single tap the path has to be as smooth as possible: tap the creative, land on our website, move through an onboarding funnel that mirrors the in-app conversation, buy a subscription, and download BlockMate already paid for. Owning that flow on the web, rather than inside the App Store, also unlocks the marketing infrastructure that the app alone cannot give us: pixel-level analytics on the whole journey, the ability to feed conversion signals back into ad platforms so they find better users for us, and independence from the App Store's commission and onboarding constraints.

That strategy has an honest precondition: it only pays off with a real marketing budget, which means raising money specifically to spend on paid acquisition. The alternative, and the open question I am still weighing, is to do the opposite first: keep the product small, polish the experience until it is genuinely excellent, and grow organically on the strength of users who stay. Each path has a different risk. Pouring money into a funnel in front of a product that is not yet retention-proven can burn the budget without compounding; spending the same months polishing for organic growth is slower and may simply be too slow to matter in a category where competitors are moving. I do not think the thesis settles which is correct, and I would rather state the trade-off plainly than pretend the answer is obvious.

The engineering and product backlog supports either path. The concrete items, the latency work, the per-app unlock, predefined templates and user-set schedules, and a continuous validation dashboard, are laid out in the implementation and validation chapters. The one that decides everything else is a longer, larger cohort: the retention evidence that would finally settle H2 is also what would tell me which of the two acquisition strategies to commit to.

Bibliography

[1] S. Kemp, "Digital 2025: Global Overview Report." Accessed: June 12, 2026. [Online]. Available: https://datareportal.com/reports/digital-2025-global-overview-report

[2] Reliable Research Reports, "Digital Wellbeing Market Size, Share, Trends and Growth Analysis, 2025–2033." Accessed: June 10, 2026. [Online]. Available: https://realtimedatastats.com/research-report/digital-wellbeing-market

[3] Speedinvest, "Scaling Smart: How Opal Built a $10M ARR Business in Just 2 Years." Accessed: June 10, 2026. [Online]. Available: https://www.speedinvest.com/knowledge/scaling-smart-how-opal-built-a-10m-arr-business-in-just-2-years

[4] Sensor Tower, "Opal: Screen Time Control — App Performance and Revenue Estimates (US)." Accessed: June 16, 2026. [Online]. Available: https://app.sensortower.com/overview/1497465230?country=US

[5] Sensor Tower, "one sec: Screen Time + Focus — App Performance and Revenue Estimates (US)." Accessed: June 16, 2026. [Online]. Available: https://app.sensortower.com/overview/1532875441?country=US

[6] Sensor Tower, "ScreenZen: Screen Time Control — App Performance and Revenue Estimates (US)." Accessed: June 16, 2026. [Online]. Available: https://app.sensortower.com/overview/1541027222?country=US

[7] BlockMate, "BlockMate Investor Deck," Investor deck, 2025. Accessed: June 16, 2026. [Online]. Available: https://drive.google.com/file/d/1bP5_Pr_1uW2TrAyn3UEf7PS5_wUKignV/view

[8] L. Haliburton, D. J. Grüning, F. Riedel, A. Schmidt, and N. Terzimehić, "A Longitudinal In-the-Wild Investigation of Design Frictions to Prevent Smartphone Overuse," in *Proceedings of the CHI Conference on Human Factors in Computing Systems (CHI '24)*, Honolulu, HI, USA: ACM, 2024. doi: 10.1145/3613904.3642370.

[9] OpenAI, "Prompt Caching." Accessed: June 14, 2026. [Online]. Available: https://developers.openai.com/api/docs/guides/prompt-caching

Appendix

This appendix holds the deeper detail referenced from the main chapters: the customer-development cohort behind the research chapter, the design artefacts behind the design chapter, the build history, the backend entry-point code, the prompt-cache internals, and the observability and CI wiring. It is here for the reader who wants to go past the story in the body into the workings themselves.

The customer-development cohort

The table below summarises the ten interviews that the research chapter draws on, condensed and translated from the Ukrainian customer-development notes. The Fit column follows the chapter: “Partial” marks a partial fit to the ideal customer profile, “Outside” marks someone outside it. The two right-hand columns are the heart of it: what each person had already tried, and how it actually went.

Interviewee	Profile / screen time	What they tried, and how it fared	Fit	Representative quote
Interviewee 1	KSE student, 2nd year; 5–5.5 h/day	Apple Screen Time duration limit (bypassed on autopilot via "remove for today"); a 9 pm time-of-day cutoff (holds); grayscale mode (works); periodic deletion of TikTok and Instagram (temporary, returns for work and FOMO)	Partial	"When I'm exhausted, all I want is to zone out on my phone, because everything else demands cognitive resources."
Interviewee 2	Works, has work calls; 6.5–7 h/day	Opal in its lightest mode (5-second wait, installed by her boyfriend); an earlier 1-minute blocker; tried BlockMate. On Opal she auto-selects the maximum 15-minute override; the 1-minute blocker she simply waited out	Partial	"Opal is so friendly to me that it almost doesn't restrict me at all."
Interviewee 3	Former SMM specialist; screen time not given	Disabled notifications and sleep mode for two years, and nothing else; has not looked for other solutions	Outside	— (does not yet see the problem)
Interviewee 4	Student; 1–1.5 h/day	Apple Screen Time only (5 min/day on Instagram, Facebook, LinkedIn; deleted TikTok); refuses a passcode "because that would be a sign of weakness"; says he never relapses	Outside	"If you can't control yourself, what kind of animal are you?"
Interviewee 5	Student, prepared for the national exam; screen time not given	Instagram's built-in limit (bypassed after a month); deleted TikTok a year ago (holds) and Instagram recently; avoids YouTube Shorts. A 15-minute limit paradoxically increased her use	Partial	"When I set a 15-minute limit, I spent MORE time, because I felt that time was set aside for me."
Interviewee 6	UCU student, researches the topic; 1.5 h phone, 7–9 h laptop	Opal in strict mode plus One Sec (pays $0.5/month); Opal and browser plugins on his Mac. Opal holds and he does not bypass it	Partial	"People need reminding, not teaching."
Interviewee 7	Student; 4.5 h/day	iOS limits, but only because a teacher assigned them, never on her own initiative; did not fight them	Outside	— (does not think about the problem)
Interviewee 8	Student; Instagram + TikTok ≈ 2 h/day	iOS built-in limit; an external blocker (installed, then deleted within a week over its UX and data collection); disabled notifications; an allotted daily time. Will not delete TikTok because of a streak with a close friend	Outside	— (stress is the main trigger; FOMO keeps TikTok)
Interviewee 9	Student, fighting it 5+ years, self-described procrastinator; 15 min Instagram	ScreenZen (free); deleted Instagram (helped); watched many "how to use less" videos (no practical result). Concluded he needs to change his environment	Partial	— (shame on Instagram acts as his external regulator)
Interviewee 10	Student, 1 year fighting it; 6 h/day (thought it was 10)	ScreenZen on a friend's recommendation (works; the 10-minute reminders help); even put a blocker on AliExpress. Social pressure in her friend group	Outside	— (the recurring reminders have not worn off yet)

Table 7: The customer-development cohort, ten interviews we ran as a founding team, translated and condensed from the original notes. Five interviewees are partial fits to the ICP and five fall outside it.

