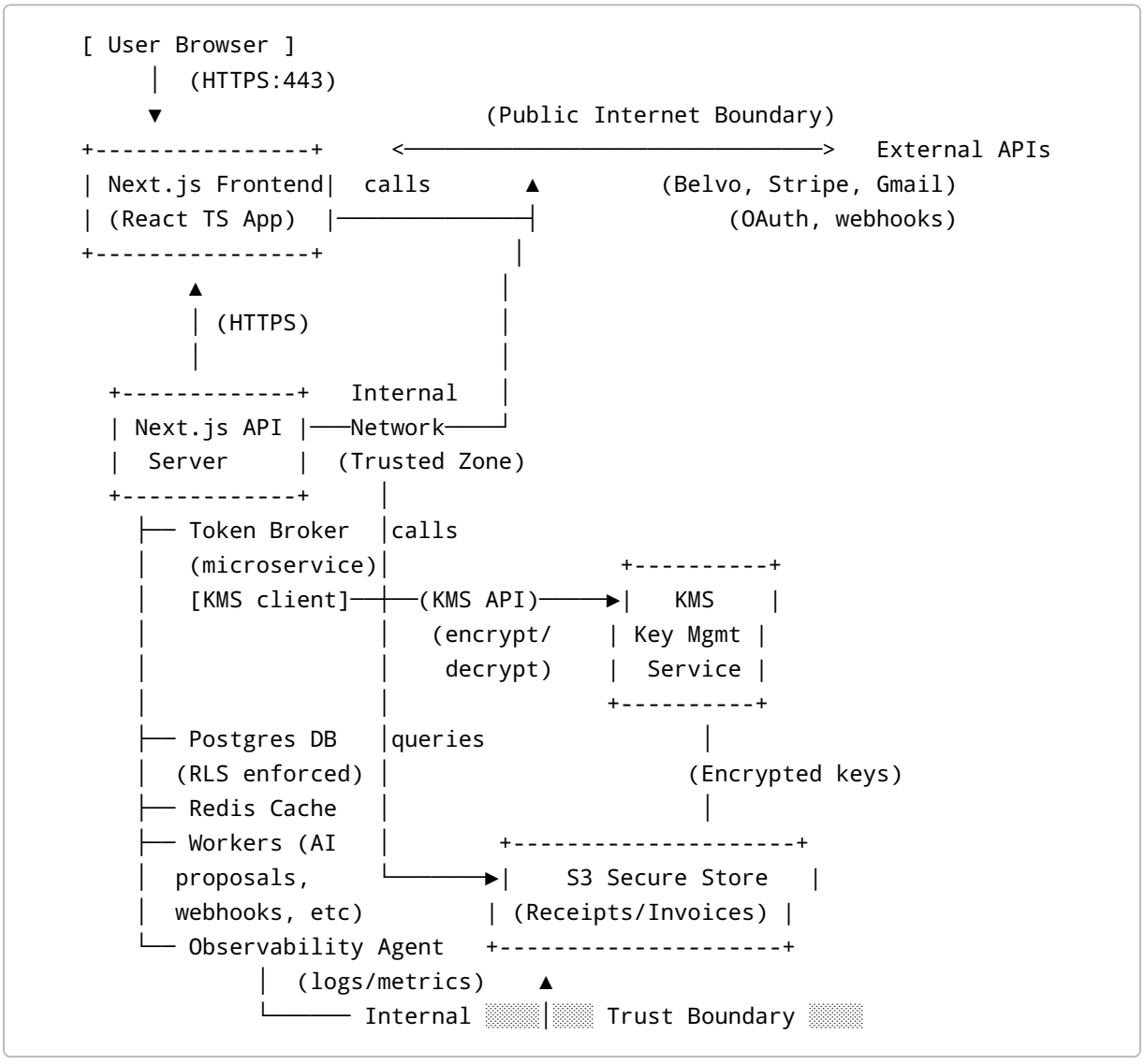
ChatGPT

Fintech Dashboard Security Architecture & Implementation Guide

Security Architecture & Data Flow Diagram

Architecture Overview: The Fintech dashboard is built on a Next.js framework (React/TypeScript) with a Node.js backend (Next API routes). The system integrates with external Open Finance APIs (e.g. Belvo/Plugi), payment gateways (Stripe/Mercado Pago), and Gmail for receipt imports. All components are containerized in a cloud environment and communicate over secure channels. The following ASCII diagram illustrates the architecture, highlighting trust boundaries and data flows:



Trust Boundaries: The primary trust boundary lies between the public internet and the internal network. All traffic from user browsers to the Next.js frontend and API is over HTTPS, terminating at a reverse proxy or the Next.js server [1] [2]. External integrations (Belvo, Stripe, Gmail) are invoked through secure APIs (HTTPS with OAuth), crossing an external boundary. Within the internal network, the Next.js API server, background workers, and supporting services (DB, cache, KMS, storage) communicate over a private VLAN or localhost interfaces. A further micro-segmentation boundary exists around the **Token Broker** component – only this service has permissions to decrypt sensitive OAuth tokens via the KMS, establishing a **high-trust zone** for secrets.

Components:

- **Next.js Frontend (Web):** Serves the user-facing React app. It handles UI, local input validation, and calls backend API routes for data. No sensitive secrets are stored on the client; it uses secure cookies or OAuth flows for auth tokens.
- **Next.js API Server (Backend):** Implements server-side API routes and business logic. This includes handling webhooks (e.g., Stripe events), orchestrating AI-driven proposals, and serving data to the frontend. The API server authenticates requests (e.g., session token or OAuth) and enforces role checks.
- **Token Broker Service:** A dedicated module (or microservice) for managing third-party tokens (Open Finance, Gmail, etc.). It receives OAuth authorization codes or tokens from the API server, uses KMS to encrypt and store them (only storing ciphertext and metadata), and later retrieves and decrypts tokens when needed for API calls. The API server never sees tokens in plaintext except at moment of use, and ideally delegates external API calls to the broker. This design limits exposure of sensitive OAuth tokens – only the broker (running with privileged KMS access) can decrypt them [3].
- **PostgreSQL Database:** The primary data store for user data, transactions, etc. It enforces **Row-Level Security (RLS)** policies to isolate tenant data by owner (each query is transparently filtered by user/organization ID) [4] [5]. The DB holds minimal sensitive data – most secrets/tokens are stored encrypted or in the KMS. Built-in encryption (e.g., TDE on the volume, or field-level encryption for particular columns) is enabled for defense in depth.
- **Redis Cache:** An in-memory store (for sessions, cache, or short-lived data). It does not persist personal data long-term. It resides in the internal network and is access-controlled by the API server.
- **S3 Secure Storage:** Used for storing files like receipt images or invoice PDFs fetched from Gmail. Files are stored encrypted at rest (S3-managed keys or KMS-managed encryption). Access control ensures only authorized users (and service roles) can read specific files.
- **KMS (Key Management Service):** A cloud KMS (e.g., AWS KMS) that manages master encryption keys. The Token Broker and other services use it for envelope encryption – generating data encryption keys to encrypt sensitive values (like OAuth refresh tokens, AI module secrets, etc.), and the master keys never leave the KMS [3]. KMS audit logs record every key use.
- **Workers & AI Module:** Background job processes handle asynchronous tasks: e.g., pulling new bank transactions via Belvo, processing Gmail inbox for receipts, and running AI analyses for change proposals. These workers operate with least privilege – e.g., they call the Token Broker to get a temporary access token for Gmail rather than storing long-lived credentials.
- **Observability Stack:** Centralized logging and monitoring (e.g., ELK stack or cloud monitoring). Application logs, audit trails, and metrics are sent here. Logs leaving the internal network are scrubbed of sensitive data (see **Log Redaction** later) to uphold privacy.

Token & Data Flows: When a user connects an external account (e.g., authorizing their bank via Belvo), the OAuth callback is handled by the API server within a secure route. The Token Broker encrypts and stores the

received refresh token using KMS (no plaintext persist) [3]. Only a reference or token ID is stored in Postgres (classed as sensitive but not usable without decryption). When data needs to be fetched (e.g., refresh bank data), the API server calls the broker, which decrypts the token in-memory (via KMS) and contacts the external API, then returns the data. Similar flows occur for Stripe (though card data never enters our system) and Gmail integration. All API calls use TLS 1.2+ to ensure transport encryption [6].

The architecture is designed for **zero trust** toward client-side input and external data. All incoming data (web requests, webhook payloads) are validated and sanitized. The ASCII DFD above marks the internet-facing boundary and the internal network boundary. These boundaries indicate transitions where data must be encrypted in transit and where new trust assumptions apply (e.g., code running inside the internal boundary is authenticated and monitored, whereas anything outside is untrusted) [6].

Data Classification Model

We adopt a **4-class data classification scheme** to tag and handle data according to sensitivity. Each class has defined rules for secure transport, encryption, retention, masking, and Data Subject Rights (DSR) handling:

- **Class 1 – Public Data:** Non-confidential information intended for public or broad consumption. *Examples:* Public blog posts, system status page, general knowledge base.
- **Transport & Storage:** May be transmitted or stored without encryption (though we generally use TLS for all web traffic by default). No special encryption at rest is required beyond standard disk encryption.
- **Retention:** No strict limitations; keep as long as business needs (e.g. published materials).
- **Masking:** Not applicable – contains no personal or sensitive data.
- **DSR:** Not personal data, so data subject rights requests (access/delete) are not applicable.

- **Class 2 – Internal Data:** Internal operational data that is not public but not highly sensitive. *Examples:* Internal project plans, system configuration without secrets, user interface preferences. Exposure could be mildly harmful or embarrassing but not a security breach.

- **Transport & Storage:** Must be encrypted in transit (TLS) whenever leaving the internal network [6]. Within the protected network, plaintext is acceptable if access controls are in place. At rest, rely on full-disk or database-level encryption as provided by infrastructure.
- **Retention:** Retain as needed for operations; periodic review to purge stale data. No regulatory requirement to delete on user request unless it contains personal data.
- **Masking:** Not usually required, but if logs or screens display internal identifiers, ensure they are not misinterpreted or leaked externally.
- **DSR:** If any personal data falls in this class (e.g., user support emails, which may be internal), it must be provided or deleted on request. Internal business data not containing personal info is outside DSR scope.

- **Class 3 – Confidential Data:** Sensitive personal or business data that **requires protection**. This includes most user account data, profile information, financial transaction details, and any personal identifiers. Unauthorized disclosure could harm individuals or the business.

- **Transport:** Always encrypted in transit (TLS 1.2+ for external, and either TLS or mutually authenticated connections internally).
- **Encryption at Rest:** Yes – store encrypted or hashed. For example, PII and financial records in the database are encrypted using strong algorithms (AES-256) or protected via field-level encryption. The system enforces that *regulated private data is encrypted at rest* [7] .
- **Retention:** Limit to what is necessary for the purpose [6] . Implement retention schedules (e.g., delete or anonymize inactive user data after X years). Retain financial records as required by law (e.g., invoices for 5+ years) but archive them securely.
- **Masking:** Apply masking in outputs and logs. E.g., show only last 4 digits of account numbers or credit card (store only last4 and tokens, never full PAN) to users; never log full personal details. Use masking or tokenization for display of identifiers (email, phone) where appropriate.

- **DSR Handling:** Subject to **LGPD rights** – upon request, the user can get an export of all their Class 3 data, or request deletion. Deletion may be fulfilled by either full removal or irreversible anonymization (if needed to keep data for stats or legal obligations) [8] [9] . We log such requests and ensure they propagate to backups or third-parties as required [10] .

- **Class 4 – Highly Sensitive (Restricted) Data:** The most sensitive information. This includes credentials and secrets (passwords, OAuth refresh tokens), encryption keys, full payment details (though we avoid storing these), and sensitive personal data defined by LGPD (racial origin, health info – which we typically **do not** collect). Unauthorized access could cause significant harm or regulatory impact.

- **Transport:** Strictly encrypted in transit; exchange only over secure channels. Internal access is compartmentalized – e.g., tokens only pass through the Token Broker service, not via broader internal APIs. Consider additional measures like mTLS or network segmentation for any service handling this class.
- **Encryption at Rest:** Mandatory. Use strong encryption and **KMS envelope encryption** for storage [3] . For example, OAuth tokens are stored only in encrypted form (ciphertext) in the database; encryption keys are managed by KMS and not stored in app code. Highly sensitive personal data (if ever stored) is encrypted field-wise, using keys in a secure vault or derived per user. Hash or HMAC secrets if possible (for passwords we hash, not encrypt). **Access to keys** is tightly controlled (KMS policies ensure only the Token Broker IAM role can decrypt those tokens) [3] .
- **Retention:** Keep to absolute minimum. If possible, avoid storing at all (e.g., do not store raw Gmail message content beyond needed analysis, just store extracted necessary fields). Secrets like API tokens are rotated or auto-expire. If a user revokes consent, delete tokens immediately. Logs containing Class 4 data are forbidden or redacted.
- **Masking:** Display and log masking is **strict**. For example, even admins cannot view full sensitive values in dashboards – they'll see placeholders (e.g., passwords never shown, API tokens only partially visible if at all). Any traces in memory are cleared when not needed.
- **DSR:** Fully honored. If a data subject requests deletion of their highly sensitive data (e.g., they disconnect an account or close account), purge those secrets/tokens from the system promptly (and confirm revocation at source, e.g., revoke OAuth token at provider side). Provide a confirmation of deletion. For access requests, include information like where their data was shared (Article 18, VII: we must inform the user of any third-parties with whom data was shared) [11] . Because this class often comprises secrets not directly useful to the user, an export may not apply except to confirm what was held and that it’s now deleted.

Each data item in the system is labeled with one of these classes in our data inventory. This classification drives our data handling and protection measures [12] [13]. For example, user profile info (name, email) is Class 3 (Confidential) – it is sent encrypted over HTTPS, stored in Postgres with encryption at rest, and included in data exports. OAuth refresh tokens are Class 4 – they are never sent to the client, stored only encrypted via KMS, and omitted from any exports (since they are credentials, not user-provided content). By clearly classifying data, we enforce **Privacy by Design** principles like data minimization, need-to-know access, and proportional security controls [6].

Token & Secrets Policy

Managing secrets (API keys, tokens, credentials) is critical for this fintech app. Our Token & Secrets Policy covers the entire lifecycle of sensitive secrets, including OAuth tokens from Open Finance and Gmail integrations, Stripe/MercadoPago API keys, and internal secrets (JWT signing keys, database passwords). Key aspects of the policy:

- **No Plaintext Secrets at Rest:** No secret or OAuth token is stored in plaintext in code repositories, config files, or databases [6]. All secrets are either injected via secure vaults or environment variables at runtime, or stored encrypted using strong cryptography (AES-256 or better). For example, when we receive a Gmail OAuth refresh token, we immediately encrypt it with a KMS key before saving to Postgres. The encryption uses envelope technique – a data key encrypts the token, and the data key itself is encrypted with our master key in KMS [3]. The database ends up storing only ciphertext and perhaps an associated key identifier. **At no point do we write the plaintext token to disk**.

