Security at yeswekanban

How a platform that puts minors, real money, and AI in one place keeps all three safe — from the big picture down to the mechanisms.

This is a technical account of how yeswekanban is secured. It starts with the big picture — why this is a hard problem and the principles we hold to — and then goes down, layer by layer, into the actual mechanisms: how identity works, how one family's data is kept away from another's, why the app server is not the security boundary, how money moves safely, how the AI is fenced in, and how we know what happened after the fact. It ends with how this all gets built and kept honest.

Read the first two sections for the shape of it. Read the rest if you want the mechanisms.

Part I — The big picture

1. Why this is a hard problem

Most products have to secure one difficult thing. yeswekanban has three, at once, and they interact:

- **Minors.** The people doing the work are kids, 13 and up. That raises the stakes on privacy, on what content can reach them, and on what data ever leaves the building — and it invites a class of regulation (COPPA-adjacent) that most B2B software never touches.
- **Real money.** Not points, not credits — actual dollars that a parent pays and a kid withdraws to a real bank account. The moment real money moves, you inherit fraud, chargebacks, disputes, money-laundering surface, and an adversary with a direct financial incentive.
- **An AI in the middle.** A language model interviews the user, writes plans, and answers questions. That model is a powerful component and an *attackable* one: it can be manipulated by cleverly-worded input, and it must never be handed the keys to the money or the account.

Any one of these is a serious security program. The combination is why security here isn't a feature bolted on at the end — it's the substrate the whole product sits on. The rest of this document is what that substrate is made of.

2. The principles

Six ideas govern almost every decision below. They're worth stating plainly, because once you know them, a lot of the specific mechanisms become predictable.

Defense in depth. No single control is trusted to be the whole answer. The same rule tends to be enforced in three or four independent places — in the browser, in the app server, in the database's own permission system, and sometimes in a hard database constraint on top of that. For a bypass to work, every layer has to fail simultaneously.

Assume the client is hostile. Anything running in a browser or holding an API token is treated as potentially controlled by an attacker. Validation in the user interface is a convenience, never a security boundary. The real checks live on the server and in the database, where the user can't reach them. This principle, taken seriously, is responsible for the single most important part of the system (Section 6).

Least privilege. Every component gets the narrowest access it can do its job with. The AI has no tool that can move money. The browser can write four columns of a task and not the fifth. A background job authenticates with a single-purpose secret. The powerful, bypass-everything database key is used only on the server, for the handful of operations that genuinely need it.

Fail closed on security; fail open on availability — deliberately, and documented. When a *security* check can't complete — say, the system can't confirm whether you've passed two-factor auth — it assumes the worst and blocks you. When a non-security *availability* dependency hiccups — briefly slow or unreachable — the system errs toward letting real users keep working, so that a transient outage can't turn into a self-inflicted denial of service. Which way each control fails is a decision made on purpose, written down in the code, and it is the thing security reviews argue about most.

Everything important is logged, and the log is kept somewhere the attacker can't reach. A break-in isn't fully contained if the intruder can then erase the record of it. So the audit trail is mirrored, in real time, to a second system outside our own database — and the write to that outside system happens *first*.

Adversarial review before shipping. Anything touching authentication, permissions, payments, the AI, or kids' data goes through a structured three-reviewer process — two independent reviewers trying to break it from different angles, then a third arbitrating — before it merges. The findings compound into a running checklist that every future review starts from.

Part II — The mechanisms

From here down it gets concrete. Each section says what it defends against and how it actually works.

3. Identity and sign-in

Authentication is built on Supabase Auth. Under the hood there are three separate database clients, and the separation matters: a **user-scoped** client that carries the signed-in person's identity and is bound by all the permission rules below; a **public** browser client that holds only a publishable key; and a **service-role** client that bypasses those permission rules entirely. That last one is the master key. It exists only on the server, and it is used only for the specific privileged writes that must not be doable from a browser. Keeping that key server-side and narrowly-used is the hinge the whole least-privilege story turns on.