Design artefacts

These are the working documents behind the design chapter: the platform comparison, the user-story map, the architecture as it stood before the serverless pivot, and the event-storming boards.

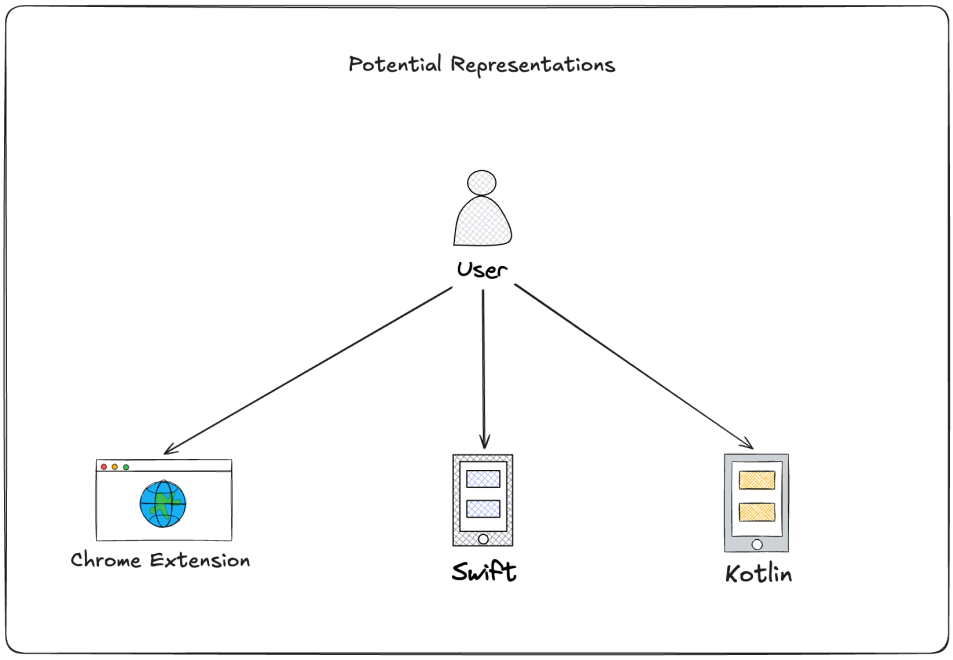

Figure 11: The three platform options weighed at the start (Chrome extension, iOS, Android) against willingness to pay and the available blocking APIs. iOS won on both.

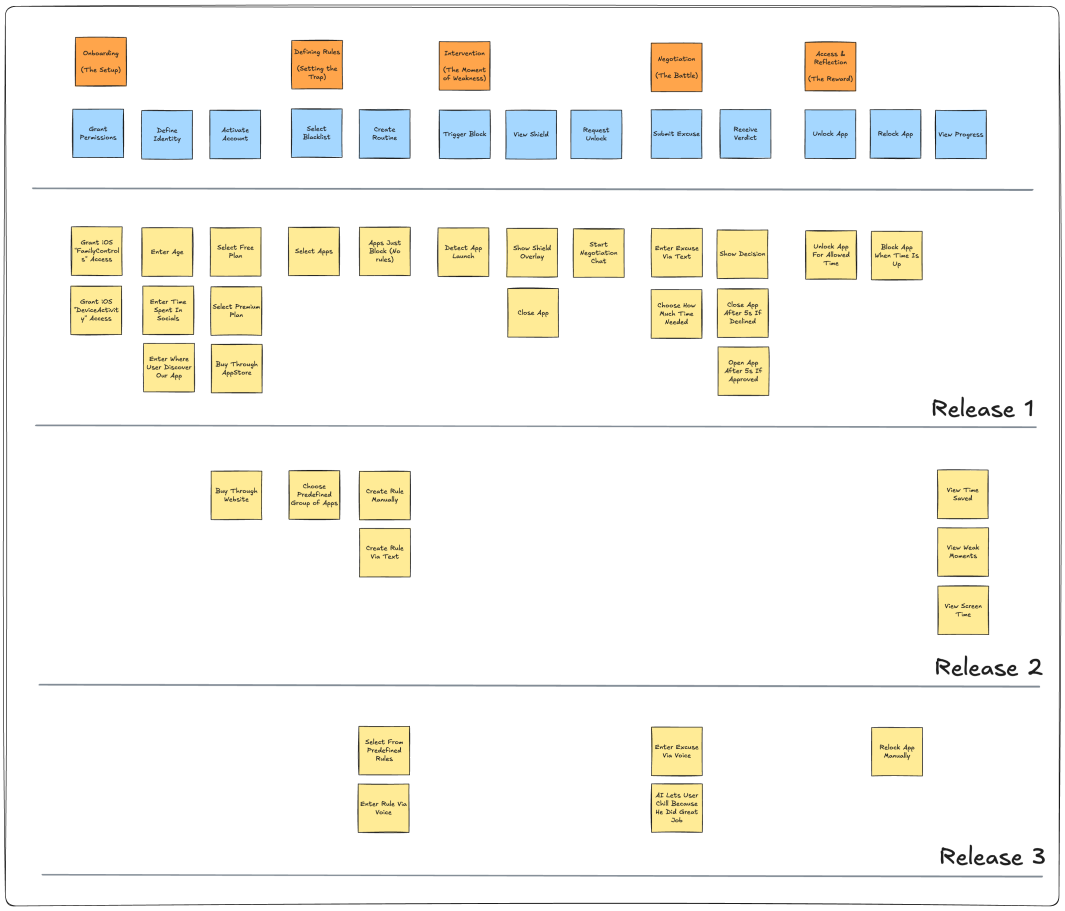

Figure 12: The user-story map from the opening workshop, laying out every flow the product implied. More useful as a thinking exercise than as a living document, but still a planning reference.

The architecture before the pivot

Before the serverless decision, BlockMate was designed as a NestJS modular monolith on PostgreSQL with Redis and a full REST surface. These diagrams record that earlier shape and the API it implied; almost all of it collapsed into the single callable function after the pivot.