- **KMS Envelope Encryption:** We use a cloud KMS to handle encryption keys. Each category of secret has a designated KMS Customer Master Key (CMK) (e.g., one key for OAuth tokens, one for Stripe keys, etc.). When encrypting a secret, the Token Broker requests a new Data Encryption Key (DEK) from KMS (or uses the KMS API to encrypt directly if the secret is <= KMS size limits) [14] [15]. The workflow:

  - Token Broker sends plaintext to KMS `Encrypt` with the CMK ID.
  - KMS returns ciphertext blob; broker stores that in the DB (often base64-encoded).
  - To use the token, broker calls KMS `Decrypt` with the blob, gets plaintext in memory, and immediately uses it (e.g., to call Gmail API), then clears it from memory.

This ensures that **only the KMS can decrypt secrets**, and only services with KMS decrypt permissions (the Token Broker's role) can request it [3]. Developers or DB admins cannot accidentally see raw tokens. Even if the DB is compromised, attackers would get only ciphertext which is useless without KMS (and KMS is protected by IAM and hardware security).

- **Token Broker Isolation:** The Token Broker is the only component allowed to handle OAuth tokens in plaintext. It runs with minimal privileges – no direct external exposure – and uses a dedicated IAM role that permits KMS decrypt on the relevant keys. The main API server does **not** have KMS decrypt permissions for those keys, preventing an attacker who compromises the web server from directly extracting tokens. Instead, the web server must call the broker (internal API call) to get, for example, an access token for Belvo; the broker logs this request (for audit) and fetches/decrypts the token internally. This design limits the attack surface of secret access [3].

- **Secrets in Transit & Use:** All secrets are transported securely. During OAuth flows, tokens are delivered from providers to our backend over HTTPS redirects. When our services retrieve secrets, they do so either in-memory or via secure channels (calls to KMS are over TLS and signed requests). We ensure secrets are not exposed in logs or error messages. In memory, long-lived secrets are zeroed out after use (where the language allows) and not unnecessarily retained.

- **Secrets in Code:** No secrets (API keys, etc.) are hardcoded in source code or stored in the repo [16]. We use environment variable injection with a validation step (see code snippet in References) to ensure all required secrets are present at startup. For local dev, a .env file is used (not committed to Git). In CI/CD, secrets are provided via the pipeline's secure secret store (GitHub Actions secrets, etc.) and never printed. Our code repository is scanned for accidental secret commits using tools (pre-commit hooks with truffleHog or GitGuardian) [17].

- **OAuth Token Lifecycles:** OAuth tokens (access/refresh) are handled with care:

- Access tokens are short-lived and **never stored** persistently. We keep them in memory or cache and let them expire. If an access token leaks, its lifetime is limited.
- Refresh tokens (long-lived) are treated as **Class 4 secrets**. As described, they are encrypted at rest. On a user's request or if the user disconnects an integration, we delete their refresh token record and also revoke it at the provider (using the OAuth revocation endpoint).
- We implement automatic refresh: the Token Broker uses refresh tokens to get new access tokens as needed, and if a refresh ever fails (e.g., revoked), it informs the app to re-initiate OAuth with the user.
- **Rotation:** If an OAuth provider supports rotating refresh tokens or setting limited lifetimes, we adopt that. For example, if Belvo issues non-expiring refresh tokens, we treat them carefully; if expiring, we ensure to capture new ones and overwrite the old (still encrypted).

- **Expiration:** For internal secrets like session tokens or API keys, we set explicit expirations. User session JWTs have short validity and require re-login or refresh with a refresh cookie after a certain time to reduce impact of theft.

- **KMS Key Management:** The KMS master keys themselves have policies and rotation:

- CMKs are rotated annually (a new backing key is generated by KMS), which does not invalidate existing ciphertext but provides forward security.
- Key policies restrict access: e.g., only our production AWS account and specifically the Token Broker service role can use the production token CMK [3]. Developers in dev environment use a different key to avoid any possibility of cross-environment access.
- For some use-cases, we might use *encryption context* with KMS to bind ciphertext to certain context (adding another check on decrypt).
- **Break-Glass Protocol:** In case the Token Broker or KMS is unavailable or an emergency demands direct access to tokens (e.g., to migrate to a new system if broker is down), we have an emergency **break-glass procedure** [3]. This entails using secure credentials (stored in a separate vault, with access sealed and requiring multi-party approval) to decrypt tokens. For example, a privileged operator could retrieve the encrypted token from DB and use a one-time KMS credential (the break-glass key) to decrypt. This process is strictly logged and monitored. Break-glass credentials themselves are stored securely (possibly a backup KMS key or printed to a sealed envelope in a safe)

and tested periodically for validity [3]. Any use of break-glass is treated as a security incident, triggering post-mortem review.

- All secret access events (normal or break-glass) are logged (who accessed, why, when) [18] for accountability.

- **Secret Injection & Isolation:** We use distinct secrets for different environments and purposes. Development and testing environments have their own API keys and tokens (often dummy or limited scope) – never using production secrets. Likewise, each microservice has unique credentials (no sharing of a single database password between all services, for instance). This limits the blast radius if one credential is compromised [19] [20]. Our configuration uses environment-specific `.env` files validated by a schema (see Zod example in references) to avoid missing or misconfigured secrets.

- **Auditing & Scanning:** We maintain an inventory of all secrets and tokens, including metadata (creation time, last use, owner service) [18]. Automated scans check repositories for leaked secrets and CI pipelines prevent adding new secrets to code. We also periodically review IAM and API key usage to revoke any that are unused or over-privileged. Secrets that support rotation (DB passwords, API keys) are rotated on a scheduled basis (e.g., every 90 days for database passwords) to reduce the window of compromise [3].

This comprehensive approach aligns with OWASP guidance on secrets management (centralize secrets, control access, automate rotation) [21] [19]. By combining **technical controls** (encryption, access restrictions) with **policy controls** (inventory, rotation schedules, emergency procedures), we ensure secrets remain confidential and their misuse can be detected or prevented.

Access Control Design (RBAC, ABAC, RLS)

Our access control model follows the principles of **least privilege** and layered enforcement. It combines Role-Based Access Control (RBAC) at the application level, Attribute-Based Access Control (ABAC) for context-specific rules, and database-level row security to prevent data leakage across tenants.

Roles (RBAC): We define three primary roles in the application: **Owner**, **Agent**, and **Viewer**.

- *Owner*: Typically the account owner or admin. Full control over their organization's data – can invite users, configure integrations, approve AI change proposals, and view/edit all records. For example, a small business owner using the fintech app is an Owner for their tenant data.
- *Agent*: A delegated user with limited edit rights. E.g., an accountant or support representative. They can perform certain actions like creating proposals or uploading receipts, but might not manage users or see sensitive settings. Their permissions are a subset of Owner's.
- *Viewer*: Read-only access. They can view dashboards, reports, and receipts but cannot make changes. E.g., an auditor or a client who should only monitor data.
- The RBAC implementation is done in our Next.js API routes and front-end: routes and UI components check the user's role and either grant or deny actions accordingly. We maintain permission matrices mapping each feature to roles allowed.

RBAC provides a clear baseline of who can do what based on role [22]. However, roles alone are not enough for fine-grained control.

Attribute-Based Rules (ABAC): We incorporate additional attributes and context into decisions: - **Tenant Isolation:** Every data record (e.g., transactions, proposals) is tagged with an `org_id` or equivalent. Users have an attribute of which organization(s) they belong to. The system ensures a user can only access data with matching org_id. This is enforced both at the app layer and the DB (via RLS). - **Record Ownership:** Some objects might be owned by specific users (e.g., a draft proposal is owned by its creator until published). ABAC rules say that only the owner or designated approvers can edit that object, even if others in the org have the same role. - **Time-based access:** If needed, we could add rules like "agents can only approve transactions during business hours" or "after an account is closed, all roles lose access after X days" – these are contextual conditions beyond static roles. - **Device/context:** Not currently implemented, but in future we might restrict certain actions to certain contexts (e.g., if we detect an untrusted device or location, we might restrict high-risk actions). This would be ABAC (using attributes of session risk).

In practice, ABAC conditions are coded as additional checks in the backend. For example, when an Agent attempts to approve a change proposal, the backend checks: does this Agent have the "approver" attribute for this workflow? If not, the action is denied even if the RBAC role might allow editing something else.

This combination (RBAC for broad strokes, ABAC for specific rules) yields a flexible yet manageable model [23] [24]. It prevents role explosion (we avoid creating dozens of roles by using attributes to refine permissions) [25] [22].

Row-Level Security (RLS) in Postgres: At the database level, we enable RLS to ensure that even if a query is executed without proper filtering, the database itself will prevent cross-tenant data exposure [4] [26].

- In the Postgres schema, every table with user data has a column `org_id` (or `tenant_id`). We create an RLS policy for each such table, e.g.:

```
ALTER TABLE transaction ENABLE ROW LEVEL SECURITY;
CREATE POLICY org_isolation ON transaction
  USING (org_id = current_setting('app.current_org_id')::uuid);
```

When a connection is made, our app sets `app.current_org_id` to the user's org ID (using Postgres `SET` command) [27] [28]. The RLS policy then ensures every query on `transaction` table only sees rows where `org_id` matches that context. If a query is attempted without setting the context or with a mismatched org, it returns no rows. This mechanism is enforced by the DB engine itself, adding a safety net under our application checks.

- We also leverage RLS for user-specific restrictions if needed. For instance, a `receipt` table might have both an `org_id` and a `visibility` column (to allow sharing receipts with external auditors via a special link). A policy can allow a read if either the org matches or a specific flag is set and the user has a certain attribute. RLS can consider session variables and even join conditions in policies.

- Additionally, all database access goes through an application service account (we do not connect as superuser for normal operations). This service account is granted minimal rights on tables and is subject to RLS (we use `ALTER ROLE ... SET force_rls = ON` to ensure it can't bypass RLS). Separate roles or schemas might be used for admin/maintenance that are not accessible from the app.

Application Enforcement: On top of RLS, our application code includes explicit checks. Before performing any operation, the API handler verifies the user's role and ownership. This dual enforcement (app + DB) mitigates mistakes: even if a developer forgets an auth check in code, RLS stops the data leak; and vice versa, if RLS was misconfigured, the app's checks still protect.

Strict Environment Isolation: We isolate access control between environments: - Developers have access only to dev/test data, which is anonymized anyway. They cannot elevate privileges to see production data. The production environment has separate credentials and is accessible only by the live app and a few ops personnel. - Within production, secrets and configs for each microservice are separate. For example, the Next.js server cannot accidentally query the analytics database (different credentials), and staging/test tenants in the system are flagged so that if by any chance their data intermixes, it's non-production data (this is an extra assurance against mistakes). - Admin interfaces (for our internal staff) are segregated from the customer app. If we provide an admin dashboard, it has its own authentication and authorization, preventing any possibility that a normal user could access admin functions.

Auditing Access: Every access control decision that is security-sensitive is logged. For instance, if an agent attempts an action and is denied due to ABAC, we log that attempt (without sensitive details, to avoid creating an oracle for guessing data). Successful admin logins, role changes, or permission grants are recorded in the audit log (see Audit Logging). This helps us demonstrate compliance (e.g., showing that only authorized users accessed personal data, as required by LGPD's security principle [6]).

Summary: With RBAC we manage broad permissions by role; with ABAC we enforce context-specific rules (who, what, when, where); and with RLS we put a hard partition between tenants' data at the lowest level [4] [29]. This multi-layer approach aligns with **OWASP ASVS 1.4** (access control design) and **NIST SP 800-53 AC-family** controls, ensuring that users can only access what they're permitted, and any attempt to bypass controls (e.g. via SQL injection or URL manipulation) will be stopped at multiple points.

Privacy by Design (LGPD Compliance)

From the outset, the application is designed with **Privacy by Design** principles to comply with Brazil's LGPD (Lei Geral de Proteção de Dados) and similar regulations. Here we outline how legal bases for data processing are documented, how we enable data subject rights (DSRs), and how we minimize and secure personal data, demonstrating compliance with key LGPD requirements.

Legal Bases & Purpose Limitation: We map each type of personal data and processing activity to one of the legal bases under LGPD (Article 7) [30] [31]: - Most data processing for our services falls under **contractual necessity** (Art. 7, V) [32] – e.g., using financial data to provide the dashboard service the user signed up for. - Some may fall under **legitimate interest** (Art. 7, IX) [33] – e.g., basic product analytics or fraud prevention, ensuring we document a Legitimate Interest Assessment and that the processing is necessary and does not override user rights [34] [35]. - Where none of those apply and especially for integrations like reading Gmail receipts or Open Finance data, we rely on **explicit consent** (Art. 7, I) [36]. For example, when a user connects their Gmail account, we present a clear consent dialog explaining that we will access their receipt emails for purpose X, and they must agree. - All consent obtained is **granular and documented**. Our privacy notice and UX make it clear what data will be accessed and for what purpose (LGPD Art. 6(I) – purpose principle [37]). We do not bundle unrelated consents. Consent records include the text presented and a timestamp of user's agreement.

Consent Logging: We maintain a **Consent Log** (part of audit logs or a separate table) to record consents given or withdrawn [38]. Each entry captures: user ID, consent type (e.g., "Gmail receipts access"), timestamp, and version of the disclosure text. If a user revokes consent (which LGPD allows at any time) [38] [39], we record the revocation timestamp as well. For instance, if a user disconnects their Gmail, that triggers a consent revocation log and deletion of fetched data (unless needed for another legal basis like contract performance).

DSR Endpoints: We provide **Data Subject Rights** endpoints to fulfill LGPD Article 18 requirements [8] [9]: - *Confirmation & Access (Art.18 I, II)*: Users can request a copy of their personal data. Through the app's settings, a user can click "Export My Data". This triggers a process that gathers all personal data about that user across our systems (account profile, transactions, receipts, logs that are attributable, etc.). The data is compiled in a structured format (JSON or CSV, and PDF for certain docs) and made available for secure download or emailed to the verified user. We ensure this includes metadata like whom data was shared with (Art.18 VII) [11]. - *Correction (Art.18 III)*: Users can update certain data directly (profile info). If they find inaccuracies they cannot change (e.g., an imported record error), our support process handles rectification on request. - *Anonymization or Deletion (Art.18 IV and VI)*: Users may request deletion of their data. Our app provides a "Delete My Account" function. Upon confirmation, we delete personal data: remove user profile, all transactions, receipts, and references. Instead of hard deleting financial records that we might need for compliance, we **anonymize** them: e.g., remove personal identifiers and retain only aggregated or generic info if needed (Art.16 of LGPD allows retention for compliance even after consent withdrawal, but data should be anonymized) [40] [9]. The deletion process covers not just the primary DB but also derivative data: we delete files from S3, redact personal data in logs (or dissociate them from user ID), and instruct third-party processors to delete data if they had any. - *Data Portability (Art.18 V)*: The "Export My Data" feature also serves portability. We use common formats (CSV for transactions, JSON for profile) so that the user could import it elsewhere. - *Withdrawal of Consent (Art.18 IX)*: As mentioned, users can revoke consent easily – e.g., a toggle to disable an integration or an overall "I withdraw consent for processing" which we treat as a request to cease processing except under other legal bases. The UI and our privacy policy inform users of the consequences of denial or withdrawal of consent (LGPD Art.18 VIII) [41] – for instance, "If you withdraw consent for Open Finance data, the dashboard will no longer update your financial info." - We have a designated channel (DPO email/portal) as well, for users to exercise rights via contact if they prefer. Requests through that channel are authenticated (we verify identity via registered email, etc.) and logged.