Sign-in supports password, Google, and Microsoft. A subtle but important detail governs how two-factor fits in: the platform's native notion of "assurance level" (has this session completed a second factor?) only understands one of our two second-factor types. So we never use it as a standalone gate — it's treated purely as a *signal* that says "a second factor exists, go prompt for it," and the real decision is made by our own unified check (Section 4). Getting this wrong is a classic way to accidentally lock out or wave through the wrong users; it's called out explicitly wherever it appears.

4. Multi-factor authentication, in depth

There are two independent second-factor systems, and a user can have either, both, or neither.

App-based one-time codes (TOTP). The standard authenticator-app flow. Recovery codes are generated ten at a time, shown exactly once, and stored only as SHA-256 hashes — we cannot recover them, only verify them. Rotating them is atomic (a single database function replaces the whole set, so a partial failure can't leave an account with zero working codes), and redeeming one is a single-round-trip operation that also invalidates it. Using a recovery code deliberately wipes the existing authenticator and forces re-enrollment. Losing both the authenticator and every recovery code is, by design, unrecoverable in-product — reset is a deliberate out-of-band operation, because a self-service path around your own second factor is a second factor in name only.

Email one-time codes. A home-grown second factor for people who don't use an authenticator app. When you pass it, the server issues a signed "pass" cookie of the form `userId.expiry.signature`, where the signature is an HMAC (a keyed cryptographic fingerprint) over the first two parts. On every subsequent request the server recomputes that fingerprint and compares it in **constant time** — a comparison that takes the same duration whether the guess is almost-right or completely-wrong, so an attacker can't feel their way to a valid signature by measuring response times. The cookie is HTTP-only, secure, and expires in seven days. The signing key

comes from a dedicated secret, and if that secret is somehow absent the code refuses to run rather than falling back to something weaker. A separate, opt-in "trust this device" token (opaque, hashed at rest, 30 days) lets a known device skip the prompt.

The enrollment hardening. Turning on email-2FA is a two-step request-then-confirm flow: the server mails a code, and only when you enter it does the preference actually flip on — and it flips two fields together, an `enabled` flag *and* an `enrolled_at` timestamp. The load-bearing rule is that the preference only counts as a real factor when *both* are set. This closes a genuine attack: without it, someone who had stolen a first-factor-only session could flip the "2FA enabled" bit on the victim and thereby *silence* enforcement, because the system would think a factor existed when none had ever been proven. A dedicated `purpose` field on each emailed code (with a database constraint limiting it to `login` or `enable_2fa`) prevents a code minted for one flow from being replayed into the other.

One predicate, used everywhere. "Has this session proven a second factor?" is answered by a single pure function. Its logic: no factor configured at all → vacuously satisfied (so a brand-new user isn't locked out of setting one up); a verified authenticator plus a completed challenge → satisfied; email-2FA enabled plus a valid pass cookie → satisfied; otherwise not. The same function is called by the request-enforcement layer and by the server-side action guards, so the two enforcement points cannot drift apart and disagree.

Mandatory for account owners, with a grace clock. The adult who owns an account is required to have a second factor. On first sight of an owner without one, the system stamps a 14-day deadline (the stamp is written with a "only if not already set" condition, so concurrent requests can't race and reset the clock). After the deadline, an owner with no factor is redirected to a set-up page — and here is the careful part. The redirect could have simply carved out "the whole settings area" as reachable. It doesn't: the *only* thing a past-deadline owner can reach is the security tab specifically. That narrowness is deliberate, because carving out all of settings would have left the account-deletion, subscription-cancel, kid-archive, and token-revoke actions reachable to someone with a stolen password — exactly the destructive actions a second factor is meant to protect.

Closing the same hole on the back end. The redirect above protects pages. But form submissions and API calls don't go through the page router, so they get their own equivalent gate — a two-layer check that, in addition to the standard "is MFA satisfied" test, has a second layer specifically catching the zero-factor owner past their deadline, so the "vacuously satisfied" branch above can't become a bypass for the very users it's meant to force.

Platform administrators face a stricter version still: a privileged action requires a fresh check of admin status, a satisfied second factor, a grace clock that's set and elapsed, *and* confirmation that a verified factor exists right now — four independent conditions, all failing closed, with a deliberately generic "Forbidden" response so the endpoint can't be used to enumerate who is and isn't an admin.