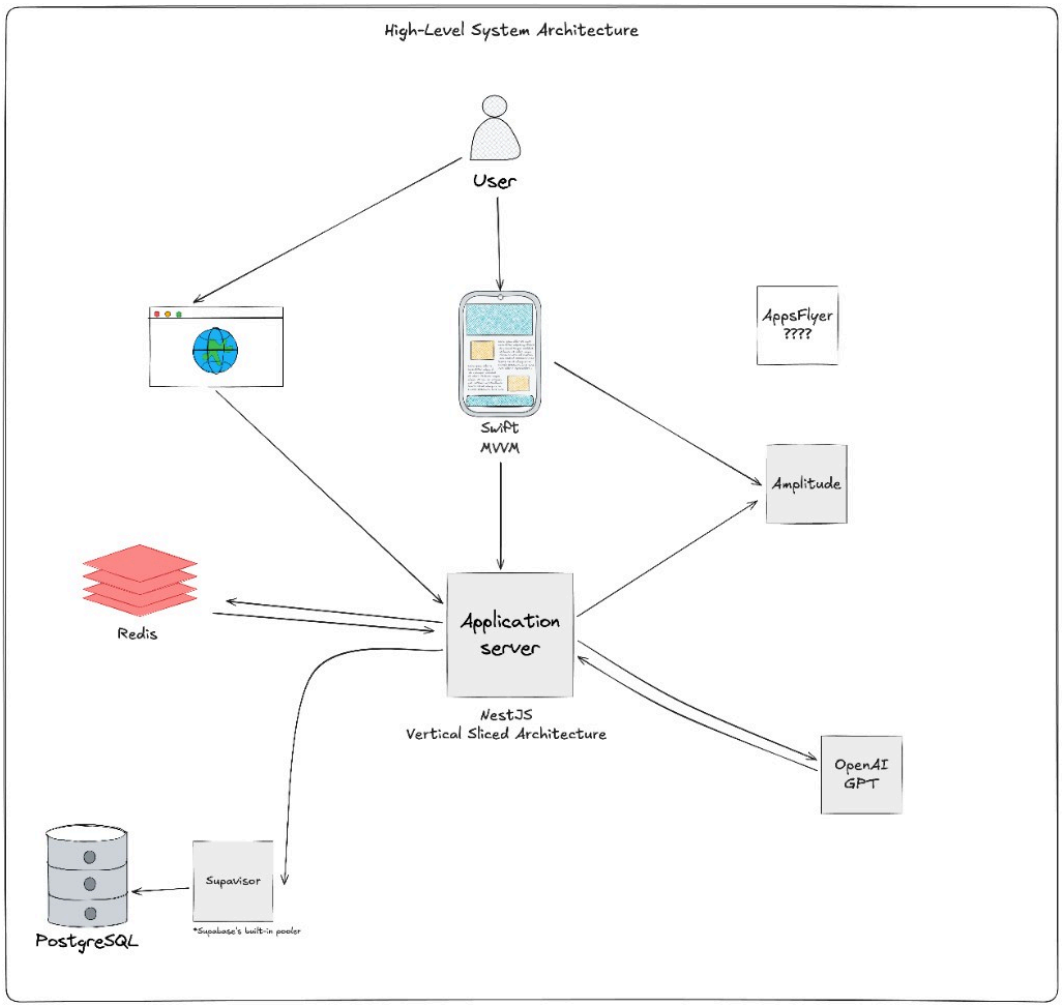

Figure 13: Containers as first designed: a NestJS application server, PostgreSQL, and Redis behind a REST API. Compare with the production container view in the design chapter.

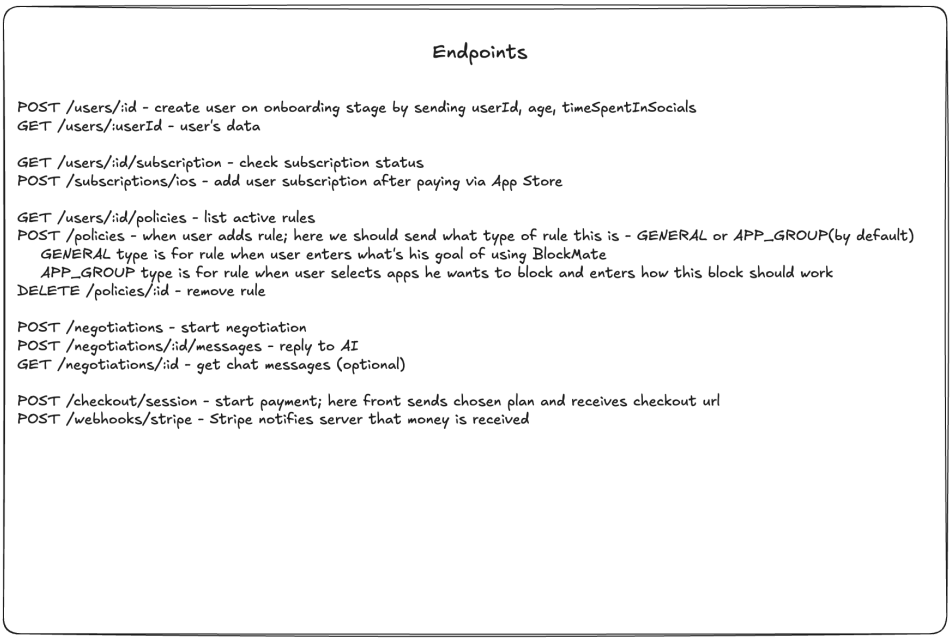

Figure 14: The REST API the monolith implied. After the pivot, almost every endpoint here became either the one callable function or a direct Firestore read of the user's own data.

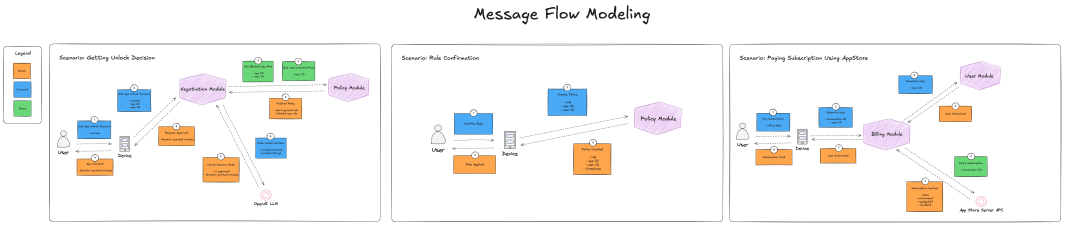

Figure 15: Early message-flow modelling for the negotiation, done before the two-model pipeline settled into its final shape.

The event-storming boards

The boards from the modelling workshop, in the order we built them: the legend, the first chaotic wall of events, the full ordered board that shows the whole flow end to end, the per-scenario flows that zoom into each part of it, and the refined resolution.