Our DSR handling is automated where possible, to meet LGPD's timelines (the law says requests must be answered "within a reasonable time", our policy aims for immediate for electronic access, or within 15 days for detailed reports, mirroring Art.19's 15-day rule for full data report) [42] [43].

Consent and Preference Management: The app provides a privacy settings panel where users can see what consents they have given (e.g., "You authorized us to access your Gmail on 2025-01-10") and revoke them. Also, any optional data uses (like using data for product improvement or marketing) are off by default unless user opts-in (Privacy by default principle). We track user communication preferences (for email notifications, etc.) separately from core consents.

Transparency & Notice: We maintain an up-to-date Privacy Policy, written clearly (no confusing legalese) covering LGPD-required disclosures: what data we collect, purposes, legal bases, rights, how to exercise them, our DPO contact, etc. At account creation, users must acknowledge this privacy notice. For any significant change in processing (new purpose), we will notify users and obtain fresh consent if required (per LGPD Art.9 §2 about changes of purpose requiring info and opt-out) [44] [45].

Data Mapping & ROPA: Internally, we keep a **Record of Processing Activities (ROPA)**. This document (or spreadsheet) lists: - Data categories (e.g., user profile, financial transaction, support ticket), - Purposes of processing each (e.g., provide service, fraud prevention, marketing), - Legal basis for each, - Where data is stored/transferred (systems, third-parties like Stripe or email service), - Retention period, - Security measures in place (e.g., "Transactions: stored in Postgres, encrypted at rest, accessible only to owner and agents, retained 7 years for financial compliance"). - We include LGPD's required info like controller and DPO identity, and any cross-border transfers (e.g., if data is hosted on a cloud outside Brazil, ensure adequacy or standard clauses). This ROPA helps demonstrate compliance and is available if the ANPD (Brazil's Data Protection Authority) requests it.

Minimization & Purpose Limitation: We practice data minimization: only collect data that is necessary. For example, we don't ask for a user's national ID or date of birth since it's not needed for the service. When connecting Gmail, we narrow the OAuth scope to only Gmail read-only access to specific labels (if possible) or we fetch only emails that look like receipts, then immediately parse and discard the raw content. We **avoid collecting sensitive personal data** (Art.5 II defines sensitive data like health, religion, etc., which we intentionally do not deal with except possibly financial, which is not considered sensitive under LGPD unless it's credit score or similar). If in future we handle any sensitive data category, we will obtain explicit consent and additional safeguards (Art.11).

Limited Retention: In line with LGPD's data quality principles, we don't keep personal data longer than needed. For instance, if a user cancels the service, we schedule their data for deletion after a grace period. Regular database cleanup jobs enforce retention limits (e.g., delete or anonymize records older than X years if they're not needed). We also respect specific user requests to delete sooner (as above).

Security of Personal Data (Art.46): Our entire security architecture (encryption, access control, audit logging, etc.) is aimed at protecting personal data from unauthorized access [6]. LGPD Art.46's mandate for technical and administrative measures is satisfied by the controls described throughout this guide (encryption, access control, monitoring, etc.), ensuring an appropriate level of security given the sensitivity of the data [6] [46].

Logging & Monitoring with Privacy in Mind: We implement **logging minimization**: Personal data is sanitized in logs. For example, instead of logging "User John Doe with email john@example.com connected bank account 1234", our logs say "User [ID:abc123] connected a new bank account." Email and names are not logged in plaintext in general logs. If an error must include some data for debugging, we ensure it's not sensitive (or we hash it). We also avoid logging contents of financial records or email bodies. Sensitive operations (like exporting data or accessing tokens) are logged, but even those logs contain user IDs or token IDs, not the data itself. This way, we have enough for security auditing without exposing PII in log stores [47]. Our observability stack is configured to **mask** known patterns (like anything looking like an email or credit card in log messages is replaced with "[REDACTED]" via log processing filters).

Third-Party Processors: We have Data Processing Agreements with any third-party that handles personal data (cloud providers, etc.), and ensure they comply with LGPD too. E.g., Stripe is a payment processor handling card data under its PCI scope, and we treat them as data processors for relevant personal data (like a customer's email if passed to Stripe for receipts). We maintain a list of sub-processors and disclose it.

Breach Response (Art.48): Our Incident Response Plan (detailed later) includes procedures to notify ANPD and users in case of a data breach that may result in risk to them [48] [49]. We are aware of the regulation

from April 2024 on breach notification and have a template ready to fill in the info required (description of incident, affected data, measures taken, etc.) [50] [51]. Notification will be within the timeline defined by ANPD (which currently requires "immediate" notice, interpreted as within 2 business days of learning of the incident, unless justified). We also have processes to communicate the incident to users with guidance on steps they should take.

In summary, privacy considerations are baked into our design: we only collect what we need (minimization), we secure and isolate it (protection), we keep records of processing (accountability), and we provide users control and transparency (DSRs, consents). This aligns with LGPD's principles of purpose limitation, necessity, transparency, security, prevention, and accountability (Art.6) [37] and demonstrates compliance through concrete features and policies.

PCI DSS Scope and Controls

Handling payment data requires special attention to the Payment Card Industry Data Security Standard (PCI DSS). We have architected the system to **minimize PCI scope by design**. Our approach is to avoid processing or storing sensitive cardholder data on our servers altogether, using tokenization and hosted payment pages provided by Stripe/Mercado Pago. This significantly reduces our compliance burden and risk [2].

Out-of-Scope by Design: - **No PAN or CVV storage:** The application *never stores full Primary Account Numbers (PAN)* (credit card numbers), CVV codes, or magnetic stripe data. When users enter card details (for subscription payments, etc.), we utilize Stripe Elements or Mercado Pago's hosted fields. These capture card data securely in the user's browser and send it **directly to Stripe/MercadoPago** servers, not through our backend [1] [2]. We receive back a *token* or *payment method ID*. This token is essentially a placeholder that represents the card in future transactions. The token itself is not the PAN; it cannot be used outside of the payment provider's context. - **Limited Card Data Stored:** We only store the minimal card info needed for user reference: typically card type (Visa/Mastercard), expiration (maybe), and last 4 digits of the card (last4). The last4 and perhaps expiration are **not considered sensitive** (they're not sufficient to reconstruct the card or charge it without the token and provider). These are Class 3 data (Confidential, personal but not highly sensitive) and are stored in our database for display ("Card ending in 1234, expires 12/25"). Full PAN never touches or persists in our system, keeping us out of PCI's Cardholder Data Environment. - **Payment Processing:** All actual payment operations (charges, refunds) are done by calling Stripe/MercadoPago APIs using the token. These API calls are server-to-server and encrypted. Our servers authenticate to Stripe with a secret API key, which is sensitive but easier to secure than handling raw card data. Stripe's platform, being PCI Level 1 certified, handles the card data and vaulting. According to Stripe's guidance, using Elements and tokens keeps our servers out of PCI scope or at most in SAQ A-EP category (if we host the payment form) [52] [2]. We follow their best practices to maintain that status.

Compensating Controls & Future Scope: We recognize that if our design ever changed to handle card data directly (not recommended), we would enter PCI scope. In such an event, we have a plan of compensating controls ready: - Use a **dedicated Cardholder Data Environment (CDE)**: segregate any system component that processes/stores PAN into an isolated network segment with strict firewall rules (fulfilling PCI Req.1 segmentation). Only necessary connections to the main app would be allowed. - **Encryption & Key Management:** PANs would be immediately encrypted with strong encryption (e.g., AES-256) with keys managed in an HSM or KMS. Key management would follow PCI requirements (dual control, separation of duties for key custodians, regular rotation). - **Hashing for indexing:** If we needed to store PAN for lookup,

we'd store a cryptographic hash for search, and the encrypted PAN for retrieval, ensuring no plain PAN in DB. - **Access Control:** Very few accounts would have access to decrypt card data. Those that do would require multi-factor authentication and justification logging. We would implement monitoring to alert on any access to full PAN. - **Workstation security:** Any machine that could access PAN (for support or whatever) would need to be secured per PCI (no internet, isolated VLAN, etc.). Ideally, even support would never see PAN – they'd use tokens. - **PCI Policies:** We'd update policies to meet PCI's 12 requirements: maintain firewalls, default passwords change, protect stored data (we already do), encrypt transmission (already do), use AV and patch (we do in general; would ensure on CDE servers), restrict access by business need (yes), identify and authenticate access (unique IDs, MFA), log and monitor (our audit logs can cover this), test security systems (regular vulnerability scans and pen tests focusing on CDE), and maintain an information security policy around card data. - **Quarterly Scans and Yearly Audit:** If in full PCI scope, we'd subject the environment to ASV (Approved Scan Vendor) external scans quarterly and potentially annual onsite assessment if volume mandates Level 1. As a small fintech, we'd try to stay at SAQ A or A-EP level by keeping scope minimal.

Tokenization & Scope Reduction: Our current method is effectively tokenization: sensitive card info is replaced by tokens by Stripe [53]. According to PCI guidelines, tokens that cannot be used to recover the original PAN are not considered cardholder data, so storing them doesn't bring the same requirements [2]. However, **we treat the tokens securely** anyway: they are stored as secrets (Class 3) and access to use those tokens (i.e., to charge a card) is limited to backend processes with the Stripe API key. The Stripe API key is stored securely (in KMS-encrypted env variable) and rotated if needed.

Compliance Attestation: Because we do not store or process PAN/CVV, we can self-attest with a simpler PCI SAQ (Self-Assessment Questionnaire) – likely SAQ A or A-EP, meaning most technical PCI controls are "Not applicable" to us, aside from securing the webpage that hosts the payment form. We ensure our website hosting the payment form is secure: we serve it over TLS, we have a strong Content Security Policy and other headers to prevent injection (since an XSS that skims cards would put us in scope!). This fulfills Stripe's criteria for SAQ A-EP (where card data is directly posted to Stripe, but our site is the payment page).

Monitoring and Alerts for Payment Flows: Even though we offload card processing, we monitor for any anomalies in payment flows that could indicate security issues. For instance, we track if there's any JavaScript changes on the payment page (to detect a Magecart-style attack where malicious script steals card details). We use Subresource Integrity and CSP to lock down scripts. We also rely on Stripe's built-in fraud detection for transactions, and if we see unusual purchase patterns, we investigate.

Compensating Control for PAN in receipts: A possible angle: user-uploaded receipts might contain card PAN (e.g., last4 or masked number on a receipt). These typically show only last4 or a masked PAN (e.g., **** **** 1234) – which is not full PAN. If a receipt image did show a full card number (rare, usually only last4 is printed), that's technically card data but as a scanned document. We treat all receipts as personal data (Confidential) and they are stored encrypted. We don't OCR or parse full PAN out of them (and our terms likely forbid uploading raw PAN). Thus, the risk is low.

Extending Scope (e.g., direct debits or PIX data): If in future we handle bank account numbers or PIX keys, these are sensitive but not PCI. However, we'd handle similarly carefully (encrypt at rest, treat akin to card data).

Conclusion: By avoiding direct handling of card data, we **minimize our PCI DSS scope** [2]. Should our business requirements change unexpectedly to require handling such data, we have a plan to quickly harden the environment to meet PCI controls. But the preferred approach is to continue leveraging PCI-compliant providers (Stripe, etc.) to process payments, while we focus on securing the data we do store (which is mostly tokenized or last4, plus personal data under LGPD).

This strategy dramatically reduces risk: even if our system is compromised, attackers cannot steal card numbers because we simply don't have them [2]. They would only get tokens (which are useless outside Stripe) or last4 (not enough to transact). Thus, our users' payment data is largely safe even beyond our security measures, thanks to this architectural choice.

Immutable Audit Logging Design

An immutable, tamper-evident audit log system is in place to record security-relevant events and provide forensic evidence in case of incidents. Every critical action in the system is logged in a way that **detects any tampering or deletion** of log entries [54] [55]. Here's how our audit logging is structured and implemented:

Audit Log Content: The audit log captures events such as: - Authentication events: user logins, logouts, MFA challenges (with outcome success/failure). - Access control events: whenever a user attempts an action beyond their privilege (denied by RBAC/ABAC), or any manual role changes (e.g., admin grants an agent higher access). - Data lifecycle: exporting data (DSR fulfillment), deleting data (user-initiated or scheduled), any changes to sensitive fields (e.g., user changed their email, or our AI modified a record upon approval). - Integration events: e.g., "User X linked bank account Y via Belvo", "Refresh token for User X was used to fetch data", "User revoked Gmail access". - Administrative actions: config changes, enabling/disabling features, key rotations, break-glass uses. If an admin or engineer uses a debug interface or break-glass credential, that's logged. - System alerts: e.g., detection of suspicious activity (we might log "Multiple failed login attempts for user X – account locked").

Schema: Audit logs are stored in an `audit_log` table (and also optionally streamed to an external WORM storage for backup). The table columns include: - `id` (big serial primary key, for ordering), - `timestamp` (with time zone), - `actor_type` and `actor_id` (who did it: could be `User:12345` or `System:Scheduler` or `Service:TokenBroker`), - `action` (a short code or phrase like `LOGIN_SUCCESS`, `EXPORT_REQUESTED`, `ROLE_CHANGED`), - `entity_type` and `entity_id` (what object was acted on, e.g., `Transaction:98765` or `User:12345` if one user changed another's data), - `details` (JSON blob with any additional info like IP address, user-agent, change diff, etc., excluding sensitive data but enough for context), - `prev_hash` (text or binary for the hash of the previous log entry), - `hash` (hash of this log entry's content + `prev_hash`).

Tamper-Evident Hash Chain: Each log entry's `hash` is computed as a cryptographic SHA-256 (for example) over the concatenation of important fields of the entry plus the previous entry's hash [55] [56]. Essentially: `hash_n = SHA256(hash_{n-1} || timestamp_n || actor_n || action_n || entity_n || details_n)`. The first log entry in the chain uses a fixed `prev_hash` (like an all-zero string or a specific prefix) and we store its `hash`. Every subsequent entry includes the last entry's hash,

linking them into a chain [54] [55]. If any log entry were to be altered or removed, the chain's integrity would break: on verification, a recomputation would yield a different hash for a subsequent entry.