5. Keeping one family's data away from another's

The database enforces tenant isolation itself, underneath the application, using Postgres **Row-Level Security** (RLS) — rules attached directly to each table that decide which rows a given user can even see or touch. The core rule is uniform: a row is visible only when its account matches the account of the person asking, resolved through a small privileged helper function that looks up "which account does this logged-in user belong to." Because that check is enforced at the database, it holds no matter how the data is reached — through the app, through a raw API call, through anything.

Over forty tables carry these rules. Some are locked down harder than the default: the AI's memory store, for instance, has RLS enabled with *zero* policies and all direct access revoked, which means no ordinary user can read or write it at all — every access goes through server-side code holding the master key. The two-factor preferences table is similarly readable-only-about-yourself and never directly writable.

Isolation is the kind of property that's easy to believe you have and hard to be sure of, so it's tested. A regression suite spins up a real Postgres, seeds two separate accounts, then switches identity and asserts, table by table, that each account sees its own rows and exactly zero of the other's. The clever part: it also runs a bare, unfiltered count and checks it equals the owned-rows count — which is what catches the catastrophic case where RLS has been accidentally switched off entirely (a normal filtered query would still look correct in that case; the unfiltered count would not). This suite runs in continuous integration on every change.

6. Why the app server is not the security boundary

This is the most important section, and the one most systems get wrong.

Here is the trap. In a Supabase-style setup, the browser can talk to the database's REST API directly, and by default the database grants the logged-in role permission to update whole tables. Row-Level Security controls *which rows* you can touch — but not *which columns*. So if a user is allowed to update their own row (which they must be, to edit their own profile), the default configuration also lets them update *any column* of that row, directly, from the browser's developer console — bypassing every check the application code so carefully performs. In that world, the validation in your server action is theater: the attacker simply doesn't call it.

The consequences are not abstract. Without the fix, a kid could set their own account's role to owner (account takeover), flip the platform-admin bit on themselves, redirect their payout bank details, or raise the pay on their own task from five dollars to nine thousand — all by editing a column the UI never exposed, using nothing but browser tools.

The fix rests on a subtle Postgres fact that we learned the expensive way, by running the exploit against a candidate fix and watching it still succeed: a column-level "revoke permission on these specific columns" is a **no-op** when a whole-table grant already covers everything. The only thing

that actually works is to revoke update on the *whole table* and then grant back only the specific safe columns. So that is the pattern, applied table by table:

- **Profiles** — the browser can write exactly one cosmetic column. Everything sensitive — the role, the admin flag, all the payout and bank fields, the MFA grace clock — is unwritable from the client. Changing any of them requires server-side code holding the master key, which only runs after the appropriate checks.
- **Tasks** — the browser can write title, description, status, and completion. It cannot write `pay_amount`.
- **Plans, weekly boards, the AI's session records, and several others** — no client write at all; every change flows through the server. This is what makes it impossible to self-approve a plan, forge a reference letter, or tamper with someone else's records directly.

Where revoking a whole table would be too blunt, the same goal is achieved with a database **trigger** — a small piece of logic that runs before any update and rejects the write (with a permission error) if a non-privileged caller is trying to change a protected column. The privacy opt-in flag and the portfolio fields are guarded this way, several of them with an *additional* hard constraint on top (length limits, and a rule that a stored URL must begin with `https://`, which blocks a whole class of injected-link attacks even against a bug in our own privileged code). Belt, and suspenders, and a second belt.

The through-line: the server action is a convenience and a place to be helpful, but it is *not* what stands between an attacker and the data. The database's own permission system is.

7. The enforcement layer

Sitting in front of every dashboard request is a middleware layer that runs a fixed sequence of gates: are you signed in; have you satisfied two-factor; if you're an owner past your grace deadline, are you being redirected to set one up; is your consent current; and is your subscription in a state that permits the paid action you're attempting. Its failure posture is exactly the principle from Part I, made concrete and commented: the security-critical reads (the assurance-level check, the two-factor-preference read) **fail closed** — if they can't complete, you're bounced to re-authenticate, on the explicit reasoning that an attacker shouldn't be able to time a request during an outage to slip past a security check.