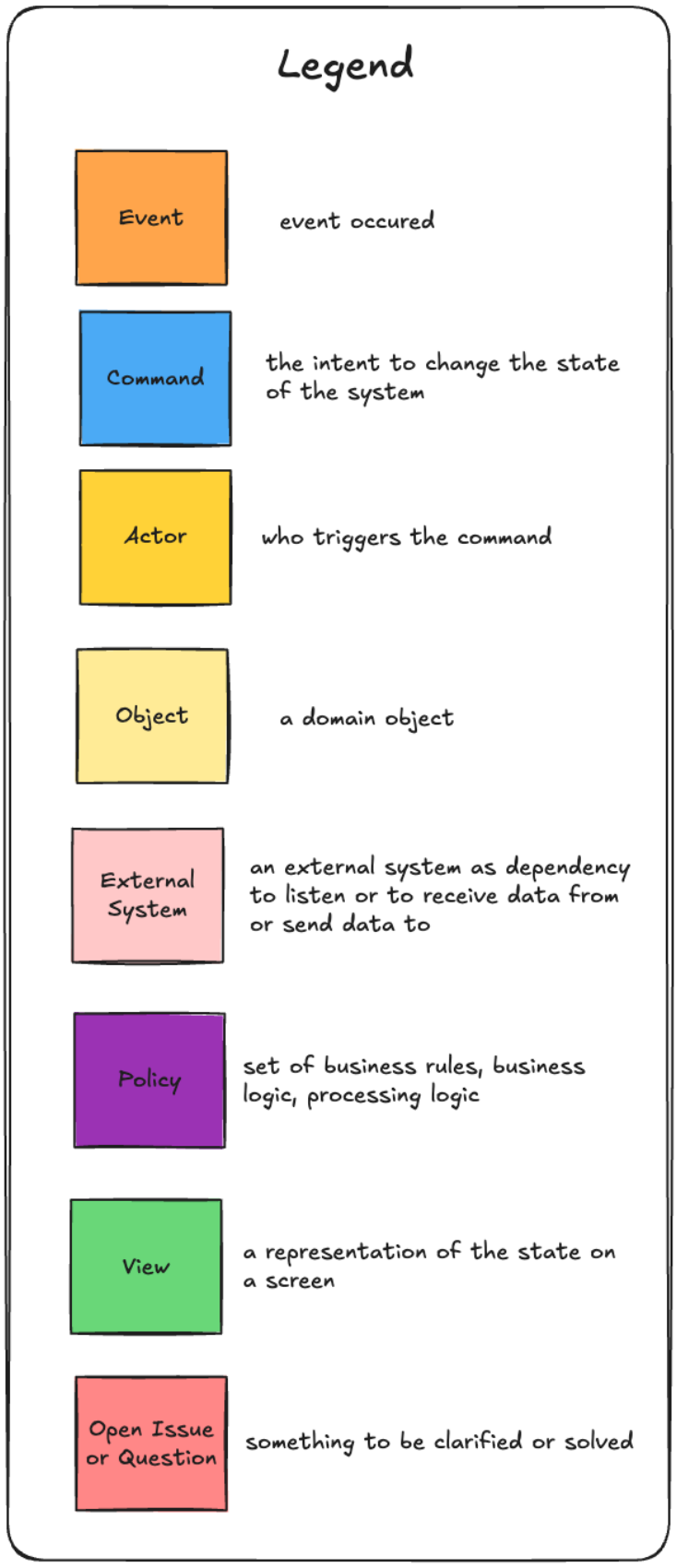

Figure 16: The event-storming notation we used: events in the past tense, with commands, actors, policies, and external systems layered around them.

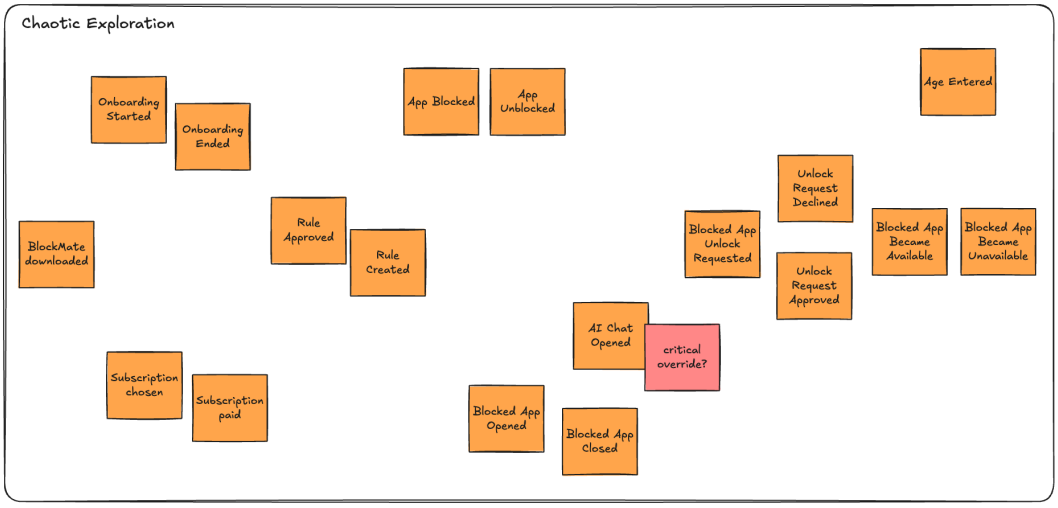

Figure 17: The first board: every event we could think of, unordered, before any structure was imposed.

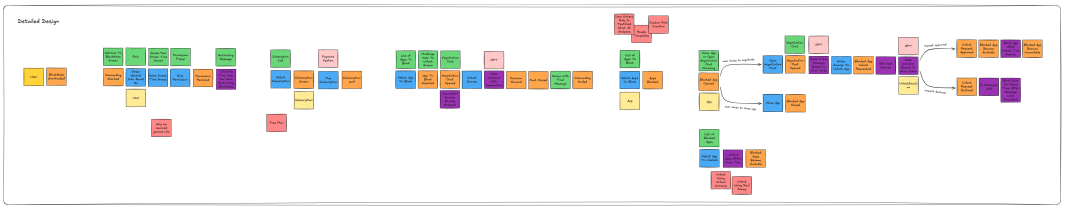

Figure 18: The full board once the events were ordered into one timeline: onboarding, rule creation, the negotiation and its outcomes, and subscription, laid out end to end. This is the general picture the per-scenario boards below then zoom into.

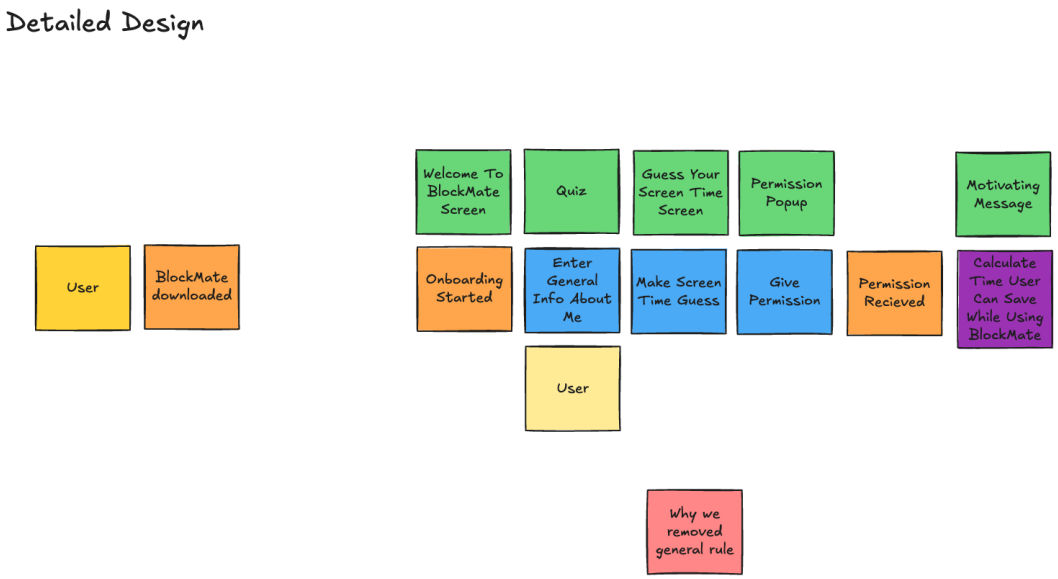

Figure 19: The onboarding flow, ordered.