- We periodically (say daily) take the latest `hash` and store it separately (like writing to an append-only file or an external trusted service). This is akin to publishing a "checkpoint" of the log. It helps detect truncation attacks (where an attacker might delete the last X entries hoping no one notices) [56], because the last known hash wouldn't match the hash in the log after deletion.
- Optionally, we could also digitally sign the hashes or entire log entries with an asymmetric key to further prevent undetected tampering (but the hash chain itself already gives detection; signing gives non-repudiation that the log came from us).

Immutable Storage: The primary log table in Postgres is append-only: our application code never issues UPDATE or DELETE on it. The database role for the app is not given permission to delete from `audit_log`. In fact, we enable the `pg_appendonly` setting or use a trigger to prevent accidental deletion. We also regularly export these logs to a secure archive (could be an S3 bucket with Write-Once-Read-Many configuration or a tamper-proof ledger service). This ensures even if a DB admin tried to tamper, we have copies.

Audit Log Verification: We provide a utility (internal) that can verify the log chain. It scans through logs in order and recomputes the hashes to ensure each matches the stored `hash` field, flagging any discrepancy. This can be run before and after major events (or continuously by a monitor process). According to secure logging practices, verifying the chain lets us detect tampering promptly [54] [57]. In the event of an incident, we can run this verification to ensure the logs we rely on were not altered by an attacker trying to cover their tracks.

Provenance and Integrity for Receipts/Invoices: For documents like receipts and invoices imported into the system, we maintain provenance metadata. Each such record stored has: - Source identifier (e.g., "sourced from Gmail API on 2025-09-01 for user X" or "uploaded manually by user Y"). - If from an email, we might store email headers or a message ID to prove origin. - We calculate a cryptographic hash of the file (receipt PDF/image) when stored. This hash is stored in the audit log or a separate integrity field. This way, if later there's a dispute about an invoice (say a user claims the invoice was altered), we can show the hash that was logged when it was ingested and verify it matches the file we have. If someone somehow modified the file in storage, the hash comparison would fail. - In the audit log, any time a receipt or invoice is viewed or downloaded by a user, we log that access (for forensic and compliance, since financial data access might need tracking).

Log Schema Example: (In SQL-like form)

```
CREATE TABLE audit_log (
   id BIGSERIAL PRIMARY KEY,
   timestamp TIMESTAMPTZ NOT NULL DEFAULT now(),
   actor_type TEXT NOT NULL,
   actor_id TEXT NOT NULL,
   action TEXT NOT NULL,
   entity_type TEXT,
   entity_id TEXT,
```

```
    details JSONB,
    prev_hash VARCHAR(64),
    hash VARCHAR(64)
);
```

We use triggers or the application to set the `hash` field on insert. A simplified trigger logic:

```
CREATE OR REPLACE FUNCTION compute_audit_hash() RETURNS trigger AS $$
DECLARE
  last_hash VARCHAR(64);
  base_str TEXT;
BEGIN
  IF NEW.id = 1 THEN
    last_hash := '0000'; -- genesis
  ELSE
    SELECT hash INTO last_hash FROM audit_log ORDER BY id DESC LIMIT 1;
  END IF;
  base_str := COALESCE(last_hash, '') || NEW.timestamp || NEW.actor_type ||
NEW.actor_id || NEW.action || COALESCE(NEW.entity_type, '') ||
COALESCE(NEW.entity_id, '') || COALESCE(NEW.details::TEXT, '');
  NEW.prev_hash := last_hash;
  NEW.hash := encode(digest(base_str, 'sha256'), 'hex');
  RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

(The actual implementation would handle ordering carefully and maybe use a sequence for `prev_hash` if parallel inserts, but conceptually this is it.)

Secure Time Source: We ensure the `timestamp` is in UTC and taken from a reliable source (the DB server is NTP-synced). If an attacker changed the system clock, it could mess with log order or chain, so we monitor clock offset.

Log Access and Retention: - Access to audit logs is restricted. Only authorized admin or compliance personnel can query them, and even then through a secure interface (read-only). We treat audit logs as sensitive since they contain records of actions (though we try not to log confidential data, the existence of certain events can itself be sensitive). - We retain audit logs for a long period (e.g., at least 2 years or more) since they are crucial for forensic investigations and compliance evidence. They are archived periodically but never altered or deleted (except when legally required to remove something, in which case we would likely redact by adding a new log entry stating "entry X redacted for GDPR request", rather than actually dropping it).

Tamper Evidence in Practice: If an attacker were to gain DB access and attempt to cover their tracks by deleting or altering logs, the hash chain would reveal gaps or mismatches. For example, if they remove an entry, then the `prev_hash` of the next entry won't match the `hash` of the previous one we have on

record. Our verification would flag "Log entry 105 expected prev_hash abc but found def – possible deletion or insertion detected." Each entry's integrity depends on all previous ones, so it's like a blockchain in linear form [54]. Even truncating the end (deleting latest entries) can be detected if we have saved checkpoints externally.

Additionally, we could cross-verify by storing log hashes in an external system. A simple method: every midnight, take the hash of the last log entry of the day and email it to our security mailbox (or post on an immutable ledger or even tweet it on a private account). If someone tried to change history later, they couldn't recompute that past hash without detection.

Compliance Mapping: These practices meet OWASP ASVS logging requirements and map to NIST 800-53 AU controls (Audit and Accountability), such as AU-2 (events to log), AU-6 (protection of logs), AU-10 (non-repudiation / integrity of logs) – which our hash chaining addresses [54] [57]. It also supports LGPD's accountability principle by providing evidence of activities and security measures (and note: LGPD Art. 46(§2) encourages keeping record of access to sensitive data, which our logs provide where applicable).

In summary, our **immutable audit log** acts as the system's black box recorder. It ensures that any significant action is recorded and that record cannot be altered without leaving obvious signs. This not only helps in breach investigations but also deters internal misuse (since employees know their actions are irrevocably logged) and provides regulators and partners assurance that we maintain strong operational oversight.

Threat Model (STRIDE & LINDDUN Analysis)

We performed a comprehensive threat modeling exercise, addressing both security threats (using the STRIDE model) and privacy threats (using the LINDDUN model). For each category of threat, we identified relevant risks in our system and implemented controls. Below is a summary of the threat model, along with mappings to controls/best practices, OWASP ASVS requirements, NIST 800-53 controls, and applicable LGPD principles/articles:

STRIDE Threats & Mitigations:

- **Spoofing (Identity Spoofing):** An attacker might attempt to impersonate a legitimate user or service (e.g., reuse a stolen token or pretend to be the Token Broker).
- *Risks:* Unauthorized login, session hijack, pretending to be a microservice to get data.
- *Controls:* Strong authentication for users (unique credentials, MFA for admin accounts) [6]. We issue secure, random session tokens (HTTP-only, secure cookies) and use JWT with signing to prevent forging. Access tokens are tied to user context and rotated. Internally, services authenticate via signed requests or IAM roles (the Token Broker verifies API server identity via network controls and maybe mutual TLS). Replay protections in OAuth (PKCE for OAuth flows) prevent token interception misuse.
- *Framework Mapping:* OWASP ASVS 2.1 (verify all logins, sessions are secure), ASVS 3.4 (protect session tokens). NIST 800-53 IA-2 (user identification and auth). LGPD: Article 46 security principle – ensuring only authorized access [6].

- **Tampering:** Unauthorized alteration of data in transit or at rest (e.g., man-in-the-middle altering API data, or attacker modifying records or logs).

- *Risks:* Changing financial transaction data, altering audit logs to hide tracks, tampering with tokens in DB.
- *Controls:* All communication uses TLS with strong ciphers (prevents transit tampering) [58] [6] . We sign or MAC certain data: e.g., audit log hash-chains to detect any modification [55] . Database Row-Level Security and application checks prevent unauthorized changes (you can't just change someone else's data because your role/org check stops it). Critical fields could be made write-once or require digital signatures (not currently, but e.g., we might sign an invoice file's hash). We also use integrity checks for files (hashing receipts).
- Backups and replicas are protected from tampering by restricting access and using append-only for logs.

- *Framework Mapping:* ASVS 9.1 (verify integrity protections for data), ASVS 10.2 (audit log integrity). NIST 800-53 SC-8 (integrity for data in transit), SC-13 (cryptographic protection), AU-10 (audit logs integrity). LGPD Art.46 requires measures against unauthorized alteration as well [6] .

- **Repudiation:** The risk that a user or system can deny performing an action without evidence, or maliciously acting without being traceable.

- *Risks:* A user claims "I never approved this payment" or an admin disavows a change they made.
- *Controls:* Our immutable audit logs with actor attribution provide non-repudiation [54] . Key actions (approvals, consent, etc.) are tied to user IDs and timestamps, so we can prove the origin. We might integrate digital signatures for very sensitive actions (not currently, but could sign transactions with user private key in some cases for complete non-repudiation). We ensure clocks are synchronized to have reliable timestamps.

- *Framework Mapping:* ASVS 10.1 (record all security-relevant events), 10.3.3 (capture user ID in logs), 10.3.6 (integrity of logs). NIST 800-53 AU-3 (content of audit records), AU-10 (non-repudiation). LGPD doesn't explicitly talk about repudiation, but having proof of consent (Art.8 §2 places burden of proof of consent on controller) [59] is akin to non-repudiation – our consent logs address that.

- **Information Disclosure:** Unauthorized access to sensitive data (data breach or privacy leak).

- *Risks:* Attacker exfiltrating personal data (user details, transactions), someone sniffing data in transit, developer misconfiguring and leaking a database, or logs exposing PII.
- *Controls:* **Encryption everywhere:** TLS for all external and internal data flows [6] ; encryption at rest for databases, backups, and S3 (we use KMS-managed keys on S3 buckets, TDE on DB volumes, etc.). Field-level encryption for the most sensitive fields (tokens, possibly salary or secret fields) using our KMS approach [3] . Access controls (RBAC/RLS) ensure even within the system, users can only see their data. We implement **least privilege** for service accounts – e.g., the web app's DB user cannot read the table that stores encrypted tokens at all; only the Token Broker's DB user can, and even then it only gets ciphertext.
- Logs and monitoring: We avoid sensitive data in logs, as discussed (minimize PII in logs). For external breach prevention, we have network security groups and cloud firewalls restricting database access to app servers only. Regular vulnerability scans and dependency checks (DevSecOps pipeline) reduce the risk of a known exploit leading to data leakage [17] .

- We also implement session timeouts and use secure cookies to prevent token theft via XSS or cookie theft (plus HttpOnly, Secure flags).

- *Framework Mapping:* ASVS 3.1 & 3.2 (all sensitive data encrypted in transit), 3.4 (sensitive data storage encryption) [7] . ASVS 9 (data protection in general). NIST 800-53 SC-28 (Protection of Information at Rest), SC-23 (session authenticity). LGPD Art.46 and Art.13/14 (which require controllers to adopt security measures and govern data protection in systems) [6] . Also LGPD's principle of necessity (Art. 6, III) – we only disclose minimum data internally on a need-to-know, fulfilling data minimization to reduce exposure surface.

- **Denial of Service (DoS):** Attackers may try to disrupt the service's availability (e.g., flooding the API, overwhelming the database, or targeting an expensive AI computation repeatedly).

- *Risks:* Service downtime (losing availability for dashboard), or performance degradation so users can't use features. Could be volumetric (massive traffic) or logical (sending a payload that causes high CPU).
- *Controls:* We implement rate limiting on our API endpoints (each IP and user account has throttling rules). Particularly expensive endpoints (like the AI proposal generation) have stricter quotas per day. We use a CDN/WAF in front of our Next.js app to absorb and filter common attacks. Our infrastructure is scalable (auto-scaling groups) to handle flash traffic within reason, and we have alerts if traffic goes beyond baseline.
- For resource-heavy tasks, we isolate them to worker queues so they can be processed at a controlled pace (preventing one user from tying up all workers). We also have timeouts and input validation to avoid pathological inputs (like someone sending a 100MB payload to an endpoint).
- Anti-DoS at network layer: Cloud provider DDoS protection, WAF rules (block obvious malicious patterns).

- *Framework Mapping:* ASVS 1.5.4 (limit resource use, e.g., upload limits) and 14.1 (configure secure rate limiting). NIST 800-53 SC-5 (Denial of Service Protection). While LGPD is about data, availability is indirectly a part of security expectations (and continuity of service – though not explicitly in LGPD, it's good practice under its security obligation).

- **Elevation of Privilege:** A user or attacker obtaining higher privileges than intended (e.g., a bug that lets a Viewer perform admin actions, or an injection attack giving shell access).

- *Risks:* Regular user becomes owner of someone else's data; an attacker without credentials manages to get code execution on our server and escalates to pivot into database or internal services.
- *Controls:* Rigorous access control checks at every layer (as described in Access Control section). We do defense in depth: even if one check fails, another catches it (e.g., the front-end shouldn't show the "delete user" button to a Viewer, but if they call the API directly, the API RBAC will still reject).
- We minimize vulnerable surfaces: keep software dependencies updated (DevSecOps pipeline with SAST/DAST to catch common vulns) [17] . The Next.js environment is configured with least privilege (no OS root runs; containers run as non-root where possible; AppArmor or seccomp profiles to limit syscalls).
- For preventing privilege escalation at system level: containers are isolated, separate roles for different processes (so if web server compromised, it doesn't automatically have full DB rights beyond its role). Secrets are scoped per service, as noted.

- Code is reviewed for injection flaws or logic bugs that could cause privilege escalation (like Insecure Direct Object References – we use framework features and our own checks to ensure one org's ID cannot fetch another org's data).
- *Framework Mapping:* ASVS 1.4.3 (enforce authorization on all requests including indirect object references), ASVS 4.X (general input validation to prevent injection leading to EoP). NIST 800-53 AC-6 (Least Privilege), CM-6 (least functionality – we disable or remove unnecessary services to reduce escalation paths). LGPD: not directly applicable, but ensuring only authorized personnel can access personal data maps to its security and access governance expectations [6] .

LINDDUN Privacy Threats & Mitigations:

- **Linkability:** Ability for an attacker (or even the system) to link two pieces of data that the subject would prefer not to be linked.
- *Example:* If the system used a single unique identifier for a user across different contexts, an external party intercepting different datasets could link them together.
- *Controls:* We separate contexts – e.g., tokens for different services are not the same or easily correlated. We avoid using long-term immutable identifiers externally; for instance, instead of exposing a user's internal UUID to external receipt links, we generate one-time tokens or random IDs.
- Data we share with third parties (like Belvo) is minimal such that it can't be used to track users across services (we might avoid sharing our internal user ID with them; use one-off codes).
- Within our system, all data is linkable by us by design (we need to compile a user's data), but we restrict linking when not necessary. E.g., analytics or monitoring might use anonymized or aggregated IDs not directly tied to personal identity.

- *LGPD Mapping:* Principle of necessity and data minimization (Art.6, III) – don't over-collect data that could be misused for linking profiles. We implement that by only storing what we need (less data to link).

- **Identifiability:** The possibility to identify a data subject from anonymized or pseudonymized data.

- *Example:* If we publish or share transaction stats, could someone identify a specific user's data? Or if an unauthorized person got access to a portion of data, could they re-identify who it belongs to?
- *Controls:* We pseudonymize data where possible for internal use. For example, in logs and analytics, we use user IDs (random UUIDs) instead of names or emails. If we need to do analytics on user behavior, we might use hashed IDs. When responding to external requests (like support or debugging), engineers use user ID references and a separate lookup if needed (so a dump of logs doesn't directly reveal identities).
- If we ever share data for research or partners (none currently, but if), we would aggregate or anonymize it (e.g., average spending per region, with k-anonymity if needed).
- Our deletion process ensures that when a user is deleted, we scrub identifiers so remaining data (like transactions kept for financial records) are fully anonymized and can't be linked back to that user (we remove personal identifiers).

- *LGPD Mapping:* LGPD encourages anonymization when data is kept beyond original purpose or for studies. We indeed anonymize for retention beyond user deletion (Art.18 IV mentions anonymization as an option) [60] . Our measures ensure that once anonymized, data cannot be re-identified easily (and if we do use reversible pseudonymization internally, those keys are protected).

- **Non-repudiation (Privacy side):** In LINDDUN, this refers to the ability of the system to prove actions of a user, which could conflict with user privacy if not properly handled (similar to Repudiation but from user perspective).

- *Example:* Our system logs every action by a user. While good for security, from a privacy view, one might consider if this could be used to profile or surveil the user beyond what they expect.
- *Controls:* We restrict audit log access (so it's not misused to track users by staff without cause). Also, we follow data minimization in logs (only what's needed to establish accountability, not contents).
- The privacy policy informs users that we keep logs for security and compliance.
- We balance between accountability and user privacy by not logging contents of what they did, just the fact of the action. E.g., log "User exported data at 10:00" but not the data itself.

- *LGPD Mapping:* This is more of an internal practice – ensure that internal use of logs doesn't violate purpose limitation (we log for security, we should not use those logs to say, build a marketing profile on user's usage patterns).

- **Detectability:** An attacker's ability to detect the presence of a user's data or an action.

- *Example:* Side-channel where an attacker can query our API with some ID and learn if that ID exists ("user not found" vs "wrong password" differences can allow enumeration of emails). Or an attacker observing timing (if user exists, maybe the response is faster).
- *Controls:* We implement uniform responses for certain queries. For login, we use generic error messages ("invalid credentials") rather than "email not registered" to avoid confirming whether a given email is in our system. For data fetch APIs, if something isn't found, we might also return a generic "not found or not authorized" message the same as if the item didn't exist to external eyes.
- At network level, no public endpoints reveal info without auth.
- We also watch out for timing attacks: for example, our cryptographic operations use constant-time comparisons for secrets (to not leak lengths or values).

- *LGPD Mapping:* Though detectability isn't directly a legal concept, it aligns with not exposing personal data to those not authorized. By preventing user enumeration, we protect personal info (like email addresses) from being harvested via our interface.

- **Disclosure of information (Privacy Breach):** This overlaps with Info Disclosure in STRIDE, specifically focused on personal data leakage.

- *Risks & Controls:* (Same as Info Disclosure above) – encryption, access control, etc. Additionally for privacy, we ensure when we share data with third parties (like sending an email via Gmail API, or in webhooks to our partner services), we share only necessary fields and use secure channels.
- If we integrate with an analytics service, we avoid sending raw personal data (we either self-host analytics or ensure any external analytics doesn't get GDPR/LGPD covered data without user consent).

- *LGPD Mapping:* This is core of LGPD – unauthorized disclosure = data breach. Our controls here (like encryption at rest, etc.) are precisely to prevent that (Art.46) [6]. We also have breach response to notify if something happens [51].

- **Unawareness:** Users may be unaware of how their data is being collected and used.

- *Risks:* If our system was opaque, users might not know that by connecting Gmail they allowed X, or that we do Y with their data.
- *Controls:* Transparency is provided via clear privacy notices and in-app explanations. We provide dashboards or info sections where users can review their linked accounts, consents, and data collected. We have an open privacy policy (and could implement a privacy dashboard for users to see what data we have – beyond export, maybe an in-app view).
- We also ensure that by default, we do privacy-friendly settings (e.g., not opting them into newsletters by default, not using data for secondary purposes unless they opt in).

- *LGPD Mapping:* Art.6 (principle of transparency) and Art.9 (user's right to clear information about processing) [44] [61] – we comply by giving all those details in an accessible format. Also, usercentrics (principle of user control) – we allow opt-outs, etc.

- **Non-compliance:** The system design failing to meet regulatory requirements.

- *Risks:* Not fulfilling a DSR timely, or collecting data without a legal basis, or transferring data abroad without safeguards.
- *Controls:* Everything in this document addresses compliance: we have DSR mechanisms to ensure compliance with rights (so we won't be non-compliant on that). We assign a Data Protection Officer (or equivalent) to oversee these processes. We conduct periodic audits against LGPD (and if relevant, other regimes). We document everything (like ROPA, consents) to be ready if authorities inquire.
- Our incident response covers breach reporting to ensure compliance with LGPD timeline (Art.48) [51].
- *LGPD Mapping:* This category basically maps to the entire law – our measures for security, privacy by design, and legal compliance ensure we meet LGPD obligations (Articles 7, 18, 33-35 for transfers if any, etc.). NIST 800-53 also has a control family (AR – Accountability and Risk) that covers compliance aspects which we adhere to by having policies and a compliance program.

For a quick visual summary, here's an **ASCII table** mapping some threats to controls and references:

```
Threat         | Example Risk                   | Key Controls (mapped to
standards)
-------------- | ------------------------------ |
-----------------------------------------------------------
Spoofing       | Stolen token used by attacker  | Auth tokens tied to sessions,
MFA for critical actions,
               | impersonate user               | mutual TLS internally [6]. ASVS
2.1, NIST IA-2.
Tampering      | Modify data or logs            | TLS everywhere, checksum &
hash-chain for logs [55],
               |                                | integrity checks on DB data.
ASVS 10.2, NIST SC-8, AU-10.
Repudiation    | User denies a transaction      | Immutable audit trails with
user ID and timestamp [54],
               | they approved                  | signed consent records [59]. ASVS
10.3, NIST AU-2, LGPD Art.8§2.
Info Disclosure| Data breach of PII             | AES-256 encryption at rest [7],
strict RLS/ACL,
```

```
              |                                  | secret redaction in logs. ASVS
3.4, NIST SC-28, LGPD Art.46.
DoS           | Overwhelm API or worker          | Rate limiting, autoscaling,
WAF. ASVS 13.3, NIST SC-5.
EoP           | Viewer gains admin rights        | Layered RBAC/ABAC checks, code
reviews for vulns [17],
              | by exploit                       | container isolation. ASVS 1.4,
NIST AC-6.
Linkability   | Correlating user's datasets      | Use pseudonymous IDs, avoid
universal identifiers.
Identifiability| Re-identifying anon data        | Remove personal identifiers
when not needed, aggregate analytics.
Unawareness   | User unaware of data use         | Clear privacy notice & consent
UI, user data export [8].
Non-compliance | Not fulfilling deletion req.    | Automated DSR workflows, DPO
oversight, regular compliance audits.
```

Each of the above controls has been implemented or planned to ensure threats are mitigated to an acceptable level. The mappings to OWASP ASVS and NIST 800-53 provide additional assurance that our security measures follow industry standards. Moreover, by addressing LINDDUN privacy threats, we align our design with privacy principles, which is particularly important for LGPD compliance (e.g., Data minimization to counter linkability/identifiability, and transparency to counter unawareness) [44].

This threat model is revisited whenever major changes occur (new features or integrations) to assess any new threats. It's also cross-checked with OWASP Top 10 and other common frameworks to ensure we didn't miss general issues like injections, which are handled via our secure coding and pipeline (and relate to Tampering/EoP categories above).

DevSecOps Pipeline (CI/CD Security)

We integrate security into our development and deployment pipeline (DevSecOps) to catch issues early and often. The pipeline automates checks for code, dependencies, infrastructure, and built artifacts, aligning with modern supply chain security practices (e.g., SLSA framework) [17] [62]. Key elements of our DevSecOps pipeline include:

- **Version Control Security:** All code resides in a Git repository (e.g., GitHub). We protect branches (require PRs, reviews, and CI passing before merge). Commits are signed where possible (developers use GPG sign, and CI verifies signature to ensure code provenance – satisfying SLSA Level 2 source integrity) [63]. We also enable GitHub's dependency graphs and secret scanning on the repo to catch any committed secrets or known vulnerable dependencies early.

- **Static Application Security Testing (SAST):** On each commit/PR, we run SAST tools:

- **Linters & Code Scans:** We use ESLint with security plugins for our JavaScript/TypeScript code to catch common vulnerabilities (like detecting use of `eval`, insecure URL constructions, etc.).

- **Semgrep or CodeQL:** We run more advanced static analyzers (CodeQL via GitHub Actions, or Semgrep ruleset for OWASP Top 10). These scan for things like SQL injection, XSS, hard-coded secrets, unsafe dependencies, and misuse of security APIs. The pipeline fails if high-severity issues are found.

- **Custom Rules:** We've added custom rules (if needed) to enforce our patterns (e.g., ensure any direct database query includes a tenant filter, to avoid RLS bypass in code).

- **Dependency Management (SBOM and Scanning):** We maintain an updated **Software Bill of Materials (SBOM)** for the app. Using tools like `npm audit` and OWASP Dependency-Check (or Snyk), the pipeline checks for known vulnerabilities in dependencies. Any critical vulnerability in a library fails the build (or at least alerts for triage). We update dependencies regularly as part of sprints.

- We generate a CycloneDX or SPDX SBOM document for each release, listing exact versions of all npm packages (and any other components like Docker base images). This SBOM is archived with the build artifacts. This aligns with SLSA recommendations and helps in case of zero-day dependency issues (we can quickly search SBOMs to see if we're affected).

- **Dynamic Application Security Testing (DAST):** In a staging environment, we deploy the application and run dynamic scans:

- We use OWASP ZAP automated scan on the staging URL to find common web vulnerabilities (XSS, SQLi via exposed forms, etc.). We also test some authenticated pages by scripting a login and scanning those.
- We run a vulnerability scan on the REST API endpoints (using tools or scripts to fuzz inputs).

- Any findings are reviewed; critical ones must be fixed before production deploy.

- **Infrastructure as Code (IaC) Scanning:** Our infrastructure (cloud resources, Terraform or CloudFormation templates, Kubernetes manifests) is scanned for misconfigurations using tools like Checkov or AWS Config rules. This checks for things like open security groups, missing encryption on resources, improper IAM roles, etc. For example, if a developer tries to spin up an S3 bucket without encryption or public access block, the IaC scanner will flag it.

- We treat IaC with same rigor as app code: changes to Terraform go through PR with checks so we don't introduce, say, a world-open database port.

- **Container Security:** If we containerize the app, we scan the Docker images for known vulns (using Trivy or Clair). Base images are kept minimal (from official Node alpine images or similar) and we rebuild regularly to get security patches. CI fails if critical CVEs are present in the image (unless there's a mitigation).

- We also use multi-stage builds to ensure no build tools or secrets remain in final image.

- **Secrets Management in CI:** CI pipelines have access to secrets (API keys for deployment, etc.) which are stored in the CI system's secret store. We ensure these are properly scoped (only accessible by

deployment job, not in test jobs triggered by external PRs). We also employ CI features to mask secrets in logs. As an extra, the pipeline runs a job to verify that no secret-like patterns appear in logs (to ensure none accidentally printed).

- We never embed production credentials in CI configs; instead, use ephemeral short-lived tokens where possible.

- **Build Integrity (SLSA):** To approach SLSA Level 3+, our builds run in isolated runners. We plan to sign our build artifacts: e.g., after building, we sign the compiled Next.js app bundle or Docker image using Sigstore/cosign, generating an attestation of the exact build process (including checksums of source and dependencies) [64] [65]. This artifact signature is checked on deployment (the CD system verifies the signature and integrity before releasing to production). This prevents tampering in the pipeline or an attacker injecting malicious code in transit.

- We also record provenance metadata: e.g., CI job ID, commit hash, who triggered it, etc., stored in an attestation. This ties into supply chain security: if later something suspect is found in prod, we can trace exactly which build it came from and under what conditions.

- **Continuous Monitoring of Dependencies:** We subscribe to vulnerability feeds (GitHub Dependabot alerts, etc.). When a new vuln emerges in a library we use, we get alerts. Our policy is to patch critical vulns within 48 hours (or apply mitigations).

- **Secret Scanning:** As mentioned, we use tools (Git pre-commit hooks and repo scanning) for secrets. In addition, we periodically run truffleHog against the entire repo history to ensure nothing slipped in historically.

- **Quality and Security Gates:** The CI pipeline is configured with quality gates. For example, tests must pass and code coverage should not drop (we have tests including security-critical functionality). Additionally, certain security tests must pass:

- All SAST high issues resolved or marked as false positive with justification.
- All DAST critical findings addressed.
- No new OWASP Top 10 issues introduced (we actually cross-map any tool findings to categories and break build on those).
- Linting and formatting (to reduce risk of sloppy errors).

- If any step fails, the code is not merged/deployed until fixed.

- **Developer Education & PR Review:** Not strictly pipeline, but part of DevSecOps culture: We educate developers on secure coding (OWASP Top 10, how to avoid XSS, etc.). During PR reviews, reviewers use a security checklist (covering things like "Did we do input validation? Could this expose something?") to catch issues that automated tools might miss (like logic issues).

- **Runtime Protection:** Though more of operations, we incorporate some runtime security: e.g., if using Node, perhaps enable security headers via Helmet (checked in staging/DAST). Possibly run an application self-assessment like Node Security Project checks. If we had a RASP (runtime app self-

protection) or CSP reporting, we'd integrate that into monitoring (like any CSP violation triggers an alert).

- **Logging and Alerting in CI:** All CI/CD actions are logged (who deployed what, when). We protect the CI itself: only authorized personnel can modify pipeline configs (to prevent someone from altering pipeline to skip tests or exfiltrate secrets). CI logs are treated as sensitive and stored securely. We have alerts if a pipeline in production deploy stage fails or if any security scan fails (to ensure it's not ignored).

The DevSecOps approach ensures security is not a one-time review but a continuous process. By the time code reaches production, it has passed through multiple automated "gates" that reduce the chance of vulnerabilities. This also aligns with compliance frameworks: - Many OWASP ASVS requirements (V1.14, V14 DevOps sections) are covered by these pipeline measures (e.g., ASVS 14.4.4: use tools to detect vulnerabilities in dependencies – we do that). - NIST 800-53 CM-5 (secure development environment), SA-11 (developer security testing and evaluation) are implemented via our pipeline scans. - Supply chain best practices like SLSA are followed to ensure trust in our software components [66] [67] .

In essence, **security testing and verification are integrated in every step of our SDLC** – from coding (linters/pre-commit) to build (scans) to deployment (signed artifacts) – giving us confidence in the integrity and security of the delivered application.

Incident Response Plan

Despite all preventative measures, we must be prepared to respond swiftly to security incidents. Our Incident Response Plan (IRP) outlines how to identify, contain, eradicate, and recover from incidents, as well as post-incident steps. It also covers breach notification obligations under LGPD and other regimes. Key components:

Incident Severity Classification: - **Severity 1 – Critical Incident:** Major security breach or outage. Examples: confirmed data breach of sensitive personal data (e.g., large-scale exposure of users' financial data or any unauthorized access likely to harm users), production system compromise (attackers control systems), ransomware affecting production, or any incident legally requiring notification (LGPD "relevant risk or damage" breaches) [51] . Also, availability incidents causing total system outage would be Sev1. - **Severity 2 – High:** Significant incident but maybe contained or less immediately damaging. Examples: A breach of non-sensitive data, a serious vulnerability discovered in our system that is at risk of exploitation, a compromise of a non-production environment with some customer data, or a targeted attack that was detected and blocked but warrants investigation. Partial outages affecting many users also fall here. - **Severity 3 – Medium:** Minor security incidents or suspicious activities. Examples: A single user's account compromised (via reused password), a malware infection on an employee workstation with access to systems (but no evidence of data access), or detection of active scanning/attempted attacks that didn't succeed. Also, moderate outages (a single module down) or bug that can cause security issue but has limited impact. - **Severity 4 – Low:** Routine or small-scale issues. Examples: A lost device that was encrypted, a user-report of a potential vulnerability that on triage is low risk, or minor policy violations. Also small outages with minimal user impact.

The severity influences the response urgency, team involvement, and notification: - Sev1: All-hands-on-deck, immediate response 24/7, executives and DPO involved, likely need to notify ANPD and users in short order

[51]. - Sev2: Security team and relevant engineers respond same day, management informed. Possibly involve outside experts if needed. - Sev3: Security team handles during business hours, monitor in case it escalates. - Sev4: Ticketed and handled in normal workflow.

Incident Response Team & Roles: - We have a defined Incident Response Team (IRT) led by our Security Lead (could be CISO or DevSecOps engineer) and involving senior engineers, Ops, and the DPO (for privacy incidents). The team includes: - **Incident Manager:** coordinates the response, keeps everyone on task, serves as primary decision-maker and communication point. For Sev1, likely the CISO or senior security engineer. - **Technical Lead:** expert in the affected area (e.g., if DB compromised, DB lead; if application bug, the principal engineer) to diagnose and fix. - **Communications Lead:** handles external/internal communications – drafting breach notifications, status updates. The DPO or PR person might do this for breach notifications to users and regulators. - **Legal/Compliance:** our DPO or legal counsel ensures we meet breach reporting obligations (like notifying ANPD within a reasonable time, which likely means ASAP, and affected users) [51]. - **Others:** Forensics specialist (maybe external), HR (if insider threat), etc. on standby.

Detection & Alerting: We have monitoring systems in place that trigger alerts on suspicious events: - CloudWatch/SIEM alerts for unusual login patterns, spikes in data access, disabled security controls, etc. - IDS/IPS alerts (if any). - Employees are trained to report any suspicious emails or system behaviors immediately (phishing attempts, strange logs). - If an alert triggers (say multiple failed admin login attempts, or a suddenly high egress of data from DB), the on-call person triages it. We have a 24/7 on-call rotation for Sev1 emergencies.

Containment Steps: Upon confirming an incident: - **Sev1 example (data breach):** Immediately isolate affected systems. E.g., take compromised server off the network (stop container, or change firewall rules). Revoke any tokens or credentials that may be compromised (invalidate all user sessions if needed, rotate API keys). If it's an ongoing data exfiltration, cut off outbound traffic for that service. We might put the site in maintenance mode if necessary to stop further damage. - **For account-specific breach:** Lock the user's account, force password resets for them and potentially all users if credentials leaked. - **Malware/ransomware outbreak:** Disconnect infected hosts, switch to backups or DR site if needed to keep service up. - Containment also means preserving evidence: when isolating, avoid powering off if memory forensics needed (instead snapshot VMs, etc.). We do minimal changes beyond what's needed to stop harm.