Crucially, this layer is *not* trusted as the only boundary. Form submissions and API endpoints don't pass through it, so they carry their own copies of the same checks. The middleware is one layer of the defense, not the wall.

8. Money

Payments are two separate systems: the parent's subscription, and the kid's earnings. Both are built on Stripe, and the security work is in the seams.

The webhook. Stripe tells us about payment events by calling a webhook. Every such call is verified by cryptographic signature against the exact raw bytes of the request, with a five-minute timestamp tolerance to defeat replay of an old-but-valid message; an unverifiable call is rejected outright. Events are de-duplicated by inserting the event's ID into a table with a uniqueness constraint *before* processing — a repeat delivery hits the constraint and is acknowledged without being processed twice. (An allow-list of Stripe's own published IP ranges backs this up as a secondary signal, but the cryptographic signature is the real gate — it's what's load-bearing.)

The earnings ledger. Every cent of a kid's balance lives in an append-only ledger. The kinds of entries are constrained both in the application's types and by a hard database constraint. Payouts are idempotent — a unique index on the transfer ID means the same payout physically cannot be recorded twice — and debits carry an **overdraw guard** that reads the current balance and refuses, loudly, to send more than exists. The ledger table has no write permission for anyone but the server's master key. Balance is presented in three honest parts: pending (finished work still inside its holding window), available (clear to withdraw), and lifetime (total ever earned).

The holding window. When a parent pays, the money doesn't transfer instantly — it's held to cover the refund-and-dispute window. The length is *risk-adjusted*, not fixed: a well-established, trusted account clears in about a day; the baseline is two days; an account showing fraud signals can be held for two weeks. The system takes the *longest* hold any signal demands. (You'll sometimes see "48-hour hold" in the interface copy — treat that as the friendly default, not the enforced number, which the code computes from live risk.)

The payout job. A scheduled background job sweeps matured holds and issues the transfers. It authenticates with a dedicated secret compared in constant time — a mismatch gets a flat rejection, a missing secret disables the job rather than running it open. It writes the ledger debit *before* creating the transfer, and if that debit doesn't land cleanly it distinguishes a harmless retry from a real inconsistency, and in the inconsistent case it stops, alerts a human, and leaves the money untouched rather than risk a double-send. Every one of the six scheduled jobs authenticates the same constant-time way.

Fraud and disputes. Early fraud warnings trigger an automatic refund — and the idempotency key on that refund is deliberately tied to the *charge*, not the warning, because a single charge can generate several warnings and keying on the warning would issue several refunds. The whole fraud handler is written never to throw an error back to Stripe, because an error would prompt a retry, and a retried refund is a double refund. When work has already been paid out and a fraud signal arrives

too late to claw it back automatically, the system doesn't quietly fail — it flags the case and pages a human. Disputes auto-submit evidence; reversals flip the affected invoice and re-credit the kid through the ledger.

The subscription gate. Access to the expensive AI features is gated on subscription status — and that gate is re-checked inside each AI endpoint, not just in the page middleware, so a raw API call that skips the front door still gets a `402 Payment Required`. On the money side the gate fails closed: canceled, past-due, and incomplete subscriptions are all refused, on the reasoning that the AI calls cost us real money we'd rather not spend on an account that isn't paying.

9. The AI, on a leash

The language model is treated as a powerful but untrusted component. Two problems have to be solved: it must not be manipulable into doing something harmful, and it must not have the *ability* to do the most harmful things even if it were manipulated.

Prompt-injection defense. Any text a user controls — a goal, a name, a task note, a bio, even a third party's code in a pulled-in diff — is wrapped in a labeled fence before it's shown to the model, and the wrapping escapes the fence's own delimiter characters so a user cannot type the closing tag themselves and "break out" to issue instructions. The system prompt then states, explicitly, that anything inside the fence is data to be used, never instructions to be followed, and enumerates the exact manipulation attempts to ignore. This wrapping is applied everywhere user text enters a prompt — the planner, the tutor, the portfolio writer, the reference-letter writer, the code reviewer.