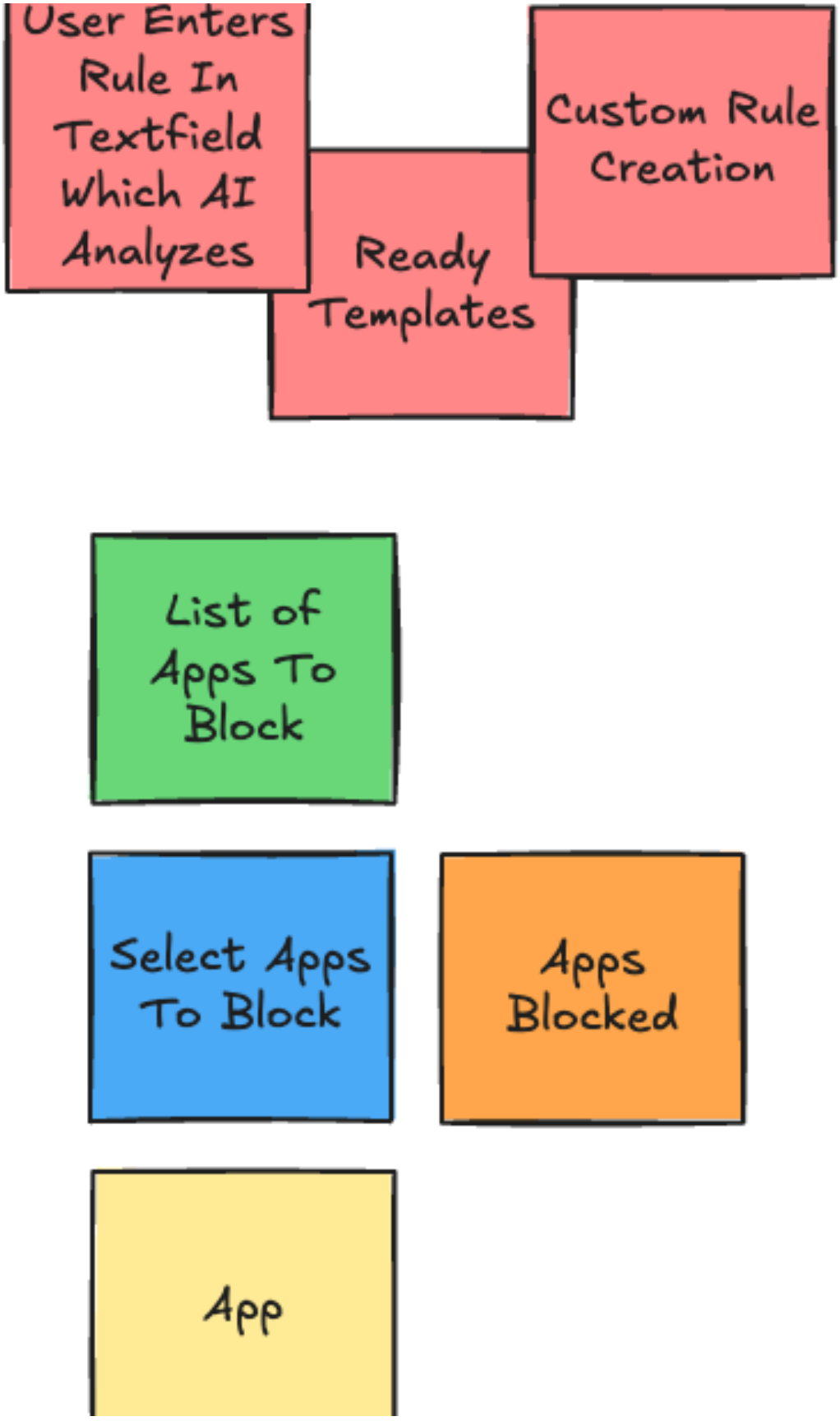

Figure 20: The rule-creation (app selection and blocklist) flow.

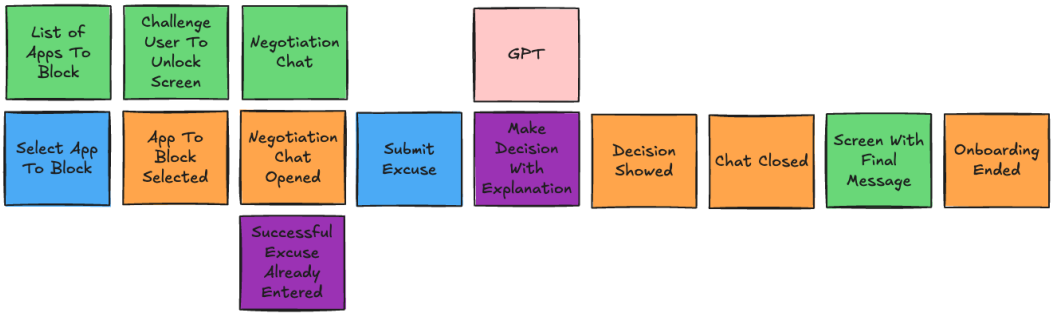

Figure 21: The negotiation flow on a first run.

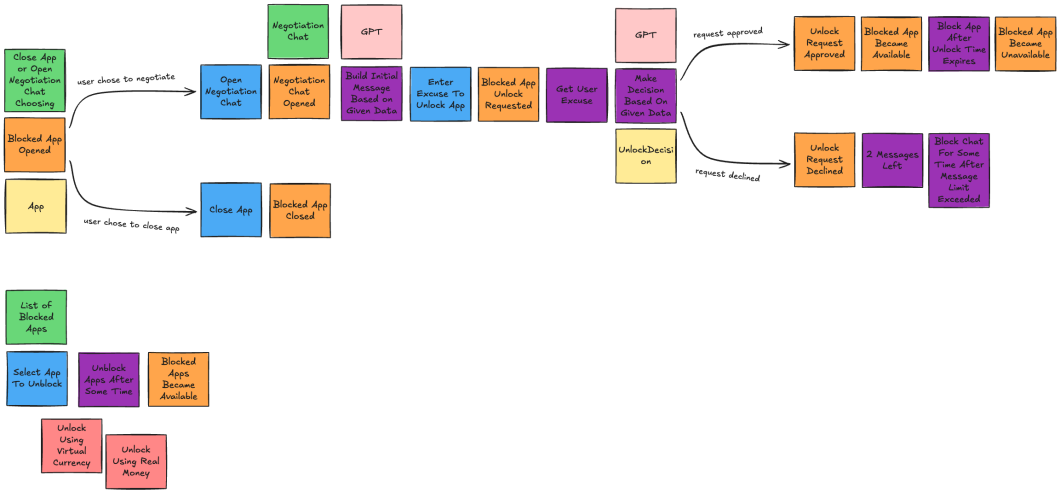

Figure 22: The negotiation outcomes (approve, reject, ask for more) resolved.