Eradication: Once contained, identify root cause and eliminate it: - If it was a software vulnerability (SQL injection, etc.), patch the code immediately (hotfix to production) or apply a virtual patch (WAF rule) until code fix can deploy. - If malware, remove it from systems (rebuild servers from scratch or known good images, run AV scans on endpoints, etc.). - If compromised credentials, change them all (rotate keys, secrets, force user password resets). - Essentially, close the holes the attacker used, double-check similar systems for same issue.

Recovery: Bring systems back to normal operation safely: - Restore data from backups if needed (e.g., if database was corrupted or ransomware encrypted something, we restore clean data). - Ensure systems are patched and secured before reconnecting to network. - Monitor closely when putting systems back online for any sign of attacker persistence. - For example, after a server compromise, we would not simply turn it back on – we'd rebuild a fresh instance, apply patches, then swap traffic to it. The compromised instance is kept offline for forensics. - Verify integrity of data: run checksums on critical data, use audit logs to see if any records were altered and restore them.

Investigation & Forensics: For serious incidents, we perform a forensic investigation: - Preserve system logs, audit trails, and potentially memory dumps/disk images of affected machines [68]. We secure these so attacker can't tamper after the fact. - If needed, hire external forensic experts especially for complex breaches. - Determine timeline: when did attacker get in, what did they access or exfiltrate, and when were they stopped. - Identify root cause clearly (this feeds into remediation). - This phase often happens in parallel with recovery, by dedicating separate personnel to forensics while others fix and bring up services.

Notification and Communication: - **Internal communication:** We follow an incident communication plan. Real-time updates in a designated Slack/Teams channel for the incident, periodic summary to leadership. Ensure no sensitive details accidentally shared outside secure channels. - **User Notification:** Under LGPD, if a breach likely results in risk to users, we must notify ANPD and the impacted data subjects [51]. We have a template ready for breach notice including: - Description of the incident (in plain language), - Types of personal data involved, - Measures we are taking (containment and next steps), - Measures users should take (e.g., reset passwords, watch bank statements), - Our contact info for questions (likely the DPO). We aim to send this notification as soon as we have basic facts, ideally within a few days at most (LGPD says "within reasonable time"; we target 48-72 hours akin to GDPR's 72h guideline for notifying authority). - The ANPD notification (as required by the April 2024 regulation) will include our technical measures, risks, and mitigation steps [50]. We coordinate with legal counsel on this report. - **Other notifications:** If payment data were involved (though we don't store PAN, but say Stripe tokens), we'd inform Stripe too. Also, if incident meets thresholds, possibly law enforcement (for example, large identity theft or if crime is evident). - **Public communication:** For severe incidents, we prepare a public statement or FAQ to publish (transparency to users and to control the narrative, showing we handle responsibly).

Runbooks/Playbooks: We have specific runbooks for common incident types: - *Credential Leak:* e.g., GitHub leaked a secret – steps: revoke secret, check logs for usage, etc. - *Phishing compromise of employee:* steps to reset their creds, check OAuth tokens, etc. - *DDoS Attack:* engage cloud mitigation, block IP ranges, communicate with users about downtime. - *Ransomware detection:* isolate host, move to DR environment, commence restore, etc.

These playbooks provide step-by-step actions and checklist so responders don't miss anything in the heat of the moment [69].

Post-Incident: - Once resolved, we conduct a **Post-Mortem** analysis. This includes: - Timeline of events, - What went well (detections, containment) and what went poorly, - Root cause analysis (the 5 whys, e.g., "SQL injection -> missing validation -> why missing? -> no input validation library in use -> etc."), - Lessons learned and concrete action items to prevent recurrence (e.g., "Implement WAF", "add SAST rule for this pattern", "improve monitoring on X"). - Identify any policy/process failures (like did response take too long? Was communication clear?). - We create an incident report document that is shared internally (and with customers if appropriate, sans sensitive details). - Track the remediation tasks in the backlog and ensure they get done (e.g., if we promise to add an alert or buy a new security tool, it goes into our roadmap). - If the incident triggers an update to policies (maybe update IR plan itself, or improve our password policy, etc.), we do that and train the team accordingly.

Testing the IR Plan: We schedule annual incident response drills (tabletop exercises) where we simulate a scenario (like "customer data breach via vulnerability X") and walk through roles and communications. This keeps everyone prepared and often reveals gaps to fix in our process (like "who had the phone number for ANPD?" or "did we know how to reach that on-call person at 2 AM?").

This IRP ensures that even in worst-case scenarios, we respond in a structured, compliant, and user-centric manner: - Minimizing damage and data loss, - Preserving evidence, - Keeping stakeholders informed, - Meeting legal duties (like LGPD Art.48 notifications) [51], - Ultimately learning and strengthening the system.

Regular reviews of this plan will incorporate changes in the threat landscape or business (for example, if we expand to new regions or store new data types, we'll adapt notification procedures accordingly).

Control Acceptance Criteria (Testable Requirements)

To ensure all the security controls and features described are actually in place and effective, we define acceptance criteria that are testable. Before go-live (and periodically thereafter), we validate each control against these criteria:

1. **Token Broker Encryption:** *Verify that no plaintext OAuth tokens or secrets are stored.* – Inspect the database entries for OAuth tokens: they should be ciphertext (random-looking binary/hex). Attempt to use the token storage without calling KMS (simulate DB read without decryption) and confirm it's unusable (e.g., the ciphertext cannot directly call APIs). Also, review KMS audit logs to ensure that decrypt operations are only happening through the broker service role (and not from other services) [3].

2. **KMS Envelope Implementation:** *Verify KMS is used for sensitive fields.* – In a lower environment, intentionally trigger encryption (connect an account) and check that the app calls `Encrypt` on KMS (via logs or KMS metrics). Ensure that trying to start the app without KMS access fails (as a positive test that we aren't bypassing KMS). Each secret in transit to DB should be encrypted using the configured CMK.

3. **Break-glass Secrecy:** *Ensure break-glass credentials are secure and tested.* – Confirm that the break-glass KMS key or backup secret is stored offline (we can check existence of a sealed envelope or a secure vault entry). Simulate an emergency retrieval in a test environment: only by following documented multi-step process (with proper approvals) can one decrypt a token. Also, verify an audit log entry is generated during such use.

4. **RBAC Role Enforcement:** *Test role-based permissions in the UI and API.* – Use test users: one Owner, one Agent, one Viewer. Try actions for which they are not authorized:

5. Viewer tries to edit data -> should get 403 Forbidden from API.
6. Agent tries to access an admin-only API -> 403.

7. Owner should succeed for own data but if Owner of Org A tries to access Org B's data (with direct ID manipulation), ensure 403 or no data is returned. We will have automated integration tests for these scenarios (mimicking a malicious or mistaken access).

8. **ABAC & RLS:** *Verify row-level isolation.* – Create two organizations with interleaved data in the same table. Run a direct SQL query as the app user (simulate via a backend test) without filtering by org, and confirm the result set only includes the current org's data (RLS is working). Also attempt to change the `app.current_org_id` session variable to a different org as an unprivileged user

(should be denied or not possible). Additionally, run a specific test: one user tries to access another's record by ID (through API), ensure it fails.

9. **Data Encryption at Rest:** *Validate encryption for confidential data.* – This includes:

10. The database storage: confirm that the cloud provider's volume encryption is enabled (check in cloud console).
11. Specific fields: e.g., check that the Prisma field-level encryption is active. You can input a known value into an encrypted field, then directly query the DB to see that it's stored encrypted (and decrypt when accessed via app).
12. S3 buckets: verify bucket default encryption setting is on (and test by uploading a file and attempting to read it outside the bucket context – should be encrypted).

13. Also verify that logs in the observability system do not contain plain sensitive info (we can search log entries for things like 16-digit numbers, email regex, etc., to ensure masking works).

14. **Transport Security:** *Validate TLS enforcement.* – Try accessing the web app over HTTP (if possible) and ensure it redirects or fails. Use a tool like SSL Labs or an internal scan to ensure only strong protocols (TLS1.2+) and ciphers are accepted (no SSLv3, etc.). Also test internal service calls – ensure even internal calls are using TLS or are within a private network (review configs). Check that HSTS header is set on responses (to prevent downgrade).

15. **Audit Log Integrity:** *Detect tampering on audit logs.* – Insert some audit log entries during normal operation. Then simulate a tampering in a test (e.g., manually modify one entry's details or remove an entry in a DB backup) and run the verification tool. It should flag the inconsistency. Also, test that the hashing mechanism works on rollover: e.g., if the application restarts (and thus maybe "first" entry after restart uses last known hash), ensure continuity.

16. Additionally, verify that for each security event (login, data export, etc.), an audit entry is indeed created and contains correct info (actor, action, timestamp). This can be done by triggering events and querying the audit log.

17. Check log retention and access: verify that an unauthorized role cannot delete audit logs (try a DELETE command as app user, it should be denied).

18. **Privacy DSR Functions:** *Test Data Subject Rights endpoints.* – As a test user with some data, request an export via API/UI. Verify the output includes all relevant personal data (cross-check a few items). Request deletion and then verify:

19. The user's data is indeed gone: try to log in (account should be disabled/deleted), query their data via admin (no data).
20. Check that some data that should remain (e.g., an invoice that was in their name might now be anonymized) is anonymized appropriately (e.g., name replaced with "Deleted User").
21. Also test consent withdrawal: give consent, then withdraw and ensure the system stops the relevant processing (e.g., after disconnecting Gmail, no further Gmail fetch jobs run for that user, and their tokens were wiped).

22. We can simulate and ensure these actions are logged as well (audit log shows "User X data export" or "deletion requested").

23. **PCI Scope Control:** *Verify no card data touches our servers.* – Conduct a penetration test simulation for payment: intercept network calls during payment checkout to confirm card number goes directly to Stripe (the network calls from browser show Stripe endpoints for card submission). Also, search our databases/logs for patterns like a 16-digit number that could be a credit card – should find none except perhaps last4 and test cards in non-prod.

    - If using Stripe's library, ensure it's the latest and configured correctly (we can see that only tokens are coming to our backend).
    - Additionally, attempt to use a token from our DB to charge via Stripe API in a test environment without our system – should only succeed with Stripe (the token by itself isn't full card, proving we aren't storing PAN) [2].

24. **DevSecOps Checks:** *Ensure CI/CD pipeline is enforcing security gates.* – For example, push a commit with an intentionally vulnerable dependency or secret and ensure the pipeline fails:

    - E.g., introduce a known vulnerable package version and see that dependency scan fails the build.
    - Commit a fake secret in a branch and see that secret scanning catches it.
    - Ensure that code coverage and lint rules for security are indeed enabled (CI logs should show SAST tool ran and found 0 issues).
    - Try an unauthorized pipeline change: attempt to run a deployment job from a forked repo – ensure secrets are not exposed (this tests CI config security).
    - Check that artifact signing is working: after a build, verify the signature on the artifact (manually or via script in CD).

25. **Monitoring & Alerting:** *Trigger security alerts to test monitoring.* – For instance:

    - Use a test account to attempt multiple failed logins to ensure our alert for brute force triggers an email/page to security.
    - Generate an unusual log pattern (maybe simulate an injection in a test env that the WAF would catch) and see if it's logged/alerted.
    - Verify that an alert reaches the responsible team member (we might create a dummy incident at 3 AM to see if on-call is reached).

26. **Incident Response Drill Outcome:** *Test readiness via drill.* – Conduct a tabletop scenario and assess:

    - Did the team follow the IR plan steps correctly?
    - Was the contact list up to date (e.g., ANPD contact template ready)?
    - Document any issues found (this becomes an acceptance criteria to fix: e.g., "Time to identify data affected was too long; need better query for user data locations").

27. **Backlog and WBS Realization:** *Check that all planned security tasks are tracked.* – Go through the Security Backlog (next section) and ensure each item is either completed or scheduled. This is more of a project acceptance: no critical control should be left unimplemented. For example, if "implement

log redaction middleware" is on backlog, ensure the code is merged and working (maybe by searching logs for known sensitive patterns to confirm they are redacted).

Each of these criteria can be tested regularly (some can be part of continuous automated tests, others during periodic audits or drills). We will formally verify them before launch (security acceptance testing phase) and include many in ongoing regression tests.

By meeting these acceptance criteria, we ensure the security architecture and controls are not just theoretical – they are functioning as intended, thereby significantly reducing risk.

Implementation Backlog (Security Work Breakdown)

To operationalize this security strategy, we maintain a backlog of tasks. Below is a breakdown of major jobs, each broken into runs and concrete steps, that we will execute. This is ready to import into a project management or Work Breakdown Structure (WBS) tool:

- **Job 1: Establish Secure Architecture Baseline**
- *Run 1.1: Network & Environment Hardening*
    - Step 1.1.1: Set up VPC with public/private subnets; enforce that DB/Redis are in private subnets accessible only by API servers.
    - Step 1.1.2: Configure security groups/firewall rules (only port 443 to web, DB port open only to API, etc.).
    - Step 1.1.3: Enable TLS certificates and HSTS on frontend.
    - Step 1.1.4: Deploy WAF with baseline OWASP Top 10 rules in front of the application.
- *Run 1.2: Deploy Observability Stack Securely*
    - Step 1.2.1: Set up logging infrastructure (ELK or cloud monitor) with access controls (readonly for devs, full for ops).
    - Step 1.2.2: Implement log redaction middleware in app (regex scrub PII, secrets) and test by generating sample logs [70] .
    - Step 1.2.3: Configure alert rules (e.g., high error rate, security events) in monitoring tool.
- **Job 2: Implement Data Classification Protections**
- *Run 2.1: Data Inventory & Tagging*
    - Step 2.1.1: Document all data fields and assign classification (Public/Internal/Confidential/Highly Sensitive).
    - Step 2.1.2: Mark fields in code or DB schema comments for awareness (e.g., `/// @encrypted` for Prisma on sensitive fields).
    - Step 2.1.3: Review logging and ensure no Class 3/4 data is logged (adjust code where needed).
- *Run 2.2: Encryption & Masking Enforcement*
    - Step 2.2.1: Enable encryption at rest: turn on RDS storage encryption, verify it's active.

- Step 2.2.2: Integrate `prisma-field-encryption` extension or custom crypto for fields marked ///@encrypted (e.g., user.email if decided, or any sensitive notes) [71] .
- Step 2.2.3: Implement masking in UI for sensitive fields (e.g., show last4 of account numbers).
- Step 2.2.4: Create unit tests to ensure when retrieving masked fields via API, they follow masking format (and full data isn't sent to unauthorized roles).

• **Job 3: Token & Secret Management System**

• *Run 3.1: KMS Integration*
- Step 3.1.1: Provision KMS keys for each secret category (e.g., `prod/tokenKey`, `prod/stripeKey`). Document key IDs.
- Step 3.1.2: Implement KMS wrapper module in code (functions `encrypt(secret)` and `decrypt(cipher)` using AWS SDK) [3] . Include context (e.g., user id) if useful.
- Step 3.1.3: Modify OAuth callback handlers to use `encrypt()` before saving tokens.
- Step 3.1.4: Modify any code that reads tokens to go through `decrypt()` (or request token broker service).
- Step 3.1.5: Test end-to-end: connect account, ensure ciphertext in DB and successful API call with decrypted token.

• *Run 3.2: Token Broker Service (if separate)*
- Step 3.2.1: Set up a minimal microservice or module designated as Token Broker with its own credentials.
- Step 3.2.2: Give Token Broker IAM role decrypt permissions on KMS keys; remove those permissions from main API role.
- Step 3.2.3: Adjust API server to call broker for external API interactions (or for decrypt operations).
- Step 3.2.4: Implement caching of decrypt results if needed (in-memory for a few minutes) to reduce KMS calls, but ensure auto-clear on expiry.
- Step 3.2.5: Pen-test attempt: call external API without going through broker to confirm it's not possible (lack of token access).

• *Run 3.3: Secrets Vault & Rotation*

- Step 3.3.1: Store environment secrets (DB passwords, Stripe API keys) in a secure vault or secrets manager. Update deployment to fetch from vault at runtime (instead of plaintext .env).
- Step 3.3.2: Implement Zod validation for env vars on startup (reference code snippet) to avoid missing secrets issues.
- Step 3.3.3: Establish a rotation schedule for keys: e.g., script to rotate Stripe webhook secret every 6 months and update in vault.
- Step 3.3.4: Document break-glass access: who holds backup credentials, process to retrieve (perhaps write in runbook and get sign-off from security lead). Do a drill for break-glass (maybe in test environment).

• **Job 4: Access Control & Multi-Tenancy**

• *Run 4.1: RBAC Implementation*
- Step 4.1.1: Define role constants (Owner/Agent/Viewer) in code and assign on user creation/invite flows.

    - Step 4.1.2: Implement middleware to check role permissions on protected API routes. Use a centralized policy (e.g., a mapping of route->allowed roles).
    - Step 4.1.3: Implement frontend logic to hide/disable UI elements not permitted by role (for usability).
    - Step 4.1.4: Write tests for each role trying each critical action (should allow or deny appropriately).
- *Run 4.2: ABAC Rules & Context*
    - Step 4.2.1: Identify scenarios for ABAC (e.g., restricting approvals to certain users). Implement attribute checks in those flows (like `if request.user.id != record.owner_id: deny`).
    - Step 4.2.2: If using context attributes (like time or IP), integrate check or use a library that supports ABAC policies. Possibly create a policy file that says e.g., "Agents can only see transactions if assigned_to == user".
    - Step 4.2.3: Test ABAC: simulate an Agent not assigned to an entity trying to access it.

- *Run 4.3: Postgres RLS Setup*

    - Step 4.3.1: Add `org_id` column to all relevant tables (if not already). Populate with correct org references.
    - Step 4.3.2: Enable RLS and create policies for each table [72] [26]. Use `current_setting('app.current_org_id')` approach for flexibility.
    - Step 4.3.3: Modify DB connection code to `SET app.current_org_id = <user's org>` on each request connection (and reset after). Ensure pooling doesn't carry over wrong org – maybe use `pg` middleware or always set on query.
    - Step 4.3.4: Test direct SQL queries via the app to ensure isolation (as earlier acceptance criteria described).
    - Step 4.3.5: Ensure admin or migration roles can bypass RLS when needed (for data migration scripts, etc.), but that app role cannot. Possibly test that a plain SELECT * by app user only returns its tenant data.

- **Job 5: Privacy Compliance Features**

- *Run 5.1: Consent Management*
    - Step 5.1.1: Create a Consent table or log to record user consents (fields: user, consent_type, given_at, revoked_at).
    - Step 5.1.2: Update UI flows: after OAuth or enabling an integration, explicitly log consent in DB.
    - Step 5.1.3: Provide UI for user to review and revoke consents (a settings page listing all active consents with toggle to revoke).
    - Step 5.1.4: Ensure revoking triggers appropriate backend logic (e.g., disabling scheduled data fetch, deleting tokens) and logs the event.
- *Run 5.2: DSR Endpoints Implementation*
    - Step 5.2.1: Implement `GET /api/dsr/export` for authenticated user – this aggregates data from all tables related to the user. Possibly generate a zip with JSON and PDFs.
    - Step 5.2.2: Implement `POST /api/dsr/delete` – this will initiate account deletion. Mark user as pending deletion (to avoid re-use during process), then asynchronously remove or anonymize data.
    - Step 5.2.3: Implement admin tool or automated step to propagate deletion to backups/archives (or flag so we don't restore deleted data inadvertently).

    - Step 5.2.4: Test these endpoints thoroughly (the acceptance criteria tests).

• *Run 5.3: ROPA & Documentation*

    - Step 5.3.1: Create internal wiki or doc for ROPA. Fill in processing activities, purposes, etc. based on earlier data inventory.
    - Step 5.3.2: Prepare an Incident response breach notification template (to ANPD and users) now, in consultation with legal, to have it ready.
    - Step 5.3.3: Ensure a communication channel with ANPD is set up (like who would send, how, contact info prepared).
    - Step 5.3.4: Appoint a person (DPO) responsible and ensure their contact is in privacy policy and in ANPD's official form.

• **Job 6: PCI Compliance & Payments**

• *Run 6.1: Payment Integration Review*
    - Step 6.1.1: Double-check Stripe integration settings: use Elements or Checkout such that our domain never sees card data.
    - Step 6.1.2: Verify that webhooks from Stripe contain no PAN (should only have token IDs and last4). Write code to ignore any sensitive fields if they ever appear.
    - Step 6.1.3: Fill out PCI SAQ A (if applicable) to validate compliance formally. This is a task for compliance officer: document how we meet each relevant question.
    - Step 6.1.4: If required, implement quarterly vulnerability scans (could use a service) to meet PCI requirement 11.2 – though our scope is limited, it's good practice.

• *Run 6.2: Payment Data Handling Controls*

    - Step 6.2.1: Ensure database columns storing last4 are marked as non-sensitive (but still treat as personal data under privacy).
    - Step 6.2.2: Implement alert if any system logs a full 16-digit number (regex trigger) – as an assurance no card data inadvertently in logs.
    - Step 6.2.3: Add unit test or monitoring that if a PAN pattern appears in DB (which should not), it flags. Possibly attempt to insert a fake PAN in dev to confirm our detection logic works (if we add such logic).

• **Job 7: Audit Logging & Monitoring**

• *Run 7.1: Audit Log Development*
    - Step 7.1.1: Create the `audit_log` table as per design.
    - Step 7.1.2: Implement the trigger or application code to compute `prev_hash` and `hash` on insert [55]. Use a stable library for SHA-256.
    - Step 7.1.3: Go through all code paths to insert audit logs at key events (login, logout, data access, data change, permission changes, etc.). Ensure we capture necessary info in `details`.
    - Step 7.1.4: Write a verification script that scans log chain integrity. Possibly schedule it to run daily and alert if any discrepancy.

• *Run 7.2: Log Security*

  ◦ Step 7.2.1: Restrict DB user permissions: app user can INSERT into audit_log but not update/delete. Only very limited admin role can archive old logs if needed.
  ◦ Step 7.2.2: Setup an archival process: e.g., every 90 days, export audit logs to secure storage (with their hashes) for long-term retention. Test restoring archive to verify it's intact.
  ◦ Step 7.2.3: Protect the logs in transit to monitoring: ensure if we send audit logs to SIEM, it's via secure channel and only accessible to security team.
  ◦ Step 7.2.4: Document procedure to provide logs if required (for audit or legal) – essentially queries with filters by user or date.
  ◦ Step 7.2.5: Non-repudiation test: demonstrate an audit log entry for a key action (like a financial transfer) and the chain proving its integrity for an external auditor.

• **Job 8: Threat Model & Testing**

• *Run 8.1: Penetration Testing*
  ◦ Step 8.1.1: Engage a third-party to conduct a pen-test on the application before launch. Share threat model so they can focus on critical areas.
  ◦ Step 8.1.2: Internal red-team exercise: attempt common attacks (SQL injection via API, XSS via forms, CSRF on state-changing actions) to validate defenses (these might already be covered by DAST, but manual attempts often find logic issues).
  ◦ Step 8.1.3: Test privacy threats manually: e.g., try to enumerate user emails via signup or password reset form (should not reveal which emails exist).
  ◦ Step 8.1.4: Resolve any issues found and re-test.

• *Run 8.2: Verify Mappings to Standards*

  ◦ Step 8.2.1: Create an OWASP ASVS checklist and map our implementation. Perform a self-assessment to ensure Level 2 (or 3 where applicable) controls are all addressed.
  ◦ Step 8.2.2: Do the same for a subset of NIST 800-53 controls we committed to (probably moderate baseline tailored). This might just be an internal audit item: ensure we have documentation or operation evidence for each (e.g., AC-2 user provisioning process documented, etc.).
  ◦ Step 8.2.3: Align controls to LGPD articles: prepare a document listing how each relevant LGPD requirement is met (could reuse content from this guide). This helps in case of inquiry or just to double-check we covered all bases (like an internal compliance audit).

• **Job 9: DevSecOps & CI/CD Hardening**

• *Run 9.1: CI Pipeline Security*
  ◦ Step 9.1.1: Add SAST job to CI (e.g., integrate CodeQL analysis GitHub Action, or run semgrep in pipeline). Ensure it breaks build on high findings.
  ◦ Step 9.1.2: Add dependency scanning job (npm audit or Snyk). Ensure pipeline fails on critical vulns unless exception documented.
  ◦ Step 9.1.3: Add IaC scanning job (if using Terraform, run Checkov; if using Docker, run Trivy image scan).
  ◦ Step 9.1.4: Implement artifact signing: set up cosign to sign Docker images after build and verify signature before deploy in CD.

    - Step 9.1.5: Restrict pipeline triggers: disallow fork PRs from running deploy jobs with secrets, etc. Turn on required status checks so unapproved code can’t be merged.

- *Run 9.2: Continuous Monitoring Integration*

    - Step 9.2.1: Integrate code coverage and tests into pipeline to ensure quality (security relies on good testing too).
    - Step 9.2.2: Setup a nightly run of dependency updates (Dependabot) and security scans to catch new issues outside of commit activity.
    - Step 9.2.3: Include a job that updates the SBOM and pushes it to an artifact repository or security dashboard for visibility.
    - Step 9.2.4: Implement notifications from CI: e.g., if a security scan fails, create an issue or alert the security channel. No silent failures.

- **Job 10: Incident Response Readiness**

- *Run 10.1: IR Plan & Tools*
    - Step 10.1.1: Finalize the written Incident Response Plan (could be this content distilled). Include on-call contacts, phone tree, communication templates.
    - Step 10.1.2: Set up an incident communication channel (e.g., a dedicated Slack war-room setup procedure) and an incident tracking system (Jira or a Google form template to record incident details).
    - Step 10.1.3: Prepare "jump bag" tools: ensure we have accounts with forensic tools, an AWS account partition for forensic analysis of snapshots, etc., ready.
    - Step 10.1.4: Establish relationships with external incident response firm (if desired) ahead of time (so NDA and contract are in place if we need emergency help).
- *Run 10.2: Conduct Training & Drill*
    - Step 10.2.1: Do an initial table-top exercise with the team. Scenario e.g., "AWS key leaked on GitHub, attacker spun up instances and accessed DB" – walk through IR steps.
    - Step 10.2.2: Document outcomes of drill: what was done well/needs improvement. Update IR plan accordingly (e.g., we realized we didn’t know who contacts ANPD; now assign that).
    - Step 10.2.3: Communicate to whole organization a summary of the IR plan and where to find it (so even non-security staff know how to report incidents and that we have a plan).
    - Step 10.2.4: Schedule periodic drills (at least annually, or whenever major changes in system happen).
    - Step 10.2.5: Ensure breach notification templates are filled with placeholder and readily available. If multi-lingual needed (Portuguese for Brazilian users, etc.), have translations pre-done to not lose time during an incident.

Each of these tasks will be assigned, tracked, and tested upon completion. By following this WBS, we move systematically from design to implementation and verification of our security strategy. This ensures no critical control is forgotten and that the end product meets the high security and privacy standards outlined.

Throughout implementation, the tasks can be adjusted as needed, but the overarching goal remains: **no launch until all acceptance criteria are met and all high-risk backlog items are done**.

Reference Schemas and Code Snippets

Below are some schema definitions and code snippets illustrating key parts of the implementation for clarity:

1. Environment Variables Validation (using Zod):
We use a Zod schema to validate that all required environment variables are present and correctly formatted at startup. This prevents the app from running with misconfigured or missing secrets (a common source of security issues).