The AI has no dangerous tools — structurally. This is the leash. The plan-builder AI has exactly five tools, and none of them can touch money; the tool that creates tasks has no field for pay at all, so the AI *cannot* set what a task pays, no matter what it's asked. The help tutor has *no* action tools whatsoever — it is read-only as a matter of code, not policy: it literally has no ability to edit a board, change a status, or approve a plan, so no amount of clever prompting gives it one. These are structural guarantees, not promises in a system prompt.

Memory, scoped and sealed. The AI's per-person memory is scoped to a single profile in *two independent places* — the calling code passes the profile, and the database search function itself filters on it — so a bug in either one alone still can't leak one kid's memories to another. Who "said" each memory (the kid, the AI, the system) is stamped by the writing code, not accepted from the caller, so authorship can't be forged. Memory is opt-out, and turning it off skips both writing *and* reading, so the slate is genuinely clean. And because a stored memory is itself a place an injection could hide, retrieved memories are wrapped in the same untrusted-data fence with an added "reference only" instruction. There's a deliberate asymmetry worth naming: if the system can't confirm your *consent* status it refuses to write memory (fail closed — a legal line), but if it merely can't read your *opt-in preference* it allows the write (fail open — a UX hiccup shouldn't erase your memory). That trade-off is marked in the code with a "do not invert without review" warning.

Spend controls. Every AI call passes a global kill switch, the subscription gate above, a per-user hourly rate limit, and monthly budget caps at both the account and the individual-kid level. Usage is metered to a ledger that the caps read from.

10. Untrusted input and content

Beyond the AI, everything a user can submit is treated as potentially hostile.

Every uploaded image is screened — before it can appear anywhere public — by a vision model that classifies it for abuse categories including CSAM, violence, and adult content. Anything in the worst category is blocked at *any* confidence; other categories block above a confidence threshold, with borderline cases routed to human review. The classifier's own prompt is hardened against images that try to smuggle instructions in as text, and every decision is recorded to a moderation log.

Uploaded files are identified by their actual bytes, not by the type the browser claims — the server reads the file's magic-number signature and stores *that*, and it rejects files whose leading bytes look like HTML, SVG, or script, which is how a "harmless image" could otherwise become a page that runs code when served.

Text is normalized to strip invisible and direction-reversing Unicode characters — the kind used to make text display as something other than what it is, or to smuggle hidden content — from the fields where it's stored.

Links are scheme-checked. Anywhere a user-supplied URL is rendered as a clickable link on a page, it's forced to be `https://`, with `javascript:` and `data:` links and raw IP addresses rejected; links labeled as pointing to a specific service (a "View on GitHub" button) are checked to actually point at that service's hostname, not merely to start with `https://`.

11. Privacy

Consent is specific and versioned. The consent a parent gives names *every single* outside company that ever touches any data, and states exactly what each one receives — the database host, the AI inference provider, the embeddings provider, the email sender, the analytics tool, the payment processor, the off-site audit store (and precisely what it does and doesn't get). The consent text carries a version; when it changes, owners are required to re-consent before continuing. The audit store's disclosure, for instance, is careful: it receives account IDs, IP, user-agent, and event types — and explicitly not chat content, plan text, kid names, or financial details.

Deletion is real and immediate. When an owner deletes their account (an action itself gated behind two-factor and a typed-confirmation), the teardown runs in a deliberate order — cancel the subscription, close the kids' payout accounts, wipe uploaded files from storage, write a final audit record, then cascade-delete the account and finally the underlying login users. That order is itself a

fix for a real bug we hit. Notably, the final audit record *survives* the deletion: the audit tables are wired to null-out the account reference rather than vanish with it, so "an account that deleted itself thirty minutes after a dispute" remains visible for fraud defense.

Data export is owner-only, rate-limited, and built on explicit column allow-lists — never a "select everything" — so a future sensitive column can't silently leak into an export, and it strips password hashes, tokens, MFA secrets, and other people's identifiers.