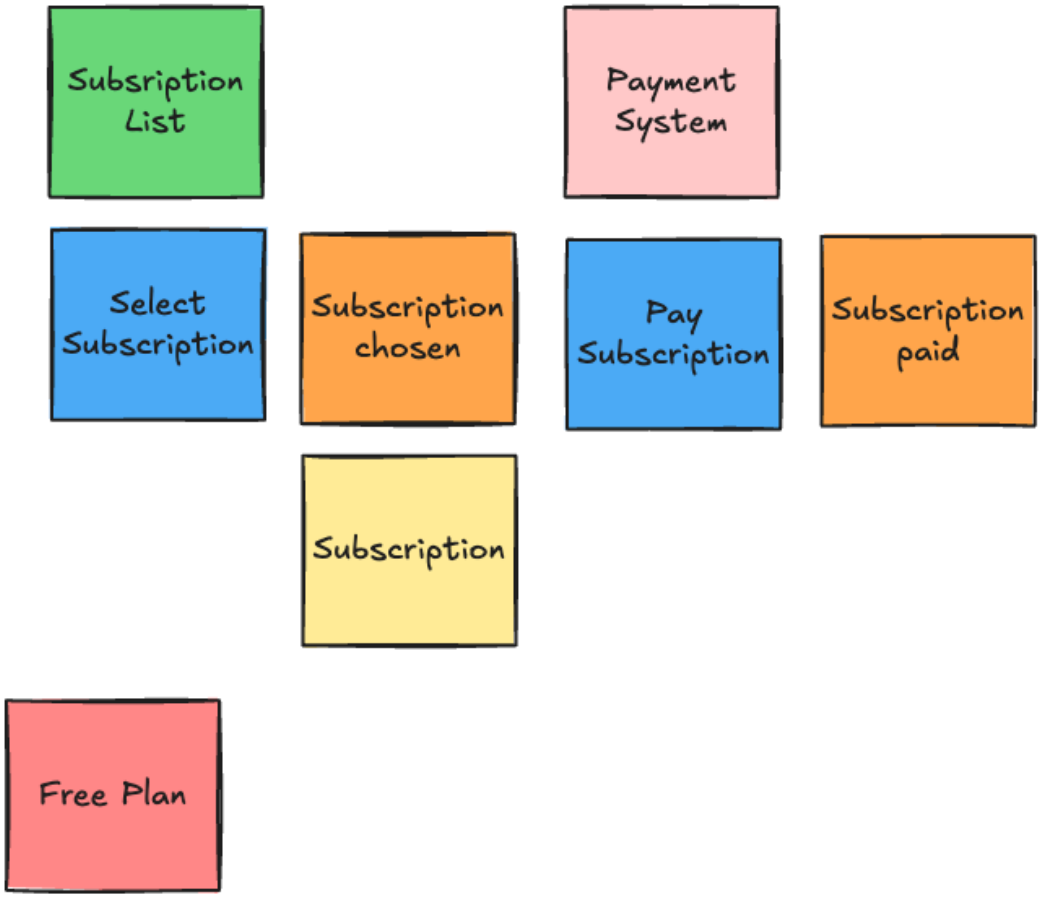

Figure 23: The subscription flow.

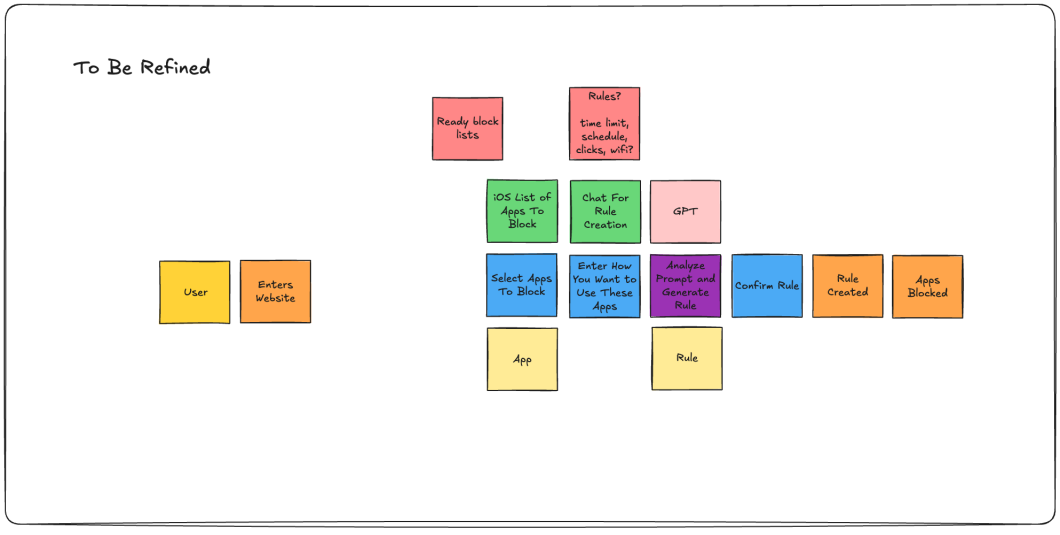

Figure 24: The refined board after resolving the sub-flows: the clean split into bounded contexts that the code went on to manipulate.

Build history: the first two iterations

BlockMate's code was rewritten from scratch three times in two months. Only the third ships; the first two are recorded here because each one set up the next.

Iteration	Period	Tech	Status
1: Chrome extension	MVP camp + hackathon	Plain JS, Manifest V3, OpenAI from the browser	Won the hackathon, archived
2: NestJS monolith	Between hackathon and MacBook	TypeScript, NestJS, Prisma, PostgreSQL	Built to a skeleton, abandoned
3: Firebase + Swift	MacBook to today	Firebase Functions, Firestore, Auth; Swift, SwiftUI	Production

Table 8: BlockMate's three iterations. Each was a full rewrite. Iteration 2 still sits in the repository at **`backend-nest/`** and no production code touches it.

Iteration 1 — the Chrome extension. The natural target for a screen-time product is the phone, and the only way to block apps on a phone is Apple's FamilyControls API, which means Swift, which means a Mac. I had a Windows laptop. So in a few days I built the thing I could build: a Chrome extension that redirects any blocked site to a takeover page and calls OpenAI straight from the browser. No backend, no database. It had three voice tones you could switch between — a bro, a strict trainer, and a meditation teacher — and those were the part the room remembered.