```
import { z } from "zod";

// Define expected env vars and basic validations (e.g., URLs, keys).
const EnvSchema = z.object({
  NODE_ENV: z.enum(["development", "test", "production"]),
  DATABASE_URL: z.string().url(),
  // For example, an API key that should be 32 hex chars:
  STRIPE_SECRET_KEY: z.string().regex(/^sk_live_[0-9a-zA-Z]{24}$/),
  KMS_KEY_ID: z.string().nonempty(),
  // ... add others as needed
});

// Parse and validate process.env
const parsedEnv = EnvSchema.safeParse(process.env);
if (!parsedEnv.success) {
  console.error("  Invalid environment configuration:",
parsedEnv.error.format());
  process.exit(1);
}
export const ENV = parsedEnv.data; // Typed env object for use in app
```

This will throw an error (and stop the app) if any required secret is missing or malformed, ensuring we never run with default or blank secrets.

2. KMS Encryption/Decryption Wrapper:
Using AWS SDK v3 for Node.js. This wraps encrypt/decrypt calls to AWS KMS for ease of use. We include an encryption context (like `{"purpose": "oauth_token"}`) to bind usage, though it's optional.

```
import { KMSClient, EncryptCommand, DecryptCommand } from "@aws-sdk/client-kms";

const kmsClient = new KMSClient({ region: ENV.AWS_REGION });

const keyId = ENV.KMS_KEY_ID; // e.g., 'arn:aws:kms:...:key/xxxxxxxx'

export async function encryptSecret(plaintext: string): Promise<string> {
```

```
  const command = new EncryptCommand({
    KeyId: keyId,
    Plaintext: Buffer.from(plaintext, "utf-8"),
    EncryptionContext: { service: "FintechDash", purpose: "oauth_token" }
  });
  const resp = await kmsClient.send(command);
  // CiphertextBlob is a Uint8Array; encode to base64 for storage
  return Buffer.from(resp.CiphertextBlob as Uint8Array).toString("base64");
}

export async function decryptSecret(ciphertextBase64: string): Promise<string> {
  const command = new DecryptCommand({
    CiphertextBlob: Buffer.from(ciphertextBase64, "base64"),
    EncryptionContext: { service: "FintechDash", purpose: "oauth_token" }
  });
  const resp = await kmsClient.send(command);
  return Buffer.from(resp.Plaintext as Uint8Array).toString("utf-8");
}
```

Usage in code: when receiving a new OAuth token, call `encryptSecret(token)` and store result. When needing to use a token, call `decryptSecret()` on the stored value. The encryption context ensures that even if a ciphertext is stolen, the decrypt requires the same context (prevent misuse). Only our AWS IAM role with decrypt permission on that key can call these.

3. Prisma Schema with Field Encryption Annotation:
Using the prisma-field-encryption extension, we annotate sensitive fields with `/// @encrypted`. Example:

```
model User {
  id        Int     @id @default(autoincrement())
  email     String  @unique
  password  String  /// @encrypted  (hashed password could also be just hashed,
but example)
  name      String? /// @encrypted
  role      String  @default("Viewer")
  orgId     Int
}

model BankAccount {
  id          Int    @id @default(autoincrement())
  orgId       Int
  accountNumber String /// @encrypted
  bankName    String
  balance     Float
  // Only last4 might be stored in clear, full accountNumber encrypted if
```

```
needed.
}
```

With the Prisma client extension set up as in the README [71], any reads/writes of those fields will be transparently encrypted/decrypted with the provided key. We ensure `PRISMA_FIELD_ENCRYPTION_KEY` is set from a secure source (could derive it from our KMS or use a secure env var).

4. Audit Log Schema and Trigger (PostgreSQL):

```
-- Assuming a UUID generation extension and a hashing extension for simplicity
CREATE EXTENSION IF NOT EXISTS "uuid-ossp";
CREATE EXTENSION IF NOT EXISTS pgcrypto;

CREATE TABLE audit_log (
  id        BIGSERIAL PRIMARY KEY,
  timestamp TIMESTAMPTZ NOT NULL DEFAULT now(),
  actor_type TEXT NOT NULL,
  actor_id   TEXT NOT NULL,
  action     TEXT NOT NULL,
  entity_type TEXT,
  entity_id   TEXT,
  details    JSONB,
  prev_hash  TEXT,
  curr_hash  TEXT
);

CREATE OR REPLACE FUNCTION audit_log_hash_trigger()
RETURNS TRIGGER AS $$
DECLARE
  last_hash TEXT;
  data TEXT;
BEGIN
  -- Get the latest hash (curr_hash of last entry)
  SELECT curr_hash INTO last_hash FROM audit_log ORDER BY id DESC LIMIT 1;
  IF last_hash IS NULL THEN
    last_hash := 'GENESIS';
  END IF;
  -- Concatenate critical fields (could choose a specific ordering or JSON
serialization)
  data := COALESCE(last_hash, '') || NEW.actor_type || NEW.actor_id ||
NEW.action
          || COALESCE(NEW.entity_type, '') || COALESCE(NEW.entity_id, '')
          || COALESCE(cast(NEW.details as text), '');
  NEW.prev_hash := last_hash;
  NEW.curr_hash := encode(digest(data, 'sha256'), 'hex');
  RETURN NEW;
```

```
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER tg_audit_hash
BEFORE INSERT ON audit_log
FOR EACH ROW EXECUTE FUNCTION audit_log_hash_trigger();
```

This trigger computes `curr_hash` from previous hash and the new row. We use `digest` from pgcrypto to SHA-256 hash. The `prev_hash` is stored for reference too (not strictly needed, but helpful for debugging chain). We might include `timestamp` in the data string too for uniqueness.

5. Log Redaction Middleware (Express.js example):
We use Winston for logging; we can add a format that redacts sensitive patterns:

```
const { createLogger, format, transports } = require('winston');

const sensitivePatterns = [
  /\b\d{13,16}\b/g,         // 13-16 digit numbers (potential credit card, will
redact fully)
  /\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}\b/g,  // email addresses
  /Bearer\s+[A-Za-z0-9\-_.]+/g // JWT tokens in Authorization header
];

const redactFormat = format((info) => {
  let msg = info.message;
  for (const pat of sensitivePatterns) {
    msg = msg.replace(pat, '[REDACTED]');
  }
  info.message = msg;
  return info;
});

const logger = createLogger({
  level: 'info',
  format: format.combine(
    redactFormat(),
    format.json()
  ),
  transports: [
    new transports.Console(),
    new transports.File({ filename: 'app.log' })
  ]
});

// Usage: logger.info(`User login: email=${user.email}, token=${token}`);
```

With this, if a developer accidentally logs an email or a token, the output will be `[REDACTED]`. We included common patterns, and we can refine as needed (maybe partial masking for credit card if we want to keep last4). Note: This runs on each log message format.

6. Example Express Route with RBAC & ABAC enforcement:

```
// Express middleware for RBAC
function requireRole(allowedRoles: string[]) {
  return (req, res, next) => {
    const userRole = req.user.role;
    if (!allowedRoles.includes(userRole)) {
      return res.status(403).json({ error: "Forbidden" });
    }
    next();
  };
}

// Example: only Owner and Agent can create a transaction
app.post('/api/transactions', requireRole(['Owner','Agent']), async (req, res)
=> {
  // ABAC: ensure if agent, they can only add within certain limits or contexts:
  if (req.user.role === 'Agent') {
    // e.g., check if Agent has permission in this category
    if (!req.user.canCreateTransaction) {
      return res.status(403).json({ error: "Forbidden" });
    }
  }
  const newTx = await prisma.transaction.create({
    data: { ...req.body, orgId: req.user.orgId }
  });
  res.json(newTx);
});
```

This shows role check first, then an additional attribute check for agents. The orgId is automatically set from user (to enforce tenancy). The RLS will also kick in if someone tries to bypass via direct query.

7. Partial STRIDE Threats Table (for documentation clarity):

We can include an ASCII table or simple mapping inline (as done in the Threat Model section) to communicate threats to developers and testers, ensuring they know what to verify. For example, as included above:

```
STRIDE  | Example           | Controls
------- | ----------------- | ----------------------------------------
Spoofing: fake identity -> | MFA, session tokens, service authentication
```

```
Tampering: alter data/logs -> | TLS, hashing logs [55], input validation
...
```

(This was already in section, referencing it here as something we'd keep in our docs.)

8. Signing Commits (Git config):

While not code in app, we ensure all engineers set `commit.gpgsign=true` and provide them instructions to generate GPG keys. This detail is captured in our contributing guide:

```
[commit]
  gpgSign = true
[user]
  signingkey = ABCD1234  (your GPG key ID)
```

Then our Git server (GitHub) is set to "Verify signed commits" for protected branches.

These snippets and schemas support the implementation of the controls discussed. They should be reviewed and adapted to the actual codebase, but serve as a concrete starting point for developers to follow the security design.

---

[1] [2] [52] [53] PCI Security Compliance (Tokenization with Stripe Elements?) | WordPress.org
https://wordpress.org/support/topic/pci-security-compliance-tokenization-with-stripe-elements-4/

[3] [18] [19] [20] [21] Secrets Management - OWASP Cheat Sheet Series
https://cheatsheetseries.owasp.org/cheatsheets/Secrets_Management_Cheat_Sheet.html

[4] [5] [26] [27] [28] [29] [72] Row Level Security for Tenants in Postgres | Crunchy Data Blog
https://www.crunchydata.com/blog/row-level-security-for-tenants-in-postgres

[6] [46] [48] [49] [50] [51] [68] Security Requirements and Breach Notification | Brazil | Global Data and Cyber Handbook | Baker McKenzie Resource Hub
https://resourcehub.bakermckenzie.com/en/resources/global-data-and-cyber-handbook/latin-america/brazil/topics/security-requirements-and-breach-notification

[7] [PDF] Application Security Verification Standard 4.0.3 - GitHub
https://raw.githubusercontent.com/OWASP/ASVS/v4.0.3/4.0/OWASP%20Application%20Security%20Verification%20Standard%204.0.3-en.pdf

[8] [9] [10] [11] [30] [31] [32] [33] [34] [35] [36] [37] [38] [39] [40] [41] [42] [43] [44] [45] [59] [60] [61] Brazilian General Data Protection Law (LGPD, English translation)
https://iapp.org/resources/article/brazilian-data-protection-law-lgpd-english-translation/

[12] [13] [71] 4 Main Data Classification Categories With Practical Examples - Numerous.ai
https://numerous.ai/blog/data-classification-categories

14 AWS KMS cryptography essentials - AWS Key Management Service
https://docs.aws.amazon.com/kms/latest/developerguide/kms-cryptography.html

15 AWS KMS for Envelope Encryption
https://wolfman.dev/posts/aws-kms-for-envelope-encryption/

16 Standard : OWASP Cheat Sheets : Secrets Management Cheat Sheet
https://www.opencre.org/node/standard/OWASP%20Cheat%20Sheets/section/Secrets%20Management%20Cheat%20Sheet

17 62 63 65 66 67 7 Ways to Use the SLSA Framework to Secure the SDLC | Jit
https://www.jit.io/resources/security-standards/7-ways-to-use-the-slsa-framework-to-secure-the-sdlc

22 23 24 25 RBAC vs ABAC: main differences and which one you should use
https://www.osohq.com/learn/rbac-vs-abac

47 Brazil's LGPD - SecuPi
https://secupi.com/solution/lgpd/

54 55 56 57 70 Audit logs security: cryptographically signed tamper-proof logs | Cossack Labs
https://www.cossacklabs.com/blog/audit-logs-security/

58 A3:2017-Sensitive Data Exposure - OWASP Foundation
https://owasp.org/www-project-top-ten/2017/A3_2017-Sensitive_Data_Exposure

64 Pipeline Integrity and Security in DevSecOps - GitGuardian Blog
https://blog.gitguardian.com/pipeline-integrity-and-security-in-devsecops/

69 Plan: Your cyber incident response processes - NCSC.GOV.UK
https://www.ncsc.gov.uk/collection/incident-management/cyber-incident-response-processes