Kids are excluded from analytics, structurally. Session recording doesn't start until a user's role is known, and for a kid the analytics SDK is opted out entirely; sign-out stops any recording before the next person on a shared device begins. Server-side business events (a subscription paid, say) are always attributed to the adult owner, never a kid, and identifiers like invoice numbers are cryptographically hashed before they ever reach the analytics tool.

Logs are redacted. In production, a redaction pass runs over everything logged, clobbering values by key name (passwords, tokens, secrets, one-time codes) and by pattern (anything shaped like a JWT, an API key, an email, a phone number, an IP). It walks nested objects and error-cause chains and never itself throws.

12. The audit trail you can't erase

Consequential actions are written to an activity log with a closed set of roughly 120 event types. The log records who, what, when, from which IP and browser — with the IP taken from the trusted infrastructure header, not the client-forgeable one.

The important property is that this trail is mirrored, in real time, to an *outside* system (Axiom) — and the mirror write happens *before* the local database write. The reasoning is precise: an attacker who compromises our database could otherwise delete the evidence of what they did. By shipping the record off-platform first, an intruder who can write our database but not the outside store still leaves a trail they can't reach. On that mirror, IP and user-agent are deliberately *not* redacted, because in a forensic record those are the load-bearing fields — a rare case where *keeping* data is the privacy-conscious choice, and it's disclosed as such. As a safety interlock, the off-site shipper only runs when the redaction pass is also active, so a developer can't accidentally ship raw local data to the production store.

13. The edge

At the network edge, before a request even reaches the application, a web-application-firewall rule-set blocks the obvious probes — requests for `.env` files, git directories, WordPress paths, backup archives (including double-extension tricks), and requests carrying credentials in the wrong place —

and applies coarse geographic and bad-bot filters. The rules are written carefully (case-insensitive so a capital letter can't bypass them, with the payment webhooks explicitly excluded so Stripe is never blocked).

Application-level rate limits sit behind that, keyed per action and per IP — on sign-up, on the AI, on uploads, on exports — so a single source can't hammer any one endpoint.

The edge is understood as one layer, not the perimeter: it cuts down background noise and turns away the obvious probes, while the substantive defenses are the ones described above — the authentication and two-factor gates, the application-level rate limits, and the per-table permission model that keeps the real boundary in the database rather than at the network edge.

Part III — How this is built

14. How security actually gets built here

Mechanisms are only as good as the process that produces them. Three things keep the standard from slipping.

Adversarial review. Any change touching authentication, permissions, payments, webhooks, the AI gate, uploads, or compliance copy goes through a three-layer review: two reviewers working in parallel and blind to each other — one hunting correctness bugs, one thinking purely like an attacker — and a third who reads both reports, arbitrates, applies fixes, and then re-reviews the fixes before anything merges. You can see the fingerprints of this all through the code, in comments that cite the specific review finding a given line exists to close.

A compounding checklist. Every finding worth remembering goes into an append-only learnings file — around forty patterns and growing — that is fed into every future security review. The system quite literally gets harder to fool over time, because each mistake becomes a permanent thing the next review looks for. (Several of the sharper mechanisms in this document — the whole-table-revoke lesson, the consent/shipper interlock, the charge-keyed refund — are entries in that file.)

Automated gates. Every change runs through a static-analysis security scanner (including an OWASP-Top-Ten ruleset and a committed-secret detector), a dependency-vulnerability audit, type-checking, and — importantly — the tenant-isolation regression suite from Section 5, which runs against a real database on every change.

15. In summary

The shape of it: **defense in depth**, so no one control is load-bearing alone. **The database, not the app, as the real boundary**, so a hostile client with developer tools still can't touch what it shouldn't. **The AI fenced off from money and account control** by structure, not by instruction. **Money moved through idempotent, overdraw-guarded, human-backstopped rails. An audit trail mirrored off-site before it's written locally**, so the record survives the breach. And a **build process that reviews adversarially and remembers every mistake.**

No security program is ever finished, and this one is treated that way — reviewed adversarially, hardened continuously, with every mistake turned into a permanent check against the next one. But it is a real one, designed from the ground up for the genuinely hard case of putting minors, real money, and AI in the same place.