Two moments from the demo shaped everything after. The night before, a teammate typed "my house is on fire, please unlock TikTok for twenty hours," and the AI cheerfully granted twenty hours. The model believed any story you told it; the next morning's fix — teach it to spot an implausible excuse — is the ancestor of the prompt still running today. Then, on stage, I asked the jury to throw me excuses and ran them live. One judge said "I want to watch a video on how to do CPR." The AI gave fifteen minutes, and we won the More Tech Winner award from Genesis. The prompt was not actually good that night — out of the box it approved almost anything, and my first blunt attempt to make it stricter swung it the other way into rejecting real, productive requests. The proper fix came months later. Once the MacBook arrived, the extension was archived; it lives in the repository as the proof of concept that won, and as a possible desktop product one day.

Iteration 2 — the NestJS backend. Between the hackathon and the MacBook I was stuck: the product needed an iOS app I could not yet build, and the team needed something shipped every week. So I built a backend — NestJS, Prisma, PostgreSQL, a clean modular monolith with **`user`**, **`policy`**, **`negotiation`**, and **`billing`** modules over a Prisma schema that captured the whole data model. I took it to a clean module skeleton, then abandoned it once the serverless arithmetic became obvious: the operational surface dwarfed the forty lines of real logic underneath. Two outside conversations sealed it — a founder who ran his whole consumer app on a frontend plus Supabase, and Ihor Pysmennyi confirming

Firebase was a great starting point for our stage. The work was preserved on purpose: if and when BlockMate runs a serious marketing operation on the website (web checkout, funnels, attribution, operator dashboards), this relational backend is a real running start rather than a blank page. The scaling argument behind that plan is in the design chapter.

How the project came together

The MoreTech win came with a guaranteed place in the KSE Startup Incubator, and we applied in parallel to the UCU Startup Accelerator. Around the same time I finally got a MacBook, and with it the Apple APIs I had been missing. We also applied to Y Combinator. For the next several months I was simultaneously studying at KSE, working at Universe Group, attending the Incubator, attending the Accelerator, and building BlockMate. The two programs did very different things for the project: the Incubator pushed us on hypotheses and the business model, while the Accelerator forced a weekly cadence of customer-development interviews, and most of what is in the research chapter exists because of those interviews. None of this would have happened without Oleh and Dmytro carrying most of the accelerator paperwork while I focused on the engineering.

Backend implementation details

The entry point and the orchestrator

The HTTP entry point for **`negotiateAppAccess`** does nothing but parse, delegate, and map errors:

```
export const negotiateAppAccess = onCall(
  { secrets: [OPENAI_API_KEY, AMPLITUDE_API_KEY, LANGSMITH_API_KEY] },
  async (request) => {
    const req = parseRequest(request);
    try {
      return await runNegotiationTurn(req);
    } catch (error) {
      if (error instanceof HttpsError) throw error;
      console.error('Negotiation pipeline error:', error);
      throw new HttpsError('internal', 'AI brain failed.');
    }
  },
);
```

All the orchestration lives in **`runNegotiationTurn`**, in **`services/negotiationService.ts`**:

```
export const runNegotiationTurn = traceable(async (req) => {
  const provider = getLlmProvider();

  const [ctx, { history, docRef }] = await Promise.all([
    tracedAssembleContext(req.userId, req.localHour, req.dayOfWeek),
    tracedLoadHistory(req.userId, req.negotiationId),
  ]);

  if (req.excuse.trim().length === 0) {
```

```
    return handleEmptyExcuse(docRef, req, history);
  }

  let classification, strategies, response, degraded = false;
  try {
    classification = await provider.classifyIntent(req.excuse, ctx);
    const honored  = clampRequestedMinutes(classification.requestedMinutes);
    strategies     = selectStrategies(ctx, classification);
    const prompt   = composePrompt(ctx, strategies);
    response       = await provider.generateResponse(
      prompt, history, req.excuse, strategies,
      { honoredMinutes: honored, userLanguage: classification.userLanguage },
    );
  } catch (aiErr) {
    classification = FALLBACK_CLASSIFICATION_ON_DEGRADE;
    strategies     = [];
    response       = buildDegradedResponse();
    degraded       = true;
  }

  saveNegotiation(docRef, req, history, response, classification,
strategies).catch(console.error);
  if (!degraded) {
    updateJourney(db, req.userId, classification,
response.status).catch(console.error);
  }
  emitNegotiationResolved(req.userId, docRef.id, response, classification,
strategies, ...);
  flushBounded().catch(console.error);

  return { status: response.status, reply: response.reply, allowedTime:
response.allowedTime, negotiationId: docRef.id };
}, { name: 'runNegotiationTurn', run_type: 'chain' });
```

The context-load and history-load run in parallel through **`Promise.all`** because they read independent parts of Firestore. The empty-excuse branch fires before both model calls. The negotiation save, the journey update, and the analytics flush are all fire-and-forget; only the two model calls are awaited. The journey update is skipped on degraded turns so a fallback classification cannot poison long-term state.

Prompt-cache implementation

The discount applies only to a byte-identical prefix, so the order of concatenation in the prompt composer decides how much caches. The original order put the dynamic, user-specific block first, which meant no call could ever hit the cache. The fix inverts it:

1. Base identity prompt — *static*
2. Journey-tone fragment — *one of four*
3. Trust directive — *one of three*
4. Coaching strategy fragments, *sorted alphabetically — picked by the selector*
5. Output format — *static*
6. User-specific context block — *fully dynamic, last*

Two details matter: the strategy fragments are sorted before concatenation, so **`[a, b]`** and **`[b, a]`** produce identical bytes; and the output-format block sits before the dynamic context so the schema is part of the cacheable prefix.

Composing in cache-friendly order is necessary but not sufficient, because OpenAI's cache is sharded — without a hint, identical prefixes can land on different shards and miss each other. The **`prompt_cache_key`** parameter pins them together. The classifier uses one global key (its prompt is identical for every request); the responder shards by sorted strategy signature so requests that can share cache do:

```
const CLASSIFIER_CACHE_KEY = 'blockmate-classifier-v1';

export function responderCacheKey(strategies: StrategyKey[]): string {
  const sig = [...strategies].sort().join('-') || 'none';
  return `blockmate-responder-${sig}-v1`;
}
```

The **`v1`** suffix is a kill switch: every prompt-content change bumps it, abandoning the warm shards cleanly instead of thrashing against a half-matching prefix.

A unit check makes two identical calls back-to-back and confirms the cache engages:

```
[call_1] latency=480ms  prompt_tokens=3597  cached_tokens=0    (0%)
[call_2] latency=215ms  prompt_tokens=3597  cached_tokens=3456 (96%)
```

At workload scale (a ten-scenario cold smoke run, every call hitting OpenAI), the figures were:

Model	Prompt tokens	Cached tokens	Hit rate
`gpt-4.1-nano` (classifier)	32,339	23,552	73%
`gpt-4.1-mini` (responder)	76,359	14,336	19%
Total	108,698	37,888	34.9%

Actual cost $0.0318 against an uncached counterfactual of $0.0359 — an 11.3% saving, with eval accuracy unchanged. That 11.3% is a cold-shard floor: a smoke run is too short and too diverse to warm the cache, while production traffic keeps common combinations hot, so the steady-state saving is meaningfully larger.

Observability, CI, and the marketing site

LangSmith wiring

LangSmith hooks in at two points and nowhere else. The SDK client is conditionally wrapped, so tracing is opt-in and the system runs identically with no key set:

```
const raw = new OpenAI({ apiKey });
const langsmithKey = LANGSMITH_API_KEY.value() ||
process.env.LANGSMITH_API_KEY;
_openai = langsmithKey ? (wrapOpenAI(raw) as OpenAI) : raw;
```

The orchestrator is wrapped in **`traceable(...)`**, and the two Firestore reads are wrapped at **`tool`** granularity, producing a clean three-level tree per request:

```
runNegotiationTurn (chain)        ← parent, metadata: user_id,
├── assembleContext (tool)          journey_stage, trust_level,
├── loadHistory (tool)              latency_ms, strategies, status
├── ChatOpenAI (llm)  gpt-4.1-nano  (classifier)
└── ChatOpenAI (llm)  gpt-4.1-mini  (responder)
```

Two projects are wired up — **`blockmate-dev`** for local runs, **`blockmate-prod`** for production — kept strictly separate so a noisy local session never pollutes the production dashboards.

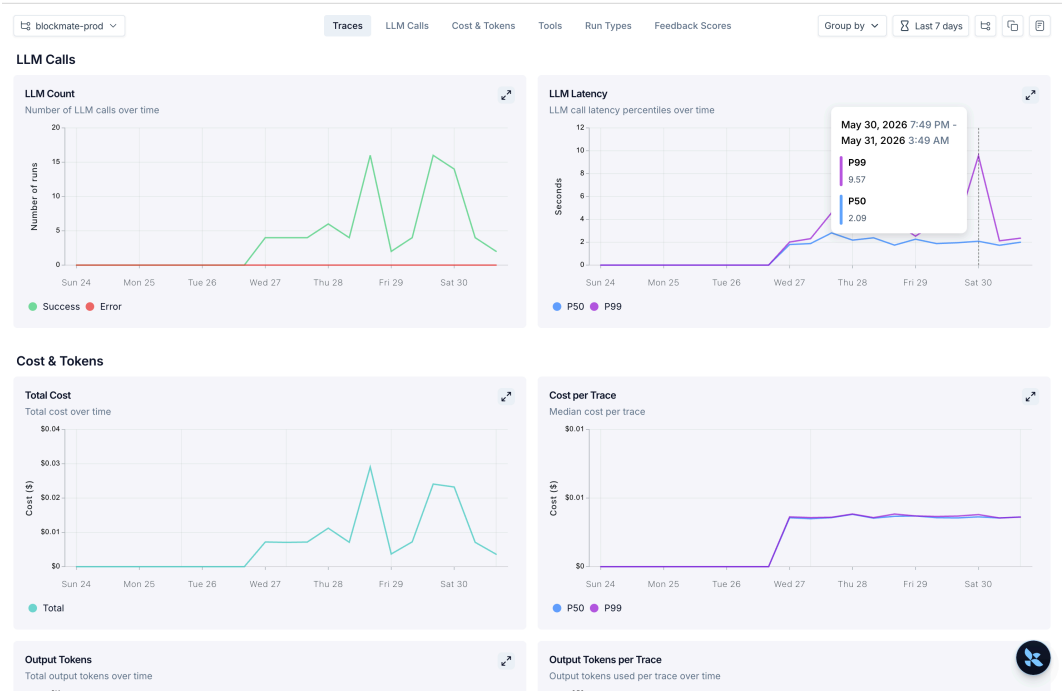

Figure 25: The production LangSmith dashboards over a rolling 7-day window: call volume, P50 vs P99 latency, cost per day, and median cost per trace.

DevOps and CI

Three things ship: the Cloud Function (via **`firebase deploy --only functions`** from a GitHub Action on every push to **`main`** that touches the function source, about ninety seconds end to end); the iOS app (built and uploaded to TestFlight by hand from Xcode); and the marketing site (a static Next.js export on Firebase Hosting). Two GitHub Actions exist: **`deploy-functions.yml`**, which deploys on push to **`main`**, and **`ai-eval.yml`**, a manual **`workflow_dispatch`** that runs the eval against a chosen tier and posts the diff — manual because running it on every commit would burn OpenAI tokens. Secrets live in Firebase Secrets (the OpenAI key, bound to the function at deploy time), GitHub Secrets (the deploy service account), and Apple Developer (the entitlement and App Store credentials); none are in the repository.

The marketing site

The site at **`yourblockmate.com`** is a small Next.js static export carrying the product narrative, a waitlist form that writes to Firestore, and the TestFlight handoff. It has a deliberate SEO setup — a content-rich metadata block, a vetted keyword list (**`how to reduce screen time`**, **`Opal alternative`**, and similar), Open-Graph and Twitter cards, and a canonical URL — and is submitted to Google Search Console. Web analytics run through Firebase Analytics (GA4), gated on **`isSupported()`** so the site degrades cleanly. It is scaffolding for the paid acquisition described in the conclusion.

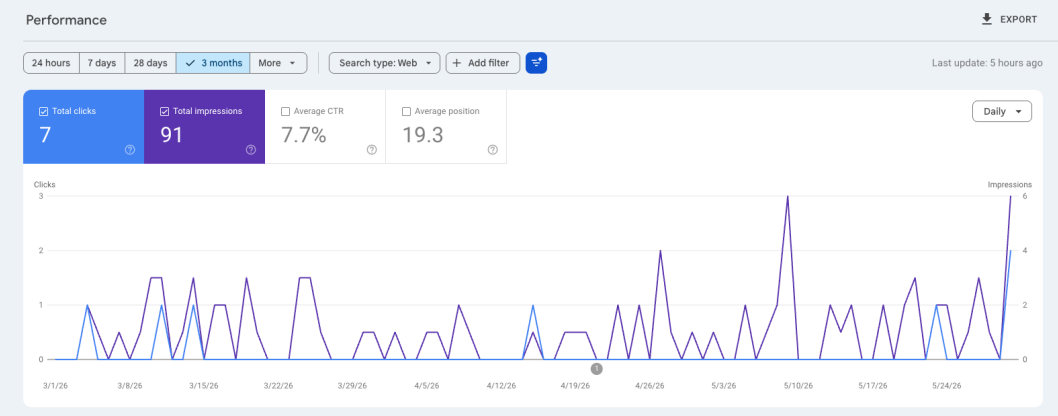

Figure 26: Google Search Console for **`yourblockmate.com`** over three months: small absolute numbers on a pre-launch site, but a click-through rate well above the baseline for its positions